A matrix-multiply unit for posits in reconfigurable logic leveraging (Open)CAPI

Chen, Jianyu ; Al-Ars, Zaid; Hofstee, H. Peter

DOI
10.1145/3190339.3190340

Publication date
2018

Document Version
Final published version

Published in
Proceedings of the Conference for Next Generation Arithmetic, CoNGA 2018

A Matrix-Multiply Unit for Posits in Reconfigurable Logic Leveraging (Open)CAPI

Jianyu Chen
Delft University of Technology
Delft, Netherlands
j.chen-13@student.tudelft.nl

Zaid Al-Ars
Delft University of Technology
Delft, Netherlands
z.al-ars@tudelft.nl

H. Peter Hofstee*
Delft University of Technology and
IBM Research
Austin, TX, US
hofstee@us.ibm.com

ABSTRACT

In this paper, we present the design in reconfigurable logic of a matrix multiplier for matrices of 32-bit posit numbers with es=2 [1]. Vector dot products are computed without intermediate rounding as suggested by the proposed posit standard to maximally retain precision. An initial implementation targets the CAPI 1.0 interface on the POWER8 processor and achieves about 10Gpops (Giga posit operations per second). Follow-on implementations targeting CAPI 2.0 and OpenCAPI 3.0 on POWER9 are expected to achieve up to 64Gpops. Our design is available under a permissive open source license at *https://github.com/ChenJianyunp/Unum_matrix_multiplier*. We hope the current work, which works on CAPI 1.0, along with future community contributions, will help enable a more extensive exploration of this proposed new format.

CCS CONCEPTS

• **Computer systems organization → Reconfigurable computing**;

KEYWORDS

Posit number, Matrix-Multiplier, Dot-product

ACM Reference Format:
Jianyu Chen, Zaid Al-Ars, and H. Peter Hofstee. 2018. A Matrix-Multiply Unit for Posits in Reconfigurable Logic Leveraging (Open)CAPI. In *Proceedings of Conference for Next Generation Arithmetic (CoNGA 2018)*, Jianyu Chen, Zaid Al-Ars, and H. Peter Hofstee (Eds.). ACM, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.1145/3190339.3190340

1 INTRODUCTION

In their paper "Beating Floating Point at its Own Game: Posit Arithmetic" [1], the authors introduce posit numbers, designed to replace the IEEE 754 standard. While the paper lays out several desirable properties of posits, application studies have been hindered by the lack of available hardware implementations. In this paper, we report our work on an implementation in reconfigurable logic of a posit matrix-multiply unit. One reason to build a matrix multiply (C=AB) is that one can consider this to be a streaming operation where each row vector of A multiplied with a stored matrix B produces an output row vector of C. When matrices are stored with entries in a row in successive memory locations, this allows for efficient memory access, both for accessing the stored matrix B (which can be read in row-wise) as well as the row vectors of matrices A and C. This organization also allows us, for matrices of sufficient size, to simultaneously maximize input and output bandwidth, as well as reach the maximum number of floating-point operations. Because a matrix multiply is decomposed into a set of vector dot product operations, this operation encourages us to implement the wide accumulator the proposed posit standard calls for to avoid intermediate rounding when calculating vector dot products [2]. We have chosen to leverage the (Open)CAPI interface [7] [8], first for ease of programming and to enable the host application to simply pass a (stack)pointer to the accelerator, and avoid both pinning and host-memory copy operations. Because the accelerator has unfettered access to host memory, the accelerator can optimize its memory access pattern to complete a full matrix multiply of any desired size without the need for different implementation-dependent host-side drivers. The OpenCAPI interface exceeds the bandwidth available with PCIe, and provides reduced-latency access to host main memory. The remainder of this paper is organized as follows: Section 2 of this document describes the 32-bit posit representation supported by this design. Section 3 describes the design of the dot product unit. Section 4 describes the design of the matrix multiply accelerator. Setion 5 discusses the measured and estimated performance results. Section 6 ends with the conclusions.

