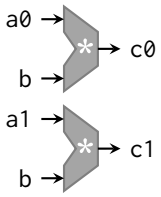
SILVIA: Automated Superword-Level Parallelism Exploitation via HLS-Specific LLVM Passes for Compute-Intensive FPGA Accelerators

SILVIA: Automated Superword-Level Parallelism Exploitation via HLS-Specific LLVM Passes for Compute-Intensive FPGA Accelerators

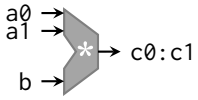
GIOVANNI BRIGNONE, Politecnico di Torino, Italy
ROBERTO BOSIO, Politecnico di Torino, Italy
FABRIZIO OTTATI, Politecnico di Torino, Italy
CLAUDIO SANSOÈ, Politecnico di Torino, Italy
LUCIANO LAVAGNO, Politecnico di Torino, Italy

```
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.

Fig. 1. Given a loop computing two multiplications in parallel (a), the standard high-level synthesis (HLS) flow generates the corresponding datapath (b) with two digital signal processor (DSP), 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.

High-level synthesis (HLS) aims at democratizing custom hardware acceleration with highly abstracted software-like descriptions. However, efficient accelerators still require substantial low-level hardware optimizations, defeating the HLS intent. In the context of field-programmable gate arrays, digital signal processors (DSPs) are a crucial resource that typically requires a significant optimization effort for its efficient utilization, especially when used for sub-word vectorization. This work proposes SILVIA, an open-source LLVM transformation pass that automatically identifies superword-level parallelism within an HLS design and exploits it by packing multiple operations, such as additions, multiplications, and multiply-and-adds, 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 state-of-the-art HLS designs such as convolutional neural networks and basic linear algebra subprograms accelerators, reducing the DSP utilization for additions by 70 % and for multiplications and multiply-and-adds by 50 % on average.

CCS Concepts: • **Hardware → Electronic design automation**; **High-level and register-transfer level synthesis**.

Additional Key Words and Phrases: HLS, FPGA, DSP, SIMD, LLVM, EDA

Authors' Contact Information: Giovanni Brignone, giovanni.brignone@polito.it, Politecnico di Torino, Turin, Italy; Roberto Bosio, roberto_bosio@polito.it, Politecnico di Torino, Turin, Italy; Fabrizio Ottati, fabrizio.ottati@polito.it, Politecnico di Torino, Turin, Italy; Claudio Sansoè, claudio.sansoe@polito.it, Politecnico di Torino, Turin, Italy; Luciano Lavagno, luciano.lavagno@polito.it, Politecnico di Torino, Turin, Italy.

Manuscript submitted to ACM

ACM Reference Format:
Giovanni Brignone, Roberto Bosio, Fabrizio Ottati, Claudio Sansoè, and Luciano Lavagno. 2024. SILVIA: Automated Superword-Level Parallelism Exploitation via HLS-Specific LLVM Passes for Compute-Intensive FPGA Accelerators. *ACM Trans. Reconfig. Technol. Syst.* 37, 4, Article 111 (August 2024), 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Digital hardware design is a complex and time-consuming process, mostly conducted at low abstraction levels using hardware description languages (HDLs). High-level synthesis (HLS) aims to simplify this process by enabling the designers to describe the hardware functionality via software programming languages, such as `C` and `C++`. The HLS compiler translates the design description to a corresponding register-transfer level (RTL) implementation. However, HLS designs still need hardware-aware manual optimizations to achieve high performance and efficiency.

In field-programmable gate array (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 state-of-the-art (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 parallelism of the FPGAs' digital signal processors (DSPs), unless the designer explicitly packs the inputs of the DSPs to efficiently utilize them. For instance, the AMD/Xilinx UltraScale DSP [22] slice supports the single-instruction multiple-data (SIMD) operating mode for additions. Moreover, ingenious DSP packing approaches [5, 10, 16, 21, 26, 28] enable other SIMD-like operations such as multiple multiply-and-adds (MADs) with shared operands.

