CUDA模型综述

- 吕文龙
- 14210720082

1. 摘要

本文是一个对CUDA的综述, 首先介绍GPGPU及CUDA出现的背景, 重点介绍CUDA背后的各种模型, 包括编程模型, 内存模型, 硬件模型与指令执行模型. 简要介绍了CUDA C语言扩展以及最新版本的CUDA中的一些新特性, 并以本科毕业设计时的一个蒙特卡洛并行程序为例说明GPU所能达到的相对CPU的加速比. 本综述是建立在我本科的毕业设计所做的一些关于CUDA的调研与设计的基础之上.

2. 背景介绍: GPU, GPGPU 与 CUDA

2.1 GPU 与 CPU 特性对比

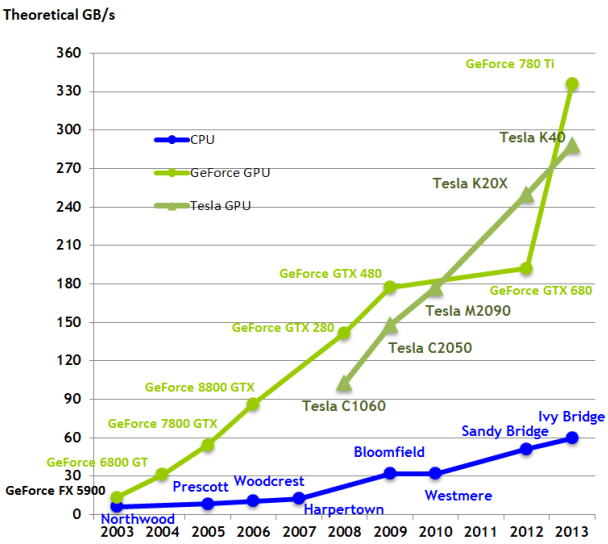
GPU最初是作为CPU的协处理器, 为了加速图形计算而设计的. 图形计算要求对大量像素点的重复高密度计算, 此种需求造成了CPU与GPU在设计时侧重点的差异.

GPU相对CPU主要有以下三点优势:

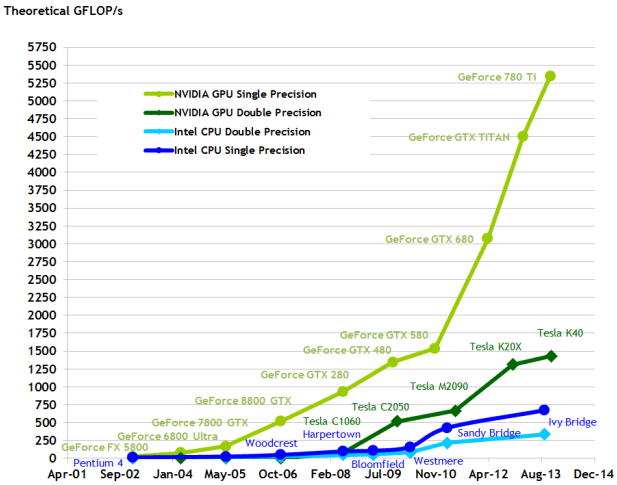
- 更强的浮点数计算能力(与之对应的是更弱的逻辑控制功能)
- 更大的内存带宽
- 轻量级的线程切换

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).

2.2 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

3. CUDA 编程模型

3.1 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.

运行在GPU上的`global`函数称为kernel(内核函数), CUDA C扩展了C的语法, 允许程序员用C语言定义在GPU上运行的函数. 内核函数通过`__global__`限定符定义, 在GPU上执行的线程层次与数量通过`<<<...>>>`语法配置.下面是一个简单的kernel函数定义与调用的例子:

```
// Kernel definition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    //...
    // Kernel invocation with N threads
    vecAdd<<<1, N>>>(A, B, C);
    //...
}
```

host与device端的代码都是用与C语言语法相似的CUDA C编写, 后面会简要介绍CUDA C的语法.

3.2 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.

