谨以此文献给母校中国海洋大学

**------------------**刘寿生

虚拟现实仿真平台异构并行计算关键技术研究

Researchon Key Techniques of Heterogeneous Parallel Computing for Virtual Reality Simulation Platform

学位论文答辩日期：

指导教师签字：

答辩委员会成员签字：

**独 创 声 明**

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含未获得 （注：如没有其他需要特别声明的，本栏可空）或其他教育机构的学位或证书使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名： 签字日期： 年 月 日

---------------------------------------------------------------------

**学位论文版权使用授权书**

本学位论文作者完全了解学校有关保留、使用学位论文的规定，并同意以下事项：

1、学校有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。

2、学校可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。同时授权清华大学“中国学术期刊(光盘版)电子杂志社”用于出版和编入CNKI《中国知识资源总库》，授权中国科学技术信息研究所将本学位论文收录到《中国学位论文全文数据库》。（保密的学位论文在解密后适用本授权书）

学位论文作者签名： 导师签字：

签字日期： 年 月 日 签字日期： 年 月 日

虚拟现实仿真平台异构并行计算关键技术研究

摘要

面向单核处理器的传统单线程算法难以满足海量数据处理的实时性需求，也无法充分发挥多核处理器的计算潜力，并行计算技术成为计算机性能优化的前沿技术。特别在多媒体、三维图形这些具有高实时性需求的领域，快速处理海量数据的需求尤为迫切。本文研究对象是虚拟现实地理信息系统一体化仿真平台VRGIS，它是一套集虚拟现实及地理信息系统为一体的软件平台，该软件具备城域级别三维模型数据以及三维地形数据的承载能力，还支持多种复杂自然现象模拟以及三维可视化特效，并且具备较高的画面逼真度和交互实时性。本文旨在研究各种并行计算技术，以解决VRGIS在三维仿真过程中以骨骼动画和粒子系统为代表的各种性能瓶颈问题。

主要研究内容如下：

1. **诊断虚拟现实仿真平台VRGIS的性能瓶颈**

经诊断发现引擎中性能瓶颈主要分布于：虚拟现实动态角色骨骼动画渲染、基于粒子系统的复杂自然现象仿真模拟等具有高密集度大运算量的模块。

1. **建立多技术方案交叉互评的并行计算性能评价模型**

本文在阿姆达尔定律的理论基础上，引入并行计算多技术方案交叉互评机制，完善并行计算性能评价模型。本文研究的五套并行计算技术方案中，最新的是OpenCL(Open Computing Language, 开放计算语言)，可以同时用于中央处理器CPU和图形处理器GPU。此外CPU和GPU有各自专用的并行计算技术，其中： OpenMP(Open Multi-Processing, 开放多线程处理)和SSE(Streaming SIMD Extensions, 流式单指令多数据扩展指令集)专门面向CPU； GLSL(OpenGL Shading Language, OpenGL着色语言)和CUDA(Compute Unified Device Architecture, 统一设备架构)专门面向GPU。

1. **为骨骼动画矩阵调色板算法设计多个并行计算方案**

实现并改进了已有的包括SSE和GLSL在内的骨骼动画并行计算方案，结合包括CUDA、OpenCL在内的新兴并行计算技术针对骨骼动画提出了新的并行计算方案，对比分析各种并行计算技术。

1. **为柏林噪声风场扰动喷泉粒子系统设计基于OpenCL的并行计算方案**

为了提升喷泉粒子系统仿真效果的逼真度，引入柏林噪声随机因子模拟风场扰动效果，动态模拟过程所需复杂运算极大影响仿真实时性，本文采用基于OpenCL的并行计算技术，同时面向CPU和GPU提出了粒子系统性能改进方案。

1. **构建多个并行计算任务与多个并行计算设备之间的映射原则**

当多个模块同时进行并行计算时，为了充分挖掘CPU和GPU等多个设备的异构并行计算能力，在前文隔离拆分并独立解决VRGIS内部包括骨骼动画和粒子系统两大瓶颈问题的基础上，将两个模块重新合并在一起作为多任务系统，研究虚拟现实仿真平台多个并行计算任务与多个并行计算设备之间的映射原则和执行方案。

本文创新点主要体现在以下三个方面：

**1. 提出了基于CUDA的骨骼动画矩阵调色板GPGPU算法**。功能创新：现有骨骼动画矩阵调色板算法基于图形库OpenGL的扩展接口GLSL，渲染和计算紧密耦合在一起，不利于算法的移植复用。本文采用脱离图形库的CUDA技术，使渲染和计算有效隔离，在软件编程方面提升了矩阵调色板算法的可移植性。性能创新：以CPU串行算法和基于SSE叠加OpenMP的传统并行算法作为性能参考基准，CUDA算法加速比分别是4.4和2.5。

**2. 提出了基于OpenCL面向CPU和GPU异构体系的骨骼动画矩阵调色板算法**。功能创新：提升面向GPU骨骼动画矩阵调色板算法的可移植性，原先基于CUDA的算法分别依赖特定的GPU，基于OpenCL面向GPU的矩阵调色板算法普遍适用于各种GPU。性能创新：面向CPU的OpenCL算法，以CPU串行算法和基于SSE叠加OpenMP的传统并行算法作为性能参考基准，加速比分别是3.9和1.5。

**3. 设计了基于OpenCL面向CPU和GPU异构体系的柏林噪声风场扰动粒子系统喷泉**。功能创新：将现有基于CUDA的柏林噪声并行算法移植到OpenCL，从而突破了硬件限制，提升了柏林噪声并行算法的可移植性和通用性。性能创新：以CPU串行算法作为性能参考基准，OpenCL在CPU和GPU上的加速比分别是3.4和65。

本文结合虚拟现实仿真平台的骨骼动画模块和粒子系统模块研究多种并行计算技术，当研发人员需要做以下决策时——是否将现有串行算法进行并行化、是否追随新的并行计算技术对已有并行算法进行移植升级、是否升级并行计算硬件设备，本文可为其提供科学有效的决策依据。

关键词：并行计算；OpenCL；CUDA；骨骼动画；粒子系统；虚拟现实

Research on Key Techniques of Heterogeneous Parallel Computing for Virtual Reality Simulation Platform

Abstract

The traditional single-threaded algorithm for single-core processor is difficult to meet the real-time requirements of mass data processing; it also cannot give full play to the potential of multi-core processors computing. Parallel computing technique has become the forefront of computer performance optimization techniques. Especially in the areas of multimedia and three-dimensional graphics in which a high demand for real-time is required, demand for rapid processing of massive data is particularly urgent. This dissertation is oriented to VRGIS, a virtual reality simulation platform integrated geographic information system. The software has a carrying capacity for three-dimensional metro-level model data and three-dimensional terrain data. VRGIS also supports a variety of complex natural phenomena simulation and visualization effects, and has a high picture fidelity and real-time interaction. In this dissertation, various parallel computing techniques are studied to solve a variety of performance bottlenecks of VRGIS from skeletal animation to particle systems in three-dimensional simulation.

The major research work includes the following five aspects:

1. To diagnose performance bottlenecks of the virtual reality simulation platform VRGIS

After diagnosis performance bottlenecks are located in: dynamic role skeletal animation rendering, complex natural phenomena simulation based on particle system with high intensity and big computation.

1. To establish an performance evaluation model of parallel computing by comparison among multiple peer technique solutions

In this dissertation, with the theoretical basis of Amdahl's law an assessment mechanism by multiple parallel computing technique solutions is introduced to improve the performance evaluation model for parallel computing. Among the five sets of parallel computing technique programs in this dissertation, the latest one is Open Computing Language (OpenCL) which can be used for both central processing units (CPUs) and graphics processing units (GPUs). Besides CPU and GPU have their own dedicated parallel computing technique. Open Multi-Processing (OpenMP) and Streaming SIMD Extensions (SSE) are oriented to CPU, while OpenGL Shading Language (GLSL) and Compute Unified Device Architecture (CUDA) are oriented to GPU.

1. To design multiple parallel computing schemes for the matrix palette algorithm of skeletal animation

Existing parallel computing schemes SSE and GLSL are implemented and improved for skeletal animation. With new techniques such as CUDA and OpenCL new parallel computing schemes are produced for skeletal animation, comparison and analysis are performed among these parallel computing schemes with different techniques.

1. To design a parallel computing scheme based on OpenCL for fountain simulation with Berlin noise wind disturbance by particle system

In order to improve the fidelity of the simulation results random disturbance factor such as Berlin noise is involved. A parallel computing scheme based on OpenCL is used to perfect the performance of particle system running on both CPU and GPU.

1. To construct a principle to map multiple parallel tasks to multiple parallel devices

While multiple tasks are running at the same time with the purpose to fully exploit the heterogeneous parallel computing capabilities of multiple devices such as CPU and GPU, two separated modules skeletal animation and particle system are treated as a whole multi-task system. Then a principle to map multiple parallel tasks to multiple parallel devices is studied to get further performance optimization for simulation platform of virtual reality.

The innovation of this dissertation is mainly reflected in three aspects:

1. To propose the matrix palette algorithm for skeletal animation based on CUDA. Performance innovation: With two set of reference benchmark including serial algorithm and traditional parallel algorithm by SSE and OpenMP on CPU, CUDA version of the matrix palette algorithm gets a speedup of 4.4 and 2.5. Function innovation: The existing skeletal animation matrix palette algorithm based on GLSL extended from graphics library OpenGL, which couples the rendering and calculations together closely. The existing algorithm is not convenient to reuse. In this dissertation the matrix palette algorithm based on CUDA is independent by the graphics library to render and to calculate, which brings portability in the software programming.

2. To propose the matrix palette algorithm for skeletal animation based on OpenCL for heterogeneous platform mixed CPU and GPU. Performance innovation: With two set of reference benchmark including serial algorithm and traditional parallel algorithm by SSE and OpenMP on CPU, OpenCL version of the matrix palette algorithm on CPU gets a speedup of 3.9 and 2.2. Function innovation: Portability is advanced for skeletal animation algorithm matrix palette from CUDA to OpenCL. Because original algorithm based on CUDA is dependent on particular NVIDIA GPU, while algorithm based on OpenCL for matrix palette is generally applicable to a variety of GPU.

3. To design a parallel computing scheme based on OpenCL for fountain simulation with Berlin noise wind disturbance by particle system for heterogeneous platform mixed CPU and GPU. Performance innovation: With a reference benchmark which is formed by serial algorithm on CPU, OpenCL gets speedup of 3.4 on CPU and 65 on GPU.

Multiple parallel computing techniques are studied to accelerate the performance of skeletal animation and particle system involved in virtual reality. Researchers and developers can get valuable proofs and clues from this dissertation while the following decisions are looking forward to be made: whether to parallelize the serial algorithm, whether to port outdated parallel algorithm to the latest version, whether to update the parallel hardware device.

Keyword: Parallel Computing; OpenCL; CUDA; Skeletal Animation; Particle System; Virtual Reality

目录

[1 绪论 1](#_Toc374631910)

[1.1 研究背景和意义 1](#_Toc374631911)

[1.2 国内外相关领域研究现状 2](#_Toc374631912)

[1.2.1 通用并行计算 2](#_Toc374631913)

[1.2.2 并行计算性能评价模型 4](#_Toc374631914)

[1.2.3 虚拟现实系统并行计算 4](#_Toc374631915)

[1.3 本文组织结构 5](#_Toc374631916)

[2 通用并行计算技术研究 7](#_Toc374631917)

[2.1 并行计算性能评价模型 8](#_Toc374631918)

[2.1.1 基于阿姆达尔定律的性能评价理论模型 8](#_Toc374631919)

[2.1.2 性能指标的选定及其准确度的考虑 9](#_Toc374631920)

[2.1.3 并行计算多方案交叉对比改进性能评价模型 12](#_Toc374631921)

[2.2 基于SSE面向CPU的通用并行计算 16](#_Toc374631922)

[2.2.1 OpenMP多线程任务级并行 16](#_Toc374631923)

[2.2.2 线程内SSE指令级并行 19](#_Toc374631924)

[2.2.3 OpenMP与SSE双层嵌套并行 21](#_Toc374631925)

[2.3 基于OpenCL面向CPU和GPU的异构通用并行计算 24](#_Toc374631926)

[2.3.1 OpenCL技术概览 24](#_Toc374631927)

[2.3.2 OpenCL的异构特性和可移植性 27](#_Toc374631928)

[2.3.3 OpenCL版本的加权图像变换算法实现 28](#_Toc374631929)

[2.3.4 OpenCL算法性能分析 29](#_Toc374631930)

[2.4 基于GLSL的GPGPU通用并行计算 30](#_Toc374631931)

[2.4.1 顶点着色器语言Shading Language简介 30](#_Toc374631932)

[2.4.2 着色器语言编程规范 32](#_Toc374631933)

[2.5 基于CUDA的GPGPU通用并行计算 35](#_Toc374631934)

[2.5.1 统一计算设备架构CUDA简介 35](#_Toc374631935)

[2.5.2 CUDA编程模型的设计特色 37](#_Toc374631936)

[2.6 本章小结 44](#_Toc374631937)

[3 骨骼动画渲染算法在CPU上的并行计算 47](#_Toc374631938)

[3.1 骨骼动画矩阵调色板算法 47](#_Toc374631939)

[3.1.1 为骨骼动画渲染算法建模 47](#_Toc374631940)

[3.1.2 串行算法性能分析和瓶颈定位 51](#_Toc374631941)

[3.2 基于SSE面向CPU的骨骼动画矩阵调色板算法 54](#_Toc374631942)

[3.2.1 OpenMP多线程并行 54](#_Toc374631943)

[3.2.2 基于SSE的线程内指令并行 56](#_Toc374631944)

[3.2.3 线程与指令双层并行 58](#_Toc374631945)

[3.3 基于OpenCL面向CPU的骨骼动画矩阵调色板算法 59](#_Toc374631946)

[3.3.1 矩阵调色板算法从SSE移植到OpenCL 59](#_Toc374631947)

[3.3.2 OpenCL与OpenGL互操作 63](#_Toc374631948)

[3.4 实验结果分析和本章小结 68](#_Toc374631949)

[4 骨骼动画渲染算法在GPU上的并行计算 69](#_Toc374631950)

[4.1 基于GLSL面向GPU的骨骼动画矩阵调色板算法 69](#_Toc374631951)

[4.1.1 GLSL骨骼动画算法移植 69](#_Toc374631952)

[4.1.2 GLSL骨骼动画算法性能分析 71](#_Toc374631953)

[4.2 基于CUDA面向GPU的骨骼动画矩阵调色板算法 72](#_Toc374631954)

[4.2.1 未经优化并且功能简化的初步算法 72](#_Toc374631955)

[4.2.2 完善算法实现对多骨骼的支持 76](#_Toc374631956)

[4.2.3 根据CUDA硬件特性和算法特性优化算法 79](#_Toc374631957)

[4.2.4 CUDA算法性能分析 85](#_Toc374631958)

[4.3 基于OpenCL面向GPU的骨骼动画矩阵调色板算法 87](#_Toc374631959)

[4.4 实验结果分析和本章小结 89](#_Toc374631960)

[5 面向虚拟现实系统的多任务异构协同并行计算 92](#_Toc374631961)

[5.1 基于柏林噪声风场扰动的喷泉粒子系统模拟 92](#_Toc374631962)

[5.1.1 喷泉粒子系统模拟 92](#_Toc374631963)

[5.1.2 柏林噪声风场扰动 96](#_Toc374631964)

[5.1.3 性能瓶颈分析 98](#_Toc374631965)

[5.1.4 OpenCL并行计算 98](#_Toc374631966)

[5.2 CPU与GPU异构协同处理VR系统多任务 99](#_Toc374631967)

[5.2.1 相同粒度的多节点并行计算系统 100](#_Toc374631968)

[5.2.2 不同粒度的多节点并行计算系统 100](#_Toc374631969)

[5.3 本章小结 101](#_Toc374631970)

[6 总结和展望 102](#_Toc374631971)

[6.1 总结 102](#_Toc374631972)

[6.2 展望 103](#_Toc374631973)

[参考文献 105](#_Toc374631974)

[研究成果 115](#_Toc374631975)

[致谢 117](#_Toc374631976)

# 

# 绪论

## 研究背景和意义

虚拟现实地理信息系统一体化仿真平台VRGIS是一套集VR虚拟现实及GIS地理信息系统为一体的三维引擎软件平台，该平台由中国海洋大学海洋信息技术实验室(Marine Information Technology Laboratory, MIT Lab)自主研发。它采用国际上标准的三维图形引擎OpenGL，集成支持多数据源、多分辨率的海量数据所构建的三维场景的稳定快速导航，组件式的开发技术使VR与GIS两大功能模块无缝融合，可实现构建真实空间的地形地貌和地物外观，表达空间实体的空间关系，进行空间信息和数据的组织与分析预测，模拟地理空间过程的发展与演变过程，VRGIS平台架构如图1-1。由于其直观化、多维度、高仿真等优势，目前已被广泛应用于地球科学、空间模拟、数字城市、军事演练、多媒体展示、旅游宣传、考古还原、房产开发、工业展示等与国民生产和人类生活相关的各个领域。

图1-1 VRGIS平台架构示意图

GIS查询与分析

应应用层

三通道环幕可视化

三维数据可视管理

视频/效果图输出

规划方案三维可视化审批

三维可视化查询与分析

关键区域保护仿真建库

变迁模式分析

功功能层

虚拟场景自动导航模块

接接口层

VRGIS数据获取模块

三维空间分析模块

三维场景编辑模块

虚拟场景漫游模块

三维空间信息查询模块

大规模数据库管理

引引擎层

GIS API、CAD API

三维模型API

实时交互API

矢量数据引擎

模型数据引擎

虚拟现实引擎

数数据层

矢量数据库

DEM高程数据库

纹理数据库

属性数据库

模型数据库

本文以国家自然科学基金（No: 40971207）“基于VR-GIS的城市噪声检测数据三维分析模型及可视化评价研究”为背景，并且在国家科技部科技人员服务企业行动基金（No: SQ2009GJA0002426）“基于CUDA并行架构的高逼真度VRGIS加速引擎”的支持下，以VRGIS平台为研究对象，通过采用并行计算技术解决VRGIS的性能瓶颈突出问题。VRGIS引擎在承载大量动态模型数据实时仿真时，画面流畅性和交互实时性大幅降低，暴露出该引擎内部存在性能瓶颈问题。为了解决虚拟现实仿真平台的性能瓶颈问题，进一步提升该引擎的执行性能，从而带动数据量承载能力的攀升以及实时性的突破，本文全面诊断VRGIS三维引擎潜在的性能瓶颈，结合最新的并行软件技术和硬件，探索应对性能瓶颈问题的解决方案，并具体解决具有典型代表性的突出瓶颈问题。经诊断发现引擎中性能瓶颈普通分为以下两种情况：

1. 频繁动态更新现有的真实数据；
2. 根据物理规律按一定行为模式动态生成大量模拟数据。

本文进一步选定以下两个典型瓶颈问题，包括：

1. VR虚拟现实的动态角色动画模型快速渲染；
2. 复杂自然现象的粒子系统高性能仿真模拟。

本文探索的三维引擎性能瓶颈诊断方案以及解决方案具备很好的通用性，可为三维图像处理提供可借鉴和可移植性的技术方案；针对骨骼动画、粒子系统自然现象两个模块典型瓶颈的优化方案，可沿用到模块内部的其它瓶颈问题。

## 国内外相关领域研究现状

### 通用并行计算

在具有大数据量高计算密度的科学运算领域，并行计算(Parallel Computing)技术一直是国内外科研人员的研究焦点。国际上，Asanovic[1,2]于2006、2009年先后对并行计算的发展现状和将来前景做了分析预测，随着处理器频率接近饱和状态，多核Multicore[3-5]必然是一个继续提升计算机性能的出路，并行计算技术扮演不可或缺的重要角色。Brodtkorb[6]于2010年总结了异构计算的最新发展趋势，主要研究对象是Cell Broadband Engine、GPUs、FPGAs这三种常用体系结构。在我国，陈国良[7]于2009年就并行计算的一体化研究现状进行简要的综述，并结合多核系统、云计算、个人高性能计算机等对于并行计算一体化研究的影响，概要的展望了其发展趋势。

并行计算技术有以下几种分类方式：

1. 按照并行层次划分为：位(bit)级别并行、指令级别并行、任务级别并行；
2. 按照面向硬件体系结构划分为：CPU多核、GPU多核、多CPU集群，对应技术有SIMD[8,9]和OpenMP[10,11]、GPGPU、MPI，其中SIMD包括SSE[12-14]和AVX[15-17]两代技术；GPGPU包括SL、CUDA、OpenCL三代技术。本文重点研究面向CPU的SIMD以及面向GPU的GPGPU这两组并行计算技术。

世界上一些著名研究机构和大学建立了专门的并行计算研究小组：

1. 斯坦福大学的普适并行实验室[18-20](The Pervasive Parallelism Laboratory, http://ppl.stanford.edu/, Stanford University)。该实验室采用应用驱动的方法使并行计算研究范围从编程模型拓展到硬件体系，核心的技术概念是特定域语言，该语言能够提升编程效率，并提供了一个集成了为并发和局部化管理提供动态或静态方案的通用并行执行环境。该实验室已经为多个领域提供了特定域的并行抽象语言，包括人工智能和机器人、商业数据分析、虚拟世界和游戏娱乐。
2. 佐治亚理工学院的Cell 宽带引擎[21-24](Cell Broadband Engine, http://sti.cc.gatech.edu/, Georgia Institute of Technology)。该引擎的研究得到索尼、东芝、IBM三家厂商的支持，并用于驱动它们的工作站服务器等大型并行计算设备。它们主要的研究目标是一台附带了多台协处理器的多核处理器，应用领域涉及图像压缩和金融模型构建。
3. 莱斯大学的哈瓦那多核软件项目[25](The Habanero Multi-core Software Project, http://habanero.rice.edu/, Rice University)。该项目主要目标是，为非专业编程人员提供便携的抽象并行技术，以较低的时间成本和努力获取多核处理器较高的性能，从而提升研究效率。项目研究对象包括抽象并行语言、编译器、并发运行库及其各种附属辅助工具，研发成果有面向并行计算的扩展语言和优化组件。

研究并行计算的主要目标是提升算法的时间效率，一直以来研究人员在总结并行计算成果时，往往仅仅参照串行单线程版本，普遍存在参照标准单一的缺陷，成果在有效性和显著性方面缺乏说服力。本文同时实现多套并行计算方案，除了分别与单线程版本对比以外，不同并行计算版本之间交叉对比彼此互为参照，为算法性能评价提供更为全面的角度。本文特别将多种并行计算方案分为两组，一组面向通用中央处理器CPU，另一组面向图形专用处理器GPU，传统GPU算法与CPU算法对比时，往往存在不公平的问题，即GPU充分优化而CPU没有优化。本文探索面向CPU的最优性能方案，为GPU性能评价提供更为公平的基准，从而使GPU性能优势更加可信，同时也为GPU并行计算方案不可行的情况提供CPU备用方案。

### 并行计算性能评价模型

经历从纯粹评价CPU多核到GPU多核，再到CPU与GPU的性能对比，最后发展到CPU与GPU异构协同并行计算。评价模型在国内外的发展过程有如下典型里程碑：

1. 最早由美国Amdahl[26]于1976年提出固定问题规模的加速比定律即Amdahl定律，该定律定义加速比S的上限为1/a，其中a为串行分量百分比；
2. 美国Gustafson[27,28]于1988年提出Gustafson定律，指出经典Amdahl定律蕴含着问题规模不变的假定，提出了固定时间加速比的概念，即当机器规模增大时使问题规模也增大的方法来获得加速比的提高；
3. Lee[29]于2010年挑战GPU拥有百倍于CPU性能的神话，证实对于大部分场合如果CPU也经过充分优化，GPU相对CPU的性能加速比在十倍以内。鉴于对并行技术是否能够真正发挥显著性能提升效果的质疑，Trinitis[30]于2012年探讨GPU编程热情是否还能持续增长。
4. 鉴于上述研究大部分关于单个任务，林一松[31]于2012年提出异构系统优化方法，关注整个程序在CPU和GPU异构体系上整体性能的优化。

传统并行计算评价模型仅针对单一并行方案，本文在多种并行方案之间建立交叉互评机制。在多模块的软件系统中，本文在系统层面评价多种并行方案组合的性能水平，引入动态规划机制在不同CPU-GPU异构平台上获取整体最优性能。

### 虚拟现实系统并行计算

计算机三维图形图像领域，基于GPGPU的并行计算技术研究热点主要集中在以下方面：骨骼动画[32,33]、粒子系统[34-38]、光线追踪[39-41]。在虚拟现实仿真平台方面，OSG[42-44]和OGRE[45]是国际上最著名最成熟的两款三维引擎，其中OSG于2005年全面支持基于GLSL的GPGPU并行计算，应用于变形动画、粒子系统、地形、光影特效；OGRE在GLSL的基础上引入了建立在OpenGL和Direct3D上一层的Cg[46,47]技术。

其中，骨骼动画是三维图形图像较为活跃的研究对象之一，Burtnyk[48]于1976年首次提出骨骼动画的概念，Lweis[49]于2000年提出姿态空间形变概念对形变动画和骨骼动画做了更高层次的抽象和统一, Lindholm[50]于2001年针对骨骼动画首次提出基于GPGPU技术实现了图像硬件并行计算加速。当时的编程语言是基于汇编的NV Vertex Program，硬件平台是Geforce3，该技术受汇编语言以及OpenGL旧版本扩展规范的影响存在很多限制，这一相当陈旧的编程技术，后来先后出现多种替代技术，包括：基于类C语言的GLSL、脱离图形API的CUDA、以及彻底脱离GPU特性的OpenCL，James[51]于2005年提出从皮肤本身萃取骨骼信息的动画方案，Baran[52]和Ju[53]分别于2007、2008年提出骨骼库的概念与骨骼自动装配的方案，Raptis[54]于2011年提出了对基于骨骼动画的舞蹈动作进行实时分类的方法。关于骨骼动画性能的研究止于Vertex Program，后来OSG和OGRE将骨骼动画算法从汇编语言移植到GLSL这一类C语言。随着GPGPU技术逐渐脱离底层图像库走向通用化，本文在此GLSL的基础上，进一步提出基于CUDA和OpenCL的两套并行计算算法；此外为了改进CPU端的参考性能水平，本文研究了基于SSE、基于OpenCL面向CPU的另外两套并行计算算法。

作为构造形状“模糊”的物体的计算模型的方法，粒子系统最早由Reeves在1983年提出[55]。其基本思想是：采用大量的、具有一定生命和属性的微小粒子图元作为基本的元素来表示不规则的模糊的物体。在粒子系统中，每一个粒子均具有：形状、大小、颜色、透明度、位置、速度和生命期等属性，所有这些属性都是时间t的函数，随着时间的变化，每个粒子都要经历“产生”、“活动”和“死亡”三个阶段。粒子系统不是一个简单的静态系统，随着时间的推移，不仅系统中已有的粒子不断运动，其属性不断变化，而且不断有新的粒子加入，超过生命期的旧粒子消失。1985年Reeves和Blau发展了粒子系统[56]，他们用volume filling基本单元去生成随时间改变形状但又基本保持不变的实体，如：随风飘动的花草树叶等。此后，Reeves又提出了关于粒子系统消隐绘制的算法，称之为“近似概率性算法”，用于解决阴影、可见面问题，取得了很好的效果。1992年Loke等人提出了节日焰火的粒子系统绘制算法，他们采用链表数据结构存储粒子的信息，设计了粒子系统绘制引擎 (Particle System Rendering Engine )，用粒子衍生的方法表现焰火粒子的轨迹并实现了多种焰火的特殊效果。Richard和David提出了基于粒子系统的变形物体表面建模方法，引入有向粒子的概念，该方法可实现变形物体表面断开、连接或延伸等效果。

## 本文组织结构

本文第一部分绪论，介绍本文研究背景和意义，总结与本文研究内容相关的国内外发展现状；第二部分介绍通用并行计算技术，CPU端计算技术包括OpenMP、SSE、CPU OpenCL，GPU端计算技术包括GLSL、CUDA、GPU OpenCL等，并以引例——二维图像加权变换算法说明各种技术的用法，定量对比各种并行技术的加速性能。该引例也是本文核心三维VR角色动画并行加速的基础，为后文做铺垫；第三部分，建立可靠严密的性能评价模型，用各种并行计算技术实现骨骼动画渲染算法，对比分析各种并行计算技术；第四部分，研究VRGIS引擎系统内多个模块同时并行计算的协同关系，使CPU和GPU异构体系发挥最优性能；第五部分，总结OpenCL等多种并行算法在VRGIS三维引擎中所发挥的功效，展望三维引擎性能优化的前景。

