CUDA编程模型综述

- 吕文龙
- 14210720082

摘要

介绍了通用GPU编程以及CUDA出现的背景, CUDA C语言语法以及CUDA在GPU上的执行模型. 介绍了最新版本CUDA的一些特性以及基于CUDA的并行计算扩展thrust库. 介绍了用于profile CUDA程序的工具. 并以一个蒙特卡洛算法为例, 展示了GPU编程的加速比.

背景介绍: GPU, GPGPU 与 CUDA

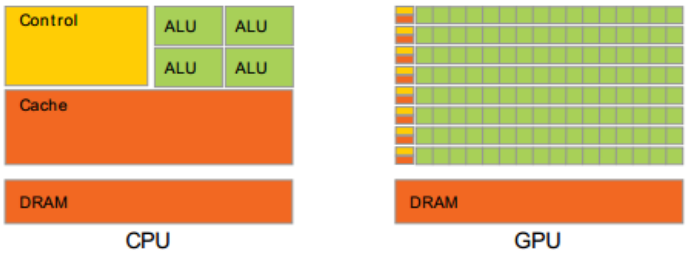
GPU 与 CPU 特性对比

GPU最初是作为CPU的协处理器, 为了加速图形计算而设计的. 图形计算要求对大量像素点的重复高密度计算, 此种需求造成了CPU与GPU在设计时侧重点的差异.

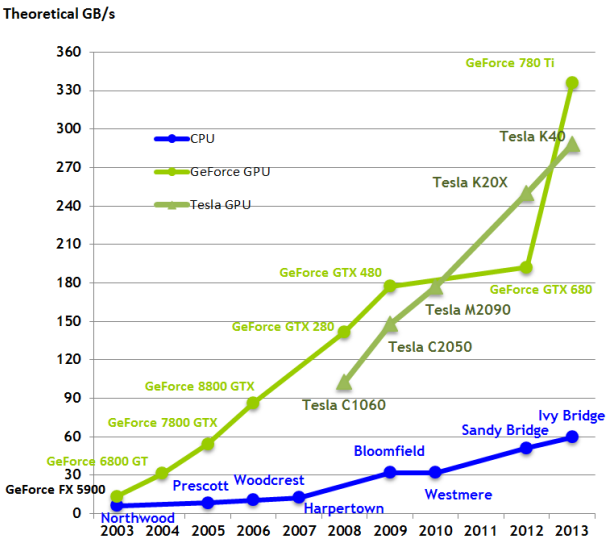
GPU相对CPU主要有以下三点优势:

- 更强的浮点数计算能力(与之对应的是更弱的逻辑控制功能)
- 更大的内存带宽
- 轻量级的线程切换

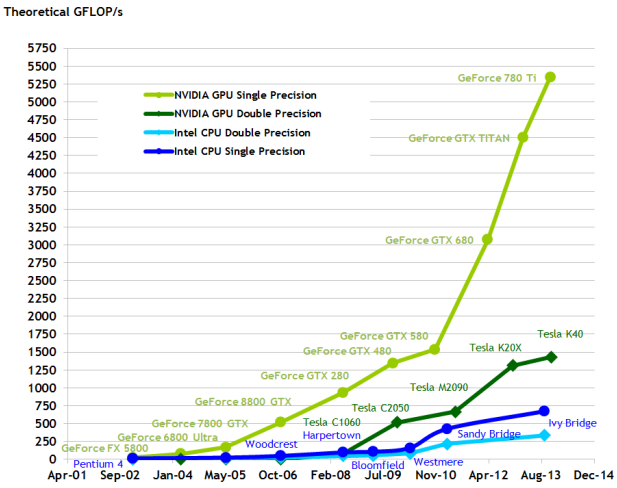
具体而言, 相比GPU, CPU需要处理更加通用的, 普适性的任务. 因而在设计时, CPU更侧重于逻辑操作, 会将大量的硬件用于控制逻辑以及缓存, 而GPU则更侧重高密度的浮点数运算. 因此在硬件设计上, 会更倾向于添加大量并行的运算单元,下图为CPU与GPU的硬件组成对比图:



