登录 | 注册

网络资源是无限的

个人资料

fengbingchun

访问： 2252593次
积分： 25003
等级： BLOG 7
排名： 第202名

原创： 341篇 转载： 144篇
译文： 0篇 评论： 1434条

CSDN学院招募微信小程序讲师啦 程序员简历优化指南！ 【观点】移动原生App开发 PK HTML 5开发 云端应用征文大赛，秀绝招，赢无人机！

CUDA基础介绍

2017-01-23 08:58 13人阅读 评论(0) 收藏 举报

分类： CUDA (10)

版权声明：本文为博主原创文章，未经博主允许不得转载。

一、GPU简介

1985年8月20日ATi公司成立，同年10月ATi使用ASIC技术开发出了第一款图形芯片和图形卡，1992年4月ATi发布了Mach32图形卡集成了图形加速功能，1998年4月ATi被IDC评选为图形芯片工业的市场领导者，但那时候这种芯片还没有GPU的称号，很长的一段时间ATi都是把图形处理器称为VPU，直到AMD收购ATi之后其图形芯片才正式采用GPU的名字。

NVIDIA公司在1999年发布GeForce 256图形处理芯片时首先提出GPU的概念。GPU使显卡削减了对CPU的依赖，并实现部分原本CPU的工作，尤其是在3D图形处理时。GPU所采用的核心技术有硬体T&L(Transform and Lighting，多边形转换和光源处理)、立方环境材质贴图与顶点混合、纹理压缩及凹凸映射贴图、双重纹理四像素256位渲染引擎等，而硬体T&L技术能够说是GPU的标志。

GPU(Graphics Processing Unit)即图形处理器，又称显示核心、视觉处理器、显示芯片，是一种专门在个人电脑、工作站、游戏机和一些移动设备(如平板电脑、智能手机等)上作图像运算工作的微处理器。

显卡作为电脑主机里的一个重要组成部分，承担输出显示图形的任务。显卡的处理器称为图形处理器(GPU)，它是显卡的"心脏"，与CPU类似，只不过GPU是专为执行复杂的数学和几何计算而设计的，这些计算是图形渲染所必需的。

时下的GPU多数拥有2D或3D图形加速功能。有了GPU，CPU就从图形处理的任务中解放出来，可以执行其他更多的系统任务，这样可以大大提高计算机的整体性能。

GPU会产生大量热量，所以它的上方通常安装有散热器或风扇。

GPU是显示卡的"大脑"，GPU决定了该显卡的档次和大部分性能，同时GPU也是2D显示卡和3D显示卡的区别依据。2D显示芯片在处理3D图像与特效时主要依赖CPU的处理能力，称为软加速。3D显示芯片是把三维图像和特效处理功能集中在显示芯片内，也就是所谓的"硬件加速"功能。显示芯片一般是显示卡上最大的芯片(也是引脚最多的)。时下市场上的显卡大多采用NVIDIA和 AMD-ATI 两家公司的图形处理芯片。

GPU已经不再局限于3D图形处理了，GPU通用计算技术发展已经引起业界不少的关注，在浮点运算、并行计算等部分计算方面，GPU可以提供数十倍乃至于上百倍于CPU的性能。

GPU通用计算方面的标准目前有OpenCL、CUDA、AMD APP、DirectCompute。

二、GPU通用计算编程

对GPU通用计算进行深入研究从2003年开始，并提出了GPGPU概念，前一个GP则表示通用目的(General Purpose)，所以GPGPU一般也被称为通用图形处理器或通用GPU。

GPU通用计算通常采用**CPU+GPU**异构模式，由CPU负责执行复杂逻辑处理和事务处理等不适合数据并行的计

HTML (3)
Image Recognition (8)
Image Processing (18)
Image Registration (13)
ImageMagick (3)
Java (5)
Linux (20)
Log (2)
Makefile (2)
Mathematical Knowledge (6)
Multi-thread (4)
Matlab (33)
MFC (8)
MinGW (3)
Mac (1)
Neural Network (13)
OCR (9)
Office (2)
OpenCL (2)
OpenSSL (7)
OpenCV (86)
OpenGL (2)
OpenGL ES (3)
OpenMP (3)
Photoshop (1)
Python (4)
Qt (1)
SIMD (14)
Software Development (4)
System architecture (2)
Skia (1)
SVN (1)
Software Testing (4)
Shell (2)
Socket (3)
Target Detection (2)
Target Tracking (2)
VC6 (6)
VS2008 (16)
VS2010 (4)
VS2013 (3)
vigra (2)
VLC (5)
VLFeat (1)
wxWidgets (1)
Watermark (4)
Windows7 (6)
Windows Core Programming (9)
XML (2)

Free Codes

pudn
freecode
Peter's Functions
CodeProject
SourceCodeOnline
Computer Vision Source Code
Codesoso
Digital Watermarking
SourceForge
HackChina
oschina

算，由GPU负责计算密集型的大规模数据并行计算。

OpenCL(Open Computing Language，开放运算语言)是第一个面向异构系统通用目的并行编程的开放式、免费标准，也是一个统一的编程环境，便于软件开发人员为高性能计算服务器、桌面计算系统、手持设备编写高效轻便的代码，而且广泛适用于多核心处理器(CPU)、图形处理器(GPU)、Cell类型架构以及数字信号处理器(DSP)等其他并行处理器，AMD-ATI、NVIDIA时下的产品都支持OpenCL。目前，OpenCL最新版本为2.2.

CUDA(Compute Unified Device Architecture)是一种将GPU作为数据并行计算设备的软硬件体系，硬件上NVIDIA GeForce 8系列以后的GPU(包括GeForce、ION、Quadro、Tesla系列)已经采用支持CUDA的架构，软件开发包上CUDA也已经发展到CUDA Toolkit 8.0，并且支持Windows、Linux、MacOS三种主流操作系统。CUDA采用比较容易掌握的类C语言进行开发。

AMD APP(AMD Accelerated Parallel Processing)是AMD加速并行处理技术。是AMD针对旗下图形处理器(GPU)所推出的通用并行计算技术。利用这种技术可以充分发挥AMD GPU的并行运算能力，用于对软件进行加速运算或进行大型的科学运算。AMD APP技术的前身称作ATI Stream。2010年10月，随着AMD Radeon HD6800系列显卡的发布，ATI品牌正式被AMD取代。ATI Stream技术也随着技术升级并更名为AMD APP技术。目前，AMD APP SDK最新版本为3.0.

DirectCompute是一种用于GPU通用计算的应用程序接口，由Microsoft(微软)开发和推广，集成在Microsoft DirectX内。目前，最新的DirectX版本为DirectX 12，安装在windows 10上。DirectX 11内集成DirectCompute 5.0，那DirectX 12内应该是集成DirectCompute 6.0吧。

其中OpenCL、DirectCompute、AMD APP(基于开放型标准OpenCL开发)是开放标准，CUDA是私有标准。

三、NVIDIA 显卡系列

NVIDIA(英伟达)创立于1993年1月，是一家以设计智核芯片组为主的无晶圆(Fabless)IC半导体公司。

NVIDIA已经开发出了五大产品系列，以满足特定细分市场需求，包括：GeForce、Tegra、ION、Quadro、Tesla。

Geforce系列主要面向家庭和企业的娱乐应用,该系列又可以分为面向性能的GTX系列,面向主流市场的GTS和GT系列，已经具有高性价比的GS系列。

Quadro系列主要应用于图形工作站中，对专业领域应用进行了专门优化。

Tesla系列是专门用于高性能通用计算的产品线。

Tegra系列是NVIDIA为便携式和移动领域推出的全新解决方案，在极为有限的面积上集成了通用处理器、GPU、视频解码、网络、音频输入输出等功能，并维持了极低的功耗。

针对Geforce显卡系列，NVIDIA各代显卡都遵循了由高至低命名规则：GTX>GTS>GT>GS

从GTX 500系开始，为避免命名复杂带来的产品线识别困扰，NVIDIA显卡将取消GTS级别的显卡，中高端全部使用GTX命名，而低端使用GT命名，带Ti后缀为更高一级显卡，如GTX 560 Ti > GTX 560.

