骨骼动画GPGPU加速——OpenCL对比CUDA与GLSL

刘寿生, 陈戈, 马纯永, 韩勇

（中国海洋大学信息科学与工程学院，青岛 266100）

**摘 要**：现有GPGPU骨骼动画矩阵调色板算法基于2种API——图形库OpenGL的扩展接口GLSL以及NVIDIA CUDA，其中GLSL因渲染和计算紧密耦合在一起不利于算法的移植复用，CUDA仅限于NVIDIA品牌的GPU。本文为骨骼动画提出基于OpenCL技术的GPGPU加速算法，提升软件和硬件可移植性。实验结果显示，OpenCL可以同时适用于AMD和NVIDIA品牌的GPU，并且OpenCL的加速比与CUDA和GLSL相当。

**关键词**：骨骼动画; GPGPU; OpenCL; CUDA; GLSL

中图分类号：TP391.9 文献标识码：A 文章编号：1004-731X (2014) xx-xxxx-x

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

LIU Shou-sheng，CHEN Ge，MA Chun-yong，HAN Yong

(Ocean University of China, Qingdao, 266100, China)

**Abstract**：Skeletal animations are accelerated by the technique GPGPU based on GLSL or CUDA. The existing skeletal animation matrix palette algorithm based on GLSL extended from graphics library OpenGL, which couples the rendering and calculations together closely and forces itself not convenient to reuse. CUDA is designed only for NVIDIA GPUs. 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 for the algorithm of skeletal animation.

**Key words**: skeletal animation; gpgpu; opencl; cuda; glsl

引言[[1]](#footnote-2)

骨骼动画是一种形变模型，在学术上也被称为网格蒙皮、骨骼子空间形变、矩阵调色板蒙皮[1]。骨骼动画用于模拟虚拟人物或动物，广泛用于医学、影视和游戏等领域[2]。早期计算机动画基于关键帧顶点混合，每个关键帧需要一个独立的模型，所以一段动画需要多个模型，占用很大的内存和外存。为了压缩动画数据量，Burtnyk[3]于1976年首次提出骨骼动画的概念，骨骼动画只需一个模型，运动效果由骨骼的运动驱动。相比顶点混合动画，骨骼动画是一种以时间换空间的方法，节省空间的代价是大幅增添计算负担，造成性能瓶颈问题。

并行技术在软硬件两方面的发展都非常迅速，特别在GPU并行方面，并行技术先后经历了Program汇编、Shader着色语言、CUDA三代更迭。Lindholm[7]于2001年提出基于汇编语言的骨骼动画GPU并行加速算法。季卓尔于2008年提出基于Shader的GPU骨骼动画[8]。胡前亮于2011年提出基于CUDA的骨骼动画实时阴影仿真方法[9]。GPU并行技术突破了骨骼动画的性能瓶颈以后，衍生出骨骼自动生成[10]、骨骼与皮肤自动绑定[11]、皮肤切割变换[12]等研究热点。

早期的GPU多核主要服务于图像处理任务，如果需要处理非图像领域的通用问题，需要对问题进行改造映射为图像问题，像运算图像一样执行程序，最后将结果返回，中间涉及问题转换的环节。将非图像领域的通用问题放在GPU上进行计算的这类技术，统称为通用目的GPU编程即GPGPU，实现这一技术的语言主要是Shading Language着色语言，包括Vertex Shader顶点着色和Fragment Shader片段着色以及Geometry Shader。典型代表技术有基于OpenGL扩展的GLSL(OpenGL Shading Language)、基于Direct3D扩展的HLSL(High LevelShading Language)、以及NVIDIA的Cg(C for Graphic)。本文采用OpenGL作为跟底层图形硬件交互的三维API接口，在OpenGL处理三维数据的流水线结构中，有两个流水线环节分别负责执行顶点坐标变换以及面片颜色处理，即Vertex Program和Fragment Program。起初，这两个环节的功能是固化的，没有提供API供程序人员调用。后来出现汇编语言和其它特殊图形硬件语言，用于改写以上两个环节的功能，但是各自语言像方言一样随着图形硬件品牌和型号的不同分门别类。直到OpenGL的官方机构Khronos Group于2002年发布统一的ARB(ArchitectureReviewBoard架构评审委员会)标准扩展ARB\_vertex\_program和ARB\_fragment\_program，这两个扩展都是基于统一规范的图形底层汇编语言，实现对Vertex Program和Fragment Program的编程控制。相对底层汇编语言，GLSL(OpenGL Shading Language)是一门基于C语言的高层语言，它同样由OpenGL ARB制定，用来顶替之前的ARB汇编语言。

