

GPU Acceleration of 3D Object Transformations

Sura Alrawy, Fakhruddin Ali
university of Mosul
Iraq
suranawfal@gmail.com
dr.fakhali@gmail.com



ABSTRACT: *Generating 3D animation scenes in computer graphics requires applying a 3D transformation on the vertices of the objects. These transformations consume most of the execution time. Hence, for high-speed graphics systems, acceleration of vertex transform is very much sought for because it requires many matrices operations that to be performed at a real-time, so the execution time is essential for such processing. In this paper, the acceleration of 3D object transformation is achieved using parallel techniques such as Multicore Core Central Processing Unit (MC CPU) or Graphic Processing Unit (GPU) or even both. Multiple geometric transformations are concatenated together at a time in any order with interactive manner. The performance results are presented for a number of 3D objects with paralleled implementations of the affine transform on the NVIDIA GPU series. The maximum execution time was about 0.508 seconds to transform 100 million vertices. Other results also showed the significant speedup compared to (CPU and MC CPU) computations for the same object complexity.*

Keywords: GPU, 3D Object, Transformation, Vertices

DOI: 10.6025/jitr/2018/9/3/94-106

Received: 17 March 2018, Revised 16 April 2018, Accepted 20 April 2018

© 2018 DLINE. All Rights Reserved

1. Introduction

In computer graphics, the most popular method for displaying a 3D object is the polygon mesh model which consists of polygons represented by a list of points or vertices. In many applications, there is a need for altering the scene objects or parts of them, especially in computer animations. As the number of objects in the animation scene increases, the number of vertices used to define these objects will increase, so the speed is needed to manipulate these vertices.

Geometric transformations are the ways of manipulating the vertices while preserving the spatial relationships among them. Changes in orientation, shape object, and size are accomplished with these transformations that are applied on each and every vertex of the object to obtain a transformed one [1][2], these transformations are called affine transformation (AT) which may consist of many operations such as translation, rotation, shearing, scaling and others. The AT plays an important role, not only

in computer graphic application, but also in various high speed applications such as medical imaging and machine vision applications as in [6][7].

The transformation operations are highly computationally intensive as it involves matrix multiplication of trigonometric functions that is applied for each vertex individually. So, the speed of these transformations is the challenge of producing realistic vision of animation scenes [3]. This realistic requires fast execution of addition, multiplication, and trigonometric which attracted and still do many research work and different architectures as presented in the following reviews:

Several attempts to accelerate such transformations have been proposed, most of them have implemented on FPGA [3]-[7], Biswal etc. proposed a parallel algorithm which calculates AT of two voxel locations using a single transform operation [5]. Then they modified this algorithm in [7] to be able to calculate AT of four voxel locations by performing a single transformation, but this algorithm assumed symmetry about four-pixel locations for the transformed object. In [6] [9] and [10], 2D transformations for images have been implemented, on the other hand, most researchers have put more emphasis on affine rotation only [9] and [10].

Nowadays, a lot of work is being done to implement parallel technique into existing algorithms to enhance their performance, matrix multiplication is a computational problem that has been accelerated using (GPU) as in [12][13] or even multicore [14].

In this work, an adaptive 3D graphics transformations is designed and accelerated using parallel techniques GPU and multicore, where all vertices are transformed using the AT matrix simultaneously by assigning each group of vertices to each thread and multiplying them as single instruction multiple data SIMD.

The rest of this paper is organized as follows: section 2 discusses the theoretical and mathematical explanation of 3D transformations. Section 3 summarizes CPU and GPU architectures. Implementation and analysis of the designed 3D transformation are described in Section 4. Section 5 describes the experimental results measured at real time for many objects, and finally, section 6 states conclusions of this paper.

2. Geometric 3D Transformation

3D transformations are the ways of moving the vertices that describe one or more 3D objects to new locations or orientations, these processes involve translation, scaling, shearing, and rotation. They are applied to each individual vertex and repeated to all object vertices to achieve the required operation. In computer graphics, the matrix notation is the suitable way to describe each of these operations, and a vector describes each vertex. Scaling, shearing and rotation transforms are linear transforms that can be represented by 3×3 matrix, but translation transform is nonlinear. Combining linear transforms and translations can be done using an affine transforms (AT), which are the ones most often used in graphics, but in 3D, AT cannot be implemented using 3×3 matrices, so 4×4 matrix is used. On the other hand, the homogeneous coordinates in the transformations are needed to promote the 3D coordinate by adding a fourth component of unity to the vectors that represent positions to be as 4×1. As a result of that, the input vertices are described as $v = (x, y, z, 1)$ and the transformed vertices as $(x', y', z', 1)$ [1][2].

2.1 Translation Matrix

Translation displaces vertices to new positions defined by a displacement vector $t = [Tx \ Ty \ Tz]$ the algebraic and matrix representation for 3D translation (T) are shown in (1).

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (1)$$

By this transform, the input vector (x, y, z) is left unaffected by a multiplication by T, because a direction vector can't be translated. In contrast, both points and vectors are affected by the rest of affine transforms [2].