NVIDIA显卡末尾数字解读，以GeForce GTX 980M：GTX代表是高端显卡的意思；980M：第一位数字9，代表第几代的意思(9是高端显卡第九代的意思，如果末尾数字有四位，则前两位表示是第多少代的意思，如GeForce GTX 1080)。第二位至关重要，因为显卡分高端显卡，中端显卡，入门级显卡就是取决于第二位数字的。第二位数字是1-2代表是入门级显卡；第二位数字是3-5代表是中端显卡；第二位数字是6-9代表是高端显卡。第三位数字是一个特殊的标志，几乎能在市场上买到的显卡都是0结尾的，如果第三位数字为5的显卡一般都是OEM显卡，即只给大厂子做品牌机的特供。数字越大，性能越好。显卡数字后缀Ti，代表加强。

如果用显卡来进行各种运算，衡量显卡性能的参数可包括：(1)、核心数目；(2)、显存带宽(GPU计算能力太强，很多时候瓶颈都在数据传输上)；(3)、峰值单精度浮点计算能力；(4)、峰值双精度浮点计算能力；(5)、时钟频率；(6)、架构版本。

四、CUDA基础

1. 简介

CUDA(Compute Unified Device Architecture，统一计算设备架构)，是显卡厂商NVIDIA在2007年推出的并行计算平台和编程模型。它利用图形处理器(GPU)能力，实现计算性能的显著提高。CUDA是一种由NVIDIA推出的通

libsvm
joys99
CodeForge
cvchina
tesseract-ocr
sift
TiRG
imgSeek
OpenSURF

Friendly Link

OpenCL
Python
poesia-filter
TortoiseSVN
imgSeek
Notepad
Beyond Compare
CMake
VIGRA
CodeGuru
vchome
aforgenet
CVLAB
Doxygen
Coursera
OpenMP

Technical Forum

Matlab China
OpenCV China
The CImg Library
Open Computer Vision Library
CxImage
ImageMagick
ImageMagick China
OpenCV_China
Subversion China

用并行计算架构，该架构使GPU能够解决复杂的计算问题，从而能通过程序控制底层的硬件进行计算。它包含了CUDA指令集架构(ISA)以及GPU内部的并行计算引擎。开发人员可以使用C/C++/C++11语言来为CUDA架构编写程序。CUDA提供host-device的编程模式以及非常多的接口函数和科学计算库，通过同时执行大量的线程而达到并行的目的。

3.0以下版本仅支持C编程，从3.0版本开始支持C++编程，从7.0版本开始支持C++11编程。

CUDA仅能在有**NVIDIA**显卡的设备上才能执行，并不是所有的NVIDIA显卡都支持CUDA，目前NVIDIA的GeForce、ION、Quadro以及Tesla显卡系列上均可支持。根据显卡本身的性能不同，支持CUDA的版本也不同。

2. 安装

(1)、在windows上的安装可以参考：http://blog.csdn.net/fengbingchun/article/details/53892997

(2)、在ubuntu上的安装可以参考：http://blog.csdn.net/fengbingchun/article/details/53840684

3. 使用CUDA C编写代码的前提条件

(1)、支持CUDA的图形处理器：从2007年开始，NVIDIA新推出的并且显存超过256MB的GPU都可以用于开发和运行基于CUDAC编写的代码。

(2)、NVIDIA设备驱动程序：NVIDIA提供了一些系统软件来实现应用程序与支持CUDA的硬件之间的通信，即显卡驱动程序。要确保安装匹配的驱动程序，选择与开发环境相符的图形卡和操作系统。

(3)、CUDA开发工具箱：CUDA Toolkit，注意选择与操作系统相匹配的CUDA Toolkit。

(4)、标准C编译器：由于CUDA C应用程序将在两个不同的处理器上执行计算，因此需要两个编译器。其中一个编译器为GPU编译代码，而另一个为CPU编译代码。下载并安装CUDA Toolkit后，就会获得一个编译GPU代码的编译器。对于CPU编译器，Windows推荐使用Visual Studio，Linux使用GNU C编译器(gcc)，Mac使用Xcode。

4. 设备计算能力

设备计算能力的版本描述了一种**GPU**对**CUDA**功能的支持程度。计算能力版本中小数点前的第一位用于表示设备核心架构，小数点后的第一位则表示更加细微的进步，包括对核心架构的改进以及功能的完善等。例如，计算能力1.0的设备能够支持CUDA，而计算能力1.1设备加入了对全局存储器原子操作的支持，计算能力1.2的设备则可以支持warp vote函数等更多功能，而计算能力1.3的设备又加入了对双精度浮点运算功能。

GeForce GTX 970型号计算能力为5.2，GeForce GT 640M型号计算能力为3.0，目前GeForce系列最高的计算能为6.1，可在https://developer.nvidia.com/cuda-gpus中查找各种系列型号的计算能力以及查找指定的显卡型号是否支持CUDA。

5. 软件体系

CUDA的软件堆栈由三层构成，如下图，CUDA Library、CUDA runtimeAPI、CUDA driver API. CUDA的核心是CUDA C语言，它包含对C语言的最小扩展集和一个运行时库，使用这些扩展和运行时库的源文件必须通过nvcc编译器进行编译。

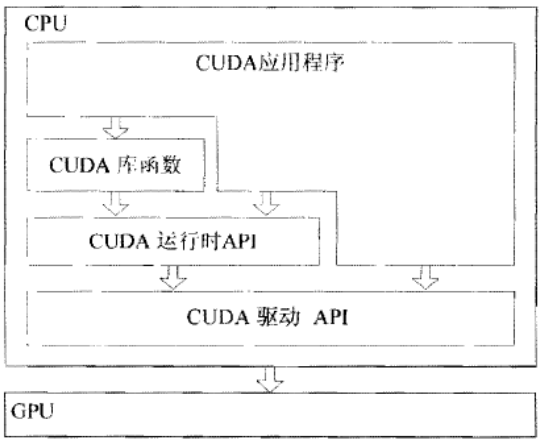

图 2-14 CUDA 软件体系

Technical Blog

邹宇华
深之JohnChen
HUNNISH
周伟明
superdont
carson2005
OpenHero
Netman(Linux)
wqvbjhc
yang_xian521
gnuhpc
gnuhpc
千里8848
CVART
tornadomeet
gotosuc
onezeros
hellogv
abcjennifer
crzy_sparrow

评论排行

Windows7 32位机上，O (120)
tiny-cnn开源库的使用(MI (93)
Ubuntu 14.04 64位机上7 (89)
tesseract-ocr3.02字符识 (63)
Windows7上使用VS201 (47)
tesseract-ocr (42)
图像配准算法 (41)
Windows 7 64位机上Ope (36)
OpenCV中resize函数五种 (34)
小波矩特征提取matlab代 (30)

最新评论

Tesseract-OCR 3.04在Windows
fengbingchun: @ilikede:没有密码，那个commit只是提示是从哪个commit fork过来的，无需管那个

Tesseract-OCR 3.04在Windows
ilikede: 问一下，你第一句中的commit的那个密码，怎么用啊

卷积神经网络(CNN)的简单实现(
fengbingchun: @hugl950123:是需要opencv的支持，你在本地opencv的环境配好了吗，配好了就应该没...

卷积神经网络(CNN)的简单实现(
hugl950123: @fengbingchun:博主请问一下，test_CNN_predict()函数是不是需要open...

卷积神经网络(CNN)的简单实现(
hugl950123: @fengbingchun:博主请问一下，test_CNN_predict()函数是不是需要open...

卷积神经网络(CNN)的简单实现(
hugl950123: @fengbingchun:谢谢，能够成功运行了现在

卷积神经网络(CNN)的简单实现(
fengbingchun: @hugl950123:NN中一共有四个工程，它们之间没有任何关系，都是独立的，如果要运行这篇文章的...