Several SOTA FPGA designs optimized for performance using DSP packing [9, 11, 12, 29] prove the effectiveness of this technique. However, current HLS compilers and 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 full RTL implementations [9], or RTL implementation of the HLS functions modeling the vectorized operations [29], or mimicking the low-level RTL expressiveness in HLS via bit operations to explicitly map the inputs and outputs to the DSP ports [11, 12], disrupting the HLS abstractions.

The automatic vectorization of loops and basic blocks (BBs) is a well-established optimization typically implemented in software compilers targeting CPUs [7, 19]. 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 [5].

A multitude of studies [1, 27, 30] automate code transformations for improving quality of results (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. 1a computes two 8-bit multiplications with a shared operand in parallel. The SOTA HLS flow allocates two DSPs, one per multiplication (Fig. 1b). On the other hand, SILVIA automatically packs

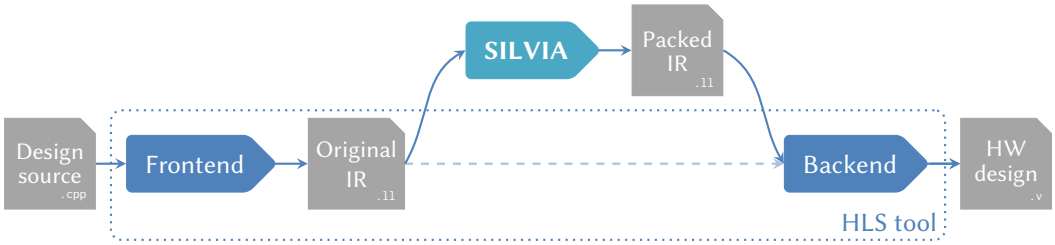


Fig. 2. The modified high-level synthesis workflow with SILVIA. SILVIA optimizes the original LLVM intermediate representation generated by the frontend for DSP-packed operations and provides it as input to the backend.

the two multiplications to a single DSP [5] (Fig. 1c) by analyzing and transforming the intermediate representation (IR) of the design.

The typical HLS flow consists of the frontend (FE) step, that translates the high-level design description into an optimized IR (e.g., from C/C++ to LLVM IR) and the backend (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. 2, because that point is accessible in most modern commercial and academic HLS flows for FPGAs.

SILVIA is based on LLVM [18], 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 [8], LegUp [4]) with minor adaptations.

The main contributions of this work are:

- The open-source[1]SILVIA framework, an LLVM transformation pass integrated in the AMD Vitis HLS flow that implements the generic functionality for packing multiple scalar operations within HLS source code to single DSPs on FPGAs and is extensible to support different operations.
- 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 results competitive with manually-optimized SOTA convolutional neural network (CNN) accelerators.

2 Background

SILVIA currently exploits the AMD/Xilinx UltraScale 48-bit [22] and Versal 58-bit [3] DSPs operation-packing capabilities. Among the multitude of proposed DSP packing methods [5, 10, 16, 21, 26, 28], SILVIA supports packing additions, subtractions, multiplications, and MADs. Nevertheless, SILVIA is designed to be easily extended to support other packed operations and DSP architectures, as explained in Section 3.

2.1 Additions and subtractions packing

[1]The SILVIA source code is available at github.com/brigio345/SILVIA.

The DSP architectures of the AMD/Xilinx UltraScale [22] and Versal [3] FPGA families support SIMD additions and subtractions. Specifically, they can sum (or subtract) four independent pairs of signed or unsigned operands on up to 12 bits (`four12` mode) or two independent pairs of signed or unsigned operands on up to 24 bits (`two24` mode).

2.2 Factor-2 multiply-and-adds packing

Fu et al. [5] proposed a methodology for computing two MADs of 8-bit operands, with one shared operand, on a single UltraScale/Versal DSP.

Specifically, if a_i and b_i are m-bits fixed-point numbers and c_i is an n-bits fixed-point number, $\forall i \in [1, N]$, N DSPs can compute

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

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

