This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

Reverse-Engineering Optimization Techniques of High-Level Synthesis: Practical Insights into Accelerating Applications with AMD-Xilinx Vitis

Jorge Koronis Oscar Garnica J. Ignacio Hidalgo Juan Lanchares Dávila

***Abstract*—**

Modern AI applications contain computationally expensive sections. Accelerator cards and tools like AMD Vitis HLS leverage high-level synthesis and hardware (HW) optimizations to create custom HW designs to accelerate them. Nevertheless, the learning curve is steep, even for those with previous knowledge of HW design, due to the complexity of the optimization techniques and limited information on their interactions and HW effects. This paper quantitatively analyzes the interactions of optimization techniques after reverse engineering Vitis' optimization directives, both in isolation and in pairs. Over 150 experiments were conducted to investigate three distinct goals: assessing pragma behavior and the rules governing pragma application and optimizations, modeling Vitis HLS latency estimates, and evaluating the impact of optimizations on design space exploration, specifically area and latency. These experiments involve different combinations and placements of optimizations in the loop and function hierarchy of the test bench. Our findings offer guidance on using Vitis pragmas and identify promising configurations for optimizing latency and area.

***Index Terms*—HLS, Xilinx, Vitis, Alveo, acceleration, optimization, tutorial**

I. Introduction

EXECUTING certain algorithms, in disciplines such as artificial intelligence (e.g., neural networks); finance (e.g., Monte Carlo simulations); or physics (e.g., weather-forecasting algorithms), on a CPU is computationally expensive. Applications in other fields of study, such as health systems (magnetic-resonance imaging) or defense systems (anti-ballistic missiles) need a real-time response, and significant CPU resources. Implementing the compute-intensive tasks of these software (SW) applications with purpose-built hardware (HW) circuits running on FPGA-based accelerators improves performance and saves manufacturing costs compared to application-specific integrated circuit (ASIC)-based accelerators. This can be achieved by: 1) Describing HW with a HW description language (HDL) such as VHDL or Verilog. 2) Describing the behavior of tasks through code in high-level programming languages, such as C/C++, and using high--level synthesis (HLS) tools to get register transfer level (RTL) descriptions of the most computationally expensive sections in the high-level code.

Vitis HLS is an AMD-Xilinx-developed tool that helps implement high-level C/C++ computationally expensive algorithms in HW. It synthesizes C/C++ function specifications (kernels) into RTL code, implemented on an FPGA through the back-end design flow. Kernels are synthesizable C/C++ top-level functions defined within the body of the clause `extern "C{ }"` and work as accelerating engines.

To implement optimal RTL code, C/C++ code must follow the recommended coding style for synthesis [1]. Moreover, Vitis offers performance optimizations that enhance latency and initiation interval (II) in the synthesized HW. The most important are pipelining, flattening, and unrolling [2]. Both Vitis HLS settings and the developer guide these optimizations. While Vitis HLS settings configure automatic optimizations, developers can use explicit synthesis directives [2], [3] to arrange specific optimizations:

1) Commands in scripting files using the Vitis-supported tool command language (Tcl). They help explore different solutions to a problem with Vitis HLS.

2) HLS pragmas embedded in C/C++ kernel source code guide RTL code optimizations. They are preferred for developing accelerated applications with Vitis and should replace Tcl directives in the final project.

Although AMD-Xilinx user guides explain optimization results in small-scale use cases, a comprehensive case study on the effects of the optimizations is lacking, leaving many details unclear. This study analyzes optimization techniques and their interplay, focusing on the behavior of Vitis HLS algorithms and their impact on area and latency. The area analysis covers flip-flops (FFs), lookup tables (LUTs), and digital signal processing units (DSPs).

Concerning Vitis HLS algorithms, the goals are to reveal the default optimizations, the precedence among optimizations, developer-guided and automatic, and compatibility when used together.

Providing in-depth knowledge on implementing the HLS methodology and optimizing with Vitis HLS will enhance process productivity and shorten the design cycle. Therefore, instead of a conceptual study, our goal is to provide practical guidelines to help designers, with limited expertise in HLS, master optimization techniques using this tool.

Regarding the optimization effects on area and latency, the objectives are to uncover the outcome of single optimiza-

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

tions in isolation, combined impact of optimizations, and rationale behind the transformations' effects.

The paper analyzes the synthesis results of a C/C++ loop nest. This structure enables evaluating performance optimizations and comparing their effects at different loop levels. Moreover, the straightforward computation in the innermost loop clarifies the differences in area and latency among optimization configurations, a.k.a. setups.

The remainder of this paper is structured as follows. Section II analyzes the state-of-the-art in HLS-optimizations interplay, Section III presents the architecture of AMD-Xilinx Alveo accelerator cards, Section IV explains Xilinx design flows and kernel execution modes, Section V describes Vitis-HLS performance optimizations, Section VI describes a kernel, Section VII describes pragmas' behavior in isolation and in pairs, Section VIII models Vitis-HLS loop and function latencies, Section IX evaluates HLS-optimization impact, and Section X concludes.

II. State of the Art

Besides Vitis by AMD-Xilinx, other HLS tools [4] include Catapult by Siemens, Symphony C by Synopsys, HDL Coder by MathWorks, or Stratus HLS by Cadence Design Systems. They accept code in various high-level programming languages, respectively, C++ and SystemC; C/C++; MATLAB; and C++ and System C. Although they use similar optimization techniques, these techniques and their results depend on the target device, which is an AMD-Xilinx-made Alveo card in our case.

[5] addressed the interdependency between two HLS optimizations: multidimensional memory partition and loop-unrolling. Its authors focus on reducing the area, in designs with this memory partition‘. This partition type can cause bank switching, which leads to area overhead. Unrolling the loops that access memory can eliminate bank switching and optimize the area.

Another study, [6], developed a tool to help designers find high-performance synthesis directive configurations within minutes under given resource constraints. The authors researched the interaction effects of different directives, performing performance and area trade-off analysis.

The authors of [7] applied Bayesian optimization to prune the HLS-transformations design space retaining those that balance low latency and efficient use of HW. They built a framework for iterative search among optimization setups, bypassing time-consuming simulation, synthesis, and place-and-route stages. The framework takes high-level code as input and outputs RTL implementations.

Another paper, [8], proposed a framework for DSE of optimization configurations in HLS. Typically, loops with variable bounds are challenging to optimize with pipelining or unrolling. These pragmas lead to variable HW underuse, making performance estimation hard and estimates of loop-latency ranges relatively broad. The authors propose source code transformations to address HW underutilization and present a model to estimate area and latency.

Finally, in [9], the authors explored several HLS solutions to accelerate the K-NN machine learning algorithm with Vivado HLS. They analyzed the interaction effects of the most relevant HLS optimizations; they concluded that only a subset can offer the same acceleration as using all of them. A different study, [10], optimized HLS directives through sequential model-based optimization (SMBO) methods, such as Bayesian optimization. SMBO helps optimize computationally expensive functions. Therefore, this strategy is effective due to the complex interaction effects on area and latency among directives, the high cost of evaluating specific optimization configurations, and the large design space exploration (DSE).

III. Alveo Cards Description

Alveo™ accelerator cards help achieve higher performance, accelerating CPU-intensive algorithms [11]. They are designed for host connectivity and heterogeneous systems. Figure 1 presents the main modules each Alveo card includes [12]: an FPGA; off-chip high-bandwidth memory (HBM) banks (global memory); and a high-bandwidth PCIe Gen3x16 link connecting to the host.

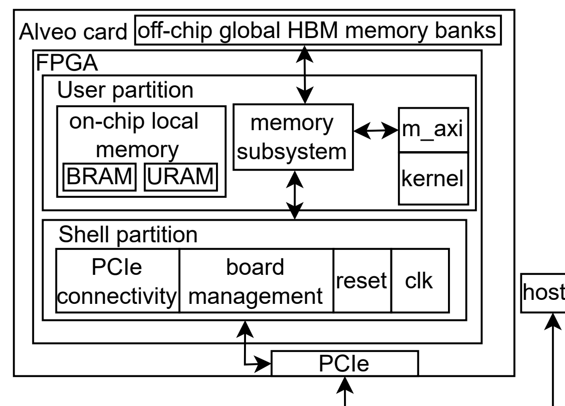


Fig. 1: Block diagram of the modules on an Alveo card.

