Online Alignment and Addition in Multi-Term Floating-Point Adders

Kosmas Alexandridis and Giorgos Dimitrakopoulos

***Abstract*—Multi-term floating-point addition appears in vector dot-product computations, matrix multiplications, and other forms of floating-point data aggregation. A critical step in multi-term floating point addition is the alignment of fractions of the floating-point terms before adding them. Alignment is executed serially by identifying first the maximum of all exponents and then shifting the fraction of each term according to the difference of its exponent from the maximum one. Contrary to common practice, this work proposes a new *online* algorithm that splits the identification of the maximum exponent, the alignment shift for each fraction, and their addition to multiple fused incremental steps that can be computed in parallel. Each fused step is implemented by a new associative operator that allows the incremental alignment and addition for arbitrary number of operands. Experimental results show that employing the proposed align-and-add operators for the implementation of multi-term floating point adders can improve delay or save significant area and power. The achieved area and power savings range between 3%–23% and 4%–26%, respectively.**

***Index Terms*—Floating point arithmetic, Multi-term adders, Online algorithm, Energy Efficiency**

I. Introduction

Machine learning (ML) algorithms have been widespread in various application domains. Their efficient and accurate computation relies mostly on matrix multiplication kernels and floating-point (FP) arithmetic for data representation [1], [2].

The FP representations used in ML algorithms cover IEEE-754 compliant formats as well as reduced-precision formats that use 16 or fewer bits in total, in an effort to balance numerical performance, and hardware and storage costs [3], [4]. In most cases, a FP number consists of three fields: the sign bit (s), the exponent (e), and the fraction (m) and its value is given by $(-1)^s \times 1.m \times 2^{e-\text{bias}}$, where bias is a constant that depends on the bit width of the exponent. Corner cases, such as not-a-number, infinity, or de-normals can be also encoded or skipped depending on the chosen format [4].

To reduce the overhead of FP arithmetic when implementing vector-wide operations, designers have turned to fusing individual FP operations to more complex ones that implement the needed computation at once [5], [6], [7], [8]. This approach allows alignment, normalization, and rounding steps to be shared among independent operations, ultimately resulting in more efficient hardware architectures.

This work was supported by a Siemens EDA research grant to Democritus University of Thrace on "High-level synthesis research for Systems-on-Chip".

Kosmas Alexandridis and Giorgos Dimitrakopoulos are with the Department of Electrical and Computer Engineering, Democritus University of Thrace, Xanthi, Greece. E-mail: {koalexan, dimitrak}@ee.duth.gr

Multi-term addition, the core of fused operators, involves adding multiple FP numbers with potentially different exponents. To align the addends for addition, the fraction of each number is shifted according to the difference of its own exponent to the maximum exponent of all terms. This serial dependency across fraction alignment and addition impacts negatively the overall hardware efficiency.

In this work, inspired by online softmax computation [9], we propose a new *online* approach for alignment and addition in multi-term FP adders. In this way, *all serial dependencies* that traditionally characterize alignment and addition steps *are removed* and maximum exponent calculation, as well as alignment and addition of fractions, are computed *incrementally and in parallel*. In practice, alignment and addition is performed using trees built from the newly proposed align-and-add operators.

The experimental evaluation shows that the proposed approach simplifies fundamentally the complexity of alignment and addition in multi-term FP adders. The corresponding hardware units that adopt the online alignment and addition paradigm, require significantly less area and power than traditional approaches. The area and power savings range between 3%–23% and 4%–26%, respectively, for various examined configurations. Also, when opting for high-speed implementations, they can also improve delay under the same number of pipeline stages.

II. Alignment and Addition in Multi-Term Floating Point Adders

A high-level description of multi-term fused addition is shown in Algorithm 1. The input is an array of FP numbers f_i and the output is their sum S. The algorithm begins by finding the exponent with the maximum value in step 1. Then, the fractions are aligned based on the difference of the local exponent and the maximum one (step 2). With the fractions aligned, the summation operation is performed in step 3. The sum is normalized and rounded in step 4.