2 POSIT HARDWARE REPRESENTATION

The value of a posit number can be represented as described in the formula below:

$$(-1)^{sign\ bit} * useed^{k} * 2^{e} * f$$

where $useed = 2^{2^{es}}$, k is the integer represented by the regime bits, e is the unsigned integer represented by the exponent bits, and f is the fraction.
Before calculating posits on an FPGA, we transform the posit representation to a hardware friendly format. To achieve this, we need to define the length of posits and the value of es. This paper describes an implementation for 32-bit posits with es=2. For this case, the largest positive value is 2^{120} and the smallest positive value is 2^{-120}. Thus, the value of a posit can be represented in the formula below (the exponent value is an 8-bit unsigned integer calculated from the regime and exponent bits):

$$(-1)^{sign\ bit} * 2^{exp\_value-128} * f, where\ exp\_value = k * 2^{es} + e + 128$$

*Corresponding author

CoNGA 2018, March 28, 2018, Singapore, Singapore

ACM ISBN 978-1-4503-6414-0/18/03…$15.00
https://doi.org/10.1145/3190339.3190340

The exponent value is an unsigned integer calculated by regime bits and exponent bits. The number of regime bits is variable in a posit. To calculate the number of regime bits and obtain the position of exponent bits and fraction bits, a leading zero/one counter (LZC) [6] is applied from the 2nd bit to the end of the posit. The LZC result is a 5-bit integer with a length that is equal to the regime bits minus one. In addition, a left shift is applied so the exponent bits and fraction bits will be in a fixed position.

Next, the exponent value can be calculated by applying the process depicted in Figure 1. First, copy the first bit of regime bits to bit7 of the exponent value and copy the exponent bits to bit1 and bit0 of the exponent value. Then, if bit7 of the exponent value is 0, reverse every bit of the result of the LZC and assign it to the range from bit6 to bit2. If not, simply copy the result of LZC to the range bit6 to bit2.

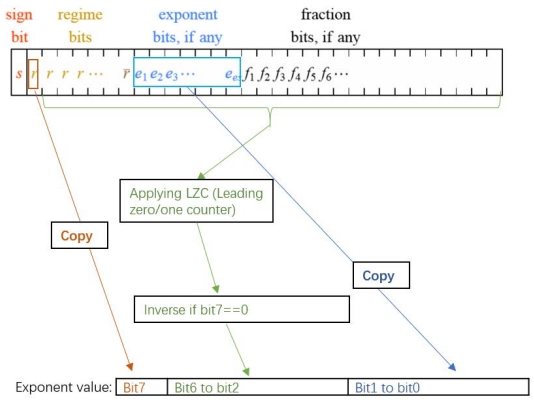

Figure 1: Transform regime bits and exponent bits to exponent value

3 DESIGN OF THE EXACT DOT-PRODUCT IN RECONFIGURABLE LOGIC

3.1 Principle of operation

For a posit, the largest magnitude is $useed^{n-2}$ and the smallest non-zero magnitude is $useed^{2-n}$, where useed is $2^{2^{es}}$ and n is the length of the posit. Therefore, the largest magnitude and smallest non-zero magnitude for a product of two posits are $useed^{2n-4}$ and $useed^{4-2n}$ respectively. Every product of two posits is an integer multiple of the smallest non-zero magnitude. So, our internal representation should be big enough to hold $useed^{4n-8}$. In practice, this number should be even bigger to account for the carries during accumulation. Thus, for our implementation, the largest magnitude and smallest non-zero magnitude are 2^{240} and 2^{-240} respectively, and therefore our representation should have at least 481 bits. To make the calculation convenient, a 512-bit quire register [1] is used to hold all the products. The 512 bits represent the magnitudes from 2^{255} to 2^{-256}. The result of the dot-product is correct within the rounding error of the final result, which is more accurate than using IEEE float, especially when accumulating a large number of values .

3.2 Principle of the carry-save dot product unit

For our choice of posit, fraction bits have a maximum length of 28 bits (including the leading 1 before a fraction, or 1.xx), thus the fraction of the product can be up to 56 bits. However, adding a 56-bit number to a 512-bit quire register requires a 512-bit adder, which is very expensive and slow.

If a short positive number is added to a much larger number and no carry occurs, only a small part of the larger number will be changed, and therefore most of the adder is wasted in most of cases. To avoid requiring a large adder for all cases, two problems should be solved [4]: 1) how to solve the carries? and 2) how to handle a negative number?

