Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electrical, Electronics and Communications Engineering (37^{th} cycle)

Making acceleration more amenable with novel high-level synthesis techniques for FPGAs

Giovanni Brignone

* * * * *

Supervisor
Prof. Luciano Lavagno

Doctoral Examination Committee:
Prof. Christian Pilato, Politecnico di Milano
Dr. Mirjana Stojilović, École Polytechnique Fédérale de Lausanne
Prof. Jordi Cortadella, Universitat Politècnica de Catalunya
Prof. Mario Roberto Casu, Politecnico di Torino
Prof. Mihai T. Lazarescu, Politecnico di Torino

Politecnico di Torino
December 2024

This thesis is licensed under a Creative Commons License, Attribution - Noncommercial-NoDerivative Works 4.0 International: see `www.creativecommons.org`. The text may be reproduced for non-commercial purposes, provided that credit is given to the original author.

I hereby declare that the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . . . .

Giovanni Brignone
Turin, December 2024

Summary

Today, the market demands increasingly sophisticated application-specific digital hardware architectures, but traditional register-transfer level (RTL)-based design methods remain expensive and time-consuming. High-level synthesis (HLS) offers a promising alternative by enabling designers to specify accelerator functionality in high-level languages (e.g., `C/C++`) and automating hardware generation. Although HLS simplifies design-space exploration (DSE) and functional verification, achieving quality of results (QoR) comparable with manually-optimized RTL still requires manual low-level optimizations that involve implementation-specific details, which extend beyond purely functional descriptions originally intended by HLS. Therefore, hardware design predominantly remains at RTL.

This dissertation argues that broader adoption of HLS is hindered by insufficient abstraction, particularly in the management of low-level details. In addition, many optimization opportunities offered by high-level abstractions remain largely underutilized.

To address these challenges, this dissertation introduces novel methodologies aimed at further raising the abstraction level in HLS designs. Key contributions include the introduction of semi-automated memory management for field-programmable gate arrays (FPGAs), via an open-source caching library that enables throughput comparable with manual on-chip buffering, with minimal engineering effort. Designs optimized with the proposed cache are from 8 × to 113 × faster than those accessing the off-chip memory directly. Moreover, this dissertation presents an automatic optimization for digital signal processor (DSP) utilization through open source compiler optimization passes, which identify superword-level parallelism and pack multiple low-precision operations to single high-precision DSPs provided by FPGAs. The methodology matches manual DSP packing, automatically saving on average 70 % and 50 % DSPs in addition-intensive and multiplication-intensive benchmarks, respectively.

This dissertation also proposes new approaches that take advantage of the high abstraction level of HLS for improving the QoR. Specifically, the task-level multi-pumping takes advantage of the HLS pipelining feature to fully exploit the available timing slack and maximize resource sharing, reducing DSP utilization by up to 40 % at equal throughput.

Contents

List of Acronyms

ALAP as late as possible

AMA add-multiply-add

API application programming interface

AXI advanced extensible interface

BB basic block

BE backend

BLAS basic linear algebra

BRAM block random access memory

CCC cache-compute-cache

CDC clock domain crossing

CFG control flow graph

CNN convolutional neural network

CPU central processing unit

DDG data dependence graph

DDR4 double data rate 4

DFG dataflow graph

DRAM dynamic random access memory

DSE design-space exploration

DSP digital signal processor

EDA electronic design automation

FE frontend

FF flip-flop

FIFO first-in-first-out

FPGA field-programmable gate array

FSM finite-state machine

FU functional unit

GAT graph attention

GSM global system for mobile communications

HBC hybrid boundary condition

HDL hardware description language

HLS high-level synthesis

HW hardware

II initiation interval

IP intellectual property

IPI intellectual property integrator

IR intermediate representation

L1 level 1

L2 level 2

LCS load-compute-store

LRU least recently used

LUT look-up table

MAC multiply and accumulate

MAD multiply-and-add

MAXI master advanced extensible interface

MCDFG multi-clock dataflow graph

MMM matrix-matrix multiplication

MSB most significant bit

MVM matrix-vector multiplication

PIPO ping-pong

PPA power, performance, and area

QoR quality of results

RAW read-after-write

RO read-only

RTL register-transfer level

RTM reverse time migration

RW read-write

SCDFG single-clock dataflow graph

SIMD single-instruction multiple-data

SNN spiking neural network

SoC system-on-chip

SOTA state-of-the-art

SW software

URAM ultra random access memory

VMS virtual molecule screening

Chapter 1

Introduction

In the post-Moore era, advanced application-specific digital hardware architectures are increasingly essential to compensate for the slowdown in technology scaling [1]. However, developing such hardware remains an expensive and time-consuming process. The design phase involves specifying the architecture's functionality and implementation by defining various interconnected modules, which often include both hardware and software components. This phase also requires design-space exploration (DSE) to find architectures that meet power, performance, and area (PPA) constraints. The verification phase follows, ensuring that the implementation satisfies both functional and non-functional (e.g., quality of results (QoR)) requirements.

Inspired by the software industry's shift to higher abstraction levels (e.g., from assembly to `C` language) that allow developers to focus on functionality while compilers handle optimizations, high-level synthesis (HLS) offers similar advancements for hardware. HLS enables designers to specify the functionality of hardware accelerators in high-level languages such as `C/C++` and automatically translates these descriptions into optimized hardware implementations. The higher abstraction level in HLS is intended to enhance designers' productivity, which is crucial for short time-to-market, low-volume applications like field-programmable gate array (FPGA) accelerators, where minimizing non-recurring engineering costs is paramount.

Compared to mainstream register-transfer level (RTL) hardware development, which resembles programming in assembly language, HLS offers higher-level abstractions that simplify the design and maintenance of complex architectures by incorporating software features, such as object-oriented programming, in hardware description. Moreover, HLS enables functional verification based on software simulations, which run orders of magnitude faster than RTL simulations and can leverage well-established software tools, such as debuggers.

A key advantage of HLS is the ease of DSE, achieved by adjusting high-level parameters such as loop unrolling factors or pipeline directives. In contrast, DSE at the RTL requires considerable manual modifications to the datapath and associated control logic. However, a given design point manually-optimized at RTL typically shows better QoR compared to the one generated by HLS. To achieve QoR comparable to RTL, HLS designs typically require extensive manual optimizations, which shift the source code away from purely functional descriptions, diminishing the benefits of HLS. Examples of such manual efforts include managing the memory hierarchies, that typically require scratchpad-like explicit data movement between on-chip and off-chip memories, and handling FPGAs' digital signal processor (DSP) resources, which require explicit inputs pre-processing to fully exploit the available precision when targeting

low precision applications.

As a result, HLS adoption lags behind RTL, since its productivity gains are often overshadowed by the manual efforts required to achieve competitive QoR. This dissertation argues that increasing the abstraction level in HLS, particularly for details that currently require manual optimization for efficient implementation, is necessary for broader adoption of HLS. Furthermore, many optimization opportunities enabled by high-level abstractions remain largely under-exploited and should be explored to improve QoR.

To tackle the aforementioned challenges, *this dissertation introduces novel methodologies to further raise the abstraction level of HLS designs targeting FPGAs and to leverage the high-level abstractions of HLS for QoR improvement.*

The novel methodologies to further raise the abstraction level of HLS include:

- Elevating FPGA memory management to cache parameters tuning, discussed in Chapter 3, published in "Array-Specific Dataflow Caches for High-Level Synthesis of Memory-Intensive Algorithms on FPGAs" [2], and implemented as an open-source HLS library, available at `github.com/brigio345/DaCH`. The library improves throughput by up to 113 × with minimal design effort.

- Automating DSP packing via a compiler-based approach, discussed in Chapter 4 and in "SILVIA: Automated Superword-Level Parallelism Exploitation via HLS-Specific LLVM Passes for Compute-Intensive FPGA Accelerators" [3]; the optimization framework is open-source at `github.com/brigio345/SILVIA`. It reduces the DSP utilization for additions by 70 % and for multiplications by 50 %, on average.

The novel techniques to leverage the high-level abstractions of HLS for QoR improvement include:

- The task-level multi-pumping methodology, discussed in Chapter 5 and "A DSP shared is a DSP earned: HLS Task-Level Multi-Pumping for High-Performance Low-Resource Designs" [4]. It saves up to 40 % DSPs without reducing performance.

Chapter 2

Background

This section introduces the foundational concepts for this dissertation. It discusses FPGA memory resources and the load-compute-store (LCS) architecture, which serve as a foundation for the cache-compute-cache (CCC) architecture defined in Chapter 3, to optimize memory accesses in FPGA designs. It describes the optimizations of FPGAs' DSP utilization for low-precision applications, which the compiler transformations described in Chapter 4 automate. It introduces key concepts in HLS, including resource sharing, controllable via the pipeline initiation interval (II) parameter, and the dataflow architecture, crucial for defining the task-level multi-pumping technique discussed in Chapter 5.

2.1 High-level synthesis

Figure 2.1 summarizes the flow from untimed `C/C++` to hardware. The HLS flow is typically split between the frontend (FE) and backend (BE) stages. The FE translates high-level code into an intermediate representation (IR) and applies software-based optimizations such as loop

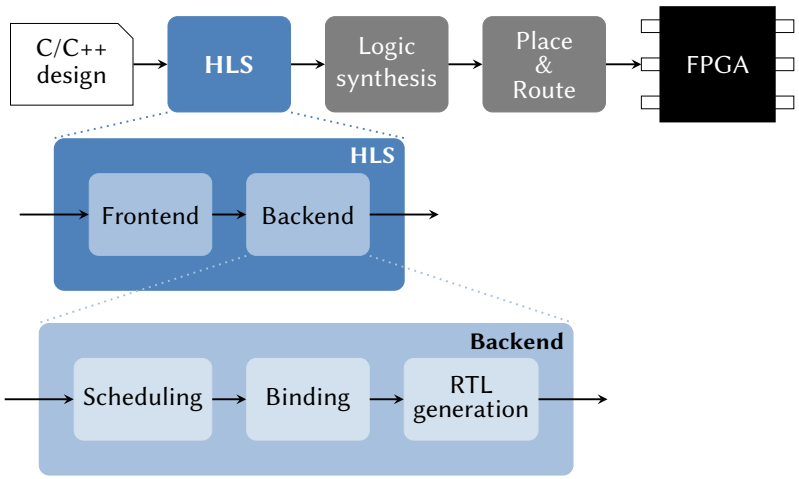
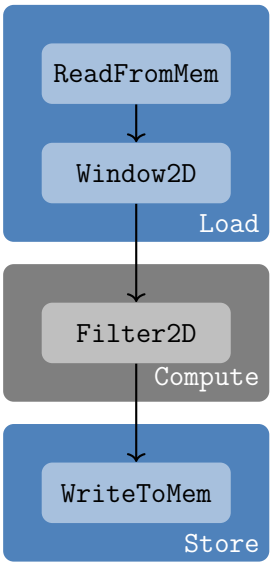


Figure 2.1: The high-level synthesis workflow.



(a) Dataflow graph

```
#define FILTER_V_SIZE 2
#define FILTER_H_SIZE 2

void Filter2D(hls::stream<window> &window_stream,
    hls::stream<char> &pixel_stream,
    char coeffs[FILTER_V_SIZE][FILTER_H_SIZE]) {
  for (int x = 0; x < width * height; x++) {
#pragma HLS PIPELINE II = II_Filter2D
    window w = window_stream.read();
    int sum = 0;
    for (int row = 0; row < FILTER_V_SIZE; row++) {
      for (int col = 0; col < FILTER_H_SIZE; col++)
        sum += w.pix[row][col] * coeffs[row][col];
    }
    pixel_stream.write(sum);
  }
}
```

(b) Source code of the compute task

Figure 2.2: A 2D convolution kernel complying with the load-compute-store design pattern.

unrolling, constant propagation, and dead code elimination.

The BE applies hardware-specific transformations. The first BE step is scheduling, that maps operations in the IR to specific clock cycles, with techniques like pipelining. Next, binding assigns the scheduled operations to hardware components, such as functional units (FUs) or memories. Finally, RTL generation translates these scheduling and binding decisions into synthesizable RTL code.

This section considers as an example a 2D convolution HLS kernel [5] modeled as a dataflow graph (DFG), shown in Fig. 2.2a, compliant with the LCS architecture.

2.1.1 Scheduling

Scheduling assigns operations to specific clock cycles; thus, it also implements loop and function *pipelining*. Designers can constrain the II of the pipelines (i.e., the number of clock cycles between the start of successive pipeline iterations), which is lower bound by the resource constraints and the data dependencies. The data dependence graph (DDG) (i.e., the graph whose nodes are the instructions in a design and whose edges are the data dependencies between the instructions) enables to compute the minimum II allowed by the data dependencies as

$$II_{\min} \triangleq \max_{\theta} \left\lceil \frac{\text{latency}_\theta}{\text{distance}_\theta} \right\rceil, \tag{2.1}$$

where θ is any cycle in the DDG, latency_θ is the sum of the latencies of each node belonging to θ and distance_θ is the total dependence distance along each edge belonging to θ [6]. The cycle maximizing Eq. (2.1) is called the *critical cycle*.

For example, consider the following loop to be scheduled:

```
for (int i = 0; i < N; i++)
    a = a + b;
```

The read-after-write (RAW) dependency on a, produced at the i-th iteration and consumed at the $i+1$-th iteration, introduces a cycle θ in the DDG. $latency_\theta$ is the latency of the adder computing a+b. $distance_\theta$ is 1 since a is consumed at the iteration after it is produced. Therefore, Eq. (2.1) implies that the minimum II for this loop equals the latency of the adder.

The clock constraints determine how many operations fit within a clock cycle, thus affecting the latency of operations, impacting the critical cycle and, in turn, the II lower bound. However, the II constraints take precedence over clock constraints in *relaxed timing* mode, yielding lower II pipelines in exchange for potential HLS timing violations. These are usually acceptable at HLS time since HLS timing estimations may be overly pessimistic [7], and downstream implementation steps may resolve them.

2.1.2 Binding

Binding assigns each operation to a compatible functional unit, depending on resource and performance (e.g., clock frequency, latency) constraints.

Resource sharing is a crucial binding optimization that maps operations of the same type to the same functional unit, scheduled on different clock cycles or under mutually exclusive conditions (e.g., on different conditional branches). The II constraints directly affect the degree of resource sharing. In particular, if a pipeline scheduled with an II of II_i cycles computes N_i^{OP} operations (OPs) of the same kind at each iteration, the binding step allocates N_i^{FU} functional units (FUs), with

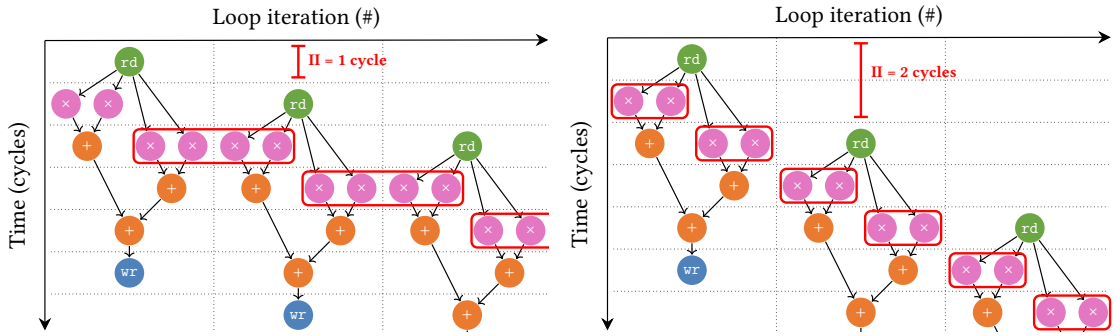
$$N_i^{\mathrm{FU}} \triangleq \left\lceil \frac{N_i^{\mathrm{OP}}}{II_i} \right\rceil. \tag{2.2}$$

Note that the operations can be either computations or memory accesses. The functional units associated with the memory operations are ports proportional to the partitioning factors (i.e., the number of sub-memories into which a memory resource is divided to increase its parallelism). Therefore, larger II values result in fewer functional units and smaller memory partitioning factors.

Consider the Filter2D task from the 2D convolution kernel, whose source is in Fig. 2.2b. Assuming a filter of size 2×2 (i.e., FILTER_V_SIZE = FILTER_H_SIZE = 2), with the schedule with an II of 1 cycle (shown in Fig. 2.3a, where the nodes represent the operations, and the edges the data dependencies), at the steady-state, four multiplications are computed in parallel on different data within the same clock cycle (highlighted by the red rectangle), thus requiring four DSP-mapped multipliers. With an II of 2 cycles instead (Fig. 2.3b), only two multiplications are computed per clock cycle. Therefore, the binding step allocates only two multipliers and shares these among two multiplications each.

2.1.3 Dataflow modeling

Most modern HLS tools support modeling designs as DFGs (e.g., with *dataflow* in AMD Vivado/Vitis HLS [8], *hierarchy* in Siemens Catapult HLS [9], or *task functions* in Intel HLS compiler [10]).



(a) Execution with II 1 cycle (b) Execution with II 2 cycles

Figure 2.3: The impact of pipeline II on resource sharing. In the `Filter2D` task, which filters a 2×2 window per iteration, the pipeline with an II of one cycle (a) computes four multiplications per clock cycle in steady state, while the one with an II of two cycles (b) only two, allocating only half multipliers.

A DFG $G(V, E)$ is a set of tasks $v \in V$ running in parallel and communicating asynchronously through first-in-first-out (FIFO) channels $e \in E$. In HLS, each task is described as a `C/C++` function whose core computational part typically consists of a pipelined loop. Given a compute-bound task v_i clocked at frequency f_i and whose core loop is scheduled with initiation interval II_i, its throughput is

$$\Phi_i \triangleq \frac{f_i}{II_i}. \tag{2.3}$$

The overall DFG throughput matches the one of the *bottleneck task* (i.e., the task with the lowest throughput)

$$\Phi_G \triangleq \min_{v_i \in V} \Phi_i. \tag{2.4}$$

According to Eq. (2.3), the high-level knobs for tuning the throughput of a task v_i are its clock frequency (f_i) and initiation interval (II_i). All tasks share the same clock in single-clock dataflow graphs (SCDFGs). Thus, f_i is the same for all tasks and is bounded by the global (i.e., among all tasks) critical path. Therefore, only the II_i can be tuned independently for each task. In a multi-clock dataflow graph (MCDFG), on the other hand, the clock frequency can be set individually for each task. This additional degree of freedom allows for higher flexibility and tasks frequencies than SCDFG since the clock frequency of a task is limited only by its local critical path and not the one of the whole DFG.

Load-compute-store paradigm

Dataflow modeling enables the LCS paradigm, that decouples the off-chip memory operations and data buffering from the computation. This architecture has the dual purpose of simplifying the design, that is split into smaller and simpler tasks, to achieve better scheduling, and overlapping the load operations of the iteration $i - 1$ with the computation of iteration i and the store operations of the iteration $i + 1$ to improve the overall throughput.