2.2 Scaling

Scales expands or contracts a 3D object with components (x, y, z) by the factors S_x, S_y and S_z along the x-, y- and z- direction respectively. The larger the $S_i, i \in \{x, y, z\}$, the larger the scaled entity gets in that direction. Setting any of the components of s to one naturally avoid the changes in scaling in that direction. The equation of scaling is described in equation (2).

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (2)$$

The scaling operation is called uniform if $Sx = Sy = Sz$ and non-uniform otherwise [1][2].

2.3 Rotation Matrix

To rotate an object in a 3D space, an axis of rotation need to be specified in addition to the angle of rotation. This can have any spatial orientation in a 3D space. The transformation matrices for rotation about the X, Y, and Z-axes, respectively are in (3), (4) and (5) equations.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (3)$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (4)$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (5)$$

All rotation matrices have a determinant of one and are orthogonal.

The order of rotation should be considered into account in 3D space where the order of rotation affects the final position of the rotated object because matrix multiplication is not a commutative operation. Rotation about the x-axis by an angle α , followed by a rotation about the y-axis by an angle β does not give the same result as the one obtained if the order of the rotations is reversed [1][2]. So based on this property there are six possibilities of choosing the rotation axes, in other words, there are six sequences of product of individual rotations about three axes, this type of formalism is called Tait–Bryan angles[2].

2.4 Combining Transformations

In addition to applying the individual transformation on objects, sequence of these transformations can be combined to form many functions that are required in computer graphics. This is done by multiplying or concatenating any sequence of the previous matrices to achieve the desired result, by this strategy it is preferable to define any arbitrary transformation directly with a single new matrix, and then, this new matrix is applied to the animation objects. This approach fits well for implementing graphics systems since it dramatically reduces computational complexity and execution time [2].

3. Implementation Platforms

The CPU and GPU are chosen in this paper as platforms for implementing many 3D transformations for sequential and parallel execution, a brief introduction of each platform is overstated.

3.1 Central Processing Unit (CPU)

CPU architecture has only one processing unit in the chip (See figure (1), for performing arithmetic or logic operations. At any time only one operation can be performed [15].

3.2 CPU with Multicore Processor

A multicore processor is a system that comprises of two or more independent cores (or CPUs). The cores are generally integrated

onto one integrated circuit die (known as a chip multiprocessor), or they are integrated onto multiple dies on a single chip package [15] as in Figure (2).

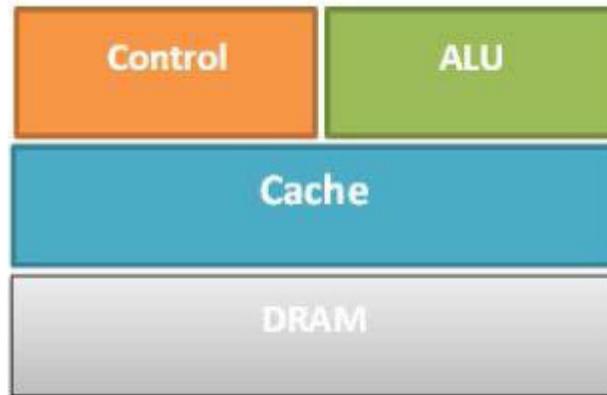


Figure 1. CPU hardware architecture

3.3 Graphics Processing Unit (GPU)

GPU is viewed as a computing device operating as a coprocessor to the main processor (CPU host). A GPU is implemented as an aggregation of multiple processors so it is called multiprocessors, which consists of a number of Single Instruction Multiple Data (SIMD) ALUs integrated as a network on a chip (See figure (3)). According to the SIMD, every processor within GPU must execute the same instruction at the same time, only data can be varying [15].

Refer to figure (3), the orange color indicates the cache memories, the blue color indicates the control units and the green color indicates the ALUs.

In this paper, GeForce GTX 1050 is used for parallel implementation, this GPU is based on Pascal architecture GP107 chip as shown in figure (4), where there are six SMs each with 128 shader units, NVIDIA has disabled some shading units on GTX 1050 to reach the product’s target shader count. It features 640 shading units, 40 texture mapping units and 32 ROPs. NVIDIA has placed 4 GB GDDR5 VRAM on this card, which is connected using a 128-bit memory interface [16].