CUDA全称Compute Unified Device Architecture，即统一计算设备架构，是NVIDIA公司专门为其显卡GPU推出的通用的并行计算架构。与上文提到的Cg等通用GPU编程语言不同，CUDA不再依赖于三维图像库OpenGL或Direct3D，也不需要将通用的非图像问题映射为图像问题。比起Cg语言宣称是C for Graphic，CUDA语言更加名副其实的贯彻C语言，它新增对内存指针这一特性的支持，使得程序开发人员可以从跟主机端内存一致的角度处理GPU设备端的数据存储，包括存储空间的申请释放与使用。CUDA可以运行于代码名称为G8x以及后来的所有NVIDIA显卡，包括GeForce、Quadro、Tesla等型号系列。CUDA程序对外在不同的操作系统下面具有良好的兼容性；对内在从G8x以来的不同的显卡硬件上也具有良好的兼容性。编译好的一份可执行程序，可以在现有的所有支持CUDA的显卡上运行，也可以在未来的显卡上运行。最早支持CUDA的一款显卡是2006年底发布的GeForce 8800 GTX，这款显卡拥有128个核心，显存带宽86 GB/s，浮点运算能力518 Gflops；截至目前性能最高的CUDA显卡是2013年初发布的GeForce GTX Titan。

随着开放并行计算语言(Open Computing Language, OpenCL)的提出，主流CPU厂商AMD和Intel先后于2009年和2010年在多核CPU上支持OpenCL技术[16]，从而为CPU并行技术的研究开辟了新的领域。OpenCL是一个标准的规范框架[17]，根据该规范编写的并行程序，可以在多种异构处理器设备端device上执行，包括CPU、GPU、DSP、FPGA和其它处理器[16]。各种主流处理器发布了各自的软件编程工具和硬件可编程能力，例如Intel的Intel SDK for OpenCL Application、AMD的APP SDK、NVIDIA的CUDA。OpenCL基于C语言设计了独特的编程语言，相比其它并行方案所用特有语言具有明显优势[18]。OpenCL还包含一套规范API供主机端host调用，用于配置OpenCL运行环境并为设备端传递数据。

在并行性能评价理论方面，Amdahl提出固定问题规模的并行加速比定律即阿姆达尔定律（Amdahl’s Law）[19]，该定律定义加速比A的上限为1/(1-P)，其中P为可并行模块的百分比，S为并行模块的局部加速比，A为全局加速比。阿姆达尔定律对应的加速比计算公式如下：

(1)

假设局部加速比S趋向∞，理论峰值加速比公式如下：

(2)

根据阿姆达尔定律公式，可以初步估算骨骼动画并行优化之后的理论峰值加速比。CPU计算顶点这一性能瓶颈占全部时间的比例约为P=80%，剩余部分只占1-P=20%。根据公式4得出理论峰值加速比为5。

本文研究基于OpenCL的骨骼动画GPU并行方法，并与GLSL和CUDA这2套传统并行方法展开对比。

1. 骨骼动画串行算法

骨骼动画渲染过程可以划分为以下四个环节，每个环节作为一个独立的子模块，四个环节分别是：1)从文件解析动画数据主要包括骨骼关节矩阵和顶点坐标；2)更新关节矩阵；3）更新顶点坐标；4）以三维面片的方式输出顶点坐标最终形成三维动画。

第三环节更新顶点坐标，占用大部分的完整渲染时间。该环节是骨骼动画的性能瓶颈，本文重点关注这个环节。本文采用矩阵调色板这一顶点更新算法，算法流程图如图1所示，算法描述如下：

1）读入1个顶点(*x,y,z*)，读入M个骨骼关节构成变换矩阵列表mL，每个顶点绑定B个关节，索引和权重分别是*d(i)*，*w(i)*，*i*=1,2…B。

2）为每个顶点求取融合矩阵*m*，*m*是4\*4的矩阵，其中包含对顶点的旋转平移缩放信息。*m*由顶点绑定多个骨骼的多个矩阵加权求和得到，计算公式如下：

*m* = (3)