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

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

2.3 Factor-4 multiplications packing

The FINN framework [20] provides an open-source implementation of the architecture proposed by Preusser and Branca [14] 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 look-up table (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

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

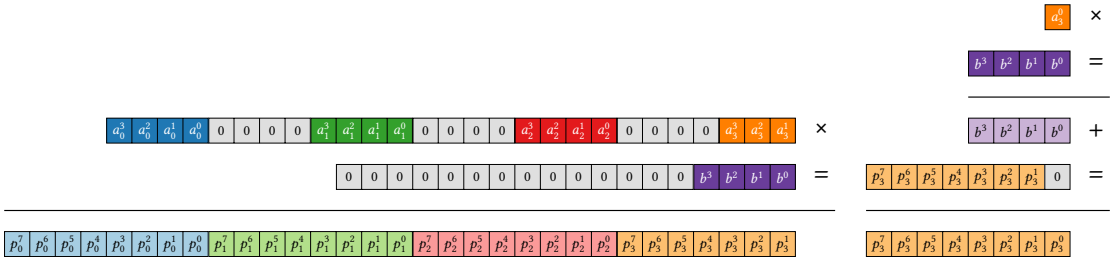
In the context of CNN accelerators [17], this packing is particular effective for the feature map reuse (i.e., the same activation multiplied by different signed weights). However, it does not support the filter reuse (i.e., the same weight multiplied by different activations, that are unsigned when the activation function is a rectified linear unit).

Therefore, this work supports 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. 3.

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 3a 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. 3b, computes the final product p_3 according to

$$p_3 = a_3 \cdot b = ((a_3^{[3:1]} \cdot b) \cdot 2) + (a_3^0 \cdot b). \tag{4}$$

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_0^0 is hardware-friendly



(a) DSP operations. (b) LUTs operations.

Fig. 3. The bit mapping of the proposed method for computing four multiplications between four 4-bit unsigned factors and one common 4-bit factor (signed or unsigned).

too (e.g., it can be implemented by the *and* operation between a_0^0 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. [5] (i.e., adding the MSB of a product p_i to the next product p_{i+1}).

3 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 [18] IR before passing it to the BE, as shown in Fig. 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 application programming interfaces (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 [18] 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 [23], 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: SILVIA's main optimization routine.

Data: BB = basic block belonging to an HLS design.
Result: BB^* = basic block functionally equivalent to BB and optimized for DSP-packed operations.
$C \leftarrow$ `getCandidates`(BB)
$BB^* \leftarrow BB$
`// Maximize the space for valid tuples.`
for $c \in C$ **do**
  $BB^* \leftarrow$ `moveUsesALAP`(c, BB^*)
`// Group the candidates in valid tuples.`
$\mathcal{T} \leftarrow$ `getTuples`(C)
`// Pack the valid tuples.`
for $T \in \mathcal{T}$ **do**
  $P \leftarrow$ `packTuple`(T)
  $BB^* \leftarrow$ `replaceTuple`(T, P, BB^*)

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 [19]. 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.

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:

- `SILVIAAdd`: four 12-bit additions (or subtractions) or two 24-bit additions (or subtractions), discussed in Section 2.1.
- `SILVIAMuladd`: two 8-bit MADs (or multiplications) or four 4-bit multiplications, discussed in Section 2.2.

3.1 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) amenable for 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.

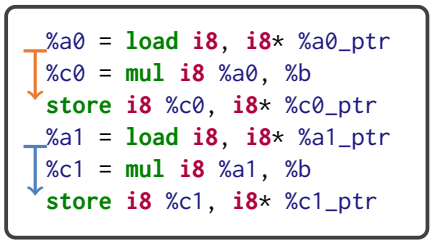
3.2 Tuple generation

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

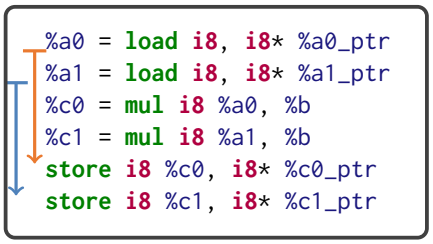
(a) whose candidates do not depend on each other,
(b) with an available insertion point between the first use of each candidate and after the last definition of each candidate's operands,
(c) that satisfy the constraints of a specific DSP-packed operation.

3.2.1 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 3.3). 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. 1a, which computes two parallel multiplications via loop unrolling, is compiled to the LLVM IR in Fig. 4a. 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



(a) Source code.



(b) ALAP-rearranged code.