To limit the overhead of the high latency and limited bandwidth of off-chip memories, load and store tasks organize memory accesses into large contiguous blocks via on-chip buffering,

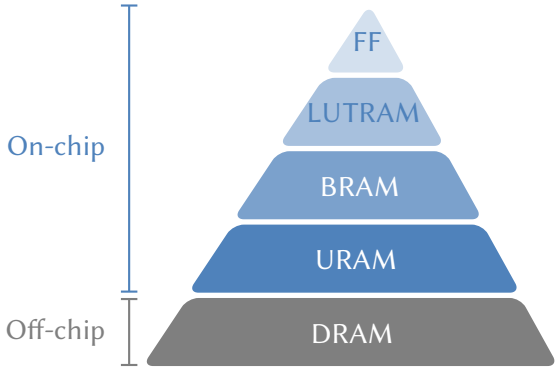


Figure 2.4: The memory hierarchy of current field-programmable gate arrays.

to take advantage of the maximum port width and enable pipelined accesses.

For instance, the 2D convolution kernel in Fig. 2.2a complies with the LCS pattern. The compute task (i.e., `Filter2D`) convolves the i-th window of the input image while the load tasks (i.e., `ReadFromMem` and `Window2D`) buffer the $(i + 1)$-th window, and the store task (i.e., `WriteToMem`) buffer the $(i - 1)$-th pixel.

2.2 Field-programmable gate arrays

FPGAs are configurable circuits that implement combinational logic in look-up tables (LUTs) and DSPs, which are specialized hardware blocks optimized for efficiently computing arithmetic operations. Moreover, they include various on-chip memory types and off-chip memory interfaces.

2.2.1 Memory resources

FPGAs' memory hierarchy (Fig. 2.4) incorporates various on-chip memory resources, including flip-flops (FFs), memory LUTs, block random access memories (BRAMs), and ultra random access memories (URAMs). FFs implement registers, while memory LUTs, BRAMs, and URAMs are larger devices, for buffering and data storage. Additionally, FPGAs often feature advanced extensible interface (AXI) ports or equivalent protocols for accessing off-chip dynamic random access memories (DRAMs).

2.2.2 DSP resources

DSPs are hardware blocks specialized for signal processing and mathematical operations. For instance, the AMD/Xilinx Versal DSP58 [11] provides a 27×24 multiplier in between a 2-input 27-bit pre-adder and a 4-input 58-bit post-adder. Additionally, it contains several multiplexers and registers to configure functionality and pipelining.

Many low-precision algorithms, such as quantized neural networks, underutilize the large precision of DSPs. To address this issue, the AMD/Xilinx UltraScale 48-bit [12] and Versal 58-bit [11] DSPs natively support packing multiple additions or subtractions to a single DSP.

Moreover, several studies introduce novel DSP packing methods [13]–[18] to compute several multiplications and multiply-and-adds (MADs) on a single DSP.

This dissertation focuses on AMD/Xilinx devices as they are the most widely used FPGAs, representing the 55 % of the FPGA market [19]. However, the DSPs from other vendors have similar capabilities. For instance, Intel/Altera DSPs support packing multiple fixed-point additions or multiplications [20].

Additions and subtractions packing

The DSP architectures of the AMD/Xilinx UltraScale [12] and Versal [11] FPGA families support single-instruction multiple-data (SIMD) additions and subtractions. Specifically, they can sum (or subtract) four independent pairs of signed or unsigned operands on up to 12 bits or two independent pairs of signed or unsigned operands on up to 24 bits.

Factor-2 multiply-and-add packing

Fu *et al.* [14] compute two MADs of 8-bit operands, with one shared operand, on a single AMD/Xilinx 48-bit DSP. Specifically, if a_i and c_i are m-bit fixed-point numbers and b_i is an n-bit fixed-point number, $\forall i \in [1, N]$, N DSPs can compute

$$p_a = \sum_{i=1}^{N} a_i \cdot b_i, \quad p_c = \sum_{i=1}^{N} c_i \cdot b_i. \tag{2.5}$$

The value of p_a is mapped to the 30 most significant bits (MSBs) and p_c to the 18 least significant bits of the DSP output. Therefore, to avoid p_c overflowing into the p_a bits,

$$N \leq \begin{cases} \left\lfloor \dfrac{2^{(18-1)} - 1}{2^{(m-1)} 2^{(n-1)}} \right\rfloor, & \text{if } c_i \cdot b_i \text{ is signed} \\ \left\lfloor \dfrac{2^{18} - 1}{(2^m - 1)(2^n - 1)} \right\rfloor, & \text{otherwise.} \end{cases} \tag{2.6}$$

For instance, with 8-bit signed operands, it is possible to chain up to 7 DSPs computing MADs without overflow. It is worth noting that a single DSP can compute two 8-bit multiplications when $N = 1$.

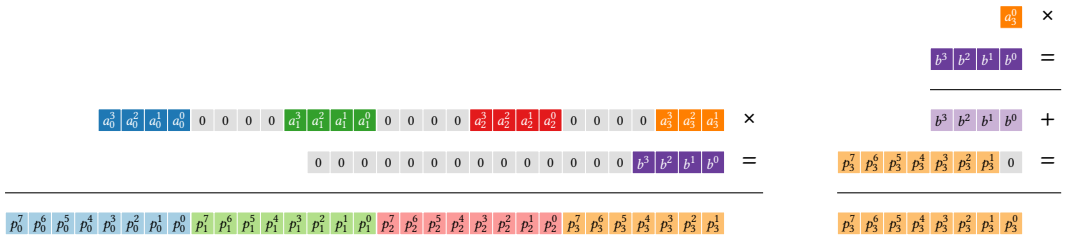
To ensure arithmetic correctness, when the shared operand b_i is signed, the MSB of p_c must be added to p_a.

Factor-4 multiplication packing

The FINN framework [21] provides an open-source implementation of the Preußer and Branca [22]'s architecture that multiplies four 4-bit signed factors by one common 4-bit factor (signed or unsigned) using a single UltraScale/Versal DSP and some additional error-correction LUT logic. In particular, if a_i are 4-bit signed fixed-point numbers and b is a 4-bit fixed-point number, their design computes

$$p_i = a_i \cdot b, \forall i \in [0, 3]. \tag{2.7}$$

In the context of convolutional neural network (CNN) accelerators [23], this packing is particularly effective for the feature map reuse (i.e., the same activation multiplied by different



(a) DSP operations (b) LUTs operations

Figure 2.5: The bit mapping of the novel method for computing four multiplications between four 4-bit unsigned factors and one common 4-bit factor (signed or unsigned).

signed weights). However, it does not support the filter reuse (i.e., the same weight multiplied by different activations, that are unsigned when the activation function is a rectified linear unit).

Therefore, this dissertation presents a DSP packing for the filter reuse too, by introducing a novel packing mechanism to multiply four 4-bit unsigned factors by one common 4-bit factor (signed or unsigned) with a single UltraScale/Versal DSP and a small amount of LUTs as shown by Fig. 2.5. Specifically, a multiplication between two 4-bit values requires 8 bits of output precision to avoid overflow. The UltraScale and the Versal DSPs provide 27×18 and 27×24 multipliers, respectively. Figure 2.5a shows that the proposed packing maps three 4-bit operands interleaved with four zeros of padding (to reserve eight output bits and avoid overflow) and the three MSBs of the fourth operand a_3 to the 27-bit input, while the other input (i.e., the 18 or 24-bit port) accommodates the common operand b. A second step, implemented in LUTs and depicted in Fig. 2.5b, computes the final product p_3 according to

$$p_3 = a_3 \cdot b = ((a_3^{[3:1]} \cdot b) \cdot 2) + (a_3^0 \cdot b). \tag{2.8}$$

The DSP calculates the most expensive portion of the computation (i.e., $a_3^{[3:1]} \cdot b$). The multiplication by 2 does not require any additional hardware, since it is a left-shift by one position. The multiplication of b by one bit a_3^0 is hardware-friendly too (e.g., it can be implemented by the logic AND operation between a_3^0 and each bit of b). Finally, the addition requires a small adder with 4-bit and 8-bit operands. If the common operand b is signed, the products p_i must be corrected similarly to the method by Fu *et al.* [14] (i.e., adding the MSB of a product p_i to the next product p_{i+1}).

Chapter 3

Cache-compute-cache architecture

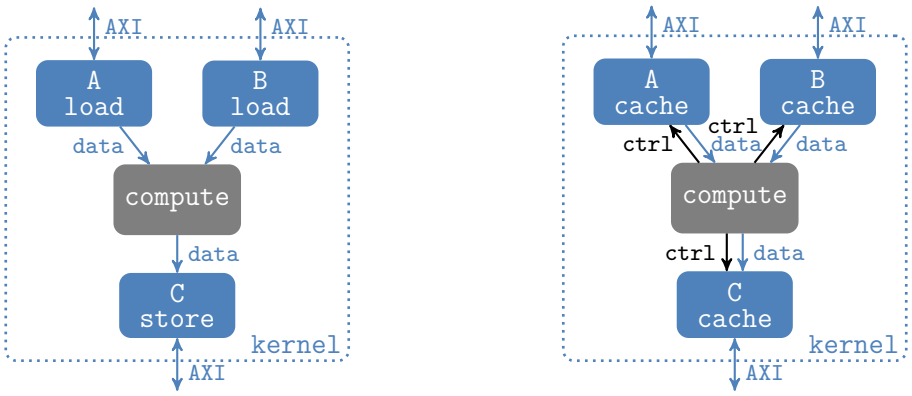
Real-world algorithms typically require more data than the FPGAs' on-chip memories capacity, necessitating the use of off-chip memory. However, off-chip memory often limits performance due to narrow bandwidth and high latency, preventing the full utilization of the FPGA's computing resources. High-performance, scalable designs follow the LCS pattern, where computation overlaps with loading data from off-chip memory to on-chip buffers for future iterations, and storing partial results of past iterations from on-chip buffers to off-chip memory.

Refactoring algorithms into the LCS form is time-intensive (e.g., designers have to extract the memory logic from the computation and to design data buffering tasks [24]) and requires low-level hardware expertise to optimize memory interfaces (e.g., designers must efficiently exploit the interface protocol, for instance, by re-organizing memory accesses for bursts compatibility with AXI interfaces [25]). This dissertation deems HLS should abstract these complexities. Therefore, this dissertation proposes the CCC architecture, implemented through a data-caching library that transforms generic designs into an LCS-like structure. This raises memory management to tuning cache parameters, such as size and number of lines. Experimental results demonstrate that the proposed approach increases the memory bandwidth and reduces the observed memory latency, leading to performance speedups of up to two orders of magnitude, with minimal design effort.

3.1 Overview and related work

The memory hierarchy of FPGA platforms includes high-performance, low-capacity on-chip memory (e.g., registers, BRAMs) and high-capacity, low-performance off-chip memory (e.g., DRAM), as discussed in Section 2.2.1. Efficient designs buffer off-chip data in on-chip memory to mitigate the high latency and low bandwidth of off-chip memory. However, implementing effective data buffering requires significant design effort. To ease this burden, several works [26]–[30] adopt caching techniques from central processing unit (CPU)-based systems for FPGA memory interfaces.

However, cache-optimized memory interfaces do not integrate well with HLS tools, which statically schedule pipelines assuming that memory operations directly access off-chip memory,



(a) Load-compute-store

(b) Cache-compute-cache

Figure 3.1: Load-compute-store designs (a) achieve high throughput at the cost of significant engineering effort for decomposing the algorithm into multiple tasks and designing ad-hoc buffering schemes. The proposed cache-compute-cache architecture (b) reduces the engineering effort with general-purpose cache-based buffering tasks whose parameters are fine-tuned for the specific memory access pattern.

missing out on caching performance benefits. In response, Ma *et al.* [31] introduce an HLS library that integrates data caching into HLS designs at the source level, making the HLS tool cache-aware for optimized kernel performance. However, their architecture blends memory interface logic with computation logic, resulting in high II pipelines.

On the other hand, HLS best practices [32] recommend decoupling memory accesses from computation by using separate tasks, as per the LCS design pattern, presented in Section 2.1.3. In this approach, only load and store tasks access off-chip memory, buffering data in on-chip memory for compute tasks. For instance, Fig. 3.1a shows the LCS architecture for a matrix multiplication kernel computing $C = A \times B$, where A, B, and C reside in off-chip memory. This approach simplifies compute tasks, enabling better static scheduling, while overlapping data loading for the future iterations with the current computation and data storage from past iterations, ultimately increasing throughput.

The main drawback of the LCS design pattern is the significant effort required to convert a generic algorithm into LCS form, often necessitating a complete kernel redesign. To address this, this dissertation proposes the CCC architecture, which combines designer-friendly data caching with the high-throughput benefits of LCS, implemented via the source-level library DaCH (Dataflow Cache for HLS). The CCC architecture decouples cache logic and memory interfaces from the kernel's main computations, as shown in Fig. 3.1b.

The high-level structure of the CCC differs from the LCS architecture only for the control FIFOs from the compute to the load tasks, which send the addresses to access. While LCS load and store tasks use custom buffering logic tailored to specific access patterns, CCC cache tasks employ general-purpose, highly configurable caches. The caches are assigned to specific arrays, allowing for easy tuning to achieve high hit rates at low hardware costs, given that access patterns for individual arrays typically exhibit good locality without interference from other

arrays. Furthermore, the HLS-level cache facilitates early performance evaluation, such as collecting hit rate statistics during software simulation, avoiding costly RTL hardware simulations.

The main contributions of this chapter are:

- Development of the DaCH HLS library that abstracts memory management to tuning cache parameters, generating a cache-compute-cache (CCC) architecture. This architecture mirrors the LCS pattern but eliminates the need for manual refactoring by utilizing general-purpose caches instead of ad-hoc buffering tasks.
- Introduction of a design methodology to achieve high-performance static schedules for irregular control flows.
- Evaluation of the CCC architecture in comparison with the original and LCS kernels.

3.2 Methodology

The DaCH `C++` library for *Vitis HLS* automates the generation of the LCS-like CCC architecture, implementing cache-based buffering tasks. DaCH provides a template-configurable `C++` cache class to enable efficient off-chip memory access with minimal source code modifications. By using DaCH, DRAM-mapped arrays are accessed through cache tasks, separating external memory interfaces and data buffers from compute tasks. Each cache task can expose one or more ports to the compute tasks, with optional private level 1 (L1) caches.

Each cache task is associated with a single off-chip mapped array; therefore, they can be easily tuned for high hit rate at minimal hardware requirements, since access patterns of single arrays typically provide good locality and there is no interference with accesses to other arrays.

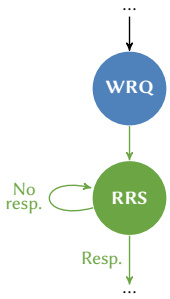
These cache tasks leverage a novel cyclic dataflow protocol to interface with compute tasks and achieve high-throughput static scheduling of irregular control flow graphs (CFGs), which stem from the different miss and hit latencies.

3.2.1 Cyclic dataflow protocol

The CCC architecture introduces cycles in the kernel's DFG both at the cache tasks' interface (i.e., the control and the data FIFOs) and within the cache tasks themselves (to achieve high-throughput, statically-scheduled pipelines even with irregular CFG, due to high-latency miss branch). Therefore, this dissertation defines a protocol to ensure functionality and high throughput in cyclic DFGs.

In this protocol, each dataflow task is designated as either a *Master* or *Slave*. The *Slave* executes operations upon requests from *Master*. Communication and synchronization occur through FIFO channels. The request FIFO, from *Master* to *Slave*, carries inputs for the *Slave*'s operations (e.g., the memory addresses for load operations). Conversely, the response FIFO, from *Slave* to *Master*, returns outputs from the *Slave*'s operation (e.g., loaded data), creating a cycle within the DFG.

The *Master* structure, illustrated in Fig. 3.2, is flexible, allowing it to execute the sub-finite-state machine (FSM) controlling its *Slave* at any point. In contrast, the *Slave* structure, depicted in Fig. 3.3, is precisely defined. It operates as an infinite loop, executing one iteration for each request received from its *Master*. To prevent deadlocks, the *Slave*'s pipeline must be flushable,

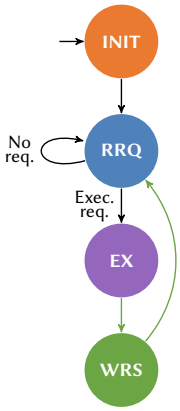


(a) FSM

State	Action	Next state
Write request (WRQ)	Write a request to the *Slave*	*RRS*
Read response (RRS)	Read pending response from the *Slave*, if any	Exit if response was read, *RRS* otherwise

(b) Behavior

Figure 3.2: The *Master*'s behavior in the *Cyclic dataflow protocol*.



(a) FSM

State	Action	Next state
Initialize (INIT)	Initialization	*RRQ*
Read request (RRQ)	Read pending request from the *Master*, if any	*EX* if a request was read, *RRQ* otherwise
Execute (EX)	Execute the requested operation	*WRS*
Write response (WRS)	Write response to the *Master*	*RRQ*

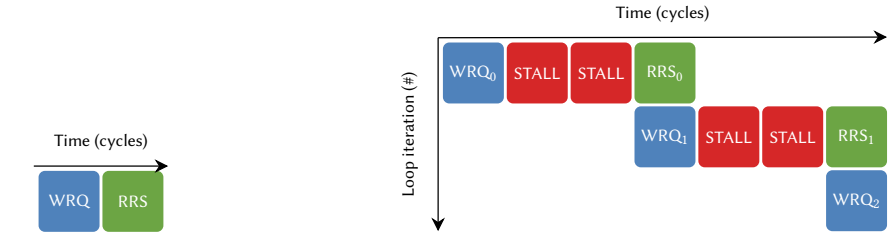
(b) Behavior

Figure 3.3: The *Slave*'s behavior in the *Cyclic dataflow protocol*.

meaning that once a request enters the pipeline, it must proceed through all stages to completion, even in the absence of new requests in the earlier stages.

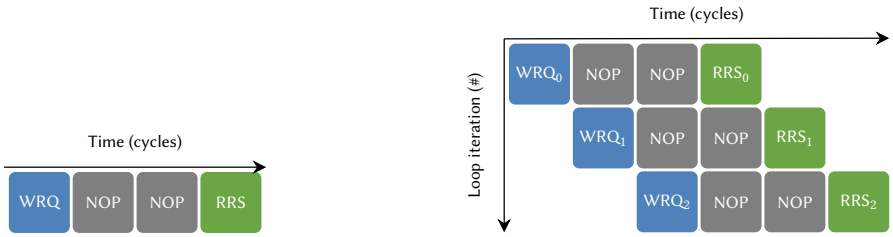
Scheduling *WRQ* and *RRS* to the same cycle leads to deadlock, because the *RRS* blocks while attempting to read from the empty response FIFO (the response would be available only after the *Slave*'s latency has elapsed), stalling the entire stage, including the *WRQ*. Thus, *RRS* waits for the response to a request which has never been written. Therefore, *WRQ* and *RRS* must be forced to separate pipeline stages by explicitly declaring a dependency between them, with a distance of at least one clock cycle. *Vitis HLS* allows declaring the dependency via the `write_dep` and `read_dep` FIFO access functions and setting the distance of one cycle with the `reg` function.

