



High-efficiency Compressor Trees for Latest AMD FPGAs

KONSTANTIN J. HOßFELD, AMD Research, Dresden, Germany

HANS JAKOB DAMSGAARD, Tampere University, Tampere, Finland and AMD Research, Dresden, Germany

JARI NURMI, Tampere University, Tampere, Finland

MICHAELA BLOTT, AMD Research, Dublin, Ireland

THOMAS B. PREUßER, AMD Research, Dresden, Germany

High-fan-in dot product computations are ubiquitous in highly relevant application domains, such as signal processing and machine learning. Particularly, the diverse set of data formats used in machine learning poses a challenge for flexible efficient design solutions. Ideally, a dot product summation is composed from a carry-free compressor tree followed by a terminal carry-propagate addition. On FPGA, these compressor trees are constructed from generalized parallel counters whose architecture is closely tied to the underlying reconfigurable fabric. This work reviews known counter designs and proposes new ones in the context of the new AMD Versal™ fabric. On this basis, we develop a compressor generator featuring variable-sized counters, novel counter composition heuristics, explicit clustering strategies, and case-specific optimizations like logic gate absorption. In comparison to the Vivado™ default implementation, the combination of such a compressor with a novel, highly efficient quaternary adder reduces the LUT footprint across different bit matrix input shapes by 45% for a plain summation and by 46% for a terminal accumulation at a slight cost in critical path delay still allowing an operation well above 500 MHz. We demonstrate the aptness of our solution at examples of low-precision integer dot product accumulation units.

CCS Concepts: • **Hardware** → **Reconfigurable logic and FPGAs; Reconfigurable logic applications; Programmable logic elements;** • **Mathematics of computing** → **Arbitrary-precision arithmetic;**

Additional Key Words and Phrases: Compressor tree, matrix compression, parallel counters

ACM Reference Format:

Konstantin J. Hoßfeld, Hans Jakob Damsgaard, Jari Nurmi, Michaela Blott, and Thomas B. Preußer. 2024. High-efficiency Compressor Trees for Latest AMD FPGAs. *ACM Trans. Reconfig. Technol. Syst.* 17, 2, Article 30 (April 2024), 32 pages. <https://doi.org/10.1145/3645097>

AMD, Versal, UltraScale, UltraScale+, Vivado, and combinations thereof are trademarks of Advanced Micro Devices, Inc. H. J. Damsgaard and J. Nurmi gratefully acknowledge funding by the European Union's Horizon 2020 Research and Innovation Program under the Marie Skłodowska Curie Grant Agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation, <http://apropos-itn.eu/>).

Authors' addresses: K. J. Hoßfeld and T. B. Preußer, AMD, c/o Regus Altmarkt, Altmarkt 10 b/d, 01067 Dresden, Germany; e-mails: konstantin.hossfeld@amd.com, thomas.preusser@amd.com; H. Jakob Damsgaard, Tampere University and J. Nurmi, Tampere University, Faculty of Information Technology and Communication Sciences, Korkeakoulunkatu 1, 33720 Tampere, Finland; e-mails: hans.damsgaard@tuni.fi, jari.nurmi@tuni.fi; M. Blott, AMD, 2020 Bianconi Avenue, Citywest Campus, Dublin, D24 T683; e-mail: michaela.blott@amd.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1936-7406/2024/04-ART30

<https://doi.org/10.1145/3645097>

1 INTRODUCTION

Field-programmable gate array (FPGA) fabric provides hardware customization to time-multiplexed as well as to low- and medium-volume application deployments in the data center, at the edge, and in embedded systems. As such, it is becoming a tightly integrated component of heterogeneous systems, be it (a) embedded **Systems-on-a-Chip (SoCs)**, like Zynq devices, or (b) data center accelerator cards, like the Versal™ VCK5000. This trend is expected to continue as application diversity and throughput demands grow [4].

Arithmetic computation in FPGA fabric is granted designated structural support in the form of carry chains and DSP slices. They enable an efficient implementation of standard arithmetic (i.e., addition and multiplication, respectively). Commercial hardware synthesis tools, like Vivado™, ship with extensive IP core libraries that supply customizable implementations of higher-level arithmetic operators. Also, the academic FloPoCo project [8] is dedicated to the generation of basic and complex custom arithmetic operators spanning a wide range of integer and floating-point data types.

Compressor tree generation is a popular field of FPGA arithmetic. Compressor trees are used for the fast, carry-free reduction of sums into a value-maintaining representation comprising fewer addends. They are key to the efficient implementation of multi-operand addition, multiplication, and dot product accumulation. Thus, the capability to generate efficient compressor trees is particularly relevant for compute-heavy machine learning workloads. This field does not only heavily rely on efficient, high-throughput dot product computations but also increasingly explores various low-precision custom data types. Medium-precision eight-bit integer formats, which are commonly supported by CPUs and GPUs, have consistently been proposed and applied for inference [23, 31]. The feasibility of neural network quantizations to integer formats of as few as a single bit have been confirmed consistently over the past few years [33]. The corresponding full-custom inference solutions integrating such very-low precision or even binarized arithmetic traditionally rely heavily on FPGA fabric [3, 21, 22].

Beyond machine learning, high-performance dot products are at the heart of **generalized matrix-matrix multiplication (GEMM)** algorithms. These are frequently used in data center and high-performance computing applications, such as, graph analytics, data mining, optimization, cryptography, and computational physics [4]. Also in this domain, FPGA fabric proves useful as it permits high degrees of parallelism [15] and the exploitation of sparsity [10].

The classic ancestor of the compressor approach is the Wallace multiplier that consecutively applies carry-save reductions to the produced matrix of partial products [27]. Each reduction leverages full adders to rewrite a sum of three into a sum of two addends as shown in Figure 1. With all full adders operating mutually independent and in parallel, each reducing a column of three equal-weight bits into a two-bit count, the entire reduction step is performed with a constant delay of a single full adder irrespective of the size of the input. Wallace's original approach was promptly followed by Dadda's generalization [7], which practices a loose instantiation of full adders based on the next achievable row reduction goal considering the bit matrix in its entirety rather than its apriori slicing into groups of three addend rows. Interpreting the full adder as a counter of asserted input bits and generalizing this concept to input shapes spanning an arbitrary number of columns of different bit weights gives rise to the notion of **generalized parallel counters (GPCs)**. For these elementary building blocks of compressor trees, we adopt the established notation of $(p_{m-1}, \dots, p_0 : q_{n-1}, \dots, q_0)$ specifying the right-aligned number of bits in each of the m input and n output columns, respectively. A valid implementation of a GPC with inputs $x_{i,j}$ and outputs $y_{i,j}$ maintains the following invariant:

$$\sum_{i=0}^{m-1} \left(2^i \cdot \sum_{j=0}^{p_i-1} x_{i,j} \right) = \sum_{i=0}^{n-1} \left(2^i \cdot \sum_{j=0}^{q_i-1} y_{i,j} \right).$$

In the frequent case that the output shape of a GPC is a single row, we replace the resulting sequence of all $q_i = 1$ by its length. Thus, a full adder will be written as $(3 : 2]$ rather than $(3 : 1, 1]$. Carefully note that this convention does not introduce ambiguity as a counter performing any compression cannot produce an output shape comprising a single column.

This work advances the field of compressor tree generation for FPGAs. It leverages a heuristic implemented in Chisel [2] to quickly construct and evaluate compressors for a given input shape. The fitness of the produced designs is established against results previously reported for legacy architecture generations. The generator tool is then utilized to explore and evaluate the obstacles and opportunities posed for the adoption of known and the creation of new GPC designs when targeting the recent AMD Versal™ **configurable logic block (CLB)** architecture [1]. Its fabric has undergone major changes in comparison to the preceding 7 Series [29] and UltraScale+™ [30] generations, particularly, with respect to the structural implementation of the carry chain.

The remainder of this article is structured as follows: Section 2 presents related work on compressor tree generation and custom dot product implementations for FPGA fabric. Section 3 covers our library of existing and newly proposed GPCs, while Section 4 presents the tailoring of our compressor construction heuristic for Versal™ devices. Sections 5 and 6 explore optimizations by explicit LUT clustering and gate absorption for in-context compressors, respectively. Section 7 explores the extension of the plain bit matrix summation to the continuous accumulation of input matrices. Finally, Section 8 evaluates available algorithmic design options against one another and also compares the attained results to previously established work. Section 9 concludes the article and outlines future work.

2 RELATED WORK

The compressor tree generation specifically for FPGAs developed significant momentum with the work by Parandeh-Afshar et al. [16–18]. They propose GPCs that map favorably to the available FPGA fabric and regularly leverage the high-speed carry chain. This set of GPC designs has been extended continuously by other groups. Brunie et al. [5] introduced additional, purely LUT-based GPCs in the context of their bit heap compression implemented in FloPoCo. Kumm and Zipf [13, 14] propose further 4-column GPCs that map directly to the slice architecture of the contemporary 7 Series devices exploiting the internal carry chain. They also propose an efficient mapping of 4:2-adders to the same fabric. Preußner [19] describes a valuable LUT-based $(2, 5 : 1, 2, 1]$ counter, organizes the slice-based GPCs by identifying systematic compositions from pairs of two-column *atoms* among them, and extends the set of metrics to evaluate GPCs. Further GPCs particularly aiming at an increased area efficiency were later introduced by Yuan et al. [32], who contribute a highly efficient $(0, 6)$ atom, which supports an additional carry-in at the beginning of a slice.

While an apt GPC library is a critical prerequisite, it takes an algorithm to construct the actual compressor from the available components. Parandeh-Afshar et al. relied on a construction heuristic introduced early in their work [16]. Kumm and Zipf introduced the formulation of the compressor generation as an **integer linear programming (ILP)** problem [14], which they hand to a standard ILP solver. Preußner [19] adopts a heuristic approach similar to the one that had been described by Parandeh-Afshar et al. exploring different fitness metrics used in the GPC selection. Kumm and Kappauf [12] later propose a hybrid approach to mitigate the long run times of ILP solvers by a heuristic breakdown of the generation task into small subproblems. Yuan et al. [32], finally, opt for a pure ILP formulation again. So, both heuristics and ILP-based solutions are well established in the field. They represent different workflow trade-offs. While ILP solvers determine

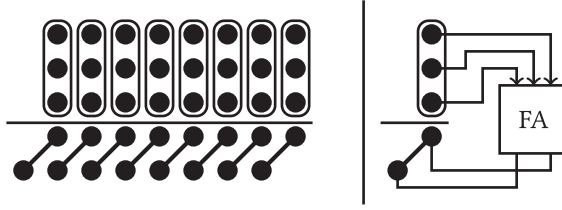


Fig. 1. Dot diagram of a carry save adder.

a solution that is optimal with respect to the problem formulation, heuristics construct a solution fast enough to be integrated into the regular design flow.

Significant research effort has been spent on highly efficient dot product implementations for the neural network acceleration on FPGAs. Some work uses the hardened DSP slices that are part of all modern AMD FPGA fabrics. Yet, these slices are scarce (Versal™ devices integrate roughly one DSP slice per two hundred LUTs) and inefficient if used naively on ultra-low precision formats or without custom operand packing schemes [11, 25]. Other authors implement custom adder trees in LUTs, specifically for convolutional neural network inference. Prost-Boucle et al. [20] focus on networks with ternary weights and activations and propose a collection of adders optimized for six-input LUTs. Their architecture further saves area by fusing multiplications and additions inside the neuron hardware. A similar strategy is examined by Véstias et al. [24], who propose an optimized dot product unit that merges parallel product lanes into fused multiply-add modules and explore design points combining it with DSP slices. In a prior work, Véstias et al. [26] have proposed parallel dot product units specifically tailored to the primitives available on previous AMD FPGA devices. These could be configured to operate with 2- or 8-bit weights. A common characteristic of this prior work is a focus on optimizations for particular operand sizes, which renders the proposed designs inflexible with respect to other choices of data types. Our approach seeks to resolve this inflexibility.

This work explores and rebuilds the set of known GPC designs for the latest update of AMD FPGA fabric architectures transitioning from UltraScale+™ to Versal™ devices. For the economic evaluation of compressor solutions built from this updated library, we opt for a heuristic generator implemented in Chisel. We thoroughly evaluate algorithmic design options and showcase the generator by constructing optimized parallel integer multiply-accumulate units.

3 COUNTER LIBRARY

3.1 Metrics and Notation

Typical criteria for the evaluation of compressors are area cost (i.e., the LUT count on an FPGA) and combinational delay. A heuristic compressor construction aiming at optimizing these criteria requires some metrics of fitness to guide the selection of GPC as building blocks in the process. We adopt two metrics: *Efficiency*, as defined by Parandeh-Afshar [18], and *strength*, as defined by Preußner [19], shown in Definitions 1 and 2. These metrics are based on the physical characteristics of a GPC ($p_{m-1}, \dots, p_0 : q_{n-1}, \dots, q_0$):

$$p = \sum_{i=0}^{m-1} p_i \quad \text{—total number of inputs,} \quad (1)$$

$$q = \sum_{i=0}^{n-1} q_i \quad \text{—total number of outputs,} \quad (2)$$

$$k \quad \text{—number of occupied LUTs.} \quad (3)$$

Table 1. Notation for Counter Inputs and Outputs

Counter	Symbol	Value
Output	s	$\sum_i 2^i s_i$
Carry	t_i	$2^i t_i$
Inputs	Latin	a, b, c, d
	Greek	α, β, \dots
		$\sum_i 2^i a_i, \quad \sum_i 2^i b_i, \quad \dots$
		$\sum_i 2^{\text{ord}(\alpha)} \alpha_i, \quad \sum_i 2^{\text{ord}(\beta)} \beta_i, \quad \dots$

Definition 1. The *efficiency* E of a GPC is the quotient of the reduction in the number of bit signals and the number of LUTs it occupies

$$E = \frac{p - q}{k}.$$

Prioritizing more efficient GPCs aims at minimizing the LUT cost in relation to the achieved problem reduction.