CUDA C语言编译得到的只是GPU端代码，而要管理GPU资源，在GPU上分配显存并启动内核函数，就必须借助CUDA运行时API(runtime API)或者CUDA驱动API(driver API)来实现。在一个程序中只能使用**CUDA**运行时**API**与**CUDA**驱动**API**中的一种，不能混合使用。

6. CUDA C语言

CUDA C语言为程序员提供了一种用C语言编写设备端代码的编程方式，包括对C的一些必要扩展和一个运行时库，CUDA对C的扩展主要包括以下几个方面：

(1)、引入了函数类型限定符，用来规定函数是在host还是在device上执行，以及这个函数是从host调用还是从device调用。这些限定符有：__device__、__host__、__global__。

(2)、引入了变量类型限定符，用来规定变量被存储在哪一类存储器上。传统的在CPU上运行的程序，编译器能自动决定将变量存储在CPU的寄存器还是内存中。在CUDA编程模型中，一共抽象出来8种不同的存储器。为了区分各种存储器，引入了一些限定符，包括：__device__、__shared__、__constant__。

(3)、引入了内置矢量类型，如char4、ushort3、double2、dim3等，它们是由基本的整形或浮点型构成的矢量类型，通过x、y、z、w访问每一个分量，在设备端代码中各矢量类型有不同的对齐要求。

(4)、引入了4个内置变量：blockIdx和threadIdx用于索引线程块和线程，gridDim和blockDim用于描述线程网格和线程块的维度。warpSize用于查询warp中的线程数量。

(5)、引入了<<<>>>运算符，用于指定线程网格和线程块维度，传递执行参数。

对__global__函数的任何调用都必须指定该调用的执行配置(execution configuration)。执行配置用于定义在设备上执行函数时的grid和block的维度，以及相关的流。

使用驱动API时，需要通过一系列驱动函数设置执行配置参数。

使用运行时**API**时，需要在调用的内核函数名与参数列表直接以**<<<Dg,Db,Ns,S>>>**的形式设置执行配置，其中：

Dg是一个dim3型变量，用于设置grid的维度和各个维度上的尺寸。设置好Dg后，grid中将有Dg.x*Dg.y个block，Dg.z必须为1.

Db是一个dim3型变量，用于设置block的维度和各个维度上的尺寸。设置好Db后，每个block中将有Db.x*Db.y*Db.z个thread。

Ns是一个size_t型变量，指定各块为此调用动态分配的共享存储器大小，这些动态分配的存储器可供声明为外部数组(extern __shared__)的其他任何变量使用；Ns是一个可选参数，默认值为0.

S为cudaStream_t类型，用于设置与内核函数关联的流。S是一个可选参数，默认值为0.

(6)、引入了一些函数：memory fence函数、同步函数、数学函数、纹理函数、测时函数、原子函数、warp vote函数。

以上扩展均有一些限制，如果违背了这些限制，nvcc将给出错误或警告信息，但有时也不会报错，程序无法运行。

7. 常用术语

(1)、主机(host)：将CPU及系统的内存称为主机。

(2)、设置(device)：将GPU及GPU本身的显示内存称为设备，在一个系统中可以存在一个主机和若干个设备。

CUDA编程模型中，CPU与GPU协同工作，CPU负责进行逻辑性强的事务处理和串行计算，GPU则专注于执行高度线程化的并行处理任务。CPU、GPU各自拥有相互独立的存储器地址空间：主机端的内存和设备端的显存。

(3)、线程(Thread)：一般通过GPU的一个核进行处理，可以表示成一维、二维、三维。一个block中的所有thread在一个时刻执行指令并不一定相同。

(4)、线程块(Block)：由多个线程组成，可以表示成一维、二维、三维；各**block**是并行执行的，**block**间无法通信，也没有执行顺序；注意线程块的数量有限制(硬件限制)。

Block内，可以通过__syncthreads()进行线程同步；thread间通过shared memory进行通信。

卷积神经网络(CNN)的简单实现(
hugl950123: @fengbingchun:下的是新的，我在CNN.cpp文件中每个函数都设置了断点，还是没有变化=...
卷积神经网络(CNN)的简单实现(
fengbingchun: @hugl950123:你用的是GitHub上最新的吗？既然能编译过，在Debug下设断点，应该很快...
卷积神经网络(CNN)的简单实现(
hugl950123: 博主，请问我按照您的代码成功编译后执行结果窗口一闪而过，并且里面什么内容也没有，应该如何解决，能不能...

阅读排行

C#中OpenFileDialog的使 (47141)
tesseract-ocr3.02字符识 (34575)
举例说明使用MATLAB C (25987)
OpenCV中resize函数五种 (24317)
利用cvMinAreaRect2求取 (24277)
Windows 7 64位机上搭建 (22586)
opencv 检测直线、线段、 (20776)
OpenCV运动检测跟踪(bl (20475)
图像配准算法 (19237)
有效的rtsp流媒体测试地 (19143)

文章存档

2017年01月 (18)
2016年12月 (11)
2016年11月 (8)
2016年10月 (7)
2016年09月 (16)
展开

碧桂园森林城市

在实际运行中，block会被分割成更小的线程束(warp)。线程束的大小由硬件的计算能力版本决定。Warp中的线程只与thread ID有关，而与block的维度和每一维的尺度没有关系。

(5)、线程格(Grid)：由多个线程块组成，可以表示成一维、二维、三维。

(6)、线程束：在CUDA架构中，线程束是指一个包含32个线程的集合，这个线程集合被"编织在一起"并且"步调一致"的形式执行,在程序中的每一行，线程束中的每个线程都将在不同数据上执行相同的命令。

(7)、核函数(Kernel)：运行在GPU上的CUDA并行计算函数称为kernel(内核函数)。内核函数必须通过__global__函数类型限定符定义，并且只能在主机端代码中调用。在调用时，必须声明内核函数的执行参数即"<<<>>>"，用于说明内涵函数中的线程数量，以及线程是如何组织的。不同计算能力的设备对线程的总数和组织方式有不同的约束。必须先为Kernel中用到的数组或变量分配好足够的空间，再调用kernel函数，否则在GPU计算时会发生错误，例如越界或报错，甚至导致蓝屏和死机。

在设备端运行的线程之间是并行执行的，其中的每个线程则按照指令的顺序串行执行一次**kernel**函数。每一个线程有自己的**block ID**和**thread ID**用于与其它线程相区分。**blockID**和**thread ID**只能在**kernel**中通过内置变量访问。内置变量不需要由程序员自己定义，是由设备中的专用寄存器提供的。因此，内置变量是只读的，并且只能在GPU端的kernel函数中使用。

Kernel是以**block**为单位执行的，CUDA引入grid只是用来表示一系列可以被并行执行的block的集合。各**block**是并行执行的，**block**间无法通信，也没有执行顺序，在同一个**block**中的线程，可以进行数据通信，在同一个block中的线程通过共享存储器(shared memory)交换数据，并通过栅栏同步(可以在kernel函数中需要同步的位置调用__syncthreads()函数)保证线程间能够正确地共享数据。这样，无论是只能同时处理一个线程块的GPU上，还是在能同时处理数十乃至上百个线程块的GPU上，这一CUDA编程模型都能很好地适用。

一个kernel函数并不是一个完整的程序，而是整个CUDA程序中一个可以被并行执行的步骤。一个完整的CUDA程序是由一系列的设备端kernel函数并行步骤和主机端的串行处理步骤共同组成的。如下图(CUDA编程模型)：