While this ensures functionality, it fails to deliver high throughput when both the *Master* and *Slave* are pipelined. The HLS scheduler, unaware of the *Slave*'s latency, schedules *RRS* immediately after *WRQ*, as shown in Fig. 3.4a. If the *Slave*'s pipeline has an II of 1 cycle and depth of D cycles, RRS_0 stalls because the *Slave* requires D cycles to produce a response. This delays all subsequent requests, as WRQ_1 can only execute after RRS_0 completes, once the response from the *Slave* is received (Fig. 3.4b). As a result, the *Slave* does not receive requests in consecutive cycles, leading to a throughput of $1/D$, as if it were not pipelined.



(a) Static schedule (b) Dynamic behavior

Figure 3.4: Stalling cyclic dataflow schedule of *Master*.



(a) Static schedule (b) Dynamic behavior

Figure 3.5: Non-stalling cyclic dataflow schedule of *Master*.

By setting the dependency distance between *WRQ* and *RRS* to D cycles, the scheduler inserts $D-1$ pipeline stages between them (Fig. 3.5a). This enables the *Master* to issue one request and receive one response per cycle (Fig. 3.5b), avoiding stalls.

3.2.2 Cache-based buffering task architecture

The cache buffering task central to the CCC architecture, illustrated in Fig. 3.6, features a level 2 (L2) cache implemented as a dataflow task that communicates with the compute task using the cyclic dataflow protocol. L2 caches expose a configurable number of ports (i.e., request and response FIFOs pairs) that are served in a round-robin order. Each port optionally includes a private L1 cache, which enables concurrent memory accesses. DRAM accesses are organized in cache lines, which are blocks of sequential elements aligned to the line size, enabling automatic port widening and burst inference.

Similarly to the cache by Ma *et al.* [31], the L1 cache is inlined within the compute logic. However, unlike Ma *et al.*'s design, the compute task's pipeline II is maintained because L1 cache misses interact with the L2 cache rather than directly accessing the external DRAM.

High-throughput cyclic dataflow graphs

The CCC architecture requires throughput comparable to the LCS architecture, where the $(i+1)$-th data load overlaps with the i-th computation. Due to the cycle in the DFG, as shown in Fig. 3.1b, the cache task adheres to the cyclic dataflow protocol.

The distance between the cache request writing and the cache response reading separates

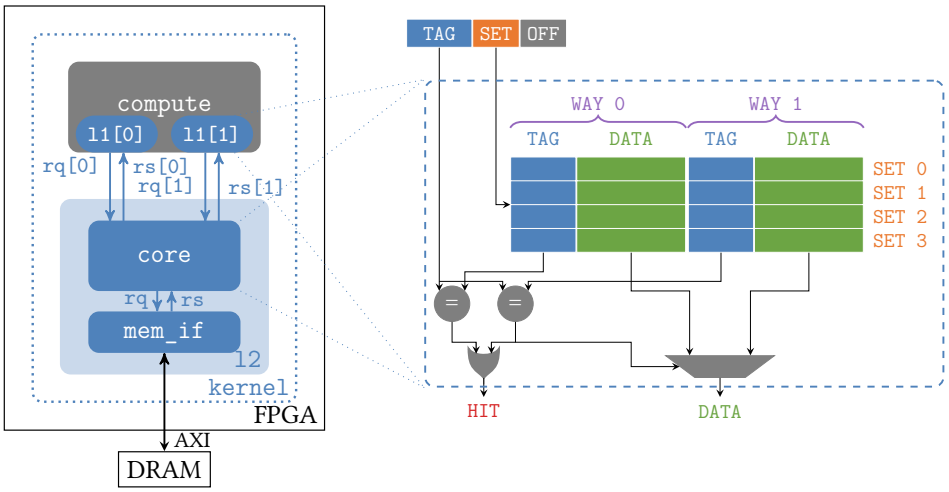


Figure 3.6: The architecture of the cache-based buffering task in its deployment environment.

the data loading from its processing, effectively overlapping memory accesses with computation. The `core` pipeline latency ranges from 5 to 7 cycles, depending on the cache parameters and the clock constraint. Therefore, a request-response distance of 6 cycles typically provides best results. In case of multi-port configurations, a shorter request-response distance (e.g., 3 cycles) is more convenient, since the `core` pipeline concurrently receives requests from the different ports. For read-write (RW) caches with data dependencies, low request-response distances (e.g., 2 cycles) limit the dependencies distances, improving the minimum pipeline II.

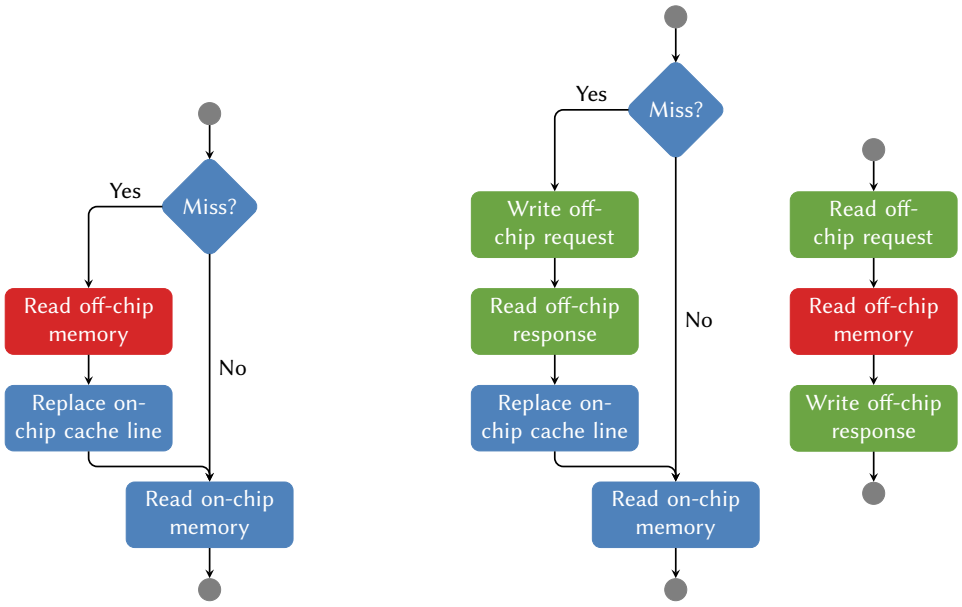
Static scheduling of irregular CFGs

State-of-the-art (SOTA) HLS tools generate statically-scheduled pipelines whose II is limited by the worst path in the CFG. Consequently, the maximum achievable throughput is constrained by the slowest branch, even if that branch is rarely executed.

Josipović *et al.* [33] propose the dynamic scheduling paradigm to improve the average runtime II and throughput for designs with unbalanced CFGs, which shifts the data dependencies management from the compiler to the generated hardware. However, Cheng *et al.* [34] show that static scheduling provides superior overall QoR, particularly for maximum clock frequency and area efficiency, whereas dynamic scheduling is advantageous in terms of II only, if the static scheduler fails achieving low-II schedules due to irregular CFGs. Therefore, this dissertation deems more convenient to build upon the static scheduling and achieve high-throughput scheduling via the alternative proposed methodology.

Specifically, to mitigate the impact of long-latency branches, DaCH applies a “divide et impera” methodology, extracting these long-latency branches (e.g., the miss branch of a cache model) into separate tasks. This allows static scheduling for the best case while stalling when executing long-latency branches, to ensure functionality, by reading from empty feedback FIFO.

The CFG of a cache is irregular, since the miss branch introduces a long-distance RAW dependency on the on-chip memory, as depicted in Fig. 3.7a. The “divide et impera” approach hides the long latency of the miss logic (i.e., mainly including the AXI adapter to interface with the off-chip memory) by extracting it into a separate `mem_if` task (Fig. 3.7b), implemented as



(a) Naive CFG

(b) Static-HLS-optimized CFGs

Figure 3.7: The HLS description of a naive read-only cache. In case of miss, the cache replaces the corresponding line in the on-chip memory with the data from the off-chip memory. Since the main memory port width is generally smaller than the cache line, the line replacement requires several clock cycles, limiting the pipeline throughput. With a static schedule, the throughput penalty happens in case of hit too, cancelling any benefit from the cache.

a *Slave* in the *Cyclic dataflow protocol Slave* task. As a result, the `core` task's pipeline, which contains the hit logic and acts as the *Master* of the `mem_if` task, is statically-scheduled for the best case (i.e., hit) with an II of one cycle, allowing it to serve one hit request per cycle. In case of miss, the `core` dynamically stalls, waiting for the response from the `mem_if` task (similarly to the schedule in Fig. 3.4b).

Masking memory latency

Writable caches contain a RAW data dependence stemming from newly written words hitting in the following read access. The one-cycle latency of on-chip memories (e.g., BRAMs or URAMs) makes the cycle in the DDG critical, preventing from optimally pipelining the *Core* task with an II of one cycle.

A small auxiliary *RAW cache* minimizes the II to 1 cycle. The *RAW cache* is a two-line fully associative cache, bound to registers, that can be read in the cycle following a write operation. Store operations write both the memory line and the *RAW cache* line, following a FIFO replacement policy. Load operations read the line from the *RAW cache* when hitting or from the underlying memory when missing. Accessing the data memory through the *RAW cache* eliminates the 2-cycle dependency created by the memory latency. This is because, if a memory line is written, it will not be read within the next two pipeline iterations. During these iterations, any subsequent accesses would hit the *RAW cache* and bypass the memory.

```
+#include "cache.h"
+
+typedef cache<DATA_TYPE, RD_ENABLED, WR_ENABLED,
+        MAIN_SIZE, N_SETS, N_WAYS, N_WORDS_PER_LINE, LRU,
+        SWAP_TAG_SET, LATENCY> cache_type;

 template <typename T>
 void compute(T &a) {
         for (auto i = 0; i < (N - 1); i++) {
 #pragma HLS pipeline
                 a[i] = a[i + 1];
         }
 }

 extern "C" void top(DATA_TYPE *a) {
 #pragma HLS interface m_axi port=a bundle=gmem0
+#pragma HLS dataflow
+        cache_type a_cache(a);
-        compute(a);
+        cache_wrapper(compute<cache_type>, a_cache);
 }
```

Figure 3.8: Source code modifications for accelerating the `compute` task with the proposed cache.

(a) Standard mapping (b) Swapped mapping

Figure 3.9: Configurable address bit mapping.

This approach allows to effectively mask the memory latency and enables the *Core* task to maintain an II of 1 cycle, enhancing overall throughput.

User-facing APIs

To use DaCH, designers simply configure the cache parameters via the class template arguments, instantiate the cache, and call the `compute` function through the `cache_wrapper` function within a dataflow region, as shown by Fig. 3.8. Complete examples can be found in the open source repository.

The cache is configurable for type and number of words per line, number of sets and ways, replacement policy (least recently used (LRU) or FIFO), memory type (BRAM, URAM, LUT memory) and address bit mapping (standard, Fig. 3.9a, or swapped, Fig. 3.9b). The swapped mapping is useful for specific cases, such as column-wise matrix access, discussed in Section 3.3.1. Read-only (RO) caches also provide a customizable arbitrary number of ports, each with an optional private L1 cache, configurable in terms of number of words per line, the number of sets and ways. The number of words per line, sets, and ways can be any power of two,

including one.

The cache application programming interfaces (APIs) manage communication between the compute and cache tasks, hiding the cyclic dataflow protocol to end users. The `get_line` function receives as input the address to read from cache and returns the line to which the address belongs. In particular, if the address hits the L1 cache, the line is read from the L1 cache. Otherwise, the request is issued to the L2 cache. The `get` function calls the `get_line` function and returns the requested word only. The `set` function marks the L1 cache line as dirty, if it hits, according to the write-through policy. Additionally, it forwards the write request to the L2 cache.

The overloaded `operator[]` simplifies usage by implicitly calling `get` and `set` functions. For example, the expression `a[i]=a[i+1]`, from Fig. 3.8, translates to `a.set(a.get(i+1),i)` automatically. The `get` and `get_line` functions also provide the optional port identifier, to specify the cache port to access in case of a multi-port configuration. In this case, the `operator[]` cannot be used.

In addition, DaCH exposes performance-profiling functions that return statistics such as the number of accesses and the number of hits for each cache port. It is worth noting that they are available in software simulation, allowing for quick performance estimation and cache parameters DSE, without needing of expensive RTL simulations or hardware deployment.

3.3 Evaluation

The CCC architecture is benchmarked on commonly-used, memory-intensive kernels (either bandwidth or latency-limited), comparing the CCC implementations with the baseline, automatically generated by the HLS tool, and with the manually-optimized LCS implementation, if available. Additionally, Section 3.3.2 showcases the advantages brought by DaCH to a SOTA FPGA accelerator, while Section 3.3.3 compares DaCH with the newly-introduced cache provided by AMD Vitis HLS.

Resource utilization is obtained from post-implementation reports, hit-rates from software simulation, and performance and power are measured using hardware timers and current sensors provided by the board, accessed through the PYNQ APIs [35].

Cache parameters, such as the size and number of lines, are manually selected based on the array access patterns. However, numerous methods exist to automate the selection of these parameters, as demonstrated by previous work, such as that analyzed by Upadhyay and Sudarshan [36]. Integration of such approaches with the DaCH cache is deferred to future work.

3.3.1 Memory-bound benchmarks

This section examines three memory-bound benchmarks, namely matrix-matrix multiplication (MMM), 2D convolution, and bitonic sorting. The MMM and 2D convolution algorithms are bandwidth-limited due to the large amounts of data consumed and produced during computation, whereas bitonic sorting is latency-bound, because of data dependencies on off-chip memory.

The baseline implementation for each algorithm consists of a single-task kernel with nested loops, where the innermost loop is pipelined with an II of one cycle. To evaluate the CCC architecture, DaCH is included in the baseline kernels with minimal source code changes, akin

(a) Standard address bit map

(b) Swapped address bit map

Figure 3.10: The sequence of B addresses accessed during the first 8 iterations of the matrix-matrix multiplication algorithm, where $B \in \mathbb{Z}^{4\times8}$ has a 4-set direct-mapped cache. With the standard address bit mapping (a), the cache always misses since the same set is sequentially accessed by addresses with different tags, evicting the related line. Contrarily, the swapped address bit mapping (b) reads different sets with the same tag, achieving high hit rates.

to those in Fig. 3.8. The single-port CCC (C-S) versions use single-port caches to increase effective memory bandwidth, by gathering off-chip memory accesses in large, contiguous cache lines and avoiding redundant off-chip accesses, and to reduce the average memory latency, and thus data dependencies distance. The multi-port CCC (C-M) versions, which are configured to expose multiple ports, enable increased the computation parallelism through loop tiling or unrolling, with the factor corresponding to the number of ports. Finally, the manually-optimized LCS implementations serve as the best-performance references for comparison.

The benchmarks are deployed on the Avnet Ultra96 [37] board, featuring a Xilinx Zynq UltraScale+ embedded-level FPGA and double data rate 4 (DDR4) off-chip memory. HLS [8] and implementation [38] are performed with the AMD tools at version 2021.2.

MMM computes $C = A \times B$, with $A \in \mathbb{Z}_{32}^{N\times M}$, $B \in \mathbb{Z}_{32}^{M\times P}$, and $C \in \mathbb{Z}_{32}^{N\times P}$, where $\mathbb{Z}_{32}$ are 32-bit integers, $N = 1024$, $M = 128$, and $P = 1024$. The access pattern for the matrix B is the most challenging, as it is accessed by columns, but it is stored in row-major order, severely limiting the spatial locality. Achieving a non-zero cache hit rate requires either a costly M-way fully associative cache or a more efficient M-set direct-mapped cache with swapped address bit mapping (Fig. 3.9b). Using the standard address bit mapping in an M-set direct-mapped cache leads to frequent cache evictions (and misses), as consecutive accesses map to the same set but with different tags (Fig. 3.10a). In contrast, the swapped bit mapping ensures that consecutive reads access distinct sets with identical tags (Fig. 3.10b), yielding a high hit rate. This illustrates how cache-based buffers achieve high hit rates with cost-effective hardware by leveraging highly customized cache configurations, enabled by array-specificity and deep cache parametrization.

In the C-S version, matrix A is assigned with a direct-mapped, 8-set, 16-word-line L2 cache to buffer an entire matrix row, reused 1024 times, while matrix B uses a direct-mapped, 128-set, 32-word-line L2 cache with swapped tag and set bits, reaching an overall hit rate of 98.4 %. Since matrix C is written once in sequential order, caching is not needed. In the C-M version,

the innermost loop is tiled by a factor of 16, requiring the A cache to be extended to a 16-port L2 cache. Each port is equipped with a direct-mapped, 8-set, 16-word-line L1 cache, each buffering a distinct A row. The cache for matrix B remains the same as in the C-S version. The overall hit rate is 99.8 %.

The 2D Convolution benchmark computes $B = A * K$, with $A \in \mathbb{N}_8^{N \times M}$, $K \in \mathbb{N}_8^{P \times Q}$, and $B \in \mathbb{N}_8^{N \times M}$, where $N = 1080$, $M = 1920$, and $P = Q = 15$. A is accessed by windows of size $P \times Q$ and stride one. The cache associated with A is configured with P ways to buffer all lines within a window, effectively functioning as a line buffer without requiring source code modifications. The cache is configured with two sets to prevent eviction of cache windows that are not aligned with the line size. Each cache line holds 64 words, enabling the prefetching of four 15-elements windows. The single-port C-S version achieves a hit rate of 99.9 %. The multi-port C-M version extends the same C-S configuration to 15 ports, supporting factor-15 loop tiling with a hit rate of 99.9 %. Neither matrix K nor B require caching, as K is small enough to be fully buffered in registers, while B is written sequentially only once. The LCS design is the open-source implementation by AMD [5].

The bitonic sorting benchmark sorts an array of 1 million 32-bit integers. Each iteration of the inner loop reads the i-th and $(i + step)$-th elements, where *step* varies exponentially throughout execution, and swaps them if they are not sorted. The swap operation introduces RAW data dependencies, which limit pipeline performance. The baseline test case, which accesses directly the off-chip memory, requires a high II of 16 cycles to guarantee the dependency on the high-latency AXI interface, even when the HLS tool is configured to assume 0-latency off-chip memory. In contrast, the proposed cache enables scheduling for the best case (i.e., hit) with an II of 6 cycles, and dynamically stalls only in case of cache miss. The C-S version employs a 64-word-line L2 cache, with two-way associativity, configured to prevent interleaved accesses to i and $(i + step)$ from overwriting the corresponding cache lines. It achieves a hit rate of 99.2 %. Due to the irregular RW access pattern and the resulting data dependencies, efficient on-chip data buffering is challenging, hindering the design of an effective LCS architecture. Moreover, the algorithm cannot benefit from the higher memory bandwidth of a C-M version.

Experimental results presented in Table 3.1 demonstrate that C-S designs effectively mitigate the memory bound. The increased effective memory bandwidth and reduced memory latency contribute to a reduction in execution time by an order of magnitude compared to the baseline. C-M designs exhibit lower clock frequencies, primarily due to routing congestion that arises from the convergence of signals from each port to a single shared AXI interface. Despite this limitation, the larger bandwidth offered by C-M designs facilitates an increase in computational parallelism through loop tiling, achieving higher throughput without modifications to the algorithm itself, outweighing the reduced frequency. This enables the C-M designs to attain performance on the same order of magnitude of the LCS designs.