设计上的差异, 使得在浮点数处理方面, 同时代的GPU往往远胜CPU, 下图为CPU-GPU的浮点数处理能力(GFLOPS)对比:



下图为不同时代CPU与GPU的内存带宽对比:



CPU的一个核心(或者通过超线程技术虚拟将一个核心虚拟成的多个核心). 同一时刻通常只能运行一个线程的指令, 当线程数超过核心数时, 多个核心共享计算资源, 而当进行线程切换时, 必须要花费大量的clock cycle来保存上下文.

而GPU上具有大量的核心, 并且能够通过硬件来管理线程, 可以实现零开销的线程切换. 当一个线程因为访存或者线程同步而等待时, 可以在下一个时钟周期立刻切换到准备计算的线程.

目前的GPU上没有很复杂的缓存机制, 也没有为单个计算单元设计复杂的流水线等指令级并行机制(至少目前为止这一点仍然是成立的, 不过随着GPU应用领域不再局限于图形处理方面而转向各种通用计算领域, GPU上应该也会出现强大的缓存与 Instruction level parallism 机制). 它主要通过零开销的线程切换来隐藏延迟. 正式由于这个特性, 在后面测试的例子中可以看到, 要使GPU达到加速比, GPU上同时运行的线程应该是数倍于GPU的计算单元(SP).

从 GPGPU 到 CUDA

随着GPU性能与可编程性的不断提高, GPU开始被用于图形处理之外的通用领域, 通过CPU+GPU异构编程的方式来进行并行计算, 即CPU负责控制逻辑, 而将大规模的数据计算交给GPU来处理. 使用GPU来进行数据并行的运算, 能够极大的提高性能/功耗比. Green500网站公布的绿色超级计算机排名中, 性能-功耗比排名考前的超级计算机中, 大多都使用了GPU作为协处理器进行加速, 下图为2014年11月公布的最新绿色超级计算机前十位, 其中, 第一名使用了AMD的GPU进行加速, 3-10都使用了NVIDIA的GPU进行加速:

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	5,271.81	GSI Helmholtz Center	L-CSC - ASUS ESC4000 FDR/G2S, Intel Xeon E5-2690v2 10C 3GHz, Infiniband FDR, AMD FirePro S9150 Level 1 measurement data available	57.15
2	4,945.63	High Energy Accelerator Research Organization /KEK	Suiren - ExaScaler 32U256SC Cluster, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, PEZY-SC	37.83
3	4,447.58	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x	35.39
4	3,962.73	Cray Inc.	Storm1 - Cray CS-Storm, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, Nvidia K40m Level 3 measurement data available	44.54
5	3,631.70	Cambridge University	Wilkes - Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20	52.62
6	3,543.32	Financial Institution	iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x	54.60
7	3,517.84	Center for Computational Sciences, University of Tsukuba	HA-PACS TCA - Cray CS300 Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, NVIDIA K20x	78.77
8	3,459.46	SURFsara	Cartesius Accelerator Island - Bullx B515 cluster, Intel Xeon E5-2450v2 8C 2.5GHz, InfiniBand 4× FDR, Nvidia K40m	44.40
9	3,185.91	Swiss National Supercomputing Centre (CSCS)	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Level 3 measurement data available	1,753.66
10	3,131.06	ROMEO HPC Center - Champagne-Ardenne	romeo - Bull R421-E3 Cluster, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR, NVIDIA K20x	81.41

GPGPU使得GPU的应用不再局限于图形处理领域, 但是, 最早的GPGPU开发需要直接使用图形学API, 要求编程人员将数据打包成纹理, 将计算任务映射为对纹理的渲染过程, 这种方式要求程序员对图形学硬件与编程接口有深入了解, 开发难度很大.

2007年,NVIDIA推出了CUDA(Compute Unified Device Architecture, 统一计算设备架构). CUDA不需要借助图形学API, 并采用了类C语言进行开发. 开发者能够从熟悉的C语言比较平稳的从CPU过渡到GPU, 而不必重新学习语法.