The FPGA of Alveo cards is empty before loading any target platform. The target platform is the initial HW design (firmware) implemented at system boot time [13], on the FPGA to set up the cards for a particular use [2]. It defines FPGA attributes with a partitioned HW configuration [14] comprising two regions [12], [13], [15]: (a) a user partition (dynamic logic region) with the reconfigurable FPGA resources still available after implementing the shell partition; and (b) a shell partition (target platform) with fixed logic for basic infrastructure, such as PCIe connectivity and board management, to operate the FPGA and to transfer data to and from the dynamic region. This partitioning enables dynamic region updates while essential infrastructure services remain operational [14].

Data transfers between the host and kernels occur through global memory. The host accesses global-HBM data via a

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

connection that crosses the card's PCIe interface, shell partition, and memory subsystem within the user partition, while kernels use their memory-mapped Master-Advanced eXtensible Interfaces (M_AXI) and the memory subsystem. The shell partition HW enables an Alveo card and its host to connect without developer intervention.

IV. Vitis Design Flows and Kernel Execution Modes

Vitis supports three design flows. Two are for acceleration cards: Vivado IP and Vitis kernel flow, respectively, for RTL designers and SW developers. The other one is the Embedded System–design flow.

This work examines the Vitis kernel flow, in which Vitis builds an FPGA binary file to program logic elements in the user partition of the target platform. This file defines the local memory topology, kernels, clocking data, and kernel connectivity [13], using AXI4 M_AXI interfaces, which connect kernels to off-chip HBM banks through read and write channels [2].

Depending on execution mode, Alveo cards' FPGA kernels can be SW-controlled, allowing host applications to interact with them, or data-driven (free-running). The latter starts automatically and only stalls when no input data is available in the streams they read. SW-controllable kernels can be either user-managed, suitable for the Vivado IP design flow, or Xilinx RunTime (XRT) managed, a type of user-managed kernel tailored for the Vitis design flow.

V. Optimizations in High-Level Synthesis

HLS performance optimizations accelerate the compute-intensive sections of C/C++ code with parallel execution. Execution parallelism inherent in HW accelerates SW with equivalent functionality running on general-purpose processors. Optimized HW should maximize concurrency, overall throughput (rate of data processing) [12], and minimize latency (cycles to produce data from input) [16].

To accomplish the acceleration goal, developer-guided optimization pragmas should generate optimized HW as shown in Figure 2. It implements three main parallelization techniques supported by Vitis HLS. These techniques guide the synthesis of C/C++ kernel functions in generating optimal RTL code that parallelizes HW execution as much as possible. The techniques are pipelining, which enables execution concurrency with different operations, each with a different dataset; unrolling, which represents pure parallelism and replicates a loop's body to run all its operations on multiple data simultaneously; and dataflow, which decouples the execution of sequentially linked HW modules that have data dependencies on their immediate predecessors. The overall parallelization results from the combined effect of these techniques.

The techniques can be classified as macro-level architectural optimization, improving concurrency or parallelism among tasks implemented as C/C++ functions, or fine-grained micro-level architectural optimization, optimizing

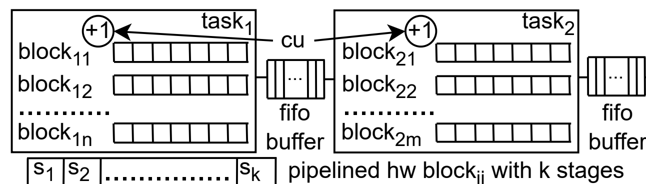


Fig. 2: Types of HW parallelism. $block_{ij}$ is the j-th HW copy of the body of the unrolled loop of the ith task of a dataflow region. CU is the control unit with a counter required by the loop of $task_i$ before unrolling.

the performance of functions and their operations [2]. Both depend on sufficient HW resources for synthesis.

A macro-level architectural optimization is task dataflow (dataflow pipelining). It exploits coarse-grain parallelism [17], enabling task-level pipelining. It follows the producer-consumer design pattern, identifying sequential tasks, stages, or C/C++ functions within a top-level function. They can execute concurrently to boost kernel throughput, even if not in parallel. This pattern connects consecutive producer and consumer functions with streaming data channels, either a first-in-first-out buffer (FIFO) or a ping-pong RAM buffer (PIPO). Each producer sequentially stores data elements in its output channel, which the consumer retrieves. If the data is consumed in the same order, an on-chip local-memory FIFO buffer can link both functions, allowing data flow between them. FIFO buffers streamline the critical data path by eliminating the need for index computation to access RAMs. However, when a producer and its consumer of a dataflow region do not produce and consume data in the same order, the top-level function design cannot link the tasks via FIFOs; instead, it needs ping-pong RAM buffers.

This technique overlaps the execution of functions local to the kernel, increasing concurrency. In addition, it decouples the execution of tasks with different throughput rates, reducing function starvation. Typically, the latency of the design decreases, and its throughput increases [2].

Additionally, Vitis HLS supports another macro-level architecture optimization, input/output burst data transfer, which is inferred automatically from the C/C++ code. It accesses multiple consecutive off-chip memory locations with a single request, optimizing latency and increasing memory bandwidth. Kernels should access consecutive memory locations in this way to minimize the burdensome global memory access overhead. Figure 3 depicts a kernel using the recommended load-compute-store design pattern, which enables the dataflow optimization and eases the burst inference. This design pattern isolates read memory access, computing work, and write memory access, respectively, in the three different functions, *load*, *compute*, and *store*, of the load-compute-store standard architecture defined in [12]. The task *compute* can also

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

consist of consecutive calls to smaller functions (*compute0*, ..., *computeN*), forming a task chain in a dataflow region accordingly. The kernel calls these functions in the order already mentioned to enable these optimizations.

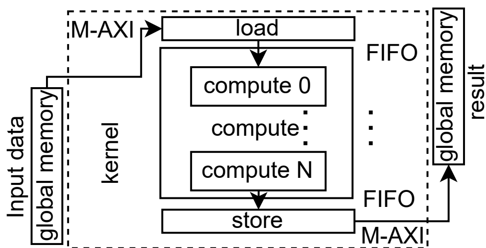


Fig. 3: Load-compute-store design pattern.

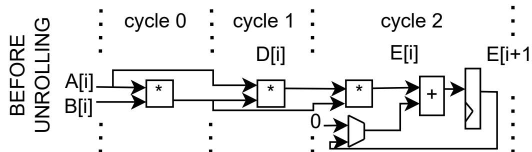
Six fine-grained, micro-architectural optimizations enhance kernel latency or throughput at the function level: two for `for` loops (flattening and unrolling), one for `for` loops and functions (pipelining), and three for functions (inlining, instantiation, and allocation). Except for allocation, they can be described as C/C++ source code modifications; however, they do not alter the code. They ensure that HLS generates optimized HW equivalent to that produced for the modified input code. According to [1], [2], [12], these optimizations operate as follows.

A. Loop Flattening

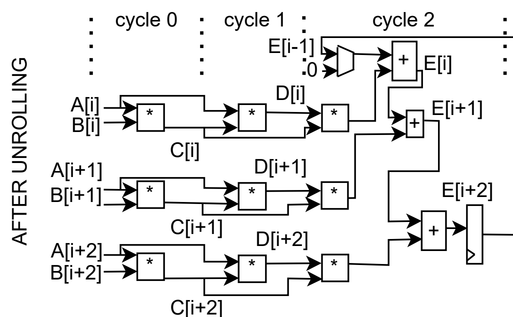
It removes the parent loop of a nested child loop, increasing the child iterations to compensate. Moreover, flattening a structure of more than two consecutive, flattenable nested loops eliminates the ancestors of the innermost loop and leaves a single loop with as many iterations as the product of the number of iterations of each loop. The optimization's fundamentals lie in the HW implementation of a loop. Each loop is controlled by a control unit (CU) implemented as a finite-state machine (FSM). Therefore, flattening a parent loop simplifies the nest's CU by eliminating the parent loop's FSM. This saves two cycles per parent loop iteration: one cycle to move execution from the FSM of the parent to that of its child right before the beginning of the first iteration of the child and one cycle to move it back to the parent's FSM right after the last iteration of the child. Therefore, this saving equals twice the loop bound of the flattened parent in cyles. It also reduces the circuit area devoted to CUs. However, only perfect and semi-perfect loop nests are flattenable. The bounds of semi-perfect loop nests are constant except for the bound of the outermost loop, whereas perfect loops do not have any variable bound. Yet both types have all their code within the innermost loop. Flattening is disabled by default[1] and can be activated or deactivated with #pragma HLS loop_flatten and #pragma HLS loop_flatten off, respectively.

B. Loop Unrolling

It partially (i.e., only some iterations) or completely eliminates a loop by replicating the loop body's RTL code based on the unrolling factor, uf. Figure 4 illustrates this. If there are no dependencies among the copies, loop unrolling enables true parallelism in HW, i.e., each iteration can run in parallel on a different HW. This decreases loop latency. Consequently, the throughput increases, and the function containing the unrolled loop requires a higher Input/Output bandwidth[2] to handle the higher data-processing rate.



(a) A non-pipelined 3-cycle multicycle critical data path implements the iterations of an unrolled loop. Before unrolling, there are M iterations: each calculates one output value in three cycles, and the loop variable $i \in [1, 2, 3, \dots, M]$.



(b) After unrolling with a factor of 3, there are $\lceil \frac{M}{3} \rceil$ iterations: each calculates 3 output values in 3 cycles, and $i \in [1, 4, 7, \dots \lceil \frac{M}{3} \rceil \times 3 - 2]$.

Fig. 4: The effect of unrolling.

Unrolling is disabled by default.[1] #pragma HLS unroll factor =uf enables this optimization, with uf being an integer unrolling factor. It must satisfy $1 < \text{uf} \le \text{TC}$, with TC being the loop trip count. For $\text{uf} = \text{TC}$, HLS performs complete unrolling, whereas for $\text{uf} < \text{TC}$, it unrolls the loop partially.

C. Loop and Function Pipelining

It optimizes HW that implements a body loop or a function [1]. In C/C++, consecutive loop iterations or function calls start only the previous one has finished. The minimum time between two back-to-back executions is the initiation interval (II), which equals iteration or function calls latency in C/C++. Pipelining reduces II while preserving the latency by scheduling the sequence of operations into consecutive stages with pipelining registers between them. Hence, each stage runs concurrently with data from subsequent loop iterations or function calls [2], which start without waiting for the previous one to finish. Thus, pipelining reduces II and optimizes throughput but does not improve

[1]In the absence of loop pipelining.

[2]If the loop accesses array data, memory partition might be necessary to reach the required I/O bandwidth.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

latency [18], [19]. II reduction is limited by dependencies among operations in different stages, particularly feedback between loop iterations or function calls. Pipelining aims for the lowest II within clock constraints unless an II is specifed. However, if set, HLS prioritizes reaching that II over clock constraints. The HLS pragma statement #pragma HLS pipeline ii=<int> controls this optimization.

When applied in a loop nest, it triggers three optimizations in sequence according to HLS build logs: (1) unrolling all the loops downward, (2) flattening loops upward until reaching a non-flattenable loop, and (3) pipelining the loop with the pipelining pragma. The order between pipelining-triggered unrolling and flattening does not affect the HW result because they target different loops. Flattening eliminates the idle cycles between the consecutive executions of the pipelined loop nested under other loops.

This optimization is enabled by default for loops (auto-pipelining) when no pipelining pragma is provided to guide HLS. To disable auto-pipelining:

1) Update the default HLS settings by setting the parameter pipeline_loops of the Vitis HLS configuration command to zero and pass a Tcl script with this setting at the call to the compile command. This disables default auto-pipelining for all kernel codes.
2) Include the pragma #pragma HLS pipeline off within a specific nested loop to disable auto-pipelining for all the loops in the nest.

If pipeline_loops is above 0, auto-pipelining should pipeline loops with a TC above this parameter. It works as a pipelining threshold above which HLS pipelines a loop.

D. Function Inlining

It replaces function calls with their logic, by default for small functions. This can be disabled with #pragma HLS inline off. Inlining is non-recursive, affecting only the function with the pragma, unless specified with the option recursive . In HW, inlining eliminates the associated RTL module.

E. Function Instantiation

It creates function instances for the given input parameters. The clause #pragma HLS FUNCTION_INSTANTIATE variable =k in the function body instantiates the parameter k, producing a customized RTL module for each value passed to the instantiated input port. Fewer remaining inputs after instantiation result in a smaller instance's implementation area, improving latency and throughput. By default, function instantiation is disabled.

F. Function Allocation

Finally, allocation limits the number of RTL instances for an operator or function. Developers set this limit, forcing shared utilization when the C/C++ code uses the function or operator more times than the threshold. This technique optimizes area but might worsen performance.

VI. Kernel Description

Our kernel design simplifies the analysis of key HW optimizations by using a simple critical data path. The computations of our kernel involve four `for` loops perfectly nested, f0 (outermost), f1, f2, and f3 (innermost), with TC two, three, six, and nine, respectively, and a straightforward HW implementation of the innermost loop's body. The core of the computation is $acc+ = la \times lb + C \times lc \times ld$, where la, lb, lc, and ld are the i-th 32-bit integer elements of four different input vectors, respectively; C is an integer constant; and, acc is a 64-bit integer accumulator defined and initialized in the outermost loop with the value zero. The kernel has four input streams, *local_a*, *local_b*, *local_c*, and *local_d*, for the four input vectors, respectively; and an output stream, *local_result_g*, for the output vector. Right before the computation, the innermost loop reads the four input elements in parallel, each from a different stream. The innermost loop also sends, via a stream, the values of the accumulator calculated at the end of the execution of each iteration of the outermost loop. These values constitute the kernel output. Figure 5 below presents the core of computation's HW implementation.

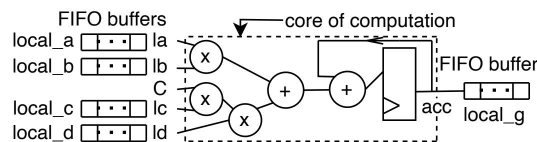


Fig. 5: HW implementation of the core of computation.

Figure 6 illustrates the kernel organization, which slightly varies the standard load-compute-store architecture. It calls five functions: load, preprocess, compute, postprocess, and store in this order. *Preprocess* and *postprocess* convert the kernel I/O's wide data width to the narrower data width of *compute*'s data path. Global memory accesses are 512 bits wide to maximize their throughput. #pragma HLS dataflow schedules the five functions to run concurrently and builds FIFO channels for the local hls :: stream variables connecting consecutive functions.

Each I/O port of the kernel is linked to a different HBM global-memory bank through an M_AXI adapter dedicated to this bank and its corresponding FIFO queue.

The function load reads host data elements from the accelerator's global memory, with burst transfers. The function preprocess unpacks input data read by *load*. The function compute processes output vector elements using the *preprocess*-unpacked data. The function postprocess packages the output elements. The function store saves the output vector to global memory, using burst transfers. Below are some clarifications regarding the burst transfers, preprocessing, and postprocessing. For input burst transfers, this design aggregates 16 32-bit memory accesses into 512-bit memory accesses via a separate M_AXI adapter for each

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

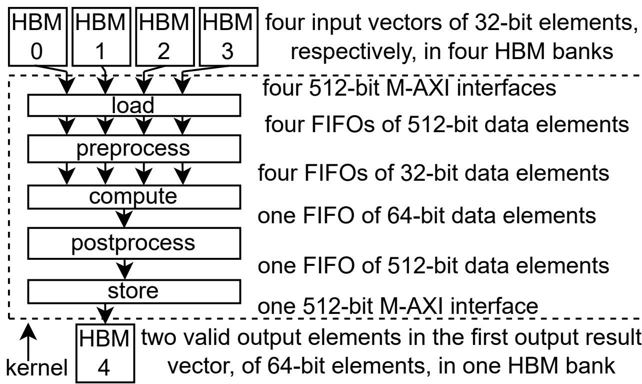


Fig. 6: Kernel organization.

kernel input port. A shared M_AXI adapter for multiple kernel input ports would prevent *load* from parallelizing a burst read from each input source. Output burst transfers aggregate eight 64-bit memory accesses into a single 512-bit memory access via the output M_AXI adapter.

Both *preprocess* and *postprocess* have a two-level nested loop each; the outer iterates over 512-bit data packages, and the inner over 16 32-bit data elements.

VII. Reverse-Engineering the Behavior of Pragmas

This section presents findings on the pragmas mentioned in Section V, not documented in AMD/Xilinx manuals [2], [12]. They were obtained through reverse engineering over a hundred experiments to provide further information on the pragma effects. They explain how Vitis HLS determines optimizations based on C/C++ code and its pragma configuration, given sufficient HW resources. Note that the correspondence between pragma and optimization is one-to-one for the pragmas flatten and unroll, but differs for the pragma pipeline, which triggers three optimizations in sequence: unrolling, flattening, and pipelining.

