GPGPU Acceleration for Skeletal Animation – Comparing OpenCL with CUDA and GLSL\*

Shousheng Liu, Ge Chen\*, Chunyong Ma, Yong Han

*College of Information Science and Engineering, Ocean University of China, Qingdao 266100, China*

**Abstract**

The existing matrix palette algorithms for skeletal animation are accelerated by the technique GPGPU based on GLSL or CUDA. Because GLSL is extended from graphics library OpenGL, it couples the rendering and calculations together closely and forces itself not convenient to reuse, meanwhile CUDA is designed only for NVIDIA GPUs. In this paper GPGPU based on OpenCL is proposed for accelerating skeletal animations. OpenCL brings portability both on software and hardware.The experimental results show that the parallel scheme based on OpenCL can run on GPUs from AMD and NVIDIA. And the speedup is comparable with CUDA or GLSL.

*Keywords*: Skeletal animation; GPGPU; OpenCL; CUDA; GLSL

Introduction[[1]](#footnote-2)

The deformation model we study in this thesis is known as skeletal animation (or skinning, skeletal subspace deformation, matrix palette skinning or simply enveloping) as demonstrated [1]. Skeletal animation has been proposed for the animation of human or animal, which is widely applied in the area of Medicine, Movies and Games. The early computer animation is based on vertex mixing between key frames. Every key frame needs a model, so a period of animation needs plenty of models, which occupy huge memory and storage space. For sake of saving memory and storage space, Burtnyk [2] proposed the concept of skeletal animation in 1976. Skeletal animation needs just one model, whose movement is generated by the skeleton inside of the model. Comparing with the animation of vertex mixing, skeletal animation is a new method to save memory and storage space at the cost of computing time, which brings a new problem – the performance of computing.

The parallel technology has been developing fast both on software and hardware, especially on GPU. The parallel computing on GPU (named General Purpose GPU, GPGPU) goes through 3 generations including assembly program, shading language and CUDA. Lindholm [3] proposed the skeletal animation on GPU based on assembly language by an OpenGL extension named nv\_vertex\_program in 2011. JI improved the parallel algorithm of skeletal animation based on Direct3D shader as demonstrated [4], then HU went deeply accelerating skeletal animation by CUDA as demonstrated [5]. As soon as the GPU technology broke the performance bottleneck, several researchers extended the technology of skeletal animation, such as generating skeleton automatically as demonstrated [6], binding the skin and skeleton automatically as demonstrated [7], grouping vertex transformation into simplex pieces as demonstrated [8].

The early GPUs are designed for processing graphic tasks. Researchers have to map the general scientific problems to graphic ones. The technology which works general problems on GPU is called GPGPU, the language is Shading Language including Vertex Shader and Fragment Shader and other shaders. The typical languages are OpenGL Shading Language, High Level Shading Language based on Direct3D and NVIDIA Cg(C for Graphic). This paper applies OpenGL as graphic hardware API. In the pipeline of OpenGL there are 2 stages responsible for executing vertex transformation and processing fragment color, which named Vertex Program and Fragment Program. In the beginning, the function of these two stages is fixed without API for programmer. Then the assembly language and specific domain language are developed to program the pipeline of OpenGL. Each GPU provider has its own different language until official organization of OpenGL called Khronos Group published the standard extension named ARB\_vertex\_program and ARB\_fragment\_program from Architecture Review Board in 2002. Comparing the low level assembly language GLSL(OpenGL Shading Language) is a new language like C, which is developed by OpenGL ARB to replace the former ARB assembly language.

CUDA, full named by Compute Unified Device Architecture is a general parallel computing architecture designed by NVIDIA specifically for its own GPUs. Unlike the former mentioned shading language, CUDA is independent on the graphic API like OpenGL or Direct3D and need not to map general problems to graphic ones. Comparing the Cg language who declared as C for graphic, CUDA language implemented C more entirely. CUDA supports memory pointer in GPU memory, which helps programmer to treat the data storage of GPU device just like the host memory, including memory allocation and free. CUDA can run on all series of NVIDIDA GPUs after G8x, including GeForce, Quadro and Tesla series. The first CUDA GPU is GeForce 8800 GTX in 2006 which equipped by 128 cores and 86 GB/s of memory bandwidth and 518 Gflops of computing ability. The latest GPU of GeForce set with best performance is GeForce GTX Titan in 2013.