同时, 与以往的GPU相比, 支持CUDA的GPU在架构上有了显著的改进, 使得CUDA架构更加适用于GPU通用计算. 例如, 引入了片内共享存储器, 使得线程间可以通过共享存储器进行通信.

CUDA只支持NVIDIA公司的GPU, 目前, 最新的CUDA的版本号为6.5

CUDA 编程模型

主机端与设备端

CUDA为CPU-GPU异构编程, 程序的执行环境分为主机端(host)与设备端(device), 主机端即CPU, 设备端即GPU. 程序由主机端的main函数开始, 从主机端调用设备端的程序. 一个系统中, 可以有一个主机和多个设备. 主机与设备端拥有各自独立的存储器(内存与显存).

CUDA通过扩展的C语法编写程序, 支持三种函数, `host`函数,`global`函数与`device`函数. 函数默认为host函数, 即CPU上运行的代码, 这部分程序经由CUDA识别后, 交给传统的C编译器如`gcc`进行编译, 而`global`函数和`device`函数是运行在GPU上的程序, `global`为CPU调用的GPU函数, 而`device`函数则为GPU调用的GPU函数.

线程层次模型 block, thread, warp

CUDA GPU中, 基本的串行执行单位称为线程(thread), 一定数量的线程被组织为一个线程块(block), 线程块的尺寸与维度由程序员确定, 即可以通过一维与两维索引(threadIdx)来索引线程块中的线程. 由程序员确保一个线程块中的线程能够乱序执行.

一定数量的线程块又被组织为一个grid, 同样的, 也可以通过多维索引(blockIdx)来索引一个grid中的block.

之所以使用两级线程组织, 是为了支持多种并行方式, 即在block之间的任务级(task level)block内线程的数据级(data level)并行. 在一个线程块中的线程在执行时会被分配到同一个Stream Multiprocessor上, 因此能够较为方便的以较小的开销进行通信与同步. 即通过片上共享内存进行通信, 通过`_syncthreads()`函数进行同步等, 而不同的block之间没有内置的通信机制, 只能够通过全局显存进行通信.

在理论上, 同一个block内的线程之间可以执行完全不同的代码, 即通过threadIdx为索引到的各个线程分配不同的任务. 但实际的执

行模型中, 还有warp这个概念, 一个warp是指一个block内线程号连续的一定数量的一束线程. 在截至目前的CUDA版本中, 一个warp有32个线程. GPU中的指令以warp为单位发射, 同一时刻, 一个warp中的线程总是执行相同的指令. 如果一个warp中的线程指令不同, 则warp中的每条指令都会被所有线程执行一遍. 后面会讲到, CUDA的SIMT(single instruction multi thread)模型, 其实仍然是SIMD的一种变种.

关于warp, 有两点结论, 第一是同一个warp内的线程(可以通过threadIdx % 32获得warp号)是不需要同步的, 因为它们在硬件层面强制同步, 程序员大可不必为属于同一个warp的两个线程设置同步机制, 第二是如果一个warp中的线程指令差异很大, 例如两个线程的程序完全不同或者线程有太多的条件控制跳转逻辑等. 就会非常影响执行效率.

需要注意的是, warp对程序员是透明的, 它并不是编程模型中的概念, 而是执行模型中的概念, 无视warp并不会影响程序的正确性, 而只是影响效率而已. 这点类似于传统CPU编程中cache的作用, cache对程序员也是透明的, 在C的模型中只有CPU与内存这两个概念, 并没有专门操作cache的机制, 然而, 如果代码不够cache-friendly, 会极大的影响程序速度. 反之, 如果程序对cache友好, 甚至可能会在多核环境下因为多核带来的更大cache而使程序达到超线性加速比.

CUDA 存储器模型

CUDA 执行模型

CUDA 新特性

thrust库

CUDA profiler

CUDA加速程序举例, 及加速比分析