Definition 2. The *strength* S of a GPC is the ratio of its input bit count versus its output bit count

$$S = \frac{p}{q}.$$

The strength captures the asymptotic height reduction of a large bit matrix when exclusively using this specific GPC in a single compression step. Prioritizing stronger GPCs aims at minimizing the number of overall compression steps. Assuming GPCs with a single LUT level (i.e., without internal general-purpose routing paths) this metric allows optimizing for a small critical delay in combinational compressors and for a small latency of pipelined implementations.

We use two distinct notations to identify the inputs and outputs of counters. These notations are presented in Table 1 for reference. The function *ord* returns the position of the input letter in the alphabet (e.g., $\text{ord}(\alpha) = 0$). In situations with multiple carry or output signals of the same weight, they will be differentiated by priming as for t'_0 and t''_0 . Observe that a specific Latin letter subsumes the value on signals within a row, whereas a Greek letter subsumes the value of signals within a column.

3.2 From 7 Series to UltraScale+™

Figure 2 depicts the LUT architecture of 7 Series and UltraScale+™ devices. Each 6-LUT can be decomposed into two 5-LUTs with shared inputs. The LUTs are vertically grouped into slices. A carry chain designated to the fast propagation of carries through a slice amounts to the logic equivalent of an XOR gate and a multiplexer for each LUT. The input controlling the multiplexer can be configured to feed from the LUT bypass input Ax or from the $O5$ output of the associated LUT.

Tables 2 and 3 reproduce the counters as categorized by Preußner [19] for 7 Series devices. The slice counters in Table 2 are composed from atoms. These are building blocks stretching across two LUTs adjacent on the carry chain. The carry chain is also used to link atoms arbitrarily to fill a slice. Each atom can process a numeric weight of, at most, 7, which is distributed across two sum and one carry output. Within these constraints, there are no more atoms beyond the three used in the shown compositions. Their input signatures are (2, 2), (1, 4), and (0, 6). The (2, 2) atom can be interpreted as a 2-bit slice of a standard ripple-carry adder implementation. The others substitute individual weight-2 inputs by two weight-1 inputs each. The details of the (0, 6) atom will be discussed later. Atoms in slice-initial position have their carry inputs exposed rather than linked. Thus, their contributions to the resulting counter signatures increment as shown in the table headings. Note that the ability to expose the carry input of the (0, 6) atom

Table 2. Whole-slice Counter Compositions [19]

Atoms	$(\dots, 2, 3)$		$(\dots, 1, 5)$		$(\dots, 0, 7)$ §	
	E	S	E	S	E	S
$(2, 2, \dots)$	1	1.8^\dagger	1.25	2	1.5	2.2
$(1, 4, \dots)$	1.25	2	1.5	2.2^\ddagger	1.75	2.4^\ddagger
$(0, 6, \dots)$	1.5	2.2	1.75	2.4	2	2.6^\ddagger

[†]Standard 4-bit ripple-carry adder (RCA).

[‡]Counters originally proposed by Kumm and Zipf [13, 14].

[§]Counters further improved by Yuan et al. [32].

Table 3. Monolithic Slice Counters and Floating LUT Counters

	Counter	E	S
Slice	$(1, 3, 2, 5 : 5]$	1.25	2
	$(3 : 2]$	1	1.5
Floating	$(6 : 3]$	1	2
	$(2, 5 : 1, 2, 1]$	1.5	1.75
	$(10 : 4, 2]$	$\frac{4}{3}$	$\frac{5}{3}$

externally is an enhancement by Yuan et al. [32], which yields the stated improved figures of merit.

On UltraScale+ devices, a slice comprises eight rather than four LUTs. Its carry chain can be partitioned after four LUTs so that it can trivially accommodate two of the slice counters proposed for the 7 Series architecture. The ternary adder and the flexible compression goal proposed by Preußner [19] for the terminal addition are also compatible with these architectures.

3.3 Versal™ Device Adaptations

Versal™ devices introduced significant changes to the CLBs. While LUTs on UltraScale+ devices and previous generations could only be connected through general-purpose routing, Versal devices allow adjacent LUTs to be chained through a fast, slice-internal cascade path. Additionally, the carry chain logic is consolidated into a carry-lookahead primitive, which is solely designated to the acceleration of the carry propagation. The sum bit computation, which was granted a designated XOR gate in previous architectures, must be accommodated within the LUTs on Versal devices. The local carry input needed by this computation is routed from the lookahead primitive to the LUT along the new cascade paths.

Figure 3 depicts the Versal LUT architecture. Its O6 output can be forwarded along the cascade link. The top and bottom cascade multiplexers can be configured independently. A carry signal of a binary word addition can be further accelerated by the slice's lookahead module shown in Figure 4. In this setting, each six-input LUT computes a propagate signal p_i in one of its four-input sub-LUTs. The LUT also computes the equivalent of a local carry generate signal g_i with some structural differences between even and odd LUT locations. For an efficient implementation of adders exceeding the size of a slice, designated carry paths link the lookahead modules of neighboring slices.

Throughout the article, we illustrate counters according to the legend in Figure 5. Full adders take two addend bits from the top and a carry input from the right. They produce a sum bit at the bottom and a carry output on the left. LUTs take up to six inputs. Those fed from general-purpose

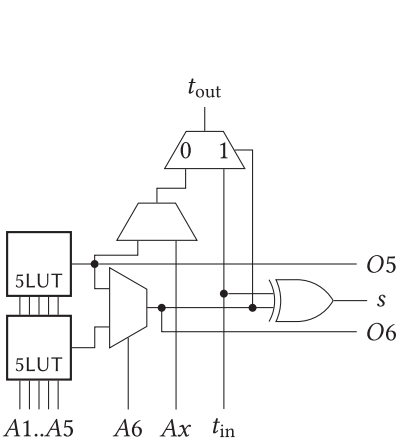


Fig. 2. 7 Series LUT.

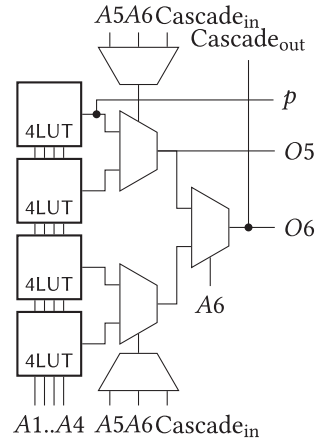


Fig. 3. Versal™ LUT.

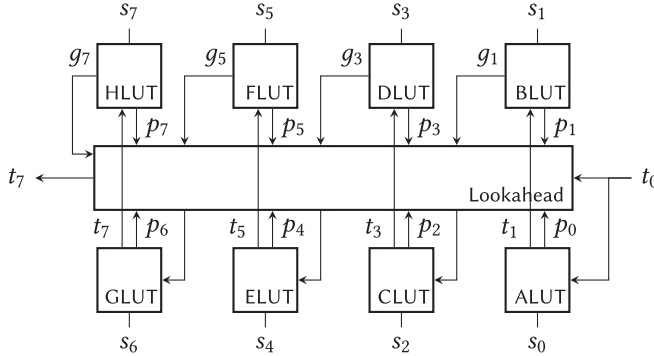


Fig. 4. Versal™ slice architecture with LOOKAHEAD8 module.

routing enter at the top. At most one input may come from the right indicating that a direct cascade link is used. Such an input must be produced on the O6 output of the preceding LUT.

Figure 6 contrasts the critical path delays of a signal propagated through lookahead primitives versus a signal forwarded along the LUT cascade links as measured in a register sandwich. The test setup is described in more detail in Section 8. For LUT cascades shorter than four LUTs, the lookahead module does not offer a measurable advantage. From there, however, the critical path delays grow with significantly different slopes putting the plain LUT cascade at a clear disadvantage for longer signal chains.

When utilizing the lookahead module, it is not feasible to initiate a new carry chain at an arbitrary position. However, when using the LUT cascade, this is, indeed, possible. The lookahead module receives all its input signals directly from the LUTs in its slice. The LUT bypass paths that were able to feed into each position of the carry chain directly from general-purpose routing in preceding architectures were abolished.

Tables 2 and 3 identify which counters are compatible with Versal devices. While both the (1, 4) and the (2, 2) atoms are, the strong and efficient (0, 6) atom shown in Figure 7 is no longer as the sum output of its right LUT is a function of seven individual inputs. Similarly, the *monolithic* non-decomposable (1, 3, 2, 5 : 5] slice counter proposed by Kumm and Zipf [13] is incompatible with the Versal fabric. All floating counters, which do not utilize the carry chain, are unaffected by

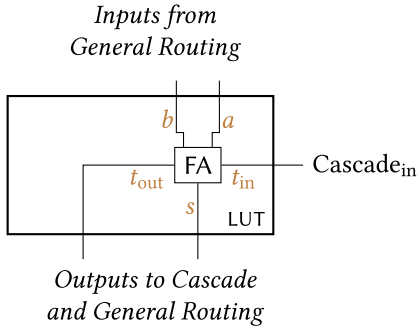


Fig. 5. Legend for Versal™ LUT.

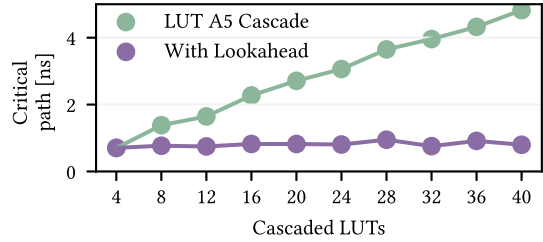


Fig. 6. Critical paths of a LUT Cascade vs. a Lookahead propagation. Results obtained with Vivado™ 2023.1 targeting a VCK190 (Versal™) evaluation board.

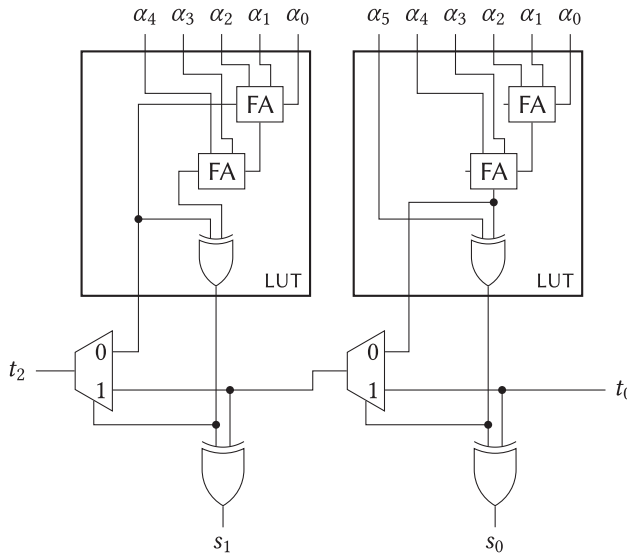


Fig. 7. (0,6) Atom.

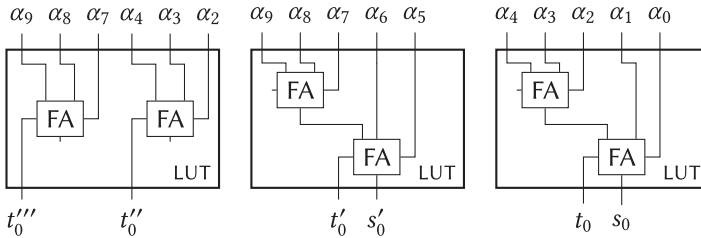


Fig. 8. (10 : 4, 2] counter.

the architectural changes and remain feasible. Figure 8 depicts a novel (10 : 4, 2] floating counter that can compress a single-input column very efficiently. Two LUTs each comprise two full adders internally cascaded through the sum bit. Only the sum and carry outputs of the second full adder in this chain can be emitted through the available LUT pins. The omitted carries of the first full

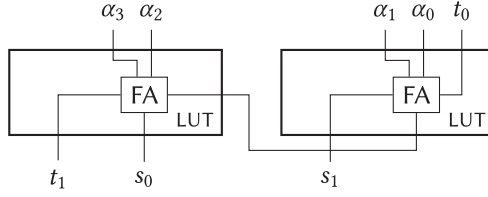


Fig. 9. Ripple-sum counter.

adders are reproduced in a third LUT, which completes the computation. This LUT exploits the independent control of the Versal cascade multiplexers to enable the computation of two separate three-input function. On preceding architectures, these two functions would have to share, at least, one common input.

On 7 Series and UltraScale+ devices, slice counters use the carry chain to propagate a carry signal. On Versal devices, this same propagation can be achieved using either the lookahead module or the plain LUT cascade. Recalling the extremely low signal delay penalty along Versal lookahead paths from Figure 6 and the flexible arbitrary partitioning of the LUT cascade without extra resource cost encourages abolishing the strict confinement of these counters to individual slices. In consequence, we propose the generalized concept of a *row GPC* comprising all counters that are assembled from atoms linked up along a carry chain regardless of their particular width. For even greater flexibility, we reduce the two-column (2, 2) atom to a single-column (2) atom. This generalizes this class of counters to include new members like the (1, 4, 2 : 4] counter.