Algorithm 1 Multi-term fused floating point addition

Input: Floats $f_1, f_2, \ldots, f_N$
Output: $S = \sum_{i=1}^{N} f_i$

1: Find maximum exponent $e_{\max} = \max(e_1, e_2, \ldots, e_N)$
2: Align every fraction $1.m_i$ by shifting right by $e_{\max} - e_i$ positions
3: Sum the aligned fractions $S = \sum_{i=1}^{N} \text{aligned}(1.m_i)$
4: Normalize and round S to produce the final FP sum

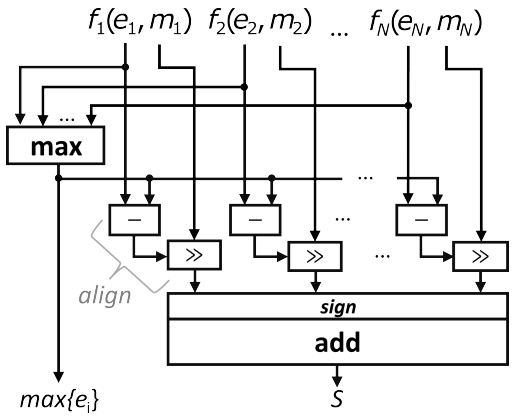

Fig. 1. Baseline approach for multi-term fraction alignment and addition.

The baseline implementation of alignment and addition (steps 1–3), which are the focus of this work, is detailed in Algorithm 2. The first loop corresponds to the first step in Algorithm 1 that computes the maximum exponent and stores it in λ_N at the end of the loop. The second loop performs steps 2 and 3 of Algorithm 1; each fraction m_i is aligned in line 5 and accumulated to a partial sum o_i in line 6. To simplify presentation in Algorithm 2, each fraction $1.m_i$ is denoted as m_i, which is assumed to be in signed (2's complement) form according to the sign s_i of f_i.

Algorithm 2 Serial fraction alignment and addition

1: **for** $i \leftarrow 1 : N$ **do**
2: $\quad \lambda_i \leftarrow \max(\lambda_{i-1}, e_i)$
3: **end for** ▷ Maximum exponent in λ_N
4: **for** $i \leftarrow 1 : N$ **do**
5: $\quad am_i \leftarrow m_i \gg (\lambda_N - e_i)$ ▷ Alignment shift
6: $\quad o_i \leftarrow o_{i-1} + am_i$ ▷ Accumulate the aligned fraction
7: **end for**
8: $S = o_N$

The two loops of Algorithm 2 cannot be merged. Thus, in hardware, each part is unrolled separately and the second loop can begin execution only after the first loop has computed the maximum exponent λ_N. This approach for alignment and addition is followed in the majority of hardware architectures for multi-term adders [10], [11], [12] and is shown in Fig. 1.

To reduce delay, other architectures perform fraction alignment based on the relative difference of exponents, and avoid the dependency to the maximum exponent that is computed in parallel [5], [6], [7], [13]. However, in all cases, this concept is applied only for 3- or 4-term adders and cannot be generalized to arbitrary number of terms. This limitation is removed by the formulation in this work. Other solutions, such as Kaul *et al.* [14], split the alignment of fractions into global and local alignment. In this way, computing exponent differences and the alignment shift are partially overlapped in time at the circuit level. However, still addition is performed separately in a following step.

Other approaches avoid the need for fraction alignment by mapping floating point accumulation to fixed-point arithmetic [15], [16]. Effectively, alignment is performed implicitly when transforming FP numbers to their equivalent fixed-point integers. Such approaches are practical when accumulation is done in time. In this work, we focus on wide parallel architectures that perform addition in space.

III. Online alignment and addition

This work aims to fuse the serial alignment and addition steps into one combined step that would perform maximum exponent calculation, alignment shift and addition incrementally and in parallel for various groups of inputs. Effectively, this transformation would allow us to merge the two separate loops of Algorithm 2 into one single loop.

To present the proposed algorithm for online alignment and addition, we first merge the shift and add operations, shown in lines 5 and 6 of Algorithm 2, into one equation as follows:

$$o_i = o_{i-1} + m_i \gg (\lambda_N - e_i), \quad \text{with} \quad \lambda_N = \max_i\{e_i\} \tag{1}$$

The right shift in (1) can be equivalently written as a multiplication with a negative power of two, i.e.,

$$o_i = o_{i-1} + m_i\, 2^{-(\lambda_N - e_i)}. \tag{2}$$

Fully unrolling (2) we can write the final sum o_N as follows:

$$o_N = o_{N-1} + o_{N-2} + \ldots + o_1 = \sum_{i=1}^{N} m_i\, 2^{-(\lambda_N - e_i)} \tag{3}$$