```
%a0 = load i8, i8* %a0_ptr
%a1 = load i8, i8* %a1_ptr
%c = call {i8, i8} @silvia_mul(i8 %a0, i8 %a1, i8 %b)
%c0 = extractvalue {i8, i8} %c, 0
%c1 = extractvalue {i8, i8} %c, 1
store i8 %c0, i8* %c0_ptr
store i8 %c1, i8* %c1_ptr
```

(c) DSP-packed code.

Fig. 4. The C code defined in Fig. 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).

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. 4a) 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.

3.2.2 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 (1) and (3).

The `SILVIAAdd`'s `canPack` function does not perform any further check, since a SIMD DSP can compute any tuple of independent additions.

3.3 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 3.2.1), generating, for instance, the code in Fig. 4c.

The called function can:

- Directly implement the optimized DSP-packed module in the LLVM IR. For instance, `SILVIAMuladd` specifies the operations to be computed by the SIMD DSP such that the HLS tool maps them to a single DSP.
- 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 3.4. 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 [20].

The `packTuple` step totally depends on the specific packed operation. For instance, the `SILVIAMuladd` pass ensures that (2) 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.

3.4 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 HLS-generated RTL modules corresponding to the placeholder functions called by `packTuple` with the related custom DSP-packed RTL modules. Vitis HLS provides the "black box" functionality, that is similar but currently has too many limitations to be used for this purpose. Therefore, SILVIA re-implemented it.

3.5 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).

3.5.1 Impact on scheduling. The pipeline initiation interval (II) (i.e., the number of clock cycles between the start of successive pipeline iterations) is a key parameter of a schedule, since the throughput is inversely proportional to it. 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

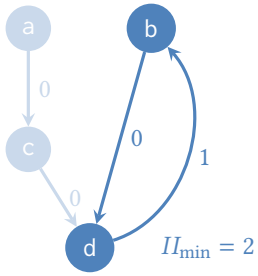
$$II_{\min} \triangleq \max_{\theta} \left\lceil \frac{\text{latency}_\theta}{\text{distance}_\theta} \right\rceil, \tag{5}$$

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 [2]. The cycle maximizing (5) is called the *critical cycle*.

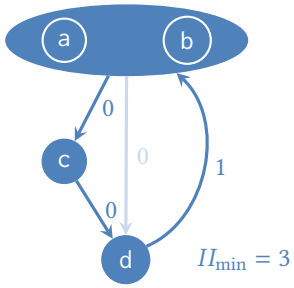
Figure 5 shows an example where packing increases the minimum II. The packing changes the original DDG (Fig. 5b), corresponding to the source code in Fig. 5a, to the DDG in Fig. 5c, where the candidates belonging to the packed tuple

