Floating-Point Matrix Product on FPGA

Faycal Bensaali
University of Hertfordshire
f.bensaali@herts.ac.uk

Abbes Amira
Brunel University
abbes.amira@brunel.ac.uk

Reza Sotudeh
University of Hertfordshire
r.sotudeh@herts.ac.uk

Abstract

The nature of some scientific computing applications involves performing complex tasks repeatedly on floating-point data, often under real-time requirements. Therefore, high performance systems are required by the developers for fast computations. Many researchers have begun to recognize the potential of reconfigurable hardware such as field-programable gate arrays in implementing floating-point arithmetic. In this paper a floating-point adder and multiplier are presented. The proposed cores are used as basic components for the implementation of a parallel floating-point matrix multiplier designed for 3D affine transformations. The cores have been implemented on recent FPGA devices. The performance in terms of area/speed of the proposed architectures has been assessed and has shown that they require less area and can be run with a higher frequency when compared with existing systems.

1. Introduction

Many scientific computing applications demand high numerical stability and accuracy and hence are usually floating-point based arithmetic. A close examination of the algorithms used in these applications reveals that many of the fundamental actions involve matrix operations such as matrix multiplication. However, these operations are of $O(N^3)$ on a sequential computer and $O(N^3/p)$ on a parallel system with p processors and hence are computer intensive for large size problems. Reconfigurable hardware devices in the form of Filed-Programmable Gate Arrays (FPGAs) have been proposed as viable system building blocks in the construction of high performance systems at an economical price. Given the importance and the use of Floating-Point Matrix Multiplication (FPMM) in scientific computing applications, it seems an ideal candidate to harness and exploit the advantages offered by FPGAs including flexibility and programmability. To increase their flexibility, recent FPGA devices provide new fixed circuit functions which may be employed in floating-point operations. Since they are already committed to the silicon die at manufacture time, there is no added cost in circuit area if a designer chooses to use them. Furthermore, a single FPGA device now contains enough logic blocks to hold multiple floating-point units that can perform computations concurrently. Thus, algorithms that exploit the inherent parallelism of matrix multiplication can be implemented on the device [1].

In this paper a single precision FP adder and multiplier are presented, which are the basic components used in the implementation of a proposed parallel FP matrix multiplier designed for 3D affine transformations. A range of algorithms such as modified Booth-encoder multiplication, techniques and design approaches have been used to efficiently implement the proposed cores in order to investigate the best performances obtained. The target hardware for the implementation and verification of the proposed architectures is Celoxica RC1000 PCI based FPGA development board equipped with a Xilinx XCV2000E Virtex FPGA [2]. Moreover, the developed architectures have been synthesized on the most recent FPGAs exploring the new available features on these devices.

The composition of the rest of the paper is as follow. Related work is presented is Section 2. Section 3 is concerned with a brief review on the IEEE 754 standard for binary FP arithmetic. In the next two Sections 4 and 5, the proposed FP adder and multiplier with their hardware implementation are described respectively. Section 6 is concerned with the proposed parallel FP matrix multiplier designed for 3D affine transformations and its hardware implementation. Section 7 concludes the paper.

2. Related Work

Researchers have investigated the use of FP arithmetic in FPGAs and quantified its cost, anticipating the time when FPGAs have sufficient density to support it. Work presented in [3, 4] investigated the use of custom FP formats in FPGAs. Other studies are concerned with translating FP to fixed-point [5] or optimizing the bit-widths of FP formats as an alternative. Compared to IEEE standards [6], these formats require significantly less area and run significantly

1-4244-1031-2/07/$25.00©2007 IEEE

faster. Customized formats enable significant speedups for certain applications, but many scientific applications depend on the dynamic range and high precision of IEEE single and double-precision FP to maintain numerical stability. A series of works [7, 8, 9] ensued, all considering only IEEE single-precision format, usually with no rounding capabilities except in [8], where the authors implement rounding to nearest. These works on IEEE FP found that single-precision implementations were feasible, but extremely slow. As FPGAs grow in capacity, a variety of work [10, 11] has demonstrated the growing feasibility of IEEE compliant, single precision FP arithmetic and other similarly complex FP formats on FPGAs. Only few researchers have studied FP matrix multiplication on FPGAs. Specifically, [8] studies several aspects of single precision FP matrix multiplication and compares it to a microprocessor. [9] studies as well the implementation of FP arithmetic and used matrix multiplication as an example application. More recently, [11, 12] considers both performance and power issues for double precision FP matrix multiplication.