As soon as the Open Computing Language (OpenCL) was proposed, the main GPU providers of AMD and NVIDIA supported OpenCL on their GPUs. OpenCL is a opening set of standard specifications as demonstrated [9], the program which was programmed according the standard could be executing across multiple processor including CPUs, GPUs, DSPs, FPGAs and other processors. Each kind of processor released their software developing toolkit, including Intel SDK for OpenCL Application, AMD APP, NVIDIA OpenCL within CUDA. OpenCL supplies a particular language based on C so there are plenty of advantages comparing other parallel schemes as demonstrated [10].OpenCL also supplies a set of API called by the host to deploy the environment for OpenCL.

At the aspect of assessment theory for parallel performance, Amdahl proposed a law to calculate the parallel speedup for fixed problems and the law was named by Amdahl’s Law as demonstrated [11]. The law defined the ceiling of speedup is 1/(1-P), while the P is the rate of parallel part, the S is the local speedup of parallel part, the A is the last speedup. The formula of Amdahl’s Law to calculate the speedup is as following:

(1)

According to the formula (1), theoretical peak speedup could be assessed for the skeletal animation after parallelism. Supposed the local speedup S has grown to infinity ∞, the theoretical peak speedup will reach 1/(1-P). In this paper, vertex calculation on CPU is the bottleneck which consumes 80% of the total time so P is equal 0.8, according the formula (1) the theoretical peak speedup is equal 5.

In this paper we study the parallel algorithm on GPU based on OpenCL, comparing to the traditional parallel schemes based on GLSL and CUDA.

1. Serial Algorithm of Skeletal Animation

The procedure of rendering of skeletal animation could be divided into four steps and very step is a stand-alone module. Those steps are as following: 1)To parse the bone matrices and vertex coordinates for the skeletal animation; 2)to update the matrices of bones; 3) To update vertex coordinate by matrices; 4)To output the vertex coordinate to form the last moving animation.

The third step is to update the vertex, which consumes most of the rendering time. This step is the bottleneck of skeletal animation. In this paper we apply the algorithm of matrix palette to update the vertex, the flat chat of the algorithm is as figuration 1. The description of the algorithm is as following:

1) To read a vertex coordinate (*x,y,z*), the list mL stored M matrices of bones. Each vertex is bound to B bones, and the index and weight for each bone is *d(i)*，*w(i)*，*i*=1,2…B;

2) To calculate the mixing matrix m for each vertex, *m* is a matrix formed by 4\*4 elements, which stand by the information of rotation , transformation and scale. The matrix m is accumulated by multiple matrices of bones bound to the vertex, the formula to accumulate the matrix is as following:

*m* = (2)

3) To transform the vertex coordinate by the matrix as the classic transformation formula.

As the algorithm complexity is not the same, the time distribution of each sub-module will be different, and the size and location of the bottleneck will be different. This article adopts the vertex number of skeletal animation is 100,000. The number of bones binding to single vertex is 2, and the CPU is Intel i7 3770k.

By measuring the time distribution of the algorithm modules as you can see, the proportion of the biggest one of the modules is 84%, and that it is the performance bottleneck of the algorithm. Another module of skeleton algorithm is complex but only cost a tiny percentage of time, it is no need to be optimized. Before bottleneck positioning and parallel optimization, serial optimization is implemented. The algorithm needs to be made more in line with the characteristics of the parallel computing. Preliminary improvement of the algorithm makes the serial code itself to obtain optimal performance, which is taken as the base of the parallel optimization. After optimization for the serial code, time ratio of calculation for vertices is 81%, which is still the performance bottleneck of the algorithm.



Fig. 1: Flow Chat of Vertex Updating and Subprogram Matrix Palette Algorithm

1. Parallel Algorithm based on GLSL

Shading language programming is a powerful tool to transform the default rendering pipeline operation process to be programmable. On the one hand it can strengthen the original graphics features and on the other hand it can be used to handle general problems in the field of general-purpose computing. Skeletal animation algorithm in this paper needs to be updated vertex coordinates dynamically, which is a case to use shading language to strengthen graphics functions, but the indexing of matrix can also be counted as general computing as well as the accumulation of weighted matrix. In this paper, we use GLSL language for rendering skeletal animation including vertex updating process.