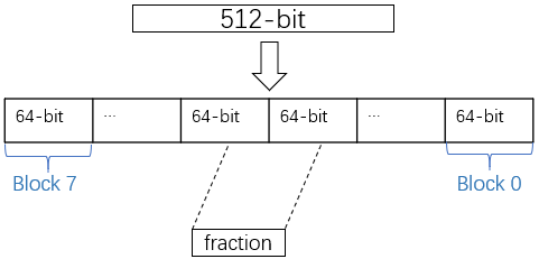

Figure 2: Each fraction has one or two corresponding blocks

As Figure 2 shows, the 512-bit number can be divided into 8 64-bit blocks, which are called Block7 to Block0. Every 56-bit fraction can be added to one block or two of these blocks, the other blocks will not be influenced if no carry happens. Therefore we use two random access memories to store the even and odd blocks. Each block stores the magnitude from $2^{0bXXX11111}$ (0b means binary number here) to $2^{0bXXX00000}$, where XXX is the number of the block in binary (from 0b111 to 0b000).

As Figure 3 shows, to store the carry, a 16-bit number for carry is appended to every 64-bit block, and each block is extended from 64 bits to 80 bits. Carry save adders are used here, once a carry exceeding the 64-bit block happens, it will be saved in its 16-bit [3].

The number of the carry bits is related to the length of adder, which determines the working frequency of this accumulator. On the other hand, the number of carry bits also determines the maximum number of carries. 16 bits are enough to hold $2^{15} - 1$ carries, which is enough for normal use. Our current implementation propagates carries at multiples of 64, which is less than this limit.

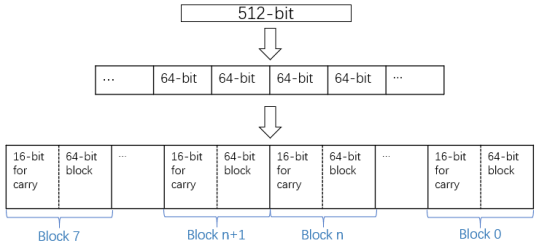

Figure 3: Append carry bits on the left

If a negative number is added, a borrow rather than carry may occur. A borrow can be regarded as “carry a negative number”. In

this case, 16-bit for carry should be a number in 2's complement, where its MSB is the sign bit.

3.3 Optimization by using FPGA RAMs

In order to access the blocks easily, the blocks are stored in RAMs. In this design, the blocks are stored in distributed RAM (registers) because the requirement of depth of RAM is only 4 in this design (one RAM for the even blocks and one for the odd blocks). However, the same idea can be applied for longer posit number with larger *es*. In that case, block RAM (BRAM) can be used to store more blocks in order to save LUT and FF resource in FPGA. It also possible to use BRAM in this design, but a lot of BRAMs storage will be wasted and working frequency will be influenced.

3.4 Calculation of the address and shift of the fraction

The fraction of the product is 56 bits long. Before the addition, the RAM address of its corresponding blocks should be found and it should be shifted to its posit. The block that contains the magnitude of the leading one of the fraction and its right block are the corresponding blocks. As Figure 4 shows, define bit8 to bit6 of exponent value as address bits and bit5 to bit0 as shift bits. From the magnitude range and address of the blocks, the block numbers of corresponding blocks can be found: address bits, address bits-1. When the address bit is 0b000, address bits-1 is 0b111, which is the leftmost block (Block 7). However, this is not a problem, because in this case the number added to Block 0b111 (Block 7) is always zero.

Figure 4: Address bits and shift bits

The address of the odd RAM and even RAM can be easily calculated from the address bits: the address of the even block is Bit8 Bit7, and the address of the odd block is calculated as Bit8 Bit7−(Bit6). Furthermore, 1.xx…xx is represented as 2^0 * 1.xx…xxx, and once we align the 1.xx…xxx to a block, its magnitude becomes $2^{0bXXXX00000}$ * 1.xx…xxx, where XXXX is the number of the block. As Figure 5 shows, the amount of the left shift is the number of shift bits. After the alignment and shift, the magnitude will become $2^{exponent\ value}$ * 1.xx…xxx.