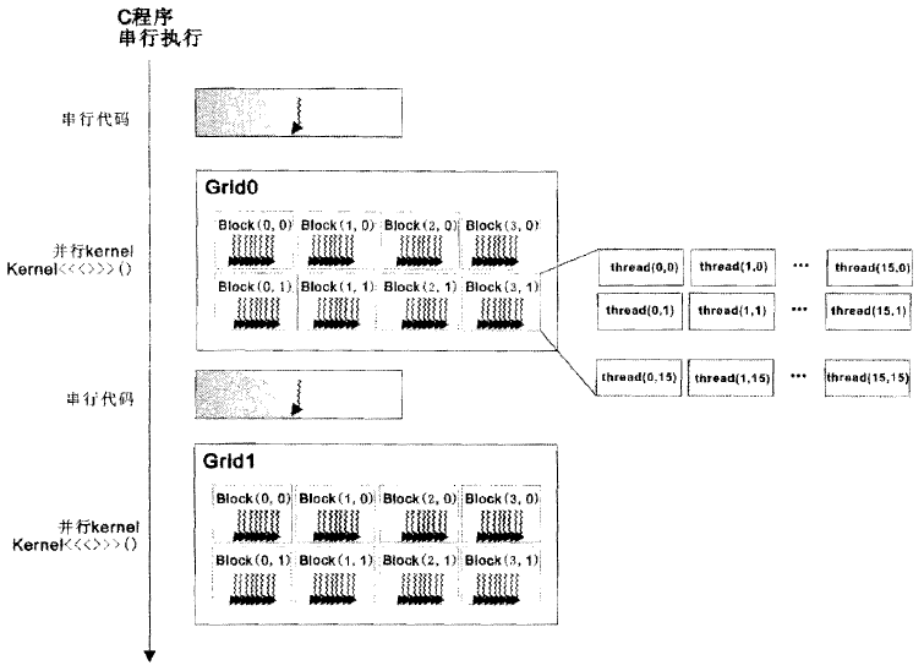

图 2-1 CUDA 编程模型

CPU串行代码完成的工作包括在kernel启动前进行数据准备和设备初始化的工作，以及在kernel之间进行一些串行计算。理想情况下，CPU串行代码的作用应该只是清理上一个内核函数，并启动下一个内核函数。在这种情况下，可以在设备上完成尽可能多的工作，减少主机与设置之间的数据传输。

8. 内置变量

内置变量用于确定**grid**和**block**的维度，以及**block**和**thread**在其中的索引。这些内置变量只能在设备端执行的函数(__global__、__device__)中使用。

(1)、dim3：基于uint3定义的矢量类型，相当于由3个unsigned int类型组成的结构体，可表示一个三维数组，在定义dim3类型变量时，凡是没有赋值的元素都会被赋予默认值1.其它常用基本数据类型可参考include/vector_types.h文件。

(2)、threadIdx：内置变量，用于说明当前thread在block中的位置；如果线程是一维的可获取threadIdx.x，如果是二维的还可获取threadIdx.y，如果是三维的还可获取threadIdx.z；为uint3类型，包含了一个thread在block中各个维度的索引信息。可参考include/device_launch_parameters.h文件。

threadIdx.x取值范围是[0,blockDim.x -1]，threadIdx.y取值范围[0, blockDim.y-1]，threadIdx.z取值范围[0, blockDim.z-1]。

(3)、blockIdx：内置变量，用于说明当前thread所在的block在整个grid中的位置，blockIdx.x取值范围是[0,gridDim.x-1]，blockIdx.y取值范围是[0, gridDim.y-1]。为uint3类型，包含了一个block在grid中各个维度上的索引信息。

对于一维的block，线程的threadID就是threadIdx.x；

对于大小为(Dx, Dy)的二维block，线程的threadID是(threadIdx.x+ threadIdx.y * Dx)；

对于大小为(Dx, Dy, Dz)的三维block，线程的threadID是(threadIdx.x+ threadIdx.y * Dx + threadIdx.z * Dx * Dy).

(4)、blockDim：内置变量，用于说明每个block的维度与尺寸。为dim3类型，包含了block在三个维度上的尺寸信息。

(5)、gridDim：内置变量，用于说明整个网格的维度与尺寸，一个grid最多只有二维。为dim3类型，包含了grid在三个维度上的尺寸信息。

```cpp
       uint3 __device_builtin__ __STORAGE__threadIdx;
uint3 __device_builtin__ __STORAGE__ blockIdx;
dim3 __device_builtin__ __STORAGE__ blockDim;
dim3 __device_builtin__ __STORAGE__ gridDim;
```

(6)、warpSize：内置变量，用于引用warpsize。为int类型，用于确定设备中一个warp包含多少个thread.

以上这些内置变量只能在设备端代码中使用，这些变量是只读的，不能对它们赋值，也不能对它们取地址。

9. 变量类型限定符

变量类型限定符用于指明变量存储在设备端的哪一类存储器上。

(1)、__device__：声明的变量存在于设备上。当__device__变量限定符不与其他限定符连用时，这个变量将：位于全局存储器空间中；与应用程序具有相同的生命周期；可以通过运行时库从主机端访问，设备端的所有线程也可访问。

(2)、__constant__：使用__constant__限定符，或者与__device__限定符连用，这样声明的变量：存在于常数存储器空间；与应用程序具有相同的生命周期；可以通过运行时库从主机端访问，设备端的所有线程也可访问。

(3)、__shared__：使用__shared__限定符，或者与__device__限定符连用，此时声明的变量：位于block中的共享存储器空间中；与block具有相同的生命周期；仅可通过block内的所有线程访问。

(4)、volatile：存在于全局或者共享存储器中的变量通过volatile关键字声明为敏感变量，编译器认为其他线程可能随时会修改变量的值，因此每次对该变量的引用都会被编译成一次真实的内存读指令。

以上限定符不能用于struct与union成员、在主机端执行的函数的形参以及局部变量。

__shared__和__constant__变量默认为是静态存储。

__device__、__shared__和__constant__不能用extern关键字声明为外部变量。在__**shared**__前可以加**extern**关键字，但表示的是变量大小由执行参数确定。

__device__和__constant__变量只能在文件作用域中声明，不能再函数体内声明。

__constant__变量不能从device中赋值，只能从host中通过host运行时函数赋值。

__shared__变量在声明时不能初始化。

在设备代码中**(__global__**或者**__device__**函数中**)**，如果一个变量前没有任何限定符，这个变量将被分配到寄存器中。但如果寄存器资源不足，编译器会把这些变量存放在**local memory**中。Local memory中的数据被存放于显存中，而且没有任何缓存可以加速local memory的读写，因此会大大降低程序的速度。

只要编译器能够解析出设备端代码中的指针指向的地址，指向shared memory或者global memory，这样的指针即受支持。如果编译器不能正确地解析指针指向的地址，那么只能使用指向global memory的指针。

在host端代码中使用指向global或者shared memory的指针，或者在device端代码中使用指向host memory的指针都将引起不确定的行为，通常会报分区错误(segmentation fault)并导致程序终止运行。

在device端通过取址符号&获得的__device__、__constant__、__shared__的地址，这样得到的地址只能在device端使用。通过在host端调用cudaGetSymbolAddress()函数可以获得__device__、__constant__的地址，这样得到的地址只能在host端使用。

10. 函数类型限定符

(1)、__global__：表明被修饰的函数在设备上执行，只能从主机端调用；

(2)、__device__：表明被修饰的函数在设备上执行，只能从设备上调用，但只能在其它__device__函数或者__global__函数中调用；

(3)、__host__：在主机端上执行，只能从主机端调用。

没有**__host__**、**__device__**、**__global__**限定符修饰的函数，等同于只用**__host__**限定符修饰的函数，函数都将仅为主机端进行编译，即编译出只能在主机端运行的版本。__host__可以与__device__一起使用，此时函数将为主机和设备进行编译，即分别编译出在主机和设备端运行的版本。

使用限制：

(1)、__device__和__global__函数不支持递归；

(2)、__device__和__global__函数的函数体内不能声明静态变量；

(3)、__device__和__global__函数的参数数目是不可变化的；

(4)、不能对__device__取指针，但可以对__global__函数取指针；

(5)、__global__与__host__不能连用；

(6)、__global__函数的返回类型必须为void；

(7)、调用__global__函数必须指明其执行配置；

(8)、对**__global__**函数的调用是异步的，控制权在设备执行完成之前就会返回；

(9)、__global__函数的参数目前通过共享存储器传递，总的大小不能超过256Byte。

11. CUDA存储器模型

每一个线程拥有自己的私有存储器寄存器和局部存储器；每一个线程块拥有一块共享存储器(shared memory)；最后,grid中所有的线程都可以访问同一个全局存储器(global memory)。除此以外，还有两种可以被所有线程访问的只读存储器：常数存储器(constant memory)和纹理存储器(texture memory)，它们分别为不同的应用进行了优化。全局存储器、常数存储器和纹理存储器中的值在一个内核函数执行完成后将被继续保持，可以被同一程序中的其他内核函数调用。