A. Basic online algorithm for alignment and addition

To remove the dependency to λ_N for the computation of the final sum o_N we define a new sequence o'_i

$$o'_i = \sum_{j=1}^{i} m_j 2^{-(\lambda_i - e_j)} \quad \text{with} \;\; \lambda_i = \max(\lambda_{i-1}, e_i) \tag{4}$$

Sequence (4) has the interesting property that its last term o'_N is equal to o_N defined in (3). Beginning from (4), our goal is to form a recursive relation that would connect o'_i to o'_{i-1}. Initially, in (4), we separate the ith term $m_i\, 2^{-(\lambda_i - e_i)}$ from the rest:

$$o'_i = \left(\sum_{j=1}^{i-1} m_j\, 2^{-(\lambda_i - e_j)} \right) + m_i\, 2^{-(\lambda_i - e_i)}$$

Then, inside the parenthesis, we add and subtract the helper term λ_{i-1}

$$o'_i = \left(\sum_{j=1}^{i-1} m_j\, 2^{-(\lambda_i - \lambda_{i-1} + \lambda_{i-1} - e_j)} \right) + m_i\, 2^{-(\lambda_i - e_i)}$$

Finally, we factor out the term $2^{-(\lambda_i - \lambda_{i-1})}$

$$o'_i = \left(\sum_{j=1}^{i-1} m_j\, 2^{-(\lambda_{i-1} - e_j)} \right) 2^{-(\lambda_i - \lambda_{i-1})} + m_i\, 2^{-(\lambda_i - e_i)} \tag{5}$$

According to (4), the term left in the parenthesis corresponds to o'_{i-1}. Thus, introducing o'_{i-1} into (5) we get the sought recursive relation:

$$o'_i = o'_{i-1}\, 2^{-(\lambda_i - \lambda_{i-1})} + m_i\, 2^{-(\lambda_i - e_i)} \quad \text{with} \;\; \lambda_i = \max(\lambda_{i-1}, e_i) \tag{6}$$

Algorithm 3 Online fused fraction alignment and addition

1: **for** $i \leftarrow 1 : N$ **do**
2: $\quad \lambda_i \leftarrow \max(\lambda_{i-1}, e_i)$
3: $\quad o'_i \leftarrow o'_{i-1} \gg (\lambda_i - \lambda_{i-1}) + m_i \gg (\lambda_i - e_i)$
4: **end for**
5: $S = o'_N$

Remapping multiplications with negative powers of two back to equivalent right arithmetic shift operations, the recursive relation in (6) can be equivalently expressed as follows:

$$o'_i = o'_{i-1} \gg (\lambda_i - \lambda_{i-1}) + m_i \gg (\lambda_i - e_i) \tag{7}$$

This mapping to shift operations is valid since λ_i is the maximum of λ_{i-1} and e_i and thus the shift amounts $\lambda_i - \lambda_{i-1}$ and $\lambda_i - e_i$ in (7), are always greater or equal to zero.

Algorithm 3 uses recursive relation (7) to compute alignment and addition *online*. At each iteration, a local maximum exponent is identified that drives local alignment shifts and accumulation of the output sum. Even if this fused align and add operation needs an extra subtraction and shift per iteration relative to Algorithm 2, the experimental results show that it leads to more efficient unrolled and pipelined hardware implementations.

B. Parallel computation of fraction alignment and addition

The computation of the sum of aligned fractions S and the identification of the maximum exponent can be performed in parallel using a new operator $\circledcirc$ that is defined as:

$$\begin{bmatrix} \lambda_i \\ o_i \end{bmatrix} \circledcirc \begin{bmatrix} \lambda_j \\ o_j \end{bmatrix} = \begin{bmatrix} \max(\lambda_i, \lambda_j) \\ o_i \gg (\max(\lambda_i, \lambda_j) - \lambda_i) + o_j \gg (\max(\lambda_i, \lambda_j) - \lambda_j) \end{bmatrix} \tag{8}$$

It can be shown by induction using a derivation similar to (5) that the final sum S and the maximum exponent of a set of FP numbers can be computed using the newly defined operator $\circledcirc$ as follows:

$$\begin{bmatrix} \max\{e_i\} \\ S \end{bmatrix} = \begin{bmatrix} e_1 \\ m_1 \end{bmatrix} \circledcirc \begin{bmatrix} e_2 \\ m_2 \end{bmatrix} \circledcirc \ldots \circledcirc \begin{bmatrix} e_N \\ m_N \end{bmatrix} \tag{9}$$

