# CUDA GPU并行编程模型综述

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(本人的本科毕设程序) 为例, 展示了GPU编程的各种标准下的加速比(GPU-CPU加速比, GPU多线程-单线程加速比)

**关键词: GPU, CUDA, 并行计算, 加速比**

## 背景介绍: GPU, GPGPU, CUDA

### GPU与CPU对比

GPU是为了进行快速的图像与视频渲染而设计的处理器. 随着其性能的不断增强, 近年来开始被用来进行通用并行计算. GPU相比CPU之优势所在, 在于其强大的浮点数处理能力以及更高的内存带宽, 如图1.1和1.2可见, GPU在浮点数处理能力与内存带宽方面, 都远超同时代的CPU. 这是因为GPU最初是为处理图图形渲染任务而设计, 在这类应用中, CPU负责根据用户的输入和一定的规则确定下一帧需要显示哪些物体, 以及这些物体的位置, 而GPU绘制这些物体并进行显示. 为了满足实时高清图形任务的需求, GPU被设计成为具有大量浮点处理单元的众核处理器.

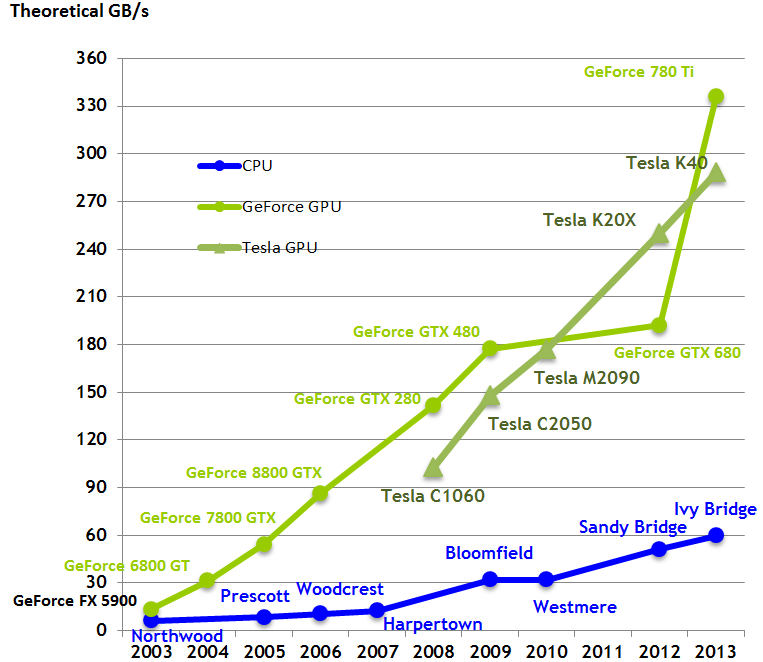


图1.1 CPU-GPU浮点处理能力对比[[[1]](#endnote-1)]

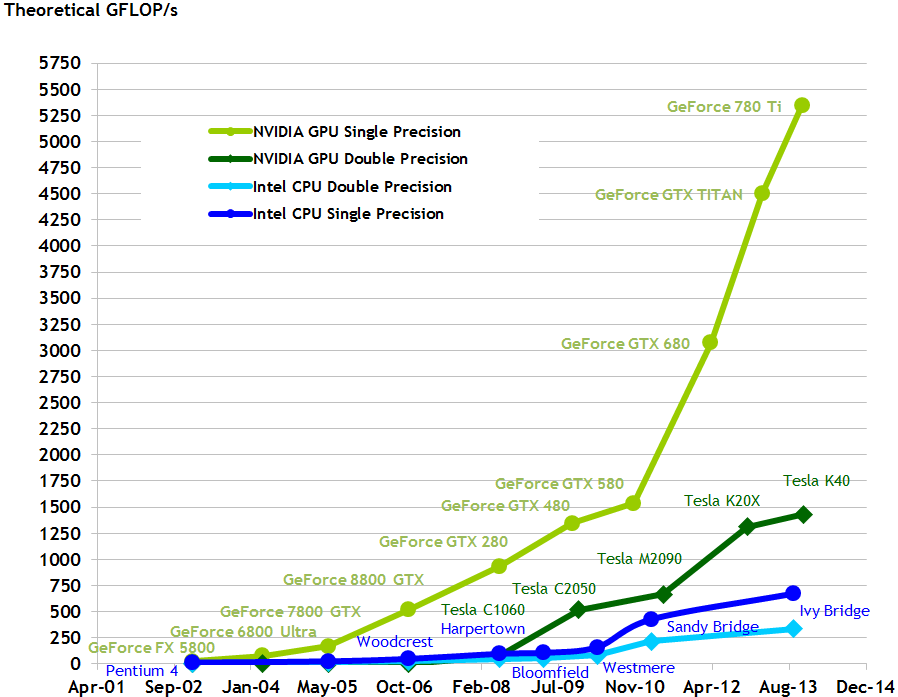


图1.2 CPU-GPU内存带宽对比[4]