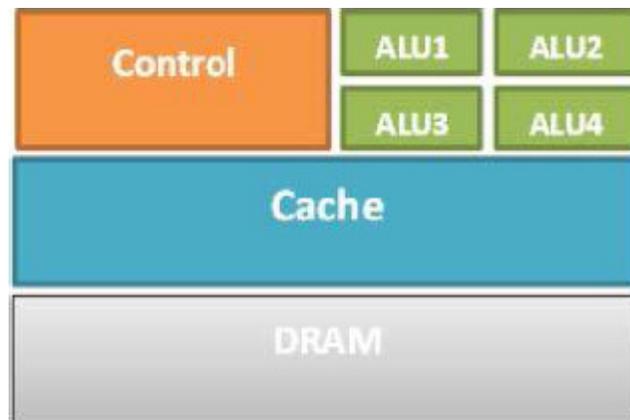


Figure 2. Multicore hardware architecture

4. Implementation and Testing

In this work LabVIEW environment is used for implementing and testing the transformations, the software version is professional 2017; two PCs are used in this work the first is notebook with 16GB RAM, Intel® Core™ i7 7700HQ @2.8 GHz processor with one

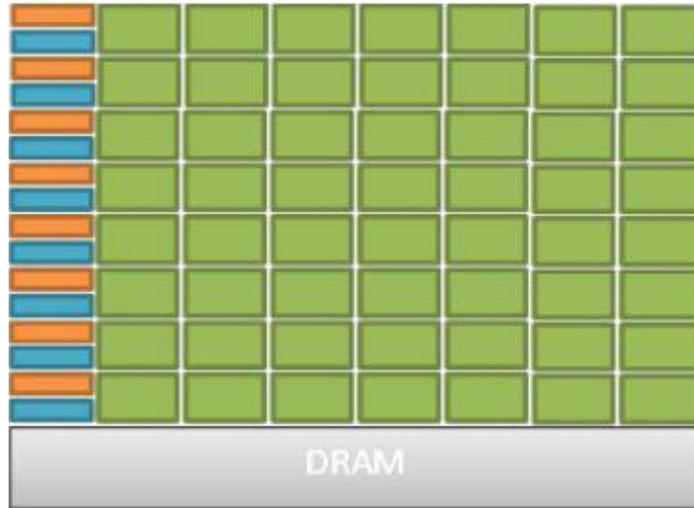


Figure 3. GPU hardware architecture

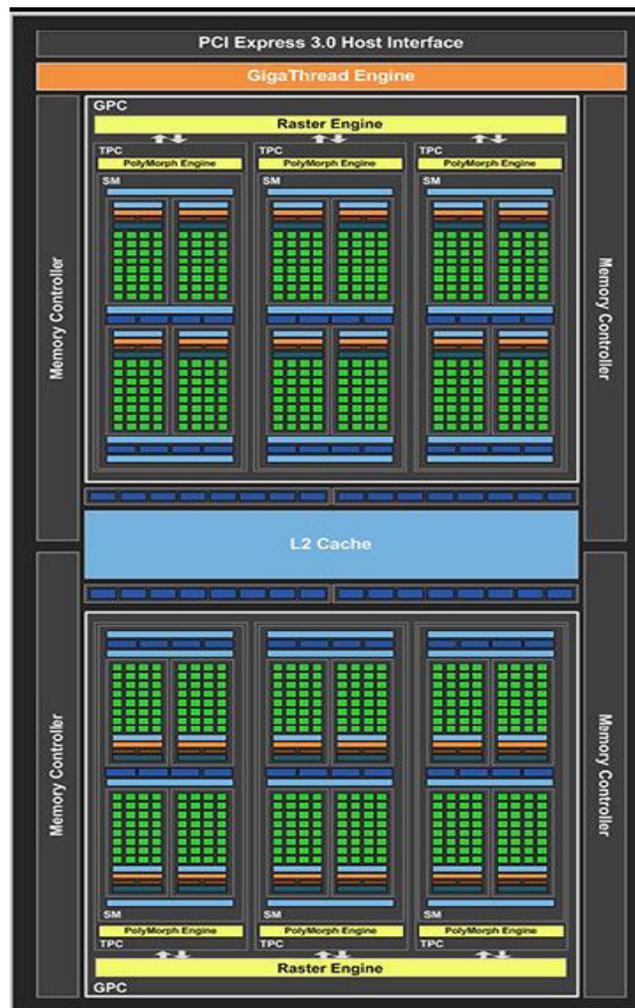


Figure 4. The GP107 graphics processor architecture

NVIDIA GeForce 1050 GTX of compute capability 6.1, containing 5 streaming multiprocessor each of them contains 128 units as mentioned before, the maximum memory data rate (112 GB/s). Applications are designed in CUDA version 9.0 and using Visual Studio 2015, Nvidia Graphics driver version 22.21.13.8554 is used for CUDA compatibility. The second PC used with Intel® Core™ i5 3210m, 2.8 GHz processor with one NVIDIA GeForce 610 GT of computing capability 2.1, containing 1 streaming multiprocessor of 48 cores, memory data rate (14GB/s). Applications are designed in CUDA version 5.0 and using Visual Studio 2010.

4.1.1 Import 3D Model

Firstly the 3D model is imported into the LabVIEW as a mathematical model forming by vertices, edges, and surfaces, the vertices of these models are extracted to apply the transformations on it, then the transformed vertices are stored back into the model to redisplay the transformed object. The block diagram.vi of this operation is shown in Figure (5).