3. The IEEE Standard FP Representation

Currently the most important FP representation for real numbers is defined by the IEEE standard 754 [6]. The IEEE FP representation makes use of an exponential notation to represent real numbers, in which any real number is decomposed into: (1) sign, (2) fraction and (3) an exponent, for the implicit binary base:

$$(-1)^s \quad 1.f \times 2^{E-bias} = (-1)^s \quad 1.f \times 2^e \qquad (1)$$

where s denotes the sign, f the fraction and E the biased exponent. The sign is represented by only one bit (0 for positive and 1 for negative numbers). The fraction represents a number less than 1, but the significand or mantissa m of the FP number is 1 plus the fraction part ($m = \{m_0.m_1m_2 \dots m_n\} = 1.f$), where m_i is the i_{th} bit of the mantissa. The FP number is "normalized" when m_0 is one. The exponent is a signed number represented using the bias method with a bias of $2^{k-1} - 1$, where k is the bit-width of the biased exponent. The term unbiased exponent (or just exponent) refers to the actual power to which 2 is to be raised [13]. The IEEE standard 754 has four different precision formats: single, single extended, double and double extended. The single and double precision formats are the most used. Single precision representation requires 32 bits, one bit for the sign, 23 bits for the mantissa and eight bits for the exponent, using the implicit binary base. Double precision in the standard makes use of 64 bits, 52 for the mantissa and 11 for the exponent. The mantissa is assumed to be normalized and only includes the bits in the fractional part f (the bit at the left of the decimal point is implicit and, normally, supposed to be 1). The single-precision biased exponents of 0 and 255 (0 and 2047 for double-precision) are used for the special values: Not a Number (NaN), Infinity and Denormalized numbers. For the denormalized numbers, because of the additional complexity and costs, this part of the standard is not commonly implemented in hardware.

3.1. Rounding and Normalizing

Subsequent to all arithmetic operations performed on the FP value, it is necessary to return it to the standard, normalized form, before presenting the result for storage or further processing. Normalizing a FP value refers to shifting right or left its mantissa until its Most Significant Bit (MSB) is 1, while incrementing or decrementing the exponent for every bit the mantissa was shifted. In this way, the FP value is brought into its normalized form.

During processing, mantissa bit-width is normally increased. Hence, to return the post-processing FP value into the correct format, its mantissa bit-width must be reduced to its original size. However, reduction of bit-width introduces the need for rounding, because the fraction part needs to be converted into a less precise representation.

The IEEE standard specifies four rounding modes: (1) Round to Zero (RZ); (2) Round to Nearest (RN);(3) Round to Positive Infinity ($+\infty$) (RPI); and (4) Round to Negative Infinity ($-\infty$). These modes are implemented by extending the relevant significands by three bits beyond their Least Significant Bit (LSB) L. These bits are referred to, from the most significant to the least significant, as "guard" (g), "round" (r) and "sticky" (st). The first two are normal extension bits, while the last one is the logical OR of all the bits that are lower than the r bit [13, 14]. There are two cases:

1. If the hight order bit of the significand is 0, it must be shifted one bit left including the extension bits.
2. If the hight order bit of the significand is 1, set $s = s \vee r$, $r = g$ and increment the exponent by one.

The precise rules for rounding depend on the rounding mode and are given in Table 1. In this table, blanks mean that truncation takes place and the bits of the significand are the actual result bits. If the condition listed is true the significand is incremented by one. Out this four, the most implemented one is the RN mode, as it minimizes the rounding error and it is the default mode in the IEEE standard. In the next sections, the proposed FP multiplier and adder are presented with their hardware implementation. In this paper, the implemented format is the IEEE single precision.

4. Floating-Point Multiplication

Let us consider three FP numbers X, Y and Z. Using the IEEE standard, the three numbers can be represented as

Table 1. Rules for rounding modes [13]

Rounding mode	Sign of the result > 0	Sign of the result < 0
$-\infty$		+1 if $r \vee s$
$+\infty$	+1 if $r \vee s$	
0		
Nearest	+1 if $r \wedge m_{msb}$ or $r \wedge s$	+1 if $r \wedge m_{msb}$ or $r \wedge s$

follows:

$$X = (-1)^{s_1} m_1 \times 2^{e_1}, \quad Y = (-1)^{s_2} m_2 \times 2^{e_2} \tag{2}$$

and $Z = (-1)^s m \times 2^e$
where $e_1 = E_1 - bias$, $e_2 = E_2 - bias$ and $e = E - bias$
The product $Z = X \times Y$ involves the following steps:

1. If one or both operands is equal to zero, return the result as zero, otherwise:
2. Compute the sign of the result: s_1 xor s_2
3. Compute the exponent of the result: $E = E_1 + E_2 - bias$
4. Compute m:
   - Multiply the mantissas: $m = m_1 \times m_2$
   - Round the result to the allowed number of mantissa bits
5. Normalize if needed, by shifting m right and incrementing E.