在图形任务中, CPU负责的是逻辑性较强的计算, GPU则负责计算密集度高的图形渲染. CPU的设计目标是使执行单元能够以很低的延迟获得数据和指令, 因此采用了复杂的控制逻辑和分支预测, 以及大量的缓存来提高执行效率, 而GPU必须在有限的面积上实现很强的计算能力和很高的存储器带宽. 这使得CPU对延迟更敏感, 而GPU则侧重于提高整体的数据吞吐量. 这使得CPU与GPU在性能与架构上存在巨大差异. 如图1.3所示, CPU往往将更多的晶体管用作缓存与控制逻辑, 而GPU则将更多晶体管用于运算单元.

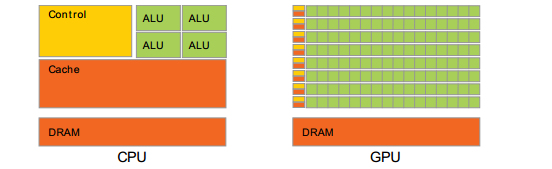


图1.3, CPU-GPU晶体管用途对比[4]

CPU的一个核心通常在一个时刻只能运行一个线程的指令, 其多线程机制通过操作系统提高能够的API执行, 当一个线程中断, 或者等待资源时, 操作系统要保存当前线程的上下文, 并切换到另一个线程的上下文. 这使得CPU的线程切换代价十分昂贵, 通常要数百个clock cycle. 某些CPU可以通过超线程技术将一个物理核心虚拟成多个核心, 以提高CPU的利用率, 但每个虚拟核心在一个时刻也只能运行一个线程. 而GPU则采用硬件管理的轻量级线程, 可以实现线程切换的零开销. 当一个线程因为访问片外存储器或者同步指令开始等待以后, 可以立即切换到另外一个就绪的线程, 用计算来隐藏延迟.当前主流的CPU中一般有2~8个核心, 每个核心中都采用了很多提高指令级并行的技术以及大容量的cache. 由于缓存和控制逻辑需要很大的面积, 一块芯片上无法继承很大数量的核心. 而GPU中则通常会集成数百乃至上千个核心. 以Nvidia Kepler架构的K20M为例, 一个芯片上含有13个包含完整前端的流多处理器(Stream Multiprocessor), 每个流多处理器包含192个流处理器(Stream Processor), 在多个流多处理器间粗粒度的任务级并行或数据级并行, 在流处理器细粒度数据并行. 由此, 尽管GPU的运行频率低于CPU, 但更多的执行单元数量还是使得GPU能够在浮点处理能力上获得优势. CPU中的缓存主要用于减小访存延迟和节约带宽, GPU中则没有复杂的缓存提议与替换机制.

由此可见, GPU设计目标是以大量线程实现面向吞吐量的数据并行, 适合处理计算密集度高/逻辑分支简单的大规模数据并行任务. CPU则有复杂的控制逻辑和大容量的缓存减小延迟, 能适应各种不同的情况, 尤其擅长复杂逻辑运算. 使用GPU处理数据并行任务, 而由CPU进行复杂逻辑和事务处理等串行计算, 就可以最大限度的利用计算机的计算能力.

### GPGPU: CPU-GPU异构并行

随着GPU性能与可编程性的不断提高, 用GPU来加速科学计算已经变得越来越普遍. 将GPU用于图形处理以外的任务称为GPGPU(General Purpose computing on Graphics Processing Units), GPGPU通常采用CPU+GPU异构模式, 由CPU负责不适合数据并行的串行部分与逻辑控制, 由GPU负责大规模数据并行. GPU异于CPU的架构, 使得它能够以更低的成本和更少的功耗完成更大量的数据计算, 在排名前500的超级计算机中, 有许多都使用了GPU作为协处理器加速. 而在2013年11月Green500网站公布的的绿色超级计算机排名中, 性能-功耗比排名前十的超级计算机, 都使用了GPU进行加速.