Modeling any animated object requires defining thousands even millions of vertices for high resolution. Two 3D test models are used with different resolutions; these models are bunny and dragon which are standard computer graphics created at Stanford University [17]. In general, any 3D object can be stored in different formats as (.obj, .stl, .ply...etc), the (stl) format is chosen in this work since LabVIEW support this format, so the standard models are firstly converted to (stl) format.

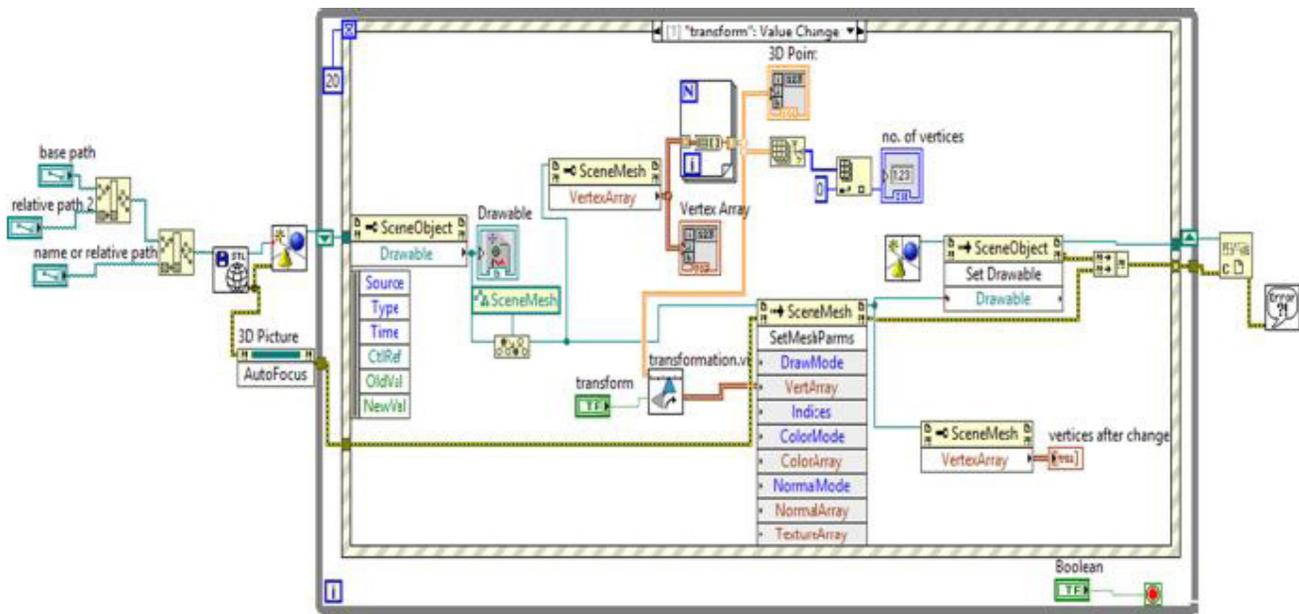


Figure 5. Block diagram of importing 3D model

4.2 Transformation Matrix

The transformation matrices are built and combined together in an accumulating manner to generate one new matrix, and then this accumulating matrix is used for per-vertex transformation of the object to reduce the execution time. In this paper, the combination form is desired to achieve all possible sequences of transformations, where in contrast to 2D graphics, the order of some transformations is a considerable issue in 3D graphics, for example translating a 3D object then rotating it, does not equivalent to rotating then translating the same object. Also in the rotation transform, the order of rotation affects the final position of the object since there are three axes of rotation, in other words, a rotation matrix has three degrees of freedom that represents a 3D rotation in every imaginable way. So, there are six possible orders: $x - y - z, x - z - y, y - x - z, y - z - x, z - x - y, z - y - x$. All these sequences are considered in designing the transformation unit. Figure (6) displays the execution front panel of the designed vi, as shown, the user can change the parameters of transformation unit in an interactive manner at real time, where the event case structure is used for these parameters to take the effect of each change and redisplay the output.

4.3 Sequential and Multicore Execution

For comparison purpose, the 3D transformations are applied on the test objects without using parallel techniques where the