图1-2本文主体组织结构图

# 通用并行计算技术研究

并行计算技术按照硬件拓扑结构可以分为多个层次，包括多计算机节点的集群并行和单机内处理器多核并行，本文重点研究后者。众多并行计算技术中间，在科研领域最引人注目的是CUDA技术，以下围绕CUDA技术介绍多种并行计算技术。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CUDA同一层次的并行技术还有OpenCL(Open Computing Language)，OpenCL是一套抽象的并行计算接口规范，各种硬件厂商根据接口规范在隐藏各自硬件特性的基础上具体实现统一的接口，包括：AMD APP (Accelerated Parallel Processing，同时支持CPU和GPU)、NVIDIA OpenCL、Intel OpenCL(同时支持CPU和GPU)。OpenCL技术出现在CUDA之后，它们之间具有很强的可比性，它们都是面向NVIDIA GPU通用并行技术。

在某些场合旧有技术可能拥有更高的性能，另外按照惯例，往往把以往的旧有技术作为新技术的基准，所以进一步研究面向GPU和CPU的旧有并行技术，包括GPU端的着色语言Shading Language以及CPU端的多线程并行技术OpenMP(Open Multi-Processing)以及指令并行技术SSE(StreamingSIMDExtensions)。这两项技术的性能，代表旧有技术的最优性能。现有技术与旧有技术的对比，不仅能体现新技术的优势，而且能为辅助决策旧有技术和旧有硬件的升级提供切实的科学依据。本文采用的三维图形库是OpenGL，所以Shading Language明确为GLSL(OpenGL Shading Language)。

优化目标是最大化挖掘硬件资源利用率，主要包括CPU浮点运算能力、内存带宽，针对前者采用多线程和指令并行，针对后者采用缓存优化。本章概述各种并行计算技术的特点，为面向CPU或GPU的算法并行化提供5套方案，CPU端包括SSE叠加OpenMP、CPU OpenCL，GPU端包括GLSL、CUDA、GPU OpenCL等，并以引例——二维图像加权变换算法辅助说明各种技术的特点，定量对比各种并行技术的加速性能，该引例也是本文核心内容之一即骨骼动画并行计算的基础。

图2-1 并行计算技术汇总

## 并行计算性能评价模型

### 基于阿姆达尔定律的性能评价理论模型

将目标程序抽象为一个完整的系统，程序内部调用的子程序抽象为系统的子模块，此时，程序性能可用一般的系统性能模型来定量评价。阿姆达尔定律（Amdahl's Law）是定量评价计算机系统性能的重要理论，该理论的创始人是IBM360机型的设计者阿姆达尔Amdahl，最早于1967年率先提出，后来命名阿姆达尔定律即Amdahl's Law。该定律描述为：系统中对某一子模块采用更快执行方式所能获得的系统性能改进程度，取决于这种执行方式被使用的频率，或所占总执行时间的比例。阿姆达尔定律定义了某个子模块性能提升后，使得整个系统获得性能提升的幅度。阿姆达尔定律可用以描述硬件系统和软件系统，描述软件系统时，性能通常指时间性能，性能改进指系统运行时间的加速比，即原来时间与改进后时间的比值。阿姆达尔定律可以定量评估局部性能向整体性能的传递效应，系统性能改进倍数可用如下公式表示：

A=1/(1-P+P/S) 式（2-1）

其中，P指性能改进模块运算量比例，1-P指剩余模块运算量比例，S指模块局部加速比，A指性能改进倍数或系统加速比。系统和子模块的关系是相对的，系统对于上一层级是一个子模块，而子模块对于进一步细分的下一级来说也是系统。所以，阿姆达尔定律只描述相邻两个层级的性能传递关系，两个层级以上的性能传递关系，需要嵌套使用阿姆达尔定律。

### 性能指标的选定及其准确度的考虑

两个指标分别是加速比A和帧速增幅D，因为比值和增幅都从直接可度量的时间参数衍生计算得到，所以把时间参数作为指标参数，时间T和帧速F互为倒数关系。时间，指的是执行一个算法所耗费的CPU执行时间；帧速，指单位时间内不间断循环执行一个算法的次数。以时间T作为指标参数时，设若加速前，系统的总耗时是T，性能改进模块的耗时是Tp，性能改进模块的耗时Tp’。阿姆达尔定律的公式参数可从以上时间指标计算得到，从而获取局部核心加速比和全局系统加速比。其中，P=Tp/T，S = Tp/Tp’，A= 1/(1-P+P/S) = T/(T-Tp+Tp’) = T/T’。假设T=100ms，Tp=90ms，Tp’=10ms，得到局部加速比为9，全局加速比为5。除加速比以外，帧速的增幅是另一个反映性能改进程度的指标，D = ΔF/F = (F’-F)/F = (1/T’-1/T)\*T = T/T’-1 = A – 1，即D = A - 1。增幅一般用%标识，所以在最后结果的基础上补充后缀\*100%。加速比比较适合作为局部加速性能的指标，衡量加速的倍数；帧速的增幅比较适用于评价整体性能的改进幅度。将加速比和帧速两者结合使用，从微观和宏观两个层面评价性能加速效果。

指标是通过时间参数衍生计算出来的，所以获取指标的前提是度量时间。获取时间有两类方法，插入测时代码手动获取，采用外部工具自动获取。如果采用手动获取时间的方法，对于CPU单线程、CPU多线程、GPU渲染，有针对它们各自的获取指标即计时方法。Windows提供微秒级精度的测时API方法QueryPerformanceCounter和QueryPerformanceFrequency，UNIX提供了gettimeofday。CPU多线程有专门的测时API方法omp\_get\_wtime。此外，还有其它时间精度的测试方法，time精确到秒s，GetTickCount精确到毫秒ms，rdtsc精确到纳秒ns。

本文采用OpenGL实现三维渲染，测量OpenGL指令时间需要特别注意三点。第一点需要注意的是，多数OpenGL指令相对CPU异步执行，也就是CPU调用OpenGL指令后立即返回，无法通过为CPU计时的传统方法进行计时。通过CPU调用的OpenGL指令并不是立即执行，而是在调用SwapBuffers后才执行完成，SwapBuffers会阻塞CPU直等到当前累积的OpenGL指令执行完成，这决定了程序员无法单独获取某个OpenGL流水线阶段或者某条OpenGL指令的时间，只能通过SwapBuffers获取一帧内整个渲染过程的总时间。第二点需要注意的是，即使OpenGL指令执行完成，SwapBuffers也不是立即返回，还有一个延迟，这个延迟取决于垂直同步间隔时长(Vsync Intelval)即显示器的刷新周期，比如刷新周期1/75秒=13.3ms，意味着无论实际耗费时间是多少，测得的时间至少是13.3ms，导致测时结果与实际不符。在测量时间时用glFinish替代SwapBuffers，将准确测得OpenGL指令时间。第三点要注意的地方是，CPU和GPU执行过程存在时间重叠。因为GPU作为CPU的从属设备，CPU和GPU之间相当于服务器和客户端关系，GPU异步执行指令，导致CPU和GPU执行过程不可避免存在时间重叠，重叠时间在统计过程需要特别考虑，如果忽略的话会导致测时误差。

如果采用自动获取时间的方法，对于CPU单线程和多线程、GPU渲染，也有针对它们各自的获取指标即计时方法。常用的CPU测时工具有Intel提供的Intel VTune Amplifier，VTune 可以作为插件集成在Visual Studio里面。GPU测时工具有NVIDIA Visual Profiler。本文同时采取代码手动测量与工具自动测量两种方式，确保时间测量的准确性。

本文采用以上测时方法，分别测量CPU读写内存的带宽以及GPU渲染速度。得到这两组数据，为后期做准备。CPU读写内存时，设计2个动态长度的浮点数组，遍历每个数组元素，从一个数组读取再写入另一个数组，实现数组拷贝。GPU渲染时，设计动态长度的表征三维坐标的三元组数组，存入VBO进行渲染。实际测量时采用适合精度范围的测试方法，在代码片段始末嵌入相应代码，比如数组较小时耗时较小，采用微妙精度计时器QueryPerformanceCounter；数组较大时，采用毫秒计时器GetTickCount或秒计时器time。

本文采用以上测时方法测试了多线程并行和多指令并行对内存带宽的影响，发现数据量小于8M（三级缓存大小）时，多线程带宽提升2至3倍；大于8M时，多线程带宽降低8%。数据量小于8M（三级缓存大小）时，多线程带宽提升2倍；大于8M时，指令并行带宽略微提升。多线程并行与多指令并行合并使用时对带宽的影响，数据量小于8M（三级缓存大小）时，多线程带宽提升2至3倍；大于8M时，并行带宽降低25%。综上，以80M数据量为例，原始内存带宽是8.1GB/s，多线程并行带宽是7.3%，多指令并行带宽是8.6GB/s，两层并行带宽是5.4GB/s，也就是说，多线程并行获取加速的同时需要牺牲一定的内存效率作为代价，而多指令并行不会影响内存效率。

指标输出可以采用控制台实时监控以及日志文件统计两种方式。本文记录的指标是程序模块的运行时间，包括总时间和子模块局部时间。为了提升指标的精确度和稳定性，本文从大量时间样本采样获取平均值，只有当偏差系数即均方差与均值比值小于5%时，才作为有效记录进行统计。对于控制台方式，鉴于VR三维平台运行在窗体中，窗体中无法直接输出信息到控制台，所以需要在窗体以外额外开辟控制台窗口，用以接受输出，实时查看和统计程序性能指标。Windows窗体通过AllocConsole/ FreeConsole开关控制台，并用语句"freopen( "CONOUT$","w",stdout)"使窗体和控制台建立联系，之后在窗体调用输出信息控制台的指令，可以从控制台看到相应信息。

性能峰值受具体问题的瓶颈和硬件性能限制，通过细化问题模块，分区间量化模块所占比重，可以找出瓶颈位置。另外，具体硬件的运算能力，也将使性能限定在一定的理论峰值范围之内。正确估价性能峰值，能够为优化过程提供一个明确的目标，为优化工作的阶段成果和后续工作展望提供评价依据。

性能问题常常发生在大数据量高密度运算的场合，按照数据流方向，将程序划分为若干子模块，数据在子模块之间单向传递。测量整个程序的耗时以及各个子模块的耗时，得到耗时分布图，进而算出各个模块耗时占总耗时的比例，比例较高者作为候选瓶颈。瓶颈暴露的地方，往往跟系统负载有关。程序各个子模块对问题复杂度的敏感程度不同，有些模块的耗时始终是常数，跟问题复杂度没有关系；有些模块的时间花费跟问题复杂度呈斜率或大或小的线性关系；有些呈折线关系，有些只能以统计方法拟合它们之间的关系。本文选取多个离散的问题复杂度，获取它们的瓶颈分布图。为便于研究，文本先从中选取一个问题复杂度，从而明确它的瓶颈，进行针对性的性能改进，获取性能改进指标。最终扩展到多个问题复杂度，分别沿用相同性能改进原则和步骤，得到各自的性能改进指标，总结本文性能优化方案对问题复杂度的适用范围。

全局加速比并非无止境的提升，优化的目标和终点是逼近理论峰值加速比，根据阿姆达尔定律及其公式，待优化模块性能无限提高即Tp’->0、S->∞时，整体系统的加速比达到峰值即A->1/(1-P)=T/(T-Tp)。也就是说，性能不可能无限提升，最终的峰值性能受限于未能提升性能的其余模块所占的比重1-P。例如1-P=10%，则峰值加速比为A=10，即原先需要10秒才能执行完的操作，现在只需要1秒，而且至少需要1秒。局部加速比也受到制约，主要受到具体硬件特性影响，包括处理器物理核心数目和线程数目及其浮点运算能力GFLOPS、内存带宽、缓存cache容量和结构。比如对于4核8线程的CPU，假设算法充分发挥运算能力和内存带宽，多线程发挥8倍加速比，指令并行发挥4倍加速比，总体上理论加速比是S=32。局部加速比的理论峰值可以作为核心算法加速的参照目标。对于指定的问题和选定的硬件，可以在优化前，初步估测全局加速比和局部加速比的理论峰值，作为性能加速研究实验的参考目标，并作为评价性能优化程度的相对基准。比起众说纷纭的绝对加速比和帧速增幅，以实测性能与理论峰值性能的逼近程度作为衡量优化成果的标准，将更加公平。

程序并行化分自动和手动两种方式，根据编译原理，现有编译器可以自动完成某些通用的优化工作，也有研究人员设计特定领域高级语言完成优化细节的封装，这些优化统称为自动并行优化。相对而言，研究人员根据硬件特性和具体软件问题特性，手动通过遍历各自参数的设置，迭代寻找最优解，这一过程属于手动优化。自动优化的特定是，优化操作简单，不需要懂得硬件和软件细节特性，缺点是无法最大化使用硬件资源从而获取最优性能；手动优化的特定是，优化过程复杂，需要同时了解硬件和软件特性，能够最大化挖掘硬件资源，从而获取最有效性能，缺点是优化周期长，移植性差。自动优化和手动优化之间的关系并非完全孤立，自动优化的基础是手动优化，把手动优化过程的通用原则加以归纳总结，形成自动优化的逻辑。本文为了最大化探究性能极限，故采用手动优化方式。

移植性包括硬件和软件两方面，首先，硬件方面，对本文研究的几个特例，包括骨骼动画和三维拓扑分析，本文设计的性能优化原则，力求在选定的处理器硬件平台上，能够发挥最佳性能，即便换用不同厂家不同型号的处理器，也照样能发挥最佳性能。这是跨越硬件平台的可移植性。其次，本文设计的性能优化原则，在VR平台的其他问题领域，也同样普遍适用，比如其它类型的三维动态仿真和三维分析，从而通过以上两个典型特例，援用其中的技术，全面优化VR仿真平台的各个功能模块的性能。

### 并行计算多方案交叉对比改进性能评价模型

在阿姆达尔定律的理论基础上，引入在多种并行计算方案之间的交叉互评机制，完善并行计算性能评价模型。本文研究的五套并行计算方案中，最新的OpenCL技术可以同时用于中央处理器CPU和图形处理器GPU，此外CPU和GPU有各自专用的并行计算技术，其中SSE和OpenMP专门面向CPU，GLSL和CUDA专门面向GPU。以下本文展开介绍这五种并行计算技术。

为了更好地描述各种并行优化技术，在介绍各自并行计算技术的同时，本文引入一个具体的典型示例——图像变换算法。图像变换算法是图形学最基本的算法之一，早有成熟的算法实现，著名的OpenCV开源库以最成熟的算法集实现了图像处理设计的各种算法，OpenCV已经实现其CPU多线程版本以及GPU多线程版本。本章重点不是算法本身，而是藉着这一算法辅助表现OpenMP多线程并行、SSE指令并行、面向CPU的OpenCL这几种优化方案的区别与联系，便于后文进一步把相应算法从二维图像过渡到三维VR图像，并从静态单帧图像变换过渡到动态多帧连续图像变换，即三维动画。

根据图形学基本原理，图像变换算法用于对图像的像素坐标(x,y)进行平移T、缩放S、旋转R三种操作，这三种操作可以一致地用以下矩阵表征：

T(Δx, Δy) =, S(a, b) = , R(θ) = 对坐标P(x, y)图像变换通过以下坐标变换公式实现：

P\* T = \* = 式（2-2）

P \* S = \* 式（2-3）

P \* R = \* 式（2-4）

如果仅仅是三种操作中的一种，变换矩阵最后1列是多余的，用3\*2矩阵就可以。但为了支持以上操作的叠加，变换矩阵增加1列变成3\*3的形式，多种变换叠加时，仅需对变换矩阵按一定顺序执行相乘操作即可。变换矩阵最终是统一为一个3\*3矩阵mat[3][3]，同时包含平移缩放旋转信息。统一矩阵示意图如下：

对应统一矩阵变换公式为：

P\*T =\*= 式（2-5）

本章以图像变换为例，一幅长宽分别为W/H个像素的图像，顺时针旋转θ°，再放大s倍，得到一幅新的图像，C/C++算法如下：

/\*

功能，图像统一变换

参数imgIn：原始图像，长宽像素个数分别为W和H，为提升坐标精度用浮点存储

参数imgOut：旋转后新图像，长宽同原始图像

参数m：统一变换矩阵

\*/

void transform2Duniform ( float imgIn[][W][2], int H, float imgOut[][W][2], float m[][3] )

{

for( int i = 0; i < H; i ++ )

for( int j = 0; j < W; j++ )

{

imgOut[i][j][0] = imgIn[i][j][0]\*m[0][0]+imgIn[i][j][1]\*m[1][0]+m[2][0];

imgOut[i][j][1] = imgIn[i][j][0]\*m[0][1]+imgIn[i][j][1]\*m[1][1]+m[2][1];

}

}

/\*

功能，图像旋转后缩放

参数imgIn：原始图像，长宽分别为W/H个像素

参数imgOut：旋转后新图像，长宽同原始图像

参数theta：旋转角度，单位°，需换算成弧度rad=theta /180\*π

参数scale：缩放倍数，宽高同样的倍数

\*/

void transform2D (float imgIn[][W][2], int H, float imgOut[][W][2], float theta, float scale )

{

// 第一步，初始化变换矩阵

floatm[3][3] = { {0, 0, 0}, {0, 0, 0}, {0, 0, 0} } ;

// 第二步，旋转矩阵，参考公式2-4 …

// 第三步，缩放矩阵，参考公式2-3 …

// 第四步，执行统一变换

transform2DUniform( imgIn, H, imgOut, m );

}

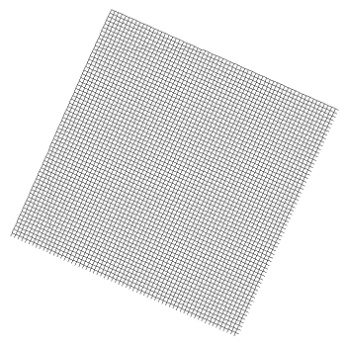
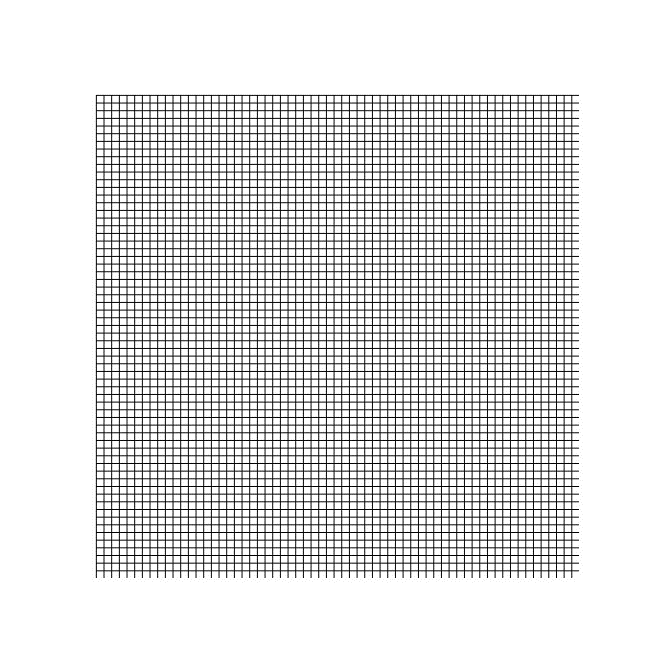
假设图像像素长宽W=H=1024，角度15，倍数1.05，执行以上算法，测量运行时间。然后改变图像像素长宽，测量时间，得到以下列表：

表2-1图像变换时间表

|  |  |
| --- | --- |
| 像素数目 | 耗时ms未优化 |
| 1024 | 5.8 |
| 2048 | 23 |
| 4096 | 94 |
| 8192 | 290 |

表2-2 内存读写时间表串行算法

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 像素  数目 | 数据量  MB | 带宽  GB/s | | 耗时  ms | |
| 1024 | 16 | | 9.8 | | 1.6 |
| 2048 | 64 | | 8.5 | | 7.5 |
| 4096 | 256 | | 7.2 | | 36 |
| 8192 | 1024 | | 7.2 | | 142 |



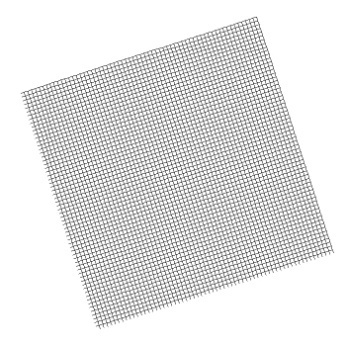
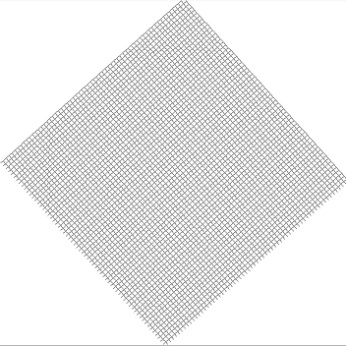


图2-2图像变换效果图

在将串行算法转换为并行算法之前，需要对串行算法本身进行优化，以求达到最优性能，从而为后文各种性能优化提供坚实而公正的参照基准。此外，优化串行算法不仅为了提升性能，也为简化算法结构，为后文的并行化改造作预备。本文继续以上文图像变换算法为例，为了使串行算法达到最优性能，并为后续并行化提供精简代码结构，本节力求在串行算法上做基本的优化。将内层循环封装成一个核心函数kernel，单独处理一行像素的坐标，性能提升。算法改进及性能改进如下：

void kernel ( float imgIn[][W][2], float imgOut[][W][2], float m[][3], int i )

{

for( int j = 0; j < W; j++ )

{

imgOut[i][j][0] = imgIn[i][j][0]\*m[0][0]+imgIn[i][j][1]\*m[1][0]+ m[2][0];

imgOut[i][j][1] = imgIn[i][j][0]\*m[0][1]+imgIn[i][j][1]\*m[1][1]+ m[2][1];

}

}

表2-3 图像变换时间表

|  |  |  |  |
| --- | --- | --- | --- |
| 像素  数目 | 耗时ms  未优化 | 耗时ms  串行优化 | 加速比 |
| 1024 | 5.8 | 3.1 | 1.9 |
| 2048 | 23 | 12 | 1.9 |
| 4096 | 94 | 50 | 1.9 |
| 8192 | 290 | 202 | 1.4 |

表2-3结合表2-2，发现串行优化后内存访问时间占总时间50%~70%，根据实验发现内存访问无法通过并行优化加速，无论是线程并行还是指令并行，所以程序可优化的空间位于剩余的30%~50%。

## 基于SSE面向CPU的通用并行计算

### OpenMP多线程任务级并行

对串行算法本身经过有效优化之后，单个线程在单个CPU物理核心上执行时，将接近理论最高性能。每个CPU常常含有4个甚至更多的核心处理器单元，开发多线程并行程序理论上将获得跟核心数目一样多的加速比。比如包含物理4核并支持物理4线程的处理器将获得的加速比为4，包含物理4核并支持物理8线程的处理器将获得的加速比为8。

OpenMP是一个开放的多线程方法库，在不同的CPU硬件厂商型号之间，以及不同的编译器软件之间，不同的操作系统之间，都具有良好的代码可移植性和性能可移植性。OpenMP已内置到最常用的几种编译器如Windows的VC和Linux的GCC以及Mac。OpenMP可以快速地在多线程和单线程两种状态之间进行切换，而程序员只需要编写很少的代码。对于算法结构不是很复杂而算法运算量巨大的情形，特别适合选用OpenMP。此外还可以使用其它多线程库，如Intel TBB(Threading Building Block)，以满足更为复杂多样的多线程编程需求。本文在Windows 7操作系统下通过VC语言调用OpenMP实现算法的多线程转化，转化后的多线程算法可以很方便地移植和重用。

调用OpenMP无需额外的库，只需在编译时添加编译选项"/openmp"，或在VC工程属性中开启"OpenMP Support"设置，该设置位于C/C++语言Language页面，将"OpenMP Support"由默认的"No"改为"Yes"。接着在需要执行多线程的文件头引入”omp.h”，接着就可以调用OpenMP提供的API方法和#pragma omp系列宏指令，这些方法包括修改线程数目、获取线程id等等，#pragma omp指令控制多线程的启停。OpenMP最常用的一个场合是并行执行循环，将循环体拆分到多个线程同步并行执行，实现这种多线程，只需要调用一条#pragma omp指令，即在for语句前一行调用"#pragma omp parallel for "。串行代码改为并行代码，示意如下：

//串行代码：

for( int i = 0 ; i < 1000000; i ++ )

{ …//loop body循环体 }

//并行代码：

#pragma omp parallel for

for( int i = 0 ; i < 1000000; i ++ )

{ …//loop body循环体 }

可见，使用OpenMP非常简单，但是不见得所有程序都能够并行化，而且有些程序即便并行化也不能获得性能提升，若要发挥多线程的并行加速能力，算法应符合以下特征。首先，线程之间很少通信或没有通信，避免线程依赖，一个线程等待另一线程的运行结果；其次，线程之间数据共享很少或没有共享，避免资源竞争，导致线程处于等待状态耗费时间，尽量通过private子句将多线程外部引入的全局变量私有化；第三，相邻线程的数据访问地址最好连续或相近，充分发挥缓存数据的预读和重用效率。

将以上描述的OpenMP并行技术用于上节图像旋转算法，有三个方案，第一种：并行指令用于外层循环；第二种：并行指令用于内层循环；第三种，先展开2层循环变为一层，再使用并行指令。3种改进方案的算法如下：

方案一：

void transform2Duniform ( float imgIn[][W][2], int H, float imgOut[][W][2], float m[][3] )

{

#pragma omp parallel for

for( int i = 0; i < H; i ++ )

kernel (imgIn, H, imgOut, m , i) ;

}

方案二：

void kernel ( float imgIn[][W][2], int H, float imgOut[][W][2], float m[][3], int i )

{

#pragma omp parallel for

for( int j = 0; j < W; j++ )

{

imgOut[i][j][0] = imgIn[i][j][0]\*m[0][0]+imgIn[i][j][1]\*m[1][0]+m[2][0];

imgOut[i][j][1] = imgIn[i][j][0]\*m[0][1]+imgIn[i][j][1]\*m[1][1]+m[2][1];

}

}

方案三：

void transform2Duniform ( float imgIn[][W][2], int H, float imgOut[][W][2], float m[][3] )

{

float \*pIn = &m\_imgIn[0][0][0];

float \*pOut = &m\_imgOut[0][0][0];

#pragma omp parallel for

for( int i = 0; i < SIZE\_HEIGHT\*SIZE\_WIDTH; i ++ )

{

pOut[2\*i] = pIn[2\*i]\*m[0][0] + pIn[2\*i+1]\*m[1][0] + m[2][0];

pOut[2\*i+1] = pIn[2\*i]\*m[0][1] + pIn[2\*i+1]\*m[1][1] + m[2][1];

}

}

表2-4图像变换时间表 OpenMP方案一

|  |  |  |  |
| --- | --- | --- | --- |
| 像素  数目 | 耗时ms  串行优化 | 耗时ms  线程并行 | 加速比 |
| 1024 | 3.1 | 3.2 | 1.0 |
| 2048 | 12 | 13 | 0.9 |
| 4096 | 50 | 52 | 1.0 |
| 8192 | 202 | 208 | 1.0 |

表2-5 图像变换时间表 OpenMP方案二

|  |  |  |  |
| --- | --- | --- | --- |
| 像素  数目 | 耗时ms  串行优化 | 耗时ms  线程并行 | 加速比 |
| 1024 | 3.1 | 3.2 | 1.0 |
| 2048 | 12 | 14 | 0.9 |
| 4096 | 50 | 55 | 0.9 |
| 8192 | 202 | 246 | 0.8 |

表2-6 图像变换时间表 OpenMP方案三

|  |  |  |  |
| --- | --- | --- | --- |
| 像素  数目 | 耗时ms  串行优化 | 耗时ms  线程并行 | 加速比 |
| 1024 | 3.1 | 3.1 | 1.0 |
| 2048 | 12 | 13 | 0.9 |
| 4096 | 50 | 55 | 0.9 |
| 8192 | 202 | 208 | 1.0 |

从以上表格可以看出，采用OpenMP对图像变换算法进行多线程优化，并没有起到优化效果。

使用OpenMP进行多线程优化时，有一点需要特别注意，编译选项/openmp开启后，即便在代码中不调用任何OpenMP并行宏，编译器会智能并行优化，编译器优化的结果往往不如不优化。所以不使用OpenMP时，不仅仅在代码中清除OpenMP宏和其它指令，还需要把编译选项/openmp去除或在工程属性中设为false。否则编译选项会影响单线程算法的性能，导致多线程与单线程的对比结果失真。