3）按照以下坐标变换公式为每个顶点进行坐标变换。

= \*

= (4)

算法复杂度不一样时，各个子模块的时间分布会有所不同，瓶颈的位置和大小也会不一样。本文采用骨骼动画数据的顶点数目为0.1M，即百万级别；单个顶点绑定骨骼的数目设定为2；采用的CPU是Intel i7 3770k。

通过测量算法各个模块的时间分布可以看到，耗时最大的模块是CPU计算顶点这一模块，比例是84%，说明它是整个算法的性能瓶颈。另外CPU计算骨骼虽然算法比较复杂但是所占时间比例极小，没有优化的必要。在进行瓶颈定位和并行优化之前，需要对算法进行串行优化，使它更加符合并行运算的特点。对算法进行初步改进，使得串行代码本身获取最优性能，在此基础上进行后文的并行优化。串行优化后，CPU计算顶点所占时间比例是81%，仍然是整个算法的性能瓶颈。

图1 骨骼动画顶点更新和矩阵调色板算法流程图

1. GLSL并行

Shading Language着色语言的缩写是SL，包括OpenGL的GLSL(OpenGL Shading Language)、Direct3D的HLSL(High Level Shading Language)以及Cg(C for Graphic)。着色语言是将默认的渲染流水线操作过程进行编程改造的有力工具，一方面可以强化原先的图形功能，另一方面可以用来处理一般的非图形领域的通用计算。本文的骨骼动画算法，需要动态更新顶点坐标，属于图形功能的加强，但在索引多个矩阵并按权重叠加矩阵方面也可算作通用计算。本文采用GLSL语言实现骨骼动画渲染的顶点更新子过程。

第一步先实现单骨骼情况，骨骼矩阵索引存在顶点第四个分量即w分量，Shader程序只需要一个自定义参数，即骨骼矩阵数组matrix。鉴于矩阵数组供Vertex Shader所有顶点共享，根据表2.4.1，本文将矩阵参数的类型选定为uniform。在默认Vertex Shader上进行以下修改：

// VertexShader

uniform mat4 matrix[100]; // 新增参数，传递骨骼矩阵

void main()

{

// gl\_Position= ftransform();默认矩阵变换改为自定义矩阵变换

int index = int(gl\_Vertex.w); // 获取矩阵索引

mat4 worldMatrix = matrix[index];

vec3 blendPos = ( vec4(gl\_Vertex.xyz, 1.0) \* worldMatrix).xyz ;

// 在视图矩阵变换前，先进行骨骼矩阵变换

gl\_Position = gl\_ModelViewProjectionMatrix \* vec4(blendPos, 1.0);



gl\_FrontColor = gl\_Color; // 默认不变

}

uniform参数使用前需要在设备端赋值，赋值分初始化绑定和更新两个步骤。初始化绑定通过API接口glGetUniformLocation实现，更新通过API接口实现glUniformXXX。本文算法用到的uniform参数变量matrix的需要执行初始化赋值和更新赋值，初始化时将变量名字matrix传给glGetUniformLocation并标记变量位置locationUniformMatrix，赋值时将主机端内存中的矩阵数据传给指定位置locationUniformMatrix。其中参数数组的长度需要特别关注，赋值时使用实际矩阵个数nSize，在Vertex Shader定义时需要定义为长度固定的数组，鉴于一般计算机动画骨骼数目在100以内所以本文定义100个矩阵。

以上实现了单骨骼动画，接下来本文继续实现多骨骼动画，之前单块骨骼的索引存储在顶点数据w分量，现在多块骨骼需要多个索引和权重，而且每个顶点都需要这些数据。这时需要将索引和权重数据作为参数传给Vertex Shader，根据表2.4.1，本文将索引和权重参数的类型选定为attribute。跟之前的矩阵数据一样，索引和权重也需要绑定位置和赋值。另外，Shader还需要知道骨骼的数目，为骨骼数目绑定一个uniform变量。在单骨骼的基础上改进GLSL算法，主要新增3个变量，改进后支持多骨骼的Vertex Shader代码如下：

// VertexShader

uniform mat4 matrix[100]; // 参数1，传递骨骼矩阵

void main()

