

Parallel Accurate Minifloat MACCs for Neural Network Inference on Versal FPGAs

Hans Jakob Damsgaard[✉], *Graduate Student Member, IEEE*, Konstantin J. Hoßfeld[✉],
Jari Nurmi[✉], *Senior Member, IEEE*, and Thomas B. Preußner[✉]

Abstract—Machine Learning (ML) is ubiquitous in contemporary applications. Its need for efficient acceleration has driven vast research efforts into the quantization of neural networks with low-precision numerical formats. Models quantized with *minifloat* formats of eight or fewer bits have proven capable of outperforming models quantized into same-size integers. However, unlike integers, minifloats require accurate accumulation to prevent the introduction of rounding errors. We explore the design space of parallel accurate minifloat Multiply-Accumulators (MACCs) targeting the AMD Versal™ FPGA fabric. We experiment with three variations of the multiply-and-shift and adder tree components of a minifloat MACC. For comparison, we apply similar alterations to a parallel integer MACC. Our results show that custom compressor trees with external sign-inversion gates reduce the mean area of the minifloat MACCs by 17.7% and increase their clock frequency by 16.2%. In comparison, custom compressor trees with absorbed partial product generation gates reduce the mean area of integer MACCs by 28.1% and increase their clock frequency by 3.60%. Comparing the best-performing designs, we observe that minifloat MACCs consume 20% to 180% more resources than integer ones with same-size operands without accounting for a conversion back into a floating-point format, and 60% to 300% more resources when including it. Our data enable engineers to make informed decisions in their designs of deeply-integrated embedded ML solutions when trading off training and fine-tuning effort vs. resource cost.

Index Terms—floating-point arithmetic, field-programmable gate array, machine learning, multiply-accumulate

I. INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) are suitable for the acceleration of compute-heavy Machine Learning (ML)

Manuscript received 8 May 2024; revised 5 October 2024; and accepted 28 November 2024. Date of publication DATE MONTH YEAR; date of current version DATE MONTH YEAR.

Hans Jakob Damsgaard and Jari Nurmi are with the Faculty of Information Technology and Communication Sciences, Electrical Engineering Unit, Tampere University, Tampere, Finland, and also with AMD Research, Dresden, Germany (e-mails: hans.damsgaard@tuni.fi and jari.nurmi@tuni.fi).

Konstantin J. Hoßfeld and Thomas B. Preußner are with AMD Research, Dresden, Germany (e-mails: konstantin@hossfeld.cc and thomas.preusser@amd.com).

Hans Jakob Damsgaard and Jari Nurmi gratefully acknowledge funding from European Union's Horizon 2020 Research and Innovation Programme under the Marie Skłodowska Curie grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation, <http://www.apropos-itn.eu/>). This work was supported in part by AMD under the Heterogeneous Accelerated Compute Cluster (HACC) program, <https://www.xilinx.com/support/university/xup-hacc.html>. Work by Hans Jakob Damsgaard and Konstantin Hossfeld was carried out during internships with AMD Research.

AMD, Vivado, Versal, UltraScale, UltraScale+, and combinations thereof are trademarks of Advanced Micro Devices, Inc.

Corresponding author: H. J. Damsgaard

inference as their hardware customization features permit on-demand adaptation to individual applications. This adaptation allows for highly parallel, efficient implementations of quantized models with bespoke integer or floating-point data types [1], [2]. Their compute requirements are facilitated by structural architectural features, such as carry chains and Digital Signal Processing (DSP) slices. Along with fabric Lookup Tables (LUTs), these elements enable an adept implementation of arithmetic operations. Commercial and academic tools like Vivado™ and FloPoCo [3] ship with customizable cores or backends that tailor high-level operations to these primitives. The dot product is one such operation whose efficiency is key to applications that rely on matrix multiplication, such as neural network inference, data mining, and cryptography [4].

Computers process numeric data in either fixed-point or floating-point format. Fixed-point numbers are simple and efficient, but have a limited dynamic range as they are stored and operated on like integers. Floating-point formats resolve this limitation at the cost of increased complexity by representing numbers in scientific exponential notation. The IEEE 754 standard defines a set of base-2 floating-point formats and formalizes the behavior of the associated arithmetic operations [5]. This standard is the foundation of floating-point support in nearly all modern computer systems [6, Sec. 5.5].

We adopt the notation $\langle S, E, M \rangle$ to specify floating-point formats by the bit-widths of their sign s , exponent e , and mantissa m fields. We denote the exponent bias as ϵ and derive $e = c - \epsilon$ from the characteristic $0 \leq c < 2^E$. The appropriate selection of ϵ has been shown to positively impact quantized model accuracy [7]. However, most formats employ $\epsilon = 2^{E-1} - 1$, leading to a balanced distribution of numbers around the center point. Our work is unaffected by the choice of ϵ . With our notation, the value of *normal* numbers with $e \neq 0$ is $(-1)^s \cdot 2^{c-\epsilon} \cdot (1.m)$, whereas the value of *subnormal* numbers with $e = 0$ is $(-1)^s \cdot 2^{1-\epsilon} \cdot (0.m)$. For hardware simplicity, we deviate from IEEE conventions and choose not to represent `inf`, assuming that values outside the representation range are saturated. With `inf` excluded, multiplications and additions cannot produce NaN. We refer to floating-point formats with $S + E + M \leq 8$ as *minifloats*. Table I lists the standardized IEEE formats, established formats from the ML domain, and several recent minifloat formats. The notation used in this paper is summarized in Table II.

Due to intermediate rounding, sequential floating-point addition is not associative [6], [18], [19]. Incurred round-off

TABLE I

COMMON FLOATING-POINT FORMATS FOR ML INFERENCE. RANGE AND PRECISION (*ulp*) COMPUTED AS IN [8, SEC. 8.1], AND MINIMUM *L* AS IN (2) WITH $E_a = E_b$, $M_a = M_b$, AND $N = 1$.

Family	Format	Range	Precision	Min. <i>L</i>
Mini-float	FP4 [9]	$\langle 1, 2, 1 \rangle = 8.00e+0$	$= 5.00e-1$	9
	FP4 ¹ [9]	$\langle 1, 3, 0 \rangle \approx 2.55e+38$	$= 1.00e-0$	— ¹
	HF6 ² [10]	$\langle 1, 4, 1 \rangle \approx 9.83e+4$	$= 5.00e-1$	33
	SFP [11]	$\langle 1, 3, 3 \rangle = 1.92e+3$	$= 1.25e-1$	21
	HFP8 ³ [12]	$\langle 1, 4, 3 \rangle \approx 4.92e+5$	$= 1.25e-1$	37
	HFP8 ³ , FP8 ⁴ [12]–[14]	$\langle 1, 5, 2 \rangle \approx 1.50e+10$	$= 2.50e-1$	67
ML	FP16 [14]	$\langle 1, 6, 9 \rangle \approx 9.44e+21$	$\approx 1.95e-3$	145
	BF16 [15], [16]	$\langle 1, 8, 7 \rangle \approx 1.48e+79$	$\approx 7.81e-3$	525
	TF32 [17]	$\langle 1, 8, 10 \rangle \approx 1.19e+80$	$\approx 9.77e-4$	531
IEEE	binary16 [5]	$\langle 1, 5, 10 \rangle \approx 4.40e+12$	$= 9.77e-4$	83
	binary32 [5]	$\langle 1, 8, 23 \rangle \approx 9.71e+83$	$\approx 1.19e-7$	557
	binary64 [5]	$\langle 1, 11, 52 \rangle \approx 1.46e+632$	$\approx 2.22e-16$	4199

¹This base-4 format is incompatible with our MACCs. ²Proposed only for weights with binary32 activations. ³Proposed for all tensors with binary32 accumulators. ⁴Proposed for all tensors with FP16 accumulators.

errors accumulated in long dot products can have significant negative consequences for minifloats. The rounding impact can be mitigated by widening the addition in a fused multiply-add unit with terminal rounding [10], [12]–[14]. In the ML domain, some authors instead employ fixed-point accumulators that introduce risks of over- and underflow if not properly managed [20]. Only a few studies consider fully accurate *Kulisch* accumulation because of its often unreasonable costs [1].

Recent AMD FPGAs offer varying degrees of support for floating-point arithmetic. The DSP48E1/2 primitives in 7 Series and UltraScale+™ devices integrate 48-bit adders and 18×27 -bit multipliers with no floating-point support [21]. The DSP58 primitives in Versal™ devices have 58-bit adders and 24×27 -bit multipliers and provide hardened support for the IEEE binary16 and binary32 formats [22]. Regardless, DSPs are scarce in comparison to LUTs (depending on the FPGA series, roughly 1:400 to 1:200 [21], [22]) and using them may not be resource-efficient for all floating-point formats. Without native support for custom floating-point formats, the usefulness of DSPs is limited to mantissa multiplication, with accumulation happening in the fabric or in additional DSPs. Moreover, minifloat formats involve low bit-width multiplications that require much fewer than 200 LUTs to implement and render DSP packing schemes inefficient [23].

Our aim is to explore the practical feasibility of minifloats in comparison to integers for ML inference from a hardware perspective. To do so, we perform a design-space exploration into parallel Multiply-Accumulators (MACCs) with *Kulisch* accumulation in FPGA fabric resources, i.e., LUTs and registers. We consider two baseline parallel minifloat and integer MACC designs, experiment with reducing their area and latency through segmentation and custom compressor trees, and report on the discovered insights. Our contributions are:

- A library of custom-precision parallel minifloat MACCs;
- An exploration of optimizing the multiply-and-shift and adder tree parts of the MACCs and the attained effects on their area and latency;

TABLE II

SYMBOLS AND NOTATION USED IN THIS PAPER.

Description	Notation
Sign, exponent, and mantissa bit-widths	S, E, M
Sign, exponent, and mantissa fields	s, e, m
Characteristic	c
Exponent bias	ϵ
Multipliers and multiplicands	a_i, b_i
Products	p_i
Number of input lanes	N
Accumulator bit-width	L
Segment bit-width	l'
Number of segments	G

- An algorithm and a corresponding hardware architecture for converting *Kulisch* accumulators to floating-point formats with a given precision; and
- A hardware cost evaluation of increasing the exponent bit-width when switching from integers to minifloats.

Our experiments show no consistent benefits from segmenting the minifloat MACC shifter. Contrarily, the MACCs benefit greatly from custom compressor trees with external sign-inversion gates, demonstrating 17.7% lower mean LUT utilization and 16.2% higher clock frequency than the baseline design. Absorbing the sign-inversion gates into the compressor trees achieves only a 10.2% reduction in the mean LUT utilization with no effect on latency. In comparison, the integer MACCs benefit the most from compressor trees with absorbed partial product generation gates, reporting 28.1% lower mean LUT utilization and 3.60% higher clock frequency. Finally, by comparing the best-performing designs, we see that minifloat MACCs for formats useful in inference [2] consume 20% to 180% more LUTs than integer ones with equal-width operand formats. Accounting for conversion back into a floating-point output increases the overhead to between 60% and 300%.