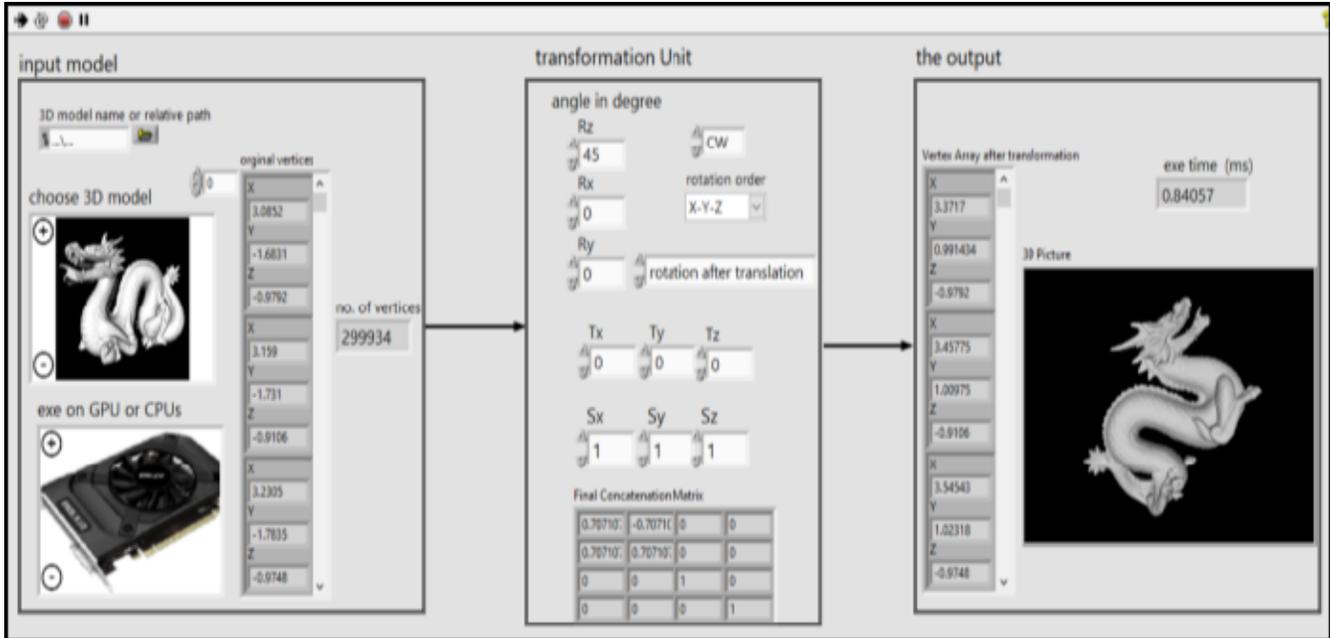


Figure 6. The front panel of the transformation unit

vertices are processed sequentially one vertex after another, then the multicore toolkit is used in LabVIEW for matrix computation instead of traditional tools to speed up the operations as shown in figure 7 where the number of CPUs can be chosen from this GUI.

The high resolution second.vi is used to measure the execution time, for all computations in this paper the run time is recorded after several trials because the run-time slightly differs after first execution as shown in figure (7), a snapshot of the front panel displays the run-times for the first three object resolutions as a waveform graph.

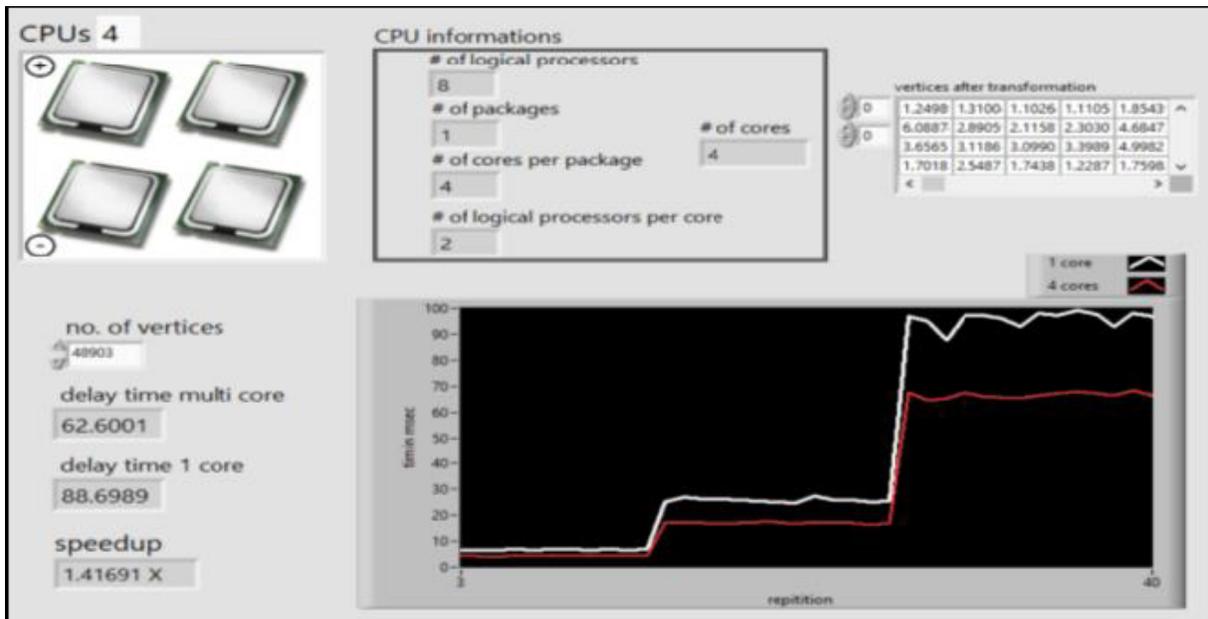


Figure 7. The execution of object transformation on multicore

4.4 Computations on GPU

To evaluate the performance of 3D transformations on GPU, The implementation was realized in NVIDIA CUDA architecture, because of hardware availability and experience with this technology. Using GPU, a large number of vertices can be executed in parallel on hundreds of cores.