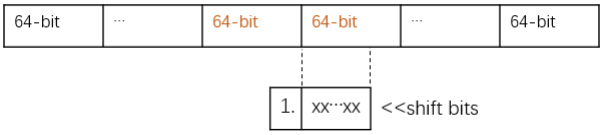

Figure 5: Align the fraction

3.5 Carry and normalization

To get the final result in 2's complement, the carry for each block should be calculated one by one from least to most significant. Thus the 2's complement number can be transformed into a 512-bit magnitude in the format below, where the sign bit is the MSB of the carry bits of the leftmost block (Block 7):

0 or 1	00···001XXX···XXX
Sign bit	512-bit fraction

Figure 6: The format of a 512-bit number in signed magnitude

The posit result can be obtained by normalizing this number. However, it is slower and more expensive to do the carry and normalization separately. A way to do carry and part of the normalization at the same time is introduced below. If the final result is positive, the result after carry (before normalization) will be in the format shown in Figure 6.

As Figure 7 shows, only the block with the leading one (the leftmost ONE) and its neighboring block to the right (the orange blocks in the figure) are useful to calculate the fraction. While the carry is done from right to left, we should buffer two blocks and buffer the address of the current block following the rules in Figure 9.

If the final result is negative, the result after carry (before normalization) will be in the format as Figure 8 shows.

0	···	00···000	00···000	00···001XX···XXX	XXX···XXX	XXX···XXX	···
Sign bit	···	64-bit block	64-bit block	64-bit block	64-bit block	64-bit block	···

Figure 7: Pattern of a 512-bit number in 2's complement when positive

1	···	11···111	11···111	11···110XX···XXX	XXX···XXX	XXX···XXX	···
Sign bit	···	64-bit block	64-bit block	64-bit block	64-bit block	64-bit block	···

Figure 8: Pattern of a 512-bit number in 2's complement when negative

In this case, only the block with the leading zero (the leftmost ZERO) and its right block (the orange blocks in figure) are useful to calculate the fraction. We should buffer two blocks and buffer the address of the current block following the rules in Figure 10.

Current block	Current address	Operation
All 0	Do not care	Nothing
Not all zero	Current address==buffered address +1	Buffer block2=current block Buffer block1=buffer block2
	Current address!=buffered address +1	Buffer block2=current block Buffer block1=all 0

Figure 9: Operation to get the result blocks when positive

As the sign of the result can only be calculated after all the carry bits have been processed, behavior of both positive numbers and negative numbers should be accounted for during the carry.

For a negative number, there is some extra work to transform it into signed magnitude format after carry. If the number is negative, the result of carry will be a sign bit and two buffer blocks as Figure 11 shows. The sign bit and buffer block2 represent a negative

number in 2's complement, which is called Number 1. Buffer block1 represents an unsigned positive number or zero, which is called Number 2. Also, the magnitude of LSB of buffer block2 is greater than MSB of buffer block1. When calculating the value of Number 1 + Number 2, Number 2 should borrow 1 from Number 1.

Current block	Current address	Operation
All 1	Do not care	Nothing
Not all one	Current address==buffered address +1	Buffer block2=current block Buffer block1=buffer block2
	Current address!=buffered address +1	Buffer block2=current block Buffer block1=all 1