4.1. Proposed Floating-Point Multiplier

Figure 1 illustrates the block diagram of the proposed FP multiplier. The two input FP numbers are unpacked and checks are performed for zero, infinity or NaN operands. For now, we can assume that neither operand is zero, infinity or NaN. The mantissa multiplier is a 24-bit parallel multiplier which consists of two parts: an array and a final adder. The array uses the modified Booth's algorithm which is an effective algorithm to reduce the number of partial product to be added by factor of two. Therefore, in our case, the number of partial products is reduced from 24 to 12.

4.1.1 Modified Booth-Encoder Multiplication

Let m_1 and m_2 be the mantissas of the two FP numbers to be multiplied. $m = m_1 \times m_2$ is the mantissa of the multiplication result. Since the mantissas are positive numbers and have a constant word-length W, modified Booth encoding can be further improved. The unsigned representation of m_2 is as follows:

$$m_2 = \sum_{x=0}^{W-1} m_{2,x} \times 2^x \tag{3}$$

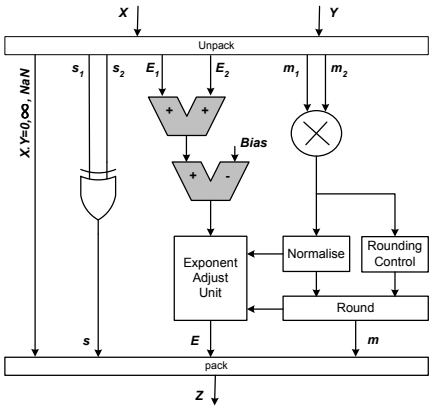

Figure 1. Block diagram for FP multiplier

Equation 3 can be rewritten as follows:

$$m_2 = \sum_{y=0}^{W/2-1} (m_{2,2y} + 2 \times m_{2,2y+1}) \times 2^{2y} \tag{4}$$

or:

$$m_2 = \sum_{y=0}^{W/2-1} (D_y) \times 4^y \tag{5}$$

where: $D_y = (m_{2,2y} + 2 \times m_{2,2y+1})$ and $D_y \in \{0, 1, 2, 3\}$
Using equation 5, the mantissas product m can be computed as follows:

$$m = \sum_{y=0}^{W/2-1} m_1 \times D_y \times 4^y = \sum_{y=0}^{W/2-1} PP_y \times 4^y \tag{6}$$

where: $PP_y = m_1 \times D_y$. Table 2 shows the possible values of PP_y and D_y.

Table 2. Possible values of D_y and PP_y

$\mathbf{m_{2,2y+1}}$	$\mathbf{m_{2,2y}}$	$\mathbf{D_y}$	$\mathbf{PP_y}$
0	0	0	0
0	1	1	m_1
1	0	2	$2 \times m_1$
1	1	3	$3 \times m_1$

By using the unsigned representation for the mantissas, the three following points, which are part of the standard modified both algorithm, have been avoided:

1. Extend the sign bit to ensure that the word-length of the multiplicand is even;
2. Append a 0 to the right of the LSB of the multiplicand; and
3. Compute the two's complement of the multiplier.

4.2. Hardware Implementation and Results

The proposed FP multiplier shown in Figure 1 has been implemented using Handel-C [15] on the RC1000 board which is a standard PCI bus card equipped with a Xilinx XCV2000E Virtex-E FPGA. It has 8 MBytes of SRAM directly connected to the FPGA in four 32-bit wide memory banks. All are accessible by the FPGA and any device on the PCI bus [2]. The mantissas multiplier has been implemented using two different approaches: (i) Modified Both algorithm described in the previous section with a parallel adder for partial products accumulation. (ii) A parallel pipelined integer multiplier. Figure 2 shows the different components used in the first approach.

