Scientific Research and Essays Vol. 5 (20), pp. 3070-3081, 18 October, 2010
Available online at http://www.academicjournals.org/SRE
ISSN 1992-2248 ©2010 Academic Journals

Full Length Research Paper

A 32-bit floating-point module design for 3D graphic transformations

Ibrahim Sahin

Department of Electronics and Computer Education, Faculty of Technical Education, Duzce University 81620, Duzce, Turkey. E-mail: ibrahimsahin@duzce.edu.tr.

Accepted 2 September, 2010

Nowadays, in computer animations, tens of, even hundreds of animation objects are placed in a scene to form a typical animation scene and thousands of vertices are used to mathematically define each object in the scene. Applying three dimensional (3D) transformations to such scenes requires huge amount of CPU time. As a result, calculation of an animation scene could take a long time. Moreover, in the case of real time animations, it becomes almost impossible to calculate transformations on time. In this presented work, a 32-bit floating-point based hardware module was designed to speed-up 3D graphic transformations using field programmable gate array (FPGA) chips. The module was tested and functional verification of the module was done by comparing the results produced by the module to the results generated by general purpose computers (PCs) for the same set of input data. Module's data processing speed was compared to various PCs. The results showed that, 3D graphic transformations can be speeded-up by a factor (up to 11.47) employing the designed module.

Key words: Field programmable gate array, computer graphics, three dimensional transformation, hardware module.

INTRODUCTION

Three dimensional (3D) graphic transformations are a vital part of graphics software libraries and graphic application programs. When these transformations are applied to complex graphic objects in which a few thousand vertices are included to define the objects, transformation functions use huge amount of the CPU resources of the computing environment. Especially in computer animations, as the number of objects in the animation scene increases and the number of vertices used to define these objects increases, calculation of the animation takes hours. In a typical animation scene of an animated movie, tens of objects can be used to describe a scene and over 100 thousand vertices can be used to describe these objects (Hearn and Baker, 2004). Such an animation scene has to be recalculated 24 times to create one second movie in DVD standard. This means that locations of the 100 thousand vertices used the describe objects on the scene have to be recalculated 24 times to create a one second movie clip (Disney/Pixar, 2008). In 3D animation, multiplications of a 4x4 matrix and a 4x1 matrix is required to calculate a new location of one vertex. A total of 38.4 million multiplications and 28.8 million additions have to be done to calculate such an animation scene for one second. Some other calculations are also need to complete animation such as rendering calculations. General purpose computers become insufficient for such an animation work and the calculations take huge amount of time.

As a solution to the problem, several approaches have been developed. Some of these approaches are using enhanced graphics cards designed specifically for computer graphics, using specially designed computers for computer graphics (Silicon Graphics International, 2007), and using super or parallel processor computers. All of these solutions are of some disadvantages. Enhanced graphics cards are designed as application specific integrated circuit (ASIC). Any error done at the design stage of these cards cannot be recovered once the card is manufactured and redesigning the card and having it ready for manufacturing takes about two months. Moreover, once these cards are manufactured, they can only perform the functionality that the card was designed for. The card hardware cannot be reconfigured to perform additional functionality. Specially designed computers, super computers, and parallel computers are costly solutions and sometimes desired performance gain cannot be obtained. Field programmable gate array (FPGA) based solutions are cost effective alternatives to the above solutions.

In this work, as a cost effective alternative to the above mentioned counterparts, a new hardware module was designed to speed-up 3D graphic transformations. The module was designed to run on FPGA devices and can operate on 32-bit floating-point numbers. It was simulated using real test data and correctness of the results produced by the module was verified. The same test data was processed using a C++ software running on three different general purpose computers. Processing speed of the module was compared to the C++ software. The comparison results showed that considerable amount of speed-ups can be achieved when the module is employed in 3D graphic transformations.

BACKGROUND

Related works

In the literature, it is possible to see several general purpose matrix multiplication module designs for FPGAs. For example, in a research work, Boullis and Tisserand presented a new algorithm for the problem of multiplication by constant matrices. They claimed that compared to the best previous results, their solution leads to a significant drop in the total number of additions/subtractions, up to 40%. They also implemented a very high speed integra-ted circuit hardware description language (VHDL) generator to generate a circuit design for their algorithm. They extended their algorithm and generator to cover some digital filters and they are now able to handle filters involving a multiplication by constant matrix and delay operations (such as FIR filters). They claimed that in the case of a 26-tap 16-bit FIR filter; a 34% reduction of the operation count was achieved, compared to recent results (Boullis and Tisserand, 2003, 2005).

In another research work, Gloster et al. designed a pipelined multiply and accumulate (MAC) unit to speed-up matrix multiplication. The MAC unit is able to multiply individual elements of input matrices' given row and given column. The accumulator inside the MAC unit continuously adds the results produced by the multiplier to the current sum. When one row and one column is processed with the MAC unit one element of the result matrix is calculated. The claimed that with MAC module matrix multiplication can be speeded-up up to ten times compared to PCs and even more speed-ups can be achieved when multiple MAC units are organized to process different rows and column at the same time (Gloster and Sahin, 2001).

El-Atfy et al. presented a new architecture for fixed-point matrix multiplication using Xilinx Virtex4 device. Their architecture utilizes the hardware resources on the entire FPGA and uses the DSP blocks inside the FPGA devices. The architecture can be implemented for non-square matrix multiplication. They claimed that the proposed implementation shows improvement in area and latency compared to recent published work. They achieved an improvement by over 50% in FMAX and 20% in area using new FPGAs (El-Atfy et al., 2007).

In 3D graphic transformations small matrices are multi-plied. Since the matrix multipliers mentioned above were usually designed to multiply huge matrices, they do not give desired performance in terms of calculation time when they are employed in 3D graphic transformations.

One specific study was conducted by Dr. A. Amira et al from Brunel University. They investigated the suitability of FPGA devices as a low cost solution for implementing 3D affine transformations. They implemented their proposed solution on a RC1000-PP Celoxica board based develop-ment platform using Handel-C and reported the implementation results. According to the results, they can achieve up to 35 MHz clock speed. They also compared their implementation with RADEON FSC 32 MB graphics card in terms of data processing speed. Although, their implementation did not outperform the graphics card, they showed that 3D affine transformation can be done using FPGAs for 22 bit fixed-point data (Bensaali et al., 2003).

Another particular work for 3D transformations was done by Franchini et al. They introduced a new coprocessor architecture called CliffoSor which was designed to support Clifford Algebra. They implemented the coprocessor on a FPGA device. Initial test results showed that they were able to speed-up 3D transfor-mation from 4x to 20x compared to GAIGEN, a standard geometric algebra library generator for general-purpose processors (Franchini et al., 2009).