Also, it can be proven that the operator $\circledcirc$ is associative since

$$\left(\begin{bmatrix} e_1 \\ m_1 \end{bmatrix} \circledcirc \begin{bmatrix} e_2 \\ m_2 \end{bmatrix} \right) \circledcirc \begin{bmatrix} e_3 \\ m_3 \end{bmatrix} = \begin{bmatrix} e_1 \\ m_1 \end{bmatrix} \circledcirc \left(\begin{bmatrix} e_2 \\ m_2 \end{bmatrix} \circledcirc \begin{bmatrix} e_3 \\ m_3 \end{bmatrix} \right) \tag{10}$$

C. Hardware Organization of Alignment and Addition

Using the new associative operator $\circledcirc$, fraction alignment and addition can be performed using various hardware configurations. For instance, Fig. 2(a) depicts a binary-tree architecture of $\circledcirc$ operators. Following the definition of the $\circledcirc$ operator in (8), at each node of the tree, the local maximum exponent is identified first and in turn drives local fraction alignment and addition.

The $\circledcirc$ operator can be generalized to higher radices as well. Fig. 2(b) shows an example of an 8-term alignment and addition using a mixture of radix-4 and radix-2 operators.

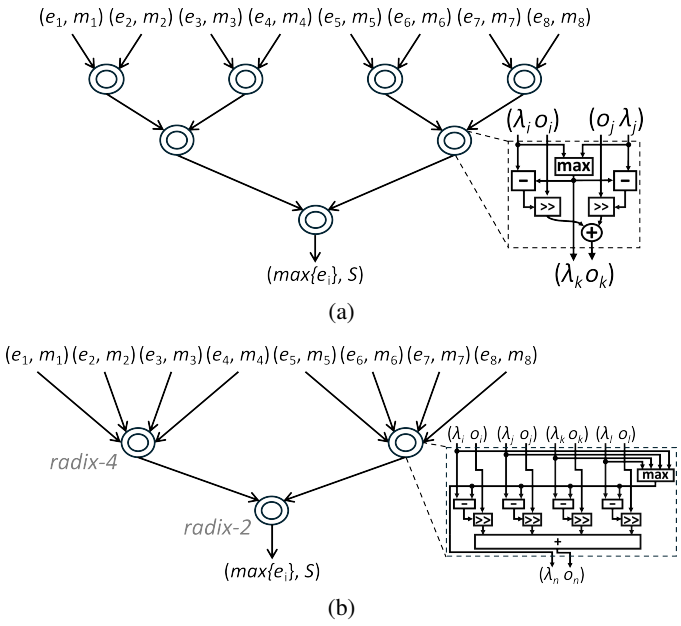

Fig. 2. Tree-based organization of parallel alignment and addition for an 8-term floating point addition using (a) the radix-2 $\circledcirc$ operator in all nodes of the tree and (b) a mixture of radix-4 and radix-2 operators.

Radix-4 operators are used in the first level and a radix-2 operator at the last level. For the rest of the paper this configuration would be denoted as a 4-2 solution. Equivalently, the 8-term adder of Fig. 2(a) would be denoted as a 2-2-2 solution highlighting the radix of the operators used in each level of the tree.

A radix-4 operator effectively follows the baseline architecture shown in Fig. 1 for 4 inputs, i.e., it finds first the maximum of the 4 exponents and subtracts it from all input exponents. The exponent differences are used for aligning the 4 fractions before adding them. In fact, the proposed approach is a generalization of the baseline alignment and addition. The baseline approach for an N-term adder, shown in Fig. 1, is effectively a sub-solution of the proposed approach and uses a *single* radix-N operator.

IV. Evaluation

Experimental evaluation aims at exploring the effectiveness of the proposed alignment and addition architecture, for building multi-term fused FP adders relative to the widely-used baseline approach. For this reason, we implemented 16, 32 and 64-term adders for the four FP-arithmetic formats shown in Fig. 3 covering single and reduced-precision formats [1]. For the proposed designs, for each multi-term adder we explored all possible configurations using align-and-add operators of various radices (i.e., number of inputs).