### 线程内SSE指令级并行

现代CPU拥有多个长度为128位的寄存器单元，支持一个指令周期同时操作128位数据，即4个float或2个double，1条加指令可以同时作用于4对float加数和被加数。CPU指令集提供相应的数据类型和指令，支持同时操作多个操作数。SSE(Streaming SIMD Extensions，流式单指令多数据扩展)系列指令集是Intel设计的支持128位操作数的指令集，常用指令及其操作数和功能如下表，完整SSE指令声明详见xmmintrin.h及Intel官方文档。

表2-7常用SSE指令及其操作数和功能

|  |  |  |  |
| --- | --- | --- | --- |
| 指令名称 | 操作数 | 返回值 | 功能 |
| \_mm\_load\_ps | float const\*\_A | \_\_m128 | 一次性读4个float |
| \_mm\_mul\_ps | \_\_m128 A, B | \_\_m128 | 4个float相乘 |
| \_mm\_add\_ps | \_\_m128 A, B | \_\_m128 | 4个float相加 |
| \_mm\_store\_ps | floatconst\*\_V, \_\_m128 A | void | 一次性写4个float |

SSE指令早在1999年的Pentium 3与2000年的Pentium 4处理器中已经被采用，所以说现在的绝大多数处理器都支持SSE指令集。在2011年推出的Sandy Bridge系列处理器，Intel将128位的SSE指令集升级为256位AVX(Advanced Vector Extensions)指令集。AVX系列指令集是Intel设计的支持256位操作数的指令集，常用指令及其操作数和功能如下表，完整AVX指令声明详见immintrin.h及Intel官方文档。

表2-8常用SSE指令及其操作数和功能

|  |  |  |  |
| --- | --- | --- | --- |
| 指令名称 | 操作数 | 返回值 | 功能 |
| \_mm256\_load\_ps | float const\*\_A | \_\_m256 | 一次性读8个float |
| \_mm256\_mul\_ps | \_\_m256 A, B | \_\_m256 | 8个float相乘 |
| \_mm256\_add\_ps | \_\_m256 A, B | \_\_m256 | 8个float相加 |
| \_mm256\_store\_ps | floatconst\*\_V,\_\_m256 A | void | 一次性写8个float |

本文采用SSE多指令实现算法在指令级别的并行化改造，将以上描述的SSE多指令并行技术用于上节图像旋转算法。改进后算法如下：

// 每次处理2个相邻点 (x1, y1, x2, y2)

for (int j=0;j<SIZE\_WIDTH/2 ;j++)

{

\_\_m128 vIn = pSrcPos[j+nOffset];

\_// (x1,x1,x2,x2) (y1,y1,y2,y2)

\_m128 vInx = \_mm\_shuffle\_ps( vIn, vIn, \_MM\_SHUFFLE(2,2,0,0) );

\_\_m128 vIny = \_mm\_shuffle\_ps( vIn, vIn, \_MM\_SHUFFLE(3,3,1,1) );

\_\_m128 tmp00 = \_mm\_mul\_ps( m0, vInx );

\_\_m128 tmp01 = \_mm\_mul\_ps( m1, vIny );

\_\_m128 vOut0 = \_mm\_add\_ps( tmp00, tmp01 );

vOut0 = \_mm\_add\_ps( vOut0, m2 );

pDestPos[j+nOffset] = vOut0 ;

}

为了支持SSE操作并发挥最优性能，SSE所访问的内存必须是按照16字节对齐，图像及矩阵对应的指针通过\_aligned\_malloc/\_aligned\_free 进行堆内存的动态分配和释放，对于静态数组，需要用到指示符" \_\_declspec(align(16))"将数组显示声明16字节对齐。16字节对齐并不是强制的，SSE提供另一套支持非对齐的内存读写指令，只是性能不如对齐方式（下降10%）。

假设图像像素长宽W=H=1024，角度15，倍数1.05，执行以上算法，测量运行时间。然后改变图像像素长宽，测量时间，得到以下列表：

表2-9图像变换时间表 SSE指令并行

|  |  |  |  |
| --- | --- | --- | --- |
| 像素  数目 | 耗时ms  串行优化 | 耗时ms  指令并行 | 加速比 |
| 1024 | 3.1 | 2.8 | 1.1 |
| 2048 | 12 | 11 | 1.1 |
| 4096 | 50 | 45 | 1.1 |
| 8192 | 202 | 187 | 1.1 |

从表中看出，SSE指令并行后，图像变换算法加速比是1.1，即小幅度提升。

### OpenMP与SSE双层嵌套并行

将以上描述的两层并行技术——OpenMP线程并行和SSE指令并行，同时用于图像旋转算法，改进后算法如下：

void kernelSSE( float\* imgIn, float\* imgOut,\_\_m128& m0,\_\_m128& m1,\_\_m128& m2, int i )

{

\_\_m128 \*pSrcPos = (\_\_m128\*)imgIn;

\_\_m128 \*pDestPos = (\_\_m128\*)imgOut;

int nOffset = i\*SIZE\_WIDTH/2

// 每次处理2个相邻点 (x1, y1, x2, y2)

#pragma omp parallel for private(nOffset)

for (int j=0;j<SIZE\_WIDTH/2 ;j++)

{

// …同上

}

}

假设图像像素长宽W=H=1024，角度15，倍数1.05，执行以上算法，测量运行时间。然后改变图像像素长宽，测量时间，得到以下列表：

表2-10 图像变换时间表OpenMP与SSE双层嵌套并行

|  |  |  |  |
| --- | --- | --- | --- |
| 像素  数目 | 耗时ms  串行优化 | 耗时ms  完全并行 | 加速比 |
| 1024 | 3.1 | 3.0 | 1.0 |
| 2048 | 12 | 13 | 0.9 |
| 4096 | 50 | 51 | 1.0 |
| 8192 | 202 | 201 | 1.0 |

表2-11 内存读写时间表OpenMP与SSE双层嵌套并行

|  |  |  |  |
| --- | --- | --- | --- |
| 像素  数目 | 数据量  MB | 带宽  GB/s | 耗时  ms |
| 1024 | 16 | 7.8 | 2.1 |
| 2048 | 64 | 6.0 | 11 |
| 4096 | 256 | 5.3 | 48 |
| 8192 | 1024 | 5.3 | 193 |

综合以上2表，可见，当完全采用多线程多指令双层并行时，算法耗时大部分(90%)是由内存读写引起，内存读写这部分无法进行加速。而多线程多指令环境下，内存带宽相对未优化时降低25%，最终导致多线程多指令双层并行优化，没有起到实际效果。

如果要发挥并行性能，算法指令执行时间与内存读写时间比例必须达到一定数值。以下举2个例子，分别是平方根算法和向量求模算法。

例1平方根算法。算法指令执行时间远远超过内存访问时间，经实验测定时间比例是10:1，而且指令执行和内存访问同步进行，时间重叠，访问内存的时间可以忽略不计。对这一算法进行并行加速，无论线程并行还是指令并行，都能发挥较高性能。

表2-12平方根算法串行耗时分布

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据量  MB | 算法总耗时ms | 访存耗时  ms | 指令耗时  ms | 指令耗时  比率% |
| 80 | 91 | 8 | 90 | 99% |

表2-13 平方根算法并行耗时分布

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 原始耗时  ms | 线程并行  ms | 指令并行  ms | 完全并行  ms | 加速比  即时间比 |
| 91 | 25 | 13 | 15 | 6.0 |

例2向量求模算法。算法指令执行时间远远超过内存访问时间，经实验测定时间比例是10:1。对这一算法进行并行加速，无论线程并行还是指令并行，都能发挥较高性能。

表2-14 向量求模算法串行耗时分布

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据量  MB | 算法总耗时ms | 访存耗时  ms | 指令耗时  ms | 指令耗时  比率% |
| 120 | 92 | 18 | 91 | 99% |

表2-15 向量求模算法并行耗时分布

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 原始耗时  ms | 线程并行  ms | 指令并行  ms | 完全并行  ms | 加速比  即时间比 |
| 92 | 25 | 18 | 21 | 4.4 |

通过以上表格的实验数据，证实当算法指令执行时间占算法大部分时间的情况下，并行加速效果显著。在考虑对一个算法进行并行加速前，需要评估算法指令执行时间的比例，根据前文提及的阿姆达尔定律，只有比例较高时，指令优化获取性能提升对全局性能的提升才有较大的贡献。

考虑到“简易图像变换算法”不适于多线程加速。本文对算法一些变更，变更后的算法成为“加权图像变换算法”。变更如下：之前整幅图像按照一样的变换矩阵进行变换，现在给变换矩阵添加一个动态权重，离中心越近变换幅度越大，反之幅度越小，权重设为距离对数的倒数。原始算法及变更算法对比如下：

// 图像变换原始算法

void kernel (float imgIn[][W][2], int H, float imgOut[][W][2], float m[][3], int i)

{

for( int j = 0; j < W; j++ )

{

imgOut[i][j][0] = imgIn[i][j][0]\*m[0][0]+imgIn[i][j][1]\*m[1][0]+m[2][0];

imgOut[i][j][1] = imgIn[i][j][0]\*m[0][1]+imgIn[i][j][1]\*m[1][1]+m[2][1];

}

}

// 图像变换变更算法——加权

void kernel ( float imgIn[][W][2], int H, float imgOut[][W][2], float m[][3], int I, float rad )

{

for( int j = 0; j < W; j++ )

{

float x = imgIn[i][j][0];

float y = imgIn[i][j][1];

// 按照权重计算每个像素点的旋转角度

float offset = 4.0f;

float distance = sqrt( x\*x + y\*y +offset );

float weight = 1.0f / sqrt(distance) ;

float radOne = rad \* weight;

// 第一步，初始化变换矩阵

floatm[3][3] = { {0, 0, 0}, {0, 0, 0}, {0, 0, 0} } ;

// 第二步，旋转矩阵，参考公式2-4 …

// 第三步，缩放矩阵，参考公式2-3 …

// 第四步，执行统一变换

imgOut[i][j][0] = x \* mOne[0][0] + y \* mOne[1][0] + mOne[2][0];

imgOut[i][j][1] = x \* mOne[0][1] + y \* mOne[1][1] + mOne[2][1];

}

按照前文方法，对变更后的“加权图像变换算法”进行多线程并行和多指令并行优化，测得性能如下表：

表2-16加权图像变换算法并行耗时

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 像素 | 耗时ms  串行优化 | 耗时ms  多线程并行 | 耗时ms  多指令并行 | 耗时ms  完全并行 |
| 1024 | 92 | 19 | 51 | 12 |
| 2048 | 364 | 76 | 204 | 46 |
| 4096 | 1460 | 308 | 813 | 184 |
| 8192 | 5800 | 1180 | 3250 | 720 |

从以上图表得出结论，对于加权图像变换算法这一类运算密集型算法，线程并行和指令并行都能获取较好加速效果，综合起来加速效果最理想。本文测得OpenMP线程并行加速比接近5，SSE指令并行加速比接近2，综合并行加速比接近8。

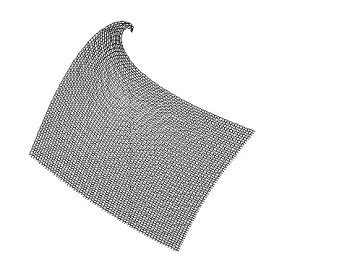
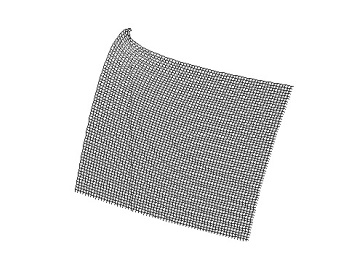
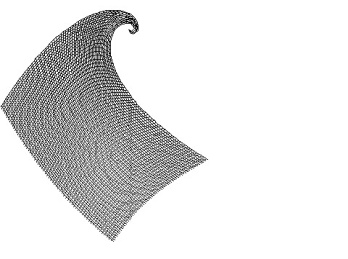
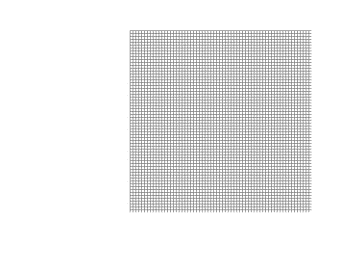


图2-3加权图像变换效果

## 基于OpenCL面向CPU和GPU的异构通用并行计算

### OpenCL技术概览

OpenCL全称Open Computing Language开放计算语言，“计算”通常特指“并行计算Parallel Computing”。与之前介绍的并行方案不同，OpenCL不仅适用于CPU而且适用于GPU和其它多处理器设备。专门面向CPU的OpenCL并行方案，具体底层的实现隐式调用之前的多线程并行及多指令并行，但编程人员只需按OpenCL规范编写程序，无需手动调用之前的多线程（例如OpenMP）以及多指令（例如SSE）。OpenCL是一个标准的规范框架，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。OpenCL基于C语言设计了独特的编程语言，用于编写运行在OpenCL设备端的代码指令，这些指令封装成kernel函数，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。OpenCL是一套开放的抽象标准，已由各种主流处理器进行实现，各种主流处理器发不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，使得基于OpenCL规范编写的并行程序可以在Intel、AMD、NVIDIA的CPU和GPU上通用。

OpenCL规范最早由Apple苹果公司于2008年6月率先提出，并由开放组织Khronos于当年11月发布正式技术规范，并被开放组织Khronos内包括Intel、AMD、NVIDIA、IBM等成员接受并具体实现。OpenCL自从2008年发布至今，历经多个版本不断完善，2008年OpenCL 版本1.0、2010年版本1.1、2011年版本1.2、2013年版本2.0。OpenCL规范一经推出，包括Intel、AMD、NVIDIA在内的各家主流硬件商随即推出相应的软硬件支持，并随着OpenCL版本的升级而升级。2009年11月，NVIDIA发布CUDA SDK v3.0及相应驱动支持OpenCL 1.0，运行于GPU；NVIDIA最新版本2013年8月CUDA 5.5支持OpenCL 1.1。2009年12月，AMD发布ATI Stream SDK v2.0及相应驱动支持OpenCL 1.0，运行于GPU；AMD最新版本2013年7月APP 2.8.1支持OpenCL 1.2。2010年9月，Intel 发布Intel OpenCL SDK支持OpenCL 1.0，运行于Sandy Bridge架构的CPU；Intel最新版本2013年2月Intel OpenCL SDK 2013支持OpenCL 1.2。

用于编写kernel函数的编程规范基于C99版本的C语言，在C语言的基础上有少量裁剪，并扩展新增一些特有的新特性。裁剪部分包含：函数指针、递归、位域、变长数组等等；新增部分包含：向量类型及其操作、同步、任务项/任务组配置函数，为了区分OpenCL设备多样的存储位置新增存储域指示符\_\_global、\_\_local、\_\_constant、\_\_private，分别表示全局共享存储、线程局部存储、全局常量存储、线程私有存储。

OpenCL程序包括以下三个部分：OpenCL设备初始化；OpenCL参数指定；OpenCL调用kernel。设备初始化包括OpenCL软件平台cl\_platform\_id、OpenCL硬件设备cl\_device\_id、OpenCL执行上下文cl\_context、OpenCL指令队列cl\_command\_queue；参数指定包括设备端输入输出参数，跟kernel参数列表一一对应，实际上就是将kernel函数的形参和实参进行绑定，另外还需指定线程结构work-item/work-group；通过函数cl\_quene启动kernel，在设备端执行相应的并行计算任务并反馈处理结果。

OpenCL指令与SSE指令对比发现，根据编译原理的语言分层方法，OpenCL指令比后者更加“高级”，OpenCL指令更加接近人类便于识别的自然语言，而SSE倾向于汇编语言。一些常用指令分别用自然语言、OpenCL、SSE、汇编对比描述如下：

表2-17 OpenCL操作与SSE及汇编的对应关系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 自然语言 | OpenCL | SSE | 汇编 |
| 1 | 四个float小数同时相加 | float4 a,b,c;  c = a + b; | \_\_mm128 a,b,c;  c=\_mm\_add\_ps(a, b) | addps xmm1,xmm0 |
| 2 | 四个float小数同时相减 | float4 a,b,c;  c = a - b; | \_\_mm128 a,b,c;  c=\_mm\_sub\_ps(a, b) | subps xmm1,xmm0 |
| 3 | 四个float小数同时相乘 | float4 a,b,c;  c = a \* b; | \_\_mm128 a,b,c;  c=\_mm\_mul\_ps(a, b) | mulps xmm1,xmm0 |
| 4 | 四个float小数同时读入 | float s[4];  float4 a = (float4)  (s[0],s[1],s[2], s[3]) | float s[4];  \_\_mm128 a =  \_mm\_load\_ps( p ); | movaps xmm0,xmmword ptr [eax] |

继续以加权图像变换算法为例，本节以CPU端OpenCL技术实现算法的并行化。跟之前用到的OpenMP和SSE并行技术不同，之前技术不用另外安装第三方库，编译环境内置支持，只要开启编译选项并包含头文件，在代码中就可以调用OpenMP和SSE的功能API。而OpenCL必须安装设备驱动和软件开发包SDK，本文选用Intel(R) SDK for OpenCL Applications 2013，SDK安装过程自动添加环境变量INTELOCLSDKROOT，指向OpenCL SDK的安装路径。OpenCL安装好以后，在工程属性中添加OpenCL的include头文件路径" $(INTELOCLSDKROOT)\include"和lib文件路径"$(INTELOCLSDKROOT)\lib\x86"并包含库文件"OpenCL.lib"，接下来在代码中就可以调用OpenCL接口。

首先初始化OpenCL运行环境。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"Intel(R) OpenCL"。

接下来先后调用以下几个关键的OpenCL标准API，完成OpenCL运行环境的配置：

表2-18几个关键的OpenCL标准API

|  |  |
| --- | --- |
| OpenCL API | 功能描述 |
| clCreateContextFromType | 创建OpenCL上下文环境，选CPU或GPU |
| clCreateCommandQueue | 创建任务队列 |
| clCreateProgramWithSource | 设置cl文件 |
| clCreateKernel | 选定cl文件内某个kernel函数 |

OpenCL运行环境的配置代码样例：

{

cl\_context\_propertiescontext\_set[3] = {

CL\_CONTEXT\_PLATFORM, (cl\_context\_properties) \_platform\_id, NULL };

// create the OpenCL context on a CPU/PG

cl\_contextg\_context=clCreateContextFromType (CL\_DEVICE\_TYPE\_CPU);

cl\_command\_queueg\_cmd\_queue=clCreateCommandQueue(g\_context,…);

cl\_programg\_program = clCreateProgramWithSource(g\_context, 1, , …);

clBuildProgram(g\_program, …);

// kernel\_name是kernel函数名

cl\_kernelg\_kernel = clCreateKernel(g\_program, kernel\_name, NULL);

}

cl文件是OpenCL语言特有文件，存放计划在OpenCL设备端执行的功能指令，功能指令封装在kernel函数，一个cl文件可以同时包含多个kernel函数作为多套执行方案，通过调用clCreateKernel生成多个cl\_kernel资源句柄，但同一时刻只有一套处于激活可用的状态，具体根据参数设置、线程结构配置、命令队列执行三个环节使用了那个kernel的cl\_kernel资源句柄。

### OpenCL的异构特性和可移植性

OpenCL技术如前文所述，它是一种开放的语言规范标准，主要面向通用并行计算，在CPU和GPU上都可以运行，而不同的CPU厂商和GPU厂商提供各自的驱动程序实现OpenCL的底层功能。GPU端OpenCL技术的典型实现是AMD的APP(Accelerated Parallel Processing)，以及NVIDIA的CUDA。因为OpenCL原生属性具有极高的可移植性，可以通过一致的接口在不同类型的设备上运行并行程序，所以本文将OpenCL算法从CPU移植到GPU，进一步发挥多核并行加速算法性能。移植程序时，只要对OpenCL运行环境配置做一处修改，在选用OpenCL平台cl\_platform时，把平台过滤条件从代表CPU的标示符改为代表GPU的标示符，例如把"Intel(R) OpenCL"改为"NVIDIA CUDA"。选用平台改变后，所有原先发往CPU的OpenCL命令都将转移方向，发布给GPU，最后也从GPU获取处理结果。对于编程人员这些过程都是透明的，编程人员只管用同样的方法调用抽象的OpenCL设备，OpenCL命令具体在CPU或是GPU上怎样执行，程序开发人员不需要深究。如果深究的话，可以发现OpenCL在CPU端具体执行时，会调用前文所述的多线程并行技术以及多指令并行技术，如OpenMP/TBB/SSE/AVX等等；而OpenCL在GPU执行时，也将调用相应厂商提供的GPGPU技术，如GLSL/CUDA等等。按照编译原理的角度描述OpenCL语言，它是一种高级语言，对底层并行计算语言进行抽象封装；也好比面向对象的继承和多态，通过父对象指针一致的接口，调用子类对象功能相似但个性不同的功能，隐藏了细节的差异，避免程序开发人员关注不同的硬件细节并且为不同硬件特性编写多套代码。

WebCL全称Web Computing Language，它是OpenCL的网页版，语言结合OpenCL及Javascript。WebCL提供一套Javascript API接口，为OpenCL配备软件和硬件运行环境，自动选择适合的OpenCL驱动以及硬件平台，支持跨操作系统跨CPU-GPU硬件，进一步提供OpenCL技术的便携性和可移植性。WebCL帮助网页应用程序使用GPU和多核CPU的并行计算能力，用于加速图像、视频、物理仿真模拟等Web应用。WebCL拥有挖掘HTML5浏览器能力的潜能，将来可以广泛拥有在网页中对计算密集型程序和可视化模拟程序进行物理加速。目前WebCL标准暂时还没有完全成型，只是在2012年初推出一个草案并于2013年10月做了版本的更新，有少数几家科研机构对WebCL进行了实现，包括Nokia和Samsung。Nokia于2011年5月开始在Firefox火狐浏览器实现WebCL并开展研究、Samsung三星于同年7月WebKit浏览器上实现WebCL。Nokia实现的WebCL需要作为外部插件的方式安装在标准的Firefox火狐浏览器上，而Samsung在开源浏览器引擎WebKit进行定制开发实现私有的WebCL分支版本。虽然WebCL官方组织还没有推出正式版的WebCL，但不影响研发人员对基于WebCL的应用进行前瞻性的研究，因为毕竟已经有草案提供了比较完整的WebCL标准接口，而且Samsung和Nokia搭建了临时的WebCL运行环境。一旦官方的WebCL标准以及原生实现WebCL的浏览器正式发布，研究人员可以快速根据WebCL标准接口对既有应用做升级，并且直接运行在免安插件的浏览器里面。通过WebCL将三维引擎部分存在性能瓶颈的算法进行加速，并通过网页直观发布加速前后的对比效果，计划作为本文研究工作的进一步拓展和延伸。

### OpenCL版本的加权图像变换算法实现

加权图像变换的OpenCL kernel函数实现如下：

\_\_kernel void

transformVectorByMatrix( const \_\_global float4 \*pInput, \_\_global float4 \*pOutput, float rad, float scale, int sizeMax )

{

// 第一步，求取线程索引index…

// 第二步，读取静态顶点，待更新

float4 vIn = pInput[index];

//第三步，求权重。距离1/4次方的倒数 weight = (x^2+y^2)^(-1/4)

float2 vInx = (float2)(vIn.x, vIn.z ), vIny = (float2)(vIn.y, vIn.w );

float2 xySqure = vInx \* vInx + vIny \* vIny + (float2)(4.0f);

float2 weight = pow( xySqure, (float2)(-0.25f) );

// 第四步，加权旋转角度

float2 radWeight = weight \* (float2)(rad);

float2 sinValue, cosValue;

sinValue = sincos( radWeight, &cosValue );

// 第五步，计算旋转变换矩阵

float4 mat[2];

… // 根据公式2-4从旋转角度构造旋转矩阵

// 第六步，旋转并且缩放

pOutput[index] = ( mat[0] \* (float4)(vIn.xx, vIn.zz) +

mat[1] \* (float4)(vIn.yy,vIn.ww) ) \* (float4)(scale);

}

调用kernel与普通的C语言立即调用方式不同，OpenCL采用异步调用，先写好调用的指令存入队列，最后通过调用clFinish批量一次性执行。先设置好参数，并传输数据，执行kernel，最后返回结果。例如对于以上加权图像变换的kernel，调用过程如下：

{

// 第一步，传入待处理的输入数据

clEnqueueWriteBuffer(g\_cmd\_queue, m\_pfInputBuffer, CL\_TRUE, 0, nBufferSize, m\_pIn, 0, NULL, NULL);

//第二步，配置kernel实参，6个实参跟kerne匹配，调用clSetKernelArg…

//第三步，在OpenCL设备端执行kernel指令

clEnqueueNDRangeKernel(g\_cmd\_queue,g\_kernel, 2, NULL, m\_argOCL.globalWorkSize, NULL, 0, NULL, NULL);

// 等待执行结束，最后获取结果，clWaitForEvents/clEnqueueReadBuffer

clFinish(g\_cmd\_queue );

}

### OpenCL算法性能分析

按照以上方法，对变更后的“加权图像变换算法”进行OpenCL并行优化。鉴于OpenCL计算必须在特定的内存空间(cl\_mem)执行，所以在普通内存和OpenCL内存之间存在读写拷贝。kernel参数需要转化为OpenCL内存才可以供设备端使用，而OpenCL处理结果需要转化为普通内存才可以供主机端使用。根据是否需要输入参数以及输出结果，OpenCL数据传输类型分三种情况，即参数输入结果输出、只有结果输出、无传输，测得性能如下表。完整实现加权图像变换功能，OpenCL的输入坐标保持不变，每次变化的是角度参数，然后通过kernel计算权重获取各自的旋转矩阵并且变换坐标，得到结果从OpenCL输出到普通内存，然后绘制图像。本文实验属于图中的“仅输出”情况。

表2-19加权图像变换算法 OpenCL版本耗时分布

|  |  |  |  |
| --- | --- | --- | --- |
| 像素 | 耗时ms  串行优化 | 耗时ms  输出 | 加速比 |
| 1024 | 92 | 12 | 7.7 |
| 2048 | 364 | 39 | 9.3 |
| 4096 | 1460 | 147 | 9.9 |

从以上图表发现，OpenCL性能随着顶点数目变大而提升，性能至少与之前的“线程和指令双层并行”优化方案持平，像素点为4K\*4K时加速比相对上一方案提升20%。如果不考虑内存拷贝，OpenCL加速效果更加明显，相对上一方案加速比提升30%~100%。可见OpenCL性能具有明显的优势，因为OpenCL实现了更高效的多线程调度以及多指令并行。对于多线程，OpenCL会智能地从OpenMP、TBB(Thread Building Block)或其它线程方案中选择最优者；对于多指令，OpenCL也会智能地从SSE、AVX指令方案选择最优者。而且OpenCL会自动判别系统硬件是否支持AVX或其它并行技术，编程人员不用手动判别。

## 基于GLSL的GPGPU通用并行计算

### 顶点着色器语言Shading Language简介

上文通过CPU多核多线程并行编程，程序性能与核心数有线性增长的关系，如果核心数进一步增加，可以预料程序的性能也将进一步提升。但是CPU核心数目的增长受到芯片面积和功耗等硬件限制，当前PC机核心数目最多是8核。焦点从CPU转到GPU，当CPU核数于2007年初首次达到4时[Intel Core 2 Quad Q6600]，那时的GPU核心数目多达128[GeForce 8800 GTX]；之后CPU核心数目于2011年底最多增加一倍到8[AMD FX-8150]，而GPU核心数目呈几何级数增长已达到1000多核[AMD Radeon HD 6970]。GPU不仅在核心数方面远远超越CPU，在带宽方面也有很大的提升。以下两幅图对比展示GPU和CPU的计算能力和带宽。

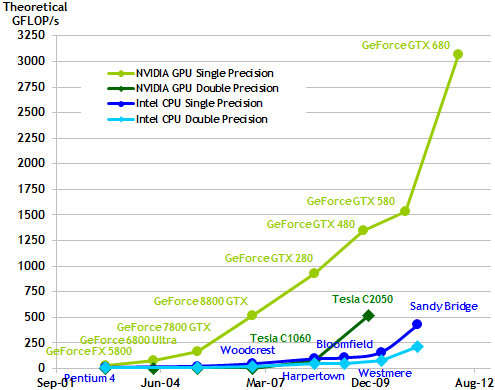


图2-4 GPU vs. CPU 浮点计算能力，前者约为后者的7倍

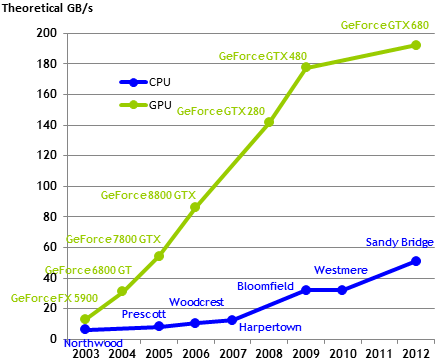


图2-5 GPU vs. CPU 存储器带宽，前者约为后者的5倍

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)。Cg语言是前两种着色语言的共性抽象接口，Cg底层更加具体情况调用可用的GLSL或HLSL，所以Cg语言编写的GPGPU程序拥有较高的可移植性，在程序三维图像库基于OpenGL和Direct3D的不同场合统统适用。