The LUT cascade path introduced by Versal is decisively different from the lookahead primitive. While the carry propagated by the lookahead is a switched signal, the cascade input to the next LUT is a switching control. This enables the construction of fast cascades by forwarding signals of different semantics. Concretely, we propose the new class of *column GPCs* that chain vertically along the sum signals within a column. Assembling full adders this way yields the *ripple-sum counter* depicted in Figure 9. With each stage producing a weight-2 carry and only the last one leaving a weight-1 sum, its signature becomes $(2n + 1 : n, 1]$ for n stages. It has the same efficiency and strength as a standard ripple carry adder.

Our third proposal, the *dual-rail ripple-sum counter*, is illustrated in Figure 10. It is based on cascading the LUTs comprising individual $(2, 5 : 1, 2, 1]$ counters along their lower-weight outputs. Its input shape also grows vertically, with the signature $(n + 1, 4n + 1 : n, n + 1, 1]$. With an efficiency of 1.5 and a strength approaching 2.5 for long cascades, the dual-rail ripple-sum counter compares favorably in terms of both efficiency and strength.

We extend the previously proposed atoms by a novel (2, 2, 2) atom shown in Figure 11. This atom accommodates a 3-bit slice of a ripple-carry addition within only two adjacent LUTs, matching the efficiency of the (1, 4) atom. The right LUT of this atom calculates the sum bits s_0 and s_1 . Besides being an output, s_1 is also copied to the next LUT using the direct cascade link available in Versal. There, it is combined with the inputs b_1 and a_1 to reconstruct carry t_1 and compute carry t_2 :

$$\begin{aligned} s_1 &= (b_1 \oplus a_1) \oplus t_1, \\ t_1 &= (b_1 \oplus a_1) \oplus s_1, \\ t_2 &= b_1 a_1 + (b_1 \oplus a_1) \cdot t_1. \end{aligned}$$

This allows the second LUT to compute both the third sum bit s_2 and the carry t_3 . Since the latter is a conventional carry, this atom can be seamlessly cascaded with any other atom. However, note that this atom cannot leverage the lookahead module for fast signal propagation, since the

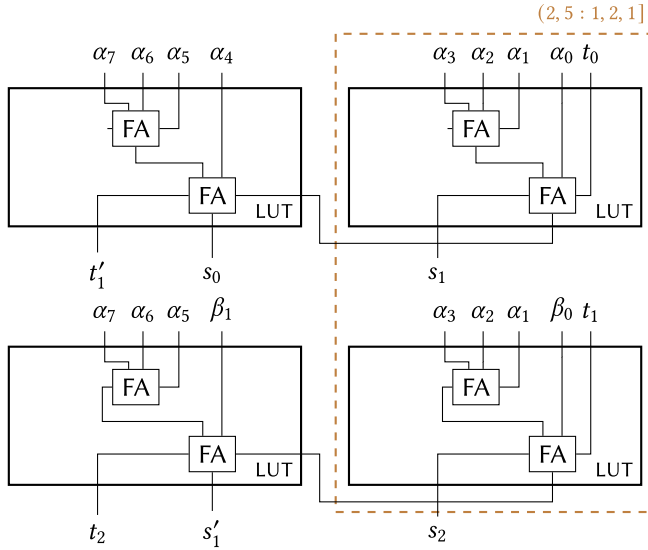


Fig. 10. Dual-rail ripple-sum counter.

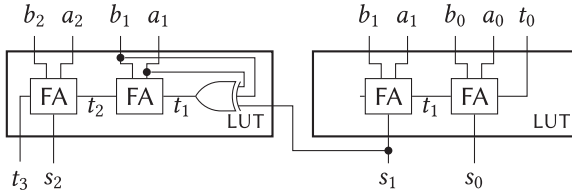


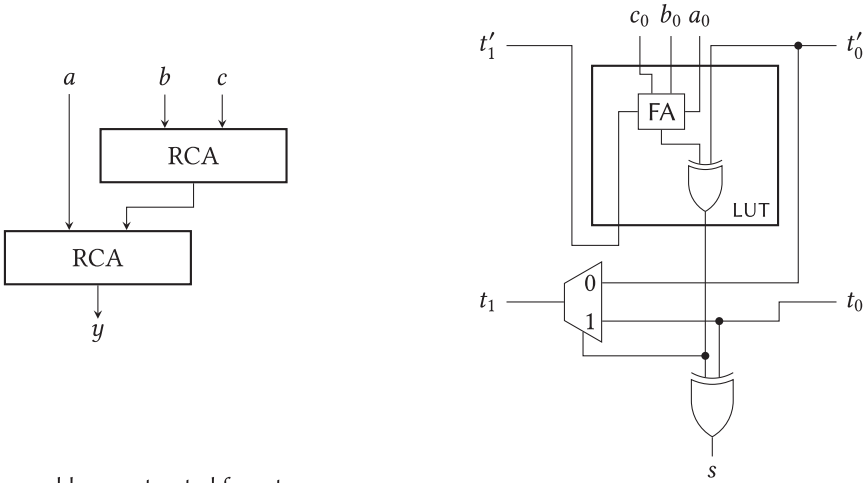
Fig. 11. (2, 2, 2) Atom.

Table 4. Variable-size Row and Column Counters on Versal™

Counter	E	S	Lookahead
(2) Atom Chain (Ripple-carry)	1	$2 - \frac{1}{n+1}$	✓
(1, 4) Atom Chain	1.5	$2.5 - \frac{3}{2n+2}$	✓
Ripple-sum	1	$2 - \frac{1}{n+1}$	✗
Dual-rail Ripple-sum	1.5	$2.5 - \frac{3}{2n+2}$	✗
(2, 2, 2) Atom Chain (Ripple-carry)	1.5	$2 - \frac{1}{n+1}$	✗

cascaded signal s_1 is not a conventional carry. The design of this atom rather relies on the more general LUT cascading capability in Versal.

Table 4 summarizes the performance metrics of the variable-size row and column counters proposed for the Versal architecture. Note that the row counter atoms can be mixed naturally producing solutions with metrics following a weighted average. Cascading a ripple-sum counter with its dual-rail counterpart is possible as well. Their arbitrary mixing does not add any benefit though as the ordering within a column does not affect the consumed input shape. Column counters cannot utilize the lookahead module and are therefore more likely than row counters to require the definition of size limits. Such limits mitigate the effect on the critical path by the higher delay penalty of



(a) Ternary adder constructed from two ripple carry adders.

(b) An element of a ternary adder in the 7 Series fabric.

Fig. 12. Ternary adder.

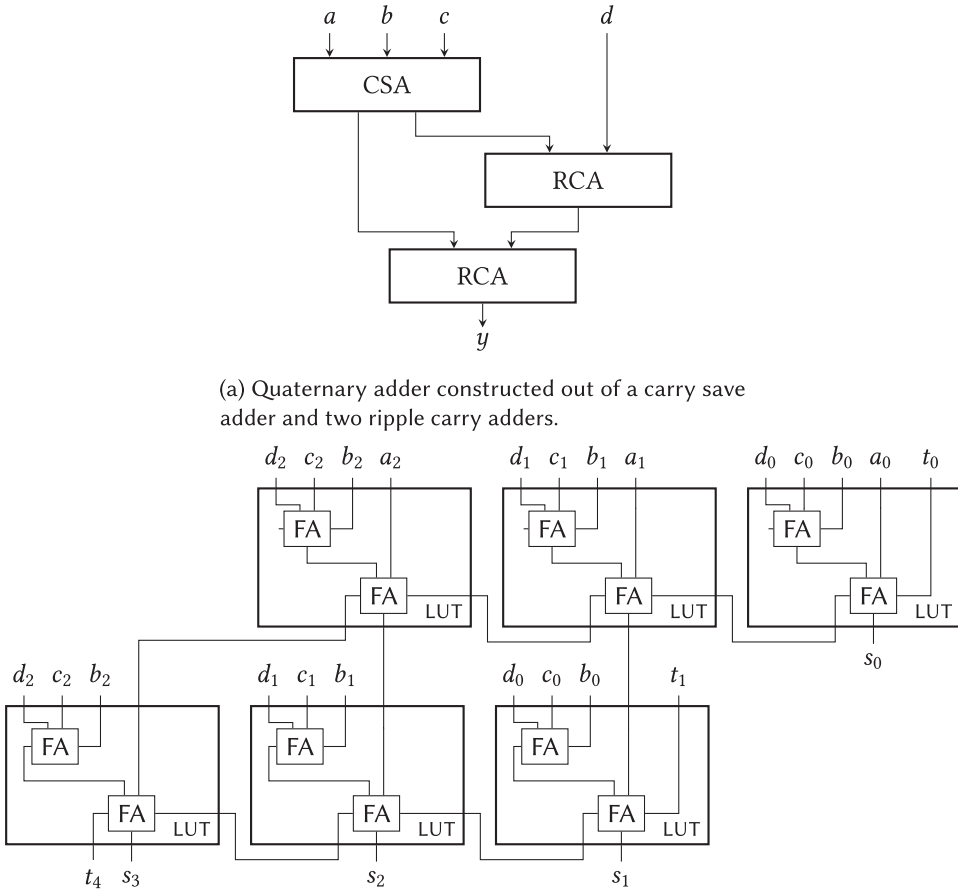
the LUT cascade. Column counters have competitive figures of merit for reducing high but narrow input matrix shapes.

3.4 Terminal Addition

Previous publications that constructed designs for 7 Series and UltraScale+ FPGAs were able to use a ternary adder for the terminal addition stage [19]. As depicted in Figure 12, a ternary adder can be implemented very efficiently for these fabric generations within the same LUT footprint like a regular two-input adder while being 50% stronger. Note that its secondary carry connections t'_i cannot be mapped onto the carry chain, which is already occupied by t_i . It is instead realized using general-purpose routing. As t'_{i+1} is independent from t'_i , the ternary adder only suffers from a single extra routing delay compared to the two-input adder irrespective of its actual width. No additional slow ripple path is created. As the ternary adder produces three outputs per bit position, it cannot be mapped onto the Versal two-output LUTs (cf. Figure 3).

For Versal devices, we propose the novel, highly efficient quaternary adder shown in Figure 13(b). It uses two LUTs per bit position and compresses four-input addends down to a single binary sum. Just like for the ternary adder, a single level of mutually independent general-purpose routing links is added to the critical paths. This makes it just as fast as a four-input adder constructed as balanced tree from standard binary adders while using 33% fewer LUTs. As its delay also matches the ternary adder, even a backport to 7 Series and UltraScale+ devices is interesting. The greater compression for the suffered delay offers an alternative resource/speed trade-off leaning more towards speed in comparison to the ternary option. Both optimized alternatives strictly outcompete the basic binary adder tree.

The quaternary adder is implemented by leveraging the idea of a carry-free pre-compression of three inputs down to an equivalent sum of two intermediate addends using a **carry-save adder (CSA)**. The resulting structure shown in Figure 13(a) does not reveal an immediate benefit. However, the two individual functions of the full adders comprising the CSA (i.e., their carry and sum bit computations) can individually be absorbed into the LUTs implementing the **ripple-carry adders (RCAs)**. This results in the CLB mapping proposed by Figure 13(b). The quaternary



(a) Quaternary adder constructed out of a carry save adder and two ripple carry adders.

(b) Two elements of a quaternary adder in the Versal™ fabric.

Fig. 13. Quaternary adder.

adder can be accelerated by the lookahead module as all carry links arise from a conventional ripple-carry structure.

As the ternary adder is the most efficient adder implementation for 7 Series and UltraScale+ devices, we choose it for the terminal adder in these fabrics. For Versal devices, we use our proposed quaternary adder.

4 HEURISTICS FOR VERSAL DEVICES

Opening up to composed variable-sized counters on Versal™ devices render the use of a finite counter library a clumsy misfit. Indeed, such a library could be pre-generated and populated with all counters up to some generous practical size limit. Nonetheless, we opt to go beyond the established static heuristics, circumventing the associated memory burden, and follow a dynamic approach instead.

As already done by Parandeh-Afshar et al. [16], we construct compressors in compression stages. Each stage applies a selection of GPCs to the available inputs and produces an output matrix of reduced height. Stages are appended to the compressor until the height of the matrix is compatible

with the input capabilities of the terminal adder. This carry-propagate adder sums up the rows remaining at this point.

The construction of a stage first identifies the rightmost column of a height that still requires compression. Starting there, we consider three counter placement candidates: (a) a plain floating $(6 : 3]$ or $(10 : 4, 2]$ counter if enough inputs are available, (b) a row GPC grown to the left favoring extensions by $(1, 4)$ atoms over (2) atoms, and (c) a column GPC grown vertically favoring the dual-rail option. The best of these candidates, either by prioritizing strength or efficiency, is selected and placed. This counter scheduling continues until the remaining inputs no longer permit a valid counter placement. Note that, at least, a full adder resulting from either a single (2) atom or an unextended carry-sum adder can always be placed in a stage unless the matrix is not higher than two rows. If this was the case, then the terminal adder would already be able to sum up the result.