The first step is to achieve single bone situation. Bone matrix indexes exist in the fourth vertex components namely w. The shading program only needs a custom parameter – the matrix array of bones namely *matrix*. Given the matrix array for the vertex shader are shared by all vertices, in this paper uniform will be the selected type of matrix parameters. Assignment is required before uniform parameters are used on the device, which includes initialization of binding relationship and updating repeatedly. Initialization of binding is performed through the API interface glGetUniformLocation and to use glUniformXXX to realize updating. The uniform parameter of the algorithm in this paper - matrix needs to be performed the initialization of value assignment and updating assignment. In the process of initialization the variable name of matrix is transferred to glGetUniformLocation and the variable position is tagged. In the process of assignment data in host memory for matrices locationUniformMatrix is sent to the location specified. Special attention needs to be paid to the length of the parameter array, in vertex shader the length of array must to be defined as fixed constant number. Given the number of general computer animation bone is no more than 100, so in this article we define the size of matrices is 100.

Above we work out single bone of skeletal animation based on GLSL, in the next part of the paper we continue to achieve more bones for skeletal animation. In the past a single bone index was stored in the w component vertex data, now multiple bones require multiple indexes and weights, and each vertex needs these data. At this moment we need to set index and weight data as two parameters to the vertex shader, in this article index and weight parameter type will be selected as attribute. As the matrix of data before, the position binding and value assignment are also needed for the weights and indexes. In addition, the shader also needs to know the number of the bones, for the number of bone binding a uniform variable. On the basis of single bone GLSL algorithm, another three new variables are appended which also need to be assigned on the outside of the shader. Among them the type for each vertex binding number of bones is uniform, whose value is assigned as the same method as the above variable. Bone index and weight variables are belonging to the attribute type, the API interface to bind location of variables is glGetAttribLocation, and the assignment API is glVertexAttribPointer after glEnableVertexAttribArray called firstly. To attribute property variable assignment, there are two kinds of methods, the first is to transfer from the host CPU memory in each frame as designated vertex coordinates. The second method is to attribute values which are not changed often as OpenGL VBO, where the assignment is completed directly.

1. Parallel Algorithm based on CUDA
   1. Basic CUDA without optimization

Compared with shading language, CUDA is independent on the graphics library so it is ease of use and has portability of the programming language with many other advantages as demonstrated [12]. This chapter is about skeletal animation algorithm based on CUDA. The kernel code is as follows:

\_\_global\_\_ void

transformVectorByMatrix4One( const Vector4 \*pInput, const int \*pIndex, Vector4 \*pMatrix, Vector4 \*pOutput, int sizeMax, const float \*pWeight)

{// struct Vector4 { float x,y,z,w};

Vector4 pIn = pInput[index]; // index is thread id

Vector4 px = make\_Vector4(pIn.x, pIn.x, pIn.x, pIn.x) ; // y, z also as x

int offset = pIndex[index]\*4 ;

Vector4 m0 = pMatrix[offset+0] ; // m1/2/3 also as m0

pOutput[index] = px \* m0 + py \* m1 + pz \* m2 + m3 ;

}

While the kernel function design is finished, configuration of kernel arguments and the thread structure is the next step. Then the host calls the kernel function which codes the instructions for the CUDA GPU device to perform; after the execution the kernel returns processing result for the host or other equipment. The details of the argument configuration, thread structure and calls to the kernel in the link of the three conventional core parts are as below:

* Arguments configuration

The host provides real parameters for CUDA Kernel according to the structure of the memory space. The host sends data from the its memory to the graphics card memory for the kernel to call. We set the parameters include the initial vertex coordinates, transformed vertex coordinates, weight and index of bone matrix. The vertex coordinates need to be done by the OpenGL rendering after the transformation. CUDA supports interoperability with OpenGL so they can share data. Shared data for OpenGL and CUDA is VBO (Vertex Buffer Object). We generate the VBO through CUDA cudaGraphicsGLRegisterBuffer which is the main interoperability interface function, then bind CUDA graphic resources using cudaGraphicsResource, making CUDA can modify the OpenGL rendering data.