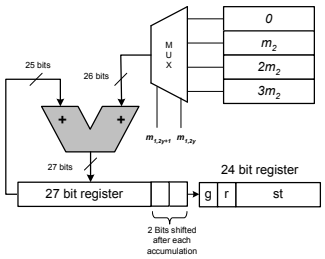

Figure 2. Mantissa multiplier

The four possible values of D_y are computed and stored in an array of registers. For each combination $(m_{2,2y}, m_{2,2y+1})$, the corresponding D_y value is used as an input for the parallel adder from Xilinx CoreGen [16]. The output from the adder is stored in a 27-bit register. Its two LSBs are copied to the two Most Significant Bits (MSB) of a 24-bit register which is shifted right by two after each accumulation. The final result is stored in both registers. The 24-bit register is used during the rounding process. The value $2 \times m_1$ is computed by simply performing a right shift, while $3 \times m_1$ is calculated by the same adder used for the partial products accumulation. For the second approach, a 24-bit parallel pipelined multiplier from Xilinx CoreGen is used to perform the multiplication of the two input mantissas. The remaining components of the proposed multiplier are common for both approaches. The sign of the result s is easily determined by XORing s_1 and s_2. As the first step of the calculation of the result exponent E, the sum $E_1 + E_2$ is performed using a parallel adder from the CoreGen. The excess bias is removed from $E_1 + E_2$ using the same adder. This is performed by giving as input the two's complement of the bias (in our case 129). The rounding mode used in our implementation is the RN mode. The rules listed in Table 1 are tested by the rounding control unit.

On the XCV2000E (bg560-6) FPGA, the proposed FP multiplier based on the first approach consumes 2% of the available FPGA area and runs with a maximum frequency of 60 MHz, while the implementation based on the second approach consumes as well 2% of the total available area and operates at a maximum clock frequency of 104 MHz.

The proposed FP multiplier has also been implemented on the XC2VP125 Virtex-II Pro FPGA in order to employ the available (18×18 bits) embedded multipliers. Table 3 illustrates the performances obtained in terms of area consumed and speed which can be achieved.

Table 3. Area/Speed implementation report

Approach used	Latency	Area		Speed (MHz)
		Logic (%)	18×18 Mults	
Booth encoding	15	1	-	120
CoreGen	7	1	4	231
Multiplier in [11]	-	1	-	220

In [11], 32-bit, 48-bit and 64-bit pipelined FP adders and multipliers were designed. The last row of Table 3 illustrates the implementation result of the single-precision FP multiplier which has been implemented on the XC2VP125 Virtex-II Pro FPGA. Our multiplier based on the Xilinx CoreGen approach exhibits better results in term of the maximum running frequency. The advantage of the other approach is that it requires less area and it can be used for some applications where the speed is not a big issue. The computation time of a design where this FP multiplier is used can be increased by increasing the number of PEs.

5. Floating-Point Addition

The main steps in the calculation of the sum $Z = X + Y$ are as follows.

1. If $E_1 < E_2$ swap the operands. This ensures that the exponents satisfie $d = E_1 - E_2$. Tentatively set the exponent of the result to E_1.
2. If $s_1 \neq s_2$, replace m_2 by its two's complement.
3. Shift m_2 d places to the right (shifting in 1's if m_2 was complemented in the previous step). From the bits shifted out, set g to the MSB, r to the next MSB, and set st to the OR of the rest.
4. Compute a preliminary significand $m = m_1 - m_2$. If $s_1 \neq s_2$, the MSB of m is 1 and there was no carry-out, then m is negative. Replace m with its two's complement. This can only happen when $d = 0$.
5. Shift m as follows. If $s_1 = s_2$ and there was a carry-out in step 4, shift m right by one, filling in the high-order position with 1 (the carry-out). Otherwise, shift it left until it is normalized. When left shifting, on the first

shift fill in the low-order position with the g bit. After that, shift in zeros. Adjust the exponent of the result accordingly.

6. Adjust s and st. If m was shifted right in step 5, set $r =$ low-order bit of m before shifting and $st = g \vee r \vee st$. If there was no shift, set $r = g$ and $st = r \vee st$. If there was a single left shift, do not change r and st. If there were two or more left shifts, $r = 0$ and $st = 0$ (in the last case, two or more sifts can only happen when $s_1 \neq s_2$ and $E_1 = E_2$).
7. Round m using the rules in Table 1.
8. Compute the sign of the result. If $s_1 = s_2$, the result has the same sign. If $s_1 \neq s_2$, then the sign of the results depends on: i) s_1 and s_2; ii) whether there was a swap in step 1; and iii) whether m was replaced by its two's complement in step 4. Table 4 shows the different possible cases.