In the overall compressor, the critical path is dominated by the sequence of compression stages. Within each stage, the computation occurs from right to left as soon as row compressors are involved. The suffered carry-chain delay is, however, significantly smaller than the inter-stage general-purpose routing paths. Thus, we choose to disregard the carry-chain delays in the compressor construction. To ensure the validity of this approximation, we will impose an educated restriction on the maximum cascade length.

In addition to the right-aligned counter placement, we explore a more costly *global* strategy. It considers a greater set of counter candidates constructed over all compressible columns for each selection decision. This experiment is to quantify the cost of the intuitive heuristic restriction to simply start at the rightmost feasible column. Both strategies are formulated as pseudocode in Algorithm 1.

5 EXPLICIT COUNTER CLUSTERING FOR VERSAL DEVICES

Many counters for Versal™ devices use the LUT cascade for propagating a rippling signal. However, when LUTs are instantiated in HDL without further guidance, Vivado™ often misses the opportunity of placing cascadable LUTs adjacently. In fact, we observe that Vivado™ only maps adjacent LUTs onto a carry chain when the lookahead module is used. This inevitably implies the fallback to slower general-purpose routing. We propose three strategies for coercing a successful cascade mapping.

When the lookahead module is used, its explicit instantiation also forces the clustering of all its input LUTs into the same slice. However, this approach also activates lookahead-acceleration logic on the cascade path, which does not match the behavior required by our ripple-sum counters. Otherwise, Vivado™ understands relative clustering constraints, which are specified as HDL attributes. These constraints allow to define local clustering relationships, combining LUTs within a slice while maintaining the global mobility of the created structure. In the context of compressor generation, we aim at minimizing the number of occupied slices while also minimizing the interference with the place and route processes to avoid an undue impact on routing quality and the achievable operating frequency.

The clustering can be achieved through the use of Vivado™ RLOC, HU_SET, and BEL constraints. The BEL constraint specifies the positioning of a specific LUT within a slice. The RLOC constraint assigns the LUT to a distinct slice within an HU_SET of slices. Dedicating a unique HU_SET to each slice allows Vivado™ to place each slice independently. These constraints were employed to implement three distinct clustering strategies that ensure LUTs are positioned in a manner that facilitates the utilization of the LUT cascade. Their behavior for an example input is illustrated by Figure 14.

ALGORITHM 1: Compressor Generation Heuristic.

Data: matrix – Matrix to be compressed

```

1 Procedure addCompressionStageRightAligned
2   outputs := Matrix.empty();
3   for  $i \leftarrow 0$  to matrix.size – 1 do
4     while (matrix[i].height > 4) do
5       scheduleBestCounter(matrix, outputs);
6   matrix := matrix.stack(outputs);
7 Procedure addCompressionStageGlobal
8   outputs := Matrix.empty();
9   counters := [];
10  while (matrix[i].height > 4) do
11    for shift ← 0 to matrix.size – 1 do
12      counters += getBestCounter(matrix, outputs, shift);
13    scheduleBestCounter(counters);
14  matrix := matrix.stack(outputs);
15 while matrix.maxHeight > 4 do
16   run addCompressionStageRightAligned() or addCompressionStageGlobal();
17   schedule pipeline registers as requested;
18 schedule quaternary adder;

```

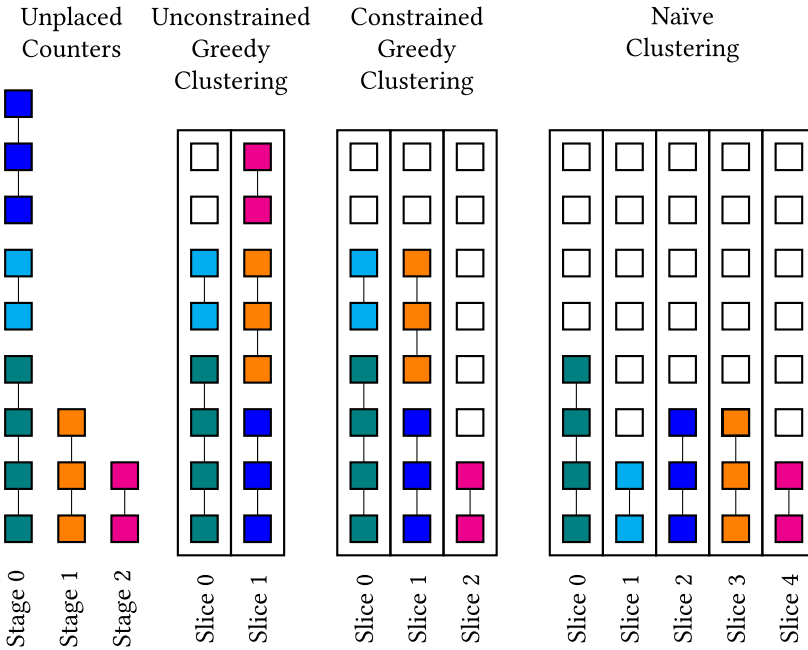


Fig. 14. Explicit clustering approaches.

Naïve Slice Clustering. This approach adopts a simplistic strategy where each counter is allocated a separate slice. Cascades are initiated from the beginning of the slice, starting at LUTA. This approach avoids the need to maintain any clustering state beyond the scope of individual counters. This approach risks a significant fragmentation and underutilization of slices.

Unconstrained Greedy Slice Clustering. This approach aims at mitigating the potential slice fragmentation resulting from the naïve approach. A counter is only allocated a new slice if no already partially occupied slice is able to accommodate it in its spare LUTs. Counters are packed densely into a slice. Whatever part of the slice remains unutilized after a counter clustering will be considered for the use by counters clustered later. We anticipate that this approach will result in a notable improvement in slice utilization. However, since it allows counters from various stages of the design to be clustered in the same physical slice, it is possible that it may adversely impact timing results. Our experiments later will prove the concern regarding the timing results of this approach unwarranted.

Constrained Greedy Slice Clustering. This approach tries to strike a balance between the two previous approaches. It allows for the clustering of multiple counters within the same slice but only if they belong to the same or neighboring compression stages. Underutilized slices with counters originating from an older but the preceding compression stage are no longer considered for a co-located clustering of another counter. We anticipate that this approach will balance between a good slice utilization and an affordable timing impact.

In all approaches, we completely exclude counters that do not utilize the cascade path from any clustering constraints to let Vivado™ place them independently. Also, the terminal adder is not constrained explicitly but instead instantiates the lookahead module directly.

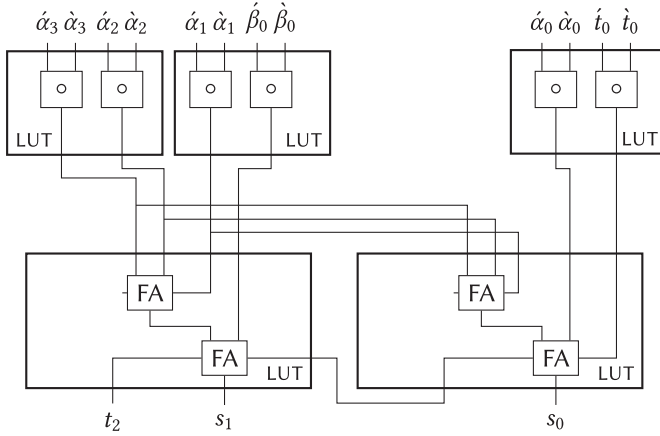
6 GATE ABSORPTION FOR COMPUTED COMPRESSOR INPUTS

As will be detailed by Section 8.10, certain implementations, such as integer dot product computations, feature compressor inputs that are the results of immediately preceding two-input gates. These gates could be mapped to designated LUTs on the input paths before the compressor. Given the fracturable 6-LUT architecture, this would amount to an additional cost of half a LUT per gate. However, some counters would be able to absorb these gates directly for yielding a more efficient design. Figure 15 elucidates the underlying concept. Note that the signals comprising an input pair to an absorbed gate are differentiated using accents.

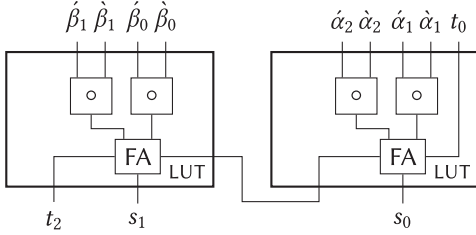
As the (1, 4) atom already occupies all available LUT inputs itself, no input gates can be absorbed. The corresponding logic would have to be implemented in designated preceding LUTs as illustrated by Figure 15(a). On the contrary, the (2) atom (i.e., the building block of a conventional fabric RCA) is completely capable of absorbing two-input gates on all of its inputs except for the very first carry. We opt for not using this input rather than connecting an externally implemented input gate. This choice downgrades the initial full adder to a half adder. Note that the same absorption is possible for our proposed ripple-sum adder. This results in an equally dense design, however, with a vertical rather than horizontal input shape.

Figure 15(c) shows an implementation for a single full adder with gate absorption as proposed by Umuroglu et al. [22]. The full adder is split into two independent LUTs, thereby fully saturating their inputs. Although this doubles the footprint of the full adder, this structure is still half a LUT smaller than an implementation with preceding external input gates. It also eliminates a complete level of general-purpose routing from the critical path of the computation.

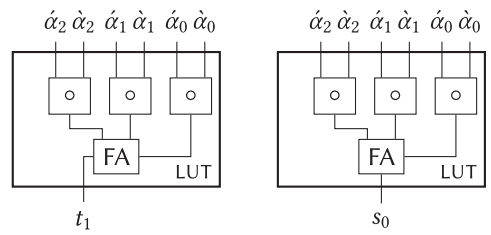
For the evaluation of the efficiency of a counter absorbing input logic gates, we introduce a derived metric of merit that accounts for the logic gate footprint:



(a) (1, 4) atom with preceding two-input gates. As all five inputs of each of the atom's LUTs are already occupied, no gates can be absorbed.



(b) Implementation of a full adder cascade with logic gate absorption. Only one LUT per bit position is needed, however the carry-in input has no preceding gate.



(c) Implementation of a full adder with logic gate absorption. The full adder is split into two LUTs, each operating in 6-input 1-output mode.

Fig. 15. Counters with absorbed logic gates.

Definition 3. The *fused efficiency* E_g of a GPC fused with preceding input logic gates is the quotient of the reduction from the number p of input signal pairs to the number q of compressed output bits over the number k of LUTs occupied by both the input logic and the counter

$$E_g = \frac{p - q}{k}.$$

Table 5 lists the efficiency and strength for counters with absorbed two-input logic gates. In contrast, Table 6 lists the efficiency and strength for counters with the same logic gates implemented externally. The fused counters achieve considerably higher efficiency values approaching 1 for long cascades whereas no implementation with external gates is able to surpass an efficiency of 0.66. This limit arises from the peak efficiency of the counters in use and the constraint that no more than two two-input gates can be accommodated within a single LUT6. It is important to note that the utility of the stated strength metric is diminished by the fact that all counters with external gates introduce an extra general-purpose routing level. This effectively introduces an additional stage to the compressor defeating the original goal of the metric.

In light of their superior efficiency and reduced combinational delay, we support preceding the standard compressor with a dedicated stage consisting solely of gate-absorbing counters. During the construction of the compressor, we promptly integrate the desired gates into these counters.

Table 5. Efficiency and Strength of Counters with Absorbed Logic Gates

Counter		E_g	S
Floating	(3 : 2]	0.5	1.5
Ripple-signal	Ripple-carry	$1 - \frac{1}{n}$	$1 - \frac{2}{n+1}$
	Ripple-sum	$1 - \frac{1}{n}$	$1 - \frac{2}{n+1}$

Table 6. Efficiency and Strength of Counters with External Logic Gates

Counter		E_g	S
Slice	(1, 3, 2, 5 : 5]	0.63	2.2
Floating	(3 : 2]	0.4	1.5
	(6 : 3]	0.5	2
	(2, 5 : 1, 2, 1]	0.55	1.75
Ripple-signal	Ripple-carry	$0.5 - \frac{1}{8n+2}$	$2 - \frac{1}{n+1}$
	(1, 4) Atom Chain	$0.66 - \frac{6}{81n+9}$	$2.5 - \frac{3}{2n+2}$
	Ripple-sum	$0.5 - \frac{1}{8n+2}$	$2 - \frac{1}{n+1}$
	Dual-rail Ripple-sum	$0.66 - \frac{6}{81n+9}$	$2.5 - \frac{3}{2n+2}$

We expect logic gate absorption to be a very effective optimization. We construct the initial compression stage with absorbed gates using the same heuristic as usual. However, input pairs that are not consumed by this stage are not copied to the next stage. Instead, they are also reduced to single-bit signals by implementing the corresponding input gates in parallel albeit without a compression.

7 BITMATRIX ACCUMULATION

Some applications require the accumulation of addends over multiple clock cycles. Time-sliced dot product computations that are split into smaller partial products establish such a scenario. The bit matrix compression and summation needs to inject both the current accumulator state and the next partial product contribution. We have evaluated two distinct approaches to this accumulation.

Initial Accumulation. The defining aim of this approach is the absorption of the next partial dot product contribution into the footprint of the accumulator state by a single layer of counters. No constraints are imposed on the representation of the accumulated value. It can be represented by a bit matrix of arbitrary shape. A solution to this task is a compressor with a height difference between its input and its output shapes, which accommodates each individual column of the next input contribution. For the construction of the desired accumulation circuit, we select counters based on their *reduction signature*.