The paper is structured as follows: Section II covers related work on post-training quantization with minifloats, *Kulisch* accumulation, and dot product designs for FPGAs. Section III presents the baseline parallel minifloat MACC and its integer counterpart. Section IV describes the options considered for optimizing the minifloat MACC’s area and latency, while Section V covers the proposed *Kulisch*-to-minifloat converter. Section VI evaluates the optimization options against one another, compares them to a similarly optimized integer MACC, and explores the overhead incurred when integrating conversion. Section VII concludes the paper.

II. RELATED WORK

This section introduces post-training quantization with minifloats, accurate accumulation of floating-point numbers, and recent dot product architectures for FPGAs.

A. Post-Training Quantization with Minifloats

Machine learning models are commonly trained in the binary32 format. Upon training, the models may be projected into lower-precision formats to reduce model size and improve inference performance [24]–[26]. Training models with emulated lower-precision formats is possible but has obvious time

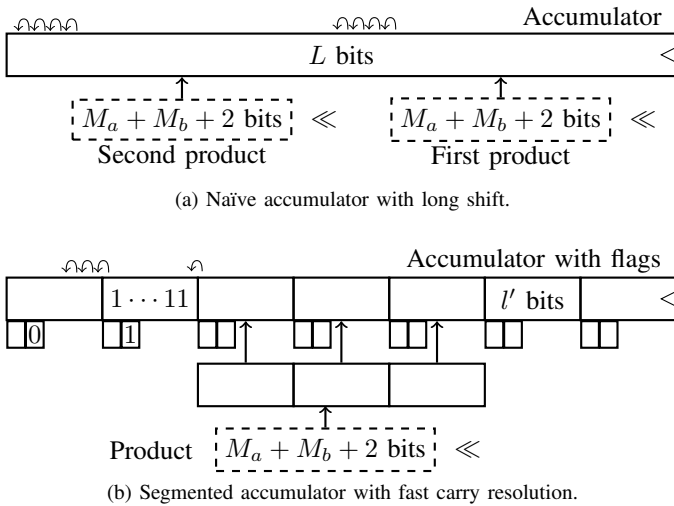


Fig. 1. Examples of Kulisch accumulator designs with (a) a long shift and adder [6, Sec. 8.4.1], which may suffer from high latency, and (b) segmentation and additional control to reduce this latency [6, Sec. 8.4.4].

overheads [26]. Post-training quantization methods avoid these overheads and incur low additional costs from fine-tuning [27]. Most methods involve scaling and clipping weights and activations to fit the range of the selected data formats [2], possibly considering the accumulator bit-width [20]. Not all models are equally tolerant to quantization. Some may require focusing on model elements at a specific granularity, such as groups of channels, individual channels, or sub-channel elements [26].

Recent work has demonstrated that minifloat quantization is feasible and can improve model accuracy compared with integers [2], [12], [28]. The preferred quantization methodology is much the same as for integers and happens after training. It permits optimization with learned or one-shot strategies, like bias correction [24], gradient-based learned rounding [27], and GPTQ [25], but requires that the hardware does not introduce any errors not known during quantization. In this regard, binary32 accumulators are accurate for most models quantized with minifloats but may cost more hardware resources than accurate fixed-point accumulators. Experiments highlight that weights require high precision, while activations need high dynamic range to retain model accuracy [2]. Prior work made similar observations for the $\langle 1, 4, 3 \rangle$ and $\langle 1, 5, 2 \rangle$ formats [12]. FPGAs are the only platform architectures that can efficiently harness the benefits of flexible custom low-precision formats.

B. Accurate Floating-Point Accumulation

The efficient implementation of floating-point accumulation is crucial for many applications [6], [29], but it is complicated by error accumulation [18], [19]. Much work has considered mitigating these errors with modified floating-point units [30], [31]. For example, the FAAC directs addends to two separate accumulators according to their sign and combines these opposite-sign subtotals only at the end of an accumulation [18]. The parallel CAP and the UAA both aim for maximum throughput over minimum error [32], [33]. The FPAR adder maintains residuals to guarantee no rounding errors at the expense of increased area [19].

TABLE III

COMPARISON OF FPGA PLATFORMS, PRIMITIVES (LUTs, DSPs, OR Both), DATA TYPES (INTEGER, FLOATING-POINT, OR Both), PARALLELISM, AND ACCUMULATION STYLE IN THE RELATED WORK.

Design	Refs.	Platforms	Primitives	Types	Parallel?	Kulisch?	
MACC	[29]	AMD Virtex E	LUTs	Both	No	No	
	[31]	Intel Stratix III	Both	FP	No	No	
	[18]	AMD Virtex II, Virtex 5	LUTs	FP	No	No	
	[35]	AMD Virtex 5	LUTs	FP	No	No	
	[32]	AMD Virtex 5	—	FP	No*	No	
	[19]	AMD Virtex 6	LUTs	FP	No*	No	
	[33]	AMD Virtex 7, Intel Stratix V	Both	FP	Yes	No	
	[30]	AMD Kintex 7	Both	FP	No	Yes	
	Dot Product	[36]	AMD Virtex 6	Both	FP	Yes	No
		[37]	AMD Virtex 6	Both	FP	No	Yes
[38]		AMD Artix 7, Virtex 7, Kintex 7	DSPs	I	Yes	No	
[39]		Intel Arria 10, Stratix 10	Both	FP	Yes	No	
[40]		AMD Zynq 7000	Both	I	Yes	No	
[41]		AMD Kintex 7	Both	I	Yes	No	
[42]		AMD Versal	LUTs	I	Yes	Yes	
Ours		AMD Versal	LUTs	Both	Yes	Yes	

*Cascadable to handle multiple parallel inputs.

None of the above architectures mitigates both rounding and re-ordering errors. This motivates the alternative strategy of using *exact* fixed-point accumulators, as proposed in the book “*Computer Arithmetic and Validity: Theory, Implementation, and Applications*” and nicknamed *Kulisch accumulation* after its author [6]. Assuming a fixed-point accumulator long enough to store n products of any two operands in a particular floating-point format, no bits are lost and the accumulation is exact. The dependence on fixed-point addition renders it attractive compared to pure floating-point designs [18], [19], [32], [33] when targeting parallelization in an FPGA, as the concatenation of several fixed-point products conforms well with the fabric’s support for high-efficiency compressor trees. Moreover, its resource efficiency approaches that of integers for small-enough floating-point formats [1].

The design space for Kulisch accumulators is large. Most implementations are based on the foundational architectures proposed by Kulisch [6] and differ in how they handle carries. The simplest design illustrated in Fig. 1a integrates a long shift and adder, avoiding some control logic at the expense of high latency. As shown in Fig. 1b, segmentation can shorten the potential carry path and reduce the need for shifting. In such designs, the segment width l' is chosen as the smallest power-of-2 greater than the width of the largest operand significand $\max(M_a + 1, M_b + 1)$. This choice ensures that a product of width $M_a + M_b + 2 \leq 2l'$ spans at most three segments. The carries out of the selected segments may be propagated by keeping local counters and iteratively resolving the sum at the end of a dot product [6], [30], by storing the accumulator in a carry-save format [34], or by using flags to indicate that a segment is all-zeros or all-ones [6]. For formats larger than minifloats, pipelining is essential for performance [30].

C. Dot Products in FPGAs

Interest in efficient dot product implementations for FPGAs has grown with the rise of modern ML. However, most existing work listed chronologically in Table III focuses on single-input MACCs. The UAA covered above is the only exception, whose parameterizable adder tree enables architectural support for input parallelism [33]. The CAP and FPAR architectures can be trivially parallelized using multiple standalone MACCs, possibly foregoing the benefits of their fusion [19], [32].

Unlike existing work, which uses FPGAs for prototyping or for comparability between designs [36], [37], we consider them as the target platform. This implies making as efficient use of their hardware primitives as possible. However, doing so may require intricate domain knowledge, for example, illustrated by the two-level, Block RAM (BRAM)-based memory scheme used to store the Kulisch accumulator in [36], the operand packing scheme for DSPs in [38], and the hybrid LUT-DSP dot product architectures of [39]–[41].

Our work stands out as it is the first to explore the design of MACCs with Kulisch accumulation for minifloats and only the second work to consider the AMD Versal™ FPGA fabric. Notably, with our focus on using purely LUTs and registers, the latter point is interesting, as the Versal fabric has fundamental differences from its predecessors, the 7 Series and UltraScale+™ architectures. The Versal architecture adds support for chaining neighboring LUTs via a fast, slice-internal cascade path. It also consolidates the carry chain logic from the 7 Series and UltraScale+ architectures into a new carry-lookahead primitive without hardened XOR gates [43]. These differences affect the logic and arithmetic it can implement and the optimization opportunities available to us [42].

III. BASELINE MACC ARCHITECTURES

To explore the efficiency of minifloat MACCs [1], we first present a baseline design with multiple input lanes and Kulisch accumulation. Throughput is crucial for the inference performance of recent ML models with very wide dot products, such as, ResNet-18 whose convolutional layers require up to 4608 products [44]. Hence, we require an initiation interval for the MACC of one clock cycle but impose no constraint on its latency. We focus on an architecture that resembles the one shown in Fig. 1a and maintain the accumulator as a flattened two's complement number because the carry logic in the FPGA is fast and interrupting it is unlikely to bring any benefits at the accumulator widths relevant in our context [42].

Our baseline MACC design is illustrated in Fig. 2. Its Hardware Description Language (HDL) description is fully parameterized by the signedness, exponent and mantissa widths of the operands, the number of parallel input lanes, and the accumulator width. While the configurations are homogeneous across all input lanes, pairs of operands may be asymmetric. As we do not model NaNs, an operand is converted internally into an accurate integer representation as follows:

$$\begin{cases} (-1)^s \cdot (1.m) \cdot 2^{c-1} \cdot 2^M, & \text{for } c \neq 0 \\ (-1)^s \cdot (0.m) \cdot 2^M, & \text{for } c = 0. \end{cases} \quad (1)$$

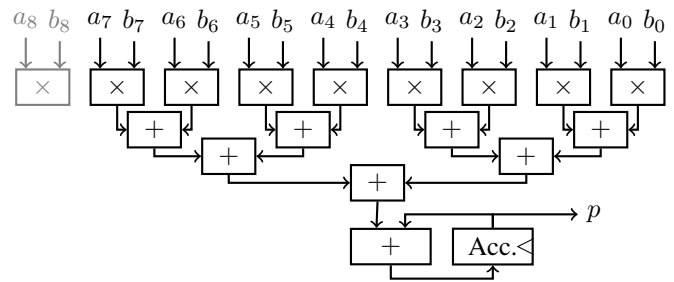


Fig. 2. Our baseline parallel minifloat and integer MACC structure.