In LabVIEW GPU computing, the code calls the GPU via CUDA toolkit interface, this interface is made up of two LabVIEW libraries *lvcuda.lvlib* and *lvcublas.lvlib*. The last library contains an optimized implementation of BLAS library which has vector-vector, vector-matrix, and matrix-matrix operations. The matrix-matrix operation is used in this work to compute new vertices after transformation. For implementing on GPU, first the device and library are initialized, when a kernel is created, memory needs to be allocated for both transformation and vertices matrices, then the transformed vertices are obtained after multiplication computation with xGEMM, this operation is able not only multiply matrices but to transpose first, second or both matrices. The upload and download data is also used for memory copy between host and device, finally we must free or deallocate the space of memory we created to allocate the matrices, this is done to overcome any associated problems of memory leak or crash as the system runs out of memory, and to ensure dealing with data of large vertices.

Using GPU, many vertices are processed in parallel rather than one vertex after another, since each vertex is four floats in 3D graphics, four threads would be required to compute each vertex.

5. Experimental Results and Analysis

5.1 Sequential and Multicore Implementation

To compare the parallel execution of 3D transformations with serial execution, the first test just did the vertex transformations using the CPU, single threaded and multi-threaded are implemented on core i7. A real 3D model with different resolutions is transformed and redisplayed. In this test the double precision format is used for vertices of these models. The execution time results are tabulated in table (1) for one core and multi-core and the speedup is calculated as a ratio of sequential execution time on one core to parallel execution time on four cores. From the result shown one can note that there is little improvement in speedup, also for small number of vertices less than 2844 the speedup became less than one due to the communication overhead for small data or vertices.

	# of vertices	Execution time (m sec.)			4 cores speedup
		1 core	2 cores	4 cores	
Bunny	2844	6.66257	5.26332	4.37644	1.52
	11553	26.5726	18.7992	14.1694	1.87
	48903	115.702	80.4398	72.0017	1.60
	208353	422.738	329.57	290.546	1.45
	432138	873.083	655.802	577.052	1.51
dragon	33306	84.4872	60.1939	48.1273	1.75
	143382	307.809	227.932	196.553	1.56
	299934	613.565	462.963	429.725	1.42
	607560	1227.63	923.047	812.955	1.51
	2614242	4984.46	4157.82	2928.59	1.70

Table 1. Execution times for single and multi-core CPU

5.2 Parallel implementation on GPUs

The parallel test is implemented on two types of GPUs, table (2) shows the execution times comparison on the first PC having GeForce 610m and core i5 CPU, the run times is measured for kernel execution in millisecond for the same previous set of objects

DBL	# of vertices	Execution time (m sec.)		Speedup
		<i>Sequential CPU (core i5)</i>	<i>GPU GeForce 610</i>	
Bunny	2844	6.550	0.6822	9.60
	11553	32.6531	1.9239	29.95
	48903	120.698	2.8905	41.57
	208353	498.503	8.95217	55.68
	432138	1035.94	18.075	57.31
Dragon	33306	84.135	2.1596	38.95
	143382	315.272	6.18281	50.99
	299934	702.657	12.5508	55.98
	607560	1410.55	24.2415	58.18
	2614242	—	—	

Table 2. Execution times and speedup on GPU geforce

in each sequential and parallel implementation. As shown from these results the 3D transformation is speeded up on GPU by a factor up to 58x. The missing values denotes using „-“, symbol means that these results was not computed due to their enormous running time.

The second parallel test is implemented on another PC with GPU having more cores GeForce 1050 and core i7 CPU, The more cores there are, means the more threads and then more vertices that can be served at the same time. Table (3) shows the execution times of the sequential and parallel CUDA implementation of 3D transformation and the speed-up obtained from these results. As can be seen in this table the GPU is up to 622x faster than CPU.

	# of vertices	Exe time (m sec.)		Speedup
		<i>Sequential core i7</i>	<i>GPU GeForce gtx 1050</i>	
Bunny	2844	6.66257	0.260741	25.55
	11553	26.5726	0.256729	103.50
	48903	115.702	0.636718	181.72
	208353	422.738	1.18045	358.11
	432138	873.083	2.02138	431.92
Dragon	33306	84.4872	0.555396	152.12
	143382	307.809	0.965289	318.87
	299934	613.565	1.61185	380.66
	607560	1227.63	2.62127	468.33
	2614242	4984.46	8.00748	622.47

Table 3. Execution times and speedup on gpu geforce gtx 1050

5.3 Single vs Double Precision

Double precision floating-point performance is important for specific applications in order to obtain the desired accuracy of the results. All the previous results are obtained for double precision 64-bit point values of the vertices (x,y,z,w). So, another test are recorded when we changed the representation of vertices data to single precision 32-bit, the speed of transformations increased as expected and the displayed output not affected more since the resolution depends on the number of vertices representing the objects. As can be seen in table (4) the speed of transformation is increased in comparing with double precision. Figure (8) shows the comparison graphs between single (SGL) and double (DBL) precision.