八种存储器比较如下图：

表 2-2 各种存储器比较

存储器	位置	拥有缓存	访问权限	变量生存周期
register	GPU 片内	N/A①	device 可读/写	与 thread 相同
local memory	板载显存	无	device 可读/写	与 thread 相同
shared memory	GPU 片内	N/A①	device 可读/写	与 block 相同
constant memory	板载显存	有	device 可读，host 可读/写	可在程序中保持
texture memory	板载显存	有	device 可读，host 可读/写	可在程序中保持
global memory	板载显存	无	device 可读/写，host 可读/写	可在程序中保持
host memory	host 内存	无	host 可读/写	可在程序中保持
pinned memory	host 内存	无	host 可读/写②	可在程序中保持

注：① N/A：Register 和 shared memory 都是 GPU 片上高速存储器。

② 通过 mapped memory 实现的 zero copy 功能，某些 GPU 可以直接在 kernel 中访问 page-locked memory。

(1)、寄存器(register)：是GPU片上高速缓存器，执行单元可以以极低的延迟访问寄存器。寄存器的基本单元是寄存器文件(register file)，每个寄存器文件大小为32 bit。

(2)、局部存储器(local memory)：对于每个线程，局部存储器也是私有的。如果寄存器被消耗完储在局部存储器中。如果每个线程使用了过多的寄存器，或声明了大型结构体或数组，或者编译器无法确定数组的大小，线程的私有数据就有可能会被分配到local memory中。一个线程的输入和中间变量将被保存在寄存器或者局部存储器中。局部存储器中的数据被保存在显存中，而不是片上的寄存器或者缓存中，因此对**local memory**的访问速度很慢。

(3)、共享存储器(shared memory)：也是GPU片内的高速存储器。它是一块可以被同一block种的所有线程访问的可读写存储器。访问共享存储器的速度几乎和访问寄存器一样快，是实现线程间通信的延迟最小的方法。共享存储器可用于实现多种功能，如用于保存共用的计数器或者block的公用结果。

可以将CUDA C的关键字__shared__添加到变量声明中，这将使这个变量驻留在共享内存中。CUDA C编译器对共享内存中的变量与普通变量将分别采取不同的处理方式。对于GPU上启动的每个线程块，CUDA C编译器都将创建该共享变量的一个副本。线程块中的每个线程都共享这块内存，但线程却无法看到也不能修改其他线程块的变量副本。这样使得一个线程块中的多个线程能够在计算上通信和协作。

(4)、全局存储器(global memory)：全局存储器位于显存(占据了显存的绝大部分)，CPU、GPU都可以进行读写访问。整个网格中的任意线程都能读写全局存储器的任意位置由于全局存储器是可写的。在目前的架构中，全局存储器没有缓存。

全局存储器能够提供很高带宽，但同时也具有较高的访存延迟。要有效地利用全局存储器带宽，必须遵守和并访问要求，并避免分区冲突。

在运行时API中，显存中的全局存储器也称为线性内存。线性内存通常使用cudaMalloc()函数分配，cudaFree()函数释放，并由cudaMemcpy()进行主机端与设备端的数据传输。通过CUDA API分配的空间未经过初始化，初始化共享存储器需要调用cudaMemset函数。

此外，也可以使用__device__关键字定义的变量分配全局存储器。这个变量应该在所有函数外定义，必须对使用这个变量的host端和device端函数都可见才能成功编译。在定义__device__变量的同时可以对其赋值。

在驱动API中，线性内存由cuMemAlloc()或cuMemAllocPitch()来分配，cuMemFree()来释放。

(5)、主机端内存(host memory)：在CUDA中,主机端内存分为两种。可分页内存(pageable memory)和页锁定(page-locked或pinned)内存。可分页内存即为通过操作系统API(malloc(), new())分配的存储器空间；而页锁定内存始终不会被分配到低速的虚拟内存中，能够保证存在于物理内存中，并且能够通过DMA加速与设备端的通信。一般的主机端内存操作方法与其他程序没有任何区别。

(6)、主机端页锁定内存(pinned memory)：它有一个重要的属性，即操作系统将不会对这块内存分页并交换到磁盘上，从而确保了该内存始终驻留在物理内存上。因此，操作系统能够安全地使某个应用程序访问该内存的物理地址，因为这块内存将不会被破坏或者重新定位。它可以提高访问速度，由于GPU知道主机内存的物理地址，因此可以通过"直接内存访问DMA(Direct Memory Access)技术来在GPU和主机之间复制数据。由于DMA在执行复制时无需CPU介入。因此DMA复制过程中使用固定内存是非常重要的。

pinned memory是一把双刃剑。当使用pinned memory时，你将失去虚拟内存的所有功能。特别是，在应用程

序中使用每个页锁定内存时都需要分配物理内存，因为这些内存不能交换到磁盘上。这意味着，与使用标准的malloc函数调用相比，系统将更快地耗尽内存。因此，应用程序在物理内存较少的机器上会运行失败，而且意味着应用程序将影响在系统上运行的其它应用程序的性能。建议，仅对cudaMemcpy()调用中的源内存或者目标内存，才使用页锁定内存，并且在不再需要使用它们时立即释放，而不是等到应用程序关闭时才释放。

在运行时API中，通过cudaHostAlloc()和cudaFreeHost()来分配和释放pinned memory。使用pinned memory有很多好处，比如：可以达到更高的主机端----设备端的数据传输带宽，如果页锁定内存以write-combined方式分配，带宽还能更高一些；某些设备支持DMA功能，在执行内核函数的同时利用pinned memory进行主机端与设置端之间的通信；在某些设备上，pinned memory还可以通过zero-copy功能映射到设备地址空间，从GPU直接访问，这样就不用在主存与显存间进行数据拷贝工作了。

虽然pinned memory能带来诸多好处，但它是系统中的一种稀缺资源。如果分配过多，会导致操作系统用于分页的物理内存变小，导致系统整体性能下降。

在驱动API中，pinned memory通过cuMemHostAlloc()和一些标志分配，通过cuMemFreeHost()释放。

(7)、常数存储器(constant memory)：是只读的地址空间。常数存储器中的数据位于显存，但拥有缓存加速。常数存储器的空间较小(只有64KB),在CUDA程序中用于存储需要频繁访问的只读参数。当来自同一half-warp的线程访问常数存储器中的同一数据时，如果发生缓存命中，那么只需要一个周期就可以获得数据。

常数存储器有缓存机制，用以节约带宽，加快访问速度。每个SM拥有8KB的常数存储器缓存。常数存储器是只读的，因此不存在缓存一致性问题。

constant memory用于保存在核函数执行期间不会发生变化的数据。NVIDIA硬件提供了64KB的常量内存，并且对常量内存采取了不同于标准全局内存的处理方式。在某些情况下，用常量内存来替换全局内存能有效地减少内存带宽。要使用常量内存，需在变量前面加上__constant__关键字。常量内存用于保存在核函数执行期间不会发生变化的数据。变量的访问限制为只读。

(8)、纹理存储器(texture memory)：是一种只读存储器，由GPU用于纹理渲染的图形专用单元发展而来，具备一些特殊功能。它并不是一块专门的存储器，而是牵涉到显存、两级纹理缓存、纹理拾取单元的纹理流水线。纹理存储器中的数据以一维、二维或者三维数组的形式存储在显存中，可以通过缓存加速访问，并且可以声明大小比常数存储器要大的多。在通用计算中，纹理存储器非常适合实现图像处理和查找表，对大量数据的随机访问或非对齐访问也有良好的加速效果。

在kernel中访问纹理存储器的操作称为纹理拾取(texture fetching).纹理拾取使用的坐标与数据在显存中的位置可以不同。

与常数存储器类似，纹理存储器也有缓存机制，纹理缓存有两个作用。首先，纹理缓存中的数据可以被重复利用，当一次访问需要的数据已经存在于纹理缓存中时，就可以避免对显存的再次读取。数据重用过滤了一部分对显存的访问，节约了带宽，也不必按照显存对齐的要求读取。其次，纹理缓存一次预取拾取坐标对应位置附近的几个像元，可以实现滤波模式，也可以提高具有一定局部性的访存的效率。