FPGA chips

FPGAs are type of chips that are completely prefabri-cated and contain special features for customization (Villasenor and Hutchings, 1998; John and Smith, 1997; Tessier and Burleson, 1998). The user of these chips can implement digital circuit designs by configuring them. The biggest advantage of these chips is their configuration time. Since the configuration time of these chips is very small (for some chips the configuration time is less than a millisecond), circuit designs can be realized very quickly compared to ASIC implementations. A typical circuit development cycle for an FPGA device includes four steps. These steps are designing the circuit, coding the design in a hardware description language (HDL), compiling the HDL code to a configuration file and loading the configuration to the chip.

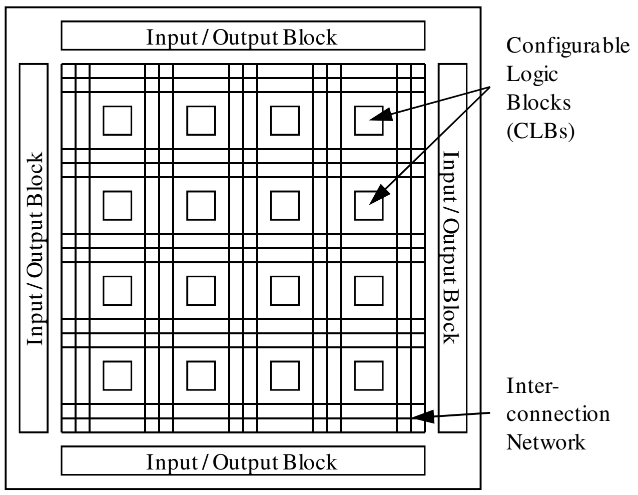


Figure 1. Structure of the Xilinx 4000 series FPGA chips (Sahin, 2002).

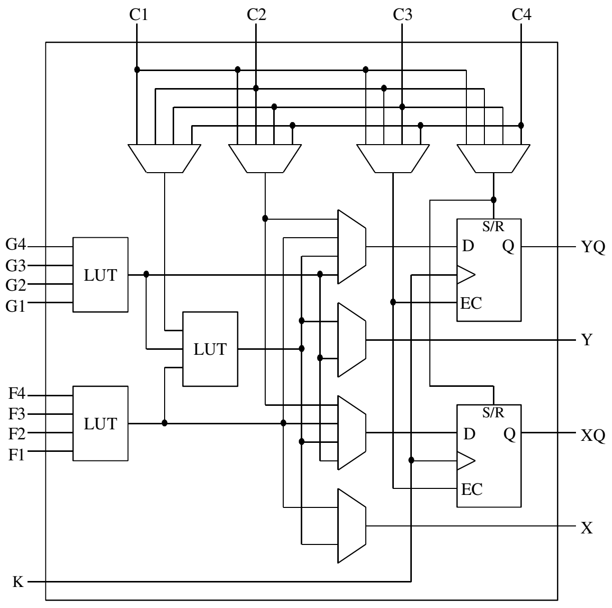


Figure 2. CLB block diagram of Xilinx 4000 series FPGA chips.

A typical FPGA device contains three configurable parts (Hauck, 1998; Brown and Rose, 2002). These parts are an array of logic cells called configurable logic blocks (CLBs), a programmable interconnection network and programmable input/output blocks. Figure 1 shows the structure of the Xilinx 4000 series FPGA devices. Each I/O block includes a number of I/O cells. These cells provide the interface between the package pins and internal signal lines of the FPGA chip. Each cell can be configured as an input, output, or bidirectional port. The interconnection network consists of switch boxes and metal wires. The CLBs are connected together by configuring the switch boxes in the interconnection network. Two most commonly used interconnection network types are island style and cellular style. In island style networks, point-to-point communications between the CLBs are possible. On the other hand, the cellular style network provides only local communication between the CLBs (Figure 1).

The CLBs are the most important parts of the FPGA device. Each FPGA manufacturer implements a different type of CLB. In this work, we briefly introduce the structure of CLBs for the Xilinx series FPGA chips. Figure 2 shows the block diagram of the CLB used in Xilinx 4000 series FPGA chips (Xilinx Inc, 1994; Vcc, 2002). This CLB includes three lookup tables (LUT), two programmable flip-flops and several programmable multiplexers. The LUTs are function generators, capable of implementing any combinational logic function of their inputs. The LUTs in Figure 2 can perform any function of up to five inputs when they are combined. SRAM controlled multiplexers are used to route signals within the CLB. The flip-flops are used to register output signals when required.

In Xilinx's FPGA chips, (the Virtex-II Pro), each CLB comprises four similar slices (Xilinx Inc, 2002). The slices are connected together with a local feedback box. The four slices in the CLB are split into two columns. Each slide includes two four-input function generators, arithmetic logic gates, carry logic, function multiplexers and data storage elements.

FPGA based custom computing machines

FPGA Based Custom Computing Machines (FCCMs), also known as reconfigurable computer (RC), are combination of hardware/software data processing platforms that include a general purpose processor and one or more FPGA devices. As shown in Figure 3, in FCCMs, one or more FPGA chips with their local memory units are organized on a printed circuit board (PCB) and they are attached to a host computer as a coprocessor through PCI bus. Some of the most famous FPGA boards are SPLASH-2 (Buell et al., 1996; Ratha and Jain, 1999; Ratha et al., 2000) and DECPeRLe (Vuillemin et al., 1996; Lewis et al., 1999; Perkowski et al., 1999). The SPLASH-2 board includes a linear array of Xilinx 4010 FPGA chips. Sixteen FPGA chips are used on the board and they are organized in a linear systolic array. One additional FPGA is used for control purposes. Each FPGA has a limited 36-bit connection to its two nearest neighbor chips. A 512 KByte local memory is also attached to each FPGA. Several SPLASH boards can be connected to form a chain and up to 16 boards can be connected together to form a 256-element linear systolic array. The DECPeRLe-I board includes 23 Xilinx 3090 FPGAs. Sixteen FPGAs were used to form a 4 x 4 array and the remaining chips were used for interfacing with

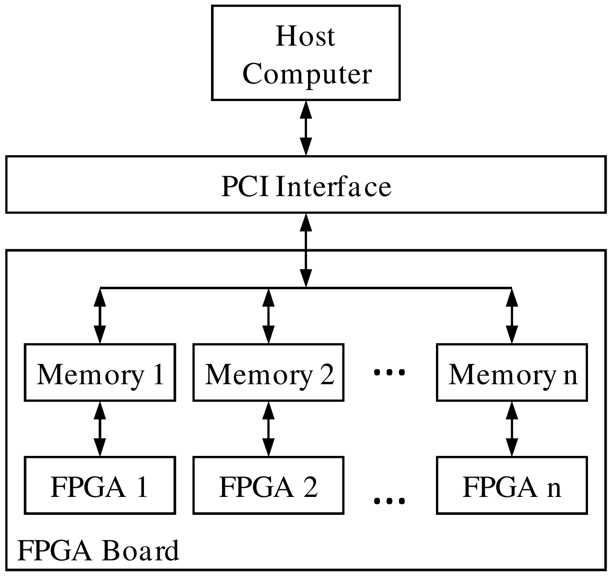


Figure 3. General structure of an FCCM.