Finally table (5) and table (6) displays the comparison between this work and other previous works. In table (5), the results in [10] are for image transformations based on visual studio platform using GeForce 635, and [4] implemented the transformations on FPGA vertex 5 chip. While table 6 compares our results with [10] when using the same GPU type GeForce 610 and the same vertices numbers.

All these results show the effectiveness of our implementation for accelerating 3D objects transformations faster than other previous works.

	# of vertices	Execution time (m sec.)		Speedup
		Sequential core i7	GPU GeForce gtx 1050	
bunny	2844	6.50065	0.278975	23.30
	11553	23.5835	0.289914	81.35
	48903	114.877	0.308513	372.35
	208353	416.79	0.642918	648.27
	432138	850.642	0.934291	910.46
dragon	33306	74.2076	0.293561	252.78
	143382	251.97	0.569983	442.06
	299934	600.19	0.750131	800.11
	607560	1177.8	1.06958	1101.18
	2614242	4851.2	3.30247	1468.96

Table 4. Execution times and speedup on gpu 1050 for sgl precision

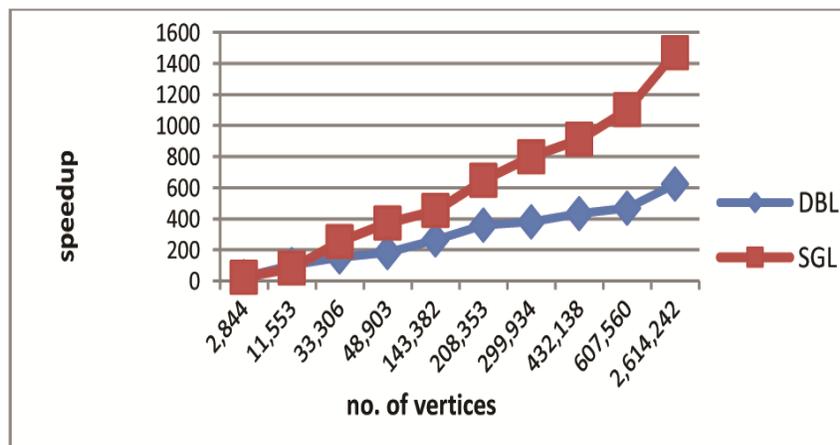


Figure 8. Speedup comparison for single and double precision vertices

# of vertices	2D transformations [10] GeForce 635	3D transformations [4] Vertex 5	Implemented 3D transformations GeForce 1050
100000	0.862	1.3870	0.42447
1000000	8.588	13.872	1.59252
10000000	85.99	-	12.3627

Table 5. Comparison times in milliseconds with previous works

# of vertices	2D transformations [10] Matlab	Implemented 3D transformations LabVIEW
1536	5.1	0.416
24576	5.6	2.038
49152	5.8	2.981

Table 6. Comparison execution times in milliseconds with [10] on the geforce 610

# of vertices	2D transformations [10] Matlab	Implemented 3D transformations LabVIEW
1536	5.1	0.416
24576	5.6	2.038
49152	5.8	2.981

Table 6. Comparison execution times in milliseconds with [10] on the geforce 610

6. Conclusions and Performance Evaluation

One of the main concerns of real-time graphics is the speed of execution, so faster processing of affine transform is extremely needed. The execution time of a graphic system is a function of the complexity of a polygonal object which can be measured by the number of vertices used to represent it. In this paper, the acceleration of 3D transformation has been achieved using the parallel technique for producing the new vertices in addition to concatenate many sequences of transformation in one adaptive matrix, so the total transform execution time has been reduced. This general 3D vertex transform has been designed using LabVIEW environment for any sequence of transformations at a time. From the results that have been discussed before, the execution time increased as the complexity of object increase. In many cores CPU, there was little improvement, where the maximum speedup obtained was about 1.87, indicating that the bottleneck was not the CPU cores but some part of the memory and cache system. On the other hand, in GPU implementation, GPU has many parallel executive units with wide bandwidth and large caches size that enabling faster execution of the vertex transform.

The first type GPU has been used to compare the performance of this design with previous work in [10], the results of CUDA LabVIEW shows that the transformations consumed less time comparing to the previous work for the same test data, although the vertex transform in [10] was for 2D models, and for the same GPU and data set, our LabVIEW design was faster than that of Matlab 2D transform as shown in table (6). Further improvement has been obtained in table (3) using the second GPU type which has more compute units and wide memory bandwidth.