```
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.

Fig. 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.

are merged into a single super-node, introducing a new critical cycle, due to the fictitious data dependencies between the candidates of the tuple.

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 suffered from it), the management of this corner case is left to future work.

3.5.2 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. Also this optimization is left to future work.

4 Results

SILVIA is integrated in the flow of the AMD Vitis HLS 2023.2 tool. To launch the SILVIA optimized flow (Fig. 2), designers just need to update the synthesis script by replacing the standard synthesis command (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. 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

```
+ 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
```

Fig. 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 custom `SILVIA::csynth_design` command.

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 N DSPs, where N is defined as (2). 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 basic linear algebra subprograms (BLAS) accelerators to complex high-performance open-source designs, as discussed in Section 4.1.

Section 4.2 analyzes the use case of CNN accelerators, specifically the SOTA frameworks NN2FPGA [12] and FINN [20]. 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.

4.1 General purpose 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 [24], summing two vectors of 192 8-bit elements.
- A spiking neural network (SNN) convolutional layer [13] 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 matrix-matrix multiplication (MMM) between two 192×192 matrices, and the `axpy` and `scal`

Table 1. Power, performance, and area of the benchmarks. The baseline DSP (BD) and baseline unconstrained (BU) results are obtained with the standard Vitis HLS flow. The SILVIA (S) results originate from the optimized SILVIA flow. Additionally, the BD and S designs forced Vitis HLS to bind the addition or multiplication operations to DSPs. The "←" signifies that Vitis HLS generated the same design for both BU and BD versions (i.e., it automatically bound every multiplication to a DSP). *Ops/Unit* is the average number of operations mapped to a single arithmetic unit. The area and power data derive from the reports of the Vivado implementation, with a clock frequency constraint of 250 MHz and the default synthesis and implementation settings. The maximum clock frequency is the highest at which post-routing timing is met, using steps of 25 MHz. 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)			$\mathbf{F}_{\mathbf{clk}}^{\mathbf{max}}$ (MHz)		
	BD,BU	S	BD	BU	S	BD	BU	S	BD	BU	S	BD	BU	S	BD	BU	S	BD	BU	S
vadd [24]	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 [13]	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)			$\mathbf{F}_{\mathbf{clk}}^{\mathbf{max}}$ (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 [25]	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 [25]	1.00	2.00	64	←	32	4.07	←	4.57	0.01	←	0.27	13.58	←	14.10	486	←	495	450	←	375
GSM [6]	1.00	1.58	41	←	24	0.80	←	0.98	0.03	←	0.04	0.78	←	1.33	333	←	324	350	←	350
RTM [25]	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 [15]	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

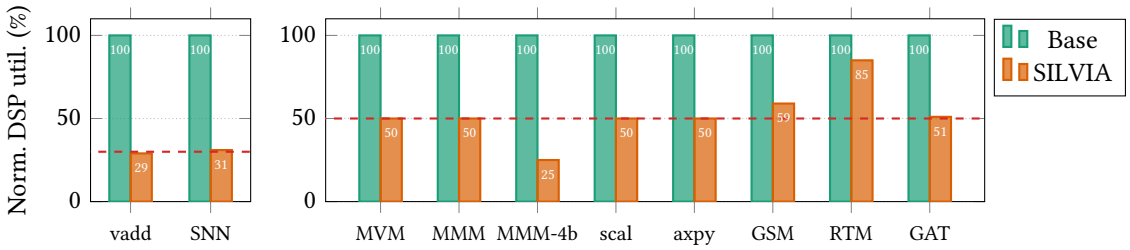
kernels on 512-element vectors from the Vitis Libraries [25]. In every case, the inputs are 8-bit integers. The MMM also includes a 4-bit unsigned integers configuration.

- The global system for mobile communications (GSM) kernel from the CHstone benchmark suite [6], with 8-bit words.
- The forward 3D hybrid boundary condition reverse time migration (RTM) accelerator from the Vitis Libraries [25], running on 8-bit fixed-precision data.
- The graph attention (GAT) graph neural network accelerator from FlowGNN [15], with 8-bit fixed-point data.

Each benchmark is implemented in different versions, including:

- 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.

Tables 1a and 1b report the power, performance, and area of the benchmarks, collected 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



(a) Addition-intensive benchmarks.

(b) Multiplication-intensive benchmarks.

Fig. 7. The DSP utilization of the different benchmarks, normalized on the baseline DSP utilization. The dashed horizontal line represents the geometric mean of the DSP utilization of the SILVIA-optimized designs.

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 defined as the ratio between the number of arithmetic operations and the number of functional units computing them, at the IR level.

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 [9, 11, 12, 29] 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 less 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 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 3.5 is never encountered). Therefore, SILVIA does not affect the throughput of the designs.

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 more pipelining registers. Consequently, SILVIA-optimized designs utilize more memory LUTs and flip-flops (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 (1). For instance, the axpy benchmark adds a third operand d_i to the products

Table 2. Convolutional neural network accelerators results. The resource utilization derives from Vivado post-implementation reports. The baseline results (B) are obtained synthesizing the designs without packing with the standard Vitis HLS flow, the manually-optimized (M) ones synthesizing the designs with manual packing with the standard Vitis HLS flow, and the SILVIA (S) ones synthesizing the designs without packing with the optimized SILVIA flow. The clock frequency constraint is 200 MHz for every design and timing is always met. The on-chip memory utilization is not reported because SILVIA does not impact it.