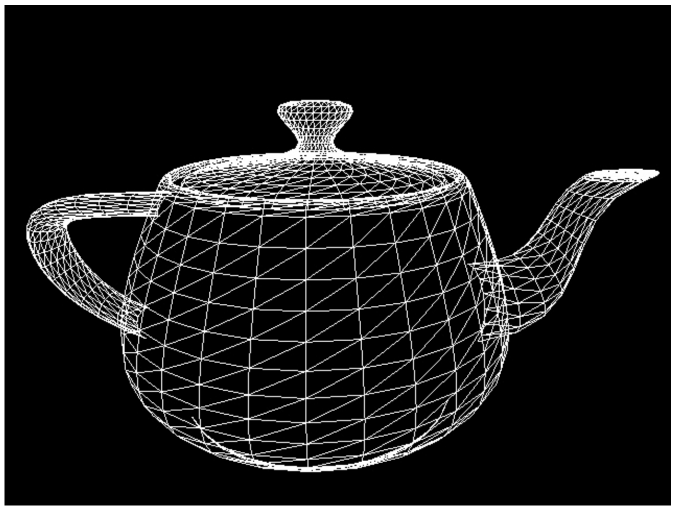


Figure 4. Sample 3D object definition (Lin, 2007).

the RAM and the host computer.

FCCMs combine the flexibility of general purpose processors with the speed of application specific processors. Usually the general purpose processor acts as the host processor and the reconfigurable hardware components are used as a coprocessor. In a typical FCCM, computationally intensive portions of algorithms are executed on FPGA devices for enhanced performance. A well designed and utilized FCCM could yield 10x to 1000x improvement in execution time over conventional general purpose processor based "software only" computers.

It has been shown that executing computationally complex sections of applications on RC systems significantly reduces the execution time of the applications compared to the general purpose processor only systems (DeHon and Wawrzynek, 1999). However, applications must be mapped to FPGA devices before they can be executed on these systems. The mapping processes can be performed either manually or automatically using software tools. Several applications were mapped to RC systems manually including image processing algorithms (Prada et al., 1999; Tavares et al., 1998; Figueiredo and Gloster, 1998; Figueiredo et al., 2000), genetic optimization algorithms (Graham and Nelson, 1996), and pattern recognition (Hogl et al., 1995).

How is computation done in FCCMs?

The computation on FPGA chips is done in four stages. First, the FPGA chips are configured by host computer with specially designed hardware modules that can execute the computationally complex sections of the algorithms. Second, data to be processed is transferred from host computers memory to the local memory units of the FPGAs. Third, the module configurations are enabled to process given data. Module configurations process given data and store results back to the local memory. Fourth, the results are collected by the host computer from the FPGA chips' local memories. Data transfer between the host computer and local memory units is done using direct memory access (DMA) technique.

Computer graphics and 3D transformations

Several software suites, graphics libraries and application programming interfaces (API) have been developed for graphic design and graphic animation purposes. Two of the well known graphics packages are open graphics library (OpenGL) and DirectX. These packages include several functions for creating computer graphics. It is possible to access functions provided by these graphics packages through most programming languages such as C/C++, C#, Java and Visual Basic.

The first step of creating animation using these packages is to form 2D or 3D mathematical models of animation objects using vertices, edges and surfaces. Modeling even a simple animation object requires to define hundreds even thousands of vertices, edges, and surfaces. Figure 4 shows a sample animation model of famous Utah Teapot.

Geometric transformations are an unavoidable part of the graphics packages. While generating animations, several 2D or 3D geometric transformations are perform on the mathematical models of the animation objects. There basic transformations are translation, rotation and scaling. While in some cases only one transformation is required, in most cases combination two or more transformation is applied to the object to create animation effects.

When using three dimensional cartesian coordinate system, the animation objects and scene are defined with three coordinate values (*x, y, z*). In cartesian coordinate system, 3D rotation or scaling operations of a single vertex requires multiplication of a 3x3 matrix and a 3x1 matrix while translation requires addition of a 3x3 matrix and a 3x1 matrix.

Most of the time more than one transformations have to be applied to objects to obtain desired results. In such a case, combining all transformations in to one transformation matrix and then applying it to the objects is the desired solution. On the other hand, translation operation is not a linear operation and cannot be calculated through matrix multiplication. Moreover, it cannot be combined with other transformations.

Homogenous coordinate representation of the objects is used to standardize all geometric transformations. In this representation, all transformations, applied to a single vertex, require multiplication of a 4x4 matrix and a 4x1 matrix. Homogenous representation also helps to combine more than one transformation in to one transformation matrix.

While converting vertices defined in 3D cartesian coordinate system (*x, y, z*) to homogeneous coordinate system, a fourth coordinate value, *w*, is added to the vertex and the vertex is defined as (*x, y, z, w*), ($w \neq 0$ should be satisfied). Usually $w = 1$ is selected and different *w* values cause scaling of the object while converting to homogeneous coordinate system (Figure 4). Below translation, rotation and scaling operations are given in parametric and matrix multiplication forms in Equations (1, 2, 3).

In translation operation, t_x, t_y, and t_z parameters define the amount of move of the object in each dimension, in rotation operation, θ parameter defines the rotation angle, and in scaling operation s_x, s_y, and s_z parameters define the scaling factors in each directions. *P* represents the original coordinate of the vertex and *P'* is the new coordinate of the vertex (Hearn and Baker, 2004).

$$P' = T(t_x, t_y, t_z) \cdot P$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{1}$$

Translation

$$P' = R(\theta) \cdot P$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2}$$

Rotation

$$P' = S(s_x, s_y, s_z) \cdot P$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3}$$

Scaling

As it can be seen from the above equations, all transformations require multiplication of a 4x4 and a 4x1 matrices. Other than these transformations, some others such as shearing and shadowing also require the same multiplications. Usually these transformations are not applied to object uniquely. First, a combination of these transformations is formed as a new transformation matrix, and then, this new matrix is applied to the animation objects to reduce computational complexity. When these transformations are combined into a new matrix, the size of the matrix is again 4x4. As a result, combined transformations also require multiplication of a 4x4 and a 4x1 matrices. Graphics packages create animation effects, by applying above mentioned transformations on to mathematically defined objects. To create a simple camera move action, new coordinates of all objects in the scene have to be calculated and these calculations are done through matrix multiplication.

TRANSFORMATION MODULE DESIGN

In this research work, for 3D homogeneous transformation, a hardware module was designed to be used with FPGA based custom computing machines. The module is designed to multiply a constant 4x4 matrix with a series of 4x1 matrices and to produce a new series of 4x1 matrixes. The module is designed to comply with IEEE 754-1985 standard and to process 32-bit floating point data. The module design is coded in VHDL and mapped to Xilinx's *Virtex5* chip using Xilinx's ISE WebPack electronic design automation (EDA) tool. Here, details of the module design are presented.