Table 4. Sign computation when $s_1 \neq s_2$

Swap	Complement	s_1	s_2	s
Yes		+	-	-
Yes		-	+	+
No	No	+	-	+
No	No	-	+	-
No	Yes	+	-	-
No	Yes	-	+	+

5.1. Proposed Floating-Point Adder

Figure 3 illustrates the block diagram of the proposed FP adder.

First, the two input FP numbers are unpacked and checks are performed for zero, infinity or NaN operands. For now, we can assume that neither operand is zero, infinity or NaN.

The exponents are compared by an 8-bit comparator and if $E_1 < E_2$, a signal and the difference $d = E_1 - E_2$ are sent to the Swap and Align Unit (SAU) where the mantissas are swaped. This ensures the difference between the two exponents to be a positive value. The sign of the result is computed by the Sign Unit (SU) according to the cases listed in Table 4. If the signs of the operands are different, a signal is sent by the SU to the SAU where the mantissa m_2 is two's complemented then shifted d positions to the right, aligning the binary point. The first two bits shifted out are set to g and r and st is set to the OR of the rest. For adding the two mantissas, a 24-bit adder is sufficient, as long as the first discarded bit (round) and the OR of the rest of the bits (sticky) are kept. If the result of the addition is negative, it is replaced by its two's complement. In the Normalize Unit (NU), the

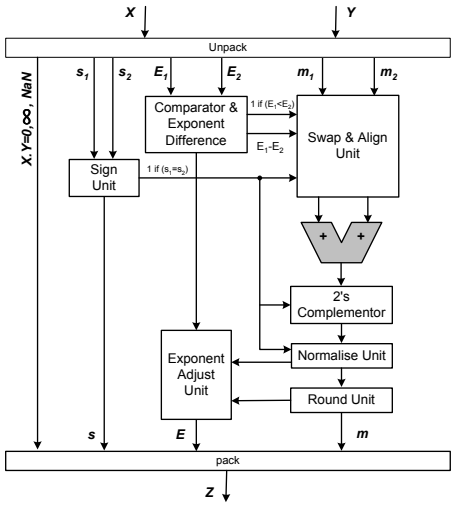

Figure 3. Block diagram for FP adder

results is shifted to the left until it is normalized. The exponent is adjusted according to the number of positions shifted. Finally, the result is rounded by the Round Unit (RU). The rounding mode used is the RN, which adds a one to the LSB of the result according to the rules in Table 1.

5.2. Hardware Implementation and Results

Likewise the FP multiplier, The proposed FP adder shown in Figure 3 has been implemented using Handel-C on the RC1000 board. The following Xilinx CoreGen Cores have been used for the implementation of the different units in the proposed FP adder:

- 24-bit parallel integer adder for mantissas addition;
- 8-bit comparator for the two input exponents comparison;
- 8-bit parallel integer subtractor. This component is used for the computation of the two input exponents difference and the incrementation or decrementation of the result's exponent; and
- 24×1-bit adder with the 1-bit input set to 1. This component is used for the two's complement computation.

The rest of the design logic (the sign computation, normalizing and rounding, etc) has been implemented using pure Handel-C coding. The proposed FP adder consumes 5% of the total available area on the XCV2000E FPGA and runs with a maximum frequency of 55 MHz. It has been also implemented on the XC2VP125 Virtex-II Pro FPGAs in order to use it with the FP multiplier in the next section where a FP matrix multiplier is presented. Table 5 illustrates the performances obtained in terms of area consumed and speed which can be achieved. The proposed adder gives better results in term of the maximum running frequency in comparison with the FP adder in [11].

Table 5. Area/Speed implementation report

FP Adder	Latency	Area (%)	Speed (MHz)
Proposed	12	2	236
Adder in [11]	-	2	220

6. Floating-Point Matrix Multiplication for 3D Affine Transformations

In this section, the proposed FP adder and multiplier cores, presented in the previous sections, are used as basic components for the implementation of a parallel FPMM designed for 3D affine transformations.

In computer graphics the most popular method for representing an object is the polygon mesh model. In a simplest case, a polygon mesh is a structure that consists of polygons represented by a list of (x, y, z) coordinates that are the polygon vertices. Thus the information we store to describe an object is finally a list of points or vertices [17] (see Figure 4).