{

// gl\_Position= ftransform();默认矩阵变换改为自定义矩阵变换

int index = int(gl\_Vertex.w); // 获取矩阵索引

mat4 worldMatrix = matrix[index];

vec3 blendPos = ( vec4(gl\_Vertex.xyz, 1.0) \* worldMatrix).xyz ;

// 在视图矩阵变换前，先进行骨骼矩阵变换

gl\_Position = gl\_ModelViewProjectionMatrix \* vec4(blendPos, 1.0);

gl\_FrontColor = gl\_Color; // 默认不变

}

新增3个参数需要在Shader外部赋值，其中：每个顶点绑定的骨骼数目是uniform类型，赋值方法上文矩阵变量；骨骼索引和骨骼权重变量属于attribute类型，绑定变量位置的API接口是glGetAttribLocation，赋值API先调用glEnableVertexAttribArray开启属性然后调用glVertexAttribPointer给属性赋值。给attribute属性变量赋值时，有两种方法，第一种方法是每帧从CPU主机端内存之间传值，就像每帧指定顶点坐标一样；第二种方法是针对属性值不常改变的情况，将属性值存在OpenGL的VBO，在GPU设备端内部完成赋值。

本文先实现从CPU主机端内存给GLSL的attribute属性赋值，然后实现OpenGL与GLSL的互操作即通过VBO赋值。前一种赋值方式较为简便，可是执行效率不高，因为需要每帧传送数据，将内存数据buffer地址作为参数传给glVertexAttribPointer即可完成赋值。后一种赋值方式，通过VBO赋值时，需要先绑定显存buffer，再将显存buffer偏移地址（一般是0）作为参数传给glVertexAttribPointer。通过CPU主机端给GLSL的attribute属性赋值，跟在GPU设备端内部通过OpenGL的VBO给GLSL赋值，性能会有很大的差别，本文实现以上两种方法以后，最后采用OpenGL的VBO向GLSL传输参数。

1. CUDA并行
   1. CUDA初步算法

CUDA即Compute Unified Device Architecture统一计算设备架构，是NVIDIA专门为自家显卡设计的通用计算架构，可以充分发挥显卡的性能。前文介绍，CUDA编程模型拥有以下5方面重要的特色设计：核心函数Kernels、线程层次结构Thread Hierarchy、存储空间层次结构Memory Hierarchy、异构编程Heterogeneous Programming、计算能力Compute Capability。

基于CUDA的骨骼动画算法的Kernel代码如下：

\_\_global\_\_ void

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

{// Vector4是顶点数据结构包含4个float，即struct Vector4 { float x,y,z,w};

// 第一步，顶点(x,y,z,1) 拆分为(x,x,x,x) 、 (y,y,y,y) 、(z,z,z,z)

Vector4 pIn = pInput[index]; // index指当前线程id

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

// 第二步，获取矩阵的索引和4个向量，同上

int offset = pIndex[index]\*4 ;

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

// 第三步，矩阵变换，多项式相乘累加

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

}

Kernel函数设计完成后，需要在Host主机端配置实参和线程结构，然后调用Kernel函数，将指令发布到GPU的CUDA Device设备端进行执行，执行结束后返回处理结果供主机端或其它设备使用。Host主机端除了需要选定CUDA设备，接下来需要配置实参和线程结构，并且调用Kernel。配置实参、线程结构、调用Kernel这三个常规环节的核心代码片段如下：

第一个环节：配置实参

Host主机端为CUDA Kernel提供实参数据，Device设备端为CUDA Kernel提供实参数据结构即显存空间，然后将数据从主机的内存发送到显卡的显存，供Kernel调用。需要设置的参数包括：初始顶点坐标、变换后的顶点坐标、骨骼矩阵、骨骼索引、骨骼权重。

其中变换后的顶点坐标需要交由OpenGL进行渲染，CUDA支持与OpenGL进行互操作，即它们可以共享数据。共享的数据对于OpenGL是VBO(Vertex Buffer Object顶点缓存对象)，在生成VBO时，通过CUDA的互操作接口cudaGraphicsGLRegisterBuffer绑定CUDA图像资源cudaGraphicsResource，通过VBO和cudaGraphicsResource的绑定使得CUDA可以操作OpenGL的渲染数据。CUDA操作VBO数据时，需要先调用API接口cudaGraphicsMapResources和cudaGraphicsResourceGet -MappedPointer将VBO转化成CUDA的全局显存，执行完kernel以后，再调用API接口cudaGraphicsUnmapResources将CUDA显存还原为VBO，从而实现通过GPU修改VBO进而产生动态顶点和动态网格，再使用OpenGL的纹理映射产生动态皮肤，最终呈现动态三维角色模型。