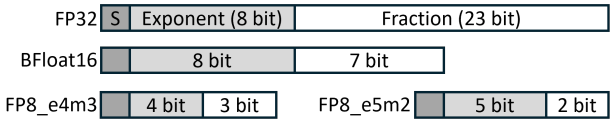

Fig. 3. Structure of commonly used FP data types.

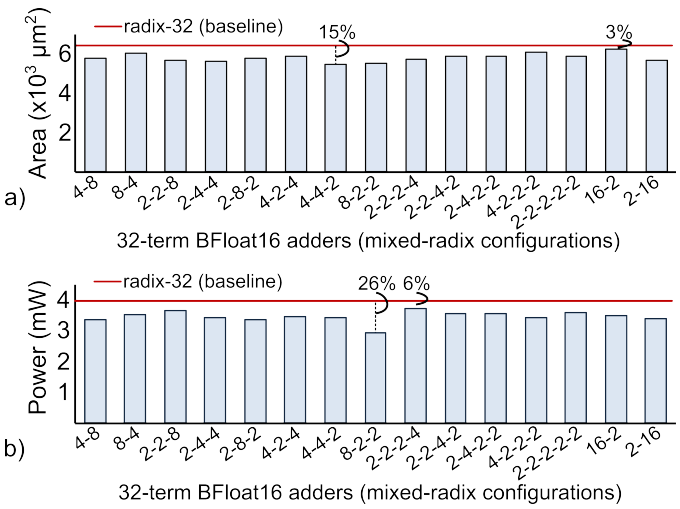

Fig. 4. The a) area and b) average power of 32-term BFloat16 adders designed with the baseline approach and the proposed approach that uses the newly introduced align-and-add operator ⊙ in various mixed-radix configurations.

All the multi-term FP adders under comparison, were implemented in C++[1] and synthesized to Verilog using Catapult HLS, using a 28-nm standard-cell library. All designs (i.e., proposed and *baseline*) operate at a clock frequency of 1 GHz and implement a complete multi-term fused FP addition that includes fraction alignment and addition as well as normalization and rounding of the final sum. To achieve the target clock frequency, HLS synthesis was instructed to produce designs with appropriate pipeline depth, depending on the number of input terms and their data type. As the number of input terms increases so does the design's pipeline depth. For an N-term FP32 adder we aimed for $\log_2 N$ pipeline stages. HLS can derive many other pipelined solutions. However, to simplify comparisons across designs, we selected the same configuration for all cases. For lower-precision data types, such as BFloat16 and FP8, one pipeline stage less relative to FP32 is enough to reach the targeted clock frequency due to smaller mantissa and exponent bit widths. The final area results were derived from Oasys logic synthesis tool. The power consumption was estimated after synthesis using the PowerPro power analysis and optimization tool. For power estimation, we employed multi-term adders in matrix multiplication kernels for the BERT Transformer [17] using input data from the GLUE dataset [18].

A. Design-space exploration for 32-term BFloat16 adders

In order to assess how mixed-radix configurations perform relative to the baseline align-and-add approach in multi-term floating-point adders, we initially focused on the case of 32-term BFloat16 adders. The designs presented represent complete multi-term floating point adders and the baseline approach differs from the proposed designs only in the alignment and addition logic. Normalization and rounding are the same for all designs under comparison.

[1]available at `github.org/ic-lab-duth/online-fp-add.git`

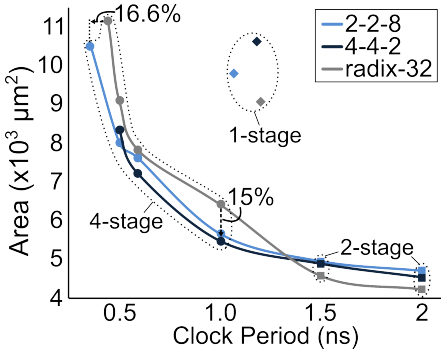

Fig. 5. The most area efficient designs achieved by each configuration for 32-term BFloat16 for various clock period targets using 1–4 pipeline stages.

Fig. 4 depicts the area and power of the proposed adders, that follow different mixed-radix configurations relative to the baseline approach, which effectively uses a single N-input operator. In all cases, utilizing a mixed-radix configuration proves more efficient than the radix-32 *baseline* configuration. From the results shown in Fig. 4(a) the proposed designs can achieve area savings that range between 3% and 15%. The 4-4-2 configuration offers the best area efficiency, reducing area by 15%. As shown in Fig. 4(b), the proposed mixed-radix designs achieve power reductions of 6% to 26%. The optimal configuration, in terms of power consumption, is the 8-2-2 design, achieves a notable 26% power reduction.

The proposed formulation splits alignment and addition to smaller hardware blocks thus increasing hardware modularity. In effect, this transformation, allows HLS to schedule intermediate alignment and addition steps to pipeline stages with better flexibility that results in more efficient designs.

This modular approach enhances also the delay characteristics of multi-term adders across different pipelined configurations. Fig. 5 illustrates the most area-efficient 32-term BFloat16 adders produced for various clock period targets. For high-frequency applications, 4-stage pipelines excel, while 2-stage pipelines are more area-optimal at lower frequencies. The proposed 2-2-8 configuration stands out for its speed, offering a 16.6% faster clock cycle than the baseline design with the same number of pipeline stages. In terms of area, similar to Fig. 4(a), the 4-4-2 design is the most compact at 1 ns. However, for less stringent clock requirements, the baseline design provides the best area-performance trade-off. For completeness, we also included the fastest single-cycle (1-stage) implementations for each design. In all cases, their equivalent pipelined solutions offer a superior combination of speed and area efficiency for 32-term adders.

B. Multi-term adders for various FP formats

As previously demonstrated, the proposed approach performs well for building 32-term BFloat16 adders. Nevertheless, it is essential to verify that this efficiency extends to adders with fewer or more inputs and to other FP data types. A more extensive analysis will offer a comprehensive understanding of the effectiveness of FP adders built using the proposed parallel align-and-add architecture.

TABLE I
THE AREA AND POWER FOR (A) 16, (B) 32 AND (C) 64-INPUT MULTI-TERM ADDERS AND FOR VARIOUS FP DATA TYPES.

N = 16	**Area** ($\times 10^3 \mu m^2$)			**Power (mW)**		
	Base	Proposed	Save	Base	Proposed	Save
FP32	8.87	6.8 (8-2)	23%	3.03	2.65 (8-2)	13%
BFloat16	2.92	2.69 (8-2)	8%	1.61	1.35 (8-2)	16%
FP8_e4m3	1.29	1.23 (8-2)	4%	0.83	0.69 (8-2)	17%
FP8_e5m2	1.17	1.23 (2-4-2)	-5%	0.62	0.70 (2-4-2)	-13%
FP8_e6m1	1.33	1.36 (4-2-2)	-2%	0.49	0.54 (4-2-2)	-10%

(a) 16-term FP adders

N = 32	**Area** ($\times 10^3 \mu m^2$)			**Power (mW)**		
	Base	Proposed	Save	Base	Proposed	Save
FP32	16.24	14.02 (2-2-2-2-2)	14%	6.69	5.78 (2-2-2-2-2)	14%
BFloat16	6.44	5.5 (8-2-2)	15%	3.97	2.92 (8-2-2)	26%
FP8_e4m3	3.02	2.51 (8-2-2)	17%	1.85	1.53 (8-2-2)	17%
FP8_e5m2	2.73	2.44 (8-2-2)	11%	1.74	1.44 (8-2-2)	17%
FP8_e6m1	2.80	2.48 (8-2-2)	11%	0.76	0.63 (8-2-2)	18%

(b) 32-term FP adders

N = 64	**Area** ($\times 10^3 \mu m^2$)			**Power (mW)**		
	Base	Proposed	Save	Base	Proposed	Save
FP32	32.51	28.67 (2-2-2-2-4)	12%	13.26	10.82 (2-2-2-2-4)	19%
BFloat16	12.84	11.73 (2-4-2-2-2)	9%	7.30	7.05 (2-4-2-2-2)	4%
FP8_e4m3	5.79	5.09 (8-4-2)	12%	3.62	3.01 (8-4-2)	17%
FP8_e5m2	5.34	4.78 (8-8)	11%	3.35	2.78 (8-8)	17%
FP8_e6m1	5.39	4.86 (2-8-4)	10%	1.62	1.35 (2-8-4)	17%

(c) 64-term FP adders

Table. IV-A presents the area and power performance of all designs under comparison for 16, 32 and 64 inputs and for the FP formats shown in Fig. 3. To examine also a corner case, where the exponent differences are large relative to the mantissa's bit width, we included also an additional 8-bit FP datatype `FP8_e6m1`.

For the proposed designs, we only report the configuration with the best area/power performance. The selected configuration is indicated inside the parenthesis below the results of the proposed designs.

The performance gains achieved by the proposed designs depend mainly on the number of input terms and are consistent across all examined FP data types. As shown in Table IV-A, adders with a large number of input terms, like 32 or 64, demonstrate a more pronounced benefit compared to those with a lower number of inputs. The size of the exponent field also influences the effectiveness of mixed-radix designs. As the size of the exponent increases, exponent calculation and fraction alignment and addition become equally critical. This convergence reduces the efficiency of interleaving maximum exponent identification and fraction alignment and addition that is leveraged by the proposed designs. Overall, the impact of the number of input terms on performance improvements is more significant than that of larger exponent fields.

V. CONCLUSIONS

This work reformulates the decades-old problem of serial alignment and addition appearing in multi-term FP adders in a new *online* form. The proposed computation paradigm allows maximum exponent identification, exponent subtraction, alignment shift and addition to be computed incrementally and in parallel. Alignment and addition logic can be structured in a tree-like structure using the newly introduced align-and-add operator. Operators with varying numbers of inputs can be employed at each level of the tree. Hardware evaluation confirms that this approach substantially reduces the complexity of alignment and addition, resulting in faster multi-term FP adders or designs with smaller area and lower power consumption compared to conventional approaches.

REFERENCES

[1] L. Bertaccini *et al.*, "MiniFloat-NN and ExSdotp: An ISA extension and a modular open hardware unit for low-precision training on RISC-V cores," in *IEEE Symp. on Computer Arithmetic (ARITH)*, 2022.

[2] N. P. Jouppi *et al.*, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *Int. Symp. on Computer Architecture (ISCA)*, 2021, pp. 1–14.

[3] S. Wang and P. Kanwar, "BFloat16: The secret to high performance on Cloud TPUs," *Google Cloud Blog*, vol. 4, 2019.

[4] P. Micikevicius *et al.*, "Fp8 formats for deep learning," *arXiv preprint arXiv:2209.05433*, 2022.

[5] Y. Tao *et al.*, "Correctly rounded architectures for floating-point multi-operand addition and dot-product computation," in *IEEE Int. Conf. on Application-Specific Systems, Architectures and Proc. (ASAP)*, 2013.

[6] J. Sohn and E. E. Swartzlander, "A fused floating-point three-term adder," *IEEE Trans. on Circuits and Systems I*, vol. 61, no. 10, pp. 2842–2850, 2014.

[7] ——, "A fused floating-point four-term dot product unit," *IEEE Trans. on Circuits and Systems I*, vol. 63, no. 3, pp. 370–378, 2016.

[8] D. Filippas, C. Peltekis, G. Dimitrakopoulos, and C. Nicopoulos, "Reduced-precision floating-point arithmetic in systolic arrays with skewed pipelines," in *IEEE Int. Conf. on Artificial Intelligence Circuits and Systems (AICAS)*, 2023.

[9] M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax," *arXiv preprint arXiv:1805.02867*, 2018.

[10] B. Hickmann *et al.*, "Intel nervana neural network processor-t (NNP-T) fused floating point many-term dot product," in *IEEE Symp. on Comp. Arith. (ARITH)*, 2020.

[11] D. Filippas, C. Nicopoulos, and G. Dimitrakopoulos, "Templatized fused vector floating-point dot product for high-level synthesis," *Journal of Low Power Electronics and Applications*, vol. 12, no. 4, 2022.

[12] O. Desrentes, B. D. de Dinechin, and F. de Dinechin, "Exact fused dot product add operators," in *IEEE Symp. on Comp. Arith. (ARITH)*, 2023.

[13] A. F. Tenca, "Multi-operand floating-point addition," in *IEEE Symp. on Computer Arithmetic (ARITH)*, 2009.

[14] H. Kaul *et al.*, "Optimized fused floating-point many-term dot-product hardware for machine learning accelerators," in *IEEE Symp. on Computer Arithmetic (ARITH)*, 2019.

[15] Y. Uguen and F. de Dinechin. (2017) Design-space exploration for the Kulisch accumulator. [Online]. Available: https://hal.science/hal-01488916

[16] J. Koenig *et al.*, "A hardware accelerator for computing an exact dot product," in *IEEE Symp. on Comp. Arith. (ARITH)*, 2017.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Assoc. for Comp. Linguistics*, 2019.

[18] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.