|  |  |  |  |
| --- | --- | --- | --- |
| 排名 | 性能/功耗  (Mflops/Watt) | 超级计算机名 | 加速器/协处理器 |
| 1 | 4503.18 | LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x | NVIDIA K20x |
| 2 | 3631.86 | Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20 | NVIDIA K20 |
| 3 | 3517.83674 | Cray 3623G4-SM Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, NVIDIA K20x | NVIDIA K20x |
| 4 | 3185.908329 | Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x | NVIDIA K20x |
| 5 | 3130.949603 | Bull R421-E3 Cluster, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR, NVIDIA K20xr | NVIDIA K20x |
| 6 | 3068.710855 | Cluster Platform SL390s G7, Xeon X5670 6C 2.930GHz, Infiniband QDR, NVIDIA K20x | NVIDIA K20x |
| 7 | 2702.15815 | iDataPlex DX360M4, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR14, NVIDIA K20x | NVIDIA K20x |
| 8 | 2629.102764 | iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x | NVIDIA K20x |
| 9 | 2629.09895 | iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x | NVIDIA K20x |
| 10 | 2358.689836 | Nitro G16 3GPU, Xeon E5-2650 8C 2.000GHz, Infiniband FDR, Nvidia K20m | NVIDIA K20m |

表1.1 Green500网站的性能功耗比排名前十的超级计算机及其GPU协处理器[[[2]](#endnote-2)]

传统GPU不允许计算单元之间通过片内存储器进行通讯, 因此GPGPU只能采用严格的SIMD模型, 同时, 最早的GPGPU开发直接使用图形学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, 增大了开发难度.

### 从GPGPU到CUDA

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软硬件体系结构简介

CUDA(Compute Unified Device Architecture, 统一计算设备架构)是nvidia公司于2006年推出的一种并行计算模型, 是一种利用GPU进行编程的软硬件体系. 与另一种CPU-GPU异构编程环境openCL不同, CUDA目前只支持nvidia公司的GPU.

相比于从前利用图形学API进行GPU通用计算的方法, CUDA编程不需要借助图形学API, 并采用了程序员比较熟悉的类C语言, 大大降低了开发者的学习成本.

CUDA可以通过操作系统的多线程API或者openMP实现多GPU并行, 让CPU上的每个线程管理一个GPU. 同时也可以与MPI连用, 实现集群或超级计算机上多节点多GPU并行计算.

CUDA使得开发者能够更加有效的利用GPU强大的性能, 因而自推出后, 被广泛的应用在各种科学计算领域, 在很多应用中, 都获得了很高的加速比.

### 2.1 CUDA 编程模型

CUDA编程模型提供了一个对并行计算的很好的抽象, 通过主机端与设备端两个概念划分程序中的串行与并行部分, 通过线程块-线程的两级并行使得不同粒度的并行都能得到支持. 同时, CUDA对GPU的抽象隐藏了许多硬件的实现细节, 这一方面使得程序员能够更加快速的编写出运行正确的代码, 但另一方面, 要想利用GPU达到最大的加速效果, 程序员还必须要熟悉CUDA的硬件与执行模型.

#### 2.1.2 主机, 设备及Kernel函数

CUDA编程模型将CPU作为主机(host), GPU作为设备(device), 在一个系统中, 可以存在一个主机和多个设备. GPU和CPU协同工作, 各司其职. CPU负责逻辑性较强的部分, 而GPU则作为协处理器, 执行由主机端分配的并行计算任务. 此外主机与设备还分别拥有各自的存储器. 如图2.1所示, 一个CUDA程序由一系列设备端的kernel函数和主机端串行执行的代码完成.

运行在GPU上的程序称为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. 内核函数只能从主机端(CPU端)调用, 在GPU上并行执行. 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);

//...

}

可以看出, 上面代码的语法与C语言的语法区别并不大, 只是在C的语法上进行了扩展. 其中\_\_global\_\_限定符表明vecAdd函数是一个由主机端调用的运行在设备端的函数. threadIdx是CUDA C的内置变量, 由于指明每一个并行线程的线程id, vecAdd<<<1,N>则是执行配置参数, 表明在GPU上并行的代码将以N个线程并行执行.

#### 2.1.3 线程层次结构

CUDA中的基本并行单位为CUDA线程(thread) , 一个线程中的代码会被GPU串行执行. 一定数量的线程被组织为一个线程块(block). 每个线程块中的线程数是确定的, 在CPU调用kernel函数时, 通过执行参数进行配置. 程序员应当保证一个block内的线程能够乱序执行, 但同一个线程块中的线程, 可以通过共享内存进行通信. 也可以通过内置函数\_\_syncthreads()进行同步. 线程在线程块中的id可以通过CUDA的内置变量threadIdx来索引. 线程可以是一维, 二维乃至三维的. 三个维度上的id分别通过threadIdx.x, threadIdx.y, threadIdx.y索引.

在编程模型上, 程序员可以通过线程id来为block中的线程指派各自的行为,但是在实际运行中, block会被分割为更小的线程束(warp), 在迄今为止的三代CUDA架构中, 一个warp的大小都是32个线程. Warp并不是CUDA编程模型中的概念, 但是如果同一warp内的线程行为有所差异, 就可能大大降低程序的执行效率. 这就如同传统编程中的cache, 无视cache存在并不会使程序结果错误, 但编写cache-friendly的代码往往能够显著提升程序运行速度.[4]

GPU中允许同时存在多个block, CUDA将一系列并行执行的block称为一个线程网格(grid). block乱序执行, block之间没有直接的通信机制. 这使得CUDA程序具有较好的可扩展性, 无论是只能处理一个block的GPU还是能够同时处理数十乃至上百个block的GPU, 这一编程模型都能很好的适用. Block可以是一维结构与二维机构, 分别通过blockIdx.x和blockIdx.y来索引两个维度上的id.

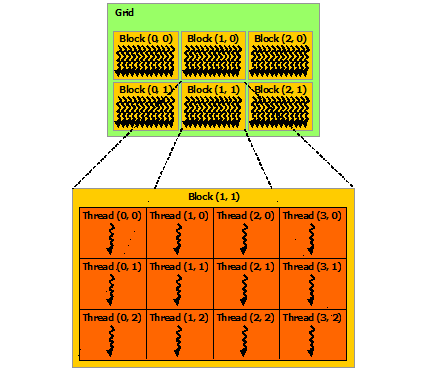
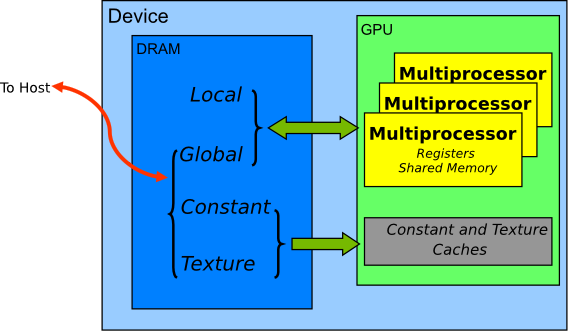


图2.1 CUDA线程层次[4]

如图2.1 CUDA通过两级的并行机制来支持不同粒度的并行机制, 不同block之间存在不需要通信的粗粒度并行, 同一block内的线程之间通过共享内存来支持需要通信的细粒度并行. 使得程序员能够方便的处理任务及并行与数据级并行.

### 2.2 CUDA存储器模型

如图2.2所示, 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内的线程块可见的共享内存(shared memory), 用于线程块内线程之间的数据同步. 同时, 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. 图2.2 CUDA的内存模型示意图[[[3]](#endnote-3)]

#### 2.2.1 线程私有内存----寄存器与局部存储器

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, 寄存器的基本单元是寄存器堆. 每个寄存器为32bit. 寄存器有很高的带宽.寄存器的数量是由GPU的计算能力决定的. 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.

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.

#### 2.2.2 线程块内共享存储器

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, 在kernel函数调用时使用<<<…>>>语法进行配置.

#### 2.2.3 全局存储器

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. 下面是一个简单的显存使用示例, 实现数组向量相加.

#### 2.2.4 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這染等作用, 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.