C-S designs use more memory LUTs and BRAMs than the baseline for the cache data memories and more logic LUTs and FFs for the cache logic and its pipeline registers. C-M designs consume more resources than C-S designs not only because they include more expensive multi-port caches, but also because they increase the parallelism of the kernel.

LCS designs are Pareto-optimal in the PPA space, albeit at a high engineering cost due to the need for algorithm restructuring and the development of ad-hoc buffering solutions. In contrast, the CCC designs necessitate minimal design effort to achieve comparable performance, although at the expense of some area overhead.

3.3.2 Real-world application

Bosio *et al.* [39] employ DaCH to optimize memory management for a hardware accelerator designed for solving sub-graph isomorphism, described in HLS and targeting an AMD Kria KV260. Given that input graphs are typically larger than the available on-chip memory capacity, these graphs are stored to off-chip memories. Designing an effective LCS architecture is unfeasible due to the irregular memory access patterns that depend on the specific input graphs, which remain unknown at design time. As a result, cache-based buffering tasks are used to dynamically exploit the data locality that emerges during execution.

The accelerator incorporates two DaCH caches: a single-port cache with 512 sets, each containing 16 128-bit words, and a two-port cache with 4096 sets, each containing 8 128-bit words. Figure 3.11 illustrates the execution time improvement provided by the DaCH caches compared to the same design that accesses off-chip memory directly. The performance advantage varies significantly depending on the database; however, in general, DaCH enables substantial speedups, up to 7×.

3.3.3 Comparison with Vitis HLS cache

AMD Vitis HLS 2023.2, released after the introduction of DaCH, adds caching capabilities to the master advanced extensible interface (MAXI) adapter, which interfaces with off-chip memory. Table 3.2 summarizes the main functional differences between the DaCH and the MAXI caches. The MAXI cache is fully integrated into the HLS tool and can be accessed through dedicated HLS pragma directives, whereas DaCH is an external library. However, the capabilities of the MAXI cache are limited; it supports only RO memories, exposes a single port, is direct-mapped only, and lacks advanced features such as address bits swapping. Furthermore, profiling hit

Table 3.1: Power, performance, and area of the memory-bound benchmarks. The C-S and C-M versions are the cache-optimized versions, single and multi-port, respectively. The Base is the design implemented by default by the HLS tool, while the load-compute-store (LCS) is manually optimized.

Bench.	Ver.	Exec. time (s)	Clock freq. (MHz)	LUT Logic (%)	LUT Mem. (%)	FF (%)	BRAM (%)	DSP (%)	Power FPGA (mW)	Power DRAM (mW)
MMM	Base	32.70	300	4	3	4	1	1	188	186
	C-S	2.40	300	11	5	14	9	1	375	156
	C-M	0.29	200	31	54	41	9	13	625	219
	LCS	0.14	250	17	36	41	4	13	750	344
Conv.	Base	24.10	300	5	3	4	2	0	219	327
	C-S	2.18	300	8	3	9	3	0	313	109
	C-M	0.70	250	44	31	42	3	4	813	125
	LCS	0.12	300	7	6	11	6	63	406	63
Bitonic	Base	66.59	300	3	1	2	0	0	118	176
	C-S	7.94	250	51	8	41	2	0	816	148

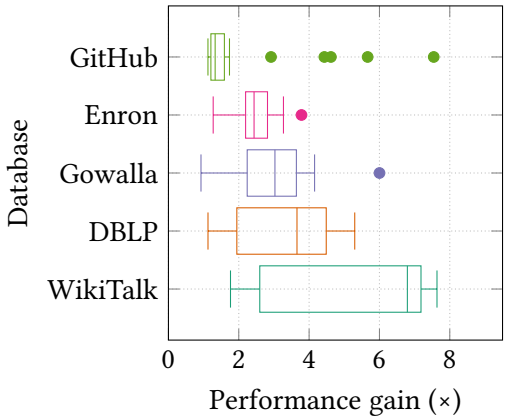


Figure 3.11: The execution time speedup of the sub-graph isomorphism accelerator provided by the cache with respect to directly accessing the off-chip memory.

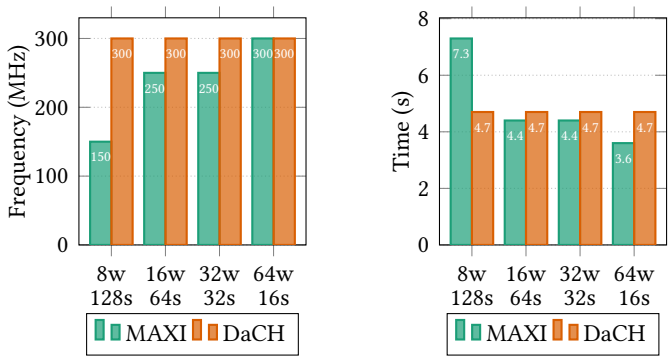
rates necessitates expensive RTL simulations, making the DSE of cache parameters orders of magnitude more expensive than the software-based approach of DaCH.

The limitations of the MAXI cache prevent from effectively using it in the benchmarks discussed in Section 3.3.1. For instance, the MMM algorithm requires set-associativity or address bit swapping to achieve non-zero hit rates, while the bitonic sorting benchmark demands RW caches. As a result, the benchmark under consideration is a simplified kernel that reads an array according to the access pattern of the A matrix in the operation $C = A \times B$, with $A \in \mathbb{Z}_{32}^{1024\times1024}$, without computing the actual matrix multiplication. Since the kernel reads each row of matrix A 1024 times, the cache is configured to buffer an entire line, which is achieved when the product between the number of cache lines and the line size (in words) is 1024. The line size is varied from 8 to 64 words.

Figure 3.12 summarizes the performance of the tested configurations, generated with AMD Vitis HLS 2024.1 targeting the Avnet Ultra96v2 board. The maximum clock frequency of the MAXI cache does not scale effectively with the number of sets, whereas DaCH consistently reaches the maximum allowed by the physical interface. However, when both operate at the same clock frequency (specifically, in the 64-words, 16-sets configuration), the MAXI cache performs 23 % better than DaCH, thanks to its tight integration within the adapter.

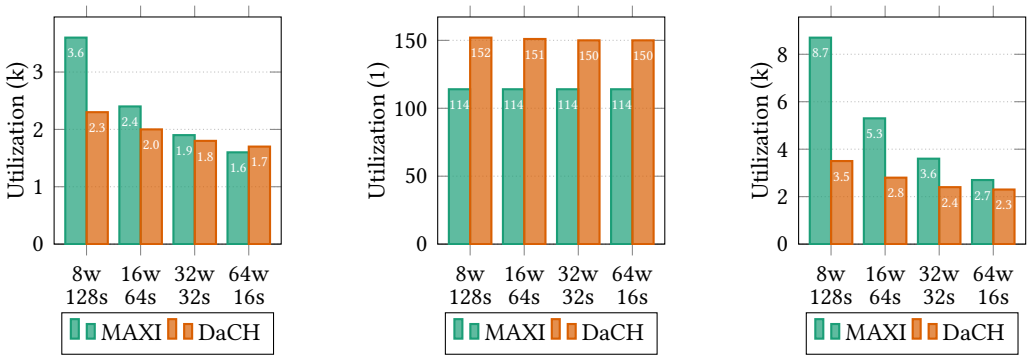
Table 3.2: Features comparison between DaCH cache and the Vitis HLS 2024.1 MAXI cache.

	MAXI	**DaCH**
Access mode	Read-only	Read-write
Multi-level	No	Yes
Multi-port	No	Yes
Set-associativity	No	Yes
Address mapping	Fixed	Customizable
Profiling	RTL	Software
API	Pragma	Library



(a) Clock (b) Execution

Figure 3.12: Performance of the proposed DaCH cache against the MAXI cache from Vitis HLS. *W*w and *S*s represent configurations with *S* sets with *W* words each.



(a) Logic LUT (b) Memory LUT (c) Flip-flop

Figure 3.13: Area of the proposed DaCH cache against the MAXI cache from Vitis HLS. *W*w and *S*s represent configurations with *S* sets with *W* words each.

Results presented in Fig. 3.13 indicate that DaCH consumes less logic LUTs and FFs than the MAXI cache, especially when scaling up the number of sets. On the other hand, MAXI cache requires fewer memory LUTs since it uses the buffers of the MAXI adapter as cache memory, whereas DaCH uses a standard MAXI adapter with its own dedicated buffers. Moreover, the cache user task communicates with the cache directly via the MAXI interface, while DaCH allocates additional dedicated FIFOs.

These results suggest that the MAXI cache is a promising solution, though it still needs further refinements, especially in terms of functionality (e.g., supporting associativity and RW accesses is crucial for applying it to real-world algorithms) and clock frequency optimization.

3.3.4 Qualitative comparison with dynamically-scheduled HLS

Statically-scheduled pipelines of irregular CFGs containing data dependencies on off-chip memory like Fig. 3.14, reach low throughput as the high latency of the memory operations on the critical cycle impose high minimum IIs. Accessing the off-chip memory through DaCH allows

```
for (int i = 1; i < N; i++) {
#pragma HLS pipeline
  if (mem[i - 1] > 0)
    mem[i] = 1;
}
```

Figure 3.14: Example of irregular control flow graph with off-chip memory dependency. DaCH allows reducing the minimum initiation interval (II) to 2 clock cycles in statically-scheduled HLS, whereas dynamically-scheduled designs reach 1 clock cycle only if the condition always holds false.

to statically schedule for the best-case II (i.e., cache hit) and dynamically stall only when necessary for guaranteeing correct functionality. The example of Fig. 3.14 achieves an II of 2 clock cycles, with a request-response distance of 1 cycle. On the other hand, dynamically-scheduled designs potentially achieve higher throughput, with an average II approaching 1 clock cycle if the condition holds false for most loop iterations.

However, DaCH caches are beneficial not only for hiding off-chip memory latency from the static scheduler, enabling better IIs, but also for reducing off-chip memory traffic by leveraging the spatio-temporal data locality that algorithms naturally expose at runtime. As a result, dynamically scheduled designs can also benefit from data caching. Since DaCH is modeled in `C/C++`, it is potentially easily portable to any HLS tool supporting `C/C++` input, including dynamically-scheduled HLS tools like Dynamatic [33].

3.4 Discussion

This dissertation introduces the CCC architecture and the DaCH open-source HLS library, designed to semi-automatically generate LCS-like architectures with minimal design effort, via highly-configurable cache-based buffering tasks.

Experimental results demonstrate that the CCC architecture enables new design points that outperform those generated by the HLS tool and require significantly less engineering effort compared to the manually-optimized LCS designs. Instead of modifying the algorithm extensively to manage on-chip data buffering, designers only need to conduct a DSE of the cache configurations. For algorithms with irregular or data-dependent memory access patterns, caching remains the only viable method for improving memory access performance.

The effectiveness of this approach is testified by the integration of DaCH into a SOTA graph processing accelerator [39], improving its performance and energy efficiency, as well as into an OpenMP-to-FPGA compiler [40]. Additionally, recent advancements, successive to DaCH, in both industrial [8] and academic [41] HLS tools show a trend towards incorporating data caching techniques, as these tools begin to feature cache-optimized off-chip memory interfaces.

To achieve the full abstraction of memory management, DaCH still requires an automated mechanism to select the cache parameters, given a memory port to accelerate and a performance or resource constraint. Whilst SOTA cache parameters optimization methods [36] are viable solutions to this problem, their integration in DaCH is left to future work.

Chapter 4

Automatic DSP packing

In the context of FPGAs, DSPs are a crucial resource that typically requires a significant optimization effort for its efficient utilization, especially in low-precision applications. This dissertation proposes SILVIA, an open-source framework based on LLVM transformation passes that automatically identifies superword-level parallelism within an HLS design and exploits it by packing multiple operations, such as additions, multiplications, and MADs, into a single DSP. SILVIA is integrated in the flow of the commercial AMD Vitis HLS tool and proves its effectiveness by packing multiple operations on the DSPs without any manual source-code modifications on several diverse SOTA HLS designs such as CNNs and basic linear algebra (BLAS) accelerators, reducing the DSP utilization for additions by 70 % and for multiplications and MADs by 50 % on average.

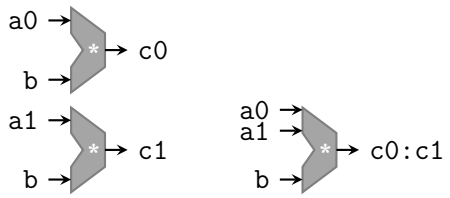
4.1 Overview and related work

In FPGA platforms, low-precision data formats (e.g., 8-bit integers) are crucial to achieve high computational intensity within the constraints of scarce on-chip memory and limited off-chip memory bandwidth, particularly for SOTA machine learning applications, where 8-bit integers are the most popular quantized data format for inference acceleration at the edge. However, these data formats underutilize the large precision of the FPGAs' DSPs, unless designers explicitly pack the inputs of the DSPs to efficiently utilize them. For instance, the AMD/Xilinx UltraScale DSP [12] slice supports the SIMD operating mode for additions. Moreover, ingenious DSP packing approaches [13]–[18] enable other SIMD-like operations, such as multiple MADs with shared operands. Section 2.2.2 discusses the aforementioned techniques.

Several SOTA FPGA designs optimized for performance using DSP packing [42]–[45] prove the effectiveness of this technique. However, current HLS compilers and hardware description language (HDL) synthesis tools lack automatic instruction vectorization capabilities exploiting DSP packing. Therefore, those designs required the manual identification of the parallelism and the explicit fine-tuning of the source code for taking advantage of the operation-packing capabilities of DSPs, relying on either RTL implementations [42], [45], or mimicking the low-level RTL expressiveness in HLS via bit operations to explicitly map the inputs and outputs to the DSP ports [43], [44], disrupting the HLS abstractions.

```
for (int i = 0; i < N; i++) {
#pragma HLS unroll factor=2
  c[i] = a[i] * b;
}
```

(a) HLS design source code



(b) Original datapath (c) SILVIA datapath

Figure 4.1: Given a loop computing two multiplications in parallel (a), the standard HLS flow generates the corresponding datapath (b) with two DSPs, one per multiplication. The SILVIA flow automatically halves the DSP utilization by packing the two multiplications to a single DSP (c) without any manual source code modification.

The automatic vectorization of loops and basic blocks (BBs) is a well-established optimization typically implemented in software compilers targeting CPUs [46], [47]. However, since software compilers target inherently different hardware than HLS, they focus on issues not relevant for FPGA designs, such as the amortization of the overhead for moving data from scalar to vector register files and vice versa. Moreover, they only support basic SIMD instructions operating on independent data, missing more complex patterns such as two MADs sharing an operand [14].

A multitude of studies [48]–[50] automate code transformations for improving QoR of HLS designs. However, to the best of the authors' knowledge, no previous work automatically identifies the compatible operations present in HLS designs and packs them to DSPs to improve the computational intensity.

SILVIA (automated Superword-level parallelIsm exploitation via HLS-specific LLVM passes for compute-Intensive FPGA Accelerators) extends the SOTA HLS flow with additional compiler transformation passes that automatically identify the compatible operations naturally present in HLS designs (e.g., exposed by loop unrolling) and map them into packed DSP operations, without any modification to the input C++ code.

For instance, the loop defined in Fig. 4.1a computes two 8-bit multiplications with a shared operand in parallel. The SOTA HLS flow allocates two DSPs, one per multiplication (Fig. 4.1b). On the other hand, SILVIA automatically packs the two multiplications to a single DSP [14] (Fig. 4.1c) by analyzing and transforming the IR of the design.

The typical HLS flow consists of the FE step, that translates the high-level design description into an optimized IR (e.g., from C/C++ to LLVM IR) and the BE step, that generates the hardware description of the functionality specified by the IR. SILVIA is executed between the FE and the BE, as shown in Fig. 4.2.

SILVIA is based on LLVM [51], a widely used open-source compiler infrastructure extensible via optimization passes that analyze and transform the programs expressed in an IR form. SILVIA is fully integrated in the flow of the commercial AMD Vitis HLS 2023.2 tool; moreover, it could support any LLVM-based FPGA HLS tools (e.g., Dynamatic [33], LegUp [52]) with minor adaptations.

The main contributions of this chapter are:

- The SILVIA framework, an LLVM transformation pass integrated in the AMD Vitis HLS flow that implements the generic functionality for packing multiple scalar operations

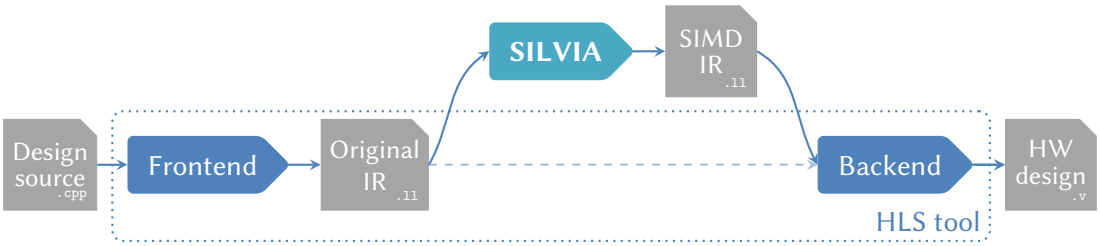


Figure 4.2: The modified HLS workflow with SILVIA. SILVIA optimizes the original LLVM intermediate representation generated by the frontend for DSP-packed operations and provides it as input to the backend.

within HLS source code to single DSPs on FPGAs and is extensible to support different operations. SILVIA is open-source at `github.com/brigio345/SILVIA`.

- The `SILVIAAdd` pass for binding two 24-bit or four 12-bit additions or subtractions to a DSP and the `SILVIAMuladd` for binding two 8-bit MADs or four 4-bit multiplications with a shared operand to a DSP.
- The validation of SILVIA on several diverse designs, showing that it saves 60 % addition DSPs and 45 % multiplication and MAD DSP on average with no impact on performance and no modification to the source code, compared to the original Vitis HLS synthesis flow. Moreover, it achieves DSP utilization on par with manually-optimized SOTA CNN accelerators.

4.2 Methodology

The SILVIA LLVM optimization pass is executed in the middle of the typical LLVM-based HLS flow (e.g., Vitis HLS), by optimizing for DSP-packed operations the FE-generated LLVM [51] IR before passing it to the BE, as shown in Fig. 4.2. With this approach, SILVIA processes an IR that is already optimized by the FE (e.g., the dead code is eliminated; the width of the instructions is minimized) and its high abstraction level allows for using the advanced analysis and transformation facilities provided by the LLVM APIs, before it is lowered to hardware description by the BE.

SILVIA is integrated in the Vitis HLS flow. Therefore, it complies with the LLVM 3.1 [51] APIs for compatibility with the FE-generated IR. The SILVIA flow does not exploit the Vitis HLS capability of inserting user-defined passes within the FE itself [53], because this approach would execute SILVIA early in the FE pipeline, preventing it from taking advantage of the FE optimizations, such as the width minimization of the instructions.

