2018 International Conference on Field-Programmable Logic and Applications

An Efficient Exact Fused Dot Product Processor in FPGA

Luís Fiolhais, Horácio Neto

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
luis.azenhas.fiolhais@tecnico.ulisboa.pt, hcn@inesc-id.pt

***Abstract*—The objective of this work is to develop an efficient hardware processor to perform the fused dot product operation with full accuracy. General purpose processors split the dot product into two independent operations, rounding the full precision results of the multiplication and accumulation to the format's precision, which may significantly affect the overall accuracy. Therefore, many systems must rely on software approaches to achieve numerical accuracy, which imposes a significant performance overhead.**

This work proposes a fused dot product processor which is able to output a partial result per cycle with exact operand arithmetic. The processor uses a long-accumulator with full fixed-point precision to achieve full accuracy. Signed addition support is achieved using a Generalized Signed-Digit redundant numeric representation, which avoids sign extending between segments every cycle. In addition, the proposed architecture uses a novel autonomous carry propagation unit, which searches, selects and propagates stray carries between segments, while the accumulation process is executing.

The proposed processor has been implemented in a Zynq 7020-1 FPGA, where a single-precision core can execute with a throughput of one partial result per 90 MHz clock cycle and occupies about 5K LUTs. Full accuracy has been demonstrated using a set of hard to correctly solve benchmarks.

***Index Terms*—exact dot product, floating-point, exact arithmetic, segmented accumulator, FPGA, Generalized Signed Digits**

I. Introduction

In scientific computations high performance floating-point arithmetic is frequently required. In fact, the most used operation sequence is a multiplication followed by an addition, which justified the IEEE standardisation of a fused multiply-add operation. However, there is still no standard for the dot product operation.

A unit optimised to compute the dot product will process a multiplication on the elements of two vectors followed by a continuous accumulation of the multiplication results. To maintain accuracy the result of the multiplication must be used directly, without any rounding, in the accumulation. The same must occur for the accumulation, all digits of the accumulator must be used, when adding.

In a General Purpose Processor (GPP) the dot product is split into two operations: one multiplication and one addition. This separation causes errors in the computation because of the rounding between operations. Consider one example where the GPP processes the dot product of two vectors of five elements represented in the IEEE 754 single-precision format [1].

$$\begin{bmatrix} 2.718281828 \times 10^{10} \\ -3.141592654 \times 10^{10} \\ 1.414213562 \times 10^{10} \\ 5.772156649 \times 10^{9} \\ 3.010299957 \times 10^{9} \end{bmatrix} \text{ and } \begin{bmatrix} 1.4862497 \times 10^{12} \\ 8.783669879 \times 10^{14} \\ -2.237492 \times 10^{10} \\ 4.773714647 \times 10^{15} \\ 185049 \times 10^{0} \end{bmatrix}$$

The correct result of the dot product is $-1.00657107 \times 10^{8}$ but the result of its computation in a GPP is wrongly given as -2.305286×10^{18}.

Programmers typically use a larger floating-point format to try to cope with the intermediate computation results, even though there is no guarantee of accurate results. For example, performing the computations for the dot product above, using double-precision arithmetic in a GPP, still gives a wrong result: $4.328386285 \times 10^{9}$. This example fully illustrates that the dot product operation should be computed as a single fused operation, and not as two independent operations.

The requirement for accurate dot product floating-point arithmetic is present in a large number of application areas, for example financial engineering [2], and 3D graphics [3]. Therefore, the availability of a fully-accurate hardware implementation of a dot product floating-point operation adds a meaningful value to a significant number of applications, both in the correctness of the final results and in the performance improvement.

II. Background

The closest operation to the dot product in modern processors is the fused multiply-add operation. The fused multiply-add operation is defined as $(A \times B) + C$, which should be processed without a range and precision limit and with only one rounding [4]. Presently, Graphical Processing Units (GPUs) and GPPs implement this operation in hardware, compliant with this specification [5], [6]. However, the standard fused multiply-add operation can only maintain precision for one addition, while full accumulation accuracy requires maintaining precision for n accumulations.