Definition 4. The *reduction signature* $R = (r_{n-1}, \dots, r_0)$ of a GPC is the vector of differences in input and output signal dimension per bit position:

$$r_i = q_i - p_i \quad \forall i \in \{n-1, \dots, 0\}.$$

The reduction signature can be positive (expansive) or negative (reductive) for each column index. The counters are assembled in a way that the sum of the total reduction signature for each column is less or equal to the height of the input contribution in the corresponding column. As before, we schedule counters from right to left. Among the counters with fitting signatures, we choose the most efficient one. This process continues until the entire input contribution is consumed. Note that the layout of the bit matrix representing the accumulator is not pre-defined but a result of this process. Only after the completion of the accumulation, its contents would be subjected to a straightforward compression and summation to obtain a flat binary number for its value.

Terminal Accumulation. This approach features a conventional binary representation of the accumulator state. The compression of the input contributions is performed first. Only the terminal carry-propagate addition is modified to include the current, registered accumulator value as another input row. In consequence, the compression of the input contribution must be continued down to three or even two rows depending on whether the terminal summation is implemented by a quaternary or ternary adder, respectively. Typically, this will add another compression stage. The effort is naturally bounded as it cannot be more complex than inserting a carry-save adder.

Both approaches maintain the same number of combinational general-purpose routing paths between pipeline stages when fully pipelined. Therefore, we anticipate similar timing results. Thus, the key distinguishing factor is likely to be the observed LUT utilization.

8 QUANTITATIVE EVALUATION

We have implemented the described algorithms in a Chisel hardware generator that produces Verilog code for compressors for any specified input matrix shape. To conduct a comprehensive assessment of this generator, we initially perform a comparative out-of-context evaluation of design options. Subsequently, we compare our solution with adder trees implemented in Vivado™. Finally, we examine the use case of integer dot product computations and analyze the impact of transitioning from a Vivado™ adder tree to the compressors generated by our tool.

8.1 Out-of-context Compressor Evaluation

We evaluate all compressors by the synthesis results obtained using Vivado™ 2023.1 with default synthesis settings targeting the Versal™ xcvc1902-vsva2197-2MP-e-S part, which is found on the VCK190 Evaluation Platform. The compressors are generated as purely combinational circuits that are embedded into a register sandwich for timing evaluation. We report the critical path delay of the tightest successful timing closure after performing interval nesting on the timing goal with a failed attempt at most 0.1 ns shorter. The LUT usage reported by Vivado™ serves as our area metric.

For a direct comparison, we generate the compressors for the same input shapes as previously used by Preußner [19]. In addition to the various explicit one- and two-column input matrices, this selection includes the bit product matrix that arises from a 16-bit radix-two integer multiplication, which is identified as *Mul16*. The single- and two-column input matrices are identified by the height of their columns (i.e., *(128)* denotes a single-column matrix with 128 inputs, and *(512, 512)* denotes a two-column matrix with 512 inputs in each column).

8.2 Choosing a Cascade Length Limit for Versal Devices

Rippling counters can be cascaded to an arbitrary length. Long cascades have a positive impact on counter strength, thereby reducing the compression stage count. However, the cascade delay can significantly slow down the design, and this delay cannot be eliminated through pipelining, since there are no registers in the cascade path. Thus, our objective is to find a balance between the two conflicting goals.

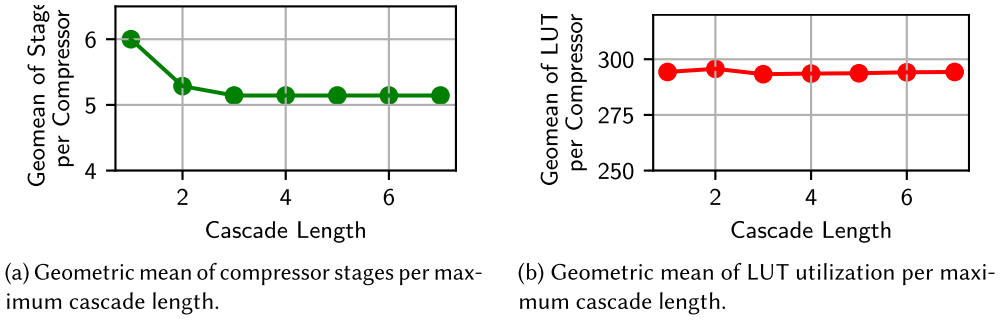


Fig. 16. Impact of cascade length on compressor latency and LUT utilization.

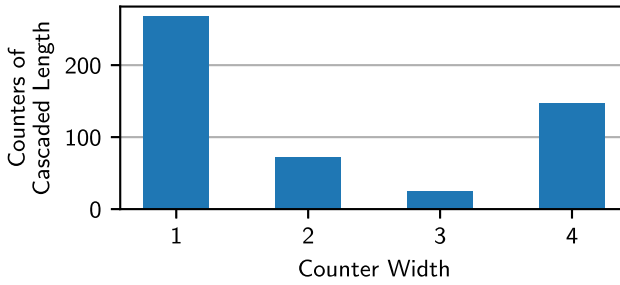


Fig. 17. Distribution of counter widths for our example compressors when limiting length to four.

To determine a desirable maximum cascade length, we construct compressors for the examples (128) , $(128, 128)$, (256) , $(256, 256)$, (512) , $(512, 512)$, and $Mul16$ while imposing different length limits on our counters. We evaluate the geometric mean of LUT utilizations and compressor stage count for each of these limits. Based on these findings presented in Figure 16, it can be inferred that the number of stages consistently decreases up to a cascade length of four. However, for larger cascade lengths, the benefits diminish rapidly and become indistinguishable from noise. In terms of LUT utilization, only a very small decline until a cascade length of three is observed. Therefore, we conclude that setting a maximum cascade length of four strikes a favorable balance. As demonstrated in Section 3.3, we can confidently forego the use of the lookahead module without experiencing a significant impact on performance. This gives us the freedom to cluster all counters within slices without the constraint of starting them at the first LUT.

Figure 17 illustrates the distribution of the lengths of cascades instantiated for the test examples. The most frequent counters either do not utilize the LUT cascade at all, being of width one, or extend all the way to the cascade length limit. If this limit is lifted, then our heuristic will construct many wider counters without gaining a relevant benefit in compression stage count. The heavy utilization of cascade-free counters can be attributed to the selection of tall but narrow input matrices. The newly introduced possibility for widths of two and three is less frequently exploited. Nonetheless, these cases still account for over 25% of all instantiated counters.

8.3 Comparison of Heuristics

Figure 18 shows the LUT utilization of all generated compressors. The prioritizations of strength versus efficiency in the counter selection produce noticeable differences. In geometric mean, efficiency-optimized counters use about 10% fewer LUTs than strength-optimized ones. The global

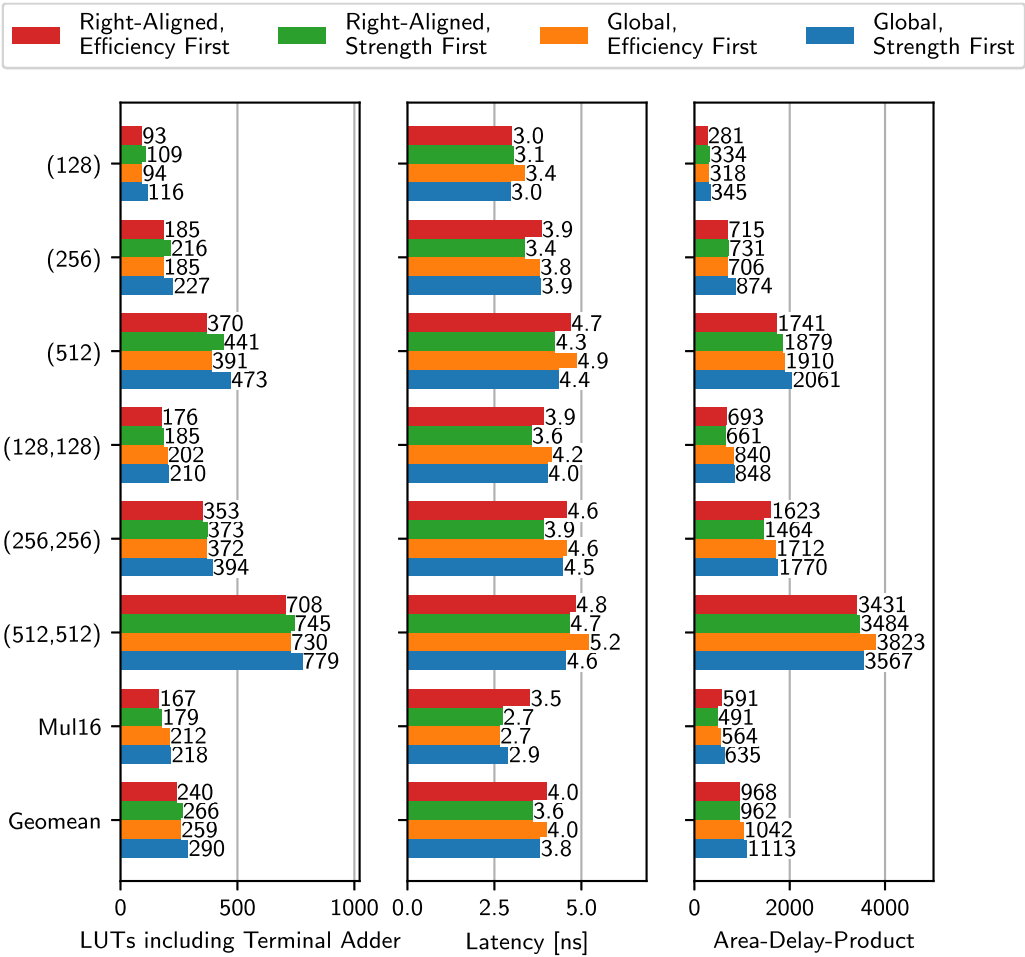


Fig. 18. Comparison of compression heuristics. Results obtained with Vivado™ 2023.1 targeting a VCK190 (Versal™) evaluation board.

selection heuristic is not able to extract any meaningful benefit from extending the search for feasible counter placements. In fact, its results are inferior in all depicted metrics. These findings suggest that equally good results are obtained when presupposing that the next counter should just be placed starting in the rightmost column that still requires compression.

The critical path delays reported by Vivado™ for the same compressors are plotted in Figure 18. Optimizing for efficiency suffers a 5% to 11% increase in delay. It still holds that the unconstrained greedy clustering heuristic is unable to dominate the simpler right-aligned counter placement. However, the variation produced by the individual heuristics for the same input shapes is clearly increased albeit without a consistent winner.

8.4 Clustering Strategies

The objective behind our clustering strategies is the reduction of the number of slices allocated by the compressor while minimizing adverse effects on the operating frequency. Therefore, we assess the slice utilization and critical path of combinational compressors generated for our test examples

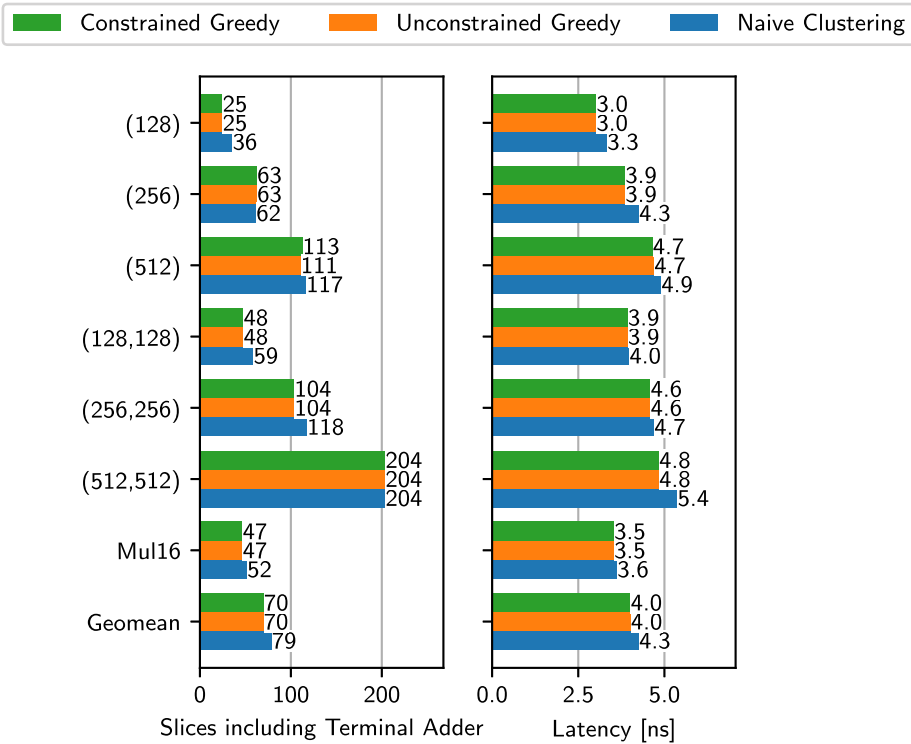


Fig. 19. Evaluation of explicit clustering strategies. Results obtained with Vivado™ 2023.1 targeting a VCK190 (Versal™) evaluation board.

within a register sandwich configuration. The slice utilization measurement is, however, affected by noise, since counters that do not utilize the cascade are not explicitly assigned to slices.

