



谭升的博客

人工智能基础



【CUDA 基础】3.2 理解线程束执行的本质(Part II)

📅 2018-03-15 | 📁 [CUDA](#) | [Freshman](#) | 💬 0 | 👁

Abstract: 本文介绍CUDA线程束执行的本质的后半部分，包括资源，延迟，同步，扩展性等严重影响性能的线，吞吐量，带宽，占用率，CUDA同步

理解线程束执行的本质(Part II)

最近这几篇应该是CUDA最核心的部分，并不是编程模型，而是执行模型，通过执行模型我们去了解GPU硬件的具体运行方式，这样才能保证我们写出更快更好的程序。

由于访问量太少，转载请保留本条广告，各位老铁欢迎访问Tony的网站：<http://www.face2ai.com>

资源分配

我们前面提到过，每个SM上执行的基本单位是线程束，也就是说，单指令通过指令调度器广播给某线程束的全部线程，这些线程同一时刻执行同一命令，当然也有分支情况，上一篇我们已经介绍了分支，这是执行的那部分，当然还有很多线程束没执行，那么这些没执行的线程束情况又如何呢？我给他们分成了两类，注意是我分的，不一定官方是不是这么讲。我们离开线程束内的角度（线程束内是观察线程行为，离开线程束我们就能观察线程束的行为了），一类是已经激活的，也就是说这类线程束其实已经在SM上准备就绪了，只是没轮到他执行，这时候他的状态叫做阻塞，还有一类可能分配到SM了，但是还没上到片上，这类我称之为未激活线程束。

而每个SM上有多少个线程束处于激活状态，取决于以下资源：

- 程序计数器
- 寄存器
- 共享内存

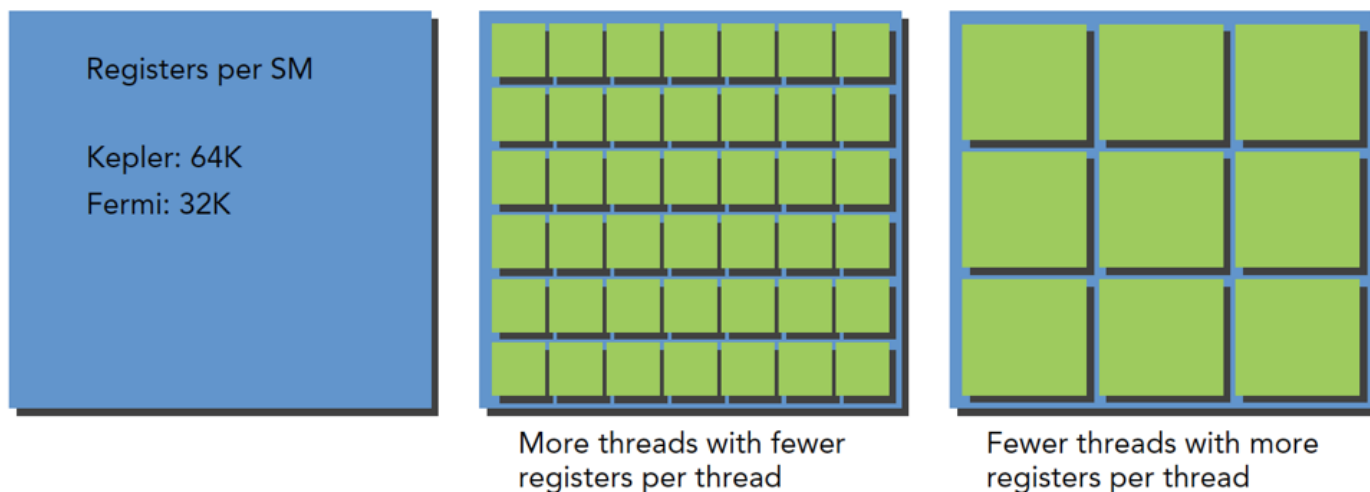
线程束一旦被激活来到片上，那么他就不会再离开SM直到执行结束。

每个SM都有32位的寄存器组，每个架构寄存器的数量不一样，其存储于寄存器文件中，为每个线程进行分配，同时，固定数量的共享内存，在线程块之间分配。

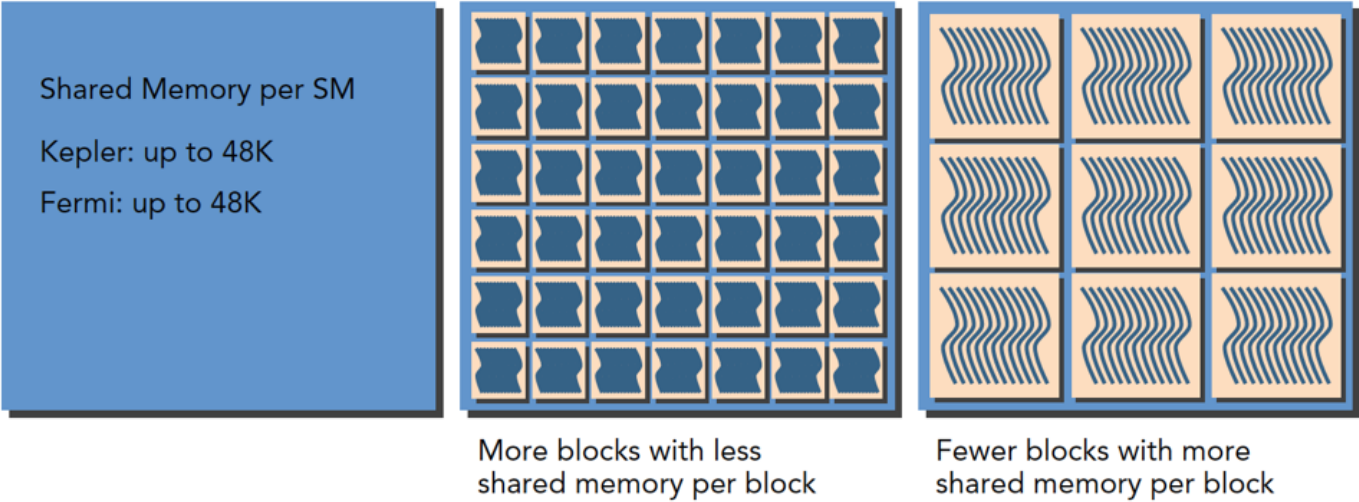
一个SM上被分配多少个线程块和线程束取决于SM中可用的寄存器和共享内存，以及内核需要的寄存器和共享内存大小。

这是一个平衡问题，就像一个固定大小的坑，能放多少萝卜取决于坑的大小和萝卜的大小，相比于一个大坑，小坑内可能放十个小萝卜，或者两个大萝卜，SM上资源也是，当kernel占用的资源较少，那么更多的线程（这是线程越多线程束也就越多）处于活跃状态，相反则线程越少。

关于寄存器资源的分配：



关于共享内存的分配：



上面讲的主要是线程束，如果从逻辑上来看线程块的话，可用资源的分配也会影响常驻线程块的数量。特别是当SM内的资源没办法处理一个完整块，那么程序将无法启动，这个是我们应该找找自己的毛病，你得把内核写的多大，或者一个块有多少线程，才能出现这种情况。

以下是资源列表：

TECHNICAL SPECIFICATIONS	COMPUTE CAPABILITY			
	2.0	2.1	3.0	3.5
Maximum number of threads per block	1,024			
Maximum number of concurrent blocks per multiprocessor	8		16	
Maximum number of concurrent warps per multiprocessor	48		64	
Maximum number of concurrent threads per multiprocessor	1,536		2,048	
Number of 32-bit registers per multiprocessor	32 K		64 K	
Maximum number of 32-bit registers per thread	63		255	
Maximum amount of shared memory per multiprocessor	48 K			