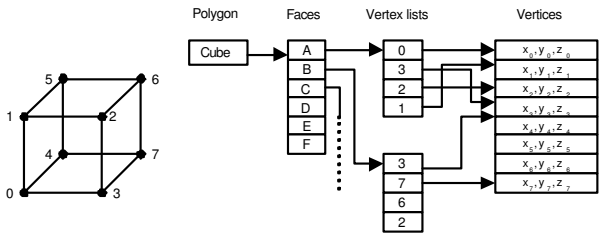

Figure 4. Data structure for object representation

3D affine transformations are the transformations that involve rotation, scaling, shear and translation. A matrix can represent an affine transformation and a set of affine transformations can be combined into a single overall affine transformation [17]. Using matrix notation, a Vertex V is transformed to V^* (* denotes the transformed vertex) under translation, scaling and rotation, which are the most commonly used transformations in computer graphics, as:

$$V^* = T \times V \tag{7}$$

$$\begin{pmatrix} x^* \\ y^* \\ z^* \\ 1 \end{pmatrix} = \begin{pmatrix} A & D & G & J \\ B & E & H & K \\ C & F & I & L \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{8}$$

Consider an object represented with N vertices. The New Position (NP) of the object when applying a transformation can be calculated as follows [18]:

$$NP = T \times OP \tag{9}$$

where T is the matrix transform, OP is a $(4, N)$ matrix contains the Old vertices Position and NP is a $(4, N)$ matrix contains the New vertices Position.

$$\begin{pmatrix} x_0^* & x_1^* & \dots & x_{N-1}^* \\ y_0^* & y_1^* & \dots & y_{N-1}^* \\ z_0^* & z_1^* & \dots & z_{N-1}^* \\ 1 & 1 & \dots & 1 \end{pmatrix} = T \times \begin{pmatrix} x_0 & x_1 & \dots & x_{N-1} \\ y_0 & y_1 & \dots & y_{N-1} \\ z_0 & z_1 & \dots & z_{N-1} \\ 1 & 1 & \dots & 1 \end{pmatrix} \tag{10}$$

6.1. Proposed Architecture

Figure 5 shows the proposed parallel FPMM architecture. The multiplier consists of four identical Processor Elements

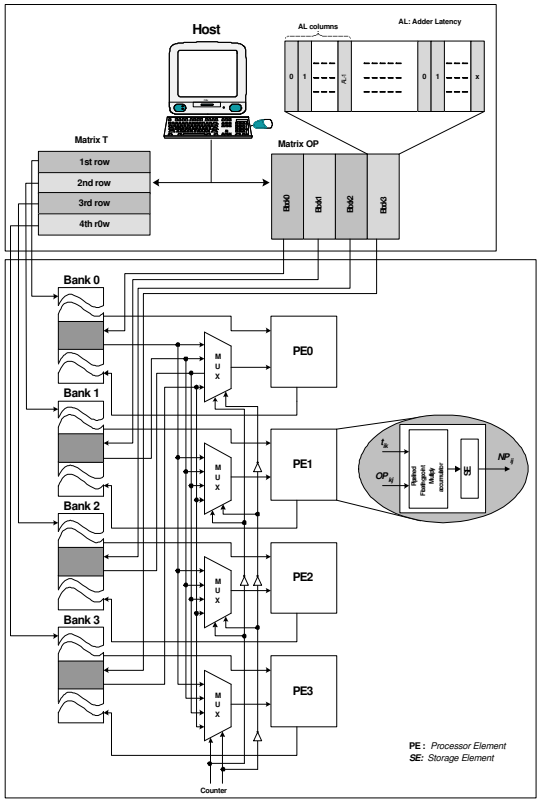

Figure 5. The proposed parallel FPMM architecture for 3D affine transformations

(PEs). Each PE comprises a pipelined FP Multiply ACcumulator (MAC) and a register for final result storage. The vertices coordinates are represented using the IEEE single-precision real numbers. The MAC has been implemented using two different approaches:

- The pipelined FP library from Celoxica [19]. It is a platform-independent core. It allows the programmer to perform FP operations in a pipelined manner on single-precision FP numbers.

- The proposed FP adder and multiplier described in the previous sections. The FP multiplier used is the Xilinx CoreGen based approach.