Algorithm 1 summarizes the main steps of the SILVIA optimization pass. SILVIA optimizes one BB at a time, similarly to the superword-level vectorizers targeting CPUs [47]. It collects the *candidate* instructions amenable for vectorization, groups them into *tuples* of compatible candidates, and finally replaces each tuple with an optimized *packed operation*. A concrete example is the automatic binding of four 12-bit additions to a single DSP configured in the `four12` mode. SILVIA searches for 12-bit `add` instruction candidates, groups them into tuples of four elements, and binds each tuple to a single DSP.

Algorithm 1 SILVIA's main optimization routine.

Require: BB = basic block belonging to an HLS design.
Ensure: BB^* = basic block functionally equivalent to BB and optimized for DSP-packed operations.

```
C ← getCandidates(BB)
BB* ← BB                                   ▷ Maximize the space for valid tuples.
for c ∈ C do
    BB* ← moveUsesALAP(c, BB*)
end for
                                           ▷ Group the candidates in valid tuples.
T ← getTuples(C)
                                           ▷ Pack the valid tuples.
for T ∈ T do
    P ← packTuple(T)
    BB* ← replaceTuple(T, P, BB*)
end for
```

The `SILVIA` class extends the LLVM `BasicBlockPass` class and implements the structure of Algorithm 1. The structure of the `SILVIA` class allows supporting different DSP-packed operations through derived classes of `SILVIA`, which simply override some virtual functions of `SILVIA` and exploit the rest of the existing framework that is common to every packed operation. Specifically, the derived classes must override the `getCandidates` and `packTuple` functions (highlighted in blue in Algorithm 1), and the `canPack` and `isTupleFull` functions, used internally by `getTuples`.

The `SILVIA` class is currently extended by two examples of DSP packing specializations, discussed in Section 2.2.2:

- `SILVIAAdd`: four 12-bit additions (or subtractions) or two 24-bit additions (or subtractions).
- `SILVIAMuladd`: two 8-bit MADs (or multiplications) or four 4-bit multiplications.

4.2.1 Candidate identification

The `getCandidates` function identifies the candidates as the initial step of the SILVIA flow. Given a BB, `getCandidates` returns the set of instructions (or patterns of instructions) compatible with the packing optimization.

For the `SILVIAAdd` pass, `getCandidates` returns the addition instructions whose operands size in bits is within the allowed range (up to 12 or 24 bits). The `getCandidate` of `SILVIAMuladd`, instead, searches for trees of addition instructions whose leaves are multiplication instructions between operands of 8-bit or less, for the factor-2 packing, or 4-bit or less, for the factor-4 packing. It is worth noting that the `SILVIAMuladd` pass also supports multiplication-only packing, since a degenerate tree composed of a single multiplication is a valid candidate too.

4.2.2 Tuple generation

Given a set of candidates, the `getTuples` function combines them into tuples

1. whose candidates do not depend on each other,
2. with an available insertion point between the first use of each candidate and after the last definition of each candidate's operands,
3. that satisfy the constraints of a specific DSP-packed operation.

Pack insertion point

SILVIA replaces a tuple with a packed operation by inserting a call to a function that implements the packed operation (further details in Section 4.2.3). The packed function call must be placed after the definition of every tuple's operand and before every use of the tuple's results to produce valid LLVM code. However, when compiling `C` code containing unrolled loops (a typical source of parallelism and potential vectorization), the LLVM compiler inserts multiple copies of the loop body in sequence.

For instance, the previously discussed design example defined in Fig. 4.1a, which computes two parallel multiplications via loop unrolling, is compiled to the LLVM IR in Fig. 4.3a. The first use of the first multiplication (i.e., the `store` of `c0`) comes before the last definition of the operands of the second multiplication (i.e., the `load` of `a1`). In this scenario, there is no room for placing the packed multiplication, since the last definition – first use intervals of the two multiplications (highlighted by the arrows in Fig. 4.3a) do not intersect.

To maximize the room for DSP-packed calls, the `moveUsesALAP` function moves the candidate's uses as late as possible (ALAP) within a BB. SILVIA preserves the data dependencies by analyzing the definition–uses chains and exploiting the memory aliasing analysis LLVM infrastructure. Moreover, it conservatively assumes that function calls may alias with other function calls or memory operations, to guarantee the functionality without expensive inter-procedural analysis outside the BB scope.

When building the tuples of candidates to pack, SILVIA checks if there exists a *packed insertion point*. The function tests if the last definition – first use interval of a candidate intersects with the interval of a tuple (i.e., they intersect when the last definition of the candidate comes before the first use of the tuple and vice versa). It is notable that the existence of a valid insertion point implies the tuple is free from interdependencies between its candidates.

Operation-specific tuple validity

The `canPack` virtual function exposed by SILVIA is an additional hook for filtering the candidates to be added to a tuple, based on the requirements of the specific packed operation.

The `SILVIAMuladd` class overrides the `canPack` function to check whether a candidate shares one of its operand with the other candidates in the tuple, in compliance with Eqs. (2.5) and (2.7). The `SILVIAAdd`'s `canPack` function does not perform any further check, since a SIMD DSP can compute any tuple of independent additions.

```
%a0 = load i8, i8* %a0_ptr
%c0 = mul i8 %a0, %b
store i8 %c0, i8* %c0_ptr
%a1 = load i8, i8* %a1_ptr
%c1 = mul i8 %a1, %b
store i8 %c1, i8* %c1_ptr
```

(a) Source code

```
%a0 = load i8, i8* %a0_ptr
%a1 = load i8, i8* %a1_ptr
%c0 = mul i8 %a0, %b
%c1 = mul i8 %a1, %b
store i8 %c0, i8* %c0_ptr
store i8 %c1, i8* %c1_ptr
```

(b) ALAP-rearranged code

```
%a0 = load i8, i8* %a0_ptr
%a1 = load i8, i8* %a1_ptr
%a0_sext = sext %a0 to i9
%a1_sext = sext %a1 to i9
%b_sext = sext %b to i9
%c_pack = call i48 @_silvia_muladd(
    i9 %a0_sext, i9 %a1_sext, i9 %b_sext, i48 0)
%c = call { i18, i18 } @_silvia_muladd_extract(
    i48 %c_pack)
%c0_sext = extractvalue { i18, i18 } %c, 0
%c1_sext = extractvalue { i18, i18 } %c, 1
%c0 = trunc i18 %c0_sext to i8
%c1 = trunc i18 %c1_sext to i8
store i8 %c0, i8* %c0_ptr
store i8 %c1, i8* %c1_ptr
```

(c) DSP-packed code

Figure 4.3: The `C` code defined in Fig. 4.1a compiles to the LLVM code (a) where the two `mul` instructions (i.e., `c0` and `c1`) are incompatible for vectorization since `c0` is used before the definition of `c1`. SILVIA rearranges the code (b) to make `c0` and `c1` compatible, by moving the uses of `c0` as late as possible (ALAP) while preserving the functionality. Finally, SILVIA replaces the two `mul` instructions with a function call to the corresponding DSP-packed implementation (c).

4.2.3 Tuple packing

In the `packTuple` function, SILVIA processes packed tuple and creates a call to the function implementing the DSP-packed functionality in a valid packed insertion point (as explained in Section 4.2.2), generating, for instance, the code in Fig. 4.3c.

The called function can:

- Implement the optimized DSP-packed operation directly within the LLVM IR. For instance, `_silvia_muladd` called by `SILVIAMuladd` computes two 8-bit MADs through a single add-multiply-add (AMA) operation with the same precision of the DSP arithmetic units, according to the DSP packing methodology by Fu *et al.* [14], as shown in Fig. 4.4a. The HLS BE automatically binds the AMA to a single DSP, using its pre-adder, multiplier, and post-adder.

  The output of `_silvia_muladd` is packed on 48 bits to be fed to the next DSP in the chain,

```
define i36 @_silvia_muladd(i9 %a, i9 %b, i9 %c, i36 %C_port) {
entry:
  %A_port = call i27 @_ssdm_op_BitConcatenate.i27.i9.i18(i9 %a, i18 0)
  %D_port = sext i9 %b to i27
  %preadd_out = add i27 %A_port, %D_port
  %mul_in1 = sext i27 %preadd_out to i36
  %B_port = sext i9 %c to i36
  %mul_out = mul i36 %mul_in1, %B_port
  %P_port = add i36 %mul_out, %C_port
  ret i36 %P_port
}
```

(a) Multiply-and-adds computation.

```
define { i18, i18 } @_silvia_muladd_extract(i48 %P_port) {
entry:
  %mad_b = trunc i48 %P_port to i18
  %mad_b_msb = call i1 @_ssdm_op_BitSelect.i1.i48.i32(i48 %P_port, i32 17)
  %mad_b_msb_zext = zext i1 %mad_b_msb to i18
  %mad_a_raw = call i18 @_ssdm_op_PartSelect.i18.i48.i32.i32(
      i48 %P_port, i32 18, i32 35)
  %mad_a_corr = add i18 %mad_a_raw, %mad_b_msb_zext
  %mads_tmp = insertvalue { i18, i18 } undef, i18 %mad_a_corr, 0
  %mads = insertvalue { i18, i18 } %mads_tmp, i18 %mad_b, 1
  ret { i18, i18 } %mads
}
```

(b) Multiply-and-adds extraction.

Figure 4.4: The LLVM intermediate representation of two 8-bit multiply-and-adds DSP packing proposed by Fu *et al.* [14], called by `SILVIAMuladd` when automatically applying packing.

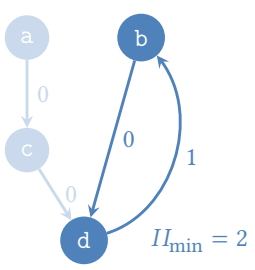
as described by Fu *et al.* [14]. Thus, `SILVIAMuladd` calls `_silvia_muladd_extract` (Fig. 4.4b) at the end of the DSP chain, to extract the actual MADs results.

- Be a functionally-equivalent placeholder, such that the HLS tool generates a dedicated module with the desired functionality and interface which can be replaced with a custom RTL module in the following step, described in Section 4.2.4. For example, the SIMD adder requires setting the `use_simd` Vivado synthesis directive that is not controllable from the LLVM IR; the multiplication between four signed 4-bit operands and one shared 4-bit operand is implemented at RTL [21].

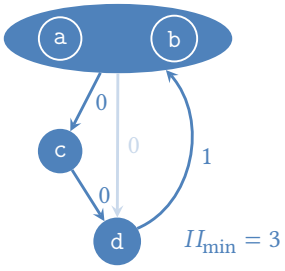
The `packTuple` step depends on the specific packed operation. For instance, `SILVIAMuladd` ensures that Eq. (2.6) is satisfied. In case the tuple contains more than N pairs of candidates, SILVIA splits them into multiple balanced DSP chains and sums the remaining partial MADs with an external adder tree to avoid overflow.

```
void top(char x[N], char y[N],
         char z[N]) {
  int d = 0;
  for (int i = 0; i < N; i++) {
    char a = x[i] + y[i];
    char b = x[i] + d;
    char c = w * a;
    d = c + b;
    z[i] = d;
  }
}
```

(a) Source code



(b) Original DDG



(c) Packed DDG

Figure 4.5: Example of an edge-case design where packing multiple operations to the same DSP is detrimental to the initiation interval (II) of the pipeline. The data dependence graph (DDG) (b) corresponding to the original source code (a), where the nodes are the instructions and the edges are the data dependencies between the instructions labeled with their distance, has a critical cycle (highlighted in dark blue) which determines a minimum II of 2 clock cycles (i.e., the maximum ceiled ratio between the total latency and the total distance along any cycle in the DDG), assuming a latency of 1 clock cycle for each operation. Packing `a` and `b` to the same DSP introduces a new critical cycle in the DDG (c) that increases the minimum II to 3 clock cycles.

4.2.4 Tuple replacement

Given a packed tuple, SILVIA replaces the uses of the tuple with the values computed by the packed function, such that the original tuple becomes dead code (i.e., without any use), and calls the *dead code elimination* LLVM pass to remove the leftover original tuple.

Finally, SILVIA replaces the HDL files of HLS-generated RTL modules corresponding to the placeholder functions called by `packTuple` with the related custom DSP-packed RTL modules. SILVIA relies on placeholder replacement, rather than directly using the equivalent "black box" functionality of Vitis HLS, since the "black box" currently has too many limitations for this purpose (e.g., Vitis HLS aborts synthesis if a "black box" is defined but not used).

4.2.5 Impact on the HLS backend

Replacing tuples of instructions with their packed counterpart impacts the behavior of the BE during scheduling (i.e., the assignment of the execution of each operation to specific clock cycles) and resource sharing (i.e., the binding of different operations to the same functional unit in different clock cycles).

Impact on scheduling

The pipeline II is a key parameter of a schedule, since the throughput is inversely proportional to it. Figure 4.5 shows an example where packing increases the minimum II. The packing changes the original DDG (Fig. 4.5b), corresponding to the source code in Fig. 4.5a, to the DDG

in Fig. 4.5c, where the candidates belonging to the packed tuple are merged into a single supernode. The fictitious data dependencies between the candidates of the tuple introduce a new cycle in the DDG, that has the same distance but a larger latency than the original critical cycle, making it the new critical cycle, according to Eq. (2.1).

There are multiple possible solutions to overcome this issue, such as discarding the candidates belonging to DDG cycles, which is computationally efficient but might be too conservative. Another solution is to drop the tuples that introduce new critical cycles, which is optimal but requires information that might be unavailable before the BE stage (e.g., the latency of the operations). However, since the depicted scenario empirically proved to be uncommon (e.g., none of the several benchmarks considered in Section 4.3 suffered from it), the management of this corner case is left to future work.

Impact on resource sharing

Resource sharing applies to operations of the same type only. For instance, in the `SILVIAAdd` case, if an addition instruction is compatible with the additions belonging to a packed tuple, the scheduler would fail to share the same functional unit between the addition and the tuple, because the BE would recognize them as different operations. To overcome this issue, SILVIA could map the addition instruction to a packed DSP, even if it is the only instruction in the tuple. With this approach, both the tuple and the unpaired addition would be recognized as operations of the same type and the resource sharing would successfully apply. This optimization is left to future work.

4.3 Evaluation

SILVIA is integrated in the flow of the AMD Vitis HLS 2023.2 tool. To execute the SILVIA optimized flow (Fig. 4.2), designers just need to update the synthesis script by replacing the standard command for launching synthesis (i.e., `csynth_design`) with the SILVIA's custom one (i.e., `SILVIA::csynth_design`), after selecting the SILVIA passes to run (i.e., by populating the `SILVIA::PASSES` variable with the ordered list of passes), as shown in Fig. 4.6. The `OP` option selects the SILVIA pass (e.g., `"add"` for the `SILVIAAdd`) and the `OP_SIZE` option specifies the maximum size of the operations to pack (e.g., `12` or `24` bits for the `SILVIAAdd`). Additionally, there are pass-specific options. `SILVIAAdd` provides the `INST` option to pack either addition or subtraction operations. `SILVIAMuladdd` exposes the `MAX_CHAIN_LEN` option to limit the length of cascaded DSP chains. By default, `SILVIAMuladdd` chains up to NDSPs, where $N$is defined as Eq. (2.6). Longer chains save more logic resources, summing the partial products with the DSP's internal post-adder, but consume more memory resources, due the deeper pipelines. Therefore, the `MAX_CHAIN_LEN` option enables trading off between logic and memory resources.

SILVIA automatically identifies and packs the parallel additions, multiplications, and MADs naturally exposed by HLS designs on a set of diverse benchmarks, ranging from simple BLAS accelerators to complex high-performance open-source designs, as discussed in Section 4.3.1.

Section 4.3.2 analyzes CNN accelerators use case, specifically the frameworks FINN [21] and NN2FPGA [44]. These frameworks implement the MAD packing supported by `SILVIAMuladdd` in a user-directed manner, while SILVIA is fully automated, allowing to compare the QoR of SILVIA with the manually-optimized designs.

```
+ source ${SILVIA_ROOT}/scripts/SILVIA.tcl
open_project ${PROJ_NAME}
open_solution ${SOL_NAME}
add_files ${SOURCE_FILES}
set_top ${TOP_NAME}
- csynth_design
+ set SILVIA::PASSES \
+     [list [dict create OP "muladd"] \
+           [dict create OP "add" OP_SIZE 12]]
+ SILVIA::csynth_design
export_design
```

Figure 4.6: Modifications to the Vitis HLS synthesis script for executing the optimized SILVIA flow. Users just need to update the Vitis HLS synthesis script by specifying which SILVIA passes to run, via the `SILVIA::PASSES` list, and running the `SILVIA::csynth_design` command.

4.3.1 Miscellaneous benchmarks

The benchmarks are groupable into two main categories, *addition-intensive benchmarks*, to evaluate the `SILVIAAdd` pass (configuring the `SILVIA::PASSES` variable to pack the 12-bit additions and the 24-bit additions), and *multiplication and MAD-intensive benchmarks*, to evaluate the `SILVIAMuladd` pass (configuring the `SILVIA::PASSES` variable to pack the 4-bit multiplications and the 8-bit multiplications/MADs with a maximum DSP chain length of 3).

The addition-intensive benchmarks include:

- The vector addition from the Xilinx example designs [54], summing two vectors of 192 8-bit elements.
- A spiking neural network (SNN) convolutional layer [55] accelerator, with a $24 \times 24 \times 64$ input feature map and $3 \times 3 \times 64 \times 128$ filter.

The multiplication-intensive benchmarks include:

- Four BLAS accelerators, namely the matrix-vector multiplication (MVM) between a 192×192 matrix and a 192×1 vector, the MMM between two 192×192 matrices, and the `axpy` and `scal` kernels on 512-element vectors from the Vitis Libraries [56]. In every case, the inputs are 8-bit integers. The MMM also includes a 4-bit unsigned integers configuration.
- The global system for mobile communications (GSM) kernel from CHStone [57] benchmark suite, with 8-bit words.
- The forward 3D hybrid boundary condition reverse time migration (RTM) accelerator from the Vitis Libraries [56], running on 8-bit fixed-precision data.
- The graph attention (GAT) graph neural network accelerator from FlowGNN [58], with 8-bit fixed-point data.

Each benchmark is implemented in different versions, including:

Table 4.1: Power, performance, and area of the benchmarks. Baseline DSP (BD) and unconstrained (BU) originate from the standard Vitis HLS flow; SILVIA (S) from the optimized SILVIA flow. BD and S designs bind the operations of interest to DSPs. The "←" indicates identical designs for BU and BD, when Vitis HLS automatically binds every operation to DSP. *Ops/Unit* is the average operations per arithmetic unit. Area and power derive from Vivado post-implementation reports, at 250 MHz. The maximum clock frequency is the highest at which post-routing timing is met. The on-chip memory utilization is omitted because SILVIA does not impact it.