(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**

(i.e., $p_i = a_i \cdot c_i + d_i$), whereas the packed MAD can only sum together DSP-packed products (i.e., $p = a_i \cdot c_i + a_{i+1} \cdot c_{i+1}$). Therefore, 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 SILVIA-optimized designs 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.

4.2 CNN acceleration case study

NN2FPGA [12] and FINN [20] are open-source frameworks that generate the hardware description of CNN inference accelerators. These frameworks support the MAD packing supported by SILVIA (FINN supports both the factor-2 and the factor-4 packing; NN2FPGA the factor-2 only) in a user-controllable manner. Therefore, they are ideal for comparing the QoR of the manually-optimized designs (i.e., with the packing enabled) with the automatically SILVIA-optimized ones (i.e., with the packing disabled and optimized with the SILVIA flow).

Table 2 shows the post-implementation results at a clock frequency of 200 MHz generated by Vivado 2022.2 targeting

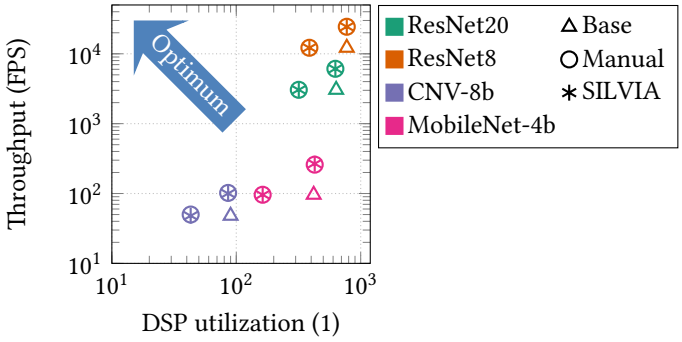


Fig. 8. The convolutional neural network inference accelerators case study in the post-implementation DSP utilization versus hardware-measured throughput space. SILVIA successfully matches the quality of results of the manually-optimized designs, both when optimizing for area and for resources.

the AMD Kria KV260 for the NN2FPGA designs and Vivado 2023.2 targeting the AMD ZCU102 for the FINN designs. The throughput is measured from hardware execution.

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. Also in this case, 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 fewer 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 a LUT-based adder.

In both experiments, the logic LUT and FF utilizations of the manually-optimized FINN designs is significantly lower 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 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.

5 Conclusion

This work proposes SILVIA, the first open-source LLVM infrastructure to automatically identify and optimize DSP-packable operations in HLS FPGA designs. SILVIA can pack 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 results comparable with manually optimized SOTA designs.

The experimental results prove that SILVIA is a further step towards higher abstraction levels for custom hardware acceleration, moving the hardware knowledge and code analysis from the designer's to the compiler's responsibility. This enables HLS source code more abstracted from hardware, resulting in cleaner and more maintainable code bases and making hardware acceleration more accessible to non-hardware-experts.

Acknowledgments

This work was partially supported by the Key Digital Technologies Joint Undertaking under the REBECCA Project with grant agreement number 101097224, receiving support from the European Union, Greece, Germany, Netherlands, Spain, Italy, Sweden, Turkey, Lithuania, and Switzerland. It was also partially supported by the Spoke 1 on Future HPC of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Mission 4 – Next Generation EU.

References

[1] Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Kaeli, and Antonino Tumeo. 2022. An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration. In *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. Association for Computing Machinery, New York, NY, USA, 1–9.

[2] Vicki H Allan, Reese B Jones, Randall M Lee, and Stephen J Allan. 1995. Software pipelining. *ACM Comput. Surv.* 27, 3 (9 1995), 367–432. https://doi.org/10.1145/212094.212131

[3] AMD 2022. *Versal ACAP DSP Engine Architecture Manual (AM004)*. AMD. https://docs.amd.com/r/en-US/am004-versal-dsp-engine.

[4] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, USA, 33–36.

[5] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. 2017. *Deep Learning with INT8 Optimization on Xilinx Devices*. Xilinx. https://japan.origin.xilinx.com/content/dam/xilinx/support/documents/white_papers/wp486-deep-learning-int8.pdf.

[6] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, Piscataway, NJ, USA, 1192–1195.