The findings are presented in Figure 19. Both the constrained and unconstrained greedy clustering strategies demonstrate a reduction in slice utilization over the naïve clustering strategy when considering the geometric mean and result in equal operating frequencies and slice utilization for every example except one. The results show that our explicit clustering strategies positively impact slice utilization, reducing it by 11.4% on average. These improvements seem to cause a 7% decrease in delay. Consequently, we assert that our explicit clustering approaches are valuable and apply the unconstrained greedy clustering approach in the subsequent sections.

8.5 Comparing Approaches to Accumulation

While both the initial and terminal accumulation approaches achieve very similar maximum operating frequencies, they differ greatly in LUT utilization. As illustrated in Figure 20, the terminal accumulation exhibits a LUT footprint that is smaller by 43% in the geometric mean across the input shapes. The larger uncompressed representation of the accumulator state used by the initial accumulation implies a significant toll on fabric resources. In further evaluations, we therefore opt for the terminal accumulation approach.

8.6 Effect of Logic Gate Absorption

For assessing the impact of logic gate absorption, we compare two implementations of an accumulator. The first implementation truly absorbs input gates. This gate absorption is implemented inside our compressor generator. Alongside the shape of the input matrix, the generator is

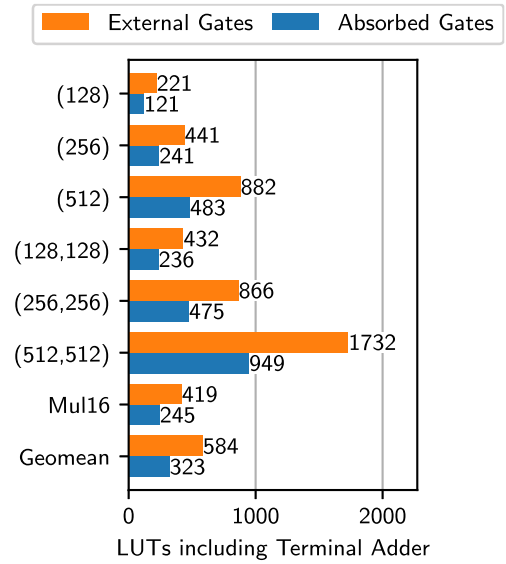
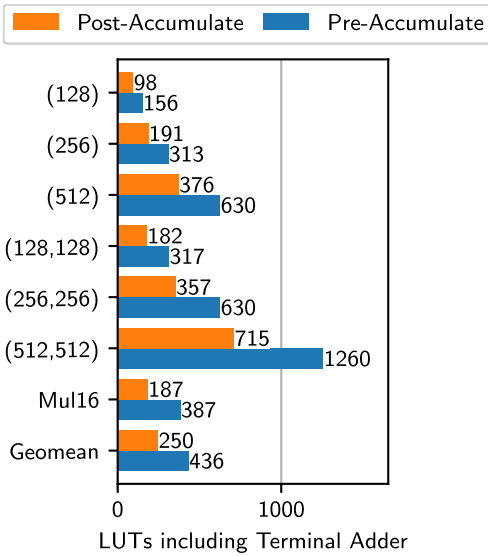


Fig. 20. Comparison of the initial and terminal accumulation approaches. Results obtained with Vivado™ 2023.1 targeting a VCK190 (Versal™) evaluation board. Fig. 21. Effect of enabling gate absorption on LUT utilization. Results obtained with Vivado™ 2023.1 targeting a VCK190 (Versal™) evaluation board.

supplied with a single hexadecimal digit for each pair of inputs. It encodes the truth table of the desired two-input gate. The generator integrates the specified functionality directly with the candidate counters. Our second implementation uses plain behavioral HDL to attain the same functionality before and separate from the actual compression.

The results, depicted in Figure 21, reveal a substantial reduction of LUT utilization by 44.6% with gate absorption by the geometric mean across all examples. The substantial surge in LUT utilization upon the introduction of two-input gates can be attributed to Vivado’s decision not to merge two two-input gates into a single LUT. An explicit manual merger of two gates into a fractured LUT would, indeed, reduce the LUT consumption considerably. Nevertheless, the achieved results would still fall short of the solution with absorption by 21%. Based on these findings, we conclude that gate absorption is a valuable optimization technique for reducing the design footprint of corresponding use cases.

8.7 Comparison to Vivado™

We have evaluated multiple approaches to compressor construction in the previous sections. To evaluate the effectiveness of our approach in comparison to state-of-the-art solutions, we compare the compressors generated by our heuristics with a straightforward HDL implementation, which achieves summation by adder trees built from high-level addition operators.

8.7.1 Compression. For plain compression, all the compressors for our examples are implemented combinatorially inside a register sandwich. The results of the comparison are displayed in Figure 22. For all test examples, we achieve a significant reduction in LUT utilization, a 45.3% decrease in geometric mean.

The critical paths of our implementations are about 17.5% slower by the geometric mean when compared to the default Vivado™ implementation. For one part, we attribute this to the constraints

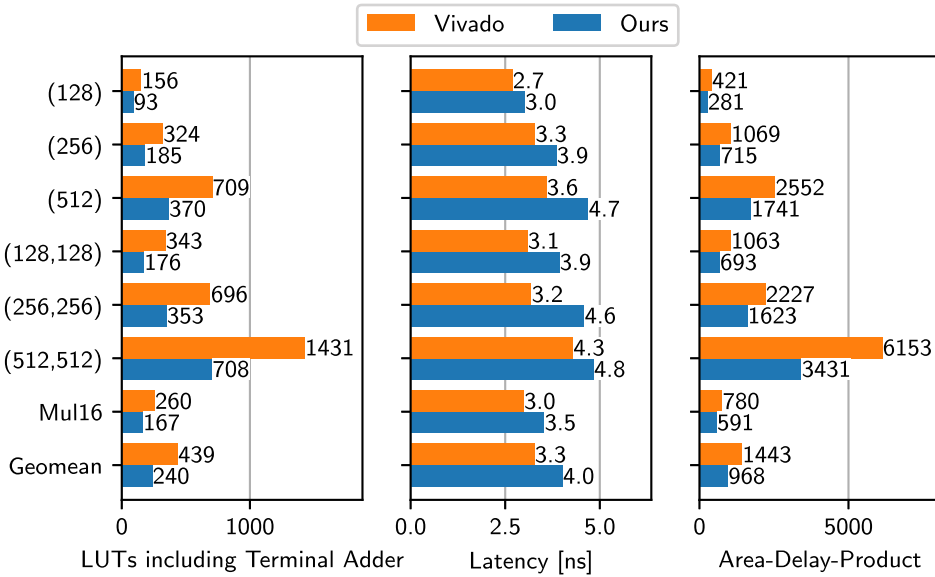


Fig. 22. Our compressor implementation compared to a Vivado™ adder tree implementation inside a register sandwich. Results obtained with Vivado™ 2023.1 targeting a VCK190 (Versal™) evaluation board.

Table 7. Counter Utilization and Stage Count

Counter Matrix	Sum-cascade		Carry-cascade			Floating		Stages	
	Ripple-sum	Dual-rail ripple-sum	Atom (2)	Atom (1, 4)	Atom (2, 2, 2)	(6 : 3]	FA	(10 : 4, 2]	
(128)	0	4	3	1	5	3	1	16	4
(128,128)	0	43	4	5	7	2	0	14	5
(256)	0	21	4	0	5	1	1	37	5
(256,256)	0	105	2	4	4	0	1	36	6
(512)	0	46	4	5	3	1	2	79	6
(512,512)	4	220	3	6	8	1	2	70	7
Mul16	0	0	2	51	0	0	1	0	3

that the explicit LUT clustering imposes on the place and route processes as outlined in Section 5. More importantly, the explicit formulation of our compressor solutions on the level of device primitives prevents Vivado from applying performance-enhancing transformations, such as, limiting signal fan-outs by logic duplication. This interpretation is consistent with the considerably larger LUT footprint of Vivado’s solutions. Combining the critical path delay and LUT utilization in the area-delay product, our implementation demonstrates a mean improvement by 33%. When the fabric is packed with as many operators as possible, as common for parallelizable tasks, this enables a significant increase in compression compute power on Versal fabric.

Table 7 enumerates the GPC construction statistics. The (6 : 3] and (10 : 4, 2] counters standalone modules are simply counted by the instance. As the atoms and the ripple-sum stages are components of the row and column counters, respectively, they are counted by each individual instantiation within an assembled counter. Lone full adders are counted separately. Looking at the new column counters, the ripple-sum counter and (6 : 3] counters are pushed into a niche role picking up leftover bits of single columns as it is dominated by the (10 : 4, 2] counter for higher

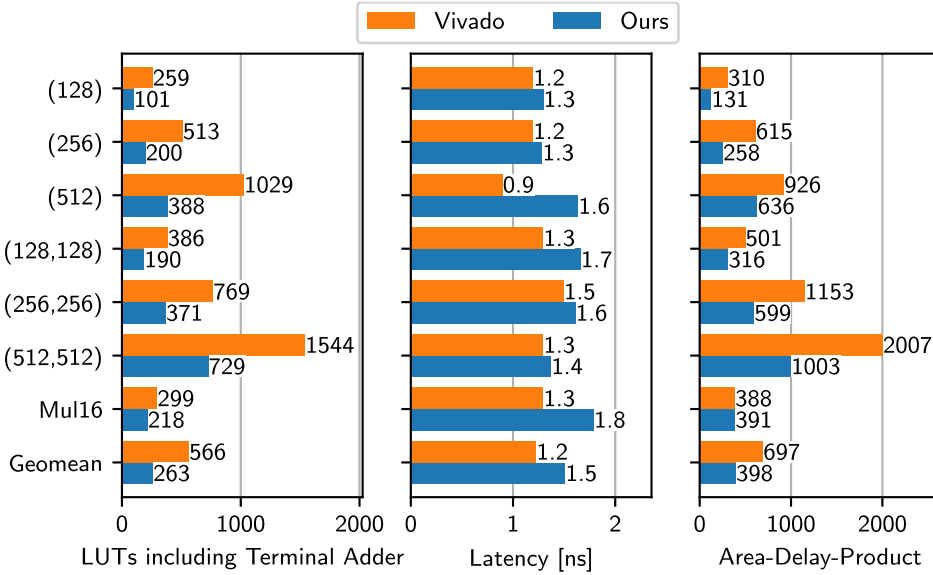


Fig. 23. Our accumulator implementation compared to a Vivado™ adder tree implementation inside a register sandwich. Results obtained with Vivado™ 2023.1 targeting a VCK190 (Versal™) evaluation board.

inputs. On the contrary, the dual-rail ripple-sum counter assumes a strong position in reducing two-column inputs, often being the most popular counter choice. As all single- and double-column examples are of a tall input shape, they make heavy use of the $(10 : 4, 2)$ and dual-rail ripple-sum counters. The wide *Mul16* example, however, makes no use of column counters at all but relies exclusively on row counters constructed from a mix of (2) and $(1, 4)$ atoms.

The inclination of our approach towards area efficiency, albeit at a potential frequency cost, prompts an inquiry into whether Vivado™ could produce competitive solutions by explicitly instructing it to prioritize area optimization during synthesis. To explore this, we employed the *AreaOptimized_High* strategy in Vivado™ for synthesizing the compressors from our test suite. The resulting implementations utilized 348 LUTs in geometric mean, indicating a utilization that is still 31% higher than our approach.

8.7.2 Accumulation. To measure how our solution compares to a reference accumulator, we construct a balanced adder tree in HDL using two-input addition operators. Both our generated compressor and the Vivado™ implementations are pipelined after every compression stage. The resulting circuits are measured inside a register sandwich. From the resulting numbers, depicted in Figure 23, we conclude that the differences in LUT utilization and operating frequency stay very significant. Our implementation improves the geometric mean of LUT utilizations by 46.5%. We expect two effects to be at play here: The extra two-input adder of the default implementation, which is used for accumulation, is fused into our quaternary adder for our implementation, providing a more efficient solution. Also, our compressor seems to handle efficiently the reduction of the compression goal from four to three rows, which is necessitated by the accumulating loopback connection into the quaternary adder. In terms of operating frequency, however, the results indicate clearly towards the default implementation. This is due to the quaternary adder having an extra general-purpose routing delay in comparison to the two-input adder. Pipelining the quaternary addition will typically remain an unattractive design option as the loop-carried dependency implied by the accumulator feedback would require further modifications in the

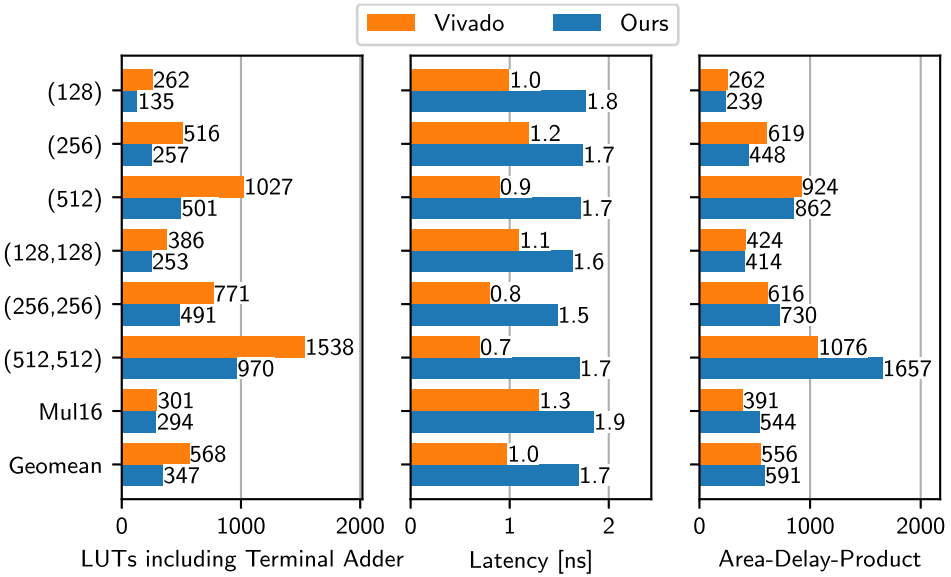


Fig. 24. Our accumulator implementation compared to a Vivado™ adder tree implementation inside a register sandwich with logic gate absorption. Results obtained with Vivado™ 2023.1 targeting a VCK190 (Versal™) evaluation board.

organization of the computation. Nevertheless, as all examples can be clocked at over 500 MHz, we expect our implementation to be sufficiently fast for the vast majority of designs while consuming considerably fewer resources.