General structure of the module

Top level block diagram of the module is shown in Figure 5. Since the module is designed as a stream processor, it has two sets of memory signals. It reads data from one memory unit, processes the data, and writes the results to the other memory unit. Using 32-bit address boxes and data boxes, the module is able to address 4 Giga address space and process 32-bit floating-point data. For each memory unit, to synchronize read/write operations, the module produces separate memory control signals, which are strobe and read/write. Reset, Start and Done signals are used for handshaking with the host or

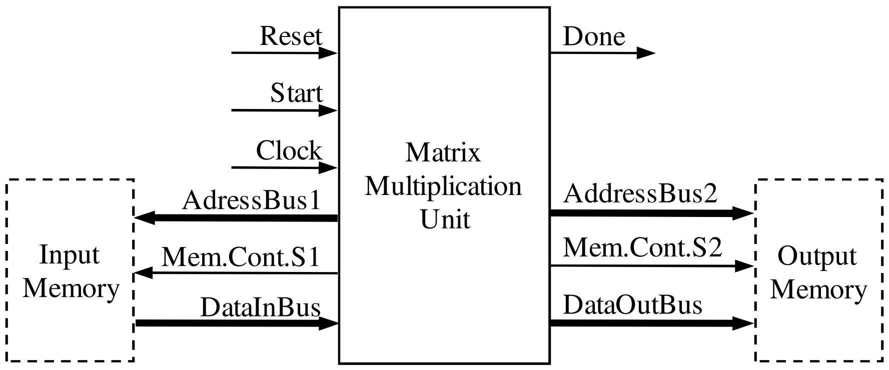


Figure 5. Top level block diagram of matrix multiplication module.

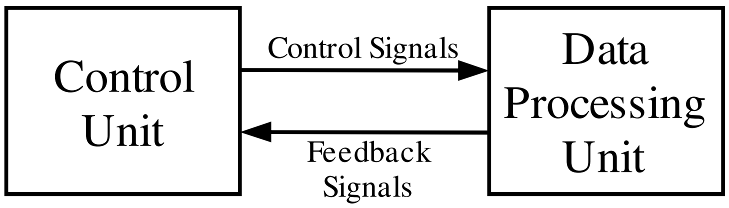


Figure 6. Second level block diagram of the module.

controlling computer.

Figure 6 show the second level block diagram of the module. The module was designed in two parts which are the control unit and the data processing unit. The purpose of the control unit is to generate required control signal for both handshaking with the controlling computer and processing data. For handshaking purpose, the controller listens Reset and Start signals and it generates an interrupt signal. For data processing, the controller is responsible for generating control signals that go to both memory units and controller signals that coordinate data flow in the data processing unit. Details of the controller are given in the following section. The Data Processing Unit consists of registers, adders, multipliers, and multiplexers, and can perform 4x4 and 4x1 matrix multiplication through parallel working multipliers and adders. This unit is also responsible for tracking source and destination memory addresses.

Block diagram of the data processing unit is shown in Figure 7. The Data Processing Unit is designed in two parts which are data access counters and core unit. Three counters are employed to manage data access. Vertex counter (VC) is used to count number of vertex to be processed. This counter is a countdown counter and is initialized to number of vertices to be processed before the unit start processing vertices. After processing a vertex, value of this counter is decremented by one. When this counter reaches to zero the Done signal is sent to the controller to let the Unit stop processing.

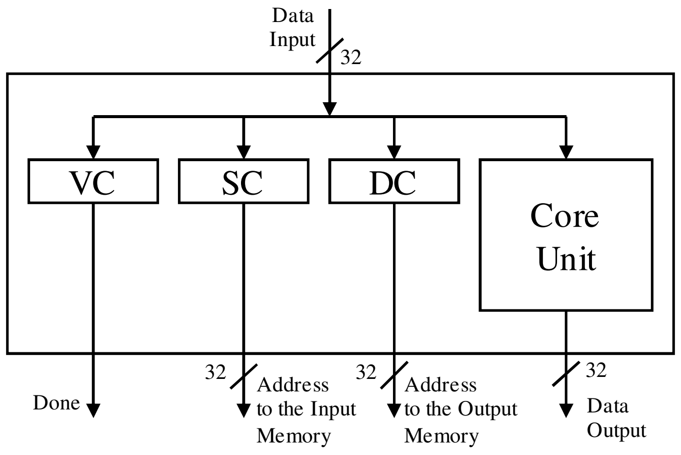


Figure 7. Data processing unit of the module.

Source counter (SC) and destination counter (DC) are used to keep track of source and destination data addresses. SC is used for addressing original vertex data in the input memory and DC is used for addressing newly calculated vertex address in the output memory. These counters are also initialized before the unit starts processing the vertices. Since, for each vertex, four floating-point numbers are kept in the memory (for *x*, *y*, *z*, and *w*), these counters are incremented by one four times

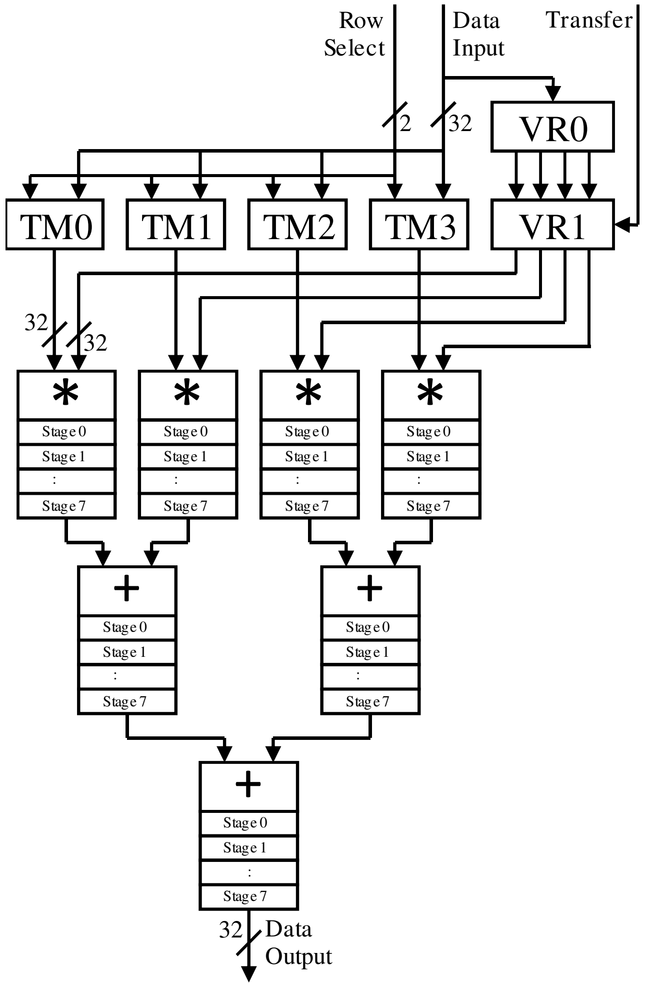


Figure 8. Block diagram of the core unit.