第二个环节：线程结构

线程块Block结构的设置原则是：线程块内部包含的线程数目是32的倍数，一般设置为256；线程网格Grid结构的设置原则是：线程块数可以划分为动态变化和静态不变这两类设置方法，线程块数动态变化的情况根据所需线程数目决定，如果每一个线程处理一个元素，总共线程数就是数据元素的个数，这时线程块数等于元素个数除于线程块内包含线程数；另一种设置方法是让线程块数保持静态不变比如设为64，而每个线程处理多个元素，具体处理多少个由元素个数决定，数值等于元素个数除以线程块数与线程块内包含线程数。

第三个环节：调用Kernel

transformVectorByMatrix4One为上文设计的Kernel函数；grid和block为上文的线程结构参数；d\_p\*等参数分别是：初始顶点坐标、变换后的顶点坐标、骨骼矩阵、骨骼索引、骨骼权重。完整调用代码如下：

{

transformVectorByMatrix4One<<< grid, block >>>( (Vector4 \*) d\_pInput, d\_pIndex, (Vector4 \*)d\_pMatrix, (Vector4 \*)d\_pOutput, sizeMax, d\_pWeight );

}

* 1. CUDA优化

### 全局显存对齐

上文用到顶点数据结构Vector4，定义为struct Vector4 { float x,y,z,w; }; 根据CUDA规范，这种结构不符合CUDA的显存对齐结构要求，显存读写带宽不能达到最优，所以需要对数据结构进行对齐声明。声明方法：在struct和Vector4中间插入符号”\_\_builtin\_align\_\_(16)”，即 struct \_\_builtin\_align\_\_(16) Vector4 { float x,y,z,w; }; 实际上CUDA内置了一种数据结构float4等效于对齐了的Vector4，所以本文将自定义的未对齐的数据结构Vector4替换为对齐了的数据结构float4，从而满足CUDA显存对齐的规范。对齐主要面向自定义数据结构，例如typen，type是float/int等，n是2、3、4；基本数据结构不需要对齐，例如：float/int，虽然CUDA也提供了float1/int1等对齐类型，但对性能没有影响。

### 处理器利用率

处理器利用率取决于一个MP(Multiprocessor多处理器)内当前激活线程数与最大可同时容纳线程数的比值，或者当前激活线程warp数与最大可同时容纳线程warp数的比值。激活线程数受限于每个MP的寄存器个数，MP内寄存器总数与每个线程所需寄存器个数的比值就是当前激活线程数。当每个线程所需寄存器个数增多时，因为MP寄存器资源是固定的，所以当前激活线程数减少，处理器利用率降低。为了提升处理器利用率，需要优化程序最大程度节省寄存器用量，并且合理配置block的维度和数量。当前kernel的寄存器用量可以通过CUDA附带工具NVIDIA Visual Profiler测量得到。除了算法的寄存器用量以外，算法的共享存储器用量也将影响激活线程块的数目，从而影响处理器利用率。

### 合并访问

本文算法涉及3类全局显存，第1类是输入输出点坐标，第2类是矩阵索引和权重，第3类是矩阵行列值。对于第1类，动态网格块每个线程处理1个元素，自然满足合并条件；静态网格每个线程处理多个元素，等间隔的满足合并条件，连续的情况破坏相邻线程访问连续元素的合并条件。针对第1类上一小节已测得，网格块配置方案为动态或者等间隔静态，性能相当并且比连续静态高30%。第2类是矩阵索引和权重，每个线程为每个顶点读取连续多个值改为一次读取，确保线程间连续，满足合并条件，性能提升10%。第3类是矩阵行列值，每个线程所需多个矩阵的当前存储结构是连续的，而且矩阵内部三行四元组即4个flaot4也是连续的，线程内连续必然破坏线程间连续，从而不满足合并条件。将每个顶点所需矩阵信息以float4为单位进行拆分，即将AOS(Array of Struct)结构转变为SOA (Array of Struct)结构，从而满足合并访问的条件，但是相邻线程随机访问矩阵，所以实际上不能合并访问。