A. Base Exact Dot Product Architectures

The first architecture to process the dot product with full accuracy was proposed by Kulisch in [1] and named the Long Accumulator (LA). The LA algorithm computes

978-1-5386-8517-4/18/$31.00 ©2018 IEEE
DOI 10.1109/FPL.2018.00062

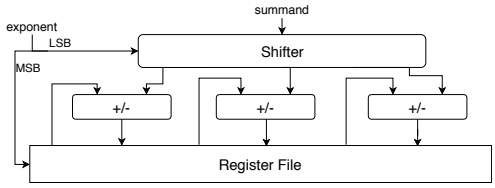

Fig. 1. Central Accumulator with three adders. Adapted from [1]

an accumulation with full accuracy, using full-width fixed-point accumulations. The result of the multiplication of two operands with l-bit significands and exponents e_1 and e_2, has a $2l$ bit significand and exponent $e_1 + e_2$. Thus, the largest/smallest possible multiplication result will have a $2e_{max}/2e_{min}$ exponent. To allow for the full operand size and including k bits to guard against possible overflows, the fixed-point accumulation must have a size [1]:

$$L = k + 2e_{max} + 2l + 2|e_{min}| \quad (1)$$

A direct implementation of the long accumulator would use a long shifter and a long adder. Performing the accumulation with a full width ripple-carry adder is unpractical as a carry propagation across it would be too expensive.

In [7], Dinechin directly implements the long accumulator optimized for a specific set of applications. By knowing the order of magnitude of the most and least significant bits of the input data, the size of the accumulation register and the alignment shift can be reduced. This type of architecture targets applications which require a small accumulation register (L). However, supporting a generic set of applications requires a full accumulation register and shifter.

To break the wide critical delay path across the accumulator and the shifter, Kulisch proposed to split the large accumulation register and adder across many small registers and adders, designating each register as *segment* [1]. Consequently, the segment's adder width is reduced to the segment size, thus breaking the long carry propagation path along all segments. This type of addition is generally referred to as *segmented accumulator*. By selecting the size y of the segments as a power of two (and greater than two), the segment selection and the number of bits to be shifted can be directly obtained from the summand's exponent.

Kulisch proposed two segmented accumulator architectures to process the dot product: the centralized accumulator and the matrix accumulator [1].

The *centralized accumulator* performs the accumulation between the summand and the corresponding subset of segments (Fig. 1). Thus, the number of adders used is equal to the number of segments to be updated. In the event of a carry generation, the accumulation processor must stall while carries propagate.

The *matrix accumulator* includes a dedicated adder in every segment, and uses a one bit carry register between consecutive segments (Fig. 2). Thus, the accumulator processor will never stall because there will always be an available accumulator to propagate any carry between any two successive segments.

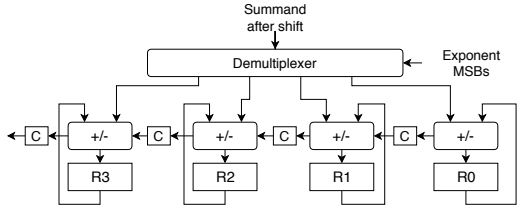

Fig. 2. First Row of the Matrix Accumulator. Adapted from [1]