Figure 10: Operation to get the result blocks when negative

Sign bit	Buffer block2	Buffer block1
1	11···110XX···XX	XX···XXX

Number 1 Number 2

Figure 11: Pattern of result blocks

3.6 Two ways of optimization by using double buffering structure

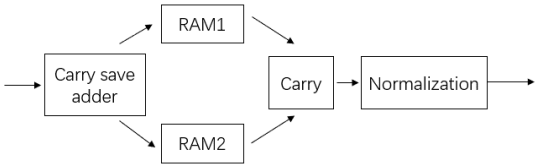

Figure 12: 1st double buffering structure

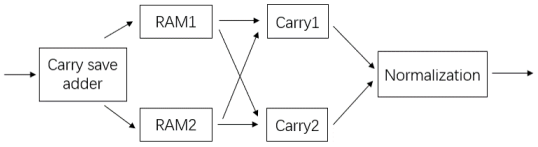

Figure 13: 2nd double buffering structure

An accumulation has three phases: addition, carry and normalization. After addition, the RAM arrays store the odd and even blocks, respectively, are occupied by the carry module, and the next accumulation cannot start immediately. To optimize our design, another RAM is added as Figure 12 shows. In the first dot-product calculation, the carry-save adder module operates on RAM1 (adds numbers to RAM1). After the addition of all the numbers, the carry-save adder module will operate on RAM2 and carry module on RAM1. Hence, the addition phase of the next accumulation can start at once. In this structure, the next accumulation can start immediately after current accumulation. However, to maintain full throughput, each accumulation should have at least eight pairs of numbers. Since the design of this matrix-multiplier focuses on calculating with larger matrices, the accumulator will be well utilized. However, there is another way to optimize. As Figure 13 shows, another carry module is added. In this structure, each accumulation should have at least four pairs of numbers to maintain full throughput.

4 DESIGN OF THE MATRIX-MULTIPLIER

For the construction of our matrix multiplier we assume our design consists of V vector dot-product units. Each dot-product unit stores a vector (a column of matrix B) of at most M elements. For the sake of simplicity this description assumes that all dot-product units are used for the same matrix B. Because we assume all matrices are stored in row-major order, it is advantageous to load the V dot-product units in parallel.

Currently, a matrix-multiplier with 64 dot-products is designed, which means it can calculate 64 numbers in result matrix (matrix C) at the same time.

When calculating two matrices with the size n*64 and 64*n, the matrix multiplication is carried out with the following sequence of steps:

1) As Figure 14 shows, all numbers in matrix A and matrix B are stored in RAM_A or RAM_B, respectively. Currently the RAM_A and RAM_B are implemented by BRAMs, which may be replaced by URAM in the future. RAM_A will output numbers in matrix A one by one. RAM_B will output numbers row by row, and 64 numbers in the same row in the result matrix will be calculated at the same time.

2) After the calculation of each row, the result will be stored in RAM_C.

3) Step 1 and 2 are repeated until every row is calculated.

When the input matrices are larger, the result matrix will be separated into some small matrix with 64 columns. Step3 is repeated until all the small matrices are processed. By processing a column of the larger matrix on a single dot-product unit without resetting the quire register, no intermediate rounding occurs.

Large matrices can be handled by dividing into sub-blocks and by performing a final carry step, but not the normalization step, and keeping the intermediate values in the quire registers. An upper limit on the size of the vectors is determined by the difference between the size of the quire registers (512 bits) and the minimum required range (481 bits) and thus vectors and array sizes of at least a billion elements can be supported.

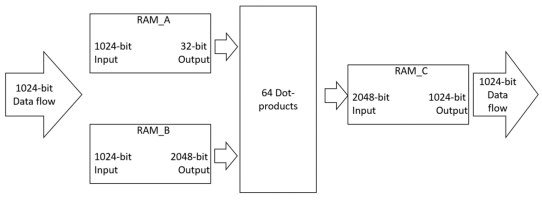

Figure 14: Structure of matrix multiplier

Because (Open)CAPI-based implementations have full access to shared memory, all the control for the matrix multiplier can be

created on the reconfigurable logic [5]. The host thread merely needs to wake up the FPGA logic and pass it a stack pointer, equivalent to doing a function call, but ensuring all parameters are in memory and not in registers. The host thread then suspends until the accelerator wakes it up on completion of the matrix multiply.

5 PERFORMANCE

We implemented our design on FPGAs and measured the performance. In the following, we report the results measured for our design running on configuration 1 below, and we estimate the performance on configurations 2&3.

Configuration 1. A Xilinx Virtex7 VX690 with a x8 CAPI 1.0 interface.

Configuration 2. A Xilinx Virtex Ultrascale Plus VU3P with a x8 OpenCAPI 3.0 interface.

Configuration 3. A Xilinx Virtex Ultrascale Plus VU37P with a x16 OpenCAPI 3.0 interface.

These devices and configurations have the characteristics listed in Table 1.