(a) Addition-intensive benchmarks

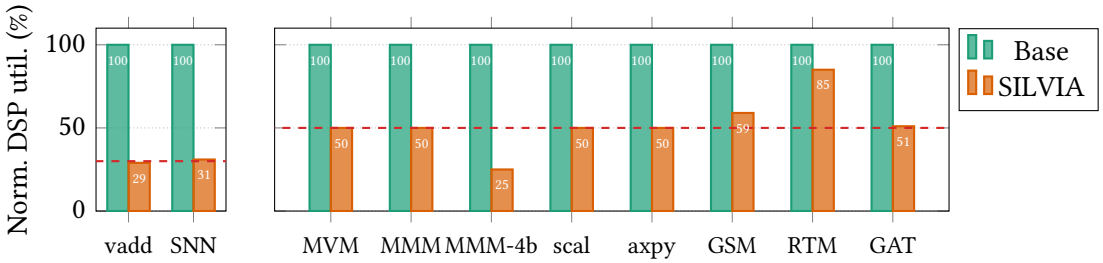
Bench.	**Ops/Unit** (1)		**DSP** (1)			**Logic LUT** (k)			**Mem. LUT** (k)			**FF** (k)			**Power** (mW)			**F_{clk}^{max}** (MHz)		
	BD,BU	S	BD	BU	S	BD	BU	S	BD	BU	S	BD	BU	S	BD	BU	S	BD	BU	S
vadd [54]	1.00	3.40	68	0	20	3.49	3.53	3.30	0.59	0.59	0.59	7.80	8.18	7.29	365	359	345	475	475	425
SNN [55]	1.00	3.19	150	0	47	1.47	2.05	1.19	0.00	0.00	0.05	2.72	3.85	2.95	439	352	373	450	600	450
N. gmean	1.00	**3.29**	1.00	0.00	**0.30**	1.00	1.19	0.87	1.00	1.00	7.07	1.00	1.22	1.01	1.00	0.89	0.90	1.0	1.15	0.95

(b) Multiplication-intensive benchmarks

Bench.	**Ops/Unit** (1)		**DSP** (1)			**Logic LUT** (k)			**Mem. LUT** (k)			**FF** (k)			**Power** (mW)			**F_{clk}^{max}** (MHz)		
	BD,BU	S	BD	BU	S	BD	BU	S	BD	BU	S	BD	BU	S	BD	BU	S	BD	BU	S
MVM	1.00	2.00	64	←	32	1.57	←	1.42	0.68	←	0.68	1.65	←	2.42	395	←	389	375	←	400
MMM	1.00	2.00	64	←	32	1.60	←	1.62	0.46	←	0.59	1.68	←	2.42	441	←	459	350	←	350
MMM-4b	1.00	4.00	64	←	16	1.50	←	2.00	0.23	←	0.27	1.26	←	2.11	438	←	440	300	←	350
`scal` [56]	1.00	2.00	64	0	32	2.46	4.54	2.46	0.01	0.01	0.01	6.85	8.44	7.36	354	411	351	475	475	475
`axpy` [56]	1.00	2.00	64	←	32	4.07	←	4.57	0.01	←	0.27	13.58	←	14.10	486	←	495	450	←	375
GSM [57]	1.00	1.58	41	←	24	0.80	←	0.98	0.03	←	0.04	0.78	←	1.33	333	←	324	350	←	350
RTM [56]	1.00	1.14	274	139	232	20.38	25.23	18.45	5.40	5.33	5.57	29.11	31.14	29.13	966	987	586	200	225	275
GAT [58]	1.00	1.97	1508	540	768	25.16	62.17	32.75	31.07	31.07	18.02	40.19	56.80	56.94	4578	4303	3208	325	325	325
N. gmean	1.00	**1.97**	1.00	0.00	**0.50**	1.00	1.24	1.09	1.00	1.00	1.54	1.00	1.08	1.33	1.00	1.01	0.92	1.00	1.01	1.05

- the baseline DSP (BD) designs, generated with the standard Vitis HLS flow, configured to bind the operations of interest (i.e., additions in the addition-intensive benchmarks and multiplications in the multiplication-intensive benchmarks) to DSPs, via the `config_op` command, for more direct comparison,
- the baseline unconstrained (BU) version, generated with the standard Vitis HLS flow without any resource binding constraint, and
- the designs optimized with the SILVIA (S) flow.

Table 4.1 reports the power, performance, and area of the benchmarks, from the post-implementation reports of AMD Vivado 2023.2, targeting UltraScale FPGA boards (specifically, the AMD Kria KV260, except for the GAT benchmark, which targets the AMD ZCU102, due to its higher resource requirements), with a clock constraint of 250 MHz (except for the RTM benchmark, constrained to 200 MHz due to the timing critical path of its BD and BU versions). The maximum clock frequency is the highest at which post-routing timing is met, at a granularity of 25 MHz. The operation density (Ops/Unit) is the ratio between the number of arithmetic operations and the number of functional units computing them, at the IR level.



(a) Addition-intensive benchmarks

(b) Multiplication-intensive benchmarks

Figure 4.7: DSP utilization, normalized over the baseline, of several benchmarks. Dashed, horizontal lines represent the DSP utilization's geometric mean of designs optimized with SILVIA.

SILVIA increases the mean operation density to 3 for the addition benchmarks and to 2 for the multiplication ones, by automatically identifying and packing the compatible operations available in the benchmarks to DSPs, without affecting the cycle count performance. These results demonstrate that SILVIA successfully achieves its goal of *automating DSP packing* within the HLS flow. Although the effectiveness of DSP packing itself is well-established, as evidenced by numerous SOTA designs [42]–[45] exploiting it, the rest of this section analyzes its impact on power, performance, and area metrics.

In principle, the DSP saving should be directly proportional to the operation density. However, the HLS and the logic synthesis reduce the DSP utilization of the baseline of some benchmarks too (i.e., the used DSPs are fewer than the total number of operations), thanks to different optimizations such as resource sharing or three-input additions mapping to a single DSP, slightly reducing the advantage of SILVIA over the baseline. Nevertheless, on average, SILVIA saves 70 % DSPs in the addition benchmarks and 50 % DSPs in the multiplication benchmarks with respect to the baseline DSP. Those results suggest that the wider space available when SILVIA optimizes the DSP utilization at higher abstraction levels (i.e., at the LLVM IR level, rather than at the RTL) enables achieving better QoR. Figure 4.7 summarizes the DSP utilization of each benchmark, normalized over the baseline DSP utilization.

The II of the pipelines of each benchmark is the same in both the baseline and the SILVIA-optimized designs (i.e., the scenario depicted in Section 4.2.5 is never encountered). Therefore, SILVIA does not affect the throughput of the designs. On the other hand, DSP packing introduces data dependencies between originally-independent instructions, increasing pipelines depth (SILVIA-optimized pipelines are, on average, 27 % deeper than baseline). While deeper pipelining does not impact throughput, which depends on the II, it requires additional pipelining registers. Consequently, SILVIA-optimized designs use more memory LUTs and FFs.

MAD packing introduces an average 9 % overhead in logic LUTs due to diverse reasons, including the error-correction logic required by the MAD packing methodologies, and the missed opportunities to compute MADs on a single DSP when the additions do not comply with Eq. (2.5). For instance, the `axpy` benchmark adds a third operand d_i to the products (i.e., $p_i = a_i \cdot c_i + d_i$), whereas the packed MAD can only sum together DSP-packed products (i.e., $p = a_i \cdot c_i + a_{i+1} \cdot c_{i+1}$). Thus, the SILVIA `axpy` adds d_i with LUT adders, while the baseline computes one MAD per DSP.

On average, SILVIA-optimized designs consume 8 % less power in multiplication-intensive benchmarks and 10 % less in addition-intensive benchmarks compared to the baseline, due to substantial reductions in DSP utilization.

The impact of DSP packing on the timing critical path is not uniform. Some designs optimized with SILVIA achieve higher clock frequencies, likely due to reduced resource usage easing placement, while others experience lower frequencies, potentially due to increased routing congestion from higher computation density in packed DSPs.

SILVIA always executes in less than one second when optimizing each benchmark, except for the largest designs (i.e., RTM and GAT require around 6 seconds and 2 minutes, respectively). In every case, the execution time of SILVIA is negligible with respect to the whole HLS (taking minutes) and implementation (taking from minutes to hours) flows.

4.3.2 CNN acceleration case study

NN2FPGA [44] and FINN [21] are open-source frameworks that generate the hardware description of CNN inference accelerators. Both frameworks instantiate pre-defined parametrized building blocks implementing the CNNs's operations. They provide convolution blocks described in HLS that do not implement any DSP packing (i.e., "baseline"). Moreover, they also provide user-selectable manually-optimized implementations which minimize the DSP utilization via packing (i.e., "manual"). FINN's "manual" implementation describes the factor-2 and factor-4 packings at RTL.NN2FPGA's "manual" implementation describes the factor-2 packing in synthesizable `C++` for HLS. The SILVIA-optimized versions automatically apply DSP packing to the "baseline" designs.

Table 4.2 shows the post-implementation results at a clock frequency of 200 MHz generated by Vivado 2022.2 targeting the AMD Kria KV260 for the NN2FPGA designs and Vivado 2023.2 targeting the AMD ZCU102 for the FINN designs. The throughput is measured in hardware.

The *Minimum DSP* experiments are set up as a throughput-constrained DSP minimization problem. Therefore, both baseline and optimized designs have the same parallelism and the DSP packing minimizes the DSP resources without affecting the throughput. SILVIA successfully matches the DSP utilization of the manually optimized designs, both from NN2FPGA and FINN.

The *Maximum performance* experiments are set up as a DSP-constrained throughput maximization problem. The higher DSP operation density of the optimized designs enables larger parallelism at the same number of DSPs, maximizing the throughput. Again, the SILVIA's QoR is on par with the manually-optimized designs, matching their performance per DSP.

As expected, the optimized designs consume more LUTs and FFs than the baseline due to the increased data demands from higher parallelism. Notably, SILVIA-optimized designs consistently use less logic LUTs than manually optimized ones, as the HLS tool fails to efficiently bind operand concatenation from NN2FPGA's source-level packing to the DSP pre-adder, instead mapping it to an LUT-based adder.

In both experiments, FINN designs utilize significantly less logic LUT and FF than the SILVIA's ones because the whole convolutional layer with packing is fine-tuned at RTL, rather than using the HLS model instantiated when the packing is disabled.

Figure 4.8 shows that SILVIA effectively matches the manually optimized designs in the DSP versus throughput space, as their design points consistently overlap. This demonstrates SILVIA's ability to automatically Pareto-dominate the baseline designs.

Table 4.2: Power, performance, and area of the convolutional neural network accelerators. Baseline (B) do not use packing, manually-optimized (M) use source-level packing, and the SILVIA (S) use automatic SILVIA-flow packing. Resource utilization is from implementation at 200 MHz.

(a) NN2FPGA accelerators

Goal	**Model**	**DSP** (1)			**Logic LUT** (k)			**Mem. LUT** (k)			**FF** (k)			**Throughput** (kFPS)		
		B	M	S	B	M	S	B	M	S	B	M	S	B	M	S
Min. DSP	ResNet20	635	318	318	43.7	47.3	43.6	10.2	9.9	9.9	53.5	54.2	54.9	3.05	3.05	3.05
	ResNet8	773	387	387	31.9	35.4	31.7	6.0	6.0	6.3	36.0	36.2	38.8	12.20	12.20	12.20
	N. gmean	1.00	**0.50**	**0.50**	1.00	1.10	1.00	1.00	0.99	1.01	1.00	1.01	1.05	1.00	1.00	1.00
Max. perf.	ResNet20	635	626	626	43.7	66.7	60.3	10.2	11.4	11.5	53.5	68.7	68.1	3.05	6.10	6.10
	ResNet8	773	773	773	31.9	61.8	53.5	6.0	8.3	8.6	36.0	63.8	64.0	12.20	24.38	24.38
	N. gmean	1.00	0.99	0.99	1.00	1.72	1.52	1.00	1.24	1.27	1.00	1.51	1.50	1.00	**2.00**	**2.00**

(b) FINN accelerators

Goal	**Model**	**DSP** (1)			**Logic LUT** (k)			**Mem. LUT** (k)			**FF** (k)			**Throughput** (kFPS)		
		B	M	S	B	M	S	B	M	S	B	M	S	B	M	S
Min. DSP	CNV-8b	90	43	43	39.4	21.1	39.4	6.6	6.8	6.6	101.7	42.4	103.5	0.05	0.05	0.05
	MobileNet-4b	419	163	163	63.0	52.9	63.3	27.7	28.1	27.8	144.0	117.1	150.8	0.10	0.10	0.10
	N. gmean	1.00	**0.43**	**0.43**	1.00	0.67	1.00	1.00	1.02	1.00	1.00	0.58	1.03	1.00	1.00	1.00
Max. perf.	CNV-8b	90	86	86	39.4	22.9	39.8	6.6	8.1	7.9	101.7	44.0	106.6	0.05	0.10	0.10
	MobileNet-4b	419	427	427	63.0	66.7	79.7	27.7	28.3	27.8	144.0	123.4	185.0	0.10	0.26	0.27
	N. gmean	1.00	0.99	0.99	1.00	0.78	1.13	1.00	1.12	1.10	1.00	0.61	1.16	1.00	**2.28**	**2.32**

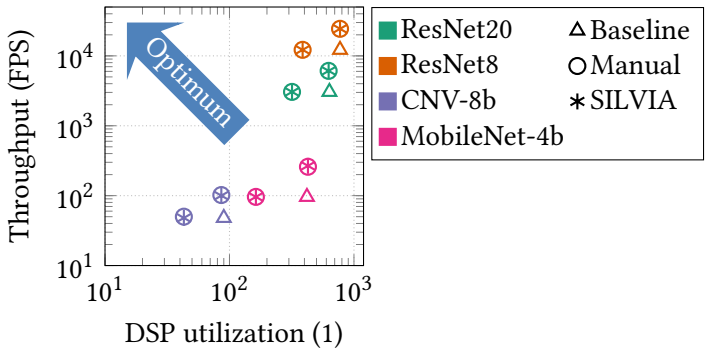


Figure 4.8: The convolutional neural network accelerators in the DSP utilization versus throughput space. SILVIA matches the quality of results of the manually-optimized designs.

4.4 Discussion

This dissertation proposes SILVIA, the first open-source LLVM infrastructure to automatically identify and optimize DSP-packable operations in HLS FPGA designs. SILVIA packs four up-to-12-bit or two up-to-24-bit additions or subtractions, two up-to-8-bit or four up-to-4-bit multiplications, or two up-to-8-bit MADs on a single DSP. Moreover, it is designed to readily support other operations, reusing most of the existing infrastructure.

SILVIA automatically identifies and optimizes for the superword-level parallelism a diverse set of benchmarks, reducing the DSP utilization for addition instructions by 70 % and for multiplications and MADs by 50 %, on average. Moreover, SILVIA achieves DSP utilization that match manually optimized SOTA designs.

Chapter 5

Task-level multi-pumping

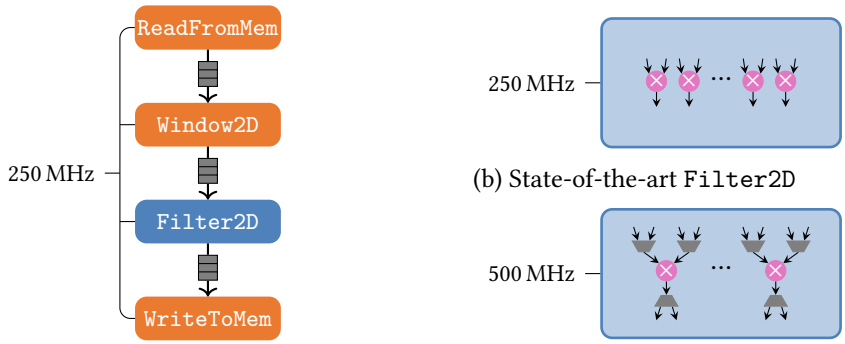
The HLS abstractions enable automated optimizations that would be challenging or unfeasible at the RTL. This dissertation introduces a task-level multi-pumping methodology aimed at reducing the utilization of resources, particularly DSPs, while maintaining the throughput of HLS kernels modeled as DFGs for FPGAs. This methodology leverages HLS resource sharing to automatically insert logic that reuses functional units across multiple operations. Additionally, it employs multi-clock DFGs to run the multi-pumped tasks at higher frequencies.

The methodology scales the pipeline II and clock frequency of resource-intensive tasks by a multi-pumping factor (M). The looser II allows sharing the same resource among M different operations, while the tighter clock frequency preserves the throughput. This approach expands the Pareto front in the throughput–resource space by applying it to open-source HLS designs, using AMD's SOTA commercial HLS and implementation tools. The multi-pumped designs reduce DSP usage by up to 40 % while maintaining the same throughput as the original, performance-optimized designs (i.e., running at the maximum clock frequency). Moreover, they achieve up to 50 % higher throughput using the same number of DSPs as the resource-optimized designs operating with a single clock.

5.1 Overview

A DFG consists of parallel computational tasks (`C/C++` functions in HLS) that communicate asynchronously via FIFO channels. Typically, HLS tools implement DFGs as SCDFGs, where all tasks share the same clock signal. Many modern HLS tools lack support for multi-clock designs [8], [10]. However, SCDFGs can be generalized to MCDFGs by assigning each task its own dedicated clock domain. This generalization improves task flexibility and allows each task to run at its maximum frequency, constrained only by the local critical timing path rather than the global one, as discussed in Section 2.1.3. Modern FPGA system-on-chips (SoCs) natively support multiple clock domains, and the area overhead for safe clock domain crossing (CDC) is minimal. This is because tasks already communicate through FIFOs, which can be configured with independent read and write clocks while maintaining comparable resource utilization [59].

Multiple clock domains enable optimizations such as multi-pumping, which reduces resource usage while maintaining the throughput by reusing a resource, typically a DSP unit in the FPGA context, M times. This is achieved by clocking the resource at a frequency M times



(a) SCDFG

(b) State-of-the-art `Filter2D`

(c) Double-pumped `Filter2D`

Figure 5.1: Task-level multi-pumping saves resources at equal throughput for HLS of dataflow graphs (DFGs). The `Filter2D` task from a 2D convolution kernel [5] (a) is double-pumped (c) by doubling its clock frequency and II to save half of the multipliers of the single-clock solution (b).

larger than the rest of the system. Traditionally, designers apply multi-pumping at the RTL by manually inserting the custom logic to share the resource and ensure safe CDC. This manual process requires significant design effort and expertise.

This dissertation implements multi-pumping at the task level by adjusting high-level parameters, specifically the pipeline II and the clock constraint at the task level, utilizing the flexibility of MCDFG. The HLS resource sharing algorithm automatically generates the logic to share resources within a dataflow task, as explained in Section 2.1.2, while the inter-task FIFOs ensure safe CDC. This dissertation focuses on DSPs, which are crucial in compute-intensive kernels and capable of operating at high frequencies. However, the technique can be applied to multi-pump any shareable resource, including entire sub-functions.