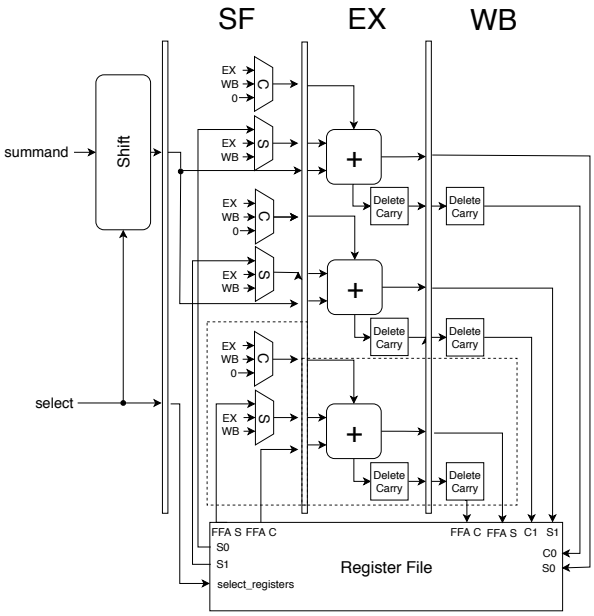

Fig. 3. Segmented Accumulator

Both architectures use the sign-magnitude representation. Thus, there is one full adder and one full subtractor per segment, and a carry and a borrow must be propagated. To reduce the number of cycles required by carry/borrow propagation, Kulisch proposed a carry skip mechanism, which searches for the first segment that will absorb the carry/borrow without generating another one [1].

B. Recent Contributions

In [8], Uguen proposed replacing the full adder and full subtractor in each segment of the matrix accumulator, with a single two's complement full adder which reduces the complexity of the carry/borrow propagation. However, when a negative number is in the accumulator its sign must be propagated to the most significant segment starting from the summand's selected segments.

Other recent works implemented Kulisch's architectures directly in FPGAs and ASICs. In [9], Nowak directly implemented Kulisch's matrix dot product processor in a Virtex-4. In [10], Koenig implemented Kulisch's original designs as accelerators in Rocket Chip, a RISC-V based SoC, in an TSMC 45nm technology.

III. Architecture Description

This work proposes an Exact Dot Product Core (EDPC), with three main stages: multiplication, accumulation, and conversion to IEEE 754 floating-point.

The first stage splits the inputs into three components (sign, exponent and significand), and multiplies the significands and adds the exponents. The result of the multiplication is then aligned using the result of the exponent addition, and is passed on to the segmented accumulator stage, which is the most complex part of the architecture.

A. Segmented Accumulator Architecture

The segmented accumulator (Fig. 3) is able to output one partial result per cycle by analyzing carry propagation as a scheduling issue. Each segment is stored in a register file, and each carry is stored in another, independent, register file. The accumulation process is subdivided in three simple and clearly defined pipelined functional stages, similarly to a GPP:

- The Segment Fetch (SF) stage: reads the segments from the register file and splits the summand into two or more segments;
- The Execute (EX) stage: performs the accumulation;
- The Write Back (WB) stage: writes the results of the EX stage to the register file.

Providing one partial result per cycle requires propagating the carries without stalling the accumulation processor. To do so, the carries are pooled in their register file, and propagated when they do not interfere with the accumulation process. For this, all carry propagation tasks are handled by a dedicated carry processing block with one adder, the Free Flow Adder (FFA).

The FFA is an autonomous block which is able to search, select and propagate any carry. In every cycle, the FFA selects the least significant segment which has a carry to propagate. As the accumulation process is split in three pipeline stages, the FFA must not select the previous two selections. If no carry register has a carry to propagate then the FFA idles. Furthermore and to reduce the FFA workload, the architecture takes advantage of temporal locality between consecutive summands, reading their updated results and carries from advanced stages.

B. Signed Addition Support

The signed numeric representation of choice should be one which allows for negative carries and segments, and each segment must be able to completely absorb any carry generated. These two requirements are necessary so that accumulating a negative summand does not force a sign propagation until the most significant segment, and so that the number of carries which can be generated in a single accumulation is minimized. Therefore, a redundant numeric representation is used, the Generalized Signed-Digits (GSDs), which provides a resource efficient support of subtraction operations, and of the carry propagation [11]. This redundant digit set supports negative digits and carries, which avoid sign extending every cycle. As such, the digit set used is bound by the maximum/minimum carries plus the maximum/minimum representable number by the base, where the base is 2^y and y is the segment size.

Using GSDs also guarantees a carry is completely absorbed by the next segment. Thus, carry propagations are always halted in the next segment.