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。GLSL最先作为OpenGL 1.4的扩展出现，后来成为OpenGL 2.0 的核心组成部分。从2004年OpenGL 2.0发布以后，所有支持OpenGL2.0的软硬件平台，内置对GLSL的支持，程序开发人员无需引用外部库或者对编程环境作额外的配置。GLSL的语言规范包含在ARB\_vertex\_shader和 ARB\_fragment\_shader两项标准里面，除了语言从汇编升级为类C语言之外，其它很多规范都继承自上文提及的旧有的两个扩展ARB\_vertex\_program和ARB\_fragment\_program。GLSL的核心理念包含在对Vertex Shader和Fragment Shader(同Pixel Shader)的设计上面，即ARB\_vertex\_shader和ARB\_fragment\_shader这两个新的扩展。

使用Vertex Shader可以以更通用的角度描述发生在每个顶点上的操作，一个Vertex Shader程序有一个字符串所代表的源代码组成，代码内编写了在每个顶点上执行的操作细节。Vertex Shader所使用的语言基于类C语言，具体规范参照OpenGL着色语言规范[57]。类似地，使用Fragment Shader可以以更通用的角度描述发生在每个像素上的操作，最终像素信息来自点、线、多边形、矩形像素区域和位图的采样和插值。如果没有定义或没有激活Vertex Shader，渲染引擎将自动将调用原先缺省固定功能的顶点和像素处理过程。

### 着色器语言编程规范

两个Shader在渲染管线Pipeline中所处位置的框图和可视化效果图如下：

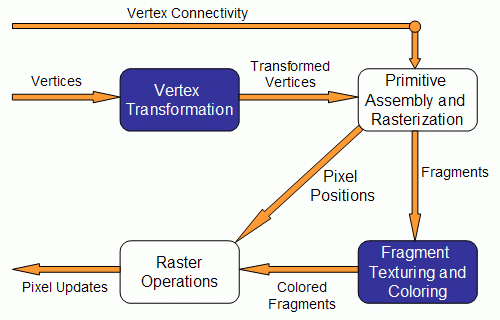


图2-6 Shader在渲染管线Pipeline中所处位置的框图

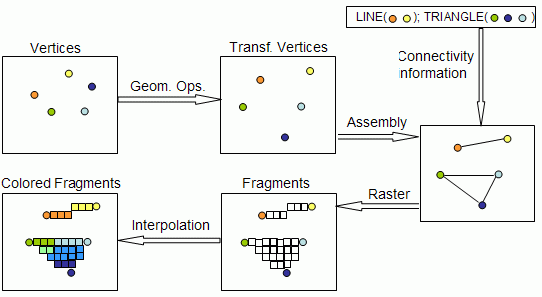


图2-7 Shader在渲染管线Pipeline中所处位置的可视化效果图

使用Shader编程涉及以下5个环节，编写、导入、编译、链接、执行，用到GLSL提供的一系列API。其中后4个环节对应的API是：glCreateShader/glShaderSource、glCompileShader、glCreateProgram/glAttachShader/glLinkProgram、glUseProgram。另外，一般在处理过程需要用到动态变化的参数，需要绑定参数、更新参数，对应API是glGetUniformLocation、glUniformXXX。Shader用完以后，需要调用glDetachShader/glDeleteProgram/glDeleteShader等API先后释放Program和Shader资源。Shader执行过程示意图如下：

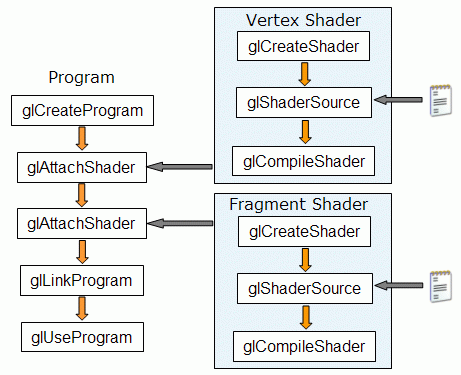


图2-8 Shader执行过程示意图

首先为Vertex Shader和Fragment Shader编写Program指令，指令是两个名字都是main的无参无返回值的功能函数，分别代表Vertex顶点和Pixel像素两个处理过程的两个字符串。等效于原先固定流水线的默认指令如下：

// VertexShader

void main()

{

gl\_Position= ftransform();

// ftransform等效于gl\_ModelViewProjectionMatrix \* gl\_Vertex;

gl\_FrontColor = gl\_Color;

}

// Pixel Shader or Fragment Shader

void main()

{

gl\_FragColor= gl\_Color ;

}

默认Vertex Shader和默认Pixel Shader完成固定流水线的功能，它们完成的功能等效于原先不可编程时代的图形设备，其中用到的变量和功能函数都是内置的，没有用户自定义的变量或者功能函数。Vertex Shader进行顶点坐标变换，将世界坐标系转化为窗口坐标系，用模型视图矩阵和投影矩阵ModelViewProjectionMatrix 对三维坐标进行坐标变换；而Pixel Shader只是接受顶点指定的颜色，并自动完成插值。其中用到内置的变量和函数，变量分为属性attribute和可变varying变量，函数ftransform执行矩阵变换将世界坐标转为屏幕坐标。attribute变量包括Vertex Shader里面的gl\_Color和gl\_Vertex，即原始的点颜色和点坐标。varying变量包括Vertex Shader里面的gl\_FrontColor和Pixel Shader里面的gl\_Color。gl\_Color作为attribute和varying类型出现两次，两次的意义是不一样的，作为attribute时gl\_Color表示传入Vertex Shader的每个顶点的最初颜色，而作为varying时gl\_Color表示从Vertex Shader传入Pixel Shader的gl\_FrontColor经过插值后的每个像素的最终颜色。

显然默认的Shader不能满足多样化的功能，Shader通过支持丰富的变量variable和算法操作，使得Shader得以完成用户定制的功能。变量分attribute、varying、uniform三类，其中：除内置变量顶点坐标gl\_Vertex、法线坐标gl\_Nomal、颜色gl\_Color、纹理坐标gl\_MultiTexCoord[X]以外，attribute用于给每个顶点传递额外的新的用户自定义参数变量；除内置变量以外，varying用于从Vertex Shader传递新的自定义参数到Pixel Shader；uniform用于给整个Vertex Shader的每个顶点或者Pixel Shader的每个像素传入统一的参数，并且这样的参数只能在Shader外修改。各种变量对比关系参考以下表格：

表2-20 Shader不同类型的变量之间的比较

|  |  |  |
| --- | --- | --- |
| 类型 | 功能区别 | 内置变量示例 |
| attribute | 从Shader外部向Vertex Shader传递参数；  每个顶点的值不同； | gl\_Vertex、gl\_Nomal、  gl\_Color、gl\_MultiTexCoord[X] |
| varying | 从Vertex Shader向Pixel Shader传递参数； | gl\_FrontColor、gl\_BackColor、  gl\_TexCoord[X] |
| uniform | 从Shader外部向Vertex/Pixel Shader传递参数；  每个顶点或每个像素的值相同； | gl\_ModelViewProjectionMatrix  gl\_ModelViewMatrix等 |

WebGL是OpenGL的网页版，WebGL语言结合OpenGL ES及Javascript。OpenGL ES本身采用可编程顶点着色程序Vertex Shading Program和片段着色程序Pixel Shading Program，在网页语言中支持OpenGL Shading Language(GLSL)并行技术。对客户端程序进行网页化，在带来便利的同时，不可避免会潜在地带来性能的损失，本文对性能进行测试，定量对比非网页化GLSL以及WebGL网页化的性能。WebGL同样是一套标准的开放接口，一开始只是少量浏览器进行实现，原型最早于2006年在浏览器Mozilla里面实现，接着在2007年底浏览器Opera实现。直到2009年初，制定包括OpenGL等大量开放标准的非盈利组织Khronos Group接管了WebGL，Apple和Google也参与了WebGL的标准化。Khronos Group与2011年3月推出WebGL 1.0正式版，随后几乎所有主流浏览器都实现不需外挂插件的内置WebGL。

## 基于CUDA的GPGPU通用并行计算

### 统一计算设备架构CUDA简介

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面向程序开发人员的开发包SDK最早于2007年2月首次发布，主要面向Windows和Linux系列操作系统，从2008年发布的2.0版本开始支持苹果Mac OS X系列系统。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。以2007年作为时间界限，2006年(含)之前的NVIDIA显卡几乎都不支持CUDA，2008年(含当年)以后的NVIDIA显卡都支持CUDA，2007年部分显卡支持。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。

本节开始采用CUDA改写骨骼动画算法，力求通过CUDA使得算法在GPU上取得性能上的进一步提升。在改写算法之前，先配备CUDA运行的软硬件环境。上文提到在开发和运行OpenCL程序时，只要安装支持OpenCL的驱动就可以，GPU端的OpenCL程序可以和CPU端共用一套SDK(包括OpenCL.h和OpenCL.lib)，而不需要特别安装针对GPU的软件开发包SDK。开发CUDA程序跟OpenCL不同，必须安装相应的软件开发包CUDA SDK，本文实验环境安装了CUDA 5.0，内含CUDA Toolkit、CUDA Samples、NVIDIA显卡驱动3部分。接下来配置CUDA工程属性，在工程属性中添加CUDA的include头文件路径”$(CUDA\_PATH)\include”和lib文件路径”$(CUDA\_PATH)\lib\$(Platform)”并包含库文件”cudart.lib”，接下来在代码中就可以调用CUDA接口。每个CUDA程序至少包含一个cu或cuh文件，设备端运行的kernel函数必需在cu文件里面进行调用，kernel函数声明必需放在cu或cuh文件。设备端的C/C++源码也可以写在cu文件里面，编译器自动分离cu源码和cpp源码并分别调用不同的编译器进行编译。本文新建一个cu文件，用于声明和调用CUDA的kernel函数。加入cu文件后，需要为cu自定义Build生成方式，设为”CUDA 5.0”，设完cu生成方式以后工程属性页将自动新增”CUDA C/C++”一栏，还需要将cu文件工程属性”项类型”设置为”CUDA C/C++”，确保cu文件由CUDA提供的编译器nvcc进行编译。

### CUDA编程模型的设计特色

CUDA编程模型支持多种语言，包括C, C++ and Fortran，本文采用CUDA C。跟以往的GPU编程模型相比，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。

* 核心函数Kernels

它是从C语言函数概念扩展而来的类C函数，它由每个线程调用并且同时执行，但每个线程处理根据线程ID索引的不同的输入输出参数。Kernel函数的声明和调用和C语言略有不同，声明时为了说明它的执行设备是GPU CUDA设备端而不是CPU主机端，特别在函数前面加上\_\_global\_\_进行标注，\_\_global\_\_表示函数调用方是主机端而执行方是设备端，与之并列的函数标注\_\_device\_\_表示函数调用方和执行方2者都是设备端，\_\_host\_\_表示都在主机端；调用时在函数名和参数左括号'('中间插入'<<<g, b>>>'结构，这一结构代表后文将进一步介绍的线程层次结构。代码样例如下：

// Kernel函数声明

\_\_global\_\_ void VecAdd(float\* A, float\* B, float\* C)

{

int i = threadIdx.x; // 获取线程ID

C[i] = A[i] + B[i];

}

int main()

{

...

// Kernel函数调用，在N个线程上并行执行

VecAdd<<<1, N>>>(A, B, C);

...

}

* 线程层次结构Thread Hierarchy

CUDA线程分两个层次，网格Grid和块Block，一个网格包含多个块，一个块内部嵌套包含多个线程。网格和块的内部结构都是多维的，网格2~3维，块都是3维，受硬件特性的制约各个维度的最大值有一定的限制，每个块包含的总线程数目也有一定限制，具体参照下文计算能力Compute Capability。一般将网格Grid设为一维，数目是GPU内MutiProcessor(MP)个数的倍数；一般将块Block设为两维，快内总线程数是32的整数倍，比如设置为(16, 16)。线程结构具体如何设计，原则上只要在硬件限制的范围之内都可以，但为了使CUDA性能达到最优，需要进一步更加硬件特性和程序特性选择合适的参数。线程层次结构示例代码和示意图如下：

// 线程网格Grid结构：(16 ) 包含16个线程块Block

int numBlocks = 16;

// 线程块Block结构：(16,16,1) 包含256个线程

dim3 threadsPerBlock(16, 16);

kernel<<<numBlocks, threadsPerBlock>>>(…);

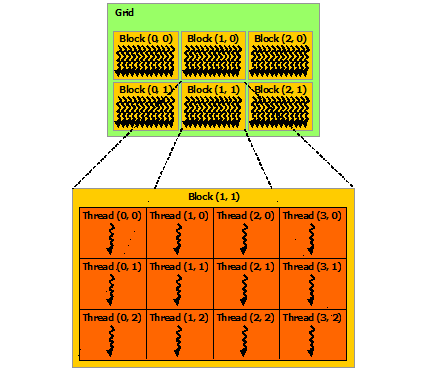


图2-9线程双层结构示意图，网格Grid和块Block

随着线程结构定义的不同，kernel内对线程的索引方式也随着改变。计算线程索引时用到CUDA内置的几个变量threadIdx、blockIdx、blockDim，分别表示当前线程在所属线程块内的坐标变量，当前线程所属线程块的坐标变量、线程块各维的长度常量。具体索引方式见下表：

表2-21 CUDA线程索引计算方法

|  |  |  |
| --- | --- | --- |
| 维数 | 各维索引 | 线性索引 |
| 1 | x = blockDim.x\*blockIdx.x+threadIdx.x | tID = threadIdx.x  bID = blockIdx.x  indexTotal = bID\*blockDim.x+tID |
| 2 | x = blockDim.x\*blockIdx.x+threadIdx.x  y = blockDim.y\*blockIdx.y +threadIdx.y | tID = threadIdx.x + threadIdx.y\*blockDim.x  bID = blockIdx.x+ blockIdx.y\*gridDim.x  indexTotal = tID + bID \* blockDim.x \* blockDim.y |
| 3 | x = blockDim.x\*blockIdx.x+threadIdx.x  y = blockDim.y\*blockIdx.y +threadIdx.y  z = blockDim.z\*blockIdx.z +threadIdx.z | tID = threadIdx.x + threadIdx.y\*blockDim.x + threadIdx.z\*blockDim.x\*blockDim.y  bID = blockIdx.x + blockIdx.y\*gridDim.x  + blockIdx.z\* gridDim.x\* gridDim.y  indexTotal = tID + bID \* blockDim.x \* blockDim.y \* blockDim.z |

* 存储空间层次结构Memory Hierarchy

CUDA存储空间分为以下几类，全局存储Global Memory、共享存储Sharaed Memory、常量存储Constant Memory、寄存器Register、局部存储Local Memory、纹理存储Texture Memory。它们的存储位置以及读写权限如下图表：

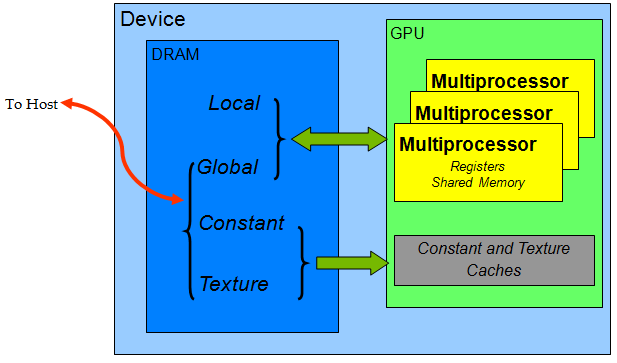


图2-10 CUDA各类存储类型的位置示意图

表2-22 CUDA各类存储类型的特性参数

| **Memory存储类型** | **Location**  **位置** | **Cached是否缓存** | **Access读写权限** | **Scope**  **共享范围** | **Lifetime**  **变量有效期** |
| --- | --- | --- | --- | --- | --- |
| Register  寄存器 | On chip  芯片内部 | n/a  不需要 | R/W  读写 | 单个线程 | 线程 |
| Local  局部 | Off chip  芯片外部 | \* | R/W  读写 | 单个线程 | 线程 |
| Shared  共享 | On chip  芯片内部 | n/a  不需要 | R/W  读写 | 线程块 | 线程块 |
| Global  全局 | Off chip  芯片外部 | \* | R/W  读写 | 所有线程  和主机 | 主机端决定 |
| Constant  常量 | Off chip  芯片外部 | Yes  有缓存 | R  只读 | 所有线程  和主机 | 主机端决定 |
| Texture  纹理 | Off chip  芯片外部 | Yes  有缓存 | R  只读 | 所有线程  和主机 | 主机端决定 |
| \* Cached only on devices of compute capability 2.x.  \* 表示一部分有、一部分没有。计算能力大于等于2.0有缓存，低于2.0没有 | | | | | |

* 异构编程Heterogeneous Programming

CUDA编程模型把硬件系统划分为主机端和设备端，调用CUDA程序的一端称为主机端通常指CPU，执行CUDA程序的一端称为设备端通常指GPU。主机端和设备端不仅有各自独立的处理器，而且有各自独立的存储空间，CPU处理的数据存在内存，而CUDA通过GPU处理的数据存储在显存。CPU和GPU是两种完全异构的硬件平台，通过CUDA这个桥梁，使得它们搭配在一起协同完成计算任务。往往CPU把高密度的运算任务打包交由擅长处理简单而大量重复任务的GPU，CPU自身因为拥有较强的逻辑分析能力擅长为GPU预处理上游数据和分析从GPU得到的下游结果。CPU搭配GPU这种异构编程结构，可以很好地发挥两种处理器各自的特长，使系统硬件利用率达到最高，发挥系统最好的性能。CPU与GPU异构协调工作示意图如下：

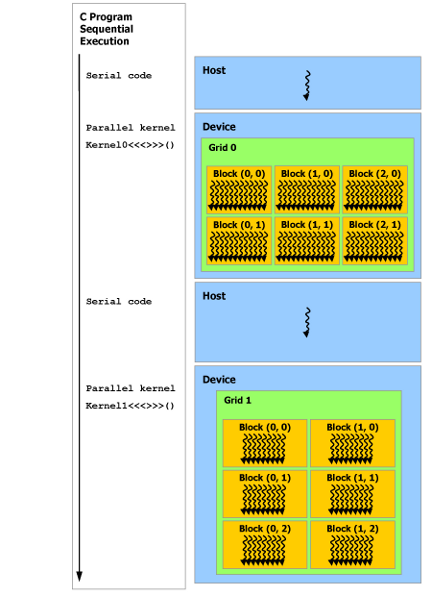


图2-11 CUDA异构编程执行过程示意图

* 计算能力Compute Capability

决定CUDA硬件功能的有无以及性能的高低，比如最早一款CUDA显卡GeForce 8800 GTX的计算能力是1.0，显卡GeForce GTX Titan的计算能力是3.5。通过查阅特定显卡所归属的计算能力级别，可以评估它所能实现的CUDA功能及性能，在程序优化过程也需要根据计算能力调整参数。

表2-23 CUDA各级计算能力的规格参数表

|  | **Compute Capability 计算能力** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Technical Specifications**  **技术规格** | **1.0** | **1.1** | **1.2** | **1.3** | **2.x** | **3.0** | **3.5** |
| Maximum dimensionality of grid of thread blocks线程网格最大维数 | 2 | | | | 3 | | |
| Maximum x-dimension of a grid of thread blocks网格x维最大块数 | 65535 | | | | | 231-1 | |
| Maximum y- or z-dimension of a grid of thread blocks 每个线程网格y/z维最大线程块数 | 65535 | | | | | | |
| Maximum dimensionality of thread block每个线程块最大维数 | 3 | | | | | | |
| Maximum x- or y-dimension of a block每个线程块x/y维最大线程数 | 512 | | | | 1024 | | |
| Maximum z-dimension of a block 每个线程块z维最大线程数 | 64 | | | | | | |
| Maximum number of threads per block每个线程块最大线程数 | 512 | | | | 1024 | | |
| Warp size | 32 | | | | | | |
| Maximum number of resident blocks per multiprocessor 每组SP最多包含常驻并发线程块的数目 | 8 | | | | | 16 | |
| Maximum number of resident warps per multiprocessor 每组SP最多包含常驻并发warp的数目 | 24 | | 32 | | 48 | 64 | |
| Maximum number of resident threads per multiprocessor 每组SP最多包含常驻并发线程的数目 | 768 | | 1024 | | 1536 | 2048 | |
| Number of 32-bit registers per multiprocessor 每组SP包含32位寄存器的数目 | 8 K | | 16 K | | 32 K | 64 K | |
| Maximum number of 32-bit registers per thread 每个线程最多可用32位寄存器的数目 | 128 | | | | 63 | | 255 |
| Maximum amount of shared memory per multiprocessor 每组SP包含共享存储器的容量 | 16 KB | | | | 48 KB | | |
| Number of shared memory banks  共享存储器产的分块数目 | 16 | | | | 32 | | |
| Amount of local memory per thread  每个线程局部存储器的容量 | 16 KB | | | | 512 KB | | |
| Constant memory size 常量存储器的容量 | 64 KB | | | | | | |
| Cache working set per multiprocessor for constant memory SP常量存储器的缓存容量 | 8 KB | | | | | | |
| Cache working set per multiprocessor for texture memory 每组SP纹理存储器的缓存容量 | Device dependent, between 6 KB and 8 KB | | | | | | |
| Maximum number of instructions per kernel -kernel指令数目的最大值 | 2 million | | | | 512 million | | |

表2-24 CUDA各级计算能力所支持的特性

|  | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Feature Support 支持的特性** | **Compute Capability 计算能力** | | | | | |
| **(没有罗列的特性表示都支持)** | **1.0** | **1.1** | **1.2** | **1.3** | **2.x 3.0** | **3.5** |
| Atomic functions operating on 32-bit integer values in global memory 全局显存32位整数原子操作 | No | Yes | | | | |
| atomicExch() operating on 32-bit floating point values in global memory 全局显存32位小数读后写 |
| Atomic functions operating on 32-bit integer values in shared memory 共享显存32位整数原子操作 | No | | Yes | | | |
| atomicExch() operating on 32-bit floating point values in shared memory 共享显存32位小数读后写 |
| Atomic functions operating on 64-bit integer values in global memory 全局显存64位整数原子操作 |
| Warp vote functions warp线程束表决操作 |
| Double-precision floating-point numbers 是否支持双精度浮点 | No | | | Yes | | |
| Atomic functions operating on 64-bit integer values in shared memory 共享显存64位整数原子操作 | No | | | | Yes | |
| Atomic addition operating on 32-bit floating point values in global and shared memory  共享显存64位小数读后写 |
| \_\_ballot() 逐个线程投票表决功能 |
| \_\_threadfence\_system() 系统级别的线程栅栏 |
| \_\_syncthreads\_count(),\_\_syncthreads\_and(), \_\_syncthreads\_or()  线程同步并检验线程是否符合某条件 |
| Surface functions 数组面操作 |
| 3D grid of thread blocks 3维线程网格 |

## 本章小结

本章首先确定并行计算性能评价模型，突出并行计算相对原先串行计算在编程方式和运行性能两方面的联系与区别，并行计算建立在串行计算的基础上继而加快运行速度从而节约执行时间。接着介绍了面向通用科学计算领域的多种前沿并行计算技术，包括面向CPU和GPU两种并行计算设备。其中，面向CPU的并行计算技术有传统的SSE技术和新兴的OpenCL技术，面向GPU的并行计算技术历经Shading Language、CUDA、OpenCL这三代GPGPU技术，OpenCL因其同时适用于CPU和GPU从而成为它们之间很好的纽带。为了更好地介绍各自并行计算技术与串行计算的关系以及它们之间的对比，本文选取了一个在遥感及其它领域常见的图像复原算法——图像旋转算法，在算法流程和功能细节前后一致并且层层递进的前提下，设计了算法的串行实现和各种版本的并行实现，从通用计算的角度在定性和定量上直观表现这些并行计算技术的特点和关联。分别面向CPU和GPU的并行计算技术演变过程如下图所示。

图2-12面向CPU的并行计算技术演变过程

图2-13面向CPU的并行技术关键特性发展趋势

图2-14面向GPU的并行计算技术演变过程

图2-15面向GPU的并行计算技术关键特性发展趋势

# 骨骼动画渲染算法在CPU上的并行计算

## 骨骼动画矩阵调色板算法

### 为骨骼动画渲染算法建模

骨骼动画是一种计算机三维角色动画，动画的产生由不可见的骨骼的运动带动角色模型的整体运动，骨骼的运动由一系列关键帧驱动，每一个关键帧内包含所有骨骼点相对静止状态的坐标变换矩阵信息。骨骼动画的优点是数据量小，缺点是运算量大，因为每一个时刻的动画都需要实时计算，运算时需要遍历动画模型所有皮肤点，并用每个皮肤点绑定的所有骨骼的变换矩阵对该点执行坐标变换，将变换结果按照一定的权重进行累加，最终得到每个皮肤点的新坐标。成人有206块骨骼，人物角色骨骼动画往往根据不同场合仅仅模拟其中的部分骨骼，如下图模拟16块关键的骨骼。

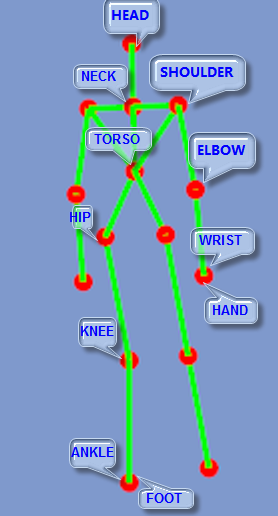
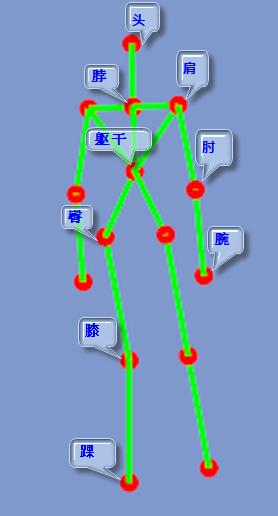


图3-1 骨骼与关节示意图

本文采用矩阵调色板算法实现骨骼动画的骨骼与皮肤联动效果。骨骼动画复杂度跟模拟皮肤的蒙皮模型的精度有关，同时跟蒙皮与骨骼的绑定关系有关。绑定关系指单个顶点关联的骨骼数目，除了位于骨骼中间的皮肤运动只受所在单个骨骼的影响以外，其它位于骨骼连接的关节处的皮肤点，往往受到多块骨骼的影响。关节处有多少块骨骼相连，附近的皮肤点就受相应块数骨骼的影响，同一关节处的骨骼数有3种可能，分别是二、三和四，比如膝盖关节受大腿小腿2处骨骼影响，臀部受左腿右腿腰部3处骨骼的影响，颈部受左手右手脖子背部4处骨骼的影响。加上远离关节位于骨骼中间的皮肤点，皮肤点与骨骼的绑定关系有4中情况。绑定关系根据影响大小有一定的权重，皮肤点里骨骼中心点越近所受影响越大，即权重越高。在制作角色动画模型的过程，已经设置好了皮肤和骨骼的绑定关系以及相应的权重值。

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。第一个环节一次性执行完成，不参与渲染循环，后三个环节可以进一步抽象为对两种数据结构的三种操作，其中数据结构指关节矩阵和顶点坐标，操作包含初始化矩阵和坐标、更新矩阵和坐标、输出坐标。以下示意图分别展示：整个场景的全局渲染流程、骨骼动画的局部渲染流程、骨骼动画和其它模块数据量相同时占整个场景的时间比重。

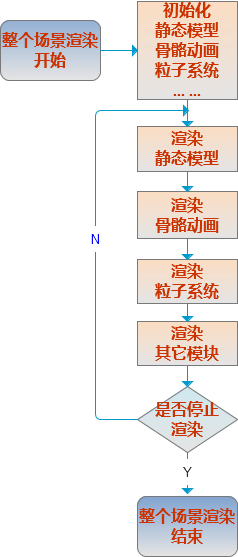
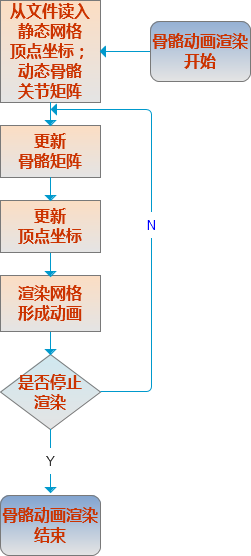
 