Two groups of experiments were conducted to study Vitis HLS' pragmas. The first group focused on each pragma's behavior, both in isolation and in combination with others. The findings from these experiments are presented in Sections VII-A and VII-B. These experiments are unrelated to those on area and latency in Section IX and involve placing pragmas within different loops of four-level hierarchies. The experiments implemented updates to the kernel described in Section VI, transforming its perfect loop nest into various loop hierarchies, including up to four-level loop nests with different loop bounds from those of the original kernel. Moreover, some nests were not even semi-perfect because of sibling loops. For each pragma and loop for pragma placement, they explored pragma behavior under all possible configurations of these settings: config_compile-pipeline_loops and config_unroll-tripcount_threshold commands in HLS default settings, nesting level of the loop; no pragma, pragma activation or deactivation; unrolling factor; isolated or combined pragmas; loop position in the code, relative to that of sibling loops; number of child loops (zero, one, or multiple child loops); several pragmas with or without overlapping scopes and colliding effects, in the same or different loops; and with or without sibling or cousin loops.

The second group's experiments examined area and latency in the HW design optimized with pragmas, using the kernel from Section VI, after establishing pragma behavior.

A. Single Pragmas

We use the tag **F<id>** to identify our findings below, for both Vitis 2022.2 and 2024.1 unless otherwise specified:

F1 To the best of our knowledge, results from automatic optimizations do not differ from those observed after embedding pragmas in the C/C++ code.

F2 HLS optimizes loop hierarchies by applying pragmas and optimizations in this order: (1) unrolling pragmas; (2) flattening pragmas; (3) pipelining-triggered optimizations: unrolling and flattening in this order; and (4) pipelining itself.

This order remains valid regardless of whether the pragmas disable or enable optimizations, and whether pipelining is developer-guided with pragmas or auto-pipeline[3]. Each step's transformations inherit the hierarchy left after going through the previous steps.

F3 Flattening (disabled/enabled) propagates from the loop with the pragma, inclusively, upward in the loop hierarchy even if it encounters non-semi-perfect loops[4]. It skips such loops and continues to flatten higher loops. The upward propagation stops when it encounters a loop with any of these elements: (a) a pragma loop_flatten off, or (b) descendant loops disabling upward flattening as explained in finding **F4**.

F4 When a loop has multiple child loops, some might disable flattening, while others might enable it. The rules to determine whether the combined effect of the child loops disables flattening upward are[5]: (a) Child loops with explicit pragma loop_flatten take priority, enabling or disabling flattening upward, and (b) The higher a child loop is in the C/C++ code, the greater the priority of a pragma loop_flatten (off) in this loop.

F5 Unrolling a loop does not trigger unrolling for its descendant loops in the loop nest hierarchy.

F6 Partially unrolling a loop has the following effects if the unrolling factor does not divide the TC, using integer division. It replaces the loop with another partially unrolled with a TC equaling the result of the integer division plus one. In the last iteration, only a portion of the replicated HW is used.

[3]Auto-pipeline is an algorithm that Vitis HLS launches automatically to select a loop for pipelining.

[4]In Vitis 2024.1, disabling or enabling flattening propagates upward from the loop with the pragma or its parent, respectively, but only through semi-perfect or perfect loops.

[5]Vitis 2024.1 disables flattening propagation upward at sibling loops.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

F7 HLS always triggers auto-pipelining for the innermost loop of a perfect or semi-perfect loop nest in the absence of pragma directives that activate pipelining, unrolling, or flattening. This contradicts [2], which states that HLS should only pipeline the innermost loop automatically if its bound exceeds the pipelining threshold set in Vitis HLS. Otherwise, it should attempt to pipeline the next loop higher in the hierarchy. This process should continue until a suitable loop is found according to the specified criteria.

F8 Adding `#pragma HLS pipeline off` within any loop disables auto-pipelining in all loops along root-to-leaf paths traversing the loop containing this statement.[6]

F9 Adding `#pragma HLS pipeline off` within a function has no effect if the function call is within a pipelined loop. This behavior holds regardless of whether the loop pipelining is developer guided or auto-pipeline.

F10 A pragma enabling pipelining takes precedence over one disabling pipelining whenever they are in two different loops, in the same branch.

F11 When there is a pipelined loop with at least one sibling loop, HLS creates a non-pipelined artificial[7] logic level directly above the pipelined loop.

Regardless of the presence of other pragmas, the auto-pipeline algorithm targets the loop nest hierarchy resulting from unrolling all loops containing a complete unrolling pragma. However, given any branch from root to leaf, auto-pipelining only applies to the branch if none of its loops contains a pragma pipeline or pipeline off[8]. It selects only one loop in the branch for pipelining based on this priority: (1) The branch's deepest loop with a partial unrolling pragma; (2) The branch's deepest loop that meets this condition: all its ancestors have one single child and no pragma disabling flattening.

If no loop meets the specified conditions, the algorithm recursively calls itself for the subtree rooted at the deepest loop in the branch with a parent satisfying this condition: having at least one descendant[9] with a pragma enabling partial unrolling or pipelining.

In the absence of HW resources to pipeline the selected loop[10], the auto-pipeline algorithm attempts to pipeline the first suitable loop downward in the loop hierarchy. If an II violation occurs, HLS does not pipeline the selected loop. However, this violation does not prevent the pipelining-triggered optimizations: upward flattening and downward unrolling.

[6] In Vitis 2024.1, disabling loop pipelining with a pragma affects only the loop containing the pragma and its ancestors.

[7] Artificial means the logic does not derive from a C/C++ function or operator in the source code, unlike standard logic.

[8] In Vitis 2024.1, for a given branch, disabling pipelining in certain loops allows auto-pipelining to select one from those below them.

[9] This condition is valid regardless of the generational distance, or the number of hierarchical levels, between ancestor and descendant.

[10] HW resources alone do not guarantee loop selection by Vitis 2024.1's auto-pipeline; a valid II under its latency is required. Otherwise, auto-pipeline searches for the first suitable loop downward in the hierarchy.

B. Combinations of Pragmas

This section presents further findings on the combined effects of pragmas. All behaviors can be explained by referring to the findings in Section VII-A.

Below is a list of all combinations of two pragmas.

- `UaP`: Unrolling above (or in same loop as) Pipelining.
- `UaF`: Unrolling above (or in same loop as) Flattening.
- `PaU`: Pipelining above (or in same loop as) Unrolling.
- `PaF`: Pipelining above (or in same loop as) Flattening.
- `FaU`: Flattening above (or in same loop as) Unrolling.
- `FaP`: Flattening above (or in same loop as) Pipelining.
- `dFaP`: disabled Flattening above (or in same loop as) Pipelining.

Combinations of higher number of pragmas are explained by reduction to two-pragma combinations. This taxonomy categorizes pragmas based on their position in a loop nest hierarchy. Each combination is denoted by an acronym, `<pragma1>a<pragma2>`, where `F` stands for flattening, `U` for unrolling, and `P` for pipelining, and `<pragma1>` is located above (in an ancestor) or in `<pragma2>`'s loop.

The interplay between pragmas occurs only when their scopes overlap and their effects collide. Considering **F5**, flattening eliminates loops upward, and pipelining operates both upward (applying flattening) and downward (applying unrolling once to each descendant of the pipelined loop and regardless of generational distance), we describe two cases for each pragma combination based on whether the pragmas are in different or the same loop.

In `UaP`, if pragmas are in different loops, their scopes do not overlap. HLS unrolls and pipelines the loops containing the corresponding pragma, respectively. It also applies flattening to the ancestors upward, based on findings **F3, and F4**. Thus, HLS creates multiple pipelined loops. If both pragmas are in the same loop, various situations can arise:

1) If unrolling is complete, HLS should first remove the scope of pipelining based on finding **F2**. However, HLS would not be able to pipeline it afterward. Thus, HLS fails and raises an error due to a pragma conflict between pipelining and unrolling.
2) If unrolling is partial, HLS unrolls the loop partially, based on finding **F8**, and pipelines it for any of the following two situations: a) The loop is orphan, has at least one child and the pipelining optimization is guided automatically; and b) the loop has a parent.
3) If unrolling is partial and the loop is orphan, for any of the following two situations, HLS converts the partial unrolling pragma into a complete unrolling

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

optimization: a) The loop does not have children; and b) the loop has children, and the pipelining is guided by pragma rather than automatically.

Whenever `UaP` triggers partial unrolling, HLS creates some artificial[7] logic just above the pipelined loop level. Loop pipelining does this whenever it cannot flatten all the loops above, up to the function level. In this case, a partially unrolled loop between the function level and the pipelined loop prevents the exhaustive flattening.

In `UaF`, if both pragmas are in the same loop and the unrolling is complete, HLS raises a conflict error. It is illogical to flatten a loop that already has a complete unrolling pragma. The loop does not exist anymore after unrolling; HLS applies unrolling first, before flattening. Otherwise, if the unrolling is incomplete or above the flattening, HLS applies flattening whenever the post-unrolling code complies with the flattening requirements[11].