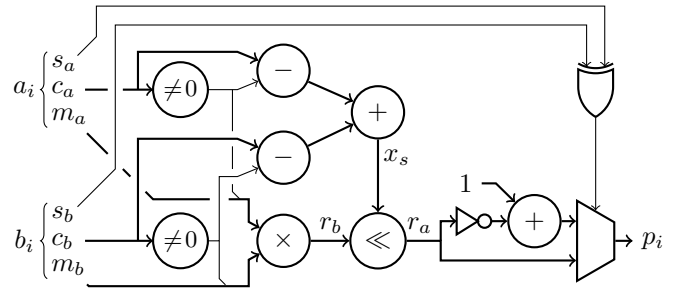


Fig. 3. Block diagram of our proposed minifloat multiply-and-shift block.

These equations resemble the ones in Section I, but differ in that the decimal point is shifted to the end of the mantissa. With this interpretation, the operand configurations $\langle S_a, E_a, M_a \rangle$ and $\langle S_b, E_b, M_b \rangle$ across N parallel input lanes demand a default accumulator width of:

$$L_{\text{default}} = 2^{E_a} + M_a + 2^{E_b} + M_b + \lceil \log_2 N \rceil - 1. \quad (2)$$

Following (1), the product of two operands may be encoded into the narrow integer product of their significands followed by a left shift by the sum of their exponents. The encoding of subnormal numbers can be rewritten as $(-1)^s \cdot (0.m) \cdot 2^{c-0} \cdot 2^M$ with 0s in the positions of the 1s for normal numbers. This permits the trivial reuse of the 1-bit result of $c \neq 0$ as input to both the significand multiplier and the subtractor needed to compute the shift amount $c_a - (c_a \neq 0) + c_b - (c_b \neq 0)$. The shifted product is selectively inverted depending on its sign $s_a \oplus s_b$. Fig. 3 details the multiply-and-shift logic, which constitutes one block of the minifloat MACC, marked by \times in Fig. 2. The products from each input lane are fed to a binary adder tree and the final sum is added to the accumulator.

Our initiation interval constraint prevents the use of techniques that involve finalizing carry propagation over a data-dependent number of cycles at the end of a dot-product computation [30], as described in Section II-B. This constraint also limits our options for timing optimization of the adder tree and the final adder to pipelining only. Nevertheless, the resulting architectures are feasible due to the relatively small accumulator widths needed to support minifloats [30] and the fast hardened carry logic available in AMD FPGAs [42]. In fact, our initial experiments showed that Vivado™ automatically converts the behavioral description of a binary adder tree into a compressor with a single terminal adder. In practice,

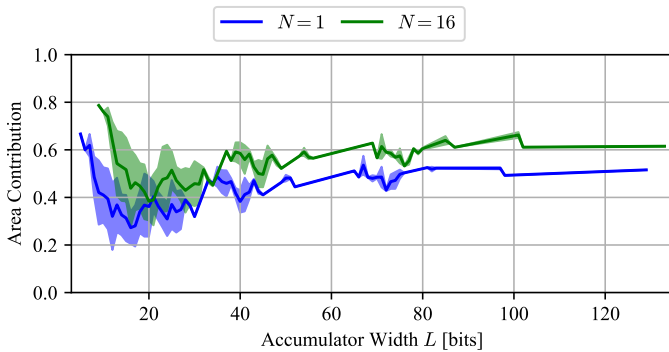


Fig. 4. Fraction of the total LUT utilization of the adder tree and accumulator sub-module in the split non-pipelined baseline design with $N = 1$ and $N = 16$ vs. accumulator width L .

this implies that only a single cascade of LOOKAHEAD8 primitives is instantiated, even for 16-lane designs.

Our baseline integer MACC follows the same structure as the minifloat MACC shown in Fig. 2, except that it implements simple multipliers in place of the minifloat MACC’s multiply-and-shift logic. As we will see later, this greatly impacts its susceptibility to optimization with custom compressor trees. We use SystemVerilog constructs for the sign-extension of the parallel products needed when at least one operand is signed.

IV. OPTIMIZATION STRATEGIES

The considered minifloat MACC is largely composed by two parts: A product generator that hosts the N multiply-and-shift blocks, and an adder tree that includes both the sign-inversion adders and the accumulator. These parts contribute differently to the area of a particular MACC configuration and represent diverse opportunities for optimization. To illustrate this point, we manually split the non-pipelined baseline design into these two parts. We apply the `keep_hierarchy = "yes"` attribute to avoid inter-module optimizations and perform implementation targeting a Versal™ FPGA at an accessible clock frequency of 25 MHz.

Fig. 4 presents the fraction of the resulting LUT utilization consumed by the combined adder tree and accumulator sub-module at different accumulator widths. We focus only on designs with $N = 1$ or $N = 16$ input lanes as they host the smallest and the largest adder trees in our chosen design space. The shaded areas in the plot illustrate the range of results for different MACC configurations with the same accumulator width, while the lines show the mean. For example, the configurations $\langle 1, 3, 3 \rangle \times \langle 1, 2, 5 \rangle$ and $\langle 1, 4, 1 \rangle \times \langle 1, 1, 1 \rangle$ both have $L = 19$ when $N = 1$. The results show that the two sub-modules contribute almost equally to the overall area, rendering them both good subjects for optimization. The adder tree and accumulator consume a larger part of the overall area in designs with multiple lanes, meaning that these designs are likely to see larger benefits from custom compressor trees.

Motivated by these observations, we explore three alterations to our baseline MACC. The first idea is tailored to the multiply-and-shift logic and targets a complexity reduction by

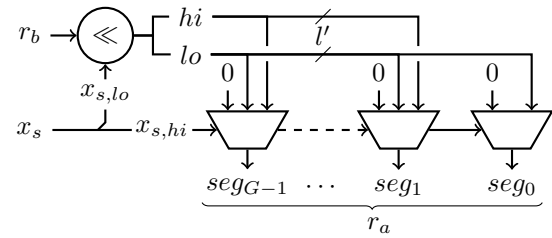


Fig. 5. Block diagram of our proposed segmented shifter.

segmenting the mantissa product shifter to perform a fine-grained shift and a coarse selection based on enable logic. The remaining two ideas are equally applicable to the adder trees in the minifloat and integer MACCs and involve custom compressor trees targeted to the Versal FPGA fabric.

A. Segmented Shifter

The multiplier, comparators, and other arithmetic units in the multiply-and-shift block, shown in Fig. 3, have relatively narrow bit-widths. In contrast, the shifter required to position the mantissa product is as wide as the accumulator and contributes significantly to the area and latency of a lane. This issue is particularly pertinent to configurations with long accumulators. We attempt to counter this with inspiration from the accumulator architecture in [30] and [6, Sec. 8.4.4], shown in Fig. 1b, and apply segmentation with a multiplexer structure when $L \geq 32$ to reduce the width of the dynamic shifter, as shown in Fig. 5. r_b and r_a represent the mantissa product before and after the shifter as shown in Fig. 3.

As described above, we choose the segment width l' as a power-of-2 to extract the fine-grained shift amount ($x_{s,lo}$) and the first segment index ($x_{s,hi}$) as the least significant $\log_2 l'$ bits and the remaining bits of the exponent sum x_s trivially. In practice, we let $l' = 8$ when $M_a + M_b + 2 \leq 8$ and $l' = 16$ otherwise, meaning a total of $G = \lceil L/l' \rceil$ segments. For the considered minifloat formats with $1 \leq M \leq 6$ and thus $4 \leq M_a + M_b + 2 \leq 14$, these choices of l' ensure that the fine-grained shift of r_b by $x_{s,lo}$ positions remains within the bounds of two segments lo and hi , i.e., $2l' \leq 32$ bits. We expect this alteration to reduce latency, possibly at the cost of some hardware overhead.

B. Compressor Trees with External Gates

The adder tree of our parallel MACCs naturally lends itself to optimization with custom compressor trees. Therefore, we use the compressor generator from [42] to decrease the area of the bit matrix reduction and accumulation. This involves different alterations to the two MACC designs. For the minifloat MACC, we remove the long adder needed for the sign-inversion in each lane and integrate it with the bit matrix of the compressor tree, including the sign bits in the last column of the matrix, as illustrated in Fig. 6a. Similarly, we implicitly integrate the final adder into the compressor tree. The lanes otherwise retain their structure.

For the integer MACC, we replace the blocks marked by \times in Fig. 2 with manually constructed Baugh-Wooley partial

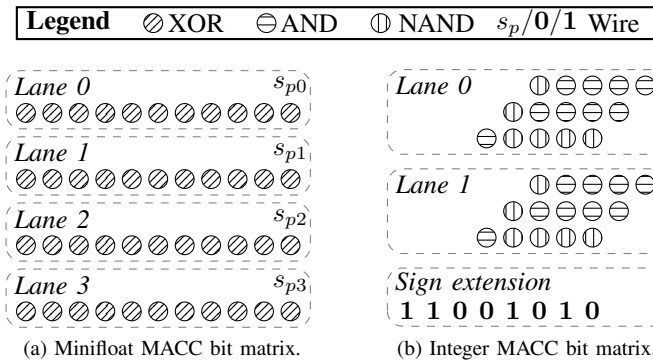


Fig. 6. Compressor tree bit matrices for (a) the minifloat MACC with $N = 4$, $L = 11$, and $\langle S_a, E_a, M_a \rangle = \langle S_b, E_b, M_b \rangle = \langle 1, 2, 1 \rangle$ operands, and (b) the integer MACC with $N = 2$, $L = 10$, and 3×5 -bit signed operands. The patterns and legend describe the types of absorbed gates.

product arrays [45], whose concatenation naturally forms the input to a compressor tree. Moreover, rather than letting the synthesis tool sign-extend the products from each lane before their summation, we generate and sum the needed constants manually into one common bit vector and include it in the resulting bit matrix, as exemplified in Fig. 6b.

Prior work [42] has shown geometric mean savings of $\sim 6\%$ LUT utilization of signed integer MACCs with the same design space constraints as those in the present paper. These savings are unlikely to translate wholly to the minifloat MACC, given its relatively higher fraction of hardware external to the compressor tree. As a result, assuming that the adder tree on average takes up 50% of the overall area, we expect roughly $0.5 \cdot 6\% \approx 3\%$ LUT utilization reductions in the minifloat MACCs with little-to-no effect on latency.

C. Compressor Trees with Absorbed Gates

Most of the inputs to the compressor trees result from two-input gates. These gates are either ANDs or NANDs needed to produce partial products in the integer MACC or XORs needed to selectively invert products in the minifloat MACC, as illustrated with patterns in Fig. 6. Combining this observation with the fact that most feasible counters for the Versal FPGAs fabric are output-constraint [42] and some, thus, do not saturate the inputs of their LUTs led us to consider absorbing these gates into the compressor tree¹.