* Thread structure design

The setting principle of thread block structure is: the number of threads contained in thread block is multiple of 32, general is set to 256. Thread grid structure setting principle is: thread blocks can be divided into dynamic and static which is determined according to the required number of threads. If each thread processes one element, the total number of threads is the number of data elements, the thread blocks is equal to the result that the number of elements divides the number of threads in thread block. Another method is to keep the thread blocks static for example to 64 when each thread is responsible to handle multiple elements. Specifically the number of elements for each thread is determined by the number of elements, the numerical value is equal to the number of element divided by the number of thread blocks and the number of threads in each thread block.

* Call to kernel

The function transformVectorByMatrix4One is designed for above the kernel function. The grid and block is the above thread structure parameters; The symbol d\_pInput and others respectively are the parameters such as initial vertex coordinates, transformed coordinates, weight and index of bone matrix. Full typical code is as follows:

transformVectorByMatrix4One<<< grid, block >>>( d\_pInput, d\_pIndex, d\_pMatrix, d\_pOutput, sizeMax, d\_pWeight );

* 1. Optimizations for CUDA

CUDA Programming model has the following five aspects of important features of the design: the core function kernel, thread hierarchy, memory hierarchy, heterogeneous programming and compute capability. In addition to the need to write a kernel function based on CUDA GPU programming, we need to do corresponding parameters optimization for CUDA features as demonstrated [13].

### Memory alignment

Above we use Vector4 as vertex data structure, defined as struct Vector4 {float x, y, z, w;}. According to the CUDA specification this kind of structure does not conform to the requirements of the memory alignment structure of CUDA, the programmer cannot achieve the optimal bandwidth to read and write memory. So it is necessary to align the data structure as we insert "\_\_builtin\_align\_\_ (16)" in the middle of the struct and Vector4. And the definition of Vector4 becomes struct \_\_builtin\_align\_\_ (16) Vector4 {float x, y, z, w;}; CUDA actually has built a data structure float4 equivalent to the aligned Vector4 so this article will replace the customed unaligned data structure Vector4 to float4 the aligned data structure so as to meet the CUDA memory alignment specification. Alignment optimization is oriented to custom data structure such as typen in which the type is float or int and n is 2, 3, 4; Basic data structure does not need to be aligned, such as: float or int. Although CUDA also provides float1 or int1 the aligned type, but there is no effect on performance.

### Processor utility rate

Processor utilization depends on the ratio of the active threads within a multiprocessor and the maximum number of threads or the ratio of the current activated warp number and the maximum warp number. Active threads are limited by number of registers within each MP. The active number of threads is the result when the total number of registers divides the register number needed for each thread. Because of MP register resources is fixed when the needed register number for each thread increases so the active threads number. At last the processor utilization decreases.

In order to enhance the processor utilization, it is necessary to optimize program to save register usage and properly to allocate the number the dimensions of thread block. The current kernel register usage can be obtained by NVIDIA CUDA attached tools Visual Profiler. In addition to register usage of the algorithm shared memory usage will also affect the activation of the number of thread blocks, thus affecting the processor utilization.

### Coalesced access

Global memory involves in three segments of memory. The first segment is input and output vertex coordinates. The second segment is indexes and weights of multiple matrices. The third segment is values of matrices. For the first segment when each thread processes one element it satisfies the condition of coalesced access. When each thread is responsible to handle multiple elements continuous accession of memory within one thread will destruct the condition for adjacent threads within a warp to access adjacently.

The second segment is indexes and weights for matrices. Each thread continues to read multiple values for each vertex at a time. It meets the condition of coalesced access so the performance improves 10%. The third segment is the values of matrices. Multiple current storage structure of the matrices for each thread are continuous, and three lines quad within matrix the three flaot4 is continuous. The continuity will be broken between threads which does not meet the condition of coalesced access. Information will be needed to split by unit of float4 for each vertex matrix then the AOS (array of struct) structure is transformed into SOA (struct of array) structure. So the third segment meets the conditions of coalesced access.

### Constant memory for caching matrix

In the third OpenCL memory we store matrix values which are the equivalent of a data table randomly accessed by all threads. This case is suitable for storing in constant memory. Constant capacity is 64 k bytes and the amount of memory can hold about 1000 bone matrix of data while the actual number of bone is about 100. So constant memory can accommodate bone matrix values completely. Through cache ability of constant memory we shorten the kernel's time so as to improve the performance kernel. Constant memory allocation must be in global scope and memory size also must be constant. In this paper we allocate memory for 100 matrices. Constant memory assignment is achieved through CUDA host API cudaMemcpyToSymbol before invoking the kernel. The use of constant memory is the same as to use common array in CPU through memory pointer or array index.