The scopes of the pragmas in `PaU` overlap when they are in different loops. However, the pragmas' effects do not collide, unless the unrolling is partial, in which case HLS ignores it. Hence, `PaU ≡ P`. The pragmas of each configuration with complete unrolling only collide if they are in the same loop. In such a case, HLS ignores the unrolling pragma, and applies pipelining, so `PaU ≡ UaP`, and `UaP`'s behavior has been described above.

`PaF` raises a conflict according to HLS build log, if its two pragmas are in different loops. However, the build finishes without errors. Unexpectedly, the loop flattening pragma is ignored, and only the pragma pipeline is applied. If both are in the same loop, HLS resolves the conflict again, ignoring the flattening pragma, so `PaF ≡ P`. Therefore, regardless of pragma location, HLS prioritizes the last one in the sequence based on finding **F2**.

In `FaU`, if both pragmas are in different loops, their scopes do not overlap. Otherwise, their scopes overlap and their effects collide. HLS raises a conflict and fails.

The scopes of the two pragmas in `FaP` overlap when in different loops, but this does not lead to a collision in their effects. Hence, `FaP ≡ P`. The pragmas of each configuration only collide if they are in the same loop. In such a case, `FaP ≡ PaF ≡ P`; HLS ignores the flattening pragma, and applies pipelining. That is, HLS behaves as if the pragmas were in different loops.

Finally, in `dFaP`, the scopes of disabling the flattening pragma and pipelining-triggered flattening overlap, and their effects collide. The recursive effect of the pipelining-triggered flattening ends when it reaches the pragma that disables flattening. Whereas pipelining is impossible without completely unrolling the loops nested below the pipelined loop, disabling flattening does not prevent pipelining. Nonetheless, it does inhibit the pipelining-triggered flattening. This behavior remains even if both pragmas are in the same loop.

[11]Findings **F3** and **F4** in Section VII-A expose the conditions under which flattening propagation upward halts.

VIII. Reverse-Engineering an Estimation Model of Loop and Function Latencies

Calculating a function's latency is crucial for optimizing its HW. Latency is the number of cycles needed to execute all the loops, operations, and function calls within the function. Similarly, loop latency is the number of cycles required to execute all iterations.

The compute function is the core of any kernel, containing computations. Vitis HLS estimates function latency adding latencies of loops and child function calls. Hence, we focus on estimating loop and function latencies.

The following latency equations below apply to any loop type. They emerge from analyzing Vitis HLS' Schedule Viewer reports for dozens of experiments, varying the kernel explained in Section VI. They changed the core complexity (e.g., additional multiplications), calculation order, loop hierarchy (e.g., adding sibling loops), TC per loop, nesting functions within loops or another function, and pipelining settings.

For a function with only a root loop (which may contain nested loops but nothing else outside), Equation 1 estimates the function's latency, L_F.

$$L_F = L_I + (L_L^0 - \text{isPipe}) + L_O * \text{isPipe} \tag{1}$$

where L_I represents the latency to start the execution of a loop, L_L^0 the root loop's latency, L_O the latency to finish the loop's execution, and isPipe is a variable that indicates whether the loop is pipelined. Note that the value of L_L^0 is different if the loop is pipelined. In the case where the loop is pipelined, (isPipe = 1), HLS schedules two sets of operations in L_I: 1) those required to start up the loop, specifically the initialization of the loop induction variables; and 2) those in the first cycle of the first iteration of the loop, particularly those checking the loop index is above the bound and incrementing the loop index. Parallelizing these two sets is the reason for subtracting one cycle from the loop latency L_L^0.

Regarding unrolling, this optimization leaves loops with fewer iterations, eventually eliminating them completely. Therefore, it modifies the TC; and the loop latency, L_L^0, if there are dependencies between iterations. Equations (3) to (5) consider both parameters.

In the case of latency of a function with a root non-pipelined loop (isPipe = 0) and nothing else outside this loop, L_F does not include L_O, and Equation (1) simplifies to Equation (2) because (i) the execution of the initialization operations does not overlap that of the operations of the first cycle of the first iteration of the loop, and (ii) the scheduler knows in advance the function execution

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

finishes right after the last cycle of the last iteration of the loop because there is not parent loop above.

$$L_F = L_I + L_L^0 \tag{2}$$

For the root of a loop nest, Equation (3) calculates its latency. It assumes that the superscripts indicate the loop position in a nest of K loops, with $i = 0$ for the root and $k-1$ for the innermost loop. Equation (3) should be applied to all loops up to the penultimate one inclusive.

$$L_L^i = TC^i \times (L_I^i + L_L^{i+1} + L_O^i) \tag{3}$$

where L_L^i, TC^i, L_I^i, and L_O^i represent the latency, trip count, the cycle to start the child loop execution, and the cycle to move the execution back to the parent, respectively. L_L^{i+1} is the child loop latency.

For the innermost loop in a nest hierarchy, i.e., $i = k-1$, Equation (4) calculates its latency, representing the most general case of the latency of a single loop.

$$L_L^{k-1} = (TC^{k-1} - 1) \times II^{k-1} + (IL^{k-1} - 1) + M^{k-1} \tag{4}$$

IL stands for loop iteration latency and is the latency of the first iteration's operations required to start the pipeline. M matches the number of cycles from cycle zero of each iteration of the pipelined loop to cycle $N-1$ inclusive, being N the cycle with the operation calculating the increment of one of the canonicalized induction variables of the optimized loop. Vitis HLS adds a canonical induction variable to each loop flattened on top of the pipelined loop. The canonicalized induction variable for cycle N corresponds to this loop: the outermost of all those flattened on top of the loop HLS will pipeline afterward. According to [20], canonicalization modifies loop induction variables and their calculations to ease analysis and transformations.

In the case of the latency of the non-pipelined innermost loop of a nest hierarchy, $II = IL$ and $M = 1$, so that Equation (4) can be written as Equation (5).

$$L_L^{k-1} = TC^{k-1} \times IL^{k-1} \tag{5}$$

IX. Impact of Optimization Configurations on DSE

This section discusses the impact of HLS optimizations on the latency and area of our kernel's compute function (Section VI). Three experiment types reveal this impact: (i) baseline experiment, without optimization, i.e., with pragmas disabling HLS auto-pipeline, setting the reference performance; (ii) experiments to evaluate the performance improvement of a single atomic pragma in isolation from others; and (iii) experiments to evaluate the effects of non-atomic optimizations or the interplay of several pragmas. They aim to explore the design space by evaluating different pragma locations and combinations, including non-atomic optimizations, such as pipelining; their objective is not to support the findings and latency estimation model presented in Sections VII and VIII, respectively. The worst negative slack of the baseline kernel is +0.055 ns and +0.032 ns, respectively, in Vivado 2022.2 and 2024.1. The 41.82% decrease is consistent with a more pessimistic model[12]

Anyway, in all cases below, our kernel is built with Vitis 2022.2, target frequency 300 MHz, the platform xilinx_u50_gen3x16_xdma_5_202210_1 and trip counts $TC^0 = 2$, $TC^1 = 3$, $TC^2 = 6$, and $TC^3 = 9$.

A. Baseline testbench

The baseline experiment disables all optimizations and does not encapsulate the core of computation within a function call. Table I presents the results. The latency of function *compute*, L_F, is 2357 cycles. Since it is not pipelined, the initiation interval equals the latency. The calculations below, using Equations (2) to (5), are based on values in Table I.

$$L_L^3 = TC^3 \times IL^3 = 9 \times 7 = 63$$
$$L_L^2 = TC^2 \times (L_I^2 + L_L^3 + L_O^2) = 6 \times (1 + 63 + 1) = 390$$
$$L_L^1 = TC^1 \times (L_I^1 + L_L^2 + L_O^1) = 3 \times (1 + 390 + 1) = 1176$$
$$L_L^0 = TC^0 \times (L_I^0 + L_L^1 + L_O^0) = 2 \times (1176 + 2) = 2356$$
$$L_F = (L_I^0 + L_L^0) = 1 + 2356 = 2357 \text{ cycles}$$

TABLE I: Latency and hardware resources without optimizations. IL stands for iteration latency, II for initialization interval, and FF, LUT and DSP is the number of FPGA flip-flops, look-up tables (LUT), and DSPs, respectively.