FPGA	Eff. BW	DSP	FF	LUT	BRAM	URAM
VX690	4GB/s	3600	866K	433K	52.9Mb	-
VU3P	16GB/s	2280	788K	394K	25.3Mb	90Mb
VU37P	32GB/s	9024	2607K	1304K	70.9Mb	270Mb

Table 1: Resources available in the 3 (Virtex) FPGA configurations

Unit type	FF	LUT	DSP	Frequency
Single posit dot-product	1031	2618	4	200MHz
32-bit float number multiply-adder	1325	866	4	300MHz

Table 2: Resource comparison and operating frequency of the posit dot-product and float number multiply-adder design

Table 2 shows the required resources and working frequency for a single dot-product unit with the comparison with a multiply-adder for 32-bit single IEEE float number which is generated by Vivado IP core. Leaving room for interface logic, configurations 1&2 can be expected to support 64 such units, limited by the number of BRAM in configuration 2 and targeting a single design for both 1&2 and configuration 3 at least 256 such units. Configuration 1 should be able to support vector lengths of at least 1,024 (stored) elements with up to 32K long vectors efficiently supported by configurations 2&3 leveraging the available URAMs in those FPGAs. The working frequency for matrix-multiplier is 125MHz. A bandwidth of 4/16/32GB/s corresponds to 1/4/8 Giga elements per second. For an FPGA operating at 125MHz, 8/32/64 elements per cycle. For a single input element per cycle and n vector units, n outputs are generated every vector-length cycles and one per cycle if vector length = n, matching the input bandwidth. For shorter vectors the design is bandwidth limited, and for longer vectors the design is arithmetic speed limited. Maximum expected posit(-multiply accumulate, counting each separately) operations per second (pops) is 16Gpops for configurations 1&2 and 64Gpops for configuration 3. Measured peak performance of our current implementation is about 10Gpops.

6 CONCLUSIONS

This paper presented an initial design of an FPGA-based posit matrix-multiply unit. The design is estimated to deliver between 16Gpops and 64Gpops 32bit (es=2) posit-operations per second using currently available reconfigurable logic devices. The design is built with dot-vector multipliers that maintain full precision for the intermediate results. A first posit implementation in hardware delivers about 10Gpops and is measured to be about 1000x faster than the (Julia-based) software implementation. The design is being contributed to open source with a permissive license, and the expectation is that this work will not only enable further exploration of the posit format, but will also spur further hardware improvements and implementations.

ACKNOWLEDGMENTS

The authors wish to acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. Also, we thank John Gustafson for feedback on an initial version of our design, and we thank the anonymous reviewers for their constructive commentary.

REFERENCES

[1] John Gustafson and Isaac Yonemoto. 2017. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations* 4, 2 (2017). http://superfri.org/superfri/article/view/137

[2] Reinhard Kirchner and Ulrich Kulisch. 1988. Arithmetic for vector processors. In *Reliability in Computing*. Elsevier, 3–41.

[3] Jack Koenig, David Biancolin, Jonathan Bachrach, and Krste Asanovic. 2017. A Hardware Accelerator for Computing an Exact Dot Product. In *Computer Arithmetic (ARITH), 2017 IEEE 24th Symposium on*. IEEE, 114–121.

[4] Michael Muller, Christine Rub, and W Rulling. 1991. Exact accumulation of floating-point numbers. In *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*. IEEE, 64–69.

[5] J.W. Peltenburg, S. Ren, and Z. Al-Ars. 2016. Maximizing Systolic Array Efficiency to Accelerate the PairHMM Forward Algorithm. In *Proc. IEEE International Conference on Bioinformatics and Biomedicine*. Shenzhen, China, 758–762.

[6] Eric Quinnell, Earl E Swartzlander, and Carl Lemonds. 2007. Floating-point fused multiply-add architectures. In *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on*. IEEE, 331–337.

[7] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. 2015. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development* 59, 1 (2015), 7–1.

[8] Jeffrey Stuechely. 2016. A New Standard for High Performance Memory, Acceleration and Networks. (March 2016). http://opencapi.org/2017/04/opencapi-new-standard-high-performance-memory-acceleration-networks/