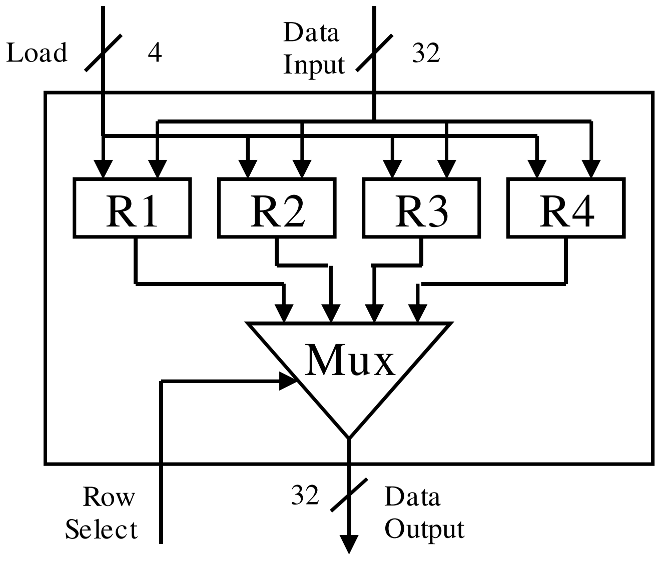


Figure 9. Block diagram of the one TM registers.

times while a vertex is being processed.

Figure 8 shows block diagram of the core unit. The core unit consists of four transformation matrix registers (TM0 … TM3), two vertex registers (VR0 and VR1), four floating-point multiplication units, and three floating-point addition units. As shown in Figure 9, each TM register contains four 32-bit loadable registers and a 32-bit 4x1 multiplexer. TM registers are used to hold constant transformation matrix values. Each register holds one column of the matrix. During a multiplication operation, through parallel working multiplexers in each register, rows of the transmission matrix are selected one by one and send to multipliers. Two vertex registers (VR0 and VR1) have different structures as shown in Figure 10a and b. VR0 act as a buffer between the Input memory and VR1 register. Continually coming vertex data from the input memory is first stored in VR0. Loading one vertex data from memory VR0 requires 4 clock cycles. When, a set of data is loaded into VR0, it is transferred to VR1 at once for transformation. The vertex data is hold in VR1 for the duration of four clock cycles and is continually fed to the multipliers. During this four clock cycles, rows of the transformation matrix are also fed to multipliers one by one and two matrices are multiplied.

Floating-point multipliers and adders were designed as an eight stage pipelined units and can process 32-bit floating-point numbers in the ways that were described in IEEE 754-1985 standard (Gloster and Sahin, 2001; Sahin, 2002; Sahin et al.,2000). Once the numbers to be multiplied or added are presented to the inputs of these units, they accept the numbers and start processing. Eight clock cycles later, the result of multiplication or addition presents at the output of the unit. This seems to be disadvantage at first, but in fact these units can accept data at every clock cycle and produce one result at every clock cycle. They can process eight pairs of number simultaneously through the pipeline stages. The only disadvantage of the units is that the first result is delayed for eight cycles. Subsequent results are produced in subsequent clock cycles.

VR0, VR1, multipliers and adders constitute a finely tuned 29-stage pipeline. Figure 11 shows the data flow through the pipelined core unit. Sequentially loaded data from input memory is recorded in VR0 registers. Once VR0 is full, all four pieces of data is transferred to VR1 registers. At the same time, a new piece of data is loaded into R1 register of VR0. Data is hold in VR1 for four clock cycles and is presented to the inputs of the multipliers. Then, data continues to propagate through the multipliers and adders, and reaches to the output, 29 cycles later than it is presented to the core unit. Every 4 clock cycles, this pipeline can multiply a given 4x4 transformation matrix with one 4x1 vertex matrix.

Module controller and operation

As shown in Figure 12, the module controller was

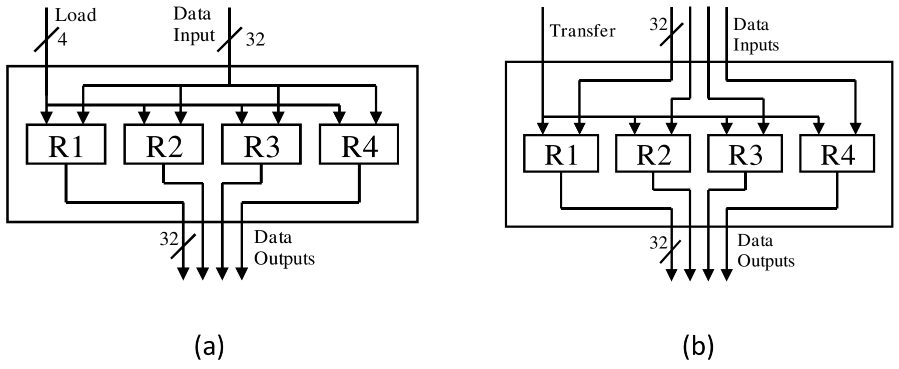
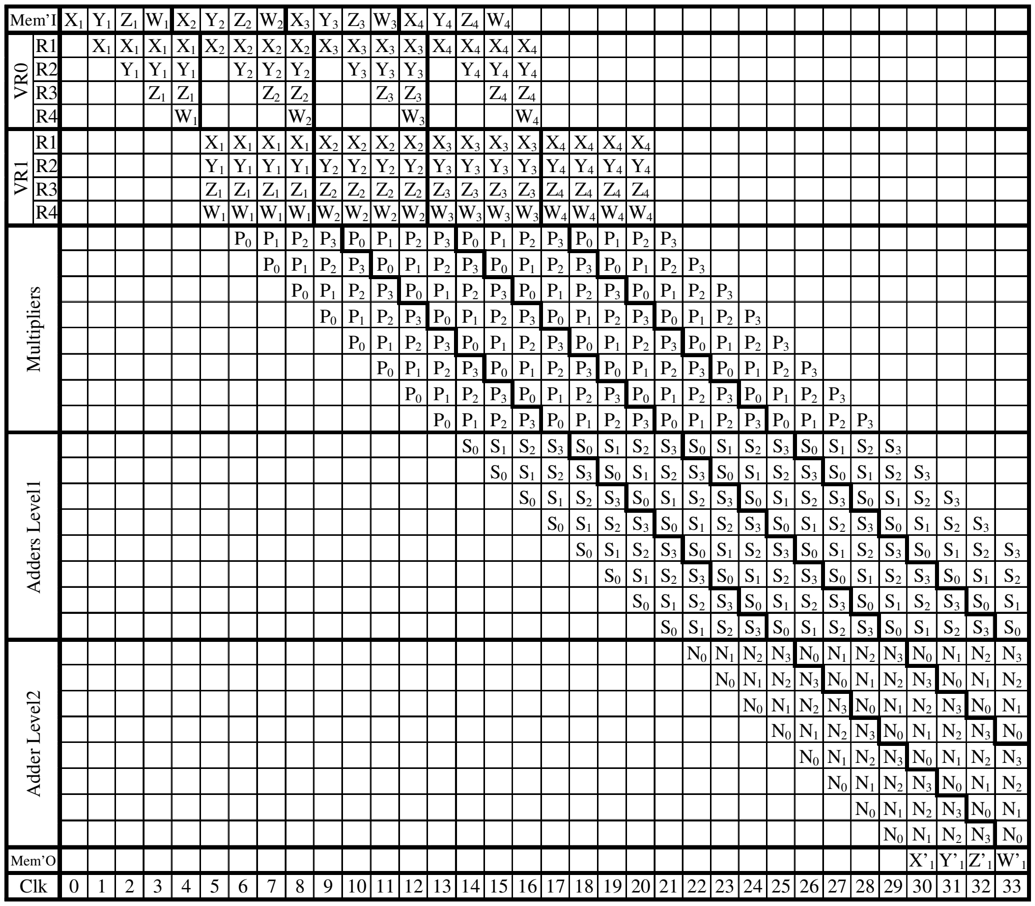