Function Loop	**Latency** (cycles)	**IL** (cycles)	**II** (cycles)	**FF**	**LUT**	**DSP**
L_F	2357	-	2357	1013	562	7
f0	2356	1178	-			
f1	1176	392	-			
f2	390	65	-			
f3	63	7	-			

The core of computation uses seven DSPs: three for $la \times lb$, one for $C \times lc$, and three for the product of $C \times lc$ and ld.

B. Flattening in Isolation

Table II summarizes the result of flattening at different loop nest levels. Flattening simplifies the loop's CU logic, reducing FFs while increasing LUTs. Deeper isolated flattening[13] within the nest hierarchy results in lower latency and a smaller increase in the total number of LUTs, though the reduction in FFs becomes less significant. Flattening multiple loops does not compromise FF reduction compared to flattening a single loop (f0). Nonetheless, most configurations increase LUTs to a greater extent, except

[12]Vitis 2023.2+ HLS updated the scheduler's timing model to match Vivado timing predictions, resulting in a more pessimistic delay model.

[13]Enabling flattening in just one loop is possible by deactivating its upward effect with the pragma loop_flatten off in the parent loop.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

for the one that flattens f1 and f2, which offers the best trade-off between latency and area.

TABLE II: Latency and hardware resources under isolated flattening at different loop nest levels. HLS only flattens the selected loops[13]. The column "Flattened Loops" lists eliminated loops in parentheses. All loops in parentheses are replaced by a single loop, and the loops in boldface contain the flattening pragma. On the right-hand side of the arrow are the resulting loops. The new loop's name is not representative of the names assigned by Vitis; their purpose is to identify that they are new inferred loops.

Flattened Loops	Latency (cycles)	FF	LUT	DSP
none (baseline)	2357	1013	562	7
(f0, **f1**) → f01	2353	854	609	7
(f1, **f2**) → f12	2345	958	584	7
(f2, **f3**) → f23	2285	959	578	7
(f0, f1, **f2**) → f012	2341	865	659	7
(f0, **f1**), (f2, **f3**) → f01, f23	2281	864	635	7
(f1, f2, **f3**) → f123	2273	857	586	7
(f0, f1, f2, **f3**)→ f0123	2269	878	700	7

C. Unrolling and Disabling Auto-Pipelining

Table III shows the effects of unrolling at different levels in a loop nest and disabling auto-pipelining. It confirms that the more loops HLS unrolls, the lower latency the result has. Furthermore, the deeper the unrolled loops were within the hierarchy, the lower latency and more area the result had: Unrolling the whole innermost loop achieved the best latency optimization yet also increased area the most, compared to unrolling any other loop. The same effect was noticeable in the other combinations.

TABLE III: Latency and hardware resources after complete-unrolling at different levels of a loop nest. All experiments unroll only the selected loops and disable auto-pipelining.

Unrolled Loops	Latency (cycles)	FF	LUT	DSP
none (baseline)	2357	1013	562	7
f0	2354	942	760	7
f1	2133	1241	871	7
f2	2045	1331	1192	7
f3	629	2519	1240	20
f0+f1	2130	1270	1348	7
f0+f2	2042	1930	2091	7
f1+f2	2001	1716	2412	7
f0+f3	626	2753	1768	20
f1+f3	597	2956	2115	20
f2+f3	383	3733	3204	20
f0+f1+f2	1998	2516	4579	7
f1+f2+f3	341	6555	8075	20
f0+f1+f2+f3	331	11006	15433	20

Elements of each input vector enter the computation core sequentially due to the FIFOs structure linking *preprocess* and *compute*. This results in sequential execution despite unrolling. Execution parallelism would be possible if the input channels allowed multiple consecutive elements of each input vector to enter *compute* in parallel as a vector dataset. Moreover, increasing execution parallelism for different input elements of each input vector is more feasible by unrolling only the innermost loop rather than a higher-level loop. Doing the latter would cause this: HLS would try to parallelize the execution of operations that might work on data from different datasets, which did not reach *compute* in parallel. Those operations might involve input elements so distant in the input sequence that they belong to different datasets, making parallelization infeasible.

Unrolling the innermost loop increased the number of DSPs: HLS implemented the products in the computation core with 20 DSPs. Otherwise, HLS used only 7 DSPs. Thus, parallelizing the execution of the products for consecutive vector elements consumed more DSPs than the baseline. Despite expectations, loop unrolling did not always replicate the HW of the loop body.

There was some replication in FFs and LUTs when unrolling only ancestor loops, but the number of DSPs remained unchanged, indicating HLS reused the HW of the unrolled loops body. For this reason, unrolling did not work as expected in the aforementioned experiments.

To determine the optimality of default Vitis HLS behaviors, the following paragraphs evaluate the effects of non-atomic optimizations and pragma directive combinations.

D. Pipelining and Pipelining-Triggered Flattening

Pipelining the innermost loop achieved the best latency among the experiments that pipelined only one loop, as shown in Table IV. This setup's area was the most optimized, even though its latency did not improve signficantly. Table IV also presents the results of `dFaP`. The deeper the pipelining, the greater the impact of pipelining-triggered upward flattening on latency improvement, compared to pipelining without the corresponding flattening[14]. This makes sense; the deeper the pipelined loop, the more ancestor loops HLS can flatten. Vitis HLS's auto-pipeline default behavior in perfect or semi-perfect loop nests suggests that pipelining the innermost loop offers the best trade-off between performance and area. However, it is unclear if this holds without the performance boost of pipelining-triggered flattening. We propose setup `dFaP`[15] to test this. Pipelining the outermost loop improved latency the most in this setup. However, it significantly increased the area; higher pipelining within the nest unrolled more loops. Moreover, the substantial area increase prevents this alternative from being considered, despite the latency improvement. Hence, this setup is incomplete, and combining it with other techniques is necessary. For example, to allow tuples of multiple input elements of each input vector to enter the kernel in parallel. The best compromise between area and latency among the

[14]The experiments without flattening disable it with a pragma.

[15]dFaP disables flattening in an ancestor loop of the target loop of pipelining or in the same one.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

mentioned experiments is pipelining the innermost loop, without an abnormal area increase. Table IV shows this.

TABLE IV: Latency and hardware resources comparing standard pipelining vs. `dFaP`, i.e., without auto-flattening[14]. Latency Reduction compares flattened and unflattened scenarios. The last row equals auto-pipelining.

Flattened Loops	Pipeline Loop	Latency (cycles)	Latency Reduction	FF	LUT	DSP
enabled	f0	334		7103	8249	26
`dFaP`	f1	351		4870	3680	31
enabled		335	5%	4872	3678	
`dFaP`	f2	407		3833	1729	34
enabled		342	16%	3785	1812	
`dFaP`	f3	737		1245	744	7
enabled		332	55%	1071	849	

E. Unrolling-Triggered Flattening

Next, we evaluate whether it would make sense for Vitis HLS to support an unrolling-triggered flattening. This means automatic flattening after insertion of an unrolling pragma, with both pragmas in different loops, auto-pipeline disabled, and complete unrolling. This configuration, labeled `FaU`, barely changed latency and the number of FFs or LUTs, as shown in Table V.

Flattening did not enhance the parallelization effects of unrolling because it occurred post-unrolling, in accordance with finding **F2**. A hypothetical flattening before unrolling would synthesize HW from a loop with a higher TC, enabling greater unwinding and parallelization.

TABLE V: Comparison of latency and HW resources for complete unrolling vs. `FaU` evaluated across all flattening combinations in loops above the unrolled one(s) . Flattening a single loop is achieved using combinations of pragmas that enable or disable flattening. The loop's tag next to the configuration acronym indicates the flattened loops. All configurations include one pragma disabling pipelining, in any loop above those unrolled.

Unrolled Loops	Flattened Loops	Latency (cycles)	FF	LUT	DSP
f2	none	2045	1331	1192	7
	`FaU`: (f0, **f1**)	2041	1332	1254	7
f3	none	629	2519	1240	20
	`FaU`: (f0, **f1**)	625	2428	1297	20
	`FaU`: (f1, **f2**)	617	2415	1229	20
	`FaU`: (f0, f1, **f2**)	613	2437	1329	20
f2+f3	none	383	3733	3204	20
	`FaU`: (f0, **f1**)	379	3732	3240	20

F. Pipelining-Triggered Optimization: Flattening vs. Unrolling

We propose testing whether replacing pipelining-triggered flattening with pipelining-triggered unrolling improves results. This configuration, similar to `UaP` with complete unrolling and both pragmas in different loops, disables flattening above the pipelined loop. Table VI shows no latency improvement over standard pipelining, as *compute* does not receive multiple input vector elements in parallel.