### 常量显存缓存矩阵

算法第三处显存即矩阵行列值，相当于一张数据表，由所有线程随机访问共同使用，适合存储在常量显存中。常量显存的总量64K，可以容纳大约1000个骨骼矩阵的数据，实际骨骼数在100左右，所以常量显存完全可以容纳骨骼的矩阵行列值。通过发挥常量显存的缓存能力，缩短kernel的访存时间，从而提升kernel性能。常量显存的申请，需要在全局访问域进行，而且容量空间必须是常数，本文申请100个矩阵空间。常量显存的赋值在调用kernel之前通过CUDA主机端API接口cudaMemcpyToSymbol完成。常量显存的使用跟在CPU的内存空间使用普通的数组一样。

### 共享显存缓存矩阵

共享显存(Shared Memory)跟常量显存的不同之处有4点：第一，共享范围不同，共享显存针对线程块而常量显存面向所有线程；第二，申请长度不同，共享显存支持动态而常量显存必须是静态；第三，变量定义位置不同，共享显存在kernel函数内而常量显存在所有函数之外的全局域；第四，存储位置不同，共享显存位于片内(on chip)而常量显存位于片外(off chip)。

1. OpenCL并行

OpenCL必须安装设备驱动和软件开发包SDK，本文选用NVIDIA CUDA，其中包含OpenCL SDK。因为OpenCL是开放标准，同一台系统里面可以安装多套OpenCL驱动和SDK，比如Intel OpenCL、AMD OpenCL、NVIDIA OpenCL。clGetPlatformIDs可以获取已经安装OpenCL的数目以及详细信息，当数目为0时，说明未安装OpenCL；当数目不为0时，按照需要选用特定的某套OpenCL，比如本文选用"NVIDIA CUDA"。

调用OpenCL之前，需要初始化OpenCL运行环境。初始化工程对于不同工程差别不大，主要选定OpenCL类型、选定CPU或是GPU、指定cl文件及kernel函数名。初始化完成后，接下来设计kernel函数，设计过程可以参照SSE算法版本。

* 1. 常规OpenCL并行

先实现单骨骼情况，即每一个顶点只关联1块骨骼，坐标变换时只有一次矩阵变换。OpenCL的kernel函数实现如下：

\_\_kernel void

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

{

// 顶点索引

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

// 矩阵索引，每个矩阵由4个flaot4组成

int offset = pIndex[index]\*4;

// (x,y,z,1) 拆分为(x,x,x,x) 、 (y,y,y,y) 、(z,z,z,z)

float4 pIn = pIn [index];

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

// 矩阵变换，多项式相乘累加

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

}

对于每个顶点关联多个骨骼的情况，在获取变换矩阵时，需要索引多个矩阵，按照权重累加。所以在kernel参数列表中需要添加矩阵索引和权重，在kernel函数体中仿照SSE按权重累加矩阵即可。OpenCL执行之前，除了指定供OpenCL设备调用的指令代码以外，还需指定工作线程结构workgroup/workitem，包括全局Global WorkSize和局部Local WorkSize。它们都支持2到3维，一般只设置GlobalWorkSize，设置时每个维度支持最大数目不同，通过clGetKernelWorkGroupInfo可以动态检测到这个值，各个维度乘积大于线程总需求量即可，本文采用二维线程结构，第一维设置为最大值，第二维用总顶点数除于最大值；而Local WorkSize采用系统默认设置。

* 1. 与OpenGL互操作

为了优化骨骼动画渲染和数据传输的效率，本文采用OpenCL与OpenGL的互操作特性(Interoperability)，共享顶点缓存对象(Vertex Buffer Object, VBO)等存储对象(Memory Object)。为了支持内存共享以及互操作，在OpenGL和OpenCL两端都需要做相应的调整。在OpenGL端，需要改变OpenGL渲染数据的存储方式。原先是立即存储方式，即存在主机端内存，每次渲染时发送到OpenGL设备，一旦发送到OpenGL设备以后处于失控状态而无法访问；现在要改为Vertex Buffer Object(VBO)方式，数据一直存在OpenGL设备，而且可以通过VBO的Buffer资源ID进行读写访问。本文将骨骼动画顶点数据存储在VBO当中，创建VBO的同时通过clCreateFromGLBuffer映射一块OpenCL内存cl \_mem，VBO和cl \_mem共享顶点数据。当顶点需要通过OpenCL更新时，通过clEnqueueAcquireGLObjects获取数据，修改完成后通过clEnqueueReleaseGLObjects返还给VBO。