纹理存储器是只读的，因此没有数据一致性可言。

与constant memory类似的是，texture memory同样缓存在芯片上，因此在某些情况中，它能够减少对内存的请求并提供更高效的内存带宽。纹理缓存是专门为那些在内存访问模式中存在大量空间局部性(Spatial Locality)的图形应用程序而设计的。纹理变量(引用)必须声明为文件作用域内的全局变量。分为一维纹理内存和二维纹理内存。

12. CUDA通信机制

(1)、同步函数：__syncthread()实现了线程块内的线程同步，它保证线程块中的所有线程都执行到同一位置。当任意一个thread运行到BAR标记处后，就会暂停运行；直到整个block中所有的thread都运行到BAR标记处以后，才继续执行下面的语句。这样，才能保证之前语句的执行结果对块内所有线程可见。如果不做同步，一个线程块中的一些线程访问全局或者共享存储器的同一地址时，可能会发生读后写、写后读、写后写错误。而通过同步可以避免这些错误的发生。

只有当整个线程块都走向相同分支时，才能在条件语句里面使用__syncthreads()，否则可能引起错误。另外，一个warp内的线程不用同步。也就是说，如果需要同步的线程处于同一warp中，则不需要调用__syncthreads()。可以使用特别的宏函数对warp内的threads进行同步。

Memory fence函数也是用来保证线程间数据通信的可靠性的。但与同步函数不同，memory fence函数并不要

求所有线程都运行到同一位置，而只保证执行memory fence函数的线程生产的数据能够安全地被其它线程消费。

kernel间通信：kernel直接的数据传递，可以通过global memory实现。

GPU与CPU线程同步：在CUDA主机端代码中使用cudaThreadSynchronize()，可以实现GPU与CPU线程的同步。Kernel启动后控制权将异步返回，利用该函数可以确定所有设备端线程均已运行结束，基本只是用来实现更加准确的计时或捕获运行错误。

(2)、原子(ATOM)操作：如果操作的执行过程不能分解为更小的部分，将满足这种条件限制的操作称为原子操作。

如函数调用，atomicAdd(addr,y)将生成一个原子的操作序列，这个操作序列包括读取地址addr处的值，将y增加到这个值，以及将结果保存回地址addr。

只有1.1或者更高版本的GPU计算功能集才能支持全局内存上的原子操作，且只能在设备端使用。此外，只有1.2或者更高版本的GPU计算功能集才能支持共享内存上的原子操作。CUDA C支持多种原子操作。可参考include/device_atomic_functions.h文件。

原子函数(atomic function)对位于全局或共享存储器的一个32位或64位字执行read-modify-write[illegible]就是说，当多个线程同时访问全局或共享存储器的同一位置时，保证每个线程能够实现对共享可写数据的互斥操作：在一个操作完成之前，其它任何线程都无法访问此地址。例如,atomicAdd()函数可以读入共享存储器或者全局存储器中的32bit字，与一个整数求和后，将结果写回到原位置上。之所以将这一过程称为原子操作，是因为每个线程的操作都不会影响到其它线程。换句话说，原子操作能够保证对一个地址的当前操作完成之前，其它线程都不能访问这个地址。

只能对有符号或者无符号整形进行原子操作(atomicExch()函数除外，该函数的操作数可以是有符号单精度浮点型)。

各种硬件对ATOM指令的支持、以及ATOM指令支持的数据类型不尽相同。

(3)、VOTE操作：VOTE指令是CUDA 2.0的新特性，只有1.2以上版本的硬件才能支持。VOTE的作用范围不是整个block，而是一个warp。

13. 异步并行执行

为了让主机端与设备端并行执行，很多函数都是异步的：控制在设备还没完成请求任务前就被返回给主机线程，这些函数有：kernel启动、以Async为后缀的内存拷贝函数、device到device内存拷贝函数、存储器初始化函数(比如cudaMemset())。

一些CUDA设备能够在kernel执行期间，执行pinnedmemory和显存间的数据传输。

异步执行的意义在于：首先，处于同一个流内的计算与数据拷贝是依次进行的，但一个流内的计算可以和另一个流的数据传输同时进行，因此通过异步执行就能够使GPU中的执行单元与存储器控制单元同时工作，提高了资源利用率；其次，当GPU在进行计算或者数据传输时就返回给主机线程，主机线程不必等待GPU运行完毕就可以继续进行一些计算，从而使得CPU和GPU可以并行工作。

如果调用了同步版本的GPU函数，在设备完成请求任务前，都不会返回主机线程，此时主机端线程将进入让步(yield)、阻滞(block)或者自旋(spin)状态。通过设置一些特定标记并调用cudaSetDeviceFlags()或cuCtxCreate()来选择主机端在进行GPU计算时进入的状态，不过和其它设置操作一样，该操作要在主机线程执行任何CUDA操作前就进行。

14. 流

程序通过流来管理并发，每个流是按顺序执行的一系列操作，而不同的流与其它的流之间乱序则是乱序执行的，也可能是并行执行的。这样，可以使一个流的计算与另一个流的数据传输同时进行，从而提高了GPU中资源的利用率。

流的定义方法，是创建一个cudaStream_t对象，并在启动内核和进行memcpy时将该对象作为参数传入，参数相同的属于同一个流，参数不同的属于不同的流。

执行参数中没有流参数，或使用0作为流参数时，不会创建流。此时，进行任何内核启动、内存设置或内存拷贝函数时，只有在之前所有的操作(包括流的部分操作)均已完成后才会开始，是异步执行方式。

驱动API提供了类似于运行时API的函数来管理流。

15. 事件

运行时API可以通过事件管理密切监控设备进度并执行准确计时，它可以异步地记录下程序内任意点的事件，并且可以查询这些事件被记录的时间。事件使用的GPU的计时器，用于测时比使用CPU的计时器更加准确。当先于该事件的所有任务(包括特定流中的所有操作)均已完成，这个事件的时戳就会被记录下来。0号流中的事件会在设备完成对所有流的操作后记录下来。事件管理可以用于测量程序运行时间，或者管理**CPU**和**GPU**同时进行计算。

驱动API提供类似于运行时API的函数来管理事件。

16. 指令与指令吞吐量

在CUDA中，吞吐量指每个多处理器在一个时钟周期下执行的操作数目。对于大小为32的warp，一条指令由32个操作构成。因此，如果记T为每个时钟下的操作数目，那么指令吞吐量就是每32/T个时钟周期一条指令。

所有的吞吐量都是针对一个多处理器而言的。所以，要计算整个设备的吞吐量需要乘以设备的多处理器个数。

17. CUDA与图形学API互操作

(1)、通过CUDA与OpenGL的互操作可以将OpenGL缓冲对象(buffer object)映射到CUDA的地址空间，这样就可以在CUDA 中读取OpenGL写入的数据，也可以用CUDA写入数据供OpenGL使用。要实现与OpenGL的互操作，必须在调用CUDA函数之前先调用cudaGLSetGLDevice()配置设备，并且在进行映射前要将OpenGL缓冲对象注册到CUDA。

要在驱动API中实现与OpenGL的互操作，就必须使用cuGLCtxCreate()而不是cuCtxCreate()创建CUDA上下文。和在运行API中一样，在进行映射前必须将缓冲对象注册到CUDA。

(2)、通过CUDA与Direct3D的互操作可以将Direct3D资源映射到CUDA地址空间，这样就可以在CUDA中读取由Direct3D写入的数据，也可以写入数据供Direct3D使用。Direct3D 9.0/10.0才支持Direct3D互操作。只有满足一些限制的Direct3D资源才能被映射到CUDA。由于DirectX 9和DirectX 10的资源有一定的差异，因此在CUDA中分别使用了不同的API与两个版本的DirectX进行互操作。