TABLE VI: Comparison of latency and hardware resources of `UaP` plus disabling pipelining-triggered flattening vs. `P` (first row in table, which coincides with last row in Table IV). In all scenarios, pipelining is applied in the innermost loop, f3. The loop's tag next to the configuration acronym indicates the loops where unrolling is applied.

Unrolled Loops	Flattened Loops	Latency (cycles)	FF	LUT	DSP
none	yes	332	1071	849	7
`UaP`: f0	no	698	1594	1265	7
`UaP`: f1		705	2319	1663	7
`UaP`: f2		695	3683	2808	7
`UaP`: f0+f1		690	3336	2831	7
`UaP`: f1+f2		685	9112	6951	7
`UaP`: f0+f1+f2		684	17722	13535	7

G. Encapsulating Loops in a Function

The experiments evaluate configurations that replace the innermost loop body with a function call (encapsulation). Table VII summarizes the results, denoted by IDs with the suffix `E` followed by a number.

We focus on the most effective configuration identified for the case without function encapsulation, i.e., pipelining the innermost loop (experiment `E1`). It worsened latency from 2357 to 3005 cycles (a 27 % increase) due to a 9-cycle latency of innermost loop iterations, 2 cycles more than the baseline's latency. Transitioning execution into and out of this non-pipelined module takes one cycle each. It also increased the number of DSPs, FFs, and LUTs by 0 %, 3 %, and 14 %, respectively, compared to the baseline. This area increase results from invoking a function, which adds a new RTL module with its corresponding FSM. HLS could not reduce latency below six cycles because of a carried dependence constraint between consecutive function calls. Each call takes the value calculated by the previous call. Hence, eliminating function encapsulation or function inlining might eliminate these constraints and improve the latency after pipelining. Data dependencies between successive calls at the same hierarchy level would enable this improvement. Our example shows this.

Experiments `E2` to `E9` show the combined effects of different pragmas, including function instantiation. They aim to identify pragma combinations that achieve HW replication of the computation core and increase HW parallelism without pipelining or auto-pipelining.

Instantiating the function without enabling unrolling did not create different instances, as illustrated in `E2`; area and latency were unchanged. Additionally, `E3` shows that allocating multiple functions without unrolling the innermost loop was also insufficient. Even though this test slightly increased the number of FFs, this did not indicate HW replication. Latency remained virtually unchanged.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

Unrolling a loop that calls a function eliminated the loop without instantiation, allocation, or inlining pragmas, as shown in `E4`. Latency decreased slightly, and the number of FFs and LUTs barely increased. Despite unrolling, HLS did not replicate the RTL function module for each call. This required either instantiation or allocation pragmas.

Unrolling the innermost loop and function inlining, `E5`, did increase the area significantly. Inlining eliminated the function, allowing HLS to replicate the HW implementing the function's code, the core of computation. Aditionally, unrolling noticeably decreased the latency.

`E6` unrolls the innermost loop and instantiates a function parameter, whose value varies with each innermost loop iteration. HLS created different function copies, each optimized for its respective call. Although latency improved visibly, the number of FFs and LUTs increased significantly.

`E7` to `E9` evaluate configurations with unrolling and allocation, varying the number of allocated instances (L). They evaluate the effect of allocation limits on HW replication. The improvement remained consistent regardless of the limit. However, the area increased based on the allocation limit. When L=2, HLS created an additional HW function with the area increase shown in `E7`. When $L = 5$, `E8`, and $L = 9$, `E9`, HLS created four and eight additional functions, respectively, compared to $L = 2$. Compared to the baseline, the area growth for $L = 5$ and $L = 9$ was approximately four and eight times that of $L = 2$, respectively. Moreover, the increase in DSPs matched the number of additional function copies excluding the original one. Hence, lower instantiation limits result in fewer function copies. Additionally, the evident area increase with allocation limits of two or more indicates that HLS optimizations produced multiple function copies, one for each function call.

In summary, inlining offered the best performance-area trade-off among the configurations in Table VII.

TABLE VII: Latency and area in setups with unrolling, function allocation and instantiation (no pipelining).

ID	**Unrolled Loop**	**Allocation**	**Inline**	**Instantiation**	**Replication**	**Latency** (cycles)	**FF**	**LUT**	**DSP**
E1	-	no	no	no	no	3005	1046	643	7
E2	-	no	no	yes	no	0%	0%	0%	0%
E3	-	L=9	no	no	no	1%	1%	0%	0%
E4	f3	no	no	no	no	-12%	1%	15%	0%
E5	f3	no	yes	no	yes	-79%	141%	93%	186%
E6	f3	no	no	yes	yes	-23%	269%	355%	0%
E7	f3	L=2	no	no	yes	-12%	86%	83%	100%
E8	f3	L=5	no	no	yes	-12%	340%	281%	400%
E9	f3	L=9	no	no	yes	-12%	674%	536%	800%

Figure 7 shows HW resources, measured using the number of FFs of the implementation as a proxy, vs. the performance, measured as the circuit latency in cycles for the 43 experiments in Tables II to VI. It distinguishes the results per table along with the Pareto front. Three non-dominated implementations in the Pareto front achieve the best design based on these metrics (sorted from the highest number of FFs to the lowest): (1) unrolling f0+f1+f2+f3 in Table III, (2) pipelining f3 in Table IV, and (3) unrolling f1+f2+f3 in Table III. It also illustrates that although there is not a significant variation among latencies in the experiments in Table VI, there is a clear difference in their number of FFs. This observation applies to the experiments in Table IV as well, except for the penultimate experiment of this table. On the other hand, the optimizations in Table III and Table V present variance in the number of FFs and circuit performance. Finally, the experiments in Table II show marginal variation in circuit performance but are the best in HW resources, as stated in the following paragraph regarding Figure 8. Based on these results, unrolling or auto-pipelining achieves the best trade-off between latency and number of FFs.

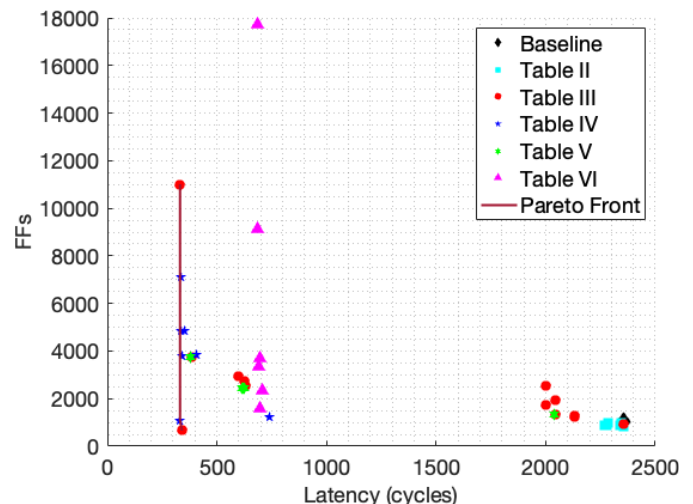


Fig. 7: FFs vs. Circuit latency for all experiments.

Figure 8 shows the number of LUTs vs. the number of FFs. The linear regression analysis ($r^2 = 0.73$) indicates that FFs mostly explain the variation in LUTs. Nevertheless, the variation in the number of LUTs for a given number of FFs suggests potential for DSE to optimize circuit resources. There are six non-dominated designs (sorted from the highest number of LUTs to the lowest): (1) baseline, (2) flatten (f2, **f3**), (3) flatten (f1, **f2**), (4) flatten (f1, f2, **f3**), (5) flatten (f0, **f1**) (all of them in Table II), and (6) unroll f1+f2+f3. Flattening different loops achieves the best HW resource efficiency, although the optimal number of flattened loops should be determined empirically. All other optimizations focus on improving circuit latency, which in turn increases resource use.

X. Conclusions

Custom hardware (HW) accelerators, designed using high-level synthesis (HLS) methodologies, can accelerate computationally expensive sections of software applications.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

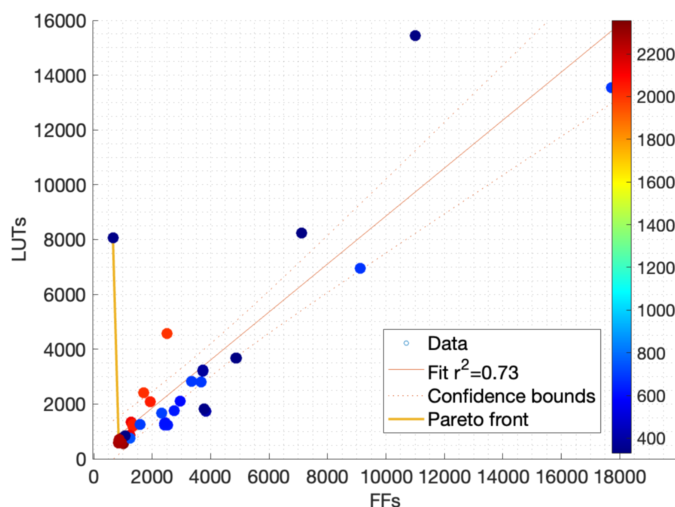


Fig. 8: Number of LUTs vs. Number of FFs for all the experiments. The color bar represents latency in cycles.