### Shared memory for caching matrix

Shared memory has four differences with constant memory. Firstly the sharing scopes are different. Shared memory is shared by threads in one thread block but constant memory is shared by all of the threads. Secondly the memory allocations are different. Shared memory supports dynamic allocation but constant memory must be static. Thirdly the defining positions of variables are different. Shared memory is defined in the kernel function but constant memory is defined in global area outside all the functions. Fourthly storage locations are different. Shared memory is on chip near parallel processing units but constant memory is located off chip.

1. Parallel Algorithm based on OpenCL

OpenCL has more broader portability than CUDA and GLSL, whose performance could be comparable while fully optimized as demonstrated [14, 15].OpenCL must install the device drivers and software development kit, this article selects the NVIDIA CUDA, including OpenCL SDK. Because OpenCL is an open standard, the same system can be installed more set of OpenCL drivers and SDK, such as Intel OpenCL, AMD OpenCL, NVIDIA OpenCL. The standard API function clGetPlatformIDs can obtain the number of drivers which have been installed and the detailed information of each driver. If the number of 0 it means there is no driver installed for OpenCL; When the number is not zero, according to the need we choose a specific set of OpenCL, for example in this paper we choose "NVIDIA CUDA". Before executing OpenCL the programmer must initialize the OpenCL runtime environment. There is few difference in initialization for different projects. What type of driver for OpenCL, whether CPU or GPU, which cl file is specified, and what is name of kernel function. While initialization is completed, the next step is to design the kernel function whose process can be referenced to SSE algorithm version.

* 1. General OpenCL

Firstly we achieve the algorithm with single bone, when each vertex is associated only to one piece of bone. So the coordinate transformation by matrix is executed only once. The OpenCL kernel function is as follows:

\_\_kernel void

transformVectorByMatrix4(\_\_global float4 \*pIn, \_\_global int \*pIndex,\_\_constant float4 \*pMatrix,\_\_global float4 \*pOut)

{

size\_t index = get\_global\_id(0) + get\_global\_id(1) \*get\_global\_size(0);

int offset = pIndex[index]\*4;

float4 pIn = pIn [index];

float4 xxxx = (float4)(pIn.x, pIn.x, pIn.x, pIn.x);//y, z also as x

pOut [index] = xxxx \* pMatrix[offset+0] + yyyy \* pMatrix[offset+1] + zzzz \* pMatrix[offset+2] + pMatrix[offset+3];

}

For each vertex is associated multiple bones, when we obtain the transformation matrix we need index multiple matrices then sum them according to the weight. So we need to add index and weight for matrix in the kernel parameter list, in imitating the SSE kernel function body to sum the weighted matrix. Before execution, besides the instruction code for the specified OpenCL device to call, we still need to specify the worker thread structure – workgroup and workitem, including Global WorkSize and Local WorkSize. They support 2 to 3 dimensions, and generally we just set Global WorkSize whose maximum number is different by different device. The value that each dimension supports could be detected through clGetKernelWorkGroupInfo dynamically, and the total value is the product of each dimension which must greater than the problem number. This paper uses the two-dimensional structure of thread, the first dimension is set to the maximum, the second dimension is set by the result when the total number of vertices divided the first dimension; and the Local WorkSize uses the system default settings.

* 1. Inter-operability with OpenGL

In order to optimize the skeleton animation rendering and the efficiency of data transmission, this paper uses the OpenCL interoperability features with OpenGL which share the vertex cache object in graphic memory such as Vertex Buffer Object (VBO). In order to support memory sharing and interoperability, OpenGL and OpenCL need to do the corresponding adjustment at both ends. In OpenGL side, we need to change the OpenGL rendering data storage style. Original style is stored immediately, which is in the memory of the existing host, every time while rendering the data need to be sent to OpenGL device which is in a state of out of control and not be able to be accessed and modified. Now with the Vertex Buffer Object (VBO), data have been stored in OpenGL device always so it can be read and write access through VBO buffer resource ID. In this article we make vertex data of skeletal animation stored in VBO, at the same time while we create VBO we call clCreateFromGLBuffer to map a OpenCL memory cl\_mem. So the VBO and cl\_mem can share the vertex data. When the vertex needs to be updated by OpenCL, through clEnqueueAcquireGLObjects we can access to the data. Afterwards modification is completed through clEnqueueReleaseGLObjects then the result is returned to the VBO.

By above renovation for OpenGL and OpenCL, the algorithm performance is improved respectively from two aspects. On the one hand, makes the OpenGL and OpenCL can share data, which realized bidirectional conversion between the OpenGL VBO and OpenCL memory. Thereby we can save the time of data transmission between OpenGL and OpenCL; On the other hand, because of the VBO rendering we improve the rendering performance dramatically.

1. Experimental Animation Simulation
   1. Comparison among multiple parallel schemes

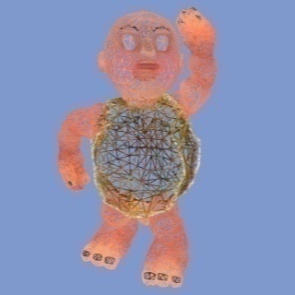
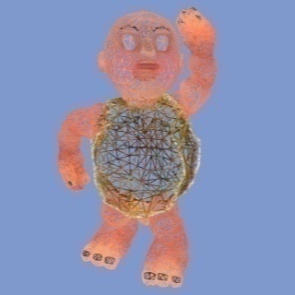
In this paper, the specifications of the skeletal animation data are as follows: the vertex number is 0.1 M, the number of bones is 77; the number of bones binding to each vertex is set to 2. Experimental platform configuration is as follows: the GPU is NVIDIA Geforce GTX 670, the CPU is Intel i7 3770k. Skeletal animation is rendered as shown in figure 2.

Fig. 2: A 3D character animation -- Little Tortoise

Based on the serial version algorithm named palette serial matrix skeletal animation we design 4 parallel versions of the algorithm for the GPU, one of them is based on GLSL and one based on CUDA, the left two are based on OpenCL which are not optimized and optimized. And comparing to 1 set of scheme based on OpenCL for CPU.

As shown in figure 3 (left), 5 sets of parallel algorithm is run on the selected data and platform, then we get statistics of speedup of 5 set of parallel schemes. From the above experiment results we are informed that: 1) the GLSL algorithm gets the best optimal performance, the optimized OpenCL is second, non-optimized OpenCL is minimum;2) the OpenCL performance is between CUDA and GLSL, OpenCL is 2 times of CUDA, 4% lower than the GLSL.

* 1. Parallel portability

In this paper, we study the portability space limited to different brand or different serials of GPU processors. Among the three GPGPU parallel technologies, only OpenCL has portability across different GPU. We selected three groups of hardware for experiment, in which 2 are the same brand with different serials of NVIDIA GPU, plus 1 AMD GPU. The three GPUs manifest the portability across hardware brands and serials.

As shown in figure 3 (right) the result of the experiment, on all three GPUs with different brands or different serials the speedup of OpenCL version of skeletal animation rendering algorithm is 3.4 - 9.1. We conclude that OpenCL has good function portability and performance portability. And besides a low GPU, the performance of GPUs is generally higher than the CPU also based on OpenCL.

Fig. 3: Speedup comparison among 4 parallel algorithms and 3 GPUs

* 1. Comparison among different complex dataset

The data complexity is determined by the following two parameters, including number of bones bound to each vertex and number of vertex within a single model. In this paper we respectively set the two parameters with four level of complexity, range of 1 to 4 is for the bone number and the vertex number range is 25k to 1600k. The two parameters interweave together 4 x 4 matrix, a total of 16 groups of data of different complexity. Under 16 groups of data we measure the acceleration ratio of OpenCL, and calculate the increase rate of speedup based on OpenCL comparing with CUDA and GLSL.

As shown from the experimental data in table 1, OpenCL parallel algorithm comparing serial algorithm has absolute acceleration performance with speedup range between 1.4 and 21. As concluded from table 2, the performance of the OpenCL is between CUDA and GLSL. For the case of the bone number greater than 1 and the vertex number more than 25 k of, performance of skeletal animation based on OpenCL declines within 30% of GLSL and grows between 20% and 170% by CUDA.

Table 1: Speedup of OpenCl of 16 dataset

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Vertex  No. No.  of Bone | 25k | 100k | 400k | 1600k |
| 1 | 1.4 | 2.5 | 7.4 | 7.4 |
| 2 | 4.1 | 9.2 | 14 | 18 |
| 3 | 4.2 | 8.7 | 15 | 20 |
| 4 | 5.3 | 12 | 16 | 21 |

Table 2: Increase rate of speedup based on OpenCL comparing with CUDA and GLSL

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Vertex  No. No.  of Bone | 25k | 100k | 400k | 1600k |
| OpenCL  vs  CUDA | 1 | 21% | 107% | 142% | 172% |
| 2 | 21% | 117% | 82% | 122% |
| 3 | 11% | 28% | 76% | 98% |
| 4 | 14% | 51% | 45% | 77% |
| OpenCL  vs  GLSL | 1 | -70% | -71% | -6% | -14% |
| 2 | -67% | -4% | -27% | -28% |
| 3 | -68% | -17% | -24% | -20% |
| 4 | -65% | 0% | -22% | -17% |

1. Conclusion

The existing matrix palette algorithms for skeletal animation are accelerated by the technique GPGPU based on GLSL or CUDA. Because GLSL is extended from graphics library OpenGL, it couples the rendering and calculations together closely and forces itself not convenient to reuse, meanwhile CUDA is designed only for NVIDIA GPUs. In this paper GPGPU based on OpenCL is proposed for accelerating skeletal animations. The experimental results show that the parallel scheme based on OpenCL can run on GPUs from AMD and NVIDIA. And the speedup is comparable with CUDA or GLSL.

**参考文献：**

[1] L. Kavan, Real-time Skeletal Animation, M.S[PH.D.] Thesis, Faculty of Electrical Engineering Department of Computer Science and Engineering, Czech Technical University, June 2007

[2] N. Burtnyk, M. Wein, Interactive skeleton techniques for enhancing motion dynamics in key frame animation, Communications of the ACM 19(1976) 564-569

[3] E. Lindholm, M.J. Kilgard, H. Moreton, A user-programmable vertex engine, in: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, 2001, pp. 149-158

[4] Z.E Ji, J.Q. Zhang, Programmable GPU Technology in skelotal animation, Computer Engineering and Applications 44(2008) 70-80

[5] Q.L. Hu, B.F. Chen, Real-time Shadow of Skeletal Animation Based on CUDA, Journal of Chinese Computer Systems 32(2011) 165-168

[6] D.L. James, C.D. Twigg, Skinning mesh animations, ACM Transactions on Graphics (TOG) 24(2005) 399-407

[7] I. Baran, J. Popovi C, Automatic rigging and animation of 3d characters, ACM Transactions on Graphics (TOG) 26(2007) 72

[8] H.B. Yan, S.M. Hu, R.R. Martin, et al, Shape Deformation Using a Skeleton to Drive Simplex Transformations, IEEE Transactions on Visualization and Computer Graphics 14(2008) 693-706

[9] Khronos, The OpenCL Specification, http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, 2012

[10] P. Du, R. Weber, P. Luszczek, et al, From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming, Parallel Computing 38(2012) 391-407

[11] X. H. Sun, Y. Chen, Reevaluating Amdahl’s law in the multicore era, Journal of Parallel and Distributed Computing 70(2010) 183-188

[12] T. Ivanovska, L. Linsen, H.K. Hahn, et al, GPU implementations of a relaxation scheme for image partitioning: GLSL versus CUDA, Computing and visualization in science 14(2011) 217-226

[13] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, et al, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, 2008, pp. 73-82

[14] J.B. Fang, A.L. Varbanescu, H. Sips, A Comprehensive Performance Comparison of CUDA and OpenCL, in: International Conference on Parallel Processing (ICPP), 2011, pp. 216-225

[15] R. Amorim, G. Haase, M. Liebmann, et al, Comparing CUDA and OpenGL implementations for a Jacobi iteration, in: High Performance Computing & Simulation (HPCS'09), 2009, pp. 22-32

1. \*Project supported by the National Nature Science Foundation of China (No. 61361136001), the National Ocean Office Opening Foundation of China (No. KLDO201303), the National Industry Information Office Developing Foundation of China (No. Finance[2012]407)

   \*Corresponding author.

   Email address: gechen@public.x.x [↑](#footnote-ref-2)