This absorption is non-trivial, as not every feasible counter has enough vacant inputs to accommodate the extra two-input gates. This is possible for ripple-carry adders, with the exception of their carry ports (denoted by transfer-in t_{in} and transfer-out t_{out}), as shown in Fig. 7, and standalone (3 : 2) counters [42]. In other words, the external-gate and absorbed-gate bit matrices have identical shapes, but the feasible counters for the first compression stage differ. Using less efficient counters in the first stage leaves more bits for compression in subsequent stages, which may compromise the latency and resource reductions. Nevertheless, absorbing the

¹Synthesis tools like Vivado™ already commonly perform such optimizations. However, the manually instantiated LUTs output by the compressor generator [42] hinder such an optimization between, say, a minifloat lane and the custom compressor tree in our MACCs, thus demanding our intervention.

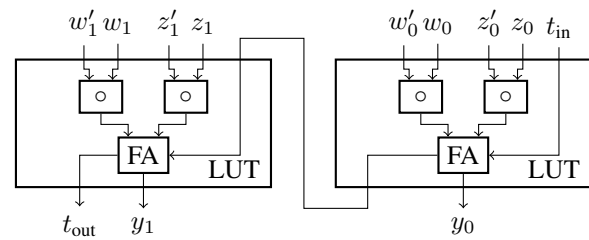


Fig. 7. Two stages of a ripple-carry adder with absorbed two-input gates. Each operand bit z_i and w_i is associated with a gating bit z'_i and w'_i .

Algorithm 1 Kulisch accumulator to floating-point conversion algorithm. $\{i_0, \dots, i_{n-1}\}$ denotes a concatenation of vectors i_0 to i_{n-1} , and $I\{x\}$ denotes the concatenation of I copies of x . Signals have widths given in their declaration.

Require: $M_p \geq 0$ and $L > 0$

- 1: $E_p \leftarrow \max(1, \lceil \log_2[L - M_p + 1] \rceil)$
- 2: $p_{in}[L - 1 : 0] \leftarrow$ Kulisch accumulator
- 3: $s_p \leftarrow p_{in}[L - 1]$
- 4: $\{c_p, m_p\} \leftarrow \{0, 0\}$
- 5: **if** $M_p \geq L$ **then** ▷ Accumulator fits in mantissa.
- 6: $m_{int}[M_p - 1 : 0] \leftarrow$ sign-extend(p_{in})
- 7: $\{c_p, m_p\} \leftarrow \{0, \overline{m_{int}}\} + 1 ? s_p : \{0, m_{int}\}$
- 8: **else** ▷ Accumulator does not fit in mantissa.
- 9: $W_{lod} \leftarrow L - M_p$
- 10: **S1** $\{lop[\lceil \log_2 W_{lod} \rceil - 1 : 0], zero\} \leftarrow$
- 11: $\text{leading-one}(p_{in}[L - 1 : M_p] \oplus \{W_{lod}\{s_p\})\}$
- 12: $m_{int}[M_p - 1 : 0] \leftarrow p_{in} \gg lop$
- 13: $c_{int}[E_p - 1 : 0] \leftarrow \{0, lop\} + \overline{zero}$
- 14: **S2** $m_{sum}[M_p : 0] \leftarrow \{0, \overline{m_{int}}\} + 1 ? s_p : \{0, m_{int}\}$
- 15: $c_{sum}[E_p - 1 : 0] \leftarrow c_{int} + m_{sum}[M_p]$
- 16: $\{c_p, m_p\} \leftarrow \{c_{sum}, m_{sum}[M_p - 1 : 0]\}$
- 17: **end if**
- 18: **return** $\{s_p, c_p, m_p\}$

aforementioned gates lets the entire integer MACC be included in the compressor tree, while it forms a hard boundary between the shifters and the compressor tree in the minifloat MACC.

Compressor trees with absorbed gates have been shown to reduce the geometric mean LUT utilization of signed integer MACCs by $\sim 19\%$ [42]. This is largely a result of the integration of the entire integer MACC with the compressor tree. For the minifloat MACC, the affected area is, again, limited to the adder tree and accumulator parts that follow the shifters. Therefore, aligned with the above, we expect roughly $0.5 \cdot 19\% \approx 9.5\%$ LUT utilization reduction in the minifloat MACCs with no effect on latency.

V. KULISCH TO MINIFLOAT CONVERSION

In most cases, our minifloat MACC will be fused with the activation function of a neural network layer. With the multi-thresholding approach, which is common for quantized neural network implementations [46], no explicit reverse conversion of the fixed-point accumulator into a floating-point format is required. Still, we find it relevant to propose a suitable converter architecture and evaluate its potential overhead.

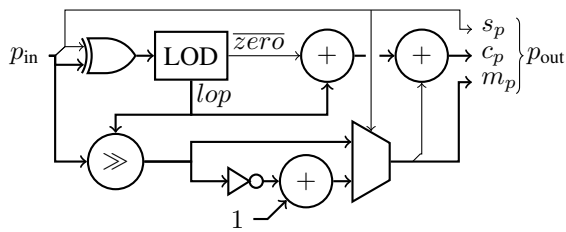


Fig. 8. Block diagram of our Kulisch to minifloat converter for $L > M_p$.

Our proposed converter architecture is parameterized by the accumulator width L and the desired number M_p of mantissa bits in the output format. We assign the output format the minimum number of exponent bits needed to cover the range of the accumulator $E_p = \max(1, \lceil \log_2[L - M_p + 1] \rceil)$. The final sign s_p is trivially extracted as our minifloat MACC maintains its accumulator as a flattened two's complement number. However, because the mantissa of a floating-point number is stored in sign-magnitude format, negative accumulators must have their sign inverted during conversion.

Our conversion algorithm is listed in Algo. 1. It comprises two steps: **S1**) estimate the exponent e_p using the most-significant $L - M_p$ bits of the accumulator p_{in} , and extract the mantissa in two's complement m_{int} by shifting the accumulator right by lop positions and selecting its least-significant M_p bits; and **S2**) invert the sign of the mantissa m_{int} if the accumulator is negative, and increment the estimated exponent e_{int} if the sign-inversion overflows. This increment is guaranteed not to overflow as a result of choosing the proper exponent bit-width² E_p , which lets us avoid including additional logic to support saturation. The algorithm is trivially reduced to just **S2** with $E_p = 1$ if the accumulator fits entirely within the mantissa of the targeted floating-point format, i.e., $L \leq M_p$.

Fig. 8 illustrates the converter architecture for the likely case when $L > M_p$ with signals labeled as in Algo. 1. We use a leading-one detector modeled after the `arith_firstone` module of [47]. This detector generates a one-hot-encoded bit vector from its input using reversion and a long adder and selects the index of the asserted bit with a binary-tree-style OR reduction. Its architecture is known to map well to the hardened carry propagation logic in AMD FPGAs.

VI. EVALUATION

In the following evaluation of the above design options, we use Vivado™ 2023.1 with default settings, targeting the Versal™ xcvc1902-vsva2197-2MP-e-S part that is found on the VCK190 Evaluation Platform. The post-implementation LUT consumption constitutes our area metric. For timing evaluation, we wrap our designs in (shift) registers to ensure proper isolation and avoid exceeding the I/O limitations of the selected FPGA. We perform interval nesting on the timing goal and report the critical path delay of the tightest successful timing closure within 0.1 ns of a failed attempt. To avoid any unintentional use of DSPs, we annotate the baseline integer MACC with the `use_dsp = "no"` attribute.

²Appendix A provides a brief proof of this feature.

Unless otherwise specified, all designs are pipelined to the depth specified by the compressor generator for their configuration with one compression stage per pipeline stage. The MACC with the segmented shifter is pipelined identically to the MACC with optimized compressor trees. Our evaluation takes into account that certain configurations of the compressors with absorbed gates require one more compression stage than those with external gates. In practice, the pipeline depth ranges from zero to four in the minifloat MACCs, and from zero to six in the integer MACCs.

We manually insert pipeline registers into the adder tree of the baseline design, while they are automatically positioned between the compression stages by the compressor generator in the altered designs. Regardless, we let Vivado move and duplicate registers by enabling the `global_retiming` parameter in all experiments. This parameter is automatically enabled for synthesis targeting Versal FPGAs [48]. We have verified that retiming works as expected across manually instantiated LUTs when they are assigned the `dont_touch = "true"` attribute and within modules assigned the `keep_hierarchy = "yes"` attribute.

We limit the configuration space and consider only signed MACCs with a number of input lanes $N = 2^i$ for $i \in \{0, 1, \dots, 4\}$ and exponent and mantissa bit-widths $E \geq 1$ and $M \geq 1$. With these restrictions, we arrive at 2205 configurations of the minifloat MACC and 180 configurations of the integer MACC, including designs with symmetric operand formats. The possible accumulator widths range from 5 to 133 bits for minifloats and from 7 to 21 bits for integers. We occasionally observe insignificant differences in the resource consumption of the MACCs with symmetric operand formats, but attribute these to randomness in the optimization heuristics that Vivado applies during synthesis and implementation. As in Section III, the shaded areas in the plots indicate the range of results for different MACC configurations with the same accumulator width. All plots show the baseline design with solid lines and the altered designs with dashed lines.

A. Reference Designs

As a reference for our designs, we implement an instance of the AMD LogiCORE Floating-Point Operator core [49]. LogiCORE cores are limited to a single input lane with support for custom floating-point formats with $E \geq 4$ and $M \geq 3$. As there is no MACC core, we connect two multiply and accumulator cores. We configure the cores for high speed with the minimum precision $\langle S_a, E_a, M_a \rangle = \langle S_b, E_b, M_b \rangle = \langle 1, 4, 3 \rangle$ format and no DSP usage. Compared to our MACC, the resulting design is a much deeper (25-stage) inaccurate pipeline that truncates products prior to accumulation.

In the following, the reference MACC is represented by a dashed black line in all the minifloat-related plots. Table IV lists its LUT utilization and maximum clock frequency when wrapped in registers. Internally, the LogiCORE MACC utilizes a long fixed-point accumulator, which we assign the default width defined for the selected formats in (2). It also features a conversion stage similar to ours that transforms the accumulator into a floating-point number in the input format

TABLE IV
RESOURCE UTILIZATION AND MAXIMUM CLOCK FREQUENCY FOR THE REFERENCE MACCS WITH $N = 1$, $L = 37$, AND $\langle S_a, E_a, M_a \rangle = \langle S_b, E_b, M_b \rangle = \langle 1, 4, 3 \rangle$ OPERANDS.

Design	LUTs	Max. Freq. [MHz]
LogiCORE Floating-Point Operator	443	881.8
FloPoCo FPDotProduct	87	847.4

*Custom floating-point format without subnormals but two designated *extra* bits to explicitly differentiate normal numbers and zero, *inf*, and NaN values.