Figure 10. Block diagram of the variable registers: (a)Variable register 0 (VR0), (b) variable register 1 (VR1).



Mem’I		X_1	Y_1	Z_1	W_1	X_2	Y_2	Z_2	W_2	X_3	Y_3	Z_3	W_3	X_4	Y_4	Z_4	W_4																		
VR0	R1		X_1	X_1	X_1	X_1	X_2	X_2	X_2	X_2	X_3	X_3	X_3	X_3	X_4	X_4	X_4	X_4																	
	R2			Y_1	Y_1	Y_1		Y_2	Y_2	Y_2		Y_3	Y_3	Y_3		Y_4	Y_4	Y_4																	
	R3				Z_1	Z_1			Z_2	Z_2			Z_3	Z_3			Z_4	Z_4																	
	R4					W_1				W_2				W_3				W_4																	
VR1	R1						X_1	X_1	X_1	X_1	X_2	X_2	X_2	X_2	X_3	X_3	X_3	X_3	X_4	X_4	X_4	X_4													
	R2						Y_1	Y_1	Y_1	Y_1	Y_2	Y_2	Y_2	Y_2	Y_3	Y_3	Y_3	Y_3	Y_4	Y_4	Y_4	Y_4													
	R3						Z_1	Z_1	Z_1	Z_1	Z_2	Z_2	Z_2	Z_2	Z_3	Z_3	Z_3	Z_3	Z_4	Z_4	Z_4	Z_4													
	R4						W_1	W_1	W_1	W_1	W_2	W_2	W_2	W_2	W_3	W_3	W_3	W_3	W_4	W_4	W_4	W_4													
Multipliers								P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3												
									P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3											
										P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3										
											P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3									
												P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3								
													P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3							
														P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3						
															P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3					
Adders Level1																S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3				
																	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3			
																		S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3		
																			S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	
																				S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3
																					S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2
																						S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1
																							S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0	S_1	S_2	S_3	S_0
Adder Level2																								N_0	N_1	N_2	N_3	N_0	N_1	N_2	N_3	N_0	N_1	N_2	N_3
																									N_0	N_1	N_2	N_3	N_0	N_1	N_2	N_3	N_0	N_1	N_2
																										N_0	N_1	N_2	N_3	N_0	N_1	N_2	N_3	N_0	N_1
																											N_0	N_1	N_2	N_3	N_0	N_1	N_2	N_3	N_0
																												N_0	N_1	N_2	N_3	N_0	N_1	N_2	N_3
																													N_0	N_1	N_2	N_3	N_0	N_1	N_2
																														N_0	N_1	N_2	N_3	N_0	N_1
																															N_0	N_1	N_2	N_3	N_0
Mem’O																																X'_1	Y'_1	Z'_1	W'_1
Clk		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33

Figure 11. Data flow through the pipelined core unit.

designed as a Finite State Machine (FSM) with 26 states. When the module is powered-up, the controller starts to wait in the Reset state. When the Start signal is asserted by the host computer, the controller goes into the B_Wait state. In this state, it sends request signals to both boxes to the memories by asserting MemBusRequest signals. When the busses are granted to the Module, it goes to 0…0 h address and starts reading parameters from the input memory. The module reads three parameters which are transformation matrix values, vertex count, and destination address. The parameters and their meanings are listed in Table 1. 16 transformation matrix values are

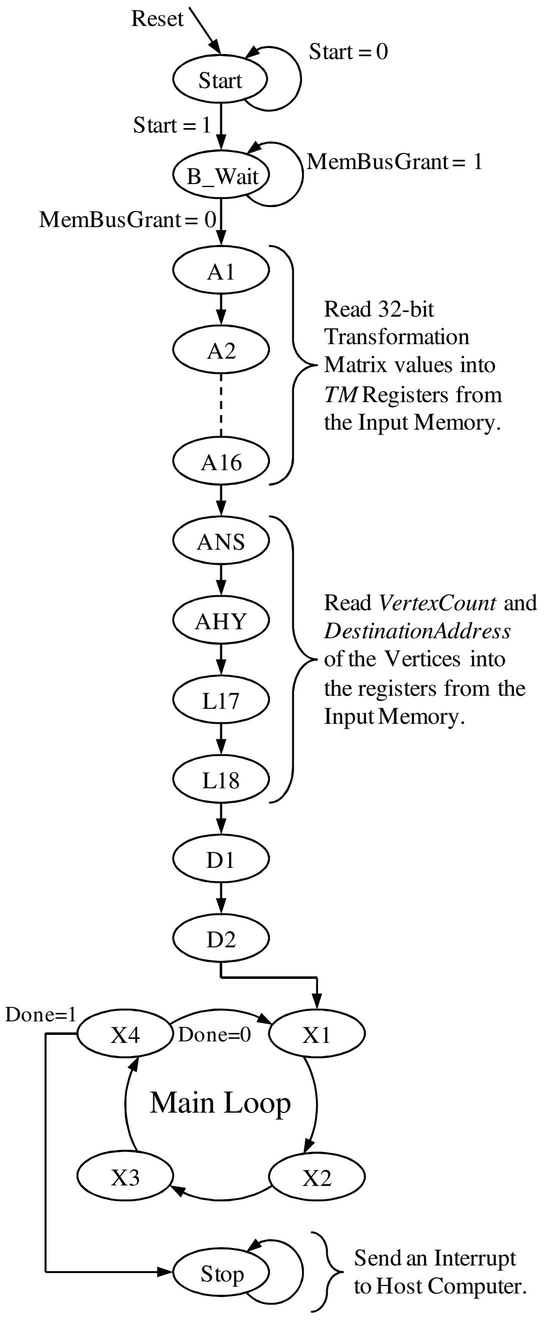

Reset
Start
Start = 0
Start = 1
B_Wait
MemBusGrant = 1
MemBusGrant = 0
A1
A2
A16
Read 32-bit Transformation Matrix values into *TM* Registers from the Input Memory.
ANS
AHY
L17
L18
Read *VertexCount* and *DestinationAddress* of the Vertices into the registers from the Input Memory.
D1
D2
Done=1
X4
Done=0
X1
Main Loop
X3
X2
Stop
Send an Interrupt to Host Computer.