通过在OpenGL和OpenCL分别进行以上改造，从2个方面对算法性能进行了改进。一方面，使得OpenGL和OpenCL之间可以共享数据，即实现了OpenGL的VBO与OpenCL的cl\_mem之间的双向转化，从而免去主机端参与中转，可以节省数据传输时间；另一方面，因为采用VBO渲染方式，所以提升了渲染性能。

1. 动画仿真实验
   1. 并行方案对照

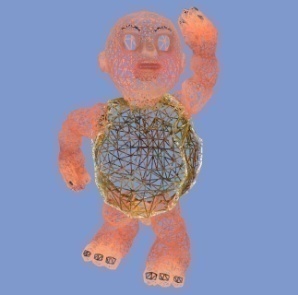
本文实验数据即骨骼动画数据的规格如下：顶点数目为0.1M，即百万级别；骨骼关节点的数目为77；单个顶点绑定骨骼的数目设定为2。实验平台配置如下：GPU——NVIDIA Geforce GTX 670、CPU——Intel i7 3770K。骨骼动画渲染效果如图2。

图2小海龟角色动画渲染效果

本文在骨骼动画矩阵调色板串行算法的基础上，面向GPU设计了4套互为对照的并行版本，其中1套GLSL方案， 1套CUDA方案，优化前后的2套OpenCL方案。并与面向CPU的1套OpenCL方案展开对比。

如图3所示，5套并行算法在选定的数据和平台上运行，统计得到5组并行加速比。从以上实验结果图得知：1）GLSL算法性能最优，经过优化的OpenCL其次，未经优化的OpenCL最低；2）OpenCL性能介于CUDA和GLSL之间，OpenCL是CUDA的2倍，比GLSL低4%。

图3 骨骼动画对比4组并行算法的加速比

* 1. 并行可移植性

本文研究的移植空间限于不同品牌不同型号的GPU处理器。本文研究的3中GPGPU并行技术里面，只有OpenCL具有跨越硬件的可移植性。本文选定3组硬件进行对照实验，其中2款同品牌不同型号的NVIDIA GPU，外加1款AMD GPU，体现跨越硬件品牌和型号的可移植性。

如图4实验结果可知，对于不同品牌不同型号GPU，OpenCL版本的骨骼动画渲染算法拥有3.4~9.1倍的加速比。说明OpenCL具有较好的功能和性能可移植性。而且除了低档GPU以外，GPU的性能普遍比CPU要高。

图4 骨骼动画对比3组CPU的加速比

* 1. 数据复杂度对比

数据复杂度由以下2个参数决定，包括单个模型内每个顶点绑定的骨骼数目、单个模型的顶点数目。本文为2个参数分别设定4个数量级，骨骼数为1至4，顶点数范围为25k~1600k，交织在一起组成4乘4矩阵共16组不同复杂度的数据。测量在16组数据下的OpenCL加速比和SSE加速比，并统计OpenCL与CUDA和GLSL的加速比比值。

从实验数据表1可以看出，OpenCL并行算法相对串行算法具有绝对性能加速能力，加速比范围在1.4~21之间。从表2得出结论，OpenCL的性能介于CUDA和GLSL之间。对于骨骼数大于1顶点数多于25k的骨骼动画，OpenCL相对GLSL的性能降幅在30%以内；OpenCL相对CUDA的性能增幅在20%~170%之间

表1 骨骼动画对比16组数据的OpenCL加速比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 顶点  骨 数  骼数 | 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 |

表2 骨骼动画对比16组数据的OpenCL与CUDA和GLSL的加速比增幅

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 顶点  骨 数  骼数 | 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. 结论

现有GPGPU骨骼动画矩阵调色板算法基于2种API——图形库OpenGL的扩展接口GLSL以及NVIDIA CUDA，其中GLSL因渲染和计算紧密耦合在一起不利于算法的移植复用，CUDA仅限于NVIDIA品牌的GPU。本文为骨骼动画提出基于OpenCL技术的GPGPU加速算法，提升软件和硬件可移植性，适用于各种GPU和CPU，提升了骨骼动画矩阵调色板算法的硬件可移植性。本文发现，在多数GPU和多种复杂度的数据上，OpenCL方案相比GLSL性能稍低但基本相当，OpenCL方案比CUDA性能更高。

**参考文献：**

[1] Kavan L. Real-time Skeletal Animation[D]. Prague: Czech Technical University, 2007.