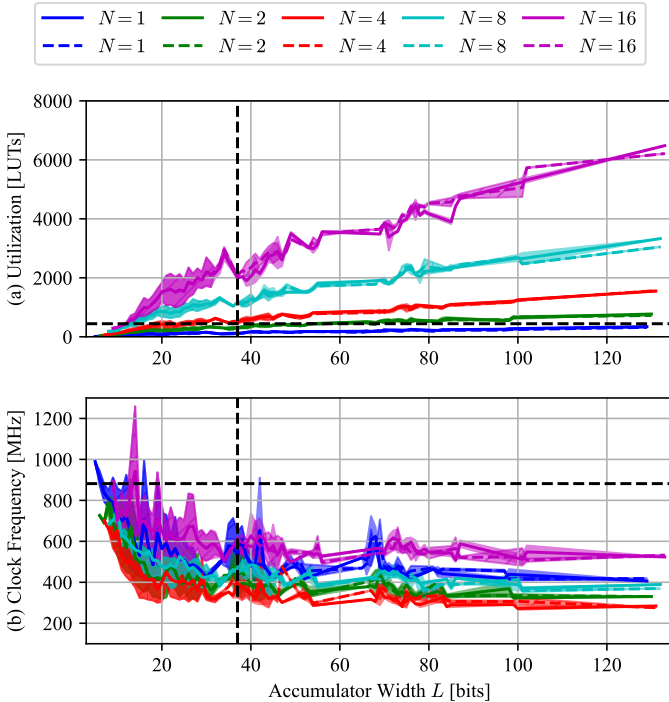


Fig. 9. Effects of the segmented shifter (dashed lines) on (a) LUT utilization and (b) maximum clock frequency of the minifloat MACCs.

after accumulation. Note that unless otherwise specified, the results presented for our MACCs in this section assume the thresholding-based implementation introduced in Section V and thus do *not* include our proposed converter.

Another conceivable reference implementation for a floating-point MACC can be generated by FloPoCo. For the very purpose pursued by us, FloPoCo offers the FPDotProduct operator [50]. It allows the accumulation of a *custom* floating-point format into an exact Kulisch-style accumulator. The produced solution achieves an attractive resource footprint and a similar operating frequency in comparison to the LogiCORE operator. However, these metrics are attained at the cost of using a simplified floating-point format that treats zero, *inf* and NaN values as exceptional cases identified by two bits added to the encoding. Subnormal numbers are not supported at all. This choice breaks the typical decrease of the step size between valid floating-point encodings when approaching zero, leaving a larger gap between the representable values around the value of zero. This circumstance renders FloPoCo’s simplified floating-point format an ill fit for ML applications. For this reason, we do not include FloPoCo designs in the following sections. We

TABLE V
GEOMETRIC MEAN CHANGES IN LUT UTILIZATION OF THE MINIFLOAT MACCS WITH THE PROPOSED ALTERATIONS VS. THE BASELINE. SEGMENTED SHIFTER RESULTS ARE FOR $L \geq 32$ ONLY.

Configuration	Number of input lanes N					
	All	1	2	4	8	16
Segmented shift	0.988	0.946	0.951	0.970	1.004	1.066
External gates	0.823	1.008	0.752	0.808	0.729	0.847
Absorbed gates	0.898	1.212	0.864	0.801	0.865	0.804

TABLE VI
GEOMETRIC MEAN CHANGES IN MAXIMUM CLOCK FREQUENCY OF THE MINIFLOAT MACCS WITH THE PROPOSED ALTERATIONS VS. THE BASELINE. SEGMENTED SHIFTER RESULTS ARE FOR $L \geq 32$ ONLY.

Configuration	Number of input lanes N					
	All	1	2	4	8	16
Segmented shift	1.007	1.008	1.018	1.035	0.999	0.979
External gates	1.162	1.485	1.356	1.409	1.044	0.717
Absorbed gates	0.996	1.101	1.302	0.966	0.758	0.933

used the latest stable version 4.1 of FloPoCo. The desired FPDotProduct operator has not been pulled along with the remaining development of the repository. Even in that stable version, the tool installation requires manual intervention, and the generated VHDL code must be fixed for syntax errors. All fixes and workarounds were conducted in the spirit of the original publication and with re-confirmation from its authors.

B. Alternative Shifting Strategy

We first explore the impact of our proposed segmented shifter for designs with $L \geq 32$ bits. Fig. 9 compares the resulting LUT utilization and maximum clock frequency to the baseline MACC. The substantially deeper pipeline of the reference LogiCORE MACC allows it to achieve a much higher maximum clock frequency than the proposed MACCs. However, its LUT utilization budget can accommodate all single-lane configurations of our MACC, even those with accumulators three times as wide.

Contrary to our expectation of reduced latency, we observe negligible LUT utilization and clock frequency impacts from the segmented shifter. This observation is underlined by the detailed results in Tables V and VI, which report numbers within $\pm 1\%$ of the baseline for the relevant designs with $L \geq 32$. Our hypothesis is that the shifter synthesized by Vivado for the baseline design already exploits segmentation, implying that it is sufficiently similar to our segmented shifter to render the designs indistinguishable after implementation. Given this lack of consistent benefits, we avoid using the segmented shifter in subsequent experiments.

C. Compressor Trees with External Gates

For these experiments, we instruct the compressor generator from [42] with signatures generated in TCL. We reduce the HDL description of the minifloat MACC to produce possibly sign-inverted, shifted mantissa products and connect them to the flattened inputs of the compressor tree. Fig. 10 presents

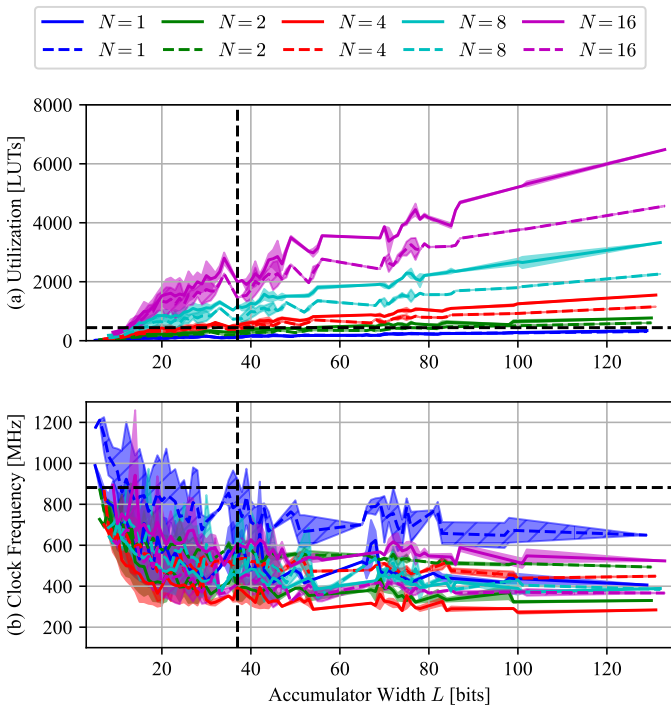


Fig. 10. Effects of the external-gate compressor trees (dashed lines) on (a) LUT utilization and (b) maximum clock frequency of the minifloat MACCs.

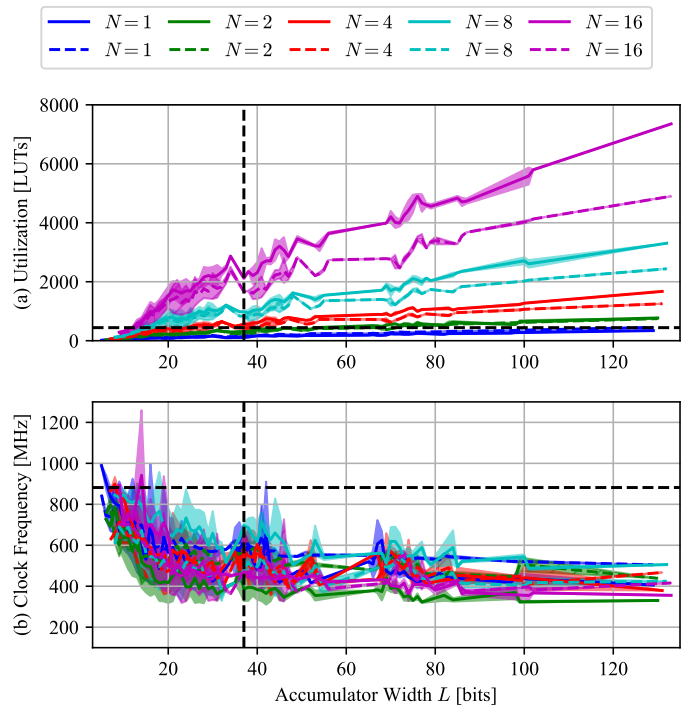


Fig. 11. Effects of the absorbed-gate compressor trees (dashed lines) on (a) LUT utilization and (b) maximum clock frequency of the minifloat MACCs.

the LUT utilization and maximum clock frequency results in comparison with the baseline MACC. The benefits of the compressor trees exceed our expectations by reducing LUT utilization *and* increasing clock frequency.

As expected, given the bit matrix shapes shown in Fig. 6a, single-lane designs show no LUT utilization benefits. All other configurations report savings of up to 27.1% and a mean of 17.7%, as reported in Table V. These numbers are much higher than our estimated 3% savings and close to the maximum mean savings of $0.5 \cdot 45\% = 22.5\%$ achievable with the compressor generator [42]. Table VI reports that single-lane designs experience the greatest mean increase in clock frequency of 48.5%, with all configurations showing a 16.2% improvement. The 16-lane designs stand out with an unexpected 28.3% decrease in clock frequency while simultaneously achieving a 15.3% reduction in LUT utilization. This observation is also evident in Fig. 10c.

D. Compressor Trees with Absorbed Gates

The positive results above raise our expectations regarding the absorption of 2-input gates into the compressor trees. We instruct the compressor generator with gate signatures generated in Tc1. The gate signatures are sequences of hexadecimal digits that capture the truth table of the two-input logic function of a bit matrix position, e.g., 6 for XOR and 8 for AND [42]. We use OR gates on the sign bits, as shown in Fig. 6a, and reduce the HDL description of the MACC to produce shifted mantissa products and replicate their sign bits for product inversion in the first stage of the compressor tree. The LUT utilization and maximum clock frequency results, compared with the baseline MACC, are shown in Fig. 11.

The LUT utilization curves shown in Fig. 11b initially resemble those shown in Fig. 10b. However, a closer inspection reveals that the LUT utilization of the baseline designs increased following the introduction of additional pipeline registers. Similarly, when comparing the baseline design curves in Figs. 10c and 11c, we see that the registers negatively affect the maximum clock frequency. These effects are unexpected since retiming is enabled in our experiments. However, the deeper baseline designs often consume more registers than LUTs, which may force some LUTs to serve as pass-through and put unnecessary pressure on general-purpose routing.

Since the baseline and absorbed-gate MACCs have equally deep pipelines, we assume that they suffer from the same effects. This enables their fair comparison. Unlike the external-gate designs, single-lane MACCs with absorbed gates show a 21.2% increase in LUT utilization, with all configurations reaching unexpectedly lower mean savings of 10.2%, as shown in Table V. This number matches our initial estimate, but falls short of the external-gate designs. The clock frequency results in Table VI are less conclusive, showing negative effects in single- and dual-lane designs, but positive effects in all others.