For example, consider the 2D convolution HLS kernel from AMD [5], already introduced in Section 2.1, implemented as an SCDFG, as shown in Fig. 5.1a, where rectangular nodes represent tasks and arrows indicate FIFOs. The `Filter2D` task processes a convolution window of up to 15×15 elements at each iteration, computing 225 multiply and accumulate (MAC) operations, which are bound to DSPs. Thus, an II of 1 cycle requires 225 DSPs. However, if the II is scaled to 2 cycles, a new pipeline iteration begins every two clock cycles, giving the pipeline two cycles to compute 225 operations. Thus, with resource sharing, the HLS tool allocates only $\lceil 225/2 \rceil = 113$ DSPs, with each DSP computing two MACs.

Assuming a target throughput of 250 MSa/s, in a traditional SCDFG flow (Fig. 5.1b), the clock frequency for the entire DFG, including the `Filter2D` task with 225 DSPs, would be set to 250 MHz. In contrast, with the proposed task-level multi-pumping approach (Fig. 5.1c), the `Filter2D` task is optimized by scaling its II to 2 cycles, reducing the DSP count by half, and increasing its clock frequency to 500 MHz to maintain the same throughput.

This dissertation introduces an area-minimization methodology that preserves throughput via task-level multi-pumping for FPGA HLS designs modeled as DFGs. The effectiveness of the methodology is validated on open-source designs using the workflow depicted in Fig. 5.2, which generates an optimized multi-pumped intellectual property (IP) block from `C/C++` source code using state-of-the-art AMD commercial tools [38].

To the best of the author's knowledge, this is the first work that combines multiple clock

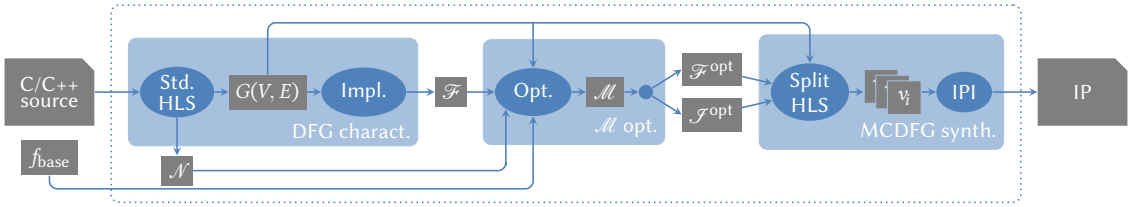


Figure 5.2: Given the C/C++ source code of a dataflow graph (DFG) application and its base clock frequency, the proposed workflow builds the optimized multi-pumped IP by (a) analyzing the DFG (*DFG charact.*), (b) optimizing the multi-pumping factors (*opt.*), and (c) synthesizing the multi-pumped IP (*MCDFG synth.*).

domains with resource sharing in HLS of DFGs for task-level multi-pumping. Empirical results demonstrate that this approach opens a new Pareto front in the PPA space, achieving up to 60 % reduction in DSP usage at maximum throughput or up to 50 % higher throughput using the same number of DSPs.

5.2 Related work

This dissertation primarily focuses on improving QoR in HLS designs by tuning HLS directives – the instructions for the HLS compiler to control hardware optimizations such as loop pipelining – with an emphasis on multi-clock designs.

Several studies [60]–[64] optimize HLS directives for performance via DSE. However, they optimize plain software code not originally intended for HLS, while this dissertation operates on source code that is already optimized for HLS. Moreover, this dissertation eliminates the need for time-consuming DSEs by analytically computing the multi-pumping factor, as well as the corresponding II and clock frequency constraints. Finally, most of these works consider only single clock designs, except for Liang *et al.* [60], which is discussed further in Section 5.2.1.

HLS design optimizations using multiple clock domains typically operate at the *operation level*, assigning domains at the low-level resource granularity (e.g., adder or multiplier) during scheduling [65]–[67], or at the *task level*, where domains are assigned at the function granularity (i.e., MCDFGs) [60], [68].

5.2.1 Operation-level multi-clock in high-level synthesis

Lhairech-Lebreton *et al.* [66] employ multiple clock domains in HLS to reduce power consumption while preserving the throughput by halving the operating frequency of two-cycle operations. In contrast, this dissertation emphasizes area and performance optimizations, as power is a secondary quality metric for FPGA designs after performance and area.

Canis *et al.* [65] and Ronak and Fahmy [67] develop double-pumped DSP modules and use them in HLS with custom resource-sharing algorithms. Although AMD Vitis HLS theoretically supports double-pumped MAC operations through user-callable functions from the `dsp_builtins` library, this functionality is undocumented and has known issues [69]. The proposed task-level multi-pumping approach achieves similar outcomes when double-pumping a task; however, being a task-level solution, it does not require custom modules, modifications

to the HLS sharing algorithm, or alteration to the source code. Furthermore, the task-level methodology allows for the selection of multi-pumping factors greater than two, leading to larger resource savings.

5.2.2 Task-level multi-clock in high-level synthesis

Ragheb and Anderson [68] focus on extending the LegUp HLS tool to support MCDFGs synthesis, but they rely on a suboptimal, time-consuming, profiling-based approach for selecting the clock frequencies. In contrast, this dissertation provides a general methodology for leveraging multiple clock domains. The workflow for synthesizing MCDFGs, based on state-of-the-art AMD tools, serves primarily to implement the proposed methodology.

Liang *et al.* [60] introduce a DSE methodology focused on maximizing the throughput while adhering to area constraints in HLS for MCDFG designs. They iteratively optimize for performance the HLS loop directives applied to the bottleneck tasks. If a task remains a bottleneck after all possible optimizations (e.g., when the pipeline II constraint is 1 cycle), they relax the directives for all tasks, increase the clock frequency of the bottleneck task, and repeat the process. In contrast, this dissertation aims to minimize area while maintaining throughput. Additionally, this optimization strategy differs, as it targets all resource-intensive tasks regardless of their bottleneck status, and it does not tighten the II constraints, which the HLS compiler may be unable to satisfy (e.g., due to data dependencies).

5.3 Methodology

Section 5.3.1 presents the rationale behind the task-level multi-pumping, emphasizing how to compute the optimal multi-pumping factors. Section 5.3.2 outlines the workflow developed for implementing this technique in Vitis HLS. Together, these elements provide a clear framework that combines the theoretical foundation with practical execution of the proposed task-level multi-pumping approach.

5.3.1 Task-level multi-pumping

This dissertation introduces a multi-pumping approach for the resources of task v_i by scaling both the II and the clock frequency of v_i simultaneously by a multi-pumping factor M_i.

The principles underpinning this approach are:

- **Independent tuning of tasks**: each task can be tuned independently without compromising the overall DFG throughput, provided that the throughput of the task does not fall below that of the bottleneck task, according to Eq. (2.4).
- **Reuse of functional units**: as outlined in Section 2.1.2, increasing the II of a pipeline by a factor M_i enables the reuse of the same functional unit for M_i operations across different clock cycles.
- **Task throughput**: Eq. (2.3) implies that the task throughput is unchanged when both the clock frequency and the II are scaled by M_i.

Assume that v_i meets the timing constraints up to a maximum frequency $f_i^{\max}$ and computes N_i^{OP} operations bound to DSPs. Additionally, the tasks that are not multi-pumped are clocked at the frequency f_{base}, which represents the clock constraint defined by the designer. The maximum multi-pumping factor for task v_i is

$$M_i^{\max} \triangleq \min\left(\left\lfloor \frac{f_i^{\max}}{f_{\mathrm{base}}} \right\rfloor, N_i^{\mathrm{OP}}\right). \tag{5.1}$$

It is important to highlight that the proposed task-level multi-pumping approach solely modifies the HLS directives, treating the HLS tool as a black box, and does not necessitate any manual restructuring of the source code.

5.3.2 Multi-pumping workflow

To validate the proposed task-level multi-pumping, this section defines a workflow to convert `C/C++` HLS source code into an optimized MCDFG IP block compatible with AMD tools [38], as depicted in Fig. 5.2. The main steps of the workflow include the *DFG characterization*, which involves extracting the maximum clock frequency and the number of DSP operations for each task, the *multi-pumping factor optimization*, to determine the optimal multi-pumping factor for each task, and the *MCDFG synthesis*, which generates the multi-pumped IP.

Dataflow graph characterization

For each task in the DFG $G(V, E)$, the number of DSP operations ($\mathcal{N} = \{N_i^{\mathrm{OP}}, \forall v_i \in V\}$) is extracted from the reports of the standard SCDFG HLS. Additionally, the post-implementation reports of the SCDFG provide the maximum frequency that meets the timing constraints ($\mathcal{F} = \{f_i^{\max}, \forall v_i \in V\}$). The implementation is set with a tight clock constraint (e.g., 500 MHz) and at the lowest pipeline II, representing the worst-case scenario for the critical cycle (as defined in Section 2.1). When multi-pumping increases the II, it relaxes the critical cycle, allowing for deeper pipelines and shorter critical paths, which enables higher clock frequencies.

The $\mathcal{F}$ is not extracted from the earlier HLS clock frequency estimations because they are often inaccurate [7]. The SCDFG implementation is run only once, making the overhead manageable. However, if a rapid flow is required (e.g., in early design phases), it is possible to collect timing data from the logic synthesis step, skipping placement and routing. The timing estimations at the logic synthesis step are more accurate than those from the HLS compiler, as they utilize lower-level information. Under-estimating maximum frequency may miss opportunities to save resources due to lower multi-pumping factors, as indicated in Eq. (5.1). Conversely, if the frequency is overestimated, timing issues may arise during implementation.

Multi-pumping factor optimization

The optimization workflow selects the multi-pumping factors $\mathcal{M} = \{M_i, \forall v_i \in V\}$ with the goal of minimizing DSP utilization. If the task v_i includes operations mapped to DSP, it assigns $M_i = M_i^{\max}$, as defined in Eq. (5.1). Otherwise, it refrains from applying multi-pumping to v_i.

Multi-clock dataflow graph synthesis

AMD Vitis HLS does not support the direct synthesis of MCDFGs, as it is limited to a single clock domain per the design. However, the `dataflow` directive generates several independent modules, one for each task, and interconnects them in a top-level module. Thus, the proposed workflow employs a *split* HLS, synthesizing each task separately by designating it as the top module and applying its specific clock constraint.

Finally, the separately synthesized tasks are interconnected using the Vivado intellectual property integrator (IPI). The AMD HLS tools use FIFOs as inter-task communication channels when data is produced and consumed in the same order; otherwise, ping-pong buffers are employed. While the proposed method could accommodate both types, the AMD IPI does not currently offer a configurable multi-clock ping-pong buffer. Therefore, this dissertation currently supports only FIFO channels, using the AMD FIFO generator [70]. These FIFOs are configured with independent clocks for the read and write ports when connecting tasks assigned to different clock domains.

5.4 Evaluation

The applicability and benefits of the task-level multi-pumping workflow, described in Section 5.3.2, are assessed on open-source HLS designs. The experiments target the embedded platform Zynq UltraScale+ FPGA SoC hosted by the Avnet Ultra96v1 board [37] and use Vitis HLS 2022.2 [8] for synthesis and Vivado 2022.2 [38] for implementation.

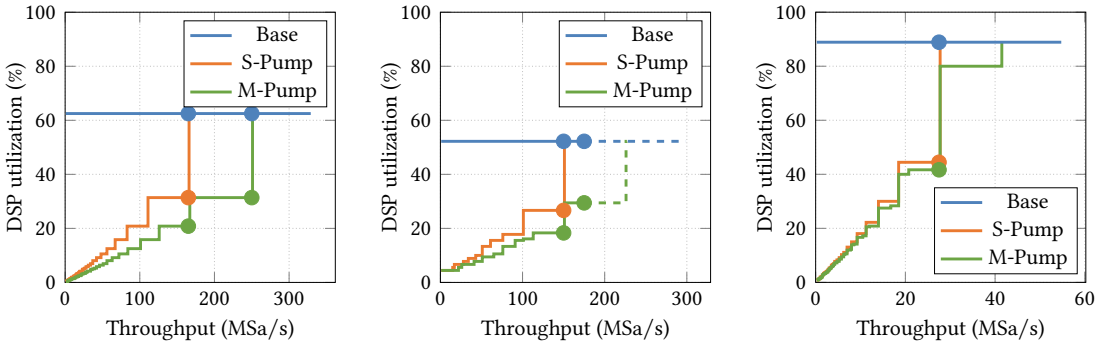
Resource utilization data is gathered from the post-implementation reports, and power estimations are obtained from post-implementation static power analysis. The throughput (i.e., the number of output samples produced in the unit of time) is evaluated by measuring the time for 10 000 executions in auto-restart mode [8] (to make the time overhead for control negligible) of the kernels in hardware, using the PYNQ APIs [35].

The open-source HLS designs under consideration include the *2D convolution* from the Vitis Tutorials [5] previously discussed in Section 5.1, the *Optical Flow* from the Rosetta suite [71], and the *virtual molecule screening (VMS)* [72], which serves as a drug discovery accelerator. For each design, the multi-pumped implementations (*M-Pump*) are compared against the original versions (*Base*) and the most efficient SCDFG implementations without altering the source code (*S-Pump*).

The *S-Pump* implementations are optimized with the proposed flow without extending them to MCDFGs. Thus, if a task v_i is "single-pumped" by a factor S_i, its II is scaled by S_i, just as in the original workflow, and the clock frequency of the entire kernel. It is important to note that the maximum "single-pumping" factor for each task is lower than the corresponding maximum multi-pumping factor (defined by Eq. (5.1)) since it is at most

$$S_i^{\max} \triangleq \left\lfloor \frac{\min\limits_{\forall v_i \in V} f_i^{\max}}{f_{\text{base}}} \right\rfloor. \tag{5.2}$$

Figure 5.3 illustrates the trade-offs between DSP utilization and throughput achieved by adjusting the base clock frequency within the limits set by the designs' critical path. The dashed lines represent computation throughput that exceeds the memory throughput. Consequently,



(a) 2D convolution (b) Optical Flow (c) VMS

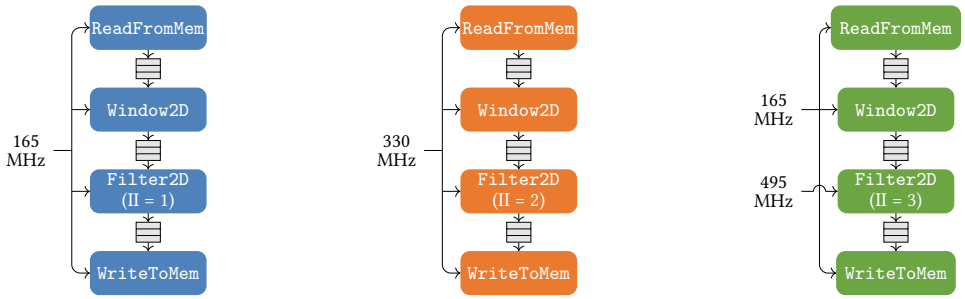
Figure 5.3: Digital signal processors (DSPs) allocated for a given throughput. The *M-Pump* designs are optimized using the proposed task-level multi-pumping technique. The *M-Pump* designs are Pareto-optimal compared to the *Base* designs, whose DSP utilization is constant since they are optimized by tuning the clock frequency only, and to the *S-Pump* designs, which are optimized for area by changing both the II and the global clock frequency of the tasks. The dashed lines represent the theoretical throughput achievable with the allocated DSPs, which are unreachable in practice due to memory bandwidth limitations. The dots show the design points implemented in hardware.

the effective throughput is practically limited to the maximum non-dashed value, which corresponds to the peak memory throughput.

For the *Base* designs, the number of DSPs remains constant regardless of the clock frequency. In contrast, the DSP utilization of the *Pump* designs is step-shaped, with discontinuities corresponding to changes in the IIs, which only assume integer values. The *Pump* solutions offer various trade-offs in the throughput versus DSP space, thanks to the tuning of the pipelines' II. The additional degree of freedom of the *M-Pump* implementations (i.e., the task clock frequency) ensures that they are always Pareto optimal.

For example, with the 2D convolution kernel targeting a throughput of 165 MSa/s, the Base implementation (Fig. 5.4a) has a single global clock at 165 MHz and every task scheduled with an II of one cycle. The S-Pump version (Fig. 5.4b) doubles the global clock frequency to double the II of the `Filter2D` task while meeting the throughput constraint, using half of the DSPs. Finally, the M-Pump case feeds the `Filter2D` task with a second clock whose frequency is three times higher than the Base clock frequency, enabling to triplicate the II, resulting in the DSP utilization reduces by three times at the same throughput.

Both *M-Pump* and *S-Pump* designs degenerate to *Base* designs (where all the pumping factors equal to one, resulting in no resource savings) at the highest throughput, as they require the lowest IIs to reach the highest performance. It is important to note that the *M-Pump* designs consistently degenerate to *Base* at higher throughput compared to *S-Pump* designs. This occurs because multiple clock domains enable the multi-pumped tasks to operate at the maximum frequency allowed by their respective local critical paths. As a result, the *M-Pump* designs achieve up to 52 % higher throughput than *S-Pump* utilizing the same number of DSPs in the 2D convolution test case. Furthermore, in the Optical Flow benchmark, the *M-Pump* reaches the maximum effective throughput using 40 % fewer DSPs compared to the *Base*.



(a) Base (b) S-Pump (c) M-Pump

Figure 5.4: Dataflow graphs of a 2D convolution kernel with a target throughput of 165 MSa/s. The Base implementation (a) schedules the `Filter2D` task with an initiation interval (II) of one clock cycle, resulting in the highest digital signal processor (DSP) utilization. The S-Pump (b) doubles the global clock frequency and the II of the `Filter2D` task to halve the DSP utilization. The M-Pump (c) multi-pumps the `Filter2D` task, reducing its DSP utilization to one third with respect to the Base.

Table 5.1: Power, performance, and area of the benchmarks targeting a Zynq UltraScale+ SoC.

Design	**T.put** (MSa/s)	**Impl.**	**Base clock** (MHz)	**Pump factors** (1)	**DSP** (%)	**LUT** Logic (%)	**LUT** Mem. (%)	**FF** (%)	**BRAM** (%)	**Power** Static (W)	**Power** Dyn. (W)	**Clock routing** (%)
2D Conv. [5]	165	Base	165	–	64	14	12	9	4	0.3	1.8	1.0
		S-Pump	330	2	33	15	12	18	4	0.3	2.3	1.0
		M-Pump	165	3	**23**	14	12	18	4	0.3	2.2	2.4
	250	Base	250	–	64	13	12	10	4	0.3	2.0	1.0
		M-Pump	250	2	**33**	15	12	19	4	0.3	2.8	2.4
Optical Flow [71]	150	Base	150	–	55	36	65	23	20	0.3	2.5	1.0
		S-Pump	300	2	29	37	65	26	20	0.3	3.2	1.0
		M-Pump	150	2, 3	**21**	37	64	27	20	0.3	3.0	3.8
	175	Base	175	–	55	36	65	23	20	0.3	2.5	1.0
		M-Pump	175	2	**33**	38	65	27	20	0.3	2.9	2.4
VMS [72]	28	Base	110	–	89	32	23	31	67	0.3	2.0	1.0
		S-Pump	220	2	44	30	23	31	67	0.3	2.4	1.0
		M-Pump	110	2, 3	**42**	32	23	37	67	0.3	2.9	3.8

Table 5.1 presents the post-implementation PPA data for the design points identified by the dots in Fig. 5.3. These points are selected because their throughput is the upper limits of the last steps in both *M-Pump* and *S-Pump* designs within the memory bound.