8.7.3 Gate Absorption. To evaluate our compressor implementations with absorbed input gates, we added logic gates to the reference bitmatrix accumulator from the previous section. As shown in Figure 24, our implementation remains superior in all examples. Its advantage in LUT utilization now is at 38.9% with similar differences in operating frequencies in the fully pipelined design. Investigating the implementation for the default design shows that Vivado™ also chooses to absorb input logic into its two-input adders. Our implementation yields tremendous improvements over the default one, mainly by the usage of strong counters from the second compression stage on and due to the more efficient terminal adder.

8.8 Evaluation as 7 Series Back Deployment

Our compressor generator allows the parameterization of the counter library. Thus, compressors for the 7 Series fabric can be created easily by replacing the flexible-length Versal counter library with the fixed-length 7 Series counterpart. For 7 Series and UltraScale+™ targets, we also substitute the terminal quaternary adder by the more efficient ternary adder. To assess the effectiveness of our heuristics compared to state-of-the-art solutions, we execute the compressor generator developed by Preußner [19] with the original (0,6) atom replaced by the improved version devised by Yuan et al. [32] and compare the results to the implementations generated by our tool. We choose to optimize for strength first with efficiency as the second criterion.

Differences in LUT utilization and critical path delays are illustrated in Figure 25. Both the resource utilization and critical delays of the implementation by Preußner improve by the adoption of the enhanced (0,6) atom and by the use of a newer Vivado™ tool version. The mean of the critical

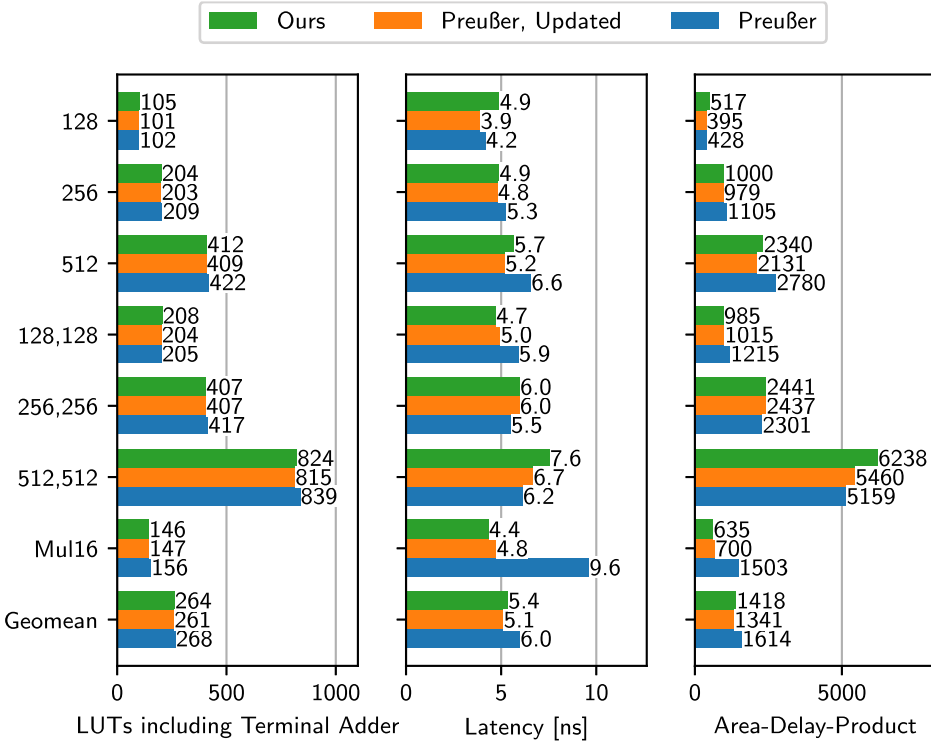


Fig. 25. Compressors for 7 series by our generator compared to Reference [19]. The implementation for Preußer was re-run with the new (0, 6) atom, yielding significant changes in operating frequency. Results obtained with Vivado™ 2023.1 targeting a xc7vh870tflg1932-2 (7 Series) part.

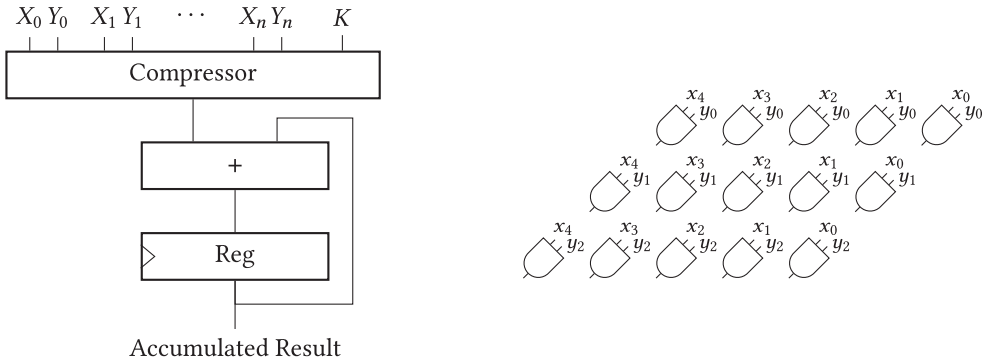
path decreased from 6 to 5.1 ns. The decrease is especially pronounced for the *Mul16* example going from 9.6 to 4.8 ns.

The differences between our compressors and the ones generated by Preußer are miniscule. The geomean in LUT utilization changes by less than 1.2%. The operating frequency of most examples is almost equal. For three of the seven examples, the difference is no bigger than 0.3 ns. There are two designs with rather big differences, (128) and (512). Here, the implementation by Preußer requires fewer compression stages. We attribute this deviation to the flexible compression goal proposed by Preußer, which we have not implemented.

The small differences between our solution and the one of Preußer prove the competitiveness of our generator, which also adds the flexibility of constructing flexible-width counters for Versal devices. However, our heuristic does not improve the compressor generation for 7 Series devices further.

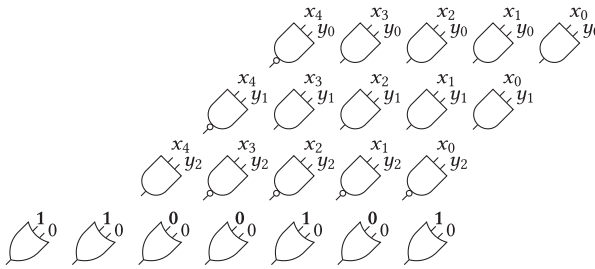
8.9 Gains of Moving to Versal Devices

Comparing our implementation for Versal with the updated implementation for 7 Series by Preußer [19], the critical path delays for the combinational compressor implementations have decreased by 30% in geometric mean (see Figures 18 and 25). This reflects the significant technological advance that has been made between these architecture generations. This improvement is made although the strong (0, 6) atom is not available for the construction of row GPCs on Versal and the ternary adder was sacrificed for the sake of the LUT cascade. The difference is particularly



(a) Block diagram of our integer multiply-accumulator design with absorbed gates.

(b) Absorbed gates for a single-lane 5×3 -bit unsigned multiplier.



(c) Absorbed gates for a single-lane 5×3 -bit two's complement multiplier. The bits of the constant K are marked in bold.

Fig. 26. Our in-context parallel multiply-accumulator.

pronounced for the two-column inputs, which employ more dual-rail ripple-sum counters. The *Mul16* example, with its 32-bit output shape that greatly benefits from fast lookahead logic, also experiences a speedup that exceeds the average improvement. The loss of the ternary adder and (0, 6) atom are almost mitigated by the introduction of the novel variable-size row and column GPCs when optimizing for strength. The LUT cost merely increases by 2%. Choosing counters for both 7 Series and Versal implementations by their efficiency rather than their strength even eliminates this small difference. Both fabrics then consume the same amount of resources in geometric mean. In terms of the fabric utilization by our arithmetic circuits, the architectural challenges and opportunities created by the revised Versal architecture balance out. If arithmetic is able to increase its relevance in the overall application mix, then a combination of the structural benefits of both architecture generations may become an interesting development trajectory.

8.10 In-context Evaluation for Integer Dot-product Computations

While it is relevant to evaluate and compare our compression and accumulation implementations out of context, in practice, they would be integrated into composite designs. As an example of such a design, we consider parallel integer multiply-accumulate units with parameterizable operand widths and number of parallel input lanes, see Figure 26(a). We implement and compare three such designs for both unsigned and signed operands: A non-pipelined baseline design that compresses products in a binary tree, and two pipelined designs integrating our compressor with terminal accumulation. We include these two designs to highlight the benefits of our proposed optimization to absorb the two-input AND and NAND gates needed to generate partial products, as illustrated

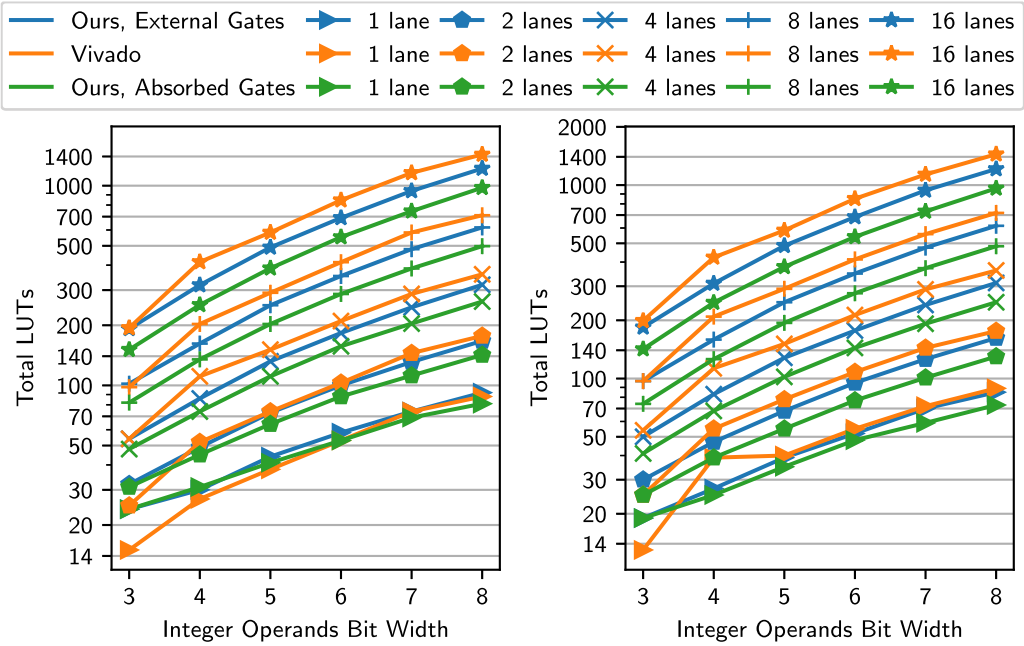


Fig. 27. Our accumulator implementation compared to a Vivado™ adder tree implementation inside multiply-accumulators for (left) signed integer and (right) unsigned integer data types. Results obtained with Vivado™ 2023.1 targeting a VCK190 (Versal™) evaluation board.

for single-lane 5×3 -bit multipliers in Figures 26(b) and 26(c). All signed multiplier designs, regardless of their number of input lanes, need one extra row in the compressor to integrate the constant K needed for the sign extension of the two's complement partial products [28]. As shown in Figure 26(c), the design with absorbed gates has OR gates in the constant row. These gates are not essential for functionality but result from our generator currently being limited to absorbing gates on all or no input positions.

This experiment assumes symmetric operands of width w_{op} ranging from 3 through 8 bits and numbers of input lanes n that are powers of two between 1 and 16. The results are shown in Figure 27. In all designs, the accumulator is wide enough to hold the result of a dot product produced over two cycles accumulating twice as many products as there are input lanes (i.e., $w_{accu} = 2 \cdot w_{op} + \lceil \log_2(n) \rceil + 1$).

We see that both the unsigned and signed integer designs integrating our compressors without gate absorption have smaller footprints than the Vivado™ baseline in all configurations but the ones with one or two input lanes. While the reductions in footprint are significant, we believe they are limited by the saturated LUTs in our compressor's first stage, as they form a hard boundary that prevents Vivado™ from performing optimizations.