图3-2流程图——整个场景渲染以及骨骼动画渲染

图3-3 骨骼动画占整个场景的时间比重

本文在Visual Studio 2010通过Visual C++实现骨骼动画的动作计算并通过OpenGL进行三维渲染。为了支持不同复杂度的动画，本文算法使以下变量参数化：三角面片数目T，骨骼数目M，顶点绑定骨骼的数目B。对算法影响最大的两个参数是三角面片数目T和顶点绑定骨骼的数目B，因骨骼数目有限而且与三角面片数目的比值很小，一般不影响算法，所以忽略骨骼数目这个参数。本文重点关注基于矩阵调色板算法的骨骼动画顶点更新算法，骨骼动画顶点更新算法如下：

1）动画模型具有T个三角面片，每个三角面片以3个索引值代表，索引值指向某个顶点；

2）获取1个三角面片；

3）获取1个顶点索引，通过该索引获取顶点信息，包括1个坐标(x,y,z)和B个骨骼索引及对应的权重索引；

4）对每个顶点调用矩阵调色板算法；

5）返回步骤3，重复步骤3~4，直到遍历完一个面片的3个顶点；

6）返回步骤2，重复步骤2~5，直到遍历完所有面片。

矩阵调色板算法描述如下：

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 = 式（3-1）

3）为每个顶点进行坐标变换，(x’, y’, z’, 1) = (x, y, z, 1) \* m。

三维变换变换用于对三维坐标(x,y,z)进行平移T、缩放S、旋转R三种操作，这三种操作可以一致地用以下矩阵表征：

T(Δx,Δy,Δy)= , S(a,b,c)=,

R(θ,x)=, R(θ,y)=,

R(θ,z)=

如果仅仅是三种操作中的一种，变换矩阵最后1列是多余的，用4\*3矩阵就可以。但为了支持以上操作的叠加，变换矩阵增加1列变成4\*4的形式，多种变换叠加时，仅需对变换矩阵按一定顺序执行相乘操作即可。变换矩阵最终是统一为一个4\*4矩阵mat[4][4]，同时包含平移缩放旋转信息。统一矩阵示意图如下：

对应统一矩阵变换公式为：

\*=式（3-2）

以下是骨骼动画顶点更新算法及其子过程矩阵调色板算法的流程图：

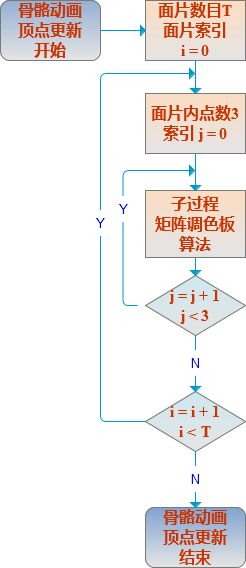
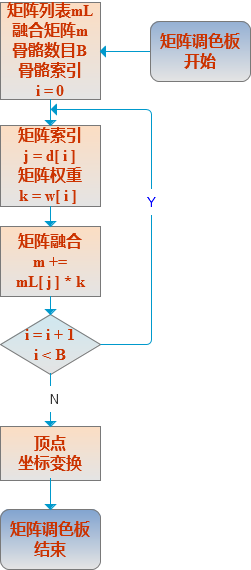
 

图3-4算法流程图——骨骼动画顶点更新算法，及其子过程矩阵调色板算法

### 串行算法性能分析和瓶颈定位

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。当前未优化骨骼动画渲染时间分布图表如下：

表3-1未优化骨骼动画渲染时间分布表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法版本 | 总共耗时ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 未优化  串行 | 5.73 | 0.18 | 3% | 4.69 | 82% | 0.75 | 13% | 14% | 12% |

对应时间分布图如下：

图3-5未优化骨骼动画的时间分布图

通过时间分布图可以看到，耗时最大的两部分，分别是CPU顶点计算和GPU渲染，比例分别是82%和13%，说明它们是整个算法效率的瓶颈。另外CPU计算骨骼所占比例极小，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。并行运算特别适用于以下类型的算法：结构方面，适于顺序语句不适于循环嵌套以及分支；索引方面，适于相邻索引是连续地址的情形，不适于随机地址索引；函数调用方面，适于少量或没有函数调用，不适于频繁或嵌套调用函数。根据以上原则，从矩阵变换模块算法可以看到，算法中存在多重嵌套循环和多处随机索引，其中多重循环嵌套是面片循环和顶点循环以及更深层的骨骼循环，随机索引包括顶点随机索引和骨骼随机索引。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。

本文将算法先进行串行改进，以期使得串行版本的性能达到最优，作为后续并行算法的参照标准。改进的地方包括：展开多重嵌套循环、对顶点索引进行重新排序。算法改进后，算法时间分布图表发生一些改变，新图表如下：

表3-2 串行优化后骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  未优化 | 5.73 | 0.18 | 3% | 4.69 | 82% | 0.75 | 13% | 14% | 12% |
| 串行  已优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |

图3-6串行优化后骨骼动画的时间对比图

图3-7串行优化后骨骼动画的加速比

图3-8串行优化后骨骼动画的时间分布图

经过串行改进后的算法，相对之前原始算法，通过新的耗时分布图表有以下几个发现：第一，整体性能小幅度提升15%；第二，CPU计算顶点占比重从之前82%将为79%，依然是潜在瓶颈点。串行优化之所以带来以上改变，因为展开多重嵌套循环，对顶点索引进行重新排序，使得CPU执行效率明显提升，主要原因是更加紧凑的内存访问，发挥了缓存cache预读和重读命中，引起数据访问效率提升，使得内存带宽得到充分地发挥。

通过上一节串行优化，算法瓶颈可以很容易锁定位置，即CPU计算顶点这一子模块。这一模块占全部时间的比例约为P=79%，剩余部分只占1-P=21%。根据阿姆达尔定律公式即式2-1：A=1/(1-P+P/S)，可以初步估算骨骼动画整体加速比，A = 1/(1-P+P/S) = 1/(1-0.79) = 1/0.21 = 4.8。结合1.4节的局部峰值加速比S=32，得到骨骼动画完整加速比的理论峰值A = 1/(1-P+P/S) = 1/(0.21 + 0.79/32) = 4.3。由1.2.1节可知，对应帧速改进幅度D = A – 1 = 330%。

## 基于SSE面向CPU的骨骼动画矩阵调色板算法

### OpenMP多线程并行

参照本文2.2.1节，执行多线程算法改进，开启OpenMP设置，在for循环调用omp parallel宏，关键改动如下：

//遍历每个顶点

#pragma omp parallel for

for(int y = 0; y < pMesh->m\_usNumTris\*3; y++)

对上一步串行优化后的算法开启OpenMP多线程并行，测得如下结果：

表3-3 线程并行优化后骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  已优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |
| 并行  线程 | 3.28 | 0.18 | 5% | 0.98 | 30% | 0.75 | 23% | 89% | 12% |

图3-9线程并行骨骼动画的时间对比图

图3-10线程并行骨骼动画的算法加速比

图3-11线程并行骨骼动画的时间分布图

局部顶点更新这一瓶颈子模块，加速比达到4.01，接近CPU核心数目4，属于理想情况。为什么超过核心数目4？因为每个核心上运行了2个线程，当一个线程因等待数据读写而处于空闲状态时，另一个线程接着处理已经预备好的数据，从而使CPU核心处于接近满负荷的状态。为什么没有达到8？因为执行多线程时，需要把任务分派给各个线程，需要耗费一定的时间开销，并且如果线程存在数据共享或其它通信，将额外耗费一定的同步时间，本文虽然不存在线程通信，但线程任务分派耗费一定的时间，使得加速比没有达到最大值8。

全局骨骼动画渲染，加速比达1.5。为什么全局加速比远小于局部加速比？因为加速部分所占比重不同，局部加速效果对整体加速的贡献就不一样，比重较大则贡献较大，反之贡献较小。对比理论峰值性能，本轮多线程取得加速比还不理想。下一步从单线程内部挖掘性能潜力。

为了对比测试OpenMP多线程效果，常常需要关闭OpenMP回到起初状态，此时不仅在代码中去除相关指令，而且需要把编译选项/openmp清除或在工程属性中设为false。否则编译选项会影响单线程算法的性能，当编译选项/openmp开启而实际上没有使用多线程指令时，单线程性能下降30%，导致多线程与单线程的对比结果失真。所以切记不使用多线程时，去除编译选项/openmp或在工程属性中设为false。

### 基于SSE的线程内指令并行

起初，顶点坐标存储结构设为(tx, ty, nx, ny, nz, x, y, z)，即每个顶点用8个浮点小数(float)表示，这是为了满足OpenGL三维图形渲染之交错数组这一模式的需要，其中(tx, ty)是用于纹理贴图的纹理坐标，(nx, ny, nz)是用于光照计算的法线方向，最后(x, y, z)才是坐标。顶点坐标存储在由8个浮点小数(float)组成结构体的后3个，也就是说顶点存储位置有5个浮点小数(float)的偏移量，无法满足SSE规定的16位对齐的要求。于是拆分交错数组，顶点计算模块只是坐标参与更新，即顶点坐标存储结构设为(x, y, z, 1)，每个顶点用4个浮点小数(float)表示。拆分交错数组改变顶点数据存储方式后，对前文未优化、串行优化、多线程并行优化3套算法性能的影响都不大，以关联2块骨骼的情况为例，拆分数据结构后，顶点计算局部算法以及动画渲染全局算法的性能改变幅度都在4%以内。改变数据结构没有引入额外的性能损失，只是从8个浮点小数(float)变为9个浮点小数(float)，多占了约10%左右的存储空间。

参照本文2.2.2节，执行多指令并行算法改进，调用SSE指令集，算法关键改动如下：

第一处移植，矩阵按权重累加改为：

// \_\_m128 pMatOne代表单个矩阵，pMatLast代表累加结果

for (int j=0;j<4;j++)

{

pMatLast[j] = \_mm\_add\_ps(pMatLast[j], \_mm\_mul\_ps( pMatOne[j], weight) );

}

第二处移植，坐标矩阵变换改为：

Step 1. 输入坐标pIn构造 \_\_m128 xxxx, y, z

\_\_m128 xxxx =\_mm\_shuffle\_ps( pIn, pIn, \_MM\_SHUFFLE(0,0,0,0) ); //y, z同

Step 2. 矩阵构造\_\_m128 m[4];

Step 3.计算输出坐标：

pOut=m[0]\*xxxx+m[1]\*yyyy+m[2]\*zzzz+ m[3]

因为SSE不支持多项式连续相加，所以拆成多步。其中，\_\_m128乘法调用\_mm\_mul\_ps，\_\_m128 加法调用\_mm\_add\_ps。

分步骤完成以上算法改进后，汇总最终SSE算法，测得性能如下：

表3-4指令并行优化后骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |
| 并行  线程 | 3.28 | 0.18 | 5% | 0.98 | 30% | 0.75 | 23% | 89% | 12% |
| 并行  指令 | 1.95 | 0.18 | 9% | 0.96 | 49% | 0.75 | 38% | 14% | 12% |

图3-12指令并行骨骼动画的时间对比图

图3-13指令并行骨骼动画的算法加速比

图3-14指令并行骨骼动画的时间分布图

局部加速比达到4.09，全局加速比2.55。局部SSE加速比已经非常接近理论峰值4，说明SSE优化可以圆满结束。

### 线程与指令双层并行

将SSE指令并行技术和OpenMP多线程技术叠加在一起，同时执行线程并行和指令并行优化。最终性能测时数据和结果如下：

表3-5线程与指令并行优化后骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |
| 并行  线程 | 3.28 | 0.18 | 5% | 0.98 | 30% | 0.75 | 23% | 89% | 12% |
| 并行  指令 | 1.95 | 0.18 | 9% | 0.96 | 49% | 0.75 | 38% | 14% | 12% |
| 并行  线程  与指令 | 3.04 | 0.18 | 6% | 0.52 | 17% | 0.75 | 25% | 89% | 12% |

图3-15线程与指令并行骨骼动画的时间对比图

图3-16线程与指令并行骨骼动画的算法加速比

图3-17线程与指令并行骨骼动画的时间分布图

分析以上图表得出以下结论：1）同时执行线程并行和指令并行时，局部加速比最高，达7.56，但是全局加速比只有1.66，并不是最高，全局加速比最高的是单独执行指令并行的版本；2）同时执行线程并行和指令并行时，虽然局部加速比最高，但是全局加速比不是最高，原因是多线程开启后影响内存读写性能，抵消局部性能的提升，而且局部所占比例才30%，局部性能的改进对全局性能的贡献不大。

## 基于OpenCL面向CPU的骨骼动画矩阵调色板算法

### 矩阵调色板算法从SSE移植到OpenCL

跟之前用到的OpenMP和SSE并行技术不同，之前技术不用另外安装第三方库，编译环境内置支持，只要开启编译选项并包含头文件，在代码中就可以调用OpenMP和SSE的功能API。而OpenCL必须安装设备驱动和软件开发包SDK，本文选用Intel(R) SDK for OpenCL Applications 2013。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"Intel(R) OpenCL"。

按照前文2.3小节，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，因为OpenCL在CPU端的底层实现细节实际上会转化为SSE或其它并行指令。

本文第一步先实现单骨骼情况，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：

Step 1. 获取OpenCL线程ID，当做顶点索引，线程ID获取公式如下：

id = get\_global\_id(0) + get\_global\_id(1) \* get\_global\_ size(0)

Step 2. 索引输入顶点pIn和矩阵pMatrix，其中每个矩阵由4个flaot4组成；

Step 3. float4数据结构封装，将pIn (x,y,z,1) 拆分为float4 (x,x,x,x) 、 (y,y,y,y) 、 (z,z,z,z)

Step 4. 顶点矩阵变换结果输出，float4多项式相乘累加，计算方法如下：

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

以上算法跟之前的SSE算法非常相似，对比算法不同之处主要有如下两点：

1）SSE的128字节数据类型\_\_m128，在OpenCL对应float4；SSE给\_\_m128的4个float赋同一个值采用\_mm\_shuffle\_ps指令，在OpenCL对应float4对象构造(float4)()；2）SSE因为不支持多项式相加所以需要通过宏模拟，而OpenCL直接支持多项式相乘累加，OpenCL算法非常简洁，拥有很高的可读性和可维护性。

kernel设计完以后，通过主机端将kernel编码的命令发送给OpenCL设备端，在设备端上执行相应功能。在此之前，需要为kernel指定实参，即原始顶点输入坐标值、矩阵变换后产生的新顶点输出值、矩阵表征的平移旋转缩放信息、以及顶点跟骨骼矩阵的索引关系，而且给OpenCL传入传出的内存需要统一转化为通过clCreateBuffer开辟的cl\_mem对象。内存转化以及实参指定过程如下：

void SetupKernel()

{

// 第一步，内存转化。普通内存转为OpenCL内存cl \_mem

cl \_mem memBufferIn = clCreateBuffer(\_context, INFlags,nBufferSize,

pVertexIn,NULL); // 以上转化输入，输出、矩阵值和索引同理

// 第二步，实参指定。包括输入输出顶点和矩阵

clSetKernelArg(\_kernel, 0, sizeof(cl\_mem), (void \*) &memBufferIn);

// 以上设置输入顶点，输出顶点、矩阵值、矩阵索引同理

}

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GlobalWorkSize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，比如CPU INTEL i7-870支持1024，各个维度乘积大于线程总需求量即可；而Local WorkSize采用系统默认设置。

最后，在主机端host调用clEnqueueNDRangeKernel将kernel编排的指令任务发布到OpenCL并行设备端。该指令调用属于异步调用，即调用后主机端不等待设备处理结果，接着执行下一句指令，如果需要等到设备端执行完成可以调用clWaitForEvents阻塞主机端使之进入空闲等待状态。OpenCL端执行完成后，结果存储在OpenCL的设备端内存cl \_mem，需要通过clEnqueueReadBuffer转化为主机端内存float\*。调用过程核心代码如下：

voidExecuteKernel()

{

// 第一步，主机端host将kernel编排的指令任务发布到OpenCL设备端

clEnqueueNDRangeKernel(\_cmd\_queue, \_kernel, 2, NULL, globalWorkSize, NULL, 0, NULL, &g\_perf\_event);

// 第二步，等到设备端执行完成

clWaitForEvents(1, &g\_perf\_event);

// 第三步，设备端结果，传输回主机端float\* pVertexOut

clEnqueueReadBuffer(\_cmd\_queue,memBufferOut,CL\_TRUE, 0,nBufferSize,pVertexOut, 0, NULL, NULL);

// 第四步，确保所有OpenCL指令执行完成

clFinish(\_cmd\_queue);

}

以上通过OpenCL在CPU端解决了每个顶点绑定1块骨骼的特例问题，接着改进算法，使得算法支持每个顶点绑定多块骨骼。对OpenCL版本的骨骼动画算法进行性能测试，分别测量：OpenCL版本算法时间分布、局部kernel核心耗时对比——局部OpenCL对比串行优化、全局完整动画更新渲染耗时对比——全局OpenCL vs 串行优化、OpenCL的kernel局部加速比和全局动画更新渲染加速比，结果如下：

表3-6在CPU上OpenCL版本骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法  版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |
| 并行  线程 | 3.28 | 0.18 | 5% | 0.98 | 30% | 0.75 | 23% | 89% | 12% |
| 并行  指令 | 1.95 | 0.18 | 9% | 0.96 | 49% | 0.75 | 38% | 14% | 12% |
| 并行  线程  与指令 | 3.04 | 0.18 | 6% | 0.52 | 17% | 0.75 | 25% | 89% | 12% |
| CPU OpenCL | 1.86 | 0.18 | 10% | 0.22 | 12% | 1 | 54% | 14% | 12% |

图3-18 CPU上OpenCL骨骼动画的时间对比图

图3-19 CPU上OpenCL骨骼动画的算法加速比

图3-20 CPU上OpenCL骨骼动画的时间分布图

从加速比图看到，OpenCL版本算法的性能与SSE结合OpenMP并行版本算法的性能相当，区别主要有：1）整体性能OpenCL下降，瓶颈局部加速比降幅30%，全局加速比降幅10%；2）随着骨骼数增多，OpenCL性能提升，最高性能是4个骨骼，局部加速比达14，全局加速比2.8；3）异步重叠，主要是因为OpenCL和OpenGL都是异步调用，所以它们分别执行时可以并行完成，导致时间重叠8%，从而节省算法整体运行时间，即实际运行总时间小于局部时间之和，局部耗时比率的总和大于100%。

从CPU上OpenCL骨骼动画的时间分布图，可以得到如下结论：经过OpenCL并行化以后的算法，顶点坐标矩阵变换所花费的计算时间大幅下降，已经不再是瓶颈。同时暴露新的潜在瓶颈即渲染时间，有待于进一步优化。

### OpenCL与OpenGL互操作

本文接下来采用OpenCL的特性，即OpenCL与OpenGL互操作，来优化渲染和数据传输。算法需要做以下几方面的修改：

第一处修改，配置支持CL-GL互操作性的环境：

首先，并不是所有OpenCL软件驱动和OpenGL之间都能互操作，经调研和实验本文发现以下规律：1）Intel的OpenCL驱动暂时只支持Intel CPU与Intel HD核芯GPU之间互操作，不支持运行在AMD或NVIDIA的GPU的OpenGL进行互操作；2）AMD的OpenCL驱动不仅支持AMD CPU和GPU之间进行OpenCL-OpenGL互操作，还支持运行在CPU上的OpenCL与运行在NVIDIA的GPU的OpenGL进行互操作；3）NVIDIA的OpenCL也只支持在自家GPU上进行OpenCL-OpenGL互操作。本文的硬件是Intel的CPU和NVIDIA的GPU，需要进行CPU端OpenCL与GPU端OpenGL之间进行互操作，所以采用AMD APP所提供的OpenCL驱动。

生成OpenCL上下文cl\_context上下文时，之前采用clCreateContextFromType创建普通的OpenCL上下文cl\_context，当需要支持CL-GL互操作性时，需要创建特殊的OpenCL上下文cl\_context。先检测当前环境是否支持CL-GL互操作，检测步骤如下：1）因为互操作性不是OpenCL核心规范要求的通用API而属于OpenCL扩展，所以需要动态检测是否支持，主要通过clGetExtensionFunctionAddressForPlatform获取"clGetGLContextInfoKHR"功能函数的实际地址，如果获取成功，说明支持CL-GL互操作；2）通过OpenCL扩展接口clGetGLContextInfoKHR获取某个特定的OpenCL设备cl\_device\_id，该设备支持互操作；3）最后通过clCreateContext创建OpenCL上下文cl\_context。可以按照以下流程图创建支持CL-GL互操作的设备上下文：



图3-21流程图——创建支持CL-GL互操作的设备上下文

检测并开启OpenCL-OpenGL互操作Interoperability是很关键的一个步骤，接下来进一步修改算法实现互操作。互操作主要体现在内存空间，即OpenCL设备内存cl\_mem与OpenGL设备内存Buffer 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。立即渲染方式改为VBO渲染方式，OpenGL端涉及以下几处修改：

表3-7 OpenGL端和OpenCL端为了支持互操作各自做出的修改

|  |  |  |
| --- | --- | --- |
|  | 修改前 | 修改后 |
| 1  OpenGL  数据存储 | //顶点坐标，在内存  float\*pVertexOut = new float [ nSize ] ;  // pVertexArray[i]=…填入实际顶点数据  //面索引，在内存  float\*pFaceIndex = new float[nSizeI] ;  // pFaceIndex[i]=…填入实际面片索引数据 | // 顶点坐标VBO，在GPU  glGenBuffers ( 1, \_idVBO );  glBindBuffer (…\_idVBO );  glBufferData (…pVertexOut);  // 面索引VBO，在GPU  glGenBuffers (1, \_idVBOFace);  glBindBuffer(\_idVBOFace);  glBufferData(…pFaceIndex); |
| 2  OpenGL  数据渲染 | // 顶点及面片指针设为内存地址  glVertexPointer(  3,GL\_FLOAT,  3\*sizeof(float), pVertexOut );  glDrawElements(  GL\_TRIANGLES, nSizeVertex,  GL\_UNSIGNED\_INT ,  pFaceIndex); | // 指定GPU寻址  glBindBuffer (…\_idVBO );  glBindBuffer(\_idVBOFace);  // 顶点及面片指针设为0  glVertexPointer(  3,GL\_FLOAT,  3\*sizeof(float),  0 );  glDrawElements(  GL\_TRIANGLES,  nSizeVertex,  GL\_UNSIGNED\_INT ,  0 ); |
| 3  OpenCL  数据存储 | // 关联内存  cl\_mem  memBufferOut =clCreateBuffer( …pVertexOut); | // 关联GPU显存VBO  cl \_mem  memBufferOut =clCreateFromGLBuffer(  …\_idVBO); |
| 4  OpenCL  数据传输  和计算 | //执行OpenCL运算  clEnqueueNDRangeKernel();  // OpenCL cl\_mem  -> CPU -> OpenGL VBO  clEnqueueReadBuffer(  …  memBufferOut,  …  pVertexOut,  …); | // OpenGL VBO  -> OpenCL cl\_mem  clEnqueueAcquireGLObjects(  …memBufferOut );  //执行OpenCL运算  clEnqueueNDRangeKernel();  // OpenCL cl\_mem  -> OpenGL VBO  clEnqueueReleaseGLObjects(  …memBufferOut ); |

通过在OpenGL和OpenCL分别进行以上几个步骤的改造，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，不用每帧从主机端发送数据到OpenGL设备。实现OpenGL和OpenCL之间的互操作以后，OpenCL新版本算法的性能记录如下：

表3-8在CPU上OpenCL与OpenGL互操作骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法  版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |
| 并行  线程 | 3.28 | 0.18 | 5% | 0.98 | 30% | 0.75 | 23% | 89% | 12% |
| 并行  指令 | 1.95 | 0.18 | 9% | 0.96 | 49% | 0.75 | 38% | 14% | 12% |
| 并行  线程  与指令 | 3.04 | 0.18 | 6% | 0.52 | 17% | 0.75 | 25% | 89% | 12% |
| CPU OpenCL | 1.86 | 0.18 | 10% | 0.22 | 12% | 1.00 | 54% | 14% | 12% |
| CPU OpenCL  互操作 | 1.28 | 0.18 | 14% | 0.22 | 17% | 0.30 | 27% | 25% | 12% |

图3-22 CPU上OpenCL与OpenGL互操作骨骼动画的时间对比图

图3-23 CPU上OpenCL与OpenGL互操作骨骼动画的算法加速比

图3-24 CPU上OpenCL与OpenGL互操作骨骼动画的时间分布图

从以上图表得出结论，执行互操作以后，全局加速比从2.67提升至3.88，增幅达45%。主要原因是：OpenCL与OpenGL直接互操作，避免通过内存在两个设备之间中转数据，大幅度节省数据传输时间。从最终的时间分布图来看，除了数据传输部分，其它模块的耗时比率介于6%~27%，各部分时间分布相对均匀，已经没有明显的瓶颈。

## 实验结果分析和本章小结

针对虚拟现实骨骼动画渲染算法这一实例，本文对比研究了各种CPU上的并行计算技术，用这些技术加速算法效率并且对比分析它们的加速能力。本文先实现了骨骼动画渲染算法的多个CPU并行版本，包括传统SSE技术和OpenCL技术。除了基于SSE面向CPU的版本是沿用现有成熟技术，骨骼动画渲染算法其它两个CPU版本的并行计算实现都由本文提出。本文对各种并行计算算法的性能做了统计和分析对比。

结合本章多个有关骨骼动画算法加速比的图表，汇总得到SSE、CPU端OpenCL这2套并行计算方案的加速比对比图，汇总图表时设定每个顶点关联骨骼数目为2。汇总骨骼动画各种并行计算方案的加速比对比图如下：

图3-25骨骼动画CPU并行计算多方案的时间对比图

图3-26骨骼动画CPU并行计算多方案的加速比对比图

针对骨骼动画在CPU上的多套并行技术算法，从以上图中得出以下几个结论：OpenCL在各种CPU并行方案中性能最优，加速比高达3.9倍，是次优方案SSE的1.5倍。

# 骨骼动画渲染算法在GPU上的并行计算

在GPU上的并行计算有一个专门的概念叫做通用计算图形处理器（General Purpose GPU，GPGPU）。GPGPU历经四代技术，从图形汇编到着色语言GLSL，再到CUDA，最后发展为OpenCL，虽然更新换代，但它们之间并不能彼此取得，都有各自适用的领域。

## 基于GLSL面向GPU的骨骼动画矩阵调色板算法

### GLSL骨骼动画算法移植

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)。如前文2.4.1节所述，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();默认矩阵变换改为以下自定义矩阵变换

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();默认矩阵变换改为以下自定义矩阵变换

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。

### GLSL骨骼动画算法性能分析

采用100K顶点规模的动画数据，在显卡NVIDIA Geforce GTX670 上测量以下几组数据：GLSL骨骼动画算法的时间分布、GLSL骨骼动画全局和局部kernel的时间对比、GLSL骨骼动画的算法加速比。

表4-1 GLSL骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |
| CPU OpenCL | 1.28 | 0.18 | 14% | 0.22 | 17% | 0.30 | 27% | 25% | 12% |
| GLSL | 0.78 | 0.18 | 23% | 0.02 | 3% | 0.30 | 44% | 13% | 1% |

图4-1 GLSL骨骼动画的时间对比图

图4-2 GLSL骨骼动画的算法加速比

图4-3 GLSL骨骼动画的时间分布图

从GLSL骨骼动画的算法加速比图可以看出，GLSL并行计算获得最优性能，相对CPU串行和CPU OpenCL并行这两个参考基准，加速比分别是6.37和1.64。GLSL虽然具有较高性能，但是算法依赖于三维图形库OpenGL，必须把待处理的数据转换为顶点或者颜色数据，并把相关参数转换为顶点属性或全局变量，对于非图形学领域的研究人员来说是一个障碍。本文接下来研究完全脱离三维图形库的GPGPU技术，真正实现通用计算，使非图形学领域的研究人员只需付出很小的代价就可以调用GPU的多核并行计算能力。

## 基于CUDA面向GPU的骨骼动画矩阵调色板算法

### 未经优化并且功能简化的初步算法

上文采用OpenCL开放并行计算语言在GPU上实现了动画算法的通用计算任务，虽然性能相对CPU串行版本取得了较高的局部核心加速比和全局加速比，但某些GPU的加速效果有时甚至比不上CPU的OpenCL加速效果。因为GPU为了支持开放的通用的OpenCL接口，限制了本身硬件特性的发挥，特别是显存的管理，所以GPU还有很大的潜能有待挖掘。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。

首先为CUDA初始化运行环境，按照一定的筛选条件选定CUDA设备，因为一台计算机可能同时有多块显卡GPU，筛选条件往往包括：计算能力Compute Capability、浮点运算吞吐量Gflops、显存带宽Bandwidth。本文以最大浮点运算吞吐量Gflops作为筛选条件，筛选代码片段如下：