When comparing the *M-Pump* designs to the *Base* designs, there is a consistent reduction in DSP utilization, averaging 54 %. However, this DSP saving is accompanied by power consumption and FFs utilization overheads. The average power increase is 24 %, primarily due to the greater switching activity in multi-pumped tasks, which results from higher resource reuse and clock frequencies. Similarly, the FF utilization increase is 33 %, as the HLS tool builds deeper pipelines (requiring more registers) to reach higher clock frequencies in the multi-pumped

tasks.

As expected [59], the PPA overhead associated with CDC in *M-Pump* is negligible. The overhead from routing multiple clocks is also minimal, utilizing only 1.4 % of the available clock routing resources per additional clock domain.

In general, the *M-Pump* solutions outperform the *S-Pump* designs in terms of resource efficiency, as indicated by their Pareto dominance. At equivalent throughput, *M-Pump* designs allocate fewer DSPs, comparable LUTs and FFs, and consume less power. This advantage stems from the use of multiple clock domains, which allows the clock frequency to be increased selectively for multi-pumped tasks, resulting in higher multi-pumping factors without incurring significant power and FF overhead in the tasks that are not multi-pumped. The only exception is the VMS test case, where most of the logic is subject to double or triple-pumping. Since only a small fraction of its logic runs at the base clock frequency, the lower-frequency tasks are not enough to balance the power and FF overhead for the multi-pumped tasks.

5.5 Discussion

This dissertation proposes a task-level multi-pumping technique for saving hardware resources while maintaining the original throughput for HLS dataflow designs for FPGAs. Given a SOTA single-clock DFG, the proposed approach first generalizes it to a multi-clock DFG. Secondly, it tunes the tasks' high-level parameters (i.e., clock frequency and pipeline II) to multi-pump their functional units. The overhead for generalization is negligible, thanks to the DFGs structure, which consists of independent blocks communicating via FIFOs, allowing for safe CDC, and modern FPGA clock architectures, which seamlessly handle multiple clock domains even if current HLS tools do not exploit them.

The experimental results reported in Section 5.4 prove that the proposed method opens a new Pareto front in the performance versus DSPs space, saving up to 40 % of resources at maximum throughput. Moreover, the proposed method does not require any manual architecture changes from the designer, since it acts only on the high-level parameters of the tasks and uses the HLS binding algorithm to automatically generate the resource sharing logic. Finally, the generalization to multi-clock DFGs simply requires replacing single-clock with multi-clock FIFOs. Therefore, the proposed technique is well suited for a fully automated HLS optimization pass, which will be the subject of future work.

Chapter 6

Conclusion

Electronic design automation (EDA) tools are essential for designing and verifying increasingly complex hardware architectures. Historically, managing growing complexity has been addressed by moving towards higher abstraction levels, as evidenced by notable examples, such as the transition from gate-level to RTL design. Today, most digital hardware development is conducted at RTL. HLS raises the abstraction to the functional level, allowing designers to focus on "what" the hardware does, while compilers determine "how" the hardware implements the desired functionality, based on high-level PPA constraints.

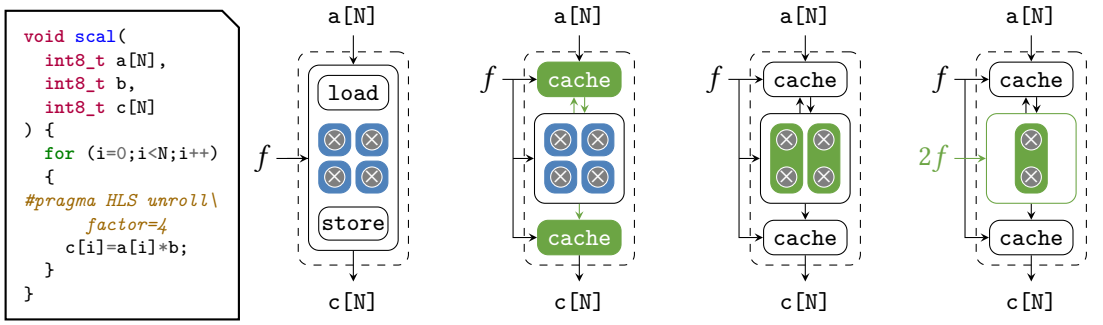
The higher abstraction level boosts designer productivity, which is especially critical in fast-paced markets, such as FPGA accelerators. However, achieving a QoR on par with manually-optimized RTL often requires substantial manual intervention in HLS, to optimize low-level, implementation-dependent details. Thus, manually-optimized HLS source code may significantly diverge from a purely functional description, defeating the goals of HLS itself.

Therefore, this dissertation extends the SOTA HLS in two orthogonal directions, by:

- Further raising the abstraction level of HLS. The CCC pattern semi-automatically generates LCS-like kernels which achieve performance comparable to manually-optimized LCS designs, without necessitating extensive algorithm refactoring. It leverages general-purpose, customizable, cache-based buffering tasks. The SILVIA framework automates DSP packing optimizations by applying compiler transformation passes that identify compatible low-precision operations and efficiently pack them to single high-precision DSPs.

- Introducing novel QoR optimization techniques that exploit the available abstractions. The task-level multi-pumping technique takes advantage of high-level pipelining control to automatically generate multi-pumping sharing logic, reducing hardware resource utilization while maintaining throughput.

The proposed techniques are orthogonal to each other and can be combined for a given HLS kernel. For instance, let us consider a kernel that multiplies an off-chip 8-bit integer array by a constant (Fig. 6.1a). The standard HLS flow generates a single-task kernel directly accessing the off-chip memory and using four DSPs (Fig. 6.1b). Accessing the arrays via the DaCH cache library decomposes the kernel to use the CCC architecture (Fig. 6.1c), extracting the memory accesses to dedicated buffering tasks. SILVIA packs two pairs of 8-bit multiplications with a shared operand to two DSP (Fig. 6.1d). Finally, assuming that the computation task reaches a



(a) Source code. (b) SOTA HLS. (c) CCC. (d) SILVIA. (e) Task-level multi-pumping.

Figure 6.1: State-of-the-art (SOTA) high-level synthesis (HLS) tools translate the `scal` algorithm (a) to a single-task kernel that mixes off-chip memory accesses with computations, mapped to four digital signal processors (DSPs). Accessing the off-chip memory via the DaCH library decomposes the kernel according to the cache-compute-cache (CCC) architecture (c). SILVIA automatically packs the four 8-bit multiplications to two DSPs (d). Finally, the task-level multi-pumping double pumps the computation task, reducing the DSP utilization to one (e).

clock frequency twice higher than the cache tasks, the task-level multi-pumping double-pumps the computation task, further reducing the DSP utilization to one.

Experimental results prove that the techniques proposed in this dissertation enable more abstracted and higher-quality HLS for FPGAs. The next step is to integrate these techniques into SOTA HLS tools, to support the EDA industry's efforts toward widespread adoption of functional-level hardware design. Notably, this transition is already underway for data caching methodologies, as evidenced by their integration in recent versions of AMD Vitis HLS.

Moreover, additional research on the proposed techniques could further improve their effectiveness. For instance, the soft-core nature of the cache-based buffering tasks and the their customization fine-tuned to memory access patterns to specific arrays, pave the way to advanced address bit mappings, starting from the one in Fig. 3.9b, or replacement policies. In addition, integrating automated cache parameters selection methods [36] in the DaCH library would contribute to completely abstract the memory management in HLS. Further research directions for task-level multi-pumping include the exploration of applicability to other resource than DSPs and the exploitation of fractional multi-pumping factor, combining II with unrolling factors scaling.

Bibliography

[1] J. S. Vetter *et al.*, "Architectures for the post-Moore era," *IEEE Micro*, vol. 37, no. 4, pp. 6–8, Aug. 2017.

[2] G. Brignone *et al.*, "Array-specific dataflow caches for high-level synthesis of memory-intensive algorithms on FPGAs," *IEEE Access*, vol. 10, pp. 118 858–118 877, 2022.

[3] G. Brignone *et al.*, "SILVIA: Automated superword-level parallelism exploitation via HLS-specific LLVM passes for compute-intensive FPGA accelerators," *ACM Transactions on Reconfigurable Technology and Systems*, Nov. 2024.

[4] G. Brignone *et al.*, "A DSP shared is a DSP earned: HLS task-level multi-pumping for high-performance low-resource designs," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*, 2023, pp. 551–557.

[5] AMD, *Convolution example*, Jun. 2024. [Online]. Available: `https://docs.amd.com/r/en-US/Vitis-Tutorials-Hardware-Acceleration/Convolution-Example`.

[6] V. H. Allan *et al.*, "Software pipelining," *ACM Computing Surveys*, vol. 27, no. 3, pp. 367–432, Sep. 1995.

[7] J. Cong *et al.*, "FPGA HLS today: Successes, challenges, and opportunities," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, Aug. 2022.

[8] AMD, *Vitis high-level synthesis user guide*, Jul. 2024. [Online]. Available: `https://docs.amd.com/r/en-US/ug1399-vitis-hls`.

[9] Siemens EDA, *Catapult® synthesis HLS bluebook*, Apr. 2021. [Online]. Available: `https://resources.sw.siemens.com/en-US/e-book-high-level-synthesis-hls-blue-book`.

[10] Intel, *Intel® high level synthesis compiler pro edition reference manual*, Dec. 2022. [Online]. Available: `https://www.intel.com/content/www/us/en/docs/programmable/683349/22-4/pro-edition-reference-manual.html`.

[11] *Versal ACAP DSP engine architecture manual (AM004)*, AMD, Sep. 2022. [Online]. Available: `https://docs.amd.com/r/en-US/am004-versal-dsp-engine`.

[12] *UltraScale architecture DSP slice*, Xilinx, Aug. 2021. [Online]. Available: `https://docs.amd.com/v/u/en-US/ug579-ultrascale-dsp`.

[13] *Convolutional neural network with INT4 optimization on Xilinx devices*, Xilinx, Jun. 2020. [Online]. Available: `https://docs.amd.com/v/u/en-US/wp521-4bit-optimization`.

[14] Y. Fu *et al.*, *Deep learning with INT8 optimization on Xilinx devices*, Xilinx, Apr. 2017. [Online]. Available: `https://docs.amd.com/v/u/en-US/wp486-deep-learning-int8`.

[15] Q. Liu *et al.*, "SSiMD: Supporting six signed multiplications in a DSP block for low-precision CNN on FPGAs," in *2023 International Conference on Field Programmable Technology (ICFPT)*, 2023, pp. 161–169.

[16] J. Sommer *et al.*, "DSP-packing: Squeezing low-precision arithmetic into FPGA DSP blocks," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 160–166.

[17] J. Yang *et al.*, "ACane: An efficient FPGA-based embedded vision platform with accumulation-as-convolution packing for autonomous mobile robots," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 533–538.

[18] J. Zhang *et al.*, "Uint-packing: Multiply your DNN accelerator performance via unsigned integer DSP packing," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023.

[19] B. Hoff. "IDC field-programmable gate array market analysis." (Jun. 2024), [Online]. Available: `https://www.idc.com/getdoc.jsp?containerId=US52380724` (visited on 12/09/2024).

[20] Intel, *Variable-precision DSP in Intel Agilex® 5 FPGAs and SoCs*, Sep. 2024. [Online]. Available: `https://www.intel.com/content/www/us/en/docs/programmable/762191/current/variable-precision-dsp-in-fpgas-and-socs.html`.

[21] Y. Umuroglu *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2017, pp. 65–74.

[22] T. B. Preußer and T. A. Branca, "Vectorization of wide integer data paths for parallel operations with side-band logic monitoring the numeric overflow between vector lanes," 10 671 388, Jun. 2020.

[23] V. Sze *et al.*, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[24] J. de Fine Licht *et al.*, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1014–1029, 2021.

[25] *AMBA AXI and ACE protocol specification*, ARM, Sep. 2023. [Online]. Available: `https://developer.arm.com/documentation/ihi0022/k`.

[26] M. Adler *et al.*, "Leap scratchpads: Automatic memory and cache management for reconfigurable logic," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 25–28.

[27] J. Choi *et al.*, "Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 17–24.

[28] G. Jo *et al.*, "SOFF: An OpenCL high-level synthesis framework for FPGAs," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 295–308.

[29] E. Matthews *et al.*, "Design space exploration of L1 data caches for FPGA-based multiprocessor systems," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 156–159.

[30] F. Winterstein *et al.*, "Custom-sized caches in application-specific memory hierarchies," in *2015 International Conference on Field Programmable Technology (FPT)*, 2015, pp. 144–151.

[31] L. Ma *et al.*, "Acceleration by inline cache for memory-intensive algorithms on FPGA via high-level synthesis," *IEEE Access*, vol. 5, pp. 18 953–18 974, Sep. 2017.

[32] AMD, *Best practices for designing with M_AXI interfaces*, Jul. 2024. [Online]. Available: `https://docs.amd.com/r/en-US/ug1399-vitis-hls/Best-Practices-for-Designing-with-M_AXI-Interfaces`.

[33] L. Josipović *et al.*, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 127–136.

[34] J. Cheng *et al.*, "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, 2022.

[35] Xilinx, *PYNQ: Python productivity for adaptive computing platforms*, 2022. [Online]. Available: `https://pynq.readthedocs.io`.

[36] B. R. Upadhyay and T. S. B. Sudarshan, "Design space exploration of cache memory – a survey," in *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, 2016, pp. 2294–2297.

[37] Avnet, *Ultra96 hardware user's guide*, Mar. 2018. [Online]. Available: `https://www.avnet.com/opasdata/d120001/medias/docus/187/Ultra96-HW-User-Guide-rev-1-0-V0_9_preliminary.pdf`.

[38] AMD, *Vivado design suite user guide: Getting started (ug910)*, May 2024. [Online]. Available: `https://docs.amd.com/r/en-US/ug910-vivado-getting-started`.

[39] R. Bosio *et al.*, "LESS: Low-power energy-efficient subgraph isomorphism on FPGA," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024.

[40] J. Brandner *et al.*, "Multilayer multipurpose caches for OpenMP target regions on FPGAs," in *International Workshop on OpenMP*, Springer, 2024, pp. 79–93.

[41] C. Barone *et al.*, "Improving memory interfacing in HLS-generated accelerators with custom caches," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2023.

[42] J. Li *et al.*, "Firefly: A high-throughput hardware accelerator for spiking neural networks with efficient DSP and memory optimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.

[43] E. Luo *et al.*, "DeepBurning-MixQ: An open source mixed-precision neural network accelerator design framework for FPGAs," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023.

[44] R. Bosio *et al.*, "NN2FPGA: Optimizing CNN inference on FPGAs with binary integer programming," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[45] Y. Zhang *et al.*, "Wsq-addernet: Efficient weight standardization based quantized addernet FPGA accelerator design with high-density int8 DSP-LUT co-packing optimization," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022.

[46] F. S. F. Inc. "Auto-vectorization in GCC." (Feb. 2023), [Online]. Available: `https://gcc.gnu.org/projects/tree-ssa/vectorization.html` (visited on 05/06/2024).

[47] LLVM team. "Auto-vectorization in LLVM." (Jun. 2024), [Online]. Available: `https://llvm.org/docs/Vectorizers.html` (visited on 05/06/2024).

[48] N. B. Agostini *et al.*, "An MLIR-based compiler flow for system-level design and hardware acceleration," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, Association for Computing Machinery, 2022.

[49] H. Ye *et al.*, "ScaleHLS: A scalable high-level synthesis framework with multi-level transformations and optimizations: Invited," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, Association for Computing Machinery, 2022, pp. 1355–1358.

[50] W. Zuo *et al.*, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Association for Computing Machinery, 2013, pp. 9–18.

[51] LLVM team. "LLVM 3.1 release notes." (May 2012), [Online]. Available: `https://releases.llvm.org/3.1/docs/ReleaseNotes.html` (visited on 05/05/2024).

[52] A. Canis *et al.*, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2011, pp. 33–36.

[53] Xilinx, *HLS*, `https://github.com/Xilinx/HLS`, 2024.

[54] Xilinx, *Vitis HLS introductory examples*, `https://github.com/Xilinx/Vitis-HLS-Introductory-Examples`, 2024.

[55] F. Ottati, "Efficient deep learning inference: A digital hardware perspective-evaluating and improving performance and efficiency of artificial and spiking neural networks hardware accelerators," Ph.D. dissertation, Politecnico di Torino, 2024.

[56] AMD, *Vitis libraries*, 2024. [Online]. Available: `https://docs.amd.com/r/en-US/Vitis_Libraries`.

[57] Y. Hara *et al.*, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2008, pp. 1192–1195.

[58] R. Sarkar *et al.*, "FlowGNN: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 1099–1112.

[59] AMD, *Resource utilization for FIFO generator v13.2*, 2023. [Online]. Available: `https://download.amd.com/docnav/documents/ip_attachments/fifo-generator.html`.

[60] T. Liang *et al.*, "Hi-ClockFlow: Multi-clock dataflow automation and throughput optimization in high-level synthesis," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019.

[61] C. Lo and P. Chow, "Model-based optimization of high level synthesis directives," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[62] A. Sohrabizadeh *et al.*, "AutoDSE: Enabling software programmers to design efficient FPGA accelerators," *ACM Transactions on Design Automation of Electronic Systems*, vol. 27, no. 4, Feb. 2022.

[63] Q. Sun *et al.*, "Correlated multi-objective multi-fidelity optimization for HLS directives design," *ACM Transactions on Design Automation of Electronic Systems*, vol. 27, no. 4, 2022.

[64] Xilinx, *Merlin compiler*, `https://github.com/Xilinx/merlin-compiler`, 2022.

[65] A. Canis *et al.*, "Multi-pumping for resource reduction in FPGA high-level synthesis," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013, pp. 194–197.

[66] G. Lhairech-Lebreton *et al.*, "Hierarchical and multiple-clock domain high-level synthesis for low-power design on FPGA," in *2010 International Conference on Field Programmable Logic and Applications*, 2010, pp. 464–468.

[67] B. Ronak and S. A. Fahmy, "Multipumping flexible DSP blocks for resource reduction on Xilinx FPGAs," vol. 36, no. 9, pp. 1471–1482, Sep. 2017.

[68] O. Ragheb and J. H. Anderson, "High-level synthesis of FPGA circuits with multiple clock domains," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 109–116.

[69] AMD, Ed. "Double pumping DSP." (2019), [Online]. Available: `https://adaptivesupport.amd.com/s/question/0D52E00006iHilaSAC/double-pumping-dsp` (visited on 10/08/2024).

[70] AMD, *FIFO generator*, 2022. [Online]. Available: `https://www.xilinx.com/products/intellectual-property/fifo_generator.html`.

[71] Y. Zhou *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software-programmable FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 269–278.

[72] T. V. Aa *et al.*, *Virtual screening on FPGA: Performance and energy versus effort*, 2022. arXiv: `2210.10386 [cs.AR]`.