Before displaying the comparison results between float and double, expected values of performance increase can be determined for single precision. For bandwidth limited applications the performance should increase by a factor of two, since the kernel has to read twice more data for the double precision version than for the single precision version (8 bytes instead of 4 bytes). The measured values in table (4) confirm the theoretical predictions, which have been based on the bandwidth limitation of this implementation, such as for the bunny object with 432138 vertices the average transform time for single precision was 0.934291

milliseconds and for double about 2.02138 milliseconds, so the average measured execution time increase is of 2.16 and hence very close to the predicted value of 2. We can conclude that the implementation of this unit on GPU is bandwidth bound rather than compute bound. However, the resolution of a model depends on the number of vertices formed that model rather than the precision of data vertices itself which represent the address values of the displayed pixels. So with single precision data, the maximum execution time has been consumed to transformed 100 million vertices was about 0.508 sec.

Comparing table (2) and (3) the execution times in the 2nd GPU decreased by a factor of around 8x since the memory bandwidth for GeForce 1050 is 112 GB/s but for GeForce 610 is 14 GB/sec.

Finally the comparison results in tables (5 and 6) show that the performance of our 3D transformation outperformed previous works in [10] and [4] where (Matlab and visual studio) and FPGA has been used respectively.

References

- [1] Taylor. (2014). Francis Group, CRC Press. Practical Algorithms For 3D Computer Graphics. Second edition R. Stuart Ferguson.
- [2] Tomas Akenine-Möller, Haines, Eric., Hoffman, Naty. (2008). Real-Time Rendering, 1045 p. from A.K. Peters Ltd., 3rd edition.
- [3] Fakhruddin Hamid Ali. (2012). Transformation Matrix for 3D computer Graphics Based on FPGA”, AI-Rafidain Engineering Journal, 20 (5) (October). <https://www.iasj.net/iasj?func=fulltext&aId=61024>
- [4] Sahin, Ibrahim. (2010). A 32-bit floating-point module design for 3D graphic transformations. *Scientific Research and Essays Journal*, 5, p 3070-3081, 18 (October). <https://pdfs.semanticscholar.org/25ac/4f1949d11323047d8e47c39299bd49a0151b.pdf>
- [5] Biswal Pradyut Kumar, Banerjee Swapna. (2010). A parallel approach for affine transform of 3D biomedical images”. International conference on electronics and information engineering (ICEIE), 1. IEEE; p. 329–32. <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5559864>
- [6] Pradyut Kumar Biswal, Pulak Mondal, Swapna Banerjee, (2011). Parallel architecture for accelerating affine transform in high-speed imaging systems, *Journal of Real-Time Image Processing*. Springer-Verlag..
- [7] Mondal, P., Biswal, P. K., Banerjee, S. (2016). FPGA based accelerated 3D affine transform for real-time image processing applications”. *Comput Electr Eng* 49(1) 69 Elsevier, <https://www.sciencedirect.com/science/article/pii/S0045790615001457>
- [8] Younis, Basma Mohammed Kamal., Sheet, Ne’am Salim Mohammed. (2013). A Real Time Dynamic 3D Graphics Processor Using FPGA” international Journal for Research and Development in Engineering (IJRDE)www.ijrde.com 2 (1) June-July p. 1-12
- [9] Liu, Zhiyuan., Zhao, Xuezhang. (2011). Research and Implementation of Image Rotation Based on CUDA” *Advanced Materials Research* 216, p. 708-712 Trans Tech Publications, Switzerland, 2011. www.scientific.net/AMR.216.708.
- [10] Hadeel Alshakargy. (2016). Execution Speed up of Image Rotation Matrix Using Parallel Technique, M.Sc. thesis, Computer Engineering University of Mosul.
- [11] Liu1, Bozhi., Sun, Tao., Zhou, Li., Wang, Jia, dand Yuanzhi Zhang. (2014). Architecture for Vertex Transformation and Triangle Clipping in 3D Graphics, 462-463, p 1040-1045 Trans Tech Publications, Switzerland, © 2014. <https://www.scientific.net/AMM.462-463.1040>
- [12] Taghiyev, N., Akcay, M. (2013). Parallel matrix multiplication for various implementations, Application of Information and Communication Technologies (AICT), 2013 7th International Conference on, p. 1-5. IEEE Conference. <http://ieeexplore.ieee.org/document/6722636/>
- [13] Malaya, Nicholas., Che, Shuai., Joseph., Greathouse, L., Ren´e van Oostrum, Michael, Schulte. J. (2017). Accelerating Matrix Processing with GPUs. Published in the Proceedings of the 24th IEEE Symposium on Computer Arithmetic (ARITH 24), (July). <http://ieeexplore.ieee.org/document/8023078/>
- [14] Michailidis, P., Margaritis, K. (2010). Performance Models for Matrix Computations on Multicore Processors using OpenMP. *The 11th Intl. Conf. on Parallel and Distributed Computing, Applications and Technologies*. <https://pdfs.semanticscholar.org/84c6/6efe3a8f50f8b857ff9bee73167093926bc2.pdf>
- [15] Nvidia corporation, “CUDA C Programming Guide” PG-02829-001_v9.0 | September 2017, <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.

[16] <https://www.nvidia.com/en-us/geforce/products.html>.

[17] <http://graphics.stanford.edu/data/3Dscanrep/.html>