Figure 12. State diagram of the module controller.

loaded into TM registers sequentially. The vertex count parameter defines the number of vertices to be processed. This parameter is loaded into VC register of the data processing unit. The last parameter, destination address, defines the location where the newly calculate vertex values will be written in the output memory. It is loaded into DC register in data processing unit. While parameters are loaded into module SC counter is used as an address counter. Later, this counter is used for addressing original vertex data.

After reading the parameters through 20 stages, the controller goes into a 4-clock cycle loop. At each iteration of the loop, one vertex data is loaded from the input memory and fed into the pipelined core unit. While the loop continues, the controller operates three different modes. In the first mode, the controller continuously feeds the core unit until the first results reaches the pipeline output, and performs no write operation. In this mode the pipeline is filled. Once it is filled, the controller goes into the second mode. In this mode, the controller reads data from the input memory, at the same time, performs calculations and writes the result to the output memory. Finally, when the last data is loaded from the memory, the controller goes into the third mode. In this mode, it continues iterating the loop, performs no read operation, and empties the pipeline. After emptying the pipeline, the controller exits from the loop and goes into the Stop state. In this state, it sends an interrupt signal to the host computer and waits for the next transformation operation.

EXPERIMENTAL SETUPS AND TEST RESULTS

The module designed in this study was mapped to Virtex5 FPGA chip, and maximum clock frequency and amount of hardware resources needed for the module were determined. Data processing speed of the module was compared to some selected general purpose computers to decide modules transformation performance. Table 1 lists the configurations of the three selected computers. PC-1 and PC-3 contain 32-bit single-core Intel processors and PC-2 contains 32-bit dual core AMD processor. To obtain timing information on PCs, a C++ program was written, compiled with VS.NET 2003 IDE, run on PCs, and given test data was processed with the program. In the C++ code, time stamp counter (TSC) of the microprocessor was read before and after the code fragment that actually performs transformations. Then, according to processor's clock speed, these TSC values are converted to CPU timing information.

The module was mapped to Virtex5 chip (xc5vlx50t-3ff1136) using Xilinx's ISE WebPack 9.2 EDA tool. Table 2 shows the mapping results in terms of hardware requirements and maximum clock frequency. According to the results, when the number of LUTs and the number of occupied slices are considered, it is possible to fit five or four copies of the module into the chip, respectively. On the other hand, when the IOBs utilization is considered, it is possible to fit only two copies of the module in to the chip. By placing multiple copies of the module into single chip, it is possible to run copies of the module in parallel and gain more speed-ups.

Five sample test data files were generated to test module data processing speed and the results were compared with general purpose computers' results. Five test files include 100, 1000, 10000, 100000 and 1000000 vertex coordinate values. For each vertex, four 32-bit floating-point numbers (x, y, z and w) were recorded in

Table 1. General purpose computers used in the experiments.

	CPU			Memory		BUS	
PC	**Type**	**Speed (GHz)**	**Cache (KB)**	**Type**	**Size (MB)**	**FSB (MHz)**	**Width (Bit)**
PC1	Intel Cel.	2.60	128	DDR 1	256	400	32
PC2	AMD Athl.	2.00	2048	DDR 2	512	667	64
PC3	Intel Pent.	1.73	2048	DDR 2	512	533	32

Table 2. FPGA chip statistics for the module.

FPGA chip	Number of slice regs./%	Number of LUTs/%	Number of occupied slices/%	Number of bounded IOBs/%	Max. clock frequency (MHz)
Virtex 5	4281 / 14	4987 / 17	1671 / 23	184 / 38	288.376

Table 3. Processing times of the PCs for different size data sets.

Number of vertices	**PC-1** (µs)	**PC-2** (µs)	**PC-3** (µs)
100	16.87	9.15	8.76
1000	64.99	72.87	68.96
10000	684.65	714.87	679.03
100000	9374.62	7271.61	6920.45
1000000	98939.94	77589.75	70163.72

Table 4. Processing time of the module on virtex 5 chip for different size data sets.

Number of vertices	**Processing time on virtex 5** (µs)
100	1.47
1000	13.87
10000	138.80
100000	1387.24
1000000	13872.08

the test files.

First, the test data files were processed with general purpose computers. Table 3 shows each computer's processing times of the sample files in microseconds. These processing times only include reading data from the main memory, processing it (applying the desired transformation on the data), and writing the results back to the main memory, and do not include data transfer times between the hard drive and the main memory.

Second, the module's processing times of the given test files were determined when the module is clocked at 288.376 MHz which is the maximum clock rate that can be applied to the module in Virtex5 Chip. Data processing times of the module are given in Table 4.

The chart in Figure 13 shows the speed-up of the module on Virtex5 compared to PCs for the same set of data. The module can perform transformations from 4.68 to 11.47 times faster than PCs depending on the data size and the PCs' configurations. As was mentioned, these speed-ups were determined when only one copy of the module utilized on a Virtex5 chip. By utilizing two copies of the module on a single FPGA or by utilizing multiples copies of the module on multiple FPGA chips even, more speed-ups can be achieved.

Conclusions

In graphics applications, matrix operations and geometric transformation are intensively used for animation effects. When these transformations are applied to scenes containing hundreds of objects, calculating transformation of even a short animated movie requires huge amount of CPU time. If it is a real time animation, it becomes impossible to perform all calculation in time with general purpose processors. Several solutions to the problem

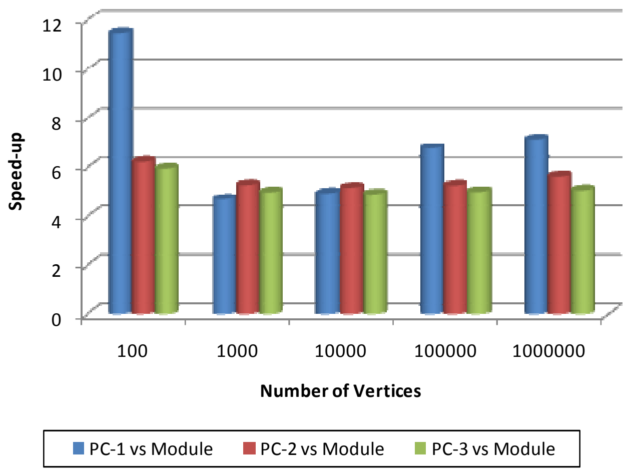


Figure 13. Speed-up of the module on Virtex5 FPGA chip compared to PCs.

such as graphics card with enhanced graphics processing units CPU, specially designed computers for computer graphics and parallel computers have already been developed (Silicon Graphics International, 2007; Bensaali et al., 2003).

In this research work, as an alternative solution, a hardware module was designed to speed-up 3D geometric transformations and thus speeding-up the graphic animations. The module was designed to run on FPGA devices and to process standard 32-bit floating-point data.