[2] 赵维，谢晓方. 虚拟人技术发展现状及其在工程中的应用[J]. 系统仿真学报, 2009(17): 5473-5476.

[3] Burtnyk N, Wein M. Interactive skeleton techniques for enhancing motion dynamics in key frame animation[J]. Communications of the ACM(S0001-0782), 1976, 19(10): 564-569.

[4] Chafi H, Sujeeth A K, Brown K J, et al. A domain-specific approach to heterogeneous parallelism[C]// Proceedings of the 16th ACM symposium on Principles and practice of parallel programming. ACM, 2011.35-46

[5] Brown K J, Sujeeth A K, Lee H J, et al. A heterogeneous parallel framework for domain-specific languages[C]// Parallel Architectures and Compilation Techniques (PACT). 2011.89-100

[6] Ismail L G D. Performance evaluation of convolution on the Cell Broadband Engine processor[J]. IEEE Transactions on Parallel and Distributed Systems(S1045-9219), 2011, 22(2): 337-351.

[7] Lindholm E, Kilgard M J, Moreton H. A user-programmable vertex engine[C]// Proceedings of the 28th annual conference on Computer graphics and interactive techniques. ACM, 2001.149-158

[8] 季卓尔，张景峤. 基于可编程 GPU 的骨骼动画[J]. 计算机工程与应用, 2008, 44(22): 77-80.

[9] 胡前亮，陈炳发. 一种采用CUDA的骨骼动画阴影实时仿真方法[J]. 小型微型计算机系统, 2011, 32(1).

[10] 郝爱民，赵永涛，吴伟和，等. 任意姿态虚拟人网格模型骨骼提取算法[J]. 中国图象图形学报, 2011, 16(6): 1008-1014.

[11] 刘登志. 人体角色的自动绑定与卡通运动[D]. 浙江大学, 2011.

[12] Han-Bing Y, Shi-Min H, Martin R R, et al. Shape Deformation Using a Skeleton to Drive Simplex Transformations[J]. IEEE Transactions on Visualization and Computer Graphics(S1077-2626), 2008, 14(3): 693-706.

[13] Lee V W, Kim C, Chhugani J, et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU[C]// ACM SIGARCH Computer Architecture News. ACM, 2010.451-460

[14] Shi G, Li M, Lipasti M. Accelerating search and recognition workloads with SSE 4.2 string and text processing instructions[C]// Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on. IEEE, 2011.145-153

[15] Jie S, Jianbin F, Sips H, et al. Performance Gaps between OpenMP and OpenCL for Multi-core CPUs[C]// Parallel Processing Workshops (ICPPW), 2012 41st International Conference on. Pittsburgh, PA: 2012.116-125

[16] Pennycook S J, Hammond S D, Wright S A, et al. An investigation of the performance portability of OpenCL[J]. Journal of Parallel and Distributed Computing(S0743-7315), 2012: 1-12.

[17] Khronos. The OpenCL Specification[EB/OL].(2012)[2012-11-14]. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[18] Du P, Weber R, Luszczek P, et al. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming[J]. Parallel Computing(S0167-8191), 2012, 38(8): 391-407.

[19] Sun X, Chen Y. Reevaluating Amdahl’s law in the multicore era[J]. Journal of Parallel and Distributed Computing(S0743-7315), 2010, 70(2): 183-188.

1. 收稿日期：xxxx-xx-xx 修回日期：xxxx-xx-xx

   基金项目：国家海洋局数字海洋科学技术重点实验室开放基金（KLDO201303）

   作者简介：刘寿生(1985-), 男, 江西宁都人, 汉族, 博士生, 研究方向为虚拟现实、并行计算、高性能； 陈戈(1965-), 男, 浙江宁波人, 汉族, 博士, 教授, 博导, 研究方向为海洋遥感、地理信息系统、虚拟现实； 马纯永(1984-), 男, 山东潍坊人, 汉族, 博士, 讲师, 研究方向为虚拟现实； 韩勇(1969-), 男, 陕西商南人, 汉族, 博士, 教授, 研究方向为虚拟地理环境、海洋地理信息系统； [↑](#footnote-ref-2)