TABLE I
Result comparison between Arbitrary Precision (AP), the Single-Precision EDPC, and the single- and double-precision floating-point units present in a GPP

Data	Taylor Exp.	HC	Higham
AP (100 d.)	1.873 410e−3	2.0e−18	1.644 689
EDPC (SP)	1.873 410e−3	2.0e−18	1.644 689
GPP (SP)	1.877 666e−3	0	1.655 725
GPP (DP)	1.867 449e−3	0	1.644 834

C. Conversion Unit

After all accumulations are completed, the final result is converted back and truncated into sign-magnitude. The conversion from GSD to IEEE 754 follows three steps: each digit is multiplied by its radix order (shift), all digits are accumulated, and the final result is converted from two's complement to sign-magnitude. Since the final result is truncated, the conversion unit only needs the three most significant non-zero digits. Finally, the sign-magnitude result is normalized.

IV. FPGA Implementation Results

The EDPC was synthesized using Vivado 2017.4 for the Zynq 7020-1 device, and was implemented for three IEEE 754 floating-point formats (16-bit, 32-bit and 64-bit) with different segment sizes.

The best segment size, for both the lowest resource usage and highest clock frequency, is: for a 16-bit floating-format a segment size of 32 bits (2k LUTs and 111 MHz), for a 32-bit floating-format a segment size of 64 bits (5k LUTs and 90 MHz), and for a 64-bit floating-point format a segment size of 128 bits (24k LUTs and 63 MHz). The critical path is in the GSD to sign-magnitude conversion unit.

V. Evaluation Results

The single- and the double-precision EDPC were evaluated by executing hard to correctly solve test sets, and their resource consumptions and latencies were compared with previous exact dot product processor implementations. The single-precision EDPC was configured with 64-bit segments and an accumulation register of 564 bits, and the double-precision EDPC was configured with 128-bit segments and an accumulator register of 4206 bits.

The single-precision EDPC was evaluated using three (heavy cancellation, taylor series expansion, and Higham's summation) hard to compute test sets described in [12], adapted for single-precision. These test sets were processed in: an arbitrary precision calculator, `bc`, with a precision of 100 decimal places; the single-precision EDPC, using the Zynq DMA to directly access external memory; and the single- and double-precision floating-point units present in an Intel Core i5-5257U.

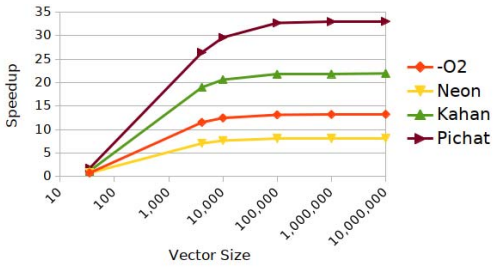

Fig. 4. Speedups for different vector sizes, using the ARM FP Unit (with -O2), the ARM NEON Unit, Kahan's and Pichat's algorithms

The results obtained for arbitrary precision, single-precision EDPC and the GPP are shown in Tab. I. As shown, the EDPC successfully computed all tests exactly.

Furthermore, the single-precision EDPC was compared with the single-precision floating-point unit present in the ARM Cortex-A9, and with software implementations of Kahan's [13] and Pichat's [14] summation algorithms, using the same test sets from [12]. The speedups obtained for each vector size are shown in Fig. 4.

The number of clock cycles taken by the current configuration of the EDPC to output the final result, in a general case, is: vector size + pipeline stages + 2. For example, the latency for two vectors of size 1000 is 1014 clock cycles.

The resource consumption and latencies of previous works, the EDPC for single- and double-precision, and a MAC unit made up of Xilinx IP blocks [15] (a floating-point multiplication unit and a floating-point accumulation unit), are compared in Tab. II.

As shown and to the authors knowledge, the EDPC uses the least amount of resources and has lower latency than current exact dot product processor architectures, for both single and double precision operands.

VI. Conclusion and Future Work

In this work, an efficient exact dot product processor is proposed, which is able to output one partial result per cycle. A FFA is used to autonomously search, select and propagate carries, which operates in parallel with the segmented accumulator. The signed addition takes advantage of GSDs which support negative digits and carries, and therefore guarantee that each digit is able to completely absorb any carry created by the previous segment, such that the sign extension is always halted at the next segment.