The module was tested using real test data and functional verification was done by comparing the results produced by the module and the results produced by the selected PCs. Module's data processing speed was measured and compared with three selected PCs' data processing speed. The results showed that when the module was running at 288 transformation operations can be speeded-up from 4.68 to 11.47 times compare to the PCs. Even more speed-ups can be attained when multiple copies of the module run in parallel.

Above accelerations were reported for FPGA implementation of the module. If the module is implemented as application specific integrated circuit again more speed-ups can be obtained. The module designed in this research work not only to speed-ups transformation but also is a cost effective solution compared to the other approaches. It can run as a co-processor on several FPGA boards that can be plugged in PCI slots. It is also a scalable solution. As needed, multiple copies of the module can be utilized on multiple FPGA chips and can be run on parallel to meet desired speed-up needs. This flexibility makes the solution offered in this work scalable for a given problem.

The module was designed to run solely on single FPGA chip. Chip statistics show that two copies of the module can perfectly fit in to a single Virtex 5 chip. In the future, some research work can be done to place two copies of the module in to single chip. Moreover, the way of utilizing even more copies of the module on more than one chip can also be studied. In such a case, an interface needs to be developed for efficiently scheduling transformation requests coming from a graphics API or from a graphics software, and for managing data flow between the software and the copies of the module.

REFERENCES

Bensaali F, Amira A, Uzun IS, Ahmedsaid A (2003). An FPGA Implementation of 3D Affine Transformations. 10th IEEE Int. Conf. Electronics, Circuits Syst., Sharjah, UAE, 2: 715-718.
Boullis N, Tisserand A (2003). Some Optimizations of Hardware Multiplication by Constant Matrices, Proceedings of the 16th IEEE Symposium on Computer Arithmetic, Spain, pp. 20-27.
Boullis N, Tisserand A (2005), Some Optimizations of Hardware Multiplication by Constant Matrices. IEEE Trans. Comput., 54(10): 1271-1282.
Brown S, Rose J (2002). Architecture of FPGSs and CPLDs: A Tutorial, http://klabs.org/richcontent /Tutorial/fpga/Toronto tutorial.pdf.
Buell DA, Arnold JM, Kleinfelder WJ (1996). SPLASH 2: FPGAs for Custom Computing Machines. IEEE Comput. Soc. Press, Los Alamitos.
Disney/Pixar (2008). How We Make a Movie, Online at: http://www.pixar.com/howwedoit/index.html.
DeHon A, Wawrzynek J (1999). Reconfigurable Computing: What, Why, and Implications for Design Automation, Proceedings of 36th Design Automation Conference, New Orleans, pp. 610-615.
El-Atfy R, Dessouky MA, El-Ghitani H (2007). Accelerating Matrix Multiplication on FPGAs, 2nd International Design and Test Workshop, Egypt, pp. 203-204.
Figueiredo MA, Gloster C (1998). Implementation of a Probabilistic Neural Network for Multi-spectral Image Classiffication on an FPGA Based Custom Computing Machine, Proceedings of 5th Brazilian Symposium on Neural Networks, pp. 174-179.
Figueiredo MA, Gloster C, Stephens M, Graves C, Nakkar M (2000). Implementation of Multi-spectral Image Classiffication on a Remote Adaptive Computer, J. VLSI Des. Special Issue Reconfigurable Comput., 10(3): 307-319.
Gloster CS, Sahin I (2001). Floating-Point Modules Targeted for Use with RC Compilation Tools, Earth Science Technology Conference, College Park, MD.
Graham P, Nelson B (1996). Genetic Algorithms in Software and in Hardware, Fourth IEEE Workshop on FPGAs for Custom Computing Machines.
Hauck S (1998). The Roles of FPGSs in Reprogrammable Systems, Proceedings of the IEEE, pp. 615-638.
Hearn D, Baker MP (2004). Computer Graphics with OpenGL (3rd Edition). Prentice Hall Publication, United States of America, p. 345.
Hogl H, Kugel A, Ludvig J, Manner R, Noffz KH, Zoz R (1995). Enable++: A Second Generation FPGA Processor, Third IEEE Workshop on FPGAs for Custom Computing Machines.
John M, Smith S (1997), Application-Specific Integrated Circuits. Addison-Wesley Inc.
Lewis T, Perkowski M, Jozwiak L (1999), Learning in Hardware:Architecture and Implementation of an FPGA-Based Rough set Machine, Proc. of 25th EUROMICRO Conference, Italy, pp. 326-334.
Lin, W (2007), "Design a program to render the wire-frame of a Utah teapot", Online at: http://caig.cs.nctu.edu.tw/course/CG2007 /assignments.htm.
Perkowski M, Chebotarev A, Mishchenko A (1999). Evolvable Hardware or Learning Hardware? Induction of State Machines from Temporal Logic Constraints, Proc. of the First NASA/DoD Workshop on Evolvable Hardware, California, pp. 129-138.
Prada EC, Charlwood SM, James-Roxby PB (1999). Image Processing

and Its Applications, Seventh Int. Conf. Image Process. Appl., 1: 450-454.

Ratha NK, Jain AK (1999), Computer Vision Algorithms on Reconfigurable Logic Arrays, IEEE Trans. Parallel Distributed Syst., 10(1): 29-43.

Ratha NK, Jain AK, Rover DT (2000), FPGA-Based Coprocessor for Text String Extraction. IEEE International Workshop on Computer Architectures for Machine Perception, Italy, pp. 217-221.

Sahin I (2002) A Compilation Tool for Automated Mapping of Algorithms onto FPGA Based Custom Computing Machines, PhD dissertation, NC State University, Raleigh-USA.

Sahin I, Gloster CS (2005). Evaluation of IC Physical Design Optimization Algorithms for Acceleration Using FPGA-Based Custom Computing Machines, 4th International Advanced Technologies Symposium, Konya, Turkey.

Silicon Graphics International (2007). Online at:http://www.sgi.com.

Tavares RCDM, Coelho CJN, Araujo ADA, Fernandes AO (1998). Implementation of an Edge Detection Algorithm in a Reconfigurable Computing System, Proceedings of the Eleventh Brazilian Symposium on Integrated Circuit Design, pp. 38-41.

Tessier R, Burleson W (1998). Reconfigurable Computing for Digital Signal Processing: A Survey. J. VLSI Signal Process., 28:7-27.

Villasenor J, Hutchings B (1998). The Flexibility of Configurable Computing. IEEE Signal Process. Mag., 15(5): 67-84.

Vuillemin J, Bertin P, Roncin D, Shand M, Touati H, Boucard Ph (1996). Programmable Active Memories:Reconfigurable Systems Come of Age, IEEE Trans. VLSI Syst., 4(1): 56-69.

Xilinx Inc (1994). The Programmable Logic Data Book, San Jose, CA.

Xilinx Inc (2002). Virtex-II ProTM Platform FPGAs: Functional Description, San Jose, CA.