E. Effects on Integer Designs

We now focus on integer MACCs and explore the effects of using compressor trees with external and absorbed gates. As in the experiments above, we instruct the compressor generator with gate signatures generated in Tc1, including OR gates on the sign-extension constant bits, as shown in Fig. 6b. We reduce the HDL description of the MACC to produce partial product bits or simply pass the necessary gating bits directly to the compressor tree. The LUT utilization and maximum

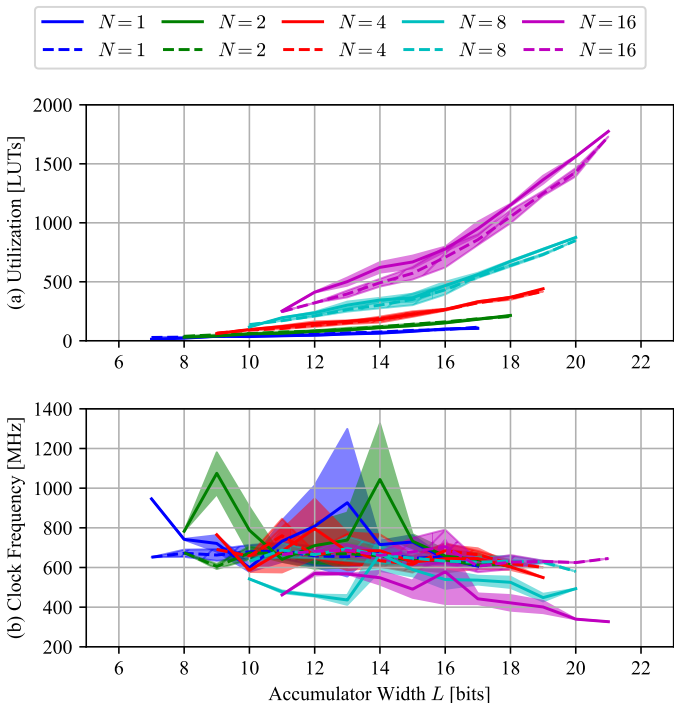


Fig. 12. Effects of the external-gate compressor trees (dashed lines) on (a) LUT utilization and (b) maximum clock frequency of the integer MACCs.

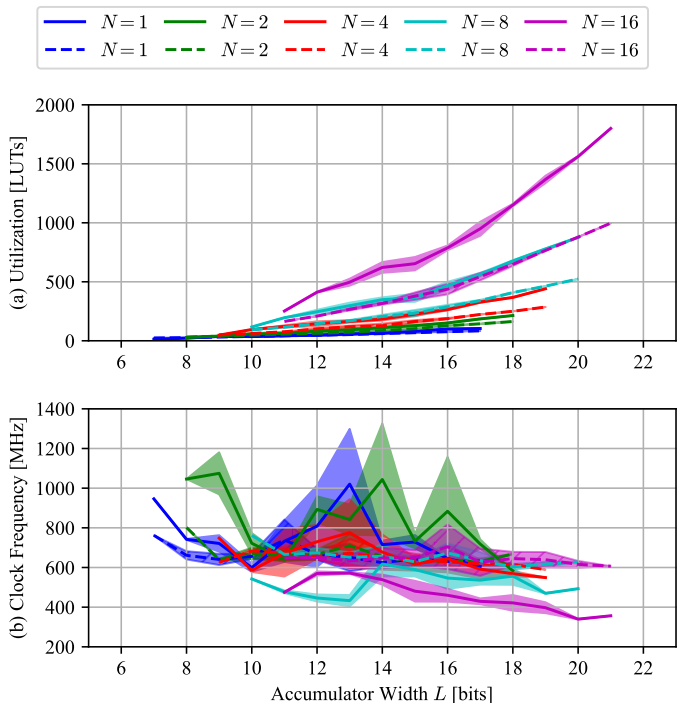


Fig. 13. Effects of the absorbed-gate compressor trees (dashed lines) on (a) LUT utilization and (b) maximum clock frequency of the integer MACCs.

TABLE VII

GEOMETRIC MEAN CHANGES IN LUT UTILIZATION OF THE INTEGER MACCS WITH THE PROPOSED ALTERATIONS VS. THE BASELINE.

Configuration	Number of input lanes N					
	All	1	2	4	8	16
External gates	0.993	1.167	1.071	0.957	0.925	0.874
Absorbed gates	0.719	0.967	0.825	0.711	0.612	0.556

TABLE VIII

GEOMETRIC MEAN CHANGES IN MAXIMUM CLOCK FREQUENCY OF THE INTEGER MACCS WITH THE PROPOSED ALTERATIONS VS. THE BASELINE.

Configuration	Number of input lanes N					
	All	1	2	4	8	16
External gates	1.064	0.908	0.880	1.004	1.213	1.400
Absorbed gates	1.036	0.879	0.818	0.990	1.215	1.382

clock frequency results are illustrated in Figs. 12 and 13 and detailed in Tables VII and VIII.

In terms of LUT utilization, we see that the compressor trees with external gates are beneficial only to designs with four or more lanes, and that, even then, the mean savings are limited to at most 13%. Absorbing the partial product gates into the compressor is beneficial to all MACC configurations and, in particular, 16-lane designs, whose utilization approaches the baseline 8-lane designs. For the maximum clock frequency, the results are less clear because of noise in certain configurations. Nevertheless, Figs. 12c and 13c and Table VIII suggest that MACCs with custom compressor trees achieve more stable and higher frequencies than the baseline designs, particularly in configurations with multiple lanes and long accumulators.

Clearly, the integer MACCs benefit much more from the proposed alterations than the minifloat designs. This is likely due to two reasons: First, as covered in Section IV, the product generator part of the minifloat MACC takes up a significant part of its resources and is unaffected by the compressor tree optimizations. Second, the most efficient counters used by the compressor generator [42] target tall, but narrow bit matrices. Concatenating the partial products in the integer

MACCs gives rise to relatively more of such columns than do the shifted products in the minifloat MACCs, as shown in Fig. 6. Similarly, the most efficient *quaternary* adder used by the compressor generator for final additions requires four rows of addend bits. The concatenated partial products in the integer MACCs can arrive at bit matrices that satisfy this requirement even for single-lane designs, whereas at least four input lanes are needed in the minifloat MACCs.

F. Minifloats vs. Integers

We now return to our outset and explore the impact of transitioning from integers to minifloats. Granted that floating-point arithmetic by nature is more complicated than its integer counterpart, we inevitably expect some overheads. However, their exact sizes are non-trivial. We provide one perspective, highlighting the costs of increasing the dynamic range – the number of exponent bits – with a subset of the most resource-efficient MACCs: The integer MACC using compressor trees with absorbed gates for $E = 0$ and the minifloat MACC using compressor trees with external gates for $E > 0$.

We illustrate this comparison with the mean per-lane LUT utilization in the single-lane and 16-lane designs with symmet-

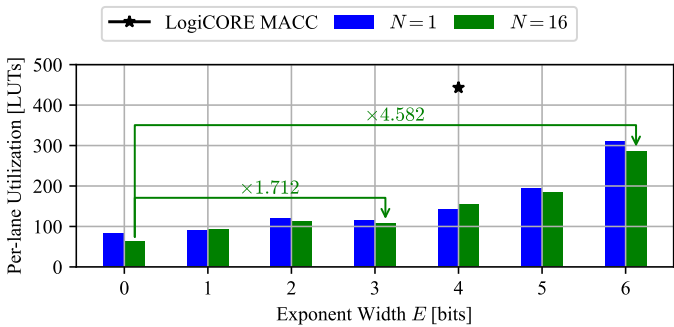


Fig. 14. Mean per-lane LUT utilization of the most resource-efficient MACCs with symmetric 8-bit operand formats vs. exponent width E .

TABLE IX
ABSOLUTE AND RELATIVE GEOMETRIC MEAN PER-LANE LUT UTILIZATION OF THE MOST RESOURCE-EFFICIENT MACCs WITH SYMMETRIC OPERAND FORMATS VS. EXPONENT WIDTH E .

	Width	Exponent width E							
	[bits]	0	1	2	3	4	5	6	
Absolute	3	14.17	13.66	—	—	—	—	—	
	4	21.61	20.16	25.64	—	—	—	—	
	5	31.84	33.50	50.17	50.05	—	—	—	
	6	42.83	50.49	75.21	68.44	93.62	—	—	
	7	57.26	69.60	86.25	92.37	107.6	159.0	—	
	8	72.46	91.41	113.6	110.2	149.1	185.6	293.9	
	Relative	3	1.000	0.963	—	—	—	—	—
		4	1.000	0.933	1.186	—	—	—	—
5		1.000	1.052	1.576	1.572	—	—	—	
6		1.000	1.179	1.756	1.598	2.186	—	—	
7		1.000	1.215	1.506	1.613	1.879	2.776	—	
8		1.000	1.261	1.568	1.520	2.058	2.562	4.056	

ric 8-bit operand formats in Fig. 14. The comparison does not consider the maximum clock frequency, as our experiments suggest that all included designs can operate well beyond 300 MHz. As foreseen, the minifloat MACCs are more expensive than the integer ones in all cases. Two cases stand out: First, the MACCs with $E = 3$ achieve a lower mean per-lane LUT utilization than those with $E = 2$, possibly because these formats suit the 6-input LUTs in the Versal fabric particularly well, as identified in [11]. Second, the MACCs with $E = 6$ represent an extreme, demonstrating more than four times as high mean LUT utilization as the integer designs.

In geometric mean across all exponent widths, the single-lane minifloat MACCs are 1.770 times as large as the integer ones and the 16-lane ones are 2.317 times as large. Part of the difference between these factors can be attributed to the relatively greater impact of parallelism on per-lane LUT utilization in the integer MACCs than in the minifloat ones. Table IX provides additional data for this comparison across all considered configurations of input lanes and bit-widths. These data reveal nearly no mean overheads at $E = 1$, up to more than 300% at $E = 6$. The most useful formats for inference lie between these extremes [2], with associated mean overheads ranging from roughly 20% to 180%.

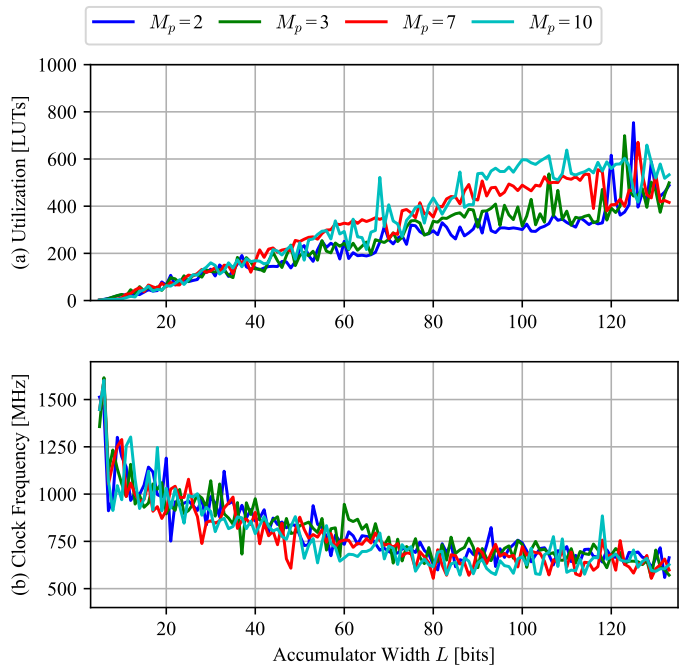


Fig. 15. Converter (a) LUT utilization and (b) maximum clock frequency for different target precisions M_p .

G. Floating-Point Conversion Costs

We now explore the additional overhead incurred when integrating the proposed Kulisch-to-minifloat converter with our MACCs. We perform implementation for all accumulator widths encountered for our minifloat MACC, $L \in [5, 133]$, but choose only four mantissa widths from Table I, $M_p \in \{2, 3, 7, 10\}$, to limit the parameter space. We deliberately include 7- and 10-bit widths despite them not fitting our definition of minifloats as some existing work argues for storing activations in higher-precision formats [10], [12]. The converter is not pipelined, as it can comfortably achieve high operating frequencies without additional registers.

Fig. 15 presents the LUT utilization and maximum clock frequency of the converter in isolation. We see that LUT utilization grows nearly linearly with L , and that designs with higher M_p tend to be costlier than others. This is explained by the fast relative growth of the shifter for increasing input and output bit-widths, which outpaces the other design elements, including the leading-one detector. The apparent jitter in the LUT utilization cannot be explained by reasons similar to those for the pipelined baseline minifloat MACCs, but may be a result of heuristic optimization. The maximum clock frequency is consistently higher than 500 MHz and in line with our most resource-efficient external-gate MACCs, shown in Fig. 10c. We deduce that the converter is unlikely to limit the performance when combined with the proposed MACCs.

Table X lists the mean LUT utilization of the converter with $M_p = 3$ relative to the mean per-lane LUT utilization in our most resource-efficient MACCs with external gates. These and the subsequent results are simply summed post-implementation and hence do not account for any potential symbiotic effects of synthesizing MACCs and converters.

TABLE X

GEOMETRIC MEAN LUT UTILIZATION OVERHEAD OF THE CONVERTER PER INPUT LANE IN THE MOST RESOURCE-EFFICIENT MINIFLOAT MACCS WITH EXTERNAL GATES.

Precision M_p	Number of input lanes N				
	1	2	4	8	16
2	0.691	0.797	0.890	0.959	1.010
3	0.784	0.914	1.010	1.078	1.117
7	0.764	0.922	1.058	1.154	1.208
10	0.629	0.772	0.899	1.007	1.078

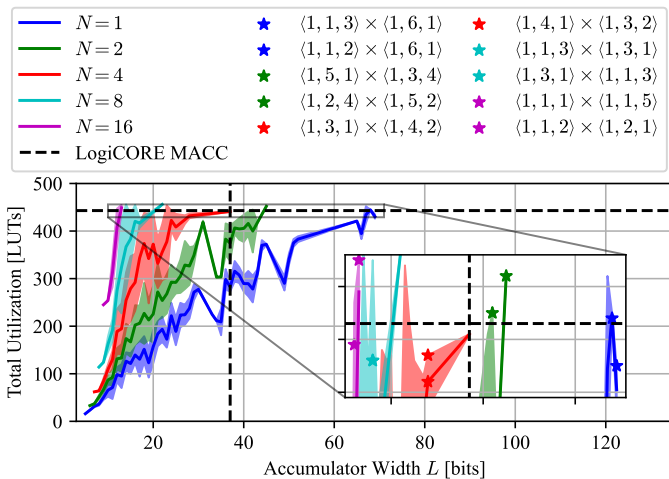


Fig. 16. Comparison of the total LUT utilization of the minifloat MACC with external gates and our converter with $M_p=3$ to the LogiCORE MACC.

We observe that the cost of conversion is comparable to a MACC lane, with the per-lane cost increasing as each lane becomes relatively less costly in parallel designs. Thus, the LUT utilization of the converter is significant, but it can be amortized effectively through parallelization.

Fig. 16 compares the total LUT utilization of our most resource-efficient external-gate MACCs and our converter with the LogiCORE MACC budget listed in Table IV. For each number of input lanes N , we highlight the two configurations with the longest accumulators within $\pm 3\%$ of this budget. These designs suggest that higher degrees of parallelism are achievable by trading off the operand precision. Unlike the LogiCORE MACC, our combined design does not necessarily produce results with 4-bit exponents. Fixing its number of exponent bits would nullify the benefits of Kulisch accumulation in configurations where the accumulator can represent numbers, whose dynamic range is not captured by this exponent.

We repeat our comparison of minifloats and integers, including one converter with $M_p = 3$ in all minifloat designs. The results listed in Table XI show a clear impact of the resources spent on the converter: Compared with Table IX, we notice that the minifloat designs now have overheads in all configurations, and that the overheads for the most useful formats in neural network inference [2] have risen to roughly 60% to 300%. We also see that the converter has a large impact on designs with particularly long accumulators, i.e., $E = 5$ and $E = 6$.

TABLE XI

ABSOLUTE AND RELATIVE GEOMETRIC MEAN PER-LANE LUT UTILIZATION OF THE MOST RESOURCE-EFFICIENT MACCS WITH SYMMETRIC OPERAND FORMATS VS. EXPONENT WIDTH E , INCLUDING THE CONVERTER WITH $M_p = 3$.

	Width [bits]	Exponent width E						
		0	1	2	3	4	5	6
Absolute	3	14.17	15.49	—	—	—	—	—
	4	21.61	25.00	34.38	—	—	—	—
	5	31.84	42.17	61.85	68.81	—	—	—
	6	42.83	61.77	90.00	90.27	135.5	—	—
	7	57.26	84.61	103.8	124.0	154.3	236.5	—
8	72.46	108.7	132.6	140.9	205.2	272.0	447.0	
Relative	3	1.000	1.093	—	—	—	—	—
	4	1.000	1.157	1.591	—	—	—	—
	5	1.000	1.324	1.942	2.161	—	—	—
	6	1.000	1.442	2.102	2.108	3.165	—	—
	7	1.000	1.477	1.812	2.165	2.695	4.129	—
8	1.000	1.500	1.829	1.944	2.831	3.754	6.169	

H. Versal vs. UltraScale+

As outlined in Section II, the Versal FPGA fabric is different from its 7 Series and UltraScale+™ predecessors. Prior work has shown that the footprint of efficient compressor trees and accumulators remain largely unchanged between these architectures, but that the technological advancements made in Versal reduce latency by 30% [42]. To better understand how these effects translate from integer to minifloat MACCs, we synthesize and implement single- and 16-lane configurations of our baseline and most resource-efficient designs with external gates targeting the UltraScale+ xcvu9p-flga2104-2L-e part that is found on the VCU118 Evaluation Kit.

We find that the UltraScale+ designs consistently utilize fewer LUTs than their Versal counterparts, with geometric mean savings of 8.80% and 18.6% for the baseline and external-gate designs. The frequency results are inconclusive. On average, the single-lane baseline designs are 29.8% faster on UltraScale+, whereas the 16-lane designs are 26.8% slower. The external-gate designs show the opposite pattern, with the single-lane configurations being 13.2% slower, and the 16-lane ones being 16.5% faster.

VII. CONCLUSION

In this paper, we carried out a design space exploration into parallel accurate minifloat MACCs with Kulisch accumulation targeting the AMD Versal™ FPGA fabric. We presented three alterations to a baseline MACC design: A segmented shifter and custom compressor trees with external or absorbed sign-inversion gates. For comparison, we applied similar alterations to a parallel integer MACC. We considered all possible configurations of these designs for 3- to 8-bit-wide operand formats.

We find that the minifloat MACCs do not benefit from the segmented shifter; that the compressor trees with external gates reduce their mean LUT utilization by 17.7% and increase their clock frequency by 16.2%; and that the compressor trees with absorbed gates only reduce mean LUT utilization by 10.2% with no impact on clock frequency. For the integer MACCs, the compressor trees with external gates do not reduce LUT

utilization but increase clock frequency by 6.40%, while the compressor trees with absorbed gates reduce mean LUT utilization by 28.1% and increase clock frequency by 3.60%. When comparing the per-lane LUT utilization of the most resource-efficient designs, we find that minifloat MACCs for formats useful in inference consume 20% to 180% more LUTs than integer ones with same-width operands, with overheads ranging from nil to over 300% for exponent bit-widths of 1 to 6. Including the Kulisch to minifloat converter increases the overheads for the popular formats to between 60% and 300% with up to 510% additional costs for 6-bit exponents.

Moving forward, we will explore the effects of implementing custom, pipelined shifters in place of the non-segmented default implementation from Vivado™. Despite our outset, we also plan to probe whether MACCs for certain, particularly popular minifloat formats can be mapped to DSP58 primitives while maintaining resource efficiency in comparison to in-fabric implementations. Finally, we plan to explore the cost overhead of our minifloat MACCs when integrated with diverse ML accelerator architectures.

REFERENCES

- [1] J. Johnson, "Rethinking Floating Point for Deep Learning," *arXiv preprint arXiv:1811.01721*, 2018.
- [2] S. Aggarwal, H. J. Damsgaard, A. Pappalardo, G. Franco, T. B. Preußer, M. Blott, and T. Mitra, "Shedding the Bits: Pushing the Boundaries of Quantization with Minifloats on FPGAs," in *Proceedings of 34th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2024.
- [3] F. De Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.
- [4] C. Bobda, J. M. Mbongue, P. Chow, M. Ewais, N. Tarafdar, J. C. Vega, K. Eguro, D. Koch, S. Handagala, M. Leeser *et al.*, "The Future of FPGA Acceleration in Datacenters and the Cloud," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 3, pp. 1–42, 2022.
- [5] *IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society, 2019.
- [6] U. Kulisch, *Computer Arithmetic and Validity: Theory, Implementation, and Applications*. Walter de Gruyter, 2008.
- [7] A. Kuzmin, M. Van Baalen, Y. Ren, M. Nagel, J. Peters, and T. Blankevoort, "FP8 Quantization: The Power of the Exponent," *Advances in Neural Information Processing Systems*, vol. 35, pp. 14 651–14 662, 2022.
- [8] M. D. Ercegovic and T. Lang, *Digital Arithmetic*. Elsevier, 2004.
- [9] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan, "Ultra-Low Precision 4-bit Training of Deep Neural Networks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1796–1807, 2020.
- [10] Y. Nevarez, A. Beering, A. Najafi, A. Najafi, W. Yu, Y. Chen, K.-L. Krieger, and A. Garcia-Ortiz, "CNN Sensor Analytics with Hybrid-Float6 Quantization on Low-Power Embedded FPGAs," *IEEE Access*, vol. 11, pp. 4852–4868, 2023.
- [11] P. Metzgen, S. Fang, S. Pareek, B. Tian, Y. Shan, E. Delaye, and A. Sirasao, "Higher Performance Neural Networks with Small Floating Point," AMD, Tech. Rep. WP530, 6 2021.
- [12] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks," *Advances in Neural Information Processing Systems*, vol. 32, 2019.
- [13] N. Mellempudi, S. Srinivasan, D. Das, and B. Kaul, "Mixed Precision Training with 8-bit Floating Point," *arXiv preprint arXiv:1905.12334*, 2019.
- [14] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training Deep Neural Networks with 8-bit Floating Point Numbers," *Advances in Neural Information Processing Systems*, vol. 31, 2018.
- [15] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "A Trans-precision Floating-Point Platform for Ultra-Low Power Computing," in *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2018, pp. 1051–1056.
- [16] S. Wang and P. Kanwar, "BFloat16: The secret to High Performance on Cloud TPUs," <https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus>, Google Cloud, 2019, accessed: 2024-02-07.
- [17] P. Kharya, "TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x," <https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/>, NVIDIA, 2020, accessed: 2024-02-07.
- [18] S. Sun and J. Zambreno, "A Floating-Point Accumulator for FPGA-based High Performance Computing Applications," in *International Conference on Field-Programmable Technology*. IEEE, 2009, pp. 493–499.
- [19] E. Kadric, P. Gurniak, and A. DeHon, "Accurate Parallel Floating-Point Accumulation," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3224–3238, 2016.
- [20] I. Colbert, A. Pappalardo, J. Petri-Koenig, and Y. Umuroglu, "A2Q+: Improving Accumulator-Aware Weight Quantization," *arXiv preprint arXiv:2401.10432*, 2024.
- [21] *UltraScale Architecture and Product Data Sheet: Overview*, AMD, 7 2023.
- [22] *Versal Architecture and Product Data Sheet: Overview*, AMD, 9 2023.
- [23] J. Sommer, M. A. Özkan, O. Keszocze, and J. Teich, "DSP-Packing: Squeezing Low-Precision Arithmetic into FPGA DSP Blocks," in *32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2022, pp. 160–166.
- [24] M. Nagel, M. v. Baalen, T. Blankevoort, and M. Welling, "Data-Free Quantization through Weight Equalization and Bias Correction," in *IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1325–1334.
- [25] E. Frantar, S. Ashkboos, T. Hoefer, and D. Alistarh, "GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers," *arXiv preprint arXiv:2210.17323*, 2022.
- [26] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A Survey of Quantization Methods for Efficient Neural Network Inference," *arXiv preprint arXiv:2103.13630*, 2021.
- [27] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort, "Up or Down? Adaptive Rounding for Post-Training Quantization," in *International Conference on Machine Learning*. PMLR, 2020, pp. 7197–7206.
- [28] S. O. Settle, M. Bollavaram, P. D'Alberto, E. Delaye, O. Fernandez, N. Fraser, A. Ng, A. Sirasao, and M. Wu, "Quantizing Convolutional Neural Networks for Low-Power High-Throughput Inference Engines," *arXiv preprint arXiv:1805.07941*, 2018.
- [29] Z. Luo and M. Martonosi, "Accelerating Pipelined Integer and Floating-Point Accumulations in Configurable Hardware with Delayed Addition Techniques," *IEEE Transactions on Computers*, vol. 49, no. 3, pp. 208–218, 2000.
- [30] Y. Uguen and F. de Dinechin, "Design-Space Exploration for the Kulisch Accumulator," CITI Laboratory, National Institute of Applied Sciences and Inria Lyon, Tech. Rep. hal-01488916, 3 2017.
- [31] A. Paidimarrri, A. Cevrero, P. Brisk, and P. Jenne, "FPGA Implementation of a Single-Precision Floating-Point Multiply-Accumulator with Single-Cycle Accumulation," in *17th IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE, 2009, pp. 267–270.
- [32] M. Lin, S. Cheng, and J. Wawrzynek, "Cascading Deep Pipelines to Achieve High Throughput in Numerical Reduction Operations," in *International Conference on Reconfigurable Computing and FPGAs*. IEEE, 2010, pp. 103–108.
- [33] D. Wilson and G. Stitt, "The Unified Accumulator Architecture: A Configurable, Portable, and Extensible Floating-Point Accumulator," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 9, no. 3, pp. 1–23, 2016.
- [34] S. Siegel and J. Wolff von Gudenberg, "A Long Accumulator like a Carry-Save Adder," *Computing*, vol. 94, pp. 203–213, 2012.
- [35] T. O. Bachir and J.-P. David, "Performing Floating-Point Accumulation on a Modern FPGA in Single and Double Precision," in *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2010, pp. 105–108.
- [36] A. Roldao Lopes and G. A. Constantinides, "A Fused Hybrid Floating-Point and Fixed-Point Dot-Product for FPGAs," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2010, pp. 157–168.

- [37] Y. Lei, Y. Dou, Y. Dong, J. Zhou, and F. Xia, "FPGA Implementation of an Exact Dot Product and Its Application in Variable-Precision Floating-Point Arithmetic," *The Journal of Supercomputing*, vol. 64, pp. 580–605, 2013.
- [38] M. Véstias, R. P. Duarte, J. T. de Sousa, and H. Neto, "Parallel Dot-Products for Deep Learning on FPGA," in *27th International Conference on Field Programmable Logic and Applications*. IEEE, 2017, pp. 1–4.
- [39] B. Pasca, "Hybrid Dot-Product Design for FP-enabled FPGAs," in *IEEE 26th Symposium on Computer Arithmetic*. IEEE, 2019, pp. 194–196.
- [40] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. Neto, "Hybrid Dot-Product Calculation for Convolutional Neural Networks in FPGA," in *29th International Conference on Field Programmable Logic and Applications*. IEEE, 2019, pp. 350–353.
- [41] M. Véstias, R. P. Duarte, J. T. de Sousa, and H. Neto, "Efficient Design of Low Bitwidth Convolutional Neural Networks on FPGA with Optimized Dot Product Units," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 1, pp. 1–36, 2022.
- [42] K. J. Hofffeld, H. J. Damsgaard, J. Nurmi, M. Blott, and T. B. Preußer, "High-Efficiency Compressor Trees for Latest AMD FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETs)*, vol. 17, no. 2, pp. 1–32, 2024.
- [43] *Versal ACAP Configurable Logic Block*, AMD, 2 2023.
- [44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.
- [45] M. Sjalander and P. Larsson-Edefors, "High-Speed and Low-Power Multipliers using the Baugh-Wooley Algorithm and HPM Reduction Tree," in *15th IEEE International Conference on Electronics, Circuits and Systems*. IEEE, 2008, pp. 33–36.
- [46] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'Brien, Y. Umuroglu, M. Leeser, and K. Vissers, "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 3, pp. 1–23, 2018.
- [47] Chair of VLSI Design, Diagnostics and Architecture. (2016) PoC - Pile of Cores. Technische Universität Dresden. [Online]. Available: <https://github.com/VLSI-EDA/PoC>
- [48] *Vivado Design Suite Tcl Command Reference Guide*, AMD, 5 2023.
- [49] *Floating-Point Operator v7.1: LogiCORE IP Product Guide*, AMD, 12 2020.
- [50] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An fpga-specific approach to floating-point accumulation and sum-of-products," in *International Conference on Field-Programmable Technology*, 2008.

APPENDIX A

OVERFLOW IN KULISCH TO MINIFLOAT CONVERSION

The following proves that the exponent increments in Algo. 1 lines 13 and 15 cannot overflow. Consider the edge case when $E_p = \lceil \log_2[L - M_p + 1] \rceil = \log_2[L - M_p + 1]$, which means:

$$2^{\log_2[L - M_p + 1]} = L - M_p + 1 > e_p \geq 0. \quad (3)$$

For m_{sum} to overflow in **S2**, the accumulator must be negative, $s_p = 1$, and $\overline{m_{\text{int}}}$ must be all ones. Furthermore, for e_{int} to be maximal, the input to the leading-one detector must have a zero following its sign bit. The maximum index of a leading one in a vector of n bits is $n - 1$. However, given that $s_p = 1$ implies that the most significant bit passed to the leading-one-detector is zero, the outputs are further constrained to:

$$L - M_p - 2 \geq \text{lop} \geq 0 \quad \text{and} \quad \overline{zer\overline{o}} = 1. \quad (4)$$

Finally, assuming the maximum index in (4) and incrementing it twice according to Algo. 1 gives:

$$e_p = \text{lop} + \overline{zer\overline{o}} + 1 = L - M_p - 2 + 2 = L - M_p, \quad (5)$$

which satisfies the inequality in (3). ■

BIOGRAPHIES



Hans Jakob Damsgaard received the B.Sc. degree in electrical engineering in 2019 and the M.Sc. degree in computer science and engineering in 2021 as part of the Honours programme at the Technical University of Denmark, DTU. He is currently pursuing a PhD in reconfigurable approximate accelerators for edge computing at Tampere University, TAU. He received the best paper award at NorCAS'21. His research interests include hardware accelerators, networks-on-chip, computer architecture, and hardware verification.



Konstantin J. Hofffeld received the B.Sc. degree in information system technology in 2022 at the Technical University of Darmstadt, Germany. Afterwards, he spent eleven months at AMD Research, Dresden, investigating efficient arithmetic for FPGAs. His research interests include computer architecture, reconfigurable hardware, and compilers.



Jari Nurmi received the D.Sc. degree in technology in 1994. He works as a Professor at the SoC Hub Research Centre and Wireless Research Centre, Electrical Engineering Unit, Tampere University, TAU (formerly Tampere University of Technology, TUT), Finland, since 1999. He is working on embedded computing systems, System-on-Chip, approximate computing, wireless localization, positioning receiver prototyping, and software-defined radio and software-defined networks. He held various research, education, and management positions at TUT since 1987 (e.g., Acting Associate Professor from 1991 to 1994), and was the Vice President of the SME VLSI Solution Oy from 1995 to 1998. He has supervised 35 Ph.D. and over 150 M.Sc. theses, and been an opponent or a reviewer of over 50 Ph.D. theses for other universities worldwide. He is a member of the Technical Committee on VLSI Systems and Applications at IEEE CASS.



Thomas B. Preußer is a principal engineer at AMD Research and Advanced Development. Thomas earned a PhD from TU Dresden in 2011. He worked as a postdoctoral researcher both there and at ETH Zürich. He also enjoyed an EU-funded Individual Maria-Sklodowska-Curie Fellowship at the Xilinx Research Labs in Dublin in 2017/18. His scientific background is computer arithmetic, application acceleration and systems design using FPGAs. His current focus lies on advancing FINN, a tool for the generation of custom embedded dataflow solutions for neural-network inference on FPGA fabric.