The architecture was implemented in a Z7020-1 FPGA for half, single and double precision formats. A single-precision exact dot product processor operates at a frequency of 90 MHz, and uses about 5k LUTs. The dot product core was evaluated using hard to correctly solve test sets, and was compared with previous exact dot product processor architectures. The results showed that the EDPC uses the least amount of resources and has lower latency than current exact dot product processor architectures.

As future work the GSD to IEEE 754 conversion unit is going to be optimized. Pipelining this block will shorten the critical path further and thus increase the overall working clock frequency.

TABLE II
Resource Usage and Latency Comparison between Single- and Double-Precision Architectures from Uguen's, Nowak's (Normalized for the Same Technology), the MAC Unit, and EDPC

Resources Single/Double	Uguen	Nowak	MAC	EDPC
LUTs	5.9k / 58k	7.3k / -	2.1k / 21k	5.3k / 24k
FF	9.7k / 82k	5.1k / -	0.7k / 5k	1.9k / 8k
DSPs	2 / 9	2 / -	12 / 55	2 / 9
Latency (clock cycles)	1098 / 1707	-	1008	1014
LA Size (bits)	554 / 4196	640 / -	331 / 2152	564 / 4206

Acknowledgments

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

References

[1] U. W. Kulisch, "Chapter 8 - Scalar Products and Complete Arithmetic," in *Computer Arithmetic and Validity - Theory, Implementations, and Applications*. Walter de Gruyter, Berlin, 2008, pp. 245–300.

[2] D. B. Thomas, "Acceleration of financial monte-carlo simulations using fpgas," in *2010 IEEE Workshop on High Performance Computational Finance*, Nov 2010, pp. 1–6.

[3] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Bdam - batched dynamic adaptive meshes for high performance terrain visualization," vol. 22, pp. 505–514, 09 2003.

[4] "IEEE standard for floating-point arithmetic," IEEE Computer Society, Tech. Rep., 2008.

[5] "Precision and performance: Floating point and IEEE 754 compliance for NVIDIA GPUs," NVIDIA Corporation, Tech. Rep., 2015.

[6] Intel, "Intel Intrinsics Guide," 2008. [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=fma&techs=FMA&expand=2407

[7] F. D. Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific Approach to Floating-Point Accumulation and Sum-of-Products," in *FieldProgrammable Technology, IEEE*, 2008, pp. 33–40.

[8] Y. Uguen and F. de Dinechin, "Design-space exploration for the Kulisch accumulator ," Mar. 2017, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01488916

[9] F. Nowak, R. Buchty, D. Kramer, and W. Karl, "Exploiting the HTX-Board as a Coprocessor for Exact Arithmetics," in *Proceedings of the First International Workshop on HyperTransport Research and Applications (WHTRA 2009)*. Universittsbibliotek Heidelberg, February 2009, pp. 20–29.

[10] J. Koenig, D. Biancolin, J. Bachrach, and K. Asanovic, "A hardware accelerator for computing an exact dot product," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, July 2017, pp. 114–121.

[11] B. Parhami, "Generalized Signed-Digit Number Systems: A Unifying Framework for Redundant Number Representations," *IEEE Transactions on Computers*, vol. 39, no. 1, pp. 89–98, 1990.

[12] J. M. McNamee, "A Comparison of Methods for Accurate Summation," *ACM SIGSAM Bulletin*, vol. 38, pp. 1–7, March 2004.

[13] W. Kahan, "Pracniques: Further Remarks on Reducing Truncation Errors," *Communications of the ACM*, vol. 8, no. 1, pp. 40, 48, January 1965.

[14] M. Pichat, "Correction d'une somme en arithmtique virgule flottante," *Numerische Mathematik*, vol. 19, no. 5, pp. 400 – 406, October 1972.

[15] Xilinx, "Floating-Point Operator v7.1," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf