Journal of Computational Information Systems 10: 1 (2014) 1–9
Available at http://www.Jofcis.com

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

1 Introduction

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. General-purpose computing on graphics processing units (abbreviated: GPGPU) has

[⋆]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.qd.sd.cn (Ge CHEN).

gone through four generations including assembly program, shading language, CUDA, OpenCL. Lindholm [3] proposed the skeletal animation on GPU based on assembly language by an OpenGL extension named nv_vertex_program in 2001. JI improved the parallel algorithm of skeletal animation based on Direct3D shader in 2008 as demonstrated [4], then HU went deeply accelerating skeletal animation by CUDA in 2011 as demonstrated [5]. As soon as the GPU technology broke through 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 if they want to solve general scientific problems by GPU. The technology which solves general problems on GPU is called GPGPU whose programming language are assembly language and shader language. Typical shader languages are GLSL (OpenGL Shading Language), HLSL (High Level Shading Language) based on Direct3D and Cg (NVIDIA C for Graphic). We select GLSL because this paper applies OpenGL as graphic hardware API. There are 2 stages responsible for executing vertex transformation and processing fragment color which are named vertex program and fragment program in the pipeline of OpenGL. In the beginning, the function of these two stages was fixed without API for programmer. Then the assembly language and specific domain language were developed to program the pipeline of OpenGL. Each GPU provider had 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.

Compute Unified Device Architecture (abbreviated: CUDA) 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[9]. 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 NVIDIA 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.

As soon as the Open Computing Language (abbreviated: 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 [10], the program which is programmed according the standard could be executing across multiple processors including CPUs, GPUs, DSPs, FPGAs and other processors. Each kind of processor has 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 [11]. OpenCL also supplies a set of API called by the host to deploy the environment for OpenCL.

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

2 Serial Algorithm of Skeletal Animation

The procedure of rendering skeletal animation could be divided into four steps and every 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 coordinates by matrices; 4)To output the vertex coordinates 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 description of the algorithm is as following:

(1) To read a vertex coordinate (x,y,z). Supposed the list mL stored M matrices of bones. Each vertex is bound to B bones from list mL, and the index and weight for each bone is d(i) and 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 Eq. (1):

$$m = \sum_{i=0}^{B-1} (mL[d(i)] * w(i)) \tag{1}$$

(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, the CPU is Intel i7 3770k and the main GPU is NVIDIA GeForce GTX 670.

By measuring the time distribution of the algorithm modules we found that the proportion of the biggest one of the modules was 84%, and obviously it was the performance bottleneck of the algorithm. Before bottleneck positioning and parallel optimization, serial optimization is implemented. The algorithm had to be made more in line with the characteristics of the parallel computing. Preliminary improvement of the algorithm made the serial code itself to obtain optimal performance, which was taken as the basis of the parallel optimization. After optimization for the serial code, time ratio of calculation for vertices was 81%, which was still the performance bottleneck of the algorithm.

3 Parallel Algorithm based on GLSL

Lindholm [3] proposed the skeletal animation on GPU based on assembly language by an OpenGL extension named nv_vertex_program in 2001. JI improved the parallel algorithm of skeletal animation based on Direct3D shader in 2008 as demonstrated [4]. In this paper, we use a vertex shader of 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 one custom parameter - the matrix array of bones. Given the matrix array for the vertex shader are shared by all vertices, in this paper uniform will be the selected type of matrix variable. 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 of vertex data, now multiple bones require multiple indexes and weights, and every vertex needs these data. At this moment we need to set index and weight data as another two parameters to the vertex shader, in this article index and weight variable type will be selected as attribute.

4 Parallel Algorithm based on CUDA

4.1 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]. HU went deeply accelerating skeletal animation by CUDA in 2011 as demonstrated [5]. The kernel definition was designed as following:

__global__ void transformVectorByMatrix4One(const Vector4 *pInput, const int *pIndex, Vector4 *pMatrix, Vector4 *pOutput, int sizeMax, const float *pWeight) { }

While the kernel function design had been done, configuration of kernel arguments and the thread structure were the next steps. Then the host called the kernel function which distributed the instructions to CUDA device to perform; after the execution the kernel returned the processed result for the host or other equipment.

- Arguments configuration
  The host provides real parameters for CUDA Kernel according to the structure of the memory space. CUDA supports inter-operability with OpenGL so they can share data. The shared data for OpenGL and CUDA is stored in VBO (Vertex Buffer Object). We generate the VBO through CUDA cudaGraphicsGLRegisterBuffer which is the main inter-operability interface function, then bind CUDA graphic resources using cudaGraphicsResource, so CUDA can modify the OpenGL rendering data on the fly.

- 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.

- 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);

4.2 Optimizations for CUDA

Beside of a basic kernel function we need to do corresponding parameters optimization for CUDA features including memory alignment, coalesced access, constant or shared memory to cache as demonstrated [13].

4.2.1 Memory alignment

We used Vector4 as vertex data structure, defined as struct Vector4 {float x, y, z, w;}. This kind of structure did not conform to the requirements of the memory alignment structure of CUDA, the programmer could not achieve the optimal bandwidth to read and write memory. So there was necessary to align the data structure as we inserted __builtin_align__ (16) in the middle of the struct and Vector4. And the definition of Vector4 became struct __builtin_align__ (16) Vector4 {float x, y, z, w;}; CUDA actually had built a data structure float4 equivalent to the aligned Vector4.

4.2.2 Coalesced access

The global memory for input and output of vertex coordinates satisfied the condition of coalesced access. When each thread was responsible to handle multiple elements, continuous access of memory within one thread would destruct the condition for adjacent threads within a warp to access adjacently. The values of matrices would 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.

4.2.3 Shared memory for caching matrix

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

5 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. In this article we selected the NVIDIA OpenCL included in CUDA 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. While initialization for OpenCL is finished the next step is to design the kernel function whose process can be referenced to CUDA algorithm version.

5.1 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 are similar to 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 to 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.

5.2 Inter-operability with OpenGL

In order to optimize the skeleton animation rendering and the efficiency of data transmission, this paper uses the OpenCL inter-operability 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 inter-operability, 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 has been stored in OpenGL device always so it can be read and written 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. After modification is finished we call clEnqueueReleaseGLObjects, then the updated data is written back to the VBO.

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

6 Experimental Animation Simulation

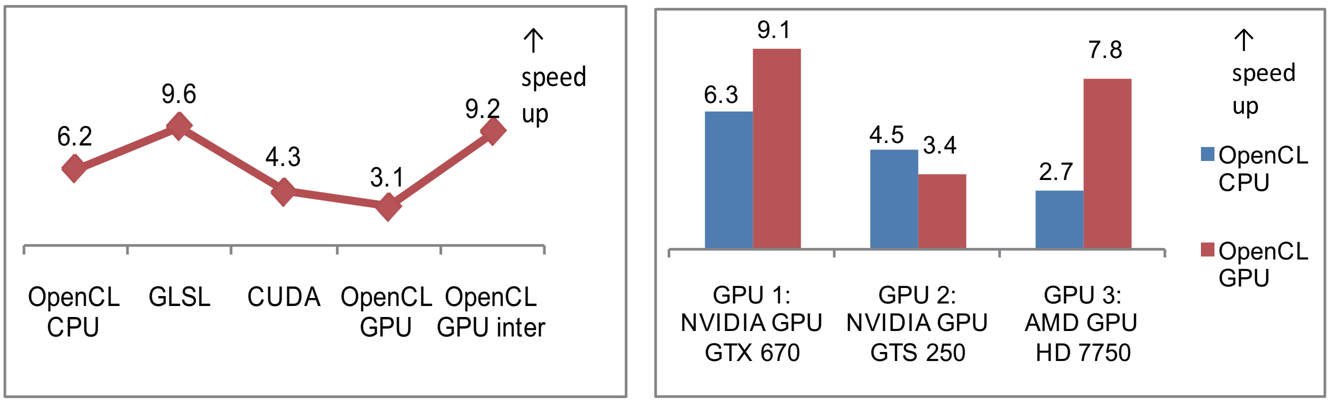

Fig. 1: Speedup comparison among 4 parallel algorithms and 3 GPUs

6.1 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.

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 Fig. 1 (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 experimental 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.

6.2 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 two are the same brand with different serials of NVIDIA GPU, plus 1 AMD GPU. The three GPUs reveal the portability across hardware brands and serials.

As shown in Fig. 1 (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 between 3.4 and 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.

Table 1: Speedup of OpenCL of 16 dataset

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

6.3 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 bone number is 1 to 4 and range of vertex number 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 speedup of OpenCL, GLSL and CUDA, and calculate the increase rate of speedup based on OpenCL comparing with CUDA and GLSL.

As shown in Table [1], 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 increase between 20% and 170% by CUDA.

7 Conclusion

GPGPU has gone through four generations including assembly program, shading language, CUDA, OpenCL. In this paper we study the parallel algorithm for skeletal animation on GPU based on OpenCL, comparing to the traditional parallel schemes based on GLSL and CUDA. 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.

References

[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] S.L. Zhang, Q.S. Zhu, J. Liu, et al, Parallel Interpretation of L-system Based on CUDA, Journal of Computational Information Systems 9(2013) 415-424

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

[11] 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

[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