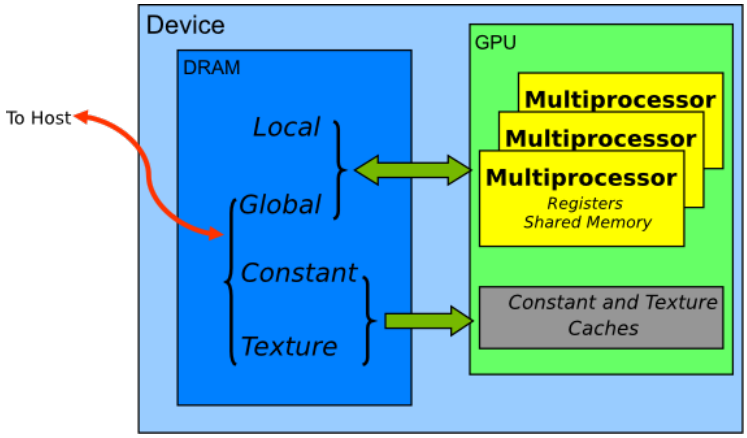
在理论上, 同一个block内的线程之间可以执行完全不同的代码, 即通过threadIdx为索引到的各个线程分配不同的任务. 但实际的执行模型中, 还有warp这个概念, 一个warp是指一个block内线程号连续的一定数量的一束线程. 在截至目前的CUDA版本中, 一个warp有32个线程. GPU中的指令以warp为单位发射, 同一时刻, 一个warp中的线程总是执行相同的指令. 如果一个warp中的线程指令不同, 则warp中的每条指令都会被所有线程执行一遍. 后面会讲到, CUDA的SIMT(single instruction multi thread)模型, 其实仍然是SIMD的一种变种.

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.

4. CUDA 存储器模型

下图为CUDA的内存模型示意图. 可以看出, CUDA模型中的内存分为GPU内部的片上存储器和片外显存.



CUDA采用多级存储器模型, 每个线程拥有自己的寄存器(register)和私有的局部内存(local memory), 每个block拥有对block内的线程可见的共享内存(shared memory), 用于线程之间的数据同步. 所有的线程, 都可以对全局内存进行访问.

为了节约内存带宽, CUDA 之中还有特殊设计的纹理内存(texture memory)和常量内存(constant memory). 其中, 寄存器与共享内存是GPU上的片内存储器, 访存速度较快, 其他类型的存储器则位于片外的存储器上, 具有较高的访存延迟. 不过CUDA可以通过零开销的线程切换来隐藏延迟.

需要注意的一点是, 线程私有的局部内存(local memory)并不是片上内存, 而是在显存上分出的一块只对某个线程可见的内存, 因此, 对local memory的访问开销很大.

4.1 寄存器与局部内存

寄存器(register)是GPU的片上高速缓存, 执行单元可以以极低的延迟访问寄存器. 寄存器有很高的带宽. 寄存器数量虽然可观, 但会平均分给并行执行的线程, 因而当线程数较多时每个线程拥有的寄存器数量就非常有限了.

当寄存器数量不够, 比如线程定义了太多私有变量或者在线程内定义了大数组时, 变量就会被分配到local

对每个线程而言, 局部存储器(local memory)也是私有的. 当寄存器被消耗完, 线程的数据就会被存储在局部存储器中. 局部存储器并不是片上存储器, 而是位于GPU之外的显存上, 因此, 对局部存储器的访问速度很慢. 因此, CUDA程序设计时, 应对线程的私有变量大小进行预判, 不应为线程分配过多的私有变量, 应尽量避免因寄存器数量不够而将变量分配到局部内存上. 如前所述, local memory位于片外显存, 访问开销很大, 因此, 不应该分配过多线程或者在单个线程中定义太多私有变量.

4.2 块内共享内存

共享存储器(shared memory)也是GPU片内高速存储器, 它是一块可以白同一block中所有的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的公用结构.

共享存储器可以静态分配, 也可以动态分配, 如果动态分配, 则共享存储器的大小需要在kernel中用`extern`声明.

下面是一个通过共享存储器进行数组求和的程序例子:

```
__global__ void sumReduction(float *sum)
{
    extern __shared__ float cache[];
    int idx    = threadIdx.x;
    cache[idx] = 0;
    /*
     * ... calc cache[idx] for each thread
     */
    __syncthreads();
    int i=blockDim.x/2;//要求blockDim.x为2的幂
```

```
    while(i != 0)
    {
        if(idx<i)
            cache[idx]+=cache[idx+i];
        __syncthreads(); //如果线程数小于32(一个warp)，则不必同步
        i/=2;
    }
    *sum=cache[0];
}
```

4.3 全局内存

全局存储器(global memory)位于片外显存, 主机端/设备端均可以进行读写. 任意线程都能读写全局存储器的任意位置. 全局存储器能够提供很高的带宽, 但同时也有较高的访存延迟.

对全局存储器的访问, 可以通过CUDA提供的运行时API或驱动API来实现, 使用CUDA C的关键字**device**定义的变量也会分配到全局存储器.

常用的CPU-GPU互动方式就是, CPU分配显存, 然后调用kernel函数, kernel函数运行完成后, 再从显存copy数据到CPU内存.

4.4 常量内存与纹理内存

常量内存(constant memory)是只读的地址空间, 位于片外显存, 但拥有缓存加速. 常数存储器空间较小, 在CUDA程序中用于存储需要频繁访问的只读参数.用以节约带宽.

纹理存储器(texture)也是只读存储器, 主要就是用于图像编程当中的纹理渲染等作用, 也可以称之为图像处理的专门单元所设置的一种存储器. 它主要存储数据的模式是以数组的形式存储在显存当中的. 这些数组包含了一维, 二维以及三维. 但是它所能声明的数组的大小要比常量存储器大的多, 而且也具有缓存加速的功能, 多被用于图像处理, 在查找表中也有着广泛的使用. 所以, 图像编程过程中经常的被用于数据量比较大的访问, 这些访问包含了对齐及非对齐的, 以及随机的数据.

5. CUDA硬件映射

支持CUDA的GPU中, 一个具有完整的取指, 译码, 发射, 执行功能的处理单元被称为一个流多处理器(SM, Stream Multiprocessor), 下图为Kepler架构GPU的一个SM的框图. 每一个SM中有多个计算单元, 被称为流处理器(SP, Stream Processor), 每个SP中都包含浮点数与整数运算逻辑, 同时SM中还有大量双精度浮点数运算单元(DPU), 以及特殊的浮点数运算(如专门为计算平方根倒数运算)而设计的特殊浮点数运算单元(SFU). 不同架构的GPU中, 每个SM中的SP数量不同, Tesla架构中, 每个SM含有8个SP, Fermi架构中, 每个SM含有48个SP, Kepler架构中, 每个SM中含有192个SP.

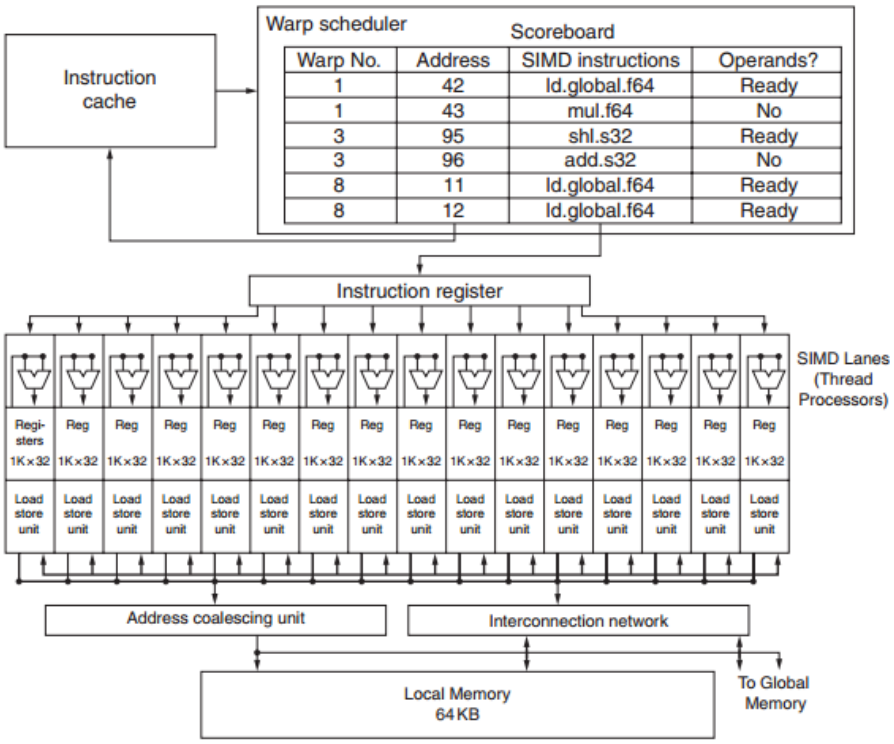
SMX: 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

在nvidia公司的商业宣传中, GPU往往被说成拥有数百个乃至上千个核, 这里的核通常指SP的数量, 而非SM的数量. 事实上, SP只是执行单元, 并不是完整的处理核心. 隶属同一SM的所有SP公用一套取指令与发射单元, 也公用一块共享存储器.

CUDA的kernel函数被配置为不同block并行执行, 同一个block中的线程需要进行通信与数据共享, 因此一个block会被映射到一个SM上执行, 而block中的每一个线程则被发射到一个SP上执行. 同一个SM中可以有多个活动线程块(active block)以隐藏延迟, 当一个block进行高延迟操作时, 另一个block可以占用GPU资源进行计算.

6. CUDA执行模型

下图为一个CUDA执行模型框图, 本节将介绍CUDA的SIMT执行模型, 以及它与传统的SIMD模型的区别与联系.



一个block中的thread, 每32个线程, 会被组织为一个warp. 一个warp内线程的具有相同的指令地址, 但是其中的每个线程拥有自己的指令地址计数器和寄存器状态, 因而能够独立的执行, 并能自由的分支.

当一个SM中有多个block等待执行时, 每个block中32个连续的线程被分为一个warp, 通过warp scheduler来调度执行. SM中的指令以warp为单位发射, 硬件发射逻辑会计算SM中warp的优先级, 当一条warp中的指令所需的资源都可用, 这条指令就被设为就绪态(ready), 每个GPU周期, 发射逻辑从指令缓冲中选取优先级最高的就绪指令进行发射.

因为每个线程的寄存器都是私有的, GPU上的线程切换并不需要像CPU上线程切换那样为保存上下文而付出昂贵代价, 相反, 从一个执行上下文切换到另一个执行上下文没有消耗, 当一个warp的程序需要等待显存资源时, SM在下一次指令发射时可以切换到准备就绪的warp.

事实上, 一个warp的行为更像是传统CPU多线程编程之中一个线程的概念, 只不过线程的指令为SIMD指令. 因而, 在第五版的<计算机体系结构——量化研究方法>中, 一个SM也可以被称为一个**多线程的SIMD处理器**.

CUDA的执行模型被称为SIMT(Single Instruction, Multiple Thread, 单指令多线程), nvidia公司用这个术语来将GPU的体系结构与SIMD架构进行区分, SIMT模型与SIMD模型相比, 主要在两方面更加灵活.

第一, warp这一概念, 也就是SIMD指令执行宽度, 被作为硬件细节隐藏, 而不在编程模型之中,开发者无需显式的把数据凑成合适的宽度. 硬件可以自动适应不同的执行宽度, 如在Kepler架构的设备上, 一个block由若干个warp组成, 执行宽度可以在1~1024个线程之间变化. 开发者无需显式的把数据凑成合适的宽度.

第二, SIMT模型强调程序员可以为一个warp内的每一个线程灵活地指派不同的分支路径与控制流. 而单纯的SIMD无法为处理同一条指令中不同的条件跳转路径. 然而, 当warp被执行时, 任何时候一个warp内的线程总是在执行相同的指令, CUDA的指令是以warp为单位调度, 发射, 执行的, 同一个warp内的线程总是处于同步状态的, 因此无需使用同步函数显式同步. 如果一个warp内的线程按照不同的分支路径执行, 此时, 一个warp内的每个分支路径都会被执行一次, warp内不在分支路径上的线程, 则会被暂时无效化, 执行时间会是执行多个分支所用的时间之和, 效率可能会受到很大影响.

因此, 可以这样认为, SIMT模型是对SIMD模型的改进与封装, 使程序员能够更加灵活的的进行编程. 指令在执行时, 仍然是按照SIMD的方式进行执行, 在对CUDA程序进行优化时, 应当注意到这一点

7. CUDA C语法

CUDA提供了一个在C/C++子集上的扩展. 以下介绍CUDA C的基本概念.

7.1 函数类型限定符

CUDA C的函数包括host, global, device 函数, CUDA C使用函数类型限定符来指明函数属于哪一种函数, 应该运行在何种设备之上, 可以被何种函数所调用.

- __device__表明函数是device函数, 只能在设备端执行, 且只能被设备端程序调用.
- __global__声明只能在设备端执行, 且只能从主机端调用的函数. __global__ 函数的返回类型必须为void
- __host__ 函数限定符用于声明在主机端执行, 且只能从主机端调用的函数, 没有限定符修饰的函数, 等同于只有__host__限定符修饰的函数.

__host__ 限定符可以与 __device__ 限定符一起使用, 此时, 函数将为主机端和设备端分别进行编译.

7.2 变量类型限定符

CUDA C中支持通过变量类型限定符声明变量在GPU中存储的位置:

- __device__ 变量限定符用于声明的变量存在于设备端, 当__device__限定符不与其他限定符连用时, 表示变量为存储在全局内存中的全局变量.
- __constant__ 变量限定符声明的变量位于常数存储器.
- __shared__ 限定符声明的变量位于block内的共享存储器中, 仅可通过block内的所有线程访问.

7.3 内置变量与内置类型

CUDA中也内置了一些向量类型, 用于表示两维或者三维的数据. 例如dim3类型, 常用于指定block或则grid在三个维度上的size, 而unit3则是三维整数类型, 可以通过var.x, var.y, var.z来索引, 通常用于索引三维线程块或者三维grid中的线程或者线程块. CUDA中定义了一些uint3或者dim3类型的内置变量, 用于表示线程块或grid的尺寸以对block和thread进行索引.

1. gridDim为dim3类型变量, 包含grid在三个维度上的尺寸信息, 可以通过gridDim.x/gridDim.y/gridDim.z来索引
2. blockIdx为uint3类型变量, 包含一个block在grid中的坐标.
3. blockDim为dim3类型变量, 包含了block在三个维度上的尺寸信息
4. threadIdx为uint3类型变量, 包含了一个thread在block中各个维度上的坐标.
5. warpSize为int类型, 用于确定一个warp包含多少个thread.

7.4 执行配置

使用执行配置来确定block中的线程以及grid中的block的维度以及各个维度上的尺寸. 对 __global__函数的任何调用都必须指定该调用的执行配置, 以定义在GPU上执行时grid和block各个维度的尺寸信息.
例如, 如下定义的**global**函数,:

```
__global__ void MyTest (int* a){
/* … */
}
```

可以以如下形式进行调用:

```
MyTest<<<Dg,Db,Ns>>>(dev_a);
```

其中:

- Dg为dim3类型变量, 用于设置grid的维度和各个维度上的尺寸, 设置好Dg后, grid中将会有Dg.x * Dg.y个block, 也可以用int类型进行配置, 此时, block将为一维.
- Db为dim3类型变量, 设置block的维度和各个维度上的尺寸.
- Ns为一个size_t类型的变量, 用于动态分配共享内存的大小.

7.5 线程同步

CUDA中没有全局线程同步函数, 对于CPU-GPU之间同步, 可以通过`cudaThreadSynchronize()`函数, 该函数使CPU等待GPU上的kernel函数执行完成之后,才继续执行. 对于同一个block内的线程, 可以通过`_syncthreads()`函数进行同步.

8. Dynamic Parallism

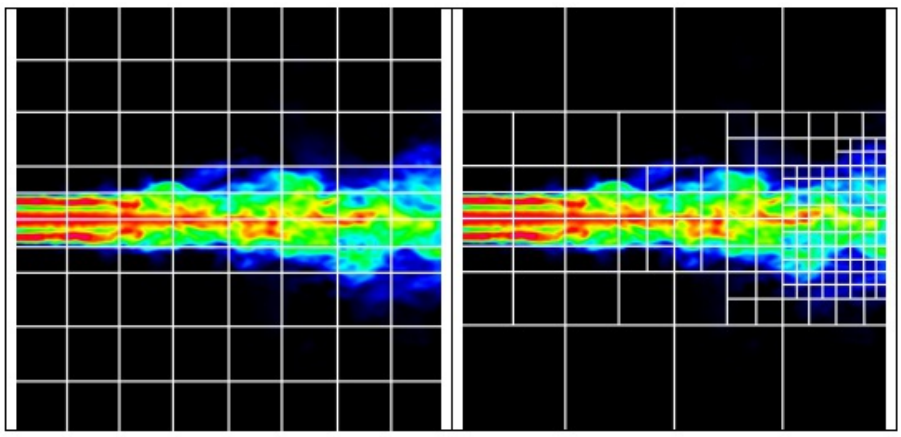
Dynamic Parallism 是CUDA 5.0版本开始支持的一项特性, 因此在之前的介绍中没有提到. 在这里简要介绍一下, 因为自己的笔记本电脑GPU计算能力(计算能力类似GPU的版本号, 并非GPU性能度量)过低, 无法支持Dynamic Parallism, 因此以下介绍只是根据查阅资料而来:

之前介绍kernel函数时提到, kernel函数是由CPU端调用, 在GPU上运行, 也就是说, 应当由CPU事先对要计算的数据进行划分, 确定并行方式, 然后调用kernel函数. 然而, 要计算的数据分布往往并不均匀, 因而, 数据集的某些部分计算量大, 而有些部分计算量小, 例如稀疏矩阵. 因而CPU来实现确定如何划分并行网格是很困难的, 而采用均匀网格则会浪费计算能力.

而支持Dynamic Parallism的GPU则允许从kernel函数内部调用新的kernel 函数, 根据father kernel的计算情况, 动态的为child kernel划分block与thread. 则可以不经由CPU而自适应的划分并行网格.
Dynamic Parallism的代码示例如下:

```
__global__ childKernel(void* data)
{
    //operate on data
}
__global__ parentKernel(void* data)
{
    //... some operation
    if(threadIdx.x == 0)
    {
        childKernel<<<1,32>>>(data);
        cudaThreadSynchronize();
    }
    __syncthreads();
}
int main()
{
    //... some operation
    parentKernel<<<8,32>>>(data);
    return 0;
}
```

一个形象的图示如下图所示:



上图是一个流体动力学仿真过程, 支持Dynamic Parallism的GPU, 可以通过仿真运行时的动态结果自适应的重新划分网格, 在不需要大量计算的位置投入较少线程, 而将大量线程投入计算量大的位置.

Dynamic Parallism之前, kernel函数是无法支持递归的, 而在Dynamic Parallism支持下, kernel函数是可以递归的. 下面的代码是可

以支持的

```
__global__ Recursivekernel(void* data){
    if(continueRecursion == true)
        RecursiveKernel<<<64, 16>>>(data);
}
```

9. 实例加速比分析

以我本人本科毕业设计时设计的蒙特卡洛法求解微分方程的程序为例, 分析CUDA程序所能达到的加速比.

9.1 求解原理

程序为基于Feynman-Kac公式, 通过WOS过程模拟粒子随机游走, 求解laplace方程, 程序的基本思路是每一个线程代表一个随机游走的粒子, 粒子到达边界后以边值加权, 即可得到满足拉普拉斯方程的函数在粒子初始点出的值. 可以看出, 不同线程之间, 只在最后求和时需要同步, 其余部分都是完全并行的.

9.2 测试环境

测试环境配置如下图:

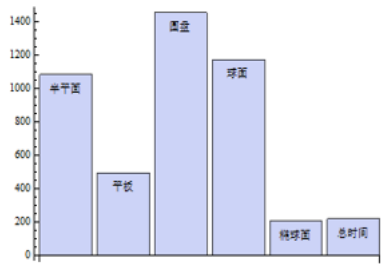
平台二: 上海交大PI 超级计算机

CPU	Intel Xeon E5-2206 @2.6GHz
GPU	Tesla K20M
GPU计算能力	3.5
SM数	13
每个SM上SP数	192
总SP数	2496
GPU主频	705MHz
操作系统	x86_64 Red Hat Enterprise linux Server release 6.3(Santiago)

9.3 CPU-GPU 加速比

下图为CPU-GPU加速比统计图, 分别统计了求解器对不同边界条件laplace方程在CPU和GPU上的求解效率, CPU时间为在CPU上运行的单线程程序运行时间.

CPU主程序时间: 98.26秒
GPU主程序时间: 0.45秒



可以看到, 对于较为简单的边界条件, 具有2496个SP的GPU在主频仅为不到CPU的1/3的情况下, 仍然可以拿到上千倍的绝对加速比, 而复杂边界条件如椭圆, 仍然能够拿到近200倍的加速比.

参考文献

1. "CUDA C Programming guide." NVIDIA Corporation (2013).
2. The Green500 List
3. Hennessy, John L., and David A. Patterson. Computer architecture: a quantitative approach. Elsevier, 2012.
4. Yan Chanhao, Wei Cai, and Xuan Zeng. "A parallel method for solving Laplace equations with Dirichlet data using local boundary integral equations and random walks." SIAM Journal on Scientific Computing 35.4 (2013): B868-B889.