Our designs with absorbed gates resolve the issue of the hard optimization boundary with great success. As a result, they have smaller footprints than the Vivado™ baseline and the design without absorbed gates for all cases but the signed single-lane ones. Our heuristic struggles to optimize this case as its input matrix has relatively many columns with fewer than four bits, hindering the use of our most efficient counters. This also explains the sub-optimal results for the designs without absorbed gates. Compared with the Vivado™ baseline, our design without absorbed gates reduce LUT utilization of unsigned and signed multiply-accumulators by 11.7%

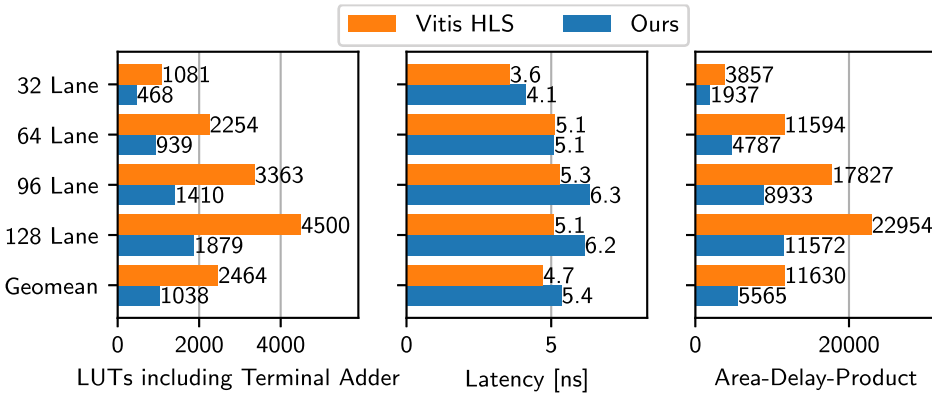


Fig. 28. Wide-fan-in dot products for quantized neural network inference: HLS-generated expanded accumulation tree vs. compressor-based summation. Results obtained with Vitis™ HLS 2023.1 and Vivado™ 2023.1 targeting a VCK190 (Versal™) evaluation board.

and 5.84% in geometric mean. Our design with absorbed gates increase these reductions to 27.8% and 19.3%. The reductions induced for a particular number of input lanes are very homogeneous across different operand widths as indicated by the mostly parallel curves in the logarithmic plot of Figure 27. Focusing on the 16-lane designs, we see the significant benefits of our compressors as designs without absorbed gates reduce LUT utilization by 17.5% and 15.6% for unsigned and signed multiply-accumulators, and absorbing input gates improve reductions to 35.3% and 33.1%. For a sanity check, consider the single-lane accumulating a 8×8 -bit product onto a 17-bit accumulator. The compressor generated for this computation occupies 81 LUTs. This compares well with an optimized state-of-the-art 8×8 -bit multiplier as proposed by Böttcher and Kumm [6]. Depending on the optimization goal, they spent 51–72 LUTs on the plain multiplication excluding the 17-bit adder for accumulation.

Particularly, machine learning solutions often require a massive amount of computation. FPGA deployments would typically leverage low-precision integer arithmetic implementing the inference of quantized models. This approach is taken by popular projects like FINN [3] and hls4ml [9]. As channel counts in the hidden layers of modern topologies, such as ResNet, easily reach the hundreds, even their seemingly small 3×3 convolutions become massive dot products. Performance demands are hence met by unrolling the computation aggressively in space with a strong incentive to accumulate many products within a clock cycle. Therefore, we have evaluated an unrolled wide-fan-in dot product computation over low-precision 4-bit operands. We sweep the product count in steps of 32 all the way to 128. The baseline used for comparison is derived from FINN’s HLS library but stripped from all control overhead by annotating all interfaces to be implemented without a protocol using the pragma annotations `ap_none` and `ap_ctrl_none`. The result is a purely combinational circuit, which is explicitly unrolled by Vitis™ HLS into a balanced adder tree with a precise data path widening as product contributions accumulate towards its root. This detailed expansion yields a high-quality design that is superior to any casual implementation even in RTL.

As illustrated by Figure 28, these designs compare well against our compressor-based implementations. In fact, the general trade-off suggested by the smaller experimental setups reported before continues into these large compute structures. The standard tool solutions demonstrate slightly faster critical paths within a register sandwich but pay a high premium by consuming more than twice the number of LUTs to secure this advantage. The resulting area-delay product

clearly identifies the compressor-based solutions as the ones that can extract more computation from given resource and time budgets. It is noteworthy that the timing behavior of either approach becomes less predictable as the designs grow large. The considerable routing needed to tie the whole computation together asserts its relevance for timing. Currently, neither our nor the standard approach take any measures to guide the routing process. In consequence, the quality of results spreads wider and makes unintuitive steps.

9 CONCLUSIONS

This article has described how generic matrix summation for Versal™ FPGAs can be implemented. Our generic implementation can generate compressors for operations such as population count, integer multiplication, and large dot products, which are ubiquitous in applications like machine learning. Targeting the recent Versal™ fabric, we have proposed two novel ripple-sum bit counters, of which the dual-rail ripple-sum counter offers state-of-the-art efficiency and strength. We further extended previously proposed counters to be of variable length, yielding valuable increases in counter strength. Furthermore, we have proposed a new quaternary adder, which uses 33% fewer LUTs than the Vivado™ default adder. We compared approaches to bitmatrix accumulation and evaluated improvements by the explicit clustering of counters and fusing preceding logic gates with the compressor. Our proposed heuristics, which can execute in milliseconds, can be integrated in regular design flows and therefore enable fast, practical, and competitive matrix summation on AMD Versal devices with demonstrated significant benefits over the default implementation.

In the future, we aim at extending our generator with support for gate absorption in only a subset of input positions to avoid the inefficiencies described in Section 8.10. Following this, we plan to open-source the generator, integrating and leveraging it within the FINN framework for generating more efficient inference solutions for quantized neural networks on FPGAs.

REFERENCES

- [1] AMD, Inc. 2023. *Versal ACAP Configurable Logic Block*. Retrieved from <https://docs.xilinx.com/r/en-US/am005-versal-clb>
- [2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanović. 2012. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the DAC Design Automation Conference*. 1212–1221. <https://doi.org/10.1145/2228360.2228584>
- [3] Michaela Blott, Thomas B. Preußner, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'Brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Trans. Reconfig. Technol. Syst.* 11, 3 (Dec. 2018), 16:1–16:23. <https://doi.org/10.1145/3242897>
- [4] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser et al. 2022. The future of FPGA acceleration in datacenters and the cloud. *ACM Trans. Reconfig. Technol. Syst.* 15, 3 (2022), 1–42.
- [5] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa. 2013. Arithmetic core generation using bit heaps. In *Proceedings of the International Conference on Field programmable Logic and Applications (FPL'13)*. 1–8. <https://doi.org/10.1109/FPL.2013.6645544>
- [6] A. Böttcher and M. Kumm. 2023. Towards globally optimal design of multipliers for FPGAs. *IEEE Trans. Comput.* 72, 05 (May 2023), 1261–1273. <https://doi.org/10.1109/TC.2023.3238128>
- [7] L. Dadda. 1965. Some schemes for parallel multipliers. *Alta Frequenza* 34 (1965), 349–356.
- [8] F. de Dinechin. FloPoCo Project Website. (n.d.). Retrieved from <http://flopoco.gforge.inria.fr>
- [9] Javier Duarte, Song Han, Philip Harris, Sergio Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran et al. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *J. Instrument.* 13, 07 (2018), P07027.
- [10] Reza Hojajr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. 2021. SPAGHETTI: Streaming accelerators for highly sparse GEMM on FPGAs. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. IEEE, 84–96.

- [11] Madis Kerner, Kalle Tammemäe, Jaan Raik, and Thomas Hollstein. 2021. Triple fixed-point MAC unit for deep learning. In *Proceedings of the Design, Automation & Test in Europe Conference and Exhibition (DATE'21)*. IEEE, 1404–1407.
- [12] M. Kumm and J. Kappauf. 2018. Advanced compressor tree synthesis for FPGAs. *IEEE Trans. Comput.* 67, 8 (2018), 1078–1091. <https://doi.org/10.1109/TC.2018.2795611>
- [13] Martin Kumm and Peter Zipf. 2014. Efficient high speed compression trees on Xilinx FPGAs. In *Proceedings of the Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'14)*, Matteo Michel Jürgen Ruf, Dirk Allmendinger (Ed.). Cuvillier Verlag.
- [14] Martin Kumm and Peter Zipf. 2014. Pipelined compressor tree optimization using integer linear programming. In *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL'14)*. IEEE, 1–8. <https://doi.org/10.1109/FPL.2014.6927468>
- [15] Tayo Oguntebi and Kunle Olukotun. 2016. Graphops: A dataflow library for graph analytics acceleration. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 111–117.
- [16] H. Parandeh-Afshar, P. Brisk, and P. Ienne. 2008. Efficient synthesis of compressor trees on FPGAs. In *Proceedings of the Asia and South Pacific Design Automation Conference*. 138–143. <https://doi.org/10.1109/ASPAC.2008.4483927>
- [17] H. Parandeh-Afshar, P. Brisk, and P. Ienne. 2009. Exploiting fast carry-chains of FPGAs for designing compressor trees. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'09)*. 242–249. <https://doi.org/10.1109/FPL.2009.5272301>
- [18] Hadi Parandeh-Afshar, Arkosnato Neogy, Philip Brisk, and Paolo Ienne. 2011. Compressor tree synthesis on commercial high-performance FPGAs. *ACM Trans. Reconfig. Technol. Syst.* 4, 4 (Dec. 2011), 39:1–39:19. <https://doi.org/10.1145/2068716.2068725>
- [19] Thomas B. Preußner. 2017. Generic and universal parallel matrix summation with a flexible compression goal for Xilinx FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'17)*. <https://doi.org/10.23919/FPL.2017.8056834>
- [20] Adrien Prost-Boucle, Alban Bourge, and Frédéric Pétrot. 2018. High-efficiency convolutional ternary neural networks with custom adder trees and weight compression. *ACM Trans. Reconfig. Technol. Syst.* 11, 3 (2018), 1–24.
- [21] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *Proceedings of the 14th European Conference on Computer Vision (ECCV'16)*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, Cham, 525–542. https://doi.org/10.1007/978-3-319-46493-0_32
- [22] Yaman Umuroglu, Davide Conficconi, Lahiru Rasnayake, Thomas B. Preußner, and Magnus Själander. 2019. Optimizing bit-serial matrix multiplication for reconfigurable computing. *ACM Trans. Reconfig. Technol. Syst.* 12, 3 (Aug. 2019). <https://doi.org/10.1145/3337929>
- [23] Mart van Baalen, Andrey Kuzmin, Suparna S. Nair, Yuwei Ren, Eric Mahurin, Chirag Patel, Sundar Subramanian, Sanghyuk Lee, Markus Nagel, Joseph Soriaga et al. 2023. FP8 versus INT8 for efficient deep learning inference. Retrieved from <https://arXiv:2303.17951>
- [24] Mário Véstias, Rui P. Duarte, José T. de Sousa, and Horácio Neto. 2022. Efficient design of low bitwidth convolutional neural networks on FPGA with optimized dot product units. *ACM Trans. Reconfig. Technol. Syst.* 16, 1 (Dec. 2022). <https://doi.org/10.1145/3546182>
- [25] Mário P. Véstias, Rui Policarpo Duarte, José T. de Sousa, and Horácio Neto. 2017. Parallel dot-products for deep learning on FPGA. In *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. IEEE, 1–4.
- [26] Mário P. Véstias, Rui Policarpo Duarte, José T. de Sousa, and Horácio Neto. 2019. Hybrid dot-product calculation for convolutional neural networks in FPGA. In *Proceedings of the 29th International Conference on Field Programmable Logic and Applications (FPL'19)*. IEEE, 350–353.
- [27] C. S. Wallace. 1964. A suggestion for a fast multiplier. *IEEE Trans. Electr. Comput.* EC-13, 1 (Feb. 1964), 14–17. <https://doi.org/10.1109/PGEC.1964.263830>
- [28] Paul N. Whatmough, Shidhartha Das, David M. Bull, and Izzat Darzaweh. 2012. Selective time borrowing for DSP pipelines with hybrid voltage control loop. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference*. IEEE, 763–768.
- [29] Xilinx, Inc. 2016. *7 Series FPGAs Configurable Logic Block*. Xilinx, Inc. Retrieved from https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB.
- [30] Xilinx, Inc. 2017. *UltraScale Architecture Configurable Logic Block*. Xilinx, Inc. Retrieved from <https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb>
- [31] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2023. ZeroQuant: Efficient and affordable post-training quantization for large-scale transformers. *Adv. Neural Info. Process. Syst.* 35 (2023), 27168–27183.

- [32] Yuelai Yuan, Le Tu, Kan Huang, Xiaoqiang Zhang, Tiejun Zhang, Dihua Chen, and Zixin Wang. 2019. Area optimized synthesis of compressor trees on Xilinx FPGAs using generalized parallel counters. *IEEE Access* 7 (2019), 134815–134827. <https://doi.org/10.1109/ACCESS.2019.2941985>
- [33] Shien Zhu, Luan H. K. Duong, and Weichen Liu. 2020. XOR-Net: An efficient computation pipeline for binary neural network inference on edge devices. In *Proceedings of the IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS'20)*. 124–131. <https://doi.org/10.1109/ICPADS51040.2020.00026>

Received 12 September 2023; revised 22 December 2023; accepted 27 January 2024