{// 遍历每个CUDA设备

// 查看CUDA设备属性

cudaDeviceProp deviceProp;

cudaGetDeviceProperties(&deviceProp, current\_device\_id);

// 设备计算能力不同，每个MP包含核数不同

int cores\_per\_multiproc = \_ConvertSMVer2Cores(deviceProp.major, deviceProp.minor);

// 浮点运算能力Gflops = MP个数 \* 每个MP包含核数 \* 每个核的 主频

int compute\_perf = deviceProp.multiProcessorCount \* cores\_per\_multiproc \* deviceProp.clockRate;

// 筛选最大Gflops

}

接着设计Kernel函数，编写CUDA程序时，一般先实现一个功能简化的程序，然后不断完善功能并且优化性能。本文先设计一个仅支持绑定一块骨骼的CUDA算法，算法基于上文的OpenCL的Kernel函数进行修改。CUDA的Kernel相比OpenCL的Kernel，有以下几处不同点：

表4-2 CUDA的Kernel与OpenCL的Kernel的语法对照

|  |  |  |
| --- | --- | --- |
| Kernel不同点 | OpenCL | CUDA |
| 函数前缀 | \_\_kernel | \_\_global\_\_ |
| 函数参数前缀 | \_\_global | 无 |
| 线程索引 | 线程ID：get\_global\_id(0)  块ID：get\_global\_id(1)  块大小：get\_global\_size(0) | 线程ID：threadIdx  块ID：blockIdx  块大小：blockDim |
| float4构造 | float x ;  float4 xxxx = (float4)x ; | float x ;  Vector4 xxxx = make\_ Vector4(x, x, x, x) ; |
| float4算法+-\*/ | 自带 | 必须自己实现 |

经过以上不同点一一转换后，CUDA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：

1. 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。

void initializeCUDA()

{

// 第一步，Host主机端为CUDA Kernel提供实参数据

float\*h\_pInput = pMesh->pVertexArrayRaw; // 输出顶点pOutput同理

// 第二步，Device设备端为CUDA Kernel提供实参数据结构

float \*d\_pInput, \*d\_pOutput, \*d\_pMatrix, \*d\_pIndex, \*d\_pWeight ;

intnElementSize = … ; // 顶点数目

// 第三步，设置Kernel实参：初始顶点坐标，Host -> Device

nBufferSize = 4 \* nElementSize \* sizeof(float);

cudaMalloc(&d\_pInput, nBufferSize) ;

cudaMemcpy(d\_pInput, h\_pInput, nBufferSize, cudaMemcpyHostToDevice ); // 输出顶点pOutput和矩阵同理

}

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API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cudaGraphicsResourceGetMappedPointer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。

1. 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。

表4-3 CUDA动态线程块对比静态线程块

|  |  |  |
| --- | --- | --- |
|  | 线程块数动态变化 | 线程块数静态不变 |
| Kernel外  初始化  线程结构 | int nCountT = 256;  dim3 block(nCountT);  int nCountB = (sizeMax + nCountT - 1) / nCountT ;  dim3 grid( nCountB); | int nCountT = 256;  dim3 block(nCountT);  int nCountB = 64;  dim3 grid( nCountB); |
| Kernel内  调用  线程索引 | // 处理索引为线程id(indexBase)的一个元素  const int  indexBase =  ( gridDim.x \* blockIdx.y  + blockIdx.x ) \* blockDim.x + threadIdx.x; | // 处理起点为线程id(indexBase)的多个元素，并且间隔为blockDim \*gridDim  for(indexBase;; index+=blockDim.x\*gridDim.x )  {  } |

1. 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 );

}

### 完善算法实现对多骨骼的支持

以上算法粗略实现了算法的雏形，功能和性能都没有完善，但是整个CUDA运行环境以及骨骼动画算法的CUDA版本基本成形。本文接下来在此基础上，渐进完善算法功能。功能方面，已实现单骨骼的情况，至于多骨骼，需要将单个顶点关联的多个骨骼的矩阵按权重进行累加。本文在OpenCL C语言版本的基础上，转换成CUDA C语言的矩阵累加算法，主要有以下几处改动：

\_\_global\_\_ void

kernelCUDA(Vector4 \*pInput,pOutput, pMatrix,int \*pIndex, float \*pWeight)

{

…

// 获取第1个矩阵，乘于权重，第1处改动

Vector4 m0 = pMatrix[offset+0] \* weight4 ; // m1/2/3 同理

for(int i=1;i<SIZE\_PER\_BONE; i++)

{ // 获取第i个矩阵的权重，同OpenCL C语言算法

offset = pIndex[index\*SIZE\_PER\_BONE+i]\*4 ;

weight = pWeight[index\*SIZE\_PER\_BONE+i] ;

weight4 = make\_Vector4( weight, weight,weight,weight ) ; //第2处改动

// 第i个矩阵乘于权重，第3处改动

m0 += pMatrix[offset+0] \* weight4 ;// m1/2/3 同理

}

// 顶点(x,y,z,1) 拆分为(x,x,x,x) 、 (y,y,y,y) 、(z,z,z,z) ，第4处改动

Vector4 pIn = pInput[index];

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

// 矩阵变换，多项式相乘累加，第5处改动

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

}

完整功能的骨骼动画算法，从OpenCL版本转成CUDA版本，涉及5处改动，可以分为两个类别。第2、4处改动属于第一类，即Vector4数据结构赋值；第1、3、5处改动属于第二类，即Vector4的算术运算。

完整算法完成后，本文测试了未经优化的CUDA算法的功能和性能，作为后续优化的参照。功能方面，本文用调试的方法验证算法的正确性，本文用到的CUDA调试程序是NVIDIA提供的Nsight软件的Visual Studio Edition版本，Nsight支持在设备端单步运行指令，便于查看程序运行到任意代码行时的状态，通过查看参数变量值以及运算结果是否符合预期验证算法功能的准确性。性能方面，本文一方面继续全面记录骨骼动画渲染总时间，一方面局部测试CUDA参数传输时间以及Kernel运行时间，因为CUDA的Kernel在GPU设备端相对CPU主机端异步执行，默认不等待Kernel执行完成CPU就继续执行，所以为了准确测时需要在Kernel启动执行后接着调用等待指令cudaDeviceSynchronize。首个功能完整版的CUDA算法在NVIDIA GPU GTS250上的测时结果如下：

表4-4未优化CUDA骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |
| CPU OpenCL | 1.28 | 0.18 | 14% | 0.22 | 17% | 0.30 | 27% | 25% | 12% |
| GLSL | 0.78 | 0.18 | 23% | 0.02 | 3% | 0.30 | 44% | 13% | 1% |
| CUDA  未优化 | 1.22 | 0.18 | 15% | 0.17 | 14% | 0.34 | 28% | 14% | 4% |

图4-4未优化CUDA骨骼动画的时间对比图

图4-5未优化CUDA骨骼动画的算法加速比

图4-6未优化CUDA骨骼动画的时间分布图

从以上图表可以看出，刚开始从别的语言移植成CUDA C语言的算法，性能连CPU单线程版本都不如。可见，CUDA性能并不是那么唾手可得，结合硬件和算法特性对CUDA进行优化的工作必不可少，接下来，本文根据CUDA的优化规范，对以上实现的功能完整但性能未优化的骨骼动画算法进一步优化，力求挖掘CUDA的性能潜力。CUDA的性能潜力主要指运算能力和带宽，这两个指标参数GPU往往是CPU的5~10倍，所以理论上GPU的性能相对CPU有线性比例的提升。GPU和CPU两项指标的运算方法如下表：

表4-5 GPU和CPU的Gflops和Bandwidth两项指标的运算方法

|  |  |  |
| --- | --- | --- |
|  | 浮点计算能力Gflops | 存储带宽Bandwidth GB/s |
| CPU | CPU核数 \* CPU主频 \*  4 flops/cycle | 内存总线位数/8 \* 内存频率/8 \*倍增系数 |
| GPU | GPU的SP个数 \* Shader频率 \*  2 flops/SP | 显存总线位数/8 \* 显存频率 |

例如本文采用2款CPU和2款GPU，根据以上表格计算方法，本文硬件指标参数如下：

表4-6本文实验设备硬件指标参数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | CPU Intel  i7 3770k  DDR3 1333Dual | CPU Intel  i7 870k  DDR3 1333Dual | GPU NVIDIAGeforce GTX 670 | GPU NVIDIAGeforce GTS 250 |
| 浮点计算能力  Gflops | 核数4  频率3.9  4 \* 3.9 \* 4 =  62.4 | 核数4  频率2.9  4 \* 2.9 \* 4 =  46.4 | 核数1344  频率0.9  1344 \* 0.9 \* 2 =  2419 | 核数128  频率1.8  128 \* 1.8 \* 2 =  461 |
| 存储带宽  Bandwidth  GB/s |  | 位数128  频率1.3  128/8 \* 1.3/8 \* 8=21 | 位数256  频率6  256 / 8 \* 6 =  192 | 位数256  频率2  256 / 8 \* 2 =  64 |

### 根据CUDA硬件特性和算法特性优化算法

* 全局显存(Global Memory)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。满足CUDA显存对齐后，测得算法性能如下：

表4-7显存对齐CUDA骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |
| CPU OpenCL | 1.28 | 0.18 | 14% | 0.22 | 17% | 0.30 | 27% | 25% | 12% |
| CUDA  未优化 | 1.22 | 0.18 | 15% | 0.17 | 14% | 0.34 | 28% | 14% | 4% |
| CUDA  对齐 | 1.15 | 0.18 | 16% | 0.14 | 9% | 0.34 | 30% | 14% | 4% |

从以上图表可以看出，将全局显存Global Memory进行对齐后，CUDA的局部kernel和全局执行效率细微提升6%，因为本组数据采用的最新显卡是NVIDIA GeForce GTX 670，弱化了显存对齐的要求。对于另一款老式显卡NVIDIA GeForce GTS 250，显存对齐使得kernel性能提升3.4倍，全局性能提升1.8倍。

* 处理器利用率(Multiprocessor Occupancy)

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，本文每个线程使用31个寄存器，注意当运行Visual Profiler时请关闭OpenGL渲染，否则无法正常搜集kernel状态。通过Visual Profiler还可以获取硬件参数，计算能力CP(Compute Capability)和每个block可用的寄存器个数rpb，例如NVIDIA Geforce GTS 250的CP是1.1，rpb是8K。然后使用CUDA Toolkit提供的工具表格“CUDA\_Occupancy\_Calculator.xls”，设定每个block线程数为256，设定每个线程使用寄存器个数为31，查表得到每个MP激活块数是1、激活线程数是256、激活warp数是8，而每个MP的线程容量是768、激活warp数是24，所以利用率是33%(8/24)。能够优化利用率的参数是每个块的线程数，按32的倍数修改线程数，得到最优数值是256，最大处理器利用率是33%。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。因为每个MP拥有的共享存储器容量有限，后文进行共享显存优化时将具体说明。

线程块block的内部线程维度设定完成后，需要设置网格grid的内部块维度，根据前文4.1节，有动态和静态2中设置方案，即块数根据问题规模需要处理的元素个数动态变化，或者保持不变。块数变化的方案，每个线程处理1个元素；而块数不变的方案，每个线程处理多个元素。前文已经使用第一种方案，本文继续实现第二组方案。第二种方案，每个线程处理多个元素，多个元素可以是连续的或者等间隔的，而后者性能比前者高30%，具体原因下一节将详细说明。另外经实测发现，动态网格维度与等间隔静态网格维度的性能相差无几。

* 合并访问(Coalesced Access)

CUDA线程执行时以warp分组进行，每个warp内包含32个线程数，半个warp内的线程同时在一个多处理器MP上执行，另外的warp处于候选状态，一旦当前warp处于数据饥饿状态，别的warp便乘虚而入填补处理器空闲时间，使得处理器始终处于运行状态。而处于更高层次上的多个MP也是各自并行执行，每个MP内有多个warp处于激活状态，每一个时刻有唯一的warp处于运行状态。MP一次读写数据有一个最大位数，往往是256bit即32字节，可以一次读取8个float。一个warp内的线程同时执行时，往往需要读写全局显存Global Memory，如果相邻线程读取相邻数据，多个线程多次读取数据可以合并为更少次数的读写操作。比如每个线程读写1个或2个float，半个warp读写16或32个float，一次就能读写完成，而不需要每个线程各自读写1次总共读写16次，从而将读写操作次数从16降到1。CUDA的kernel执行时间主要包括指令执行时间和内存读写时间，对于数据密集型问题来说，内存读写时间往往是性能的瓶颈，所以在上一环节对齐的基础上，进一步调整数据结构以满足合并访问条件，将很大程度提升程序的性能。

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。第3类是矩阵行列值满足合并访问条件以后，性能没有提升反而降低，所以取消第3类的改造。综合全部3类显存访问，将不满足合并访问的存储结构改造成满足合并访问的结构以后，实际上只有第2类进行了改造。第1类本来就满足合并条件，第3类是随机访问无法构造合并条件。改造第2类显存访问即矩阵索引和权重，使之满足合并条件，kernel性能提升10%，全局渲染性能提升3%。合并访问显存对一般的CUDA算法性能提升影响较大，本文算法比较特殊，3处待优化点，1处本来就满足合并条件无需优化，1处无法优化，仅有访问显存数据量最小的1处可以优化，所以整体优化效果不明显。

遍历1个顶点关联的多块骨骼的索引和权重代码片段修改如下：

表4-8是否满足合并访问条件的索引方式对比

|  |  |  |
| --- | --- | --- |
|  | 不满足合并访问条件 | 满足合并访问条件 |
| 矩阵位置 | offset = pIndex[index\*SIZE\_PER\_BONE+ i ] ; | offset = pIndex[index+ i \*sizeMax] ; |
| 矩阵权重 | weight=pWeight[index\*SIZE\_PER\_BONE+ i ] ; | weight = pWeight[index+i \*sizeMax] ; |

其中，SIZE\_PER\_BONE是骨骼数，sizeMax是顶点数，index是线程ID。通过数据结构和访问方式的改造，原先单个线程内连续访问多个值的情况，变成相邻多个线程连续访问多个值，从而满足合并条件。满足合并条件的标志是，没有与线程ID相乘的系数。例如本文算法改进前index有与之相乘的系数SIZE\_PER\_BONE，改进后去除系数。

合并访问矩阵索引和权重，性能实测结果如下：

表4-9合并访问CUDA骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |
| CPU OpenCL | 1.28 | 0.18 | 14% | 0.22 | 17% | 0.30 | 27% | 25% | 12% |
| CUDA  未优化 | 1.22 | 0.18 | 15% | 0.17 | 14% | 0.34 | 28% | 14% | 4% |
| CUDA  合并 | 1.22 | 0.18 | 15% | 0.16 | 13% | 0.34 | 28% | 14% | 5% |

从以上图表可以看出，进行合并访问优化后，CUDA的局部kernel和全局执行效率不变，因为本实验主体数据顶点坐标已经支持合并访问，矩阵索引和权重的改进只算小改，所以改进效果不明显。

* 常量显存(Global Memory)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。常量显存的申请、赋值、使用代码片段如下：\_\_constant\_\_ float4 const\_pMatrix[ 100 \* 4 ]; // 100个矩阵，4个float4/矩阵

cudaMemcpyToSymbol( (void\*)const\_pMatrix\_f4, (float\*) pMatrix, …);

float4 mat0 = const\_pMatrix\_f4[ index ];

其中，常量显存的实际用量nSizeConst必须是常数，大小就是数组的字节长度。即便实际骨骼数少于100，主机端存储骨骼矩阵的内存pMatrix也需要填充到容纳100个矩阵的空间长度。如果nSizeConst跟\_\_constant\_\_申请常量显存的实际长度不符，cudaMemcpyToSymbol将无法正常拷贝数据，导致kernel运行结果不正确。将骨骼矩阵存入常量显存后，算法性能实测结果如下：

表4-10 常量显存CUDA骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |
| CPU OpenCL | 1.28 | 0.18 | 14% | 0.22 | 17% | 0.30 | 27% | 25% | 12% |
| CUDA  未优化 | 1.22 | 0.18 | 15% | 0.17 | 14% | 0.34 | 28% | 14% | 4% |
| CUDA  常量 | 1.15 | 0.18 | 16% | 0.11 | 10% | 0.34 | 30% | 14% | 5% |

采用常量显存后，效率小幅度提升6%，因为显卡NVIDIA GeForce GTX 670的显存本身有缓存，导致常量显存的缓存功能效果不明显。对于另一款老式显卡NVIDIA GeForce GTS 250，常量显存使得kernel性能提升5.8倍，全局性能提升2.2倍。常量显存申请的长度也必须是常数，但实际应用中需要缓存的数据长度往往是动态变动的，所以在使用常量显存时不大方便，往往需要估计待缓存数据最大长度，而待缓存数据的长度需要统一填充成静态长度。本文接下来使用CUDA的另一种共享存储器即共享显存(Shared Memory)。

* 共享显存(Shared Memory)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)，如表2.4.2b《CUDA各类存储类型的特性参数》。共享显存的申请，针对静态长度和动态长度有2种不同的途径：静态长度直接在kernel内定义的数组前填写前缀\_\_shared\_\_；动态长度在调用kernel时传入所需共享显存的容量大小参数。共享显存的赋值。共享显存的使用跟在CPU的内存空间使用普通的数组一样。共享显存的申请、赋值、使用代码片段如下：

\_\_shared \_\_ float4shared\_pMatrix[ 100 \* 4 ];// 100个矩阵，4个float4/矩阵

//第二步，赋值，在kernel函数最开始所有线程使用变量之前用\_\_syncthreads确保赋值完成

if( threadIdx.x < sizeJoint )

{

for(int i=0;i<MATRIX\_SIZE\_LINE;i++)

shared\_pMatrix\_f4[ threadIdx.x + i\*sizeJoint ] = pMatrix[ threadIdx.x + i\*sizeJoint ];

}

\_\_syncthreads();

//第三步，使用共享显存，用法同普通数组

float4 mat0 = const\_pMatrix\_f4[ index ];

本文采用静态长度的共享显存，也可以采用动态长度。动态长度共享显存的申请方法是，在调用kernel时传入参数指定长度，在kernel内直接使用。指定长度的参数设置为<<<grid,block, size>>>第三个参数，kernel内使用前需要做形如”extern \_\_shared\_\_ float sharedVariable[]”的声明，即声明为没有长度参数而且有前缀extern的外部引用数组。因为每个MP配备共享显存的容量有限，所以每个block线程块的用量会影响MP可同时容纳的线程块的数目。比如GPU计算能力1.x，每个MP共享显存容量是16K；计算能力2.x以上容量是48K。本文算法每个块用量是100个矩阵6K，支持8个块，完全不影响可同时激活的线程数目。将骨骼矩阵存入共享显存后，算法性能实测结果如下：

表4-11 共享显存CUDA骨骼动画算法的时间分布对比表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法版本 | 总共耗时  ms | CPU  计算骨骼 | | CPU  计算顶点 | | GPU  渲染面片 | | 处理器  占用率 | |
| 耗时 | 比率 | 耗时 | 比率 | 耗时 | 比率 | CPU | GPU |
| 串行  优化 | 4.97 | 0.18 | 4% | 3.93 | 79% | 0.75 | 15% | 14% | 12% |
| CPU OpenCL | 1.28 | 0.18 | 14% | 0.22 | 17% | 0.30 | 27% | 25% | 12% |
| CUDA  未优化 | 1.22 | 0.18 | 15% | 0.17 | 14% | 0.34 | 28% | 14% | 4% |
| CUDA  共享 | 1.20 | 0.18 | 15% | 0.13 | 10% | 0.34 | 28% | 14% | 4% |

从以上加速比图可以看出，使用共享显存时的性能，不如常量显存的性能，所以选用常量显存作为缓存骨骼矩阵行列值的CUDA存储空间类型。共享显存改进效果不明显，因为显卡NVIDIA GeForce GTX 670的显存本身有缓存，导致共享显存的缓存功能效果不明显。对于另一款老式显卡NVIDIA GeForce GTS 250，共享显存使得kernel性能提升4.3倍，全局性能提升1.9倍。

### CUDA算法性能分析

本文按照CUDA C的语言规范，将骨骼动画渲染算法中的顶点更新部分，改写成能够在CUDA设备上执行的kernel。最初版本只是移植算法，使得算法功能运行成功，即成功更新顶点。至于骨骼动画的完整渲染过程，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API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cudaGraphicsMapResources和cudaGraphicsResourceGetMappedPointer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。

CUDA版本的骨骼动画渲染算法初步实现以后，性能跟CPU单线程版本差不多，即几乎没有加速。需要按照CUDA规范结合硬件和算法特性经过一系列优化，发挥CUDA的真正性能。CUDA性能优化过程至关重要，程序设计的大部分时间将花费在这上面。

优化包括以下5个环节，全局显存(Global Memory)对齐、处理器利用率(Multiprocessor Occupancy)、合并访问(Coalesced Access)、常量显存(Global Memory)缓存共享数据、共享显存(Shared Memory)缓存共享数据。5个环节构成一个优化空间，其中处理器利用率环节没有优化余地，所以没有性能提升，其它4个环节均不同程度地提升CUDA性能。CUDA优化加速比对照图表如下：

表4-12 CUDA独立优化加速比对照表

|  |  |  |
| --- | --- | --- |
|  | 局部 | 全局 |
| 未优化 | 23 | 4.1 |
| 对齐 | 28 | 4.3 |
| 合并 | 25 | 4.1 |
| 常量 | 36 | 4.3 |
| 共享 | 30 | 4.1 |

图4-7CUDA独立优化加速比对照图

表4-13 CUDA叠加优化加速比对照表

|  |  |  |
| --- | --- | --- |
|  | 局部 | 全局 |
| 未优化 | 23 | 4.1 |
| 对齐 | 36 | 4.3 |
| 合并 | 33 | 4.3 |
| 常量 | 33 | 4.3 |
| 共享 | 39 | 4.4 |

图4-8 CUDA叠加优化加速比对照图

从图表中分析得到如下结论，当同时采用显存对齐、合并访问、常量显存这3种优化组合时，算法获得最高性能，局部加速比39，全局加速比4.4。确立最终的CUDA优化方案后，与之前的GLSL和CPU方案对比如下：

图4-9 CUDA叠加优化骨骼动画的时间对比图

图4-10 CUDA叠加优化骨骼动画的时间对比图

## 基于OpenCL面向GPU的骨骼动画矩阵调色板算法

上一章在CPU上实现了OpenCL算法，因OpenCL本身具有良好的可移植性，可以快速移植到其它OpenCL设备，比如GPU。本文接下来在GPU上实现并运行OpenCL算法，验证是否能在核数成百上千的GPU上使算法性能得到质的飞越。

移植过程需要设计一些代码上的修改，同时需要更新GPU显卡驱动，比如本文采用NVIDIA显卡GTS250，NVIDIA 2010年以后的显卡驱动都支持OpenCL，建议更新为最新版本的驱动。至于显卡特有的OpenCL SDK，可以不安装。回到代码修改这部分，主要涉及下文所述的修改。OpenCL的设备运行平台Platform的选择，之前通过标示符”Intel(R) OpenCL”选择Intel CPU，现在通过标示符”NVIDIA CUDA”选择NVIDIA GPU。代码片段摘录如下：

// 返回所需的OpenCL设备平台

cl\_platform\_id GetOCLPlatform( char\* namePlatform )

{

for (cl\_uint ui = 0; ui < uiPlatformsCount; ++ui)

{

// 第一步，获取每个OpenCL驱动平台

clGetPlatformInfo(pPlatforms[ui], CL\_PLATFORM\_NAME,128 \* sizeof(char), pPlatformName, NULL);

// 第二步，按条件选择平台，标示符可能是以下几种:

// “Intel(R) OpenCL”、“Advanced Micro Devices, Inc.”、“NVIDIA CUDA”

if (!strcmp(pPlatformName, namePlatform ))

return pPlatforms[ui];

}

}

调用GetOCLPlatform(”NVIDIA CUDA”)，得到NVIDIA GPU对应的平台cl\_platform\_id，将它作为参数传递给clCreateContext创建OpenCL上下文cl\_context，使得后续的OpenCL指令运行在GPU上。

将OpenCL程序从CPU移植到GPU上，需要修改的地方只有上文提到的一个地方，即把平台标示符改为表示GPU的”NVIDIA CUDA”。至于OpenCL程序的其它环节，包括OpenCL编译环境、运行环境、执行参数配置、结果返回，都是可移植的，不需要作任何改动。在GPU上实现了OpenCL算法以后，算法性能测试结果如下：

图4-11 GPU上OpenCL骨骼动画的时间对比图

图4-12 GPU上OpenCL叠加优化骨骼动画的时间对比图

从以上图表看来，GPU上OpenCL的性能，介于CUDA和GLSL之间，相对于CPU上的串行和OpenCL并行版本，加速比分别是4.97和1.28。虽然GPU上OpenCL的性能不如GLSL，但是OpenCL具有最好的可移植性，可以同时运行于CPU、GPU和其它设备。性能次优方案OpenCL的加速比大约是最优方案GLSL的80%，权衡算法的移植性，OpenCL方案可作为整体最优方案。

## 实验结果分析和本章小结

在上一章研究骨骼动画在CPU上的各种并行技术技术的基础上，本文进一步对比研究了在GPU上的各种并行计算技术。本文为骨骼动画渲染算法新增了多个GPU版本，从Shading Language到CUDA再到OpenCL，连续三代GPGPU技术，第一代到第二代解除图形库依赖提升软件通用性和可移植性，第二代到第三代解除图形硬件依赖提升硬件通用性和可移植性。除了基于GLSL面向GPU的版本这个版本是沿用现有成熟技术，骨骼动画渲染算法其它两个版本的并行计算实现都由本文提出。本文对各种并行计算算法的性能做了统计和分析对比。

结合本章多个有关骨骼动画算法加速比的图表，汇总得到各种并行计算方案的加速比对比图，包括GLSL、CUDA、GPU端OpenCL总共3套并行计算方案，汇总图表时设定每个顶点关联骨骼数目为2。汇总过程，采用两套不同硬件配置，分别统计时间和加速比。汇总骨骼动画各种并行计算方案的加速比对比图如下：

图4-13骨骼动画GPU并行计算方案的耗时对比图

图4-14骨骼动画GPU并行计算方案的加速比对比图

本章3套方案，连同上一章在CPU上的2套并行计算方案，汇总得到以下对比图：

图4-15骨骼动画各种并行计算方案的耗时对比图

图4-16骨骼动画各种并行计算方案的加速比对比图

图4-17 GPU上OpenCL骨骼动画的算法加速比与模型复杂度的关系

针对骨骼动画算法从以上图中得出以下几个结论：1）GLSL在各种GPU方案中最优，OpenCL在各种CPU方案中最优，而且GLSL性能高于OpenCL；2）无论是CPU还是GPU，OpenCL算法都是最优或次优方案，作为次优方案时性能不低于最优方案的7成；3）为了使算法的功能和性能都具有良好的跨平台可移植性，建议统一采用OpenCL方案，虽然OpenCL在GPU方案子集中有时不是最优；4）基于CUDA的骨骼动画矩阵调色板GPGPU算法，以CPU串行算法和基于SSE的传统并行算法作为性能参考基准，CUDA算法加速比分别是4.4和2.5。

# 面向虚拟现实系统的多任务异构协同并行计算

## 基于柏林噪声风场扰动的喷泉粒子系统模拟

### 喷泉粒子系统模拟

虚拟场景中特效的模拟仿真的首要工作是建立特效的模型，即真实环境中的雨雪、喷泉等在计算机中呈现的静态或动态画面，本文采用了粒子系统建模方法。粒子系统的基本思想是：采用大量的、具有一定生命和属性的微小粒子图元作为基本的元素来表示不规则的模糊的物体。粒子的“产生”、“活动”和“死亡”使粒子系统可以生成一系列运动的场景，通常，粒子系统生成一帧画面的基本步骤为: (1)、在系统中产生新的粒子；(2)、赋予每一个新粒子一定的初始属性；(3)、删除系统中已存在且超过生存期的粒子；(4)、对剩下的粒子根据运动规律及相关算法对进行移动和更新；(5)、绘制由有生命的粒子组成的图像。