### 2.3 CUDA硬件映射与执行模型

#### 2.3.1 CUDA硬件映射

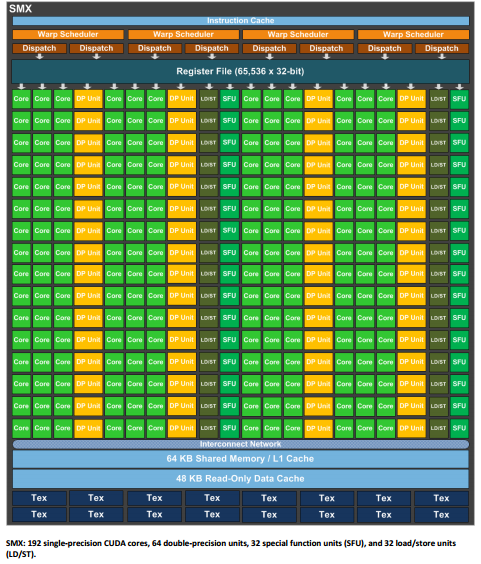


图2.3 Kepler GK110框图[[[4]](#endnote-4)]

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), 图2.3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(如平方根倒数)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. 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.

#### 2.3.2 CUDA执行模型

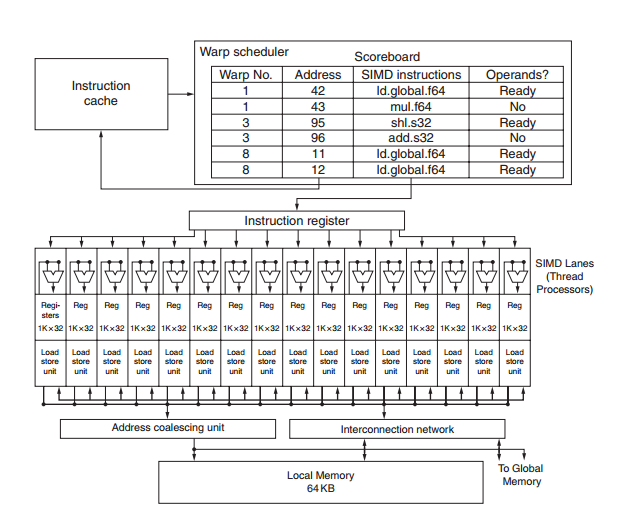


图2.4, 一个SM的执行模型框图[[[5]](#endnote-5)]

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, 只不过是一个由SIMD指令构成的线程. 因而, 一个SM也可以被称为一个多线程的SIMD处理器. 图2.4便是以这种观点来理解CUDA执行模型的一个SM执行框图.

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模型中, 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, 则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.

SIMT模型与SIMD的一个不同, 便是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.

### 2.4 CUDA C 语言扩展

CUDA提供了一个在C/C++子集上的扩展. 使得熟悉C/C++的开发者不需要太高的学习成本, 就可以进行CUDA C的开发, 以下只介绍其中比较基本或及于WOS算法GPU上实现相关的概念.

#### 2.4.1 函数类型限定符

函数类型限定符用来制定函数是在主机上还是设备上执行, 以及函数是否可以被主机端或设备端程序调用.

\_\_device\_\_ 限定符用于声明只能在设备端执行, 且只能被设备端程序调用的函数.

\_\_global\_\_ 函数限定符用于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.

#### 2.4.2 变量类型限定符

\_\_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. \_\_device\_\_, \_\_constant\_\_, \_\_shared\_\_不能由extern关键字声明为外部变量, \_\_shared\_\_前可以加extern关键字, 表明变量所占内存大小由执行参数确定.

#### 2.4.3 内建向量类型与内建变量

CUDA C中定义了许多有基本整数与浮点数派生二来的N元向量类型.

例如, dim3类型, 是基于用于指定维度的uint3的三维整数向量类型, 常用来指定线程和线程块的三维结构.

CUDA中还定义了各种内建变量来确定grid和block的组织维度的尺寸, 以及block在grid中的索引/thread在block中的索引.

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.

#### 2.4.4 执行配置

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);

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.

## CUDA新特性

### 动态并行

### 统一内存寻址

## CUDA 工具与扩展

### CUDA profiler

使用方法

### Thrust库

使用方法

## 加速比测试分析

### 测试环境

### 与单线程CPU程序对比

### 与GPU单线程程序对比

1. [] “CUDA C Programming guide." *NVIDIA Corporation* (2013). [↑](#endnote-ref-1)
2. [] The Green500 List <http://www.green500.org/?q=lists/green201311> [↑](#endnote-ref-2)
3. [] “CUDA C Best Practices Guide." *Nvidia Corporation,* (2012). [↑](#endnote-ref-3)
4. [] “NVIDIA’s next generation CUDA compute architecture: Kepler GK110. Whitepaper” [*http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf*](http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf) [↑](#endnote-ref-4)
5. [] Hennessy, John L., and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012. [↑](#endnote-ref-5)