For both approaches, the FP adders and multipliers used are pipelined and have different latencies. The FP Adder and Multiplier Latencies from Celoxica's library are $AL = 10$ and $ML = 7$ respectively. The input transform matrix T is partitioned into four rowwise blocks, which gives one row per block. Each block is stored in one of the four available banks. The matrix OP is partitioned into four columnwise blocks, likewise matrix T each block is stored in one of the banks. In addition, due to AL value, each block of the matrix OP is partitioned into columnwise sub-blocks. each sub-block contains AL columns and the last one is padded with columns of zeros if $N \ mod \ AL \ \neq 0$ (N is the number of vertices).

Figure 6 illustrates the timing diagram when performing a multiplication of one row of the transform matrix T with one sub-block of the matrix OP as shown in equation 11 in the case of $AL = 10$:

$$\begin{pmatrix} NP_{i0} & NP_{i1} & \dots & NP_{i9} \end{pmatrix} = \begin{pmatrix} t_{i0} & t_{i1} & t_{i2} & t_{i3} \end{pmatrix} \times \begin{pmatrix} OP_{00} & OP_{01} & \dots & OP_{09} \\ OP_{10} & OP_{11} & \dots & OP_{19} \\ OP_{20} & OP_{21} & \dots & OP_{29} \\ OP_{30} & OP_{31} & \dots & OP_{39} \end{pmatrix} \quad (11)$$

6.2. Results and Analysis

The proposed architecture has been implemented and tested on the RC1000 prototyping board. The implementation based Celoxica's FP library consumes 99% of the available FPGA area and runs with a maximum frequency of 50 MHz, while the implementation based on the proposed FP adder and multiplier consumes 70% of the target FPGA area and can be run at a maximum clock frequency of 85 MHz. The parallel FPMM architecture has been synthesized on Virtex-II Pro FPGA in order to exploit the additional features and resources available on this device. Table 6 illustrates the performance obtained for the proposed architectures when using the two different design approaches for the MAC implementation.

Table 6. Area/Speed implementation report

Mac used	Area		Speed (MHz)
	Logic (%)	18×18 Mults	
Proposed FP Adder/Multiplier	39	16	215
Celoxica Pipelined FP library	43	64	119

Results obtained show that the parallel FPMM based on our proposed FP addition and multiplication cores gives better performance in comparison with the one based on the pipelined FP library from Celoxica. This mainly, because of the suitability of the different components used in our implementation for the targeted FPGA device. It is worth noting that, due to the application requirements, the sizes of the manipulated matrices T and OP are (4×4) and $(4 \times N)$ respectively. The maximum value of N depends only on the available off-chip storage resources. On the RC1000 board, N can be any value up to 2^{18}. Other sizes can be performed using the proposed architecture by applying a different partitioning strategy on the matrices T and OP at the host level.

In [11], which is the most recent work concerning 32-bit FP matrix multiplication, two implementations for FPMM are presented. The results presented are only the one obtained for 64-bit input data. Therefore, a fair comparison can not be made with our implementation. The work presented in [9], studied the implementation of FP arithmetic and used matrix multiplication as an example application. Results obtained show that a 96×96 FPMM, implemented on a Virtex XC4044XL FPGA device, can achieve a maximum running frequency of 50 MHz. In [8], the authors investigated the influence of the FP MACs on the performance of a matrix multiplication algorithm. Results obtained, show that the maximum and best running frequency for a 1024×1024 FP matrix multiplier is 33 MHz on the Annapolis MicroSystems board.

7. Conclusion

The abundant hardware resources on current FPGAs provide new opportunities to improve the performance of hardware implementations of scientific computations requiring FP arithmetic. In this paper, a FP adder and multiplier have been developed and tested on the RC1000 prototyping board. The proposed cores have been used as basic components for the implementation of a parallel FPMM designed for 3D affine transformations. Results obtained demonstrate the suitability of recent FPGAs for applications based on FP arithmetic. Our implementations have achieved superior FP performance over state-of-the-art FPGA-based implementations and has shown significant speed-up compared to software based implementation.

References

[1] F. Bensaali, "Accelerating Matrix Product on Reconfigurable Hardware for Image Processing Applications," *PhD thesis,* The Queen's University of Belfast, 2005.

[2] Datasheet, "RC1000 Development Platform Product Brief," v1.1, Celoxica Ltd., August 2002.