作为构造形状“模糊”的物体的计算模型的方法，粒子系统最早由Reeves在1983年提出[55]。其基本思想是：采用大量的、具有一定生命和属性的微小粒子图元作为基本的元素来表示不规则的模糊的物体。在粒子系统中，每一个粒子均具有：形状、大小、颜色、透明度、位置、速度和生命期等属性，所有这些属性都是时间t的函数，随着时间的变化，每个粒子都要经历“产生”、“活动”和“死亡”三个阶段。粒子系统不是一个简单的静态系统，随着时间的推移，不仅系统中已有的粒子不断运动，其属性不断变化，而且不断有新的粒子加入，超过生命期的旧粒子消失。1985年Reeves和Blau发展了粒子系统[56]，他们用volume filling基本单元去生成随时间改变形状但又基本保持不变的实体，如：随风飘动的花草树叶等。此后，Reeves又提出了关于粒子系统消隐绘制的算法，称之为“近似概率性算法”，用于解决阴影、可见面问题，取得了很好的效果。1992年Loke等人提出了节日焰火的粒子系统绘制算法，他们采用链表数据结构存储粒子的信息，设计了粒子系统绘制引擎 (Particle System Rendering Engine )，用粒子衍生的方法表现焰火粒子的轨迹并实现了多种焰火的特殊效果。Richard和David提出了基于粒子系统的变形物体表面建模方法，引入有向粒子的概念，该方法可实现变形物体表面断开、连接或延伸等效果。

粒子系统理论主要由以下几个部分组成：(1)、物质的粒子组成假设。粒子系统中，任何模拟对象，不论是固态的、液态的或者气态的，例如：火焰、云、雨和雪等，都是由有限数目的简单粒子构成，而且这些粒子处于不断的运动状态中，这个假设符合物理学定律：“物体都是由最基本的微粒构成的”，但是这些粒子与物理学中的原子等基本微粒不同，它们具有一定的形状、大小、颜色、透明度、位置、速度和生命期等属性，而且这些属性随着时间不断地发生变化。(2)、粒子的独立关系假设。这里包含两个意思，一是粒子系统中各个粒子不与场景中任何物体相交，二是粒子之间不存在相交关系，并且粒子是不可穿透的。(3)、粒子的生命机制。粒子系统中每个粒子都具有生命周期，在一定的时间周期内，粒子经历“产生”、“活动”和“死亡”三个基本的生命历程。(4)、粒子的运动机制。粒子在存活期间按照一定的运动规律在场景中运动，粒子的运动机制是粒子系统模拟动态景物的关键。(5)、粒子的绘制。因为构建一幅场景需要大量的粒子，因此在绘制时我们通常假设： I)单个的粒子不具有阴影，但是由粒子组成的实际物体则根据模拟场景的需要来决定是否进行阴影处理；II)单个的粒子不反射光线，但是由粒子组成的实际物体则根据模拟场景的需要来决定是否反射光线。

粒子系统理论是迄今为止用于描述不规则物体最成熟的理论之一，它采用了一套完全不同以往造型、绘制系统的方法构造、绘制景物，景物被定义为由成干上万个不规则的、随机分布的粒子组成，它们不断改变形状、不断运动，因此，粒子系统的被广泛应用于自然景物的模拟。下图总结了粒子系统在各个方面的应用，其中结构化的粒子系统通常用来模拟树木、树叶、水滴、草、彩虹和云等;随机粒子系统通常用来模拟焰火、爆炸和雪等;方向粒子系统通常用来模拟可变形的物体或者刚性物体，如：织物和岩浆等。

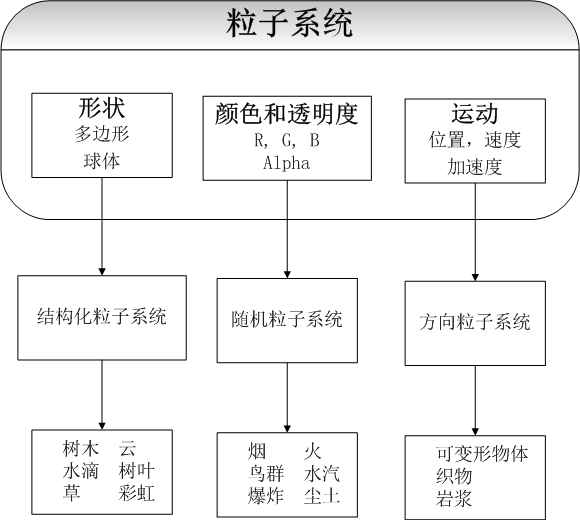


图5-1 粒子系统的应用

新粒子初始化完成后，粒子根据初始化时速度的大小及方向开始在系统环境中运动，同时粒子的其它属性值也随时间的推移发生变化。粒子的空间位置主要由粒子的前一位置、粒子的速度和每帧的时间间隔决定。假设第j+1帧与第j帧的时间间隔为 (下文皆同)，则第j+1帧时粒子的空间位置为：

|  |
| --- |
| ＝ + \* 式（5-1）  ＝ + \* 式（5-2）  ＝ + \* 式（5-3） |

粒子的速度由粒子的初速度和粒子所受的和外力决定，粒子系统中常见的力有：重力、风力、浮力和粒子之间碰撞产生的力等，为了模拟复杂环境中粒子的运动，可以引入“场”的概念，如重力场、风力场和漩涡场等，通过控制这些场的参数，就可以控制粒子的运动轨迹，使粒子按照一定的规律运动。本文采用柏林噪声Perlin Noise模拟风场干扰源。假设粒子在和外力F的作用下产生的加速度为Acceleration，则第j+1帧时粒子的速度为：

|  |
| --- |
| ＝ + Acceleration \* 式（5-4） |

其中Velocity和Acceleration是三位空间中的矢量。

粒子的其它属性的变化一般由下式计算：

|  |
| --- |
| ＝PropertyDelta \* 式（5-5） |

本文采用基于OpenCL的并行计算方案改进原先的粒子系统算法，主要改进粒子的坐标移动和属性更新环节。首先根据粒子系统概念，为粒子系统建立数据结构和算法，粒子数据结构主要是属性数据，包括球形坐标、世界坐标、速度、加速度、生命期、激活状态，定义数据结构如下：

struct tagDROP

{

Vec3 vgen; // 球形坐标系(r , a, b)，柱状发射模式

Vec3 position; // 世界坐标系(x , y, z)

Vec3 vlen; // 速度

Vec3 acc; // 加速度

int lifeFrame; // 生命期

bool active; // 激活状态

}; // 粒子属性

算法方面主要涉及坐标重置与坐标更新，其中坐标更新操作每一帧都要执行，而坐标重置操作在生命期结束的条件下执行。坐标重置与坐标更新算法示意如下：

void vgFountain::resetPosition(int index)

{

// 第一步，重置坐标

m\_pdrop[index].position = m\_Position ;

// 第二步，重置加速度

m\_pdrop[index].vgen.x = m\_height/100.0f; //5.0f/100

m\_pdrop[index].vgen.y = float (rand()%(int)m\_width + 2 ) \* (float)PI / 180.0f ;

m\_pdrop[index].vgen.z = float (rand()%360) \* (float)PI / 180.0f ;

// 第三步，重置速度。柱坐标r,a,b转x,y,z坐标

m\_pdrop[index].vlen.x=m\_pdrop[index].vgen.x\* sin(m\_pdrop[index].vgen.y )\*cos( m\_pdrop[index].vgen.z );

m\_pdrop[index].vlen.y= m\_pdrop[index].vgen.x \* cos( m\_pdrop[index].vgen.y ) ;

m\_pdrop[index].vlen.z=m\_pdrop[index].vgen.x \* sin( m\_pdrop[index].vgen.y )\*sin( m\_pdrop[index].vgen.z ) ;

}

void vgFountain::updatePosition(int index)

{

// 加速度更新速度，速度更新坐标

m\_pdrop [index].position+= m\_pdrop [index].vlen ;

m\_pdrop [index].vlen += m\_pdrop [index].acc;

// 柏林噪声Perlin Noise模拟风场，影响x轴(由东向西方向)加速度

m\_pdrop [index].acc.x = perlinNoise(m\_pdrop [index].position );

}

### 柏林噪声风场扰动

人们通常采用随机数发生器为程序提供随机数，用于模拟自然现象，使得现象的动作和行为看起来更加随意和自然。随机数发生器有它的用途，但有时它们的输出过于生硬而不自然，如下左图。柏林噪声的发明就是为了解决随机数过于生硬的问题，它最早由柏林（Perlin）提出，后来被命名为柏林噪声（Perlin Noise），柏林噪声如下右图。



图5-2 普通随机数和柏林噪声随机数产生的灰度图

如果你仔细看大自然中的许多事物，你会发现它们很多是分形的，它们拥有不同层次的相似细节。以山脉的轮廓为例，它在海拔上具有大规模的变化（山峰），又有中等规模的变化（小山），还有小型变化（巨石）和微型变化（小石头）；诸如此类的分形现象还有，海浪的波动、树枝的摇摆、风场的分布。所有这些现象呈现出大规模变化叠加小范围变化的某种模式。柏林噪声通过叠加不同规模的噪声函数来重建这种模式。构成柏林噪声的每一个噪声函数都是一个连续函数，由离散随机序列插值生成。柏林噪声生成算法的流程图如下：

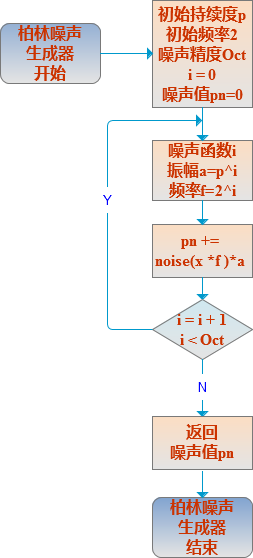
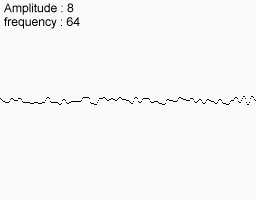
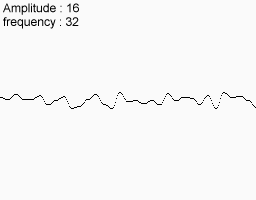
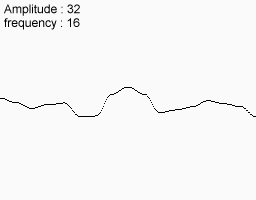
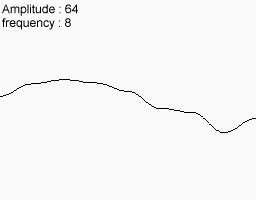
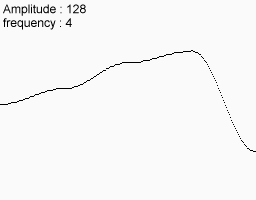


图5-3 柏林噪声生成算法的流程图

柏林噪声随机数分为一维和二维，一维柏林噪声由一组不同幅度的随机曲线叠加形成，类似地，二维柏林噪声由一组随机图像叠加形成。一维柏林噪声和二维柏林噪声的形成过程示意图如下：



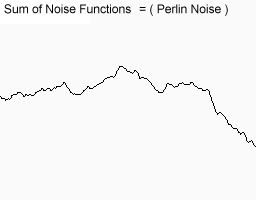


图5-4 一维柏林噪声





图5-5 二维柏林噪声

### 性能瓶颈分析

坐标重置与坐标更新再加上渲染部分，粒子系统比较耗时间的环节主要是这三部分，本文先实现串行算法并测量粒子系统每帧的时间分布。本文设置如下：数据方面，粒子数目设定为10000。

图5-6粒子系统串行版本耗时分布比率图

其中，可并行部分包括计算顶点和重置顶点，比例总共达到70%；GPU渲染部分占30%必须串行无法参与并行优化，除非替换成VBO渲染方式，因为顶点运算在CPU中进行，数据必须在CPU和GPU之间频繁更新和传输，暂时不适合采用VBO渲染方式，后文解除顶点运算必须在CPU的限制后，将采用VBO提升这部分性能；异步等待或重叠指CPU与GPU计算时间的关系，本次测量时间时发现有11%时间重叠，即CPU和GPU同时运行。

### OpenCL并行计算

明确粒子系统瓶颈位置并且论证并行计算的可行性以后，从本文第二章得知，并行计算方案有两类，一类面向CPU，共有SSE和OpenCL两套方案；一类面向GPU，共有GLSL、CUDA、OpenCL三套方案。通过第四章实验结论可知，OpenCL是面向CPU的最优并行方案，OpenCL也是面向GPU的较优并行方案，考虑到OpenCL具有良好的可移植性，同一份源码既可以运行在CPU上也可以运行在GPU上，便于在CPU和GPU之间切换方案，所以本文采用OpenCL这一方案。

在进行并行计算之前，对原先串行版本算法进行改造，将直接索引寻址改为相对偏移寻址，并行将渲染模式改为VBO方式，初步串行优化主要目标是为并行化做预备，没有性能提升。接下来作并行优化，按照第二章和第三章介绍使用OpenCL的方法步骤配置运行环境并且移植算法源码。将以上计算顶点和重置顶点两个环节改造成OpenCL的kernel函数，包括其中调用的柏林噪声Perlin Noise生成过程。基于OpenCL面向CPU的粒子系统并行计算加速比如下图：

图5-7基于OpenCL的粒子系统加速比

从上图得出以下4个结论，1）对于10万粒子数目的喷泉，CPU和GPU的加速比分别是3.4和65；2）同样基于OpenCL，GPU性能却比CPU高出6~30倍；3）随着粒子数增多，GPU并行加速比持续增长，运算量越大，GPU越能发挥并行加速能力；4）随着粒子数增多，CPU并行加速比先增再降。

## CPU与GPU异构协同处理VR系统多任务

本文以上内容的主要研究目标都是单一的对象节点，对于单一对象节点，为了寻求最优并行方案，只要遍历CPU和GPU上的各种方案寻求最优解即可。对于一个系统来说，往往同时包含同一类型节点的多个对象节点，并且多个类型并存。以本文涉及的骨骼动画和粒子系统为例，一个VR系统中往往同时包含多个骨骼动画实例以及多个粒子系统实例，而且同属骨骼动画的节点复杂度也可能不同，本文也将节点复杂度称为粒度。这时寻求并行方案最优解，不再是简单寻求某一类型某一节点的最优解，而是综合考虑所有对象节点，将一部分节点配置到CPU上，将另一部分节点配置到GPU上，达到CPU和GPU负载均衡，使得系统整体并行计算性能最优化。根据类型和粒度的不同，本文接下来分以下4种情况探讨多任务并行问题。

### 相同粒度的多节点并行计算系统

以骨骼动画为例，进行基于OpenCL的并行计算之后，根据上文的图4-16得知，在CPU上加速比是3.9，在GPU上加速比是5.0，两者之间的比率是：0.44比0.56。如果有多个骨骼动画，按照44%与56%的比率分配给CPU和GPU。假设有N个节点，CPU和GPU分配节点的个数分别是N\*0.44和N\*0.56，数目按四舍五入取整。本文以2、4、8、16个节点为例进行实验，发现给CPU和GPU分别分配节点个数的比率符合0.44比0.56时，总体性能达到最优。

同理对于粒子系统，进行基于OpenCL的并行计算之后，根据上文的图5-3得知，设定粒子数为1万，在CPU上加速比是1.9，在GPU上加速比是13，两者之间的比率是：0.13比0.87。如果有多个粒子系统，按照13%与87%的比率分配给CPU和GPU。假设有N个节点，CPU和GPU分配节点的个数分别是N\*0.13和N\*0.87，数目按四舍五入取整。本文以10个节点为例进行实验，发现CPU和GPU分配节点个数的比率符合0.13比0.87即1比9时，总体性能最优。

### 不同粒度的多节点并行计算系统

粒度不同，有可能是类型不同，或者类型相同但复杂度不同，本文设定类型相同但复杂度不同的情况并且选定2种粒度。其中粒度一和粒度二的节点数目分别为M和N。对两种粒度分别采用上一节分配方案，得到最优方案。

对于骨骼动画，进行基于OpenCL的并行计算之后，根据上文的图4-16得知，1）对于顶点数目为10万的节点粒度情况，在CPU上加速比是3.9，在GPU上加速比是5.0；2）对于顶点数目为100万的节点粒度情况，在CPU上加速比是2.3，在GPU上加速比是3.1。如果有多个骨骼动画，假设粒度一和粒度二各10个，对于粒度一的10个，按0.44比0.56的比率分配给CPU和GPU各4和6个；对于粒度二的10个，按0.43比0.57的比率分配给CPU和GPU各4和6个。

对于粒子系统，进行基于OpenCL的并行计算之后，根据上文的图5-3得知，1）对于粒子数目为1万的节点粒度情况，在CPU上加速比是1.9，在GPU上加速比是13；2）对于粒子数目为10万的节点粒度情况，在CPU上加速比是3.4，在GPU上加速比是65。如果有多个粒子系统，假设粒度一和粒度二各10个，对于粒度一的10个，按0.13比0.87的比率分配给CPU和GPU各1和9个；对于粒度二的10个，按0.05比0.95的比率分配给CPU和GPU各0和10个。

## 本章小结

本章研究对象是虚拟现实又一关键模块——基于粒子系统的自然现象仿真模块，本文实现带有柏林噪声风场干扰的喷泉模拟。通过对算法进行耗时分布统计定位了瓶颈位置，进而采用二三章提到的并行计算技术加速算法。鉴于三章得到以下结论：无论是CPU还是GPU，OpenCL算法都是最优或次优方案，作为次优方案时性能不低于最优方案的7成；为了使算法的功能和性能都具有良好的跨平台可移植性，建议统一采用OpenCL方案，虽然OpenCL在GPU方案子集中有时不是最优。针对粒子系统本文采用OpenCL方案，分别运行在CPU和GPU这两种设备上，在CPU端获得2~3倍加速比，在GPU端获得10~90倍加速比，加速比随着粒子系统复杂度的增长而增长。

当把粒子系统和骨骼动画以及更多的模块结合在一起时，研究目标从单一模块演变为多模块多任务系统。对于单一对象节点，为了寻求最优并行方案，只要遍历CPU和GPU上的各种方案寻求最优解即可。对于一个系统来说，往往同时包含同一类型节点的多个对象节点，并且多个类型并存。以本文涉及的骨骼动画和粒子系统为例，一个VR系统中往往同时包含多个骨骼动画实例以及多个粒子系统实例，而且同属骨骼动画的节点复杂度也可能不同，本文也将节点复杂度称为粒度。这时寻求并行方案最优解，不再是简单寻求某一类型某一节点的最优解，而是综合考虑所有对象节点，将一部分节点配置到CPU上，将另一部分节点配置到GPU上，达到CPU和GPU负载均衡，使得系统整体并行计算性能最优化。本文为相同粒度和不同粒度的多节点并行计算探索了寻求最优系统性能的解决方案，即根据OpenCL在CPU和GPU上的加速比，为CPU和GPU分配同等比率的任务节点。

# 总结和展望

## 总结

本文研究了面向通用科学计算领域的多种前沿并行计算技术，包括面向CPU和GPU两种并行计算设备。其中，面向CPU的并行计算技术有传统的流式单指令多数据扩展指令集SSE（StreamingSIMDExtensions）和新兴的开放计算语言OpenCL (Open Computing Language)；面向GPU的并行计算技术历经OpenGL着色语言GLSL（OpenGL Shading Language）、统一设备架构CUDA（Compute Unified Device Architecture）、OpenCL这三代GPGPU技术。OpenCL因其同时适用于CPU和GPU从而成为它们之间很好的纽带。同时研究多种并行计算技术有以下几个目的：1）遍历各种并行计算方案寻找性能最优方案；2）当最优方案受硬件和软件限制无法使用时，次优方案成为候补方案；3）多任务系统需要多种并行计算方案并存并且支持方案灵活切换。

针对虚拟现实骨骼动画渲染算法这一模块，本文对比研究了各种并行计算技术。本文先后实现了骨骼动画渲染算法的五个并行版本，包括面向CPU的SSE版本和OpenCL版本，面向GPU的GLSL版本、CUDA版本、OpenCL版本。除了基于SSE面向CPU的版本以及基于GLSL面向GPU的版本这2个版本是沿用现有成熟技术，骨骼动画渲染算法其它多个版本的并行计算实现都由本文提出。本文对各种并行计算算法的性能做了统计和分析对比，最终得到各种并行计算方案的加速比，包括SSE、CPU端OpenCL、GPU端OpenCL、GLSL、CUDA共5套并行计算方案。通过对骨骼动画多套并行计算方案对比分析得出以下几个结论：1）GLSL在各种GPU方案中最优，OpenCL在各种CPU方案中最优，而且GLSL性能高于OpenCL；2）无论是CPU还是GPU，OpenCL算法都是最优或次优方案，作为次优方案时性能不低于最优方案的7成；3）为了使算法的功能和性能都具有良好的跨平台可移植性，建议统一采用OpenCL方案，虽然OpenCL在GPU方案子集中有时不是最优。

本文还研究了基于柏林噪声风场扰动的喷泉模拟，并对其进行了基于OpenCL的并行计算加速。当把粒子系统和骨骼动画以及更多的模块结合在一起时，研究目标从单一模块演变为多模块多任务系统。对于多任务系统研究人员需要综合考虑所有对象节点，将一部分节点配置到CPU上，将另一部分节点配置到GPU上，达到CPU和GPU负载均衡，使得系统整体并行计算性能最优化。本文为相同粒度和不同粒度的多节点并行计算探索了寻求最优系统性能的解决方案，即根据OpenCL在CPU和GPU上的加速比，为CPU和GPU分配同等比率的任务节点。

在探索将并行计算技术系统地引入虚拟现实引擎的研究过程中，本文创新点主要体现在以下三个方面：

1. 提出了基于CUDA的骨骼动画矩阵调色板GPGPU算法。性能创新：以CPU串行算法和基于SSE叠加OpenMP的传统并行算法作为性能参考基准，CUDA算法加速比分别是4.4和2.5。功能创新：现有骨骼动画矩阵调色板算法基于图形库OpenGL的扩展接口GLSL，渲染和计算紧密耦合在一起，不利于算法的移植复用。本文采用脱离图形库的CUDA技术，使渲染和计算有效隔离，在软件编程方面提升了矩阵调色板算法的可移植性。

2. 提出了基于OpenCL面向CPU和GPU异构体系的骨骼动画矩阵调色板算法。性能创新：面向CPU的OpenCL算法，以CPU串行算法和基于SSE叠加OpenMP的传统并行算法作为性能参考基准，加速比分别是3.9和2.2。功能创新：提升面向GPU骨骼动画矩阵调色板算法的可移植性，原先基于CUDA的算法分别依赖特定的GPU，基于OpenCL面向GPU的矩阵调色板算法普遍适用于各种GPU。

3. 设计了基于OpenCL面向CPU和GPU异构体系的柏林噪声风场扰动粒子系统喷泉。性能创新：以CPU串行算法作为性能参考基准，OpenCL在CPU和GPU上的加速比分别是3.4和65。

本文结合虚拟现实仿真平台的骨骼动画模块和粒子系统模块研究多种并行计算技术，当研发人员需要做以下决策时——是否将现有串行算法进行并行化、是否追随新的并行计算技术对已有并行算法进行移植升级、是否升级并行计算硬件设备，本文提供更为科学有效的决策依据。

## 展望

鉴于研究时间有限，无法同时兼顾研究深度和广度，论文存在以下三方面不足之处：

* 虽然本文初步从系统的角度研究了多任务异构协同并行计算，但任务数量还局限在少数环节

一个完整的虚拟现实系统内部将有许多个有待执行并行计算的潜在环节，本文研究工作告一段落后，将继续寻找这些环节，在对这些环节进行隔离单独研究后，合并到一起作为一个系统进一步探索系统并行计算方案。

* 本文还对Web并行计算作了前期探索，但没有完整的研究案例

将来计划继续深入研究WebCL这一类似OpenCL的通用并行计算技术。特别是虚拟现实三维图像领域，WebGL已经是相当成熟的技术，因其内置支持类似OpenGL Shading Language的OpenGL ES，可以实现基于Shading Language的Web并行计算。如果说WebGL因其依赖三维图像库，限制了它在通用领域的应用和发展，WebCL的出现突破了这种限制，WebCL完全支持通用并行计算，不再依赖于任何三维图形库，可以在网页中通过浏览器调用客户端设备的CPU或GPU多和设备，支持多媒体、科学计算等领域的高密集度计算任务的加速执行。到本文截稿为止，WebCL正式标准尚未发布，还处于前沿研究阶段，作为对本文工作的延伸和拓展，将来将虚拟现实三维仿真系统通过WebGL完整移植到Web，并将通用并行计算任务通过WebCL移植到Web，将逐步取代原先的客户端应用。

* 在系统应用方面，本文尚未将研究重点角色动画与工程应用需求对接形成专业的系统

论文告一段落后将来打算继续深入研究面向三维角色动画的行业子系统，专门探索三维动画的性能瓶颈解决方案。结合Microsoft新近推出的Kinect体感设备研究面向面部细节和全身轮廓的实时动作捕捉系统，高效地录制和编辑各种动作，可以广泛用于各种虚拟教学和虚拟实训系统，以及其他跟角色动画有关的领域。

在将来的研究工作中，希望继续在并行计算技术和虚拟现实应用等方面走得更深更广，在广阔的科研世界找到更为精准和专业的分支领域和坐标位置，与广大研究人员彼此交流相互学习共同进步。

参考文献

[1] Asanovic K, Bodik R, Demmel J, et al. A view of the parallel computing landscape. Communications of the ACM, 2009, 52(10): 56~67

[2] Asanovic K, Bodik R, Catanzaro BC, et al. The landscape of parallel computing research: A view from Berkeley.Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[3] Williams SW. Auto-tuning performance on multicore computers: [Ph.D Dissertation]. Berkeley: University of California, 2008

[4] Kurzak J, Bader DA, Dongarra J. Scientific Computing with Multicore and Accelerators. Florida: CRC Press, Inc., 2010.

[5] Sodan AC, Machina J, Deshmeh A, et al. Parallelism via multithreaded and multicore CPUs. Computer, 2010, 43(3): 24~32

[6] Brodtkorb AR, Dyken C, Hagen TR, et al. State-of-the-art in heterogeneous computing. Scientific Programming, 2010, 18(1): 1~33

[7] 陈国良, 孙广中, 徐云, 等. 并行计算的一体化研究现状与发展趋势. 科学通报, 2009, (8): 1043~1049

[8] Franchetti F, Kral S, Lorenz J, et al. Efficient utilization of SIMD extensions. Proceedings of the IEEE, 2005, 93(2): 409~425

[9] Hassaballah M, Omran S, Mahdy YB. A review of SIMD multimedia extensions and their usage in scientific and engineering applications. The Computer Journal, 2008, 51(6): 630~649

[10] Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, IEEE, 1998, 5(1): 46~55

[11] Kegel P, Schellmann M, Gorlatch S. Using openmp vs. threading building blocks for medical imaging on multi-cores. Euro-Par 2009 Parallel Processing.Springer, 2009. 654~665

[12] Strey A, Bange M. Performance Analysis of Intel’s MMX and SSE: A case Study. Euro-Par 2001 Parallel Processing.Springer, 2001. 142~147

[13] Aberdeen D, Baxter J. Emmerald: a fast matrix–matrix multiply using Intel's SSE instructions. Concurrency and Computation: Practice and Experience, 2001, 13(2): 103~119

[14] Vaško A, Šrámek M. Optimizing Gaussian filtering of volumetric data using SSE. Concurrency and Computation: Practice and Experience, 2011, 23(1): 100~116

[15] Larsson P, Palmer E. Image Processing Acceleration Techniques using Intel Streaming SIMD Extensions and Intel Advanced Vector Extensions. Intel. Corp. Whitepaper, 2009:

[16] Intel. Introduction to intel advanced vector extensions 2012. http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions

[17] Intel. Advanced vector extensions programming reference 2011. https://secure-software.intel.com/sites/default/files/m/d/4/1/d/8/319433-011.pdf

[18] Chafi H, Sujeeth AK, Brown KJ, et al. A domain-specific approach to heterogeneous parallelism.Proceedings of the 16th ACM symposium on Principles and practice of parallel programming.ACM, 2011. 35~46

[19] Sujeeth AK, Rompf T, Brown KJ, et al. Composition and reuse with compiled domain-specific languages. Proceedings of ECOOP. 2013.