Mastery of the implementation of the HLS methodology and optimizations in the tool Vitis HLS improves process productivity and shortens the design cycle.

We evaluated 150+ setups to understand pragma application and interactions, model Vitis HLS' latency estimation, and measure their impact on latency and area.

These pragmas guide RTL generation using three parallelization techniques: unrolling, pipelining, and dataflow. Ideally, the C/C++ coding style aligns with Xilinx recommendations, with perfect or semi-perfect loops, and no interloop dependencies. In this scenario, a unique pragma scenario provides the best results. Otherwise, designers can combine pragmas to achieve optimal HW design.

The default pipelining behavior, auto-pipelining, achieves the best balance between performance and area. Nevertheless, unrolling the innermost loops is the second-best approach, after pipelining. On the other hand, loop flattening optimizes resources and marginally reduces latency.

Nevertheless, for a given code, multiple implementations can exist with up to two metrics fixed and still a significant variance in the third. Hence, implementations with the same latency can have different numbers of LUTs or FFs, with up to 2× variance in HW resources. Likewise, multiple latencies can exist for a given number of FFs or LUTs, with some implementations having up to 2.5× more LUTs than others with the same FFs and similar latencies. Hence, decreasing FFs usually tends to reduce LUTs and vice versa due to their correlation. However, there are still different implementations with the same number of FFs and a significant variability in LUTs. This variability of one metric, even preserving the other two, requires DSE for optimal implementation; designers should not settle for the first candidates that optimize only one figure of merit.

Decreasing design area can indirectly improve performance for multi-kernel applications by allowing more kernels in the FPGA, thus minimizing reconfigurations.

All in all, pipelining the innermost loop, as suggested in the Xilinx user guides, and avoiding function encapsulation offers the best compromise between area and performance.

Future research could develop a tool for automatic DSE and address why unrolling non-innermost loops in a perfect loop did not replicate DSP slices.

XI. Data Availability

The benchmarks used in this paper will be accessible after emailing any of the following authors: jkoronis@ucm.es or ogarnica@ucm.es. Access to the data will be granted to researchers from public research institutions (universities and research institutes) upon commitment to reference the data.

References

[1] M. Fingeroff, "High-Level Synthesis Blue Book," 2010. [Online]. Available: https://www.cse.usf.edu/~haozheng/teach/cda4253/doc/hls/hls_bluebook_uv.pdf.

[2] "Introduction • Vitis High-Level Synthesis User Guide (UG1399) • Reader • AMD Adaptive Computing Documentation Portal," [Online]. Available: https://docs.xilinx.com/r/2022.2-English/ug1399-vitis-hls.

[3] "5. Using the HLS Kernel in the Vitis Tool — Vitis™ Tutorials 2020.1 documentation," [Online]. Available: https://xilinx.github.io/Vitis-Tutorials/2020-1/docs/build/html/docs/Getting_Started/Vitis_HLS/using_the_kernel.html.

[4] "A Survey and Evaluation of FPGA High-Level Synthesis Tools | IEEE Journals & Magazine | IEEE Xplore," doi: 10.1109/TCAD.2015.2513673.

[5] A. Cilardo and L. Gallo, "Interplay of loop unrolling and multidimensional memory partitioning in HLS," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2015, pp. 163–168. doi: 10.7873/DATE.2015.0798.

[6] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Performance Modeling and Directives Optimization for High-Level Synthesis on FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 7, pp. 1428–1441, Jul. 2020, issn: 1937-4151. doi: 10.1109/TCAD.2019.2912916.

[7] A. Mehrabi, A. Manocha, B. C. Lee, and D. J. Sorin, "Prospector: Synthesizing Efficient Accelerators via Statistical Learning," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2020, pp. 151–156. doi: 10.23919/DATE48585.2020.9116473.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3526053

[8] Y.-k. Choi and J. Cong, "HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8. DOI: 10.1145/3240765.3240815.

[9] D. Jamma, O. Ahmed, S. Areibi, and G. Grewal, "Hardware accelerators for the K-nearest neighbor algorithm using high level synthesis," in *2017 29th International Conference on Microelectronics (ICM)*, Dec. 2017, pp. 1–4. DOI: 10.1109/ICM.2017.8268827.

[10] C. Lo and P. Chow, "Model-based optimization of High Level Synthesis directives," *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–10, Aug. 2016, [TLDR] This paper evaluates the use of SMBO for selecting HLS directives and extends the method to relate multiple uses of the same directive within a design and finds that the proposed extension can further improve the convergence rate over the standard method. DOI: 10.1109/FPL.2016.7577358.

[11] "Using Alveo Cards to Accelerate Dynamic Workloads," AMD, [Online]. Available: https://www.amd.com/en/training/customer/adaptive-computing/customer-training/using-xilinx-alveo-accelerate-workloads.html.

[12] "Getting Started with Vitis • Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393) • Reader • AMD Adaptive Computing Documentation Portal," [Online]. Available: https://docs.xilinx.com/r/2022.1-English/ug1393-vitis-application-acceleration.

[13] "Xilinx® Runtime (XRT) Architecture — XRT Master documentation," [Online]. Available: https://xilinx.github.io/XRT/master/html/index.html.

[14] "SOM Landing Page — SOM documentation." (Sep. 17, 2023), [Online]. Available: https://xilinx.github.io/kria-apps-docs/home/build/html/index.html#.

[15] "Alveo U280 Data Center Accelerator Card User Guide," 2019. [Online]. Available: https://www.mouser.com/pdfDocs/u280userguide.pdf.

[16] "Optimizing the Hardware Function," [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2019_1/sdsoc_doc/optimizing-hardware-function-uww1504034429970.html.

[17] "Enabling Concurrent Processing with DATAFLOW," [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/obx1504034310502.html.

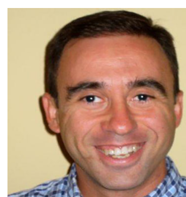
[18] N. Weaver, "Pipelining RISC-V," 2020. [Online]. Available: https://inst.eecs.berkeley.edu/~cs61c/sp20/pdfs/lectures/lec14.pdf.

[19] N. Riasanovsky, "RISC-V CPU Control, Pipelining." [Online]. Available: https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture12.pdf.

[20] "LLVM's Analysis and Transform Passes — LLVM 18.0.0git documentation," [Online]. Available: https://llvm.org/docs/Passes.html.

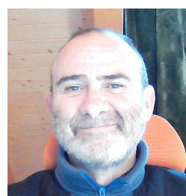
A. Biographies and Author Photos

Jorge Koronis is currently pursuing a Ph.D. in Computer Engineering at Universidad Complutense de Madrid (UCM). He holds a B.S. in Computer Engineering from UCM and an M.S. in Finance from Universidad Pontificia Comillas. He currently works at the Department of Computer Architecture and Automation at UCM's Faculty of Computer Science. His research focuses on hardware acceleration of software applications and high-level synthesis.



Oscar Garnica holds a PhD in Physical Sciences (2002) from the Computer Science program at Universidad Complutense de Madrid. He has been an Associate Professor at the Faculty of Computer Science at UCM since 2007. Previously, he worked as an ASIC designer at Lucent Bell Labs and LSI Inc. His research focuses on adaptive fault-tolerant hardware, AI accelerators for biomedical engineering applications, and chip verification.

J. Ignacio Hidalgo is Full Professor at Universidad Complutense de Madrid. His research interests are in hardware and software bioinspired techniques focusing on medical applications. He has a degree in Physics (Electronics) from UCM and a Ph. D. from the same university in Physics in Computer Science and automation program. He has co-authored more than 70 journal papers and has acted as an Organizer of different evolutionary computation conferences and workshops.



Juan Lanchares Dávila is a Physics graduate (1990) with a PhD in Physical Sciences (1995), and a Full Professor at the Department of Computer Architecture and Automation, Faculty of Computer Science, Universidad Complutense de Madrid. His research focuses on adaptive fault-tolerant hardware for autonomous systems, namely digital filters, and deep neural networks to estimate blood glucose levels using Kalman and particle filters.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/