[3] P. Belanovic and M. Leeser, "A Library of Parameterized Floating-Point Modules and Their Use," *Proceed-*

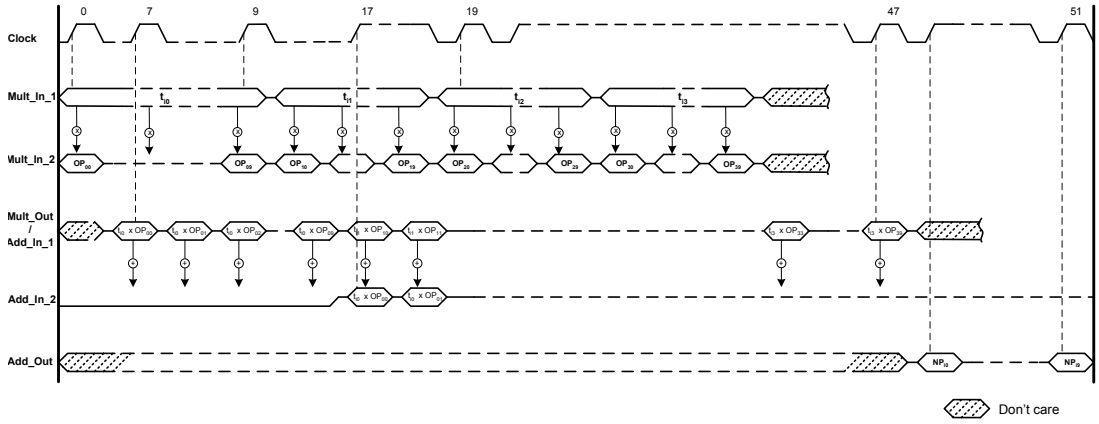

Figure 6. Timing diagram when performing a multiplication of one row of the transform matrix T with one sub-block of the matrix OP

ings of the 12th International Conference on Field Programmable Logic and Application, pp. 657-666, Montpellier, France, September 2002.

[4] J. Dido et al., "A Flexible Floating-Point Format for Optimizing Data-Paths and Operators in FPGA Based DSPs," *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays,* pp. 50-55, CA, February 2002.

[5] M. P. Leong et al., "Automatic Floating to Fixed Point Translation and its Application to Post-Rendering 3D Warping," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines,* pp. 240-248, Napa, California, April 1999.

[6] "IEEE Stndard for Binary Floating-Point Arithmetic," *ANSI/IEEE Std 754-1985,*NY, USA, August 1985.

[7] L. Louca et al., "Implementation of IEEE single precision floating point addition and multiplication on FPGAs," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines,* pp. 107116, Napa, California, April 1996.

[8] W. B. Ligon III et al., "A Re-Evaluation of the Practicality of Floating-Point Operations on FPGAs," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines,* pp. 206-215, Napa, California, April 1998.

[9] I. Sahin et al. "Feasibility of Floating-Point Arithmetic in Reconfigurable Computing Systems," *Proceedings of the 3rd Military and Aerospace Applications of Programmable Devices and Technology Conference,* Maryland, USA, September 2000.

[10] J. Liang et al. "Floating Point Unit Generation and Evaluation for FPGAs," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines,* pp. 185-194, Napa, California, April 2003.

[11] L. Zhuo and V. K. Prasanna, "Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs," *in Proceedings of the 18th International Parallel and Distributed Processing Symposium,* pp. 94-103, New Mexico, USA, April 2004.

[12] G. Govindu et al., "Analysis of High Performance Floating-Point Arithmetic on FPGAs," *Proceedings of the 18th International Parallel and Distributed Processing Symposium,* pp. 316-323, New Mexico, April 2004.

[13] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach," *Third Edition, Morgan Kaufmann Publisher,* 2003.

[14] S. Paschalakis and P. Lee "Double Precision Floating-Point Arithmetic on FPGAs," *Proceedings of the IEEE International Conference on Field-Programmable Technology,* pp. 352-358, Japan, December 2003.

[15] Manual , "Handel-C Language Reference Manual," RM-1003-4.2, Celoxica Ltd., 2004

[16] Application Note, "Xilinx CoreGen and Handel-C," AN 58 v1.0, 2001.

[17] A. Watt, "3D Computer Graphics," Addison-Wesley, 2000.

[18] F. Bensaali et al., "Accelerating Matrix Product on Reconfigurable Hardware for Image Processing Applications," *IEE Proceedings on Circuits, Devices and Systems,* Vol. 152, Issue 3, pp 236-246, June 2005.

[19] Manual, "Pipelined Floating-Point Library," *Celoxica Ltd.,* 2004.