[7] Free Software Foundation Inc. 2023. *Auto-vectorization in GCC*. Free Software Foundation Inc. Retrieved June 6, 2024 from https://gcc.gnu.org/projects/tree-ssa/vectorization.html

[8] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, USA, 127–136.

[9] Jindong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng. 2023. 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* 31, 8 (2023), 1178–1191.

[10] Qi Liu, Mo Sun, Jie Sun, Liqiang Lu, Jieru Zhao, and Zeke Wang. 2023. SSiMD: Supporting Six Signed Multiplications in a DSP Block for Low-Precision CNN on FPGAs. In *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE, Piscataway, NJ, USA, 161–169.

[11] Erjing Luo, Haitong Huang, Cheng Liu, Guoyu Li, Bing Yang, Ying Wang, Huawei Li, and Xiaowei Li. 2023. 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)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1109/ICCAD57390.2023.10323831

[12] Filippo Minnella, Teodoro Urso, Mihai T. Lazarescu, and Luciano Lavagno. 2023. Design and Optimization of Residual Neural Network Accelerators for Low-Power FPGAs Using High-Level Synthesis. arXiv:2309.15631 [cs.AR]

[13] Fabrizio Ottati. 2024. *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.

[14] Thomas B. Preusser and Thomas A. Branca. 2020. Vectorization of wide integer data paths for parallel operations with side-band logic monitoring the numeric overflow between vector lanes. US Patent 10,671,388.

[15] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. 2023. FlowGNN: A Dataflow Architecture for Real-Time Workload-Agnostic Graph Neural Network Inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Piscataway, NJ, USA, 1099–1112.

[16] Jan Sommer, M. Akif Özkan, Oliver Keszocze, and Jürgen Teich. 2022. DSP-Packing: Squeezing Low-precision Arithmetic into FPGA DSP Blocks. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, Piscataway, NJ, USA, 160–166.

[17] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (2017), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

[18] LLVM team. 2012. *LLVM 3.1 Release Notes*. LLVM team. Retrieved May 5, 2024 from https://releases.llvm.org/3.1/docs/ReleaseNotes.html

[19] LLVM team. 2024. *Auto-Vectorization in LLVM*. LLVM team. Retrieved June 6, 2024 from https://llvm.org/docs/Vectorizers.html

[20] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 65–74.

[21] Xilinx 2020. *Convolutional Neural Network with INT4 Optimization on Xilinx Devices*. Xilinx. https://docs.amd.com/v/u/en-US/wp521-4bit-optimization.

[22] Xilinx 2021. *UltraScale Architecture DSP Slice*. Xilinx. https://docs.amd.com/v/u/en-US/ug579-ultrascale-dsp.

[23] Xilinx. 2024. HLS. https://github.com/Xilinx/HLS. Accessed: 2024-05-04.

[24] Xilinx. 2024. Vitis HLS Introductory Examples. https://github.com/Xilinx/Vitis-HLS-Introductory-Examples. Accessed: 2024-05-04.

[25] Xilinx. 2024. Vitis Libraries. https://github.com/Xilinx/Vitis_Libraries. Accessed: 2024-05-04.

[26] Jinho Yang, Sungwoong Yune, Sukbin Lim, Donghyuk Kim, and Joo-Young Kim. 2024. 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)*. Association for Computing Machinery, New York, NY, USA, 533–538.

[27] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: a scalable high-level synthesis framework with multi-level transformations and optimizations: invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. Association for Computing Machinery, New York, NY, USA, 1355–1358.

[28] Jingwei Zhang, Meng Zhang, Xinye Cao, and Guoqing Li. 2023. Uint-Packing: Multiply Your DNN Accelerator Performance via Unsigned Integer DSP Packing. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. Association for Computing Machinery, New York, NY, USA, 1–6.

[29] Yunxiang Zhang, Biao Sun, Weixiong Jiang, Yajun Ha, Miao Hu, and Wenfeng Zhao. 2022. 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*. Association for Computing Machinery, New York, NY, USA, 1–9.

[30] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. 2013. 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, New York, NY, USA, 9–18.

Received 30 June 2024; revised 30 September 2024; accepted 13 November 2024