CUDA上下文一次仅可与一个Direct3D设备互操作，并且此时CUDA上下文和Direct3D设备必须是在同一个GPU上创建的。

驱动API提供了类似于运行时API的函数管理与Direct3D的互操作。

18. Runtime API和Driver API

Runtime API比Driver API更高级，封装的更好，在Runtime之上就是封装的更好的cuFFT等库。这两个库的函数都是能直接调用的，但Driver API相对于Runtime API对底层硬件驱动的控制会更直接更方便。Driver API向后兼容支持老版本的。大部分的功能两组API都有对应的实现，一般基于**Driver API**的开头会是**cu**，而基于**RuntimeAPI**的开头是**cuda**，但基于Driver API来写程序会比RuntimeAPI要复杂。

CUDA runtime API和CUDA driverAPI提供了实现设备管理(Device management)、上下文管理(Context management)、存储器管理(Memory management)、代码块管理(Code Module management)、执行控制(Execution Control)、纹理索引管理(Texture Reference management)、与OpenGL和Direct3D的互操作性(Interoperity with OpenGL and Direct3D)的应用程序接口。

(1)、CUDA runtimeAPI在CUDA driver API的基础上进行了封装，隐藏了一些实现细节，编程更加方便，代码更加简洁。CUDA runtime API被打包存放在CUDArt包里，其中的函数都有CUDA前缀。CUDA运行时没有专门的初始化函数，它将在第一次调用运行时函数时自动完成初始化。

(2)、CUDA driverAPI是一种基于句柄的底层接口(大多对象通过句柄被引用)，可以加载二进制或汇编形式的内核函数模块，指定参数，并启动计算。CUDA driver API编程复杂，但有时能通过直接操作硬件的执行实现一些更加复杂的功能，或者获得更高的性能。由于它使用的设备端代码是二进制或者汇编代码，因此可以在各种语言中调用。CUDA driver API被存放在nvCUDA包里，所有函数前缀为cu。

在调用任何一个驱动API函数之前，必须先调用cuInit()完成初始化，创建一个CUDA上下文。

19. 多设备与设备集群

在一台计算机中可以存在多个CUDA设备，通过CUDA API提供的上下文管理和设备管理功能可以使这些设备

并行工作。采取这种方式建立的多设备系统可以提高单台机器的性能，节约空间和成本。

CUDA的设备管理功能是由不同的线程管理各个GPU，每个GPU在一个时刻只能被一个线程使用。除了采用C提供的多线程库外，CUDA还支持使用OpenMP管理多个设备。

除了在单个系统中使用多个GPU外，也可以使用CPU+GPU异构系统作为节点构造集群，或者设计更大规模的CPU+GPU异构超级计算机。CUDA可以与MPI一起使用，提供成本更低，体积和功耗更小，性能更强的高性能计算解决方案。

(1)、CUDA设备控制：一个系统中可以有一个主机或多个设备。可以通过CUDA枚举这些设备，并查询它们的属性，每个主机端线程可以选取其中的一个设备执行内核程序。每个主机端线程各自管理一个设备，当主机端存在多个下线程时，就可以使多个设备能够并行工作。一个主机端线程通过CUDA运行时分配的CUDA资源不能被其它的主机端线程使用。

在默认情况下，如果没有调用设备管理函数，主机端线程将会在运行第一个运行时函数时自动使用设备**0**.

CUDA runtime API通过设备管理功能对多个设备进行管理。由CUDA运行时API管理多设备，需要使用多个主机端线程。每个主机端线程在第一次调用其它CUDA运行时API函数之前，必须先由设备管理函数cudaSetDevice()与一个设备关联，并且以后也不能再次调用cudaSetDevice()函数与其它设备关联。主机端线程的数量可以大于设备数量，但一个时刻一个设备上只有一个主机端线程的上下文。为了达到最高性能，最好使主机端线程数量与设备数量相同，每个线程与设备一一对应。

通过CUDA驱动API管理多设备与多个上下文要略微复杂一些。CUDA驱动API通过上下文管理功能将上下文与主机端线程关联，一个线程在一个时刻只能有一个与之关联的上下文。

(2)、CUDA与OpenMP：除了直接使用操作系统提供的API管理多线程外，CUDA也可以与OpenMP一起使用。

(3)、CUDA与集群：MPI(MessagePassing Interface, 消息传递接口)是国际上最流行的并行编程开发环境。CUDA也可以与MPI联用，实现集群或者超级计算机中的多节点多GPU并行计算。

20. 测量程序运行时间

CUDA的内核程序运行时间可以在设备端测量，也可以在主机端测量。而CUDA API的运行时间则只能从主机端测量。无论是主机端测时还是设备端测时，最好都测量内核函数多次运行的时间，然后再除以运行次数以获得更加准确的结果。使用CUDA runtime API时，会在第一次调用runtime API函数时启动CUDA环境，为了避免将这一部分时间计入，最好在正式测时开始前先进行一次包含数据输入输出的计算，这样也可以使GPU从平时的节能模式进入工作状态，使测时结果更加可靠。

(1)、设备端测时：使用GPU中的计时器的时戳计时。实现设备端测时有两种不同的方法，分别是调用clock()函数和使用CUDA API的事件管理功能。

使用clock()函数计时，在内核函数中要测量的一段代码的开始和结束的位置分别调用一次clock()函数，并将结果记录下来。由于调用__syncthreads()函数后，一个block中的所有thread需要的时间是相同的，因此只需要记录每个block执行需要的时间就行了，而不需要记录每个thread的时间。Clock()函数的返回值的单位是GPU的时钟周期，需要除以GPU的运行频率才能得到以秒为单位的时间。

在设备端执行clock()函数，将返回每一个多处理器的时间计数器中的值。该时间计数器在每一个时钟周期递增1.在内核启动和结束时对时间计数器取样，比较两个值，并由每个线程记录各自的结果，就可以知道每个线程在多处理器上运行了多长时间。但是这并不是每个线程在多处理器上实际执行的时间。实际执行的时间比按照上述测试得到的时间短，因为多处理器上的执行时间是由多个线程按照时间分片共享的。

(2)、主机端测时：与普通程序测时一样，CUDA的主机端测时也采用CPU的计时器测时。通常取得CPU中计时器的值的方法是调用汇编中的相应指令，或者操作系统提供的API。此外，一些函数库，如C标准库中的time库的clock_t()函数也可以用来测时。不过，clock_t()函数的精度很低，建议在两次调用clock_t()时，让待测程序运行至少数十次，运行时间达到数秒，再取平均求得每次运行时间。

使用**CPU**测时，一定要牢记**CUDA API**的函数都是异步的。这就是说，在一个CUDA API函数在GPU上执行完成之前，CPU线程就已经得到了它的返回值。内核函数和带有asyn后缀的存储器拷贝函数都是异步的。

要从主机端准备的测量一个或者一系列CUDA调用需要的时间，就要先调用cudaThreadSynchronize()函数，同步CPU线程与GPU之后，才能结束CPU测时。cudaThreadSynchronize()函数的功能是阻塞CPU线程，直到

cudaThreadSynchronize()函数之前所有的CUDA调用都已经完成。

与cudaThreadSynchronize()函数类似的函数有cudaStreamSynchronize()和cudaEventSynchronize()。它们的作用是阻塞所有Stream/CUDA Events，直到这条函数前的所有CUDA调用都已完成。注意，同一串流中的各个流可能会交替执行，因此即使使用了cudaStreamSynchronize()函数，也很难测得准确的执行时间。不过，一串流中的第一个流(ID为0的流)的行为总是同步的，因此使用这些函数对0号流进行测试，得到的结果是可靠的。

21. CUDA函数库

(1)、cuFFT(CUDA Fast Fourier Transform)：是一个利用GPU进行傅里叶变换的函数库，提供了与广泛使用的FFTW库相似的接口。

(2)、cuSparse：稀疏矩阵运算。

(3)、cuDNN：深度学习网络库。

(4)、cuBlas(CUDA Basic Linear Algebra Subprograms)：线性代数函数库，是一个基本的矩阵与向量的运算库，提供了与BLAS相似的接口，可以用于简单的矩阵计算，也可以作为基础构建更加复杂的函数包。