[20] Brown KJ, Sujeeth AK, Lee HJ, et al. A heterogeneous parallel framework for domain-specific languages.2011. 89~100

[21] Chen T, Raghavan R, Dale JN, et al. Cell broadband engine architecture and its first implementation—a performance view. IBM Journal of Research and Development, 2007, 51(5): 559~572

[22] Gschwind M. The Cell Broadband Engine: exploiting multiple levels of parallelism in a chip multiprocessor. International Journal of Parallel Programming, 2007, 35(3): 233~262

[23] Bader DA, Patel S. High performance MPEG-2 software decoder on the cell broadband engine.2008. 1~10

[24] Agarwal V, Liu L, Bader DA. Financial modeling on the cell broadband engine.2008. 1~12

[25] Shirako J, Kasahara H, Sarkar V. Language extensions in support of compiler parallelization. Languages and Compilers for Parallel Computing.Springer, 2008. 78~94

[26] Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. Proceedings of the April 18-20, 1967, Spring Joint Computer Conference.ACM, 1967. 483~485

[27] Gustafson JL, Montry GR, Benner RE. Development of parallel methods for a 1024-processor hypercube. SIAM journal on Scientific and Statistical Computing, 1988, 9(4): 609~638

[28] Gustafson JL. Reevaluating Amdahl's law. Communications of the ACM, 1988, 31(5): 532~533

[29] Lee VW, Kim C, Chhugani J, et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. ACM SIGARCH Computer Architecture News.ACM, 2010. 451~460

[30] Trinitis C. Is GPU enthusiasm vanishing?.High Performance Computing and Simulation (HPCS), 2012 International Conference.IEEE, 2012. 410

[31] 林一松, 杨学军, 唐滔, 等. 一种基于关键路径分析的 CPU-GPU 异构系统综合能耗优化方法. 计算机学报, 2012, 35(1): 123~133

[32] Kalra P, Magnenat-Thalmann N, Moccozet L, et al. Real-time animation of realistic virtual humans. Computer Graphics and Applications, IEEE, 1998, 18(5): 42~56

[33] 尚华强. 基于Kinect的虚拟人物动作仿真研究: [硕士学位论文].杭州电子科技大学, 2013

[34] Reynolds CW. Flocks, herds and schools: A distributed behavioral model. ACM SIGGRAPH Computer Graphics.ACM, 1987. 25~34

[35] Müller M, Charypar D, Gross M. Particle-based fluid simulation for interactive applications. Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation.Eurographics Association, 2003. 154~159

[36] Kolb A, Latta L, Rezk-Salama C. Hardware-based simulation and collision detection for large particle systems. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.ACM, 2004. 123~131

[37] Kipfer P, Segal M, Westermann R. UberFlow: a GPU-based particle engine. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.ACM, 2004. 115~122

[38] Kruger J, Kipfer P, Konclratieva P, et al. A particle system for interactive visualization of 3D flows. IEEE Transactions on Visualization and Computer Graphics, 2005, 11(6): 744~756

[39] Purcell TJ, Buck I, Mark WR, et al. Ray tracing on programmable graphics hardware. Proceedings of the 29th annual conference on Computer graphics and interactive techniques.San Antonio, Texas: ACM, 2002. 703~712

[40] Parker SG, Bigler J, Dietrich A, et al. Optix: a general purpose ray tracing engine. ACM Transactions on Graphics (TOG), 2010, 29(4): 66

[41] Huff R, Gierlinger T, Kuijper A, et al. A comparison of xpu platforms exemplified with ray tracing algorithms. Virtual Reality (SVR), 2011 XIII Symposium on.IEEE, 2011. 1~8

[42] Martz P. OpenSceneGraph Quick Start Guide: A Quick Introduction to the Cross-Platform Open Source Scene Graph API. Skew Matrix Software, 2007.

[43] Li F, Sun J, Yang Q. Design and research of virtual assembly system based on OSG. Digital Manufacturing and Automation (ICDMA), 2011 Second International Conference on.IEEE, 2011. 385~388

[44] 孟效轲, 华泽玺, 何春. 基于 OSG 的爆炸装置拆除视景仿真及关键技术. 计算机仿真, 2010, (007): 234~238

[45] Junker G. Pro OGRE 3D programming. Apress, 2006.

[46] Mark WR, Glanville RS, Akeley K, et al. Cg: a system for programming graphics hardware in a C-like language. 2003, 22(3): 896~907

[47] Fernando R, Kilgard MJ. The Cg Tutorial: The definitive guide to programmable real-time graphics. Addison-Wesley Longman Publishing Co., Inc., 2003.

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

[49] Lewis JP, Cordner M, Fong N. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation.2000. 165~172

[50] Lindholm E, Kilgard MJ, Moreton H. A user-programmable vertex engine.Proceedings of the 28th annual conference on Computer graphics and interactive techniques.ACM, 2001. 149~158

[51] James DL, Twigg CD. Skinning mesh animations.ACM SIGGRAPH 2005 Papers. Los Angeles, California: ACM, 2005. 399~407

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

[53] Ju T, Zhou Q, van de Panne M, et al. Reusable skinning templates using cage-based deformations.2008.

[54] Raptis M, Kirovski D, Hoppe H. Real-time classification of dance gestures from skeleton animation. Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation. 2011. 147~156

[55] Reeves WT. Particle systems—a technique for modeling a class of fuzzy objects. ACM SIGGRAPH Computer Graphics.ACM, 1983. 359~375

[56] Reeves WT, Blau R. Approximate and probabilistic algorithms for shading and rendering structured particle systems. ACM Siggraph Computer Graphics.ACM, 1985. 313~322

[57] Khronos. OpenGL Shading Language Specification(GLSL SPEC) v4.4. 2013. http://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf

[58] Buck I, Foley T, Horn D, et al. Brook for GPUs: stream computing on graphics hardware. ACM SIGGRAPH 2004 Papers. Los Angeles, California: ACM, 2004. 777~786

[59] Owens JD, Luebke D, Govindaraju N, et al. A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum, 2007, 26(1): 80~113

[60] Owens JD, Houston M, Luebke D, et al. GPU Computing. Proceedings of the IEEE, 2008, 96(5): 879~899

[61] Nickolls J, Buck I, Garland M, et al. Scalable Parallel Programming with CUDA. Queue, 2008, 6(2): 40~53

[62] Lindholm E, Nickolls J, Oberman S, et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture. Micro, IEEE, 2008, 28(2): 39~55

[63] Che S, Boyer M, Meng J, et al. A performance study of general-purpose applications on graphics processors using CUDA. Journal of parallel and distributed computing, 2008, 68(10): 1370~1380

[64] Ryoo S, Rodrigues CI, Baghsorkhi SS, et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. Salt Lake City, UT, USA: ACM, 2008. 73~82

[65] Garland M, Le Grand S, Nickolls J, et al. Parallel Computing Experiences with CUDA. Micro, IEEE, 2008, 28(4): 13~27

[66] Jianbin F, Varbanescu AL, Sips H. A Comprehensive Performance Comparison of CUDA and OpenCL. Parallel Processing (ICPP), 2011 International Conference on. Taipei City: 2011. 216~225

[67] Jie S, Jianbin F, Sips H, et al. Performance Gaps between OpenMP and OpenCL for Multi-core CPUs. Parallel Processing Workshops (ICPPW), 2012 41st International Conference on. Pittsburgh, PA: 2012. 116~125

[68] Lee S, Min S, Eigenmann R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. Raleigh, NC, USA: ACM, 2009. 101~110

[69] Hormati AH, Samadi M, Woh M, et al. Sponge: portable stream programming on graphics engines. Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems. Newport Beach, California, USA: ACM, 2011. 381~392

[70] Moreland K, Angel E. The FFT on a GPU.Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. San Diego, California: Eurographics Association, 2003. 112~119

[71] Fatahalian K, Sugerman J, Hanrahan P. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication.2004. 133~137

[72] Nukada A, Matsuoka S. Auto-tuning 3-D FFT library for CUDA GPUs. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.ACM, 2009. 30

[73] Govindaraju NK, Manocha D. Cache-efficient numerical algorithms using graphics hardware. Parallel Computing, 2007, 33(10): 663~684

[74] Jiang C, Snir M. Automatic tuning matrix multiplication performance on graphics hardware. Parallel Architectures and Compilation Techniques, 2005.PACT 2005.14th International Conference on.IEEE, 2005. 185~194

[75] 吴恩华, 柳有权. 基于图形处理器 (GPU) 的通用计算. 计算机辅助设计与图形学学报, 2004, 16(5): 601~612

[76] 吴恩华. 图形处理器用于通用计算的技术, 现状及其挑战. 软件学报, 2004, 15(10): 1493~1504

[77] Lefohn AE, Sengupta S, Kniss J, et al. Glift: Generic, efficient, random-access GPU data structures. ACM Transactions on Graphics (TOG), 2006, 25(1): 60~99

[78] Harish P, Narayanan PJ. Accelerating large graph algorithms on the GPU using CUDA. High performance computing--HiPC 2007. Springer, 2007. 197~208

[79] Blythe D. Rise of the graphics processor. Proceedings of the IEEE, 2008, 96(5): 761~778

[80] Christophe E, Michel J, Inglada J. Remote sensing processing: From multicore to GPU. Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of, 2011, 4(3): 643~652

[81] De Caro D, Petra N, Strollo A. High-performance special function unit for programmable 3-D graphics processors. Circuits and Systems I: Regular Papers, IEEE Transactions on, 2009, 56(9): 1968~1978

[82] Diao M, Kim J. Multimedia mining on manycore architectures: The case for gpus. Advances in Visual Computing.Springer, 2009. 619~630

[83] Cederman D, Chatterjee B, Tsigas P. Understanding the performance of concurrent data structures on graphics processors. Euro-Par 2012 Parallel Processing.Springer, 2012. 883~894

[84] Teschner M, Kimmerle S, Heidelberger B, et al. Collision detection for deformable objects.2005. 61~81

[85] Hodgins JK. Capturing and Animating Skin Deformation in Human Motion. ACM transactions on graphics, 2006, (3): 881~889

[86] Yang X, Somasekharan A, Zhang JJ. Curve skeleton skinning for human and creature characters. Computer Animation and Virtual Worlds, 2006, 17(3-4): 281~292

[87] Jacka D, Reid A, Merry B, et al.A comparison of linear skinning techniques for character animation.2007. 177~186

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

[89] Elsen E, Houston M, Vishal V, et al. N-Body simulation on GPUs. Proceedings of the 2006 ACM/IEEE conference on Supercomputing. Tampa, Florida: ACM, 2006. 188

[90] Nyland L, Harris M, Prins J. Fast n-body simulation with cuda. GPU gems, 2007, 3: 677~695

[91] Ma A, Cai J, Cheng Y, et al. Performance Optimization Strategies of High Performance Computing on GPU. Advanced Parallel Processing Technologies. Springer, 2009. 150~164

[92] 马安国, 成玉, 唐遇星, 等. GPU 异构系统中的存储层次和负载均衡策略研究. 国防科技大学学报, 2009, 31(5): 38~43

[93] 卢风顺, 宋君强, 银福康, 等. CPU/GPU 协同并行计算研究综述. 计算机科学, 2011, 38(3): 5~9

[94] Harris M. Mapping computational concepts to GPUs.ACM SIGGRAPH 2005 Courses.ACM, 2005. 50

[95] Muyan-Ozcelik P, Owens JD, Xia J, et al. Fast deformable registration on the GPU: A CUDA implementation of demons. Computational Sciences and Its Applications, 2008.ICCSA'08.International Conference on.IEEE, 2008. 223~233

[96] Zhang X, Kim YJ. Interactive collision detection for deformable models using streaming AABBs. Visualization and Computer Graphics, IEEE Transactions on, 2007, 13(2): 318~329

[97] Amorim R, Haase G, Liebmann M, et al. Comparing CUDA and OpenGL implementations for a Jacobi iteration. High Performance Computing & Simulation, 2009.HPCS'09.International Conference on.IEEE, 2009. 22~32

[98] Joselli M, Clua E, Montenegro A, et al. A new physics engine with automatic process distribution between cpu-gpu.Proceedings of the 2008 ACM SIGGRAPH symposium on Video games.ACM, 2008. 149~156

[99] Rødal KES, Storli G. Physically based simulation and visualization of fire in real-time using the gpu: Norwegian University of Science and Technology, 2006

[100] Seshadrinathan M, Dempski KL.Implementation of advanced encryption standard for encryption and decryption of images and text on a gpu.Computer Vision and Pattern Recognition Workshops, 2008.CVPRW'08.IEEE Computer Society Conference on.IEEE, 2008. 1~6

[101] 韩俊刚, 蒋林, 杜慧敏, 等. 一种图形加速器和着色器的体系结构. 计算机辅助设计与图形学学报, 2010, 23(3): 363~372

[102] Ohmer JF. Computer vision applications on graphics processing units: [Master Thesis]. Queensland: Queensland University of Technology, 2007

[103] Kumar R, Tullsen DM, Jouppi NP, et al. Heterogeneous chip multiprocessors. Computer, 2005, 38(11): 32~38

[104] Benner RE, Gustafson JL, Montry GR. Development and analysis of scientific application programs on a 1024-processor hypercube. SAND 88-0317, Sandia National Laboratories, 1988:

[105] 杨际祥, 谭国真, 王荣生. 并行与分布式计算动态负载均衡策略综述. 电子学报, 2010, 38(5): 1122~1130

[106] Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM, 2009, 52(4): 65~76

[107] Guz Z, Bolotin E, Keidar I, et al. Many-core vs. many-thread machines: Stay away from the valley. Computer Architecture Letters, 2009, 8(1): 25~28

[108] Yao E, Bao Y, Tan G, et al. Extending Amdahl's law in the multicore era. ACM SIGMETRICS Performance Evaluation Review, 2009, 37(2): 24~26

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

[110] Daga M, Aji AM, Feng W. On the efficacy of a fused cpu+ gpu processor (or apu) for parallel computing. Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on. IEEE, 2011. 141~149

[111] Joao JA, Suleman MA, Mutlu O, et al. Bottleneck identification and scheduling in multithreaded applications. ACM SIGARCH Computer Architecture News, 2012, 40(1): 223~234

[112] Williams S, Datta K, Carter J, et al. PERI-Auto-tuning memory-intensive kernels for multicore. Journal of Physics: Conference Series. IOP Publishing, 2008. 1~15

[113] Rul S, Vandierendonck H, D'Haene J, et al.An experimental study on performance portability of OpenCL kernels.2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10). 2010.

[114] Bailey DH, Lucas RF, Williams SW. Performance tuning of scientific applications.CRC Press, 2011.

[115] Scott LE, David M. Fast matrix multiplies using graphics hardware. Proceedings of Supercomputing, Denver, 2001, 60: 55

[116] Govindaraiu Naga K, Scott L, Jim G, et al.A memory model for scientific algorithms on graphics processors. Supereomputing, Proceedings Of The ACM. Tampa, Florida: 2006. 6

[117] Ryoo S, Rodrigues CI, Stone SS, et al. Program optimization space pruning for a multithreaded gpu.2008. 195~204

[118] Nukada A, Ogata Y, Endo T, et al. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA.2008. 1~11

[119] Baskaran MM, Ramanujam J, Sadayappan P. Automatic C-to-CUDA code generation for affine programs.2010. 244~263

[120] Terriberry TB, French LM, Helmsen J. GPU accelerating speeded-up robust features.2008. 355~362

[121] Endo T, Matsuoka S. Massive supercomputing coping with heterogeneity of modern accelerators.2008. 1~10

[122] Eladio G, Sergio R, Maria AT, et al. Memory Locality Exploitation Strategies for FFT on the CUDA Architecture. Lecture Notes in Computer Science, High Performance Computing for Computational Science-VECPAR 2008, Springer Berlin/Heidelberg, 2008, 5336: 16

[123] Govindaraju NK, Lloyd B, Wang W, et al. Fast computation of database operations using graphics processors.2004. 215~226

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

[125] Tomov S, McGuigan M, Bennett R, et al. Benchmarking and implementation of probability-based simulations on programmable graphics cards. Computers \& Graphics, 2005, 29(1): 71~80

[126] Trancoso P, Charalambous M. Exploring graphics processor performance for general purpose applications.2005. 306~313

[127] Bodin F, Bihan S. Heterogeneous multicore parallel programming for graphics processing units. Scientific Programming, 2009, 17(4): 325~336

[128] Buck I, Hanrahan P. Data parallel computation on graphics hardware. Graphics Hardware 2003: Panel Presentation, 2003:

[129] Zeller C, Fernando R, Wloka M, et al. Programming graphics hardware. Proc. Eurographics—Tutorials, Sept, 2004: 1~17

[130] Lahabar S, Agrawal P, Narayanan PJ.High performance pattern recognition on GPU. Proceedings of NCVPRIPG, 2008, 2008: 154~159

[131] Bolz J, Farmer I, Grinspun E, et al.The GPU as numerical simulation engine. ACM SIGGRAPH, 2003: 1~9

[132] Brodtkorb AER.The graphics processor as a mathematical coprocessor in MATLAB.2008. 822~827

[133] NVIDIA.NVIDIA Visual Profiler v5.0. 2012. https://developer.nvidia.com/nvidia-visual-profiler

[134] NVIDIA.Nsight Visual Studio Edition v3.1. 2012. https://developer.nvidia.com/nvidia-nsight-visual-studio-edition

[135] NVIDIA.CUDA C Programming Guide v5.5. 2013. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[136] NVIDIA.CUDA C Best Practices Guide v5.5. 2013. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[137] NVIDIA.Cg Users Manual v1.4. 2005.

[138] NVIDIA.Cg Reference Manual v3.1. 2012.

[139] Khronos.The OpenCL Specification v1.2. 2011. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[140] Khronos.The OpenCL Reference v1.2. 2011. http://www.khronos.org/registry/cl/sdk/1.2/docs/OpenCL-1.2-refcard.pdf

[141] Khronos. ARB fragment program 2002. http://www.opengl.org/registry/specs/ARB/fragment\_program.txt

[142] Khronos.ARB vertex program 2002. http://www.opengl.org/registry/specs/ARB/vertex\_program.txt

[143] Microsoft.Programming Guide for HLSL. 2013. http://msdn.microsoft.com/en-us/library/bb509635(v=vs.85).aspx

[144] Microsoft.Reference for HLSL. 2013. http://msdn.microsoft.com/en-us/library/bb509638(v=vs.85).aspx

[145] AMD. Accelerated Parallel Processing (APP) SDK v2.8. 2013. http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/

[146] Intel. The Intel SDK for OpenCL Applications v2013. 2013. http://software.intel.com/en-us/vcsource/tools/opencl-sdk

[147] Pheatt C. Intel threading building blocks. Journal of Computing Sciences in Colleges, 2008, 23(4): 298

[148] Reinders J. Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc., 2010.

[149] Unat D. Domain-specific translator and optimizer for massive on-chip parallelism: [Ph.D Dissertation]. San Diego, California: University of California, 2012

[150] 吕晖, 李宏亮, 郑方, 等. 高性能计算领域向量技术应用分析. 高性能计算技术, 2007, (003): 1~5

[151] Fritz N. SIMD Code Generation in Data-Parallel Programming.epubli, 2009.

[152] Chiu J, Chou Y.A multi-streaming SIMD multimedia computing engine. Microprocessors and Microsystems, 2010, 34(7): 247~258

[153] Peleg A, Weiser U. MMX technology extension to the Intel architecture. Micro, IEEE, 1996, 16(4): 42~50

[154] Peleg A, Wilkie S, Weiser U. Intel MMX for multimedia PCs. Communications of the ACM, 1997, 40(1): 24~38

[155] Leupers R. Code selection for media processors with SIMD instructions.Proceedings of the conference on Design, automation and test in Europe.ACM, 2000. 4~8

[156] Shahbahrami A, Juurlink B, Vassiliadis S. Performance impact of misaligned accesses in SIMD extensions.Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2006). 2006. 334~342

[157] Shahbahrami A, Juurlink B, Vassiliadis S. SIMD vectorization of histogram functions.Application-specific Systems, Architectures and Processors, 2007.ASAP.IEEE International Conf. on.IEEE, 2007. 174~179

[158] Lee RB.Multimedia extensions for general-purpose processors.Signal Processing Systems, 1997. SIPS 97-Design and Implementation., 1997 IEEE Workshop on. IEEE, 1997. 9~23

[159] Barik R, Zhao J, Sarkar V. Efficient selection of vector instructions using dynamic programming.Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on.IEEE, 2010. 201~212

[160] Lorenz JUR. Automatic SIMD vectorization: Ph. D. Thesis, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2004

[161] Shi X, Zhou K, Tong Y, et al. Example-based dynamic skinning in real time.ACM Transactions on Graphics (TOG).ACM, 2008. 29

[162] 钟庆. 基于 CUDA 并行计算的三维形状变形编辑: [硕士学位论文]. 大连: 大连理工大学, 2012

[163] Pennycook SJ, Hammond SD, Wright SA, et al.An investigation of the performance portability of OpenCL. Journal of Parallel and Distributed Computing, 2012: 1~12

[164] Weber R, Gothandaraman A, Hinde RJ, et al. Comparing hardware accelerators in scientific applications: A case study. Parallel and Distributed Systems, IEEE Transactions on, 2011, 22(1): 58~68

[165] Park H, Han J. Fast rendering of large crowds using GPU.Entertainment Computing-ICEC 2008.Springer, 2009. 197~202

[166] Ivanovska T, Linsen L, Hahn HK, et al. GPU implementations of a relaxation scheme for image partitioning: GLSL versus CUDA. Computing and visualization in science, 2011, 14(5): 217~226

[167] Gomes T, Estevao L, de Toledo R, et al.A Survey of GLSL Examples.Graphics, Patterns and Images Tutorials (SIBGRAPI-T), 2012 25th SIBGRAPI Conference on.IEEE, 2012. 60~73

[168] Marroquim R, Maximo A. Introduction to GPU Programming with GLSL. Computer Graphics and Image Processing (SIBGRAPI TUTORIALS), 2009 Tutorials of the XXII Brazilian Symposium on. IEEE, 2009. 3~16

[169] Jacob F. CUDACL+: a framework for GPU programs. Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM, 2011. 55~58

[170] Oliveira RS, Rocha BM, Amorim RMCC, et al. Comparing cuda, opencl and opengl implementations of the cardiac monodomain equations. Parallel Processing and Applied Mathematics. Springer, 2012. 111~120

[171] Zhang Y, Sinclair II M, Chien AA.Improving Performance Portability in OpenCL Programs.2013. 136~150

[172] 卢贺齐. 基于 OpenCL 的实时 KD-Tree 与动态场景光线跟踪: [硕士学位论文]. 杭州: 浙江大学, 2011

[173] 黄鑫. CUDA 光线跟踪渲染器设计与实现: [硕士学位论文]. 北京: 北京邮电大学, 2012

[174] Rudy G. CUDA-CHiLL: A programming language interface for GPGPU optimizations and code generation: [Ph.D Dissertation]. Utah: The University of Utah, 2010

[175] Huang B, Plaza AJ.High-performance computing in remote sensing.Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series. 2011.

[176] 黄强强. 基于 GPU 的粒子系统数值模拟研究及其应用: [硕士学位论文]. 南昌: 南昌大学, 2012

[177] Leißa R, Hack S, Wald I. Extending a C-like language for portable SIMD programming. ACM SIGPLAN Notices, 2012, 47(8): 65~74

[178] McFarlin DS, Arbatov V, Franchetti F, et al. Automatic SIMD vectorization of fast fourier transforms for the larrabee and AVX instruction sets. Proceedings of the international conference on Supercomputing.ACM, 2011. 265~274

[179] Kofsky SM, Johnson DR, Stratton JA, et al. Implementing a GPU programming model on a Non-GPU accelerator architecture. Computer Architecture.Springer, 2012. 40~51

[180] 李建江, 路川, 张磊. 基于指导语句的 CUDA 程序性能分析工具研究与实现. 电子科技大学学报, 2012, 41(2): 280~284

[181] 王恒, 高建瓴. 基于 GPU 的 MATLAB 计算与仿真研究. 贵州大学学报 (自然科学版), 2012, 6: 23

[182] Mayanglambam S, Malony AD, Sottile MJ.Performance measurement of applications with gpu acceleration using cuda. Parallel Computing: From Multicores and GPU's to Petascale, 2010, 19: 341

[183] Malony AD, Biersdorff S, Spear W, et al.An experimental approach to performance measurement of heterogeneous parallel applications using cuda.Proceedings of the 24th ACM International Conference on Supercomputing.ACM, 2010. 127~136

[184] Farber R. CUDA application design and development.Access Online via Elsevier, 2011.

[185] Sim J, Dasgupta A, Kim H, et al.A performance analysis framework for identifying potential benefits in GPGPU applications. ACM SIGPLAN Notices. ACM, 2012. 11~22

[186] Torres Y, Gonzalez-Escribano A, Llanos DR. Understanding the impact of CUDA tuning techniques for Fermi.High Performance Computing and Simulation (HPCS), 2011 International Conference on.IEEE, 2011. 631~639

[187] KIM Y, SHRIVASTAVA A. Memory Performance Estimation of CUDA Programs.:

[188] Yablonski D. Numerical accuracy differences in CPU and GPGPU codes: [Master Thesis]. Boston: Northeastern University, 2011

[189] 乔香珍. 并行计算时间模型和并行机系统性能. 计算机学报, 1998, 21(5): 413~418

# 研究成果

个人简历

1985年7月23日出生于江西省赣州市宁都县。

2003年9月考入中国海洋大学信息科学与工程学院电子信箱工程专业，2007年7月本科毕业并获得工学学士学位。

2007年9月考入中国海洋大学信息科学与工程学院信号与信息处理专业，2009年7月硕博连读至今，于计算机应用技术专业攻读博士学位。

论文成果

1. 黄宝香, 韩勇, 刘寿生, 李文庆, 陈戈.多维动态噪声信息的可视化研究.中国海洋大学学报(自然科学版). 2011 , 41(1/2):181~184
2. 边少君, 陈戈, 刘寿生, 马纯永, 王华江.海底隧道三维仿真监控系统的研究与实现. 计算机工程与设计. 2012, 33(12): 4667~4671
3. 李文, 马纯永, 刘寿生, 张涛, 陈戈.虚拟城市仿真平台日照分析方法的研究和实现. 计算机技术与发展.2010, 20(11): 189~192

软件著作权/专利

1. 陈戈, 马纯永, 李文, 刘寿生, 陈勇. 一种基于金字塔模型的海量地形绘制网格裂缝消除算法. 中国, G06T 17/50, 专利号: 2008102020076, 2010
2. 世界园艺博览会三维虚拟仿真展示系统. 计算机软件著作权, 登记号:2013R11L185576, 2013
3. 基于数据手套的多通道环幕立体投影系统人机交互软件. 计算机软件著作权, 登记号:2012SR057146, 2012
4. 基于VR-GIS的城市噪声可视化评价系统. 计算机软件著作权,登记号: 2011SR030443, 2011
5. 面向VR-GIS平台的3DSMAX模型导出插件系统. 计算机软件著作权, 登记号:2010SR027931, 2010
6. 城市级VR-GIS一体化虚拟现实驱动系统软件V2.0. 计算机软件著作权, 登记号:2009SR019434, 2009
7. VR-GIS城域景观仿真平台系统 V1.0. 计算机软件著作权, 登记号:2007SR03397, 2007

参与项目

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 名称 | 来源 | 基金号 |
| 1 | 基于VR-GIS的城市噪声检测数据三维分析模型及可视化评价研究 | 国家自然科学基金 | 40971207 |
| 2 | 基于CUDA并行架构的高逼真度VRGIS加速引擎 | 国家科技部  科技人员服务企业行动基金 | SQ2009GJA0002426 |
| 3 | 2014世界园艺博览会三维虚拟仿真系统 | 青岛 | 0656-1241C10000316 |
| 4 | 2011世界园艺博览会三维虚拟仿真系统 | 西安 |  |
| 5 | 空间应用系统仿真设计VR支持平台 | 中科院 |  |
| 6 | 数字化消防演练培训系统 | 中石化安工院 |  |
| 7 | 胶州湾海底隧道3D监控系统 | 青岛海信 |  |

# 致谢

首先我要感谢辅导我完成论文的两位指导老师——陈戈教授和马纯永博士。感谢陈戈教授在研究生期间为我和其他同学搭建了一个训练科研素质和实践动手能力的软硬件平台。本文实验过程中陈老师还专门多方筹集甚至购置新的实验设备，并且在整个过程中深切关注实验和论文的每个环节，适时帮我把握研究的方向，平衡深度和广度。在我毕业论文的整个撰写过程中，马纯永博士给我提供了很多技术上的帮助和精神上的鼓励，感谢他在研究生期间的各个项目实践中的引导，论文写作期间给我多次提出宝贵的修改意见，使得论文和实验数据结果更为清晰。如果没有以上两位老师在本文研究过程中的无私奉献和耐心引导，论文和实验不能取得如今的进展。

在毕业论文实验及论文撰写过程中，我还得到了多位师长和同学的帮助，在此一并表示感谢。感谢张淑军师姐、邵宝民师兄、韩勇教授、田丰林博士等师长，感谢周圣川、任子健等同学。

最感激的是远在家乡的父亲母亲和妹妹，他们无私的关爱和一如既往的期望，使我的心保持温暖和火热，在项目实践、研究实验以及论文撰写过程中，不断从家人那里得到勇气和力量，克服各种困难和挫折。感谢在青岛求学期间认识的所有弟兄姐妹，因为大家的陪伴让我对这个地方倍感亲切，我会将每一位铭记于心。