(5)、cuRand：随机数生成库。

(6)、cuDpp(CUDA Data Parallel Primitives)：提供了很多基本的常用的并行操作函数，如排序、搜索等，可以作为基本组件快速地搭建出并行计算程序。

22. 注意事项

(1)、在GPU上进行整数的除法和求模非常慢，避免这些运算能够有效地提高程序效率。

(2)、通常，block的数量都应该至少是处理核心的数量的几倍，才能有效地发挥GPU的处理能力。

(3)、在开发CUDA程序时应尽量避免分支，并尽量做到warp内不分支，否则将会导致性能急剧下降。

23. CUDA Toolkit

Toolkit是CUDA的核心软件包，打开toolkit的安装目录，如C:\ProgramFiles\NVIDIA GPU Computing Toolkit\CUDA\v7.5，此目录下主要目录介绍：

(1)、bin目录：包含一些工具程序如nvcc.exe(CUDAC编译器)、ptxas.exe(ptx转机器码)；一些动态链接库文件，包含w32和x64，如cudart64_75.dll(CUDA运行时API动态链接库)。

(2)、doc目录：里面包含了各种文档，包括pdf和html，可以根据实际需要查看相关文档说明。

(3)、include目录：包含常用的头文件，如cuda.h(CUDA驱动API头文件)。

(4)、lib目录：包含静态链接库，包含win32和x64，如cuda.lib(CUDA驱动库)、cudart.lib(CUDA运行时库)。

24. Samples

在C:\ProgramData\NVIDIACorporation\CUDA Samples\v7.5 目录下包含了很多CUDA例子程序，对进一步掌握CUDA很有帮助。

五、CUDA架构

NVIDIA GPU是基于CUDA架构而构建的。可以将CUDA架构视为NVIDIA构建GPU的模式，其中GPU既可以完成传统的图形渲染任务，又可以完成通用计算任务。要在CUDA GPU上编程，需要使用CUDA C语言。

CUDA架构包含了一个统一的着色器流水线,使得执行通用计算的程序能够对芯片上的每个数学逻辑单元(Arithmetic Logic Unit, ALU)进行排列。由于NVIDIA希望使新的图形处理器能适应于通用计算，因此在实现这些ALU时都确保它们满足IEEE单精度浮点数学运算的需求，并且可以使用一个裁剪后的指令集来执行通用计算，而不是仅限于执行图形计算。此外，GPU上的执行单元不仅能任意地读/写内存，同时还能访问由软件管理的缓存，也称为共享内存。CUDA架构的所有这些功能都是为了使GPU不仅能执行传统的图形计算，还能高效地执行通用计算。

NVIDIA采取工业标准的C语言，并且增加了一小部分关键字来支持CUDA架构的特殊功能。NVIDIA公布了一款编译器来编译CUDA C语言。这样，CUDA C就成为了第一款专门由GPU公司设计的编程语言，用于在GPU上编写通用计算。

除了专门设计一种语言来为GPU编写代码之外，NVIDIA还提供了专门的硬件驱动程序来发挥CUDA架构的大规模计算功能。

六、NVCC编译器

NVCC编译器根据配置编译CUDA C代码，可以生成三种不同的输出：PTX、CUDA二进制序列和标准C。nvcc是一种编译器驱动，通过命令行选项，nvcc可以在编译的不同阶段启动不同的工具完成编译工作。

nvcc工作的基本流程是：首先通过CUDAfe分离源文件中的主机端和设备端代码，然后再调用不同的编译器分别编译。设备端代码由nvcc编译成ptx代码或者二进制代码；主机端代码则将以C文件形式输出，由其他高性能编译器，如ICC、GCC或者其他合适的高性能编译器等进行编译。不过，也可以直接在编译的最后阶段，将主机端代码交给其他编译器生成.obj或者.o文件。在编译时，可以将设备端代码链接到所生成的主机端代码，将其中的cubin对象作为全局初始化数据数组包含进来。此时，内核执行配置也要被转换为CUDA运行启动代码，以加载和启动编译后的内核函数。使用CUDA驱动API时，可以单独执行ptx代码或者cubin对象，而忽略nvcc编译得到的主机端代码。

nvcc大概的编译流程如下图：

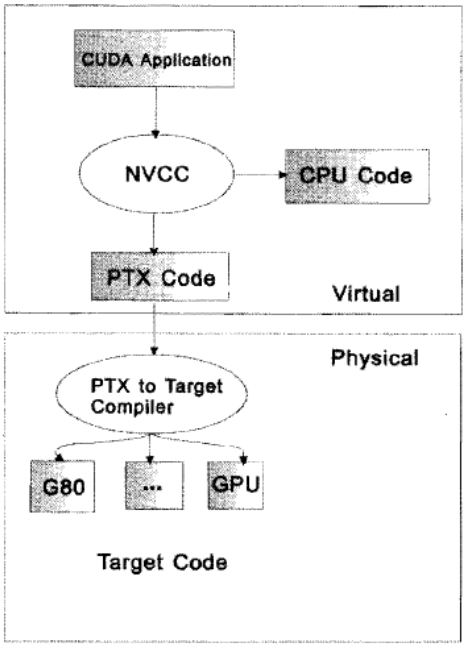

图 2-15 nvcc 编译

PTX(Parallel Thread eXecution)类似于汇编语言，是为动态编译器JIT(Just in time compiler, JIT包含在标准的NVIDIA驱动中)设计的输入指令序列。这样，虽然不同的显卡使用的机器语言不同，JIT却可以运行同样的PTX。这样做使PTX成为一个稳定的接口，带来了很多好处：向后兼容性、更长的寿命、更好的可扩展性和更高的性能，但在一定程度上也限制了工程上的自由发挥。这种技术保证了兼容性，但也使新一代的产品必须拥有上代产品的所有能力，这样才能让今天的PTX代码在未来的系统上仍然可以运行。

编译器前端按照C++语法规则对CUDA源文件进行处理。CUDA主机端代码可以支持完整的C++语法，而设备端代码则不能完全支持。

内核函数可以通过PTX编写，但通常还是通过CUDA C一类的高级语言进行编写。PTX或CUDA C语言编写的内核函数都必须通过nvcc编译器编译成二进制代码。一部分PTX指令只能在拥有较高计算能力的硬件上执行。nvcc通过-arch编译选项来指定要输出的PTX代码的计算能力。

在程序编译时，要使目标代码和目标硬件版本与实际使用的硬件一致，可以使用-arch、-gencode和-code编译选项。

关于nvcc编译选项的更详细信息可以参考：C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v7.5\doc\html\cuda-compiler-driver-nvcc

以上部分内容整理自：《GPU高性能运算之CUDA》、《GPU高性能编程CUDA实战》

GitHub：https://github.com/fengbingchun/CUDA_Test

顶 0 踩 0

上一篇 一维码UPC E简介及其解码实现(zxing-cpp)

我的同类文章

CUDA（10）

- windows7 64位机上配置支… 2016-12-27 阅读 111
- windows7 64位机上安装配… 2016-12-27 阅读 506
- Ubuntu14.04 64位机上安装… 2016-12-23 阅读 186
- Ubuntu14.04 64位机上安装… 2016-12-23
- 《GPU高性能编程CUDA实… 2015-05-24 阅读 5077
- CUDA Runtime API 汇总 2015-04-26 阅读 2495
- windows7 64位机上CUDA7.… 2015-04-09 阅读 17677
- GPU及GPU通用计算编程模… 2014-02-21 阅读 4017
- Windows7 32位机上，Open… 2013-08-08 阅读 17385
- OpenCV中GPU模块(CUDA)… 2013-07-25 阅读 8670

猜你在找

Python编程基础视频教程(第二季)
从此不求人:自主研发一套PHP前端开发框架
韦东山嵌入式Linux第一期视频
Swift视频教程(第三季)
Swift视频教程(第四季)
vim
Convolutional Networks for MNIST in Tensorflow
opencv30分析
开源的物理引擎
Ubuntu1604+matlab2014a+anaconda2+OpenCV31+caf

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

京 ICP 证 09002463 号 | Copyright © 1999-2016, CSDN.NET, All Rights Reserved