当寄存器和共享内存分配给了线程块，这个线程块处于活跃状态，所包含的线程束称为活跃线程束。

活跃的线程束又分为三类：

- 选定的线程束
- 阻塞的线程束
- 符合条件的线程束

当SM要执行某个线程束的时候，执行的这个线程束叫做选定的线程束，准备要执行的叫符合条件的线程束，如果线程束不符合条件还没准备好就是阻塞的线程束。

满足下面的要求，线程束才算是符合条件的：

- 32个CUDA核心可以用于执行
- 执行所需要的资源全部就位

Kepler活跃的线程束数量从开始到结束不得大于64，可以等于。

任何周期选定的线程束小于等于4。

由于计算资源是在线程束之间分配的，且线程束的整个生命周期都在片上，所以线程束的上下文切换是非常快速的，。

下面我们介绍如何通过大量的活跃的线程束切换来隐藏延迟

延迟隐藏

延迟隐藏，延迟是什么，就是当你让计算机帮你算一个东西的时候计算需要用的时间，举个宏观的例子，比如一个算法验证，你交给计算机，计算机会让某个特定的计算单元完成这个任务，共需要十分钟，而接下来这十分钟，你就要等待，等他算完了你才能计算下一个任务，那么这十分钟计算机的利用率有可能并不是100%，也就是说他的某些功能是空闲的，你就想能不能再跑一个同样的程序不同的数据（做过机器学习的这种情况不会陌生，大家都是同时跑好几个版本）然后你又让计算机跑，这时候你发现还没有完全利用完资源，于是有继续加任务给计算机，结果加到第十分钟了，已经加了十个了，你还没加完，但是第一个任务已经跑完了，如果你这时候停止加任务，等陆陆续续的你后面加的任务都跑完了共用时20分钟，共执行了10个任务，那么平均一个任务用时 $\frac{20}{10} = 2$ 分钟/任务。但是我们还有一种情况，因为任务还有很多，第十分钟你的第一个任务结束的时候你继续向你的计算机添加任务，那么这个循环将继续进行，那么第二十分钟你停止添加任务，等待第三十分钟所有任务执行完，那么平均每个任务的时间是： $\frac{30}{20} = 1.5$ 分钟/任务，如果一直添加下去， $\lim_{n \rightarrow \infty} \frac{n+10}{n} = 1$ 也就是极限速度，一分钟一个，隐藏了9分钟的延迟。

当然上面的另一个重要参数是每十分钟添加了10个任务，如果每十分钟共可以添加100个呢，那么二十分钟就可以执行100个，每个任务耗时： $\frac{20}{100} = 0.2$ 分钟/任务 三十分钟就是 $\frac{30}{200} = 0.15$ 如果一直添加下去， $\lim_{n \rightarrow \infty} \frac{n+10}{n \times 10} = 0.1$ 分钟/任务。

这是理想情况，有一个必须考虑的就是虽然你十分钟添加了100个任务，可是没准添加50个计算机就满载了，这样的话 极限速度只能是： $\lim_{n \rightarrow \infty} \frac{n+10}{n \times 5} = 0.2$ 分钟/任务 了。

所以最大化是要最大化硬件，尤其是计算部分的硬件满跑，都不闲着的情况下利用率是最高的，总有人闲

着，利用率就会低很多，即最大化功能单元的利用率。利用率与常驻线程束直接相关。

硬件中线程调度器负责调度线程束调度，当每时每刻都有可用的线程束供其调度，这时候可以达到计算资源的完全利用，以此来保证通过其他常驻线程束中发布其他指令的，可以隐藏每个指令的延迟。

与其他类型的编程相比，GPU的延迟隐藏及其重要。对于指令的延迟，通常分为两种：

- 算术指令
- 内存指令

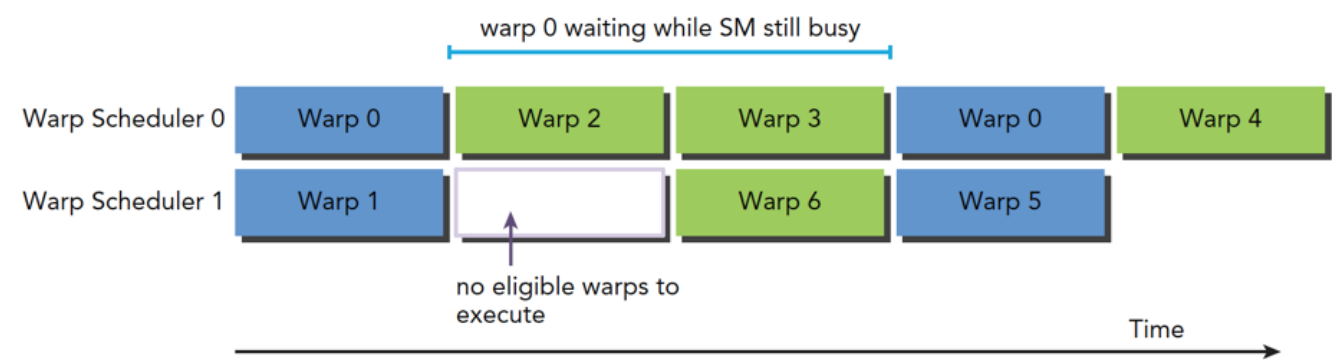
算数指令延迟是一个算术操作从开始，到产生结果之间的时间，这个时间段内只有某些计算单元处于工作状态，而其他逻辑计算单元处于空闲。

内存指令延迟很好理解，当产生内存访问的时候，计算单元要等数据从内存拿到寄存器，这个周期是非常长的。

延迟：

- 算术延迟 10-20 个时钟周期
- 内存延迟 400-800 个时钟周期

下图就是阻塞线程束到可选线程束的过程逻辑图：



其中线程束0在阻塞两短时间后恢复可选模式，但是在这段等待时间中，SM没有闲置。

那么至少需要多少线程，线程束来保证最小化延迟呢？

little法则给出了下面的计算公式

$$\text{所需线程束} = \text{延迟} \times \text{吞吐量}$$

注意带宽和吞吐量的区别，带宽一般指的是理论峰值，最大每个时钟周期能执行多少个指令，吞吐量是指实际操作过程中每分钟处理多少个指令。

这个可以想象成一个瀑布，像这样，绿箭头是线程束，只要线程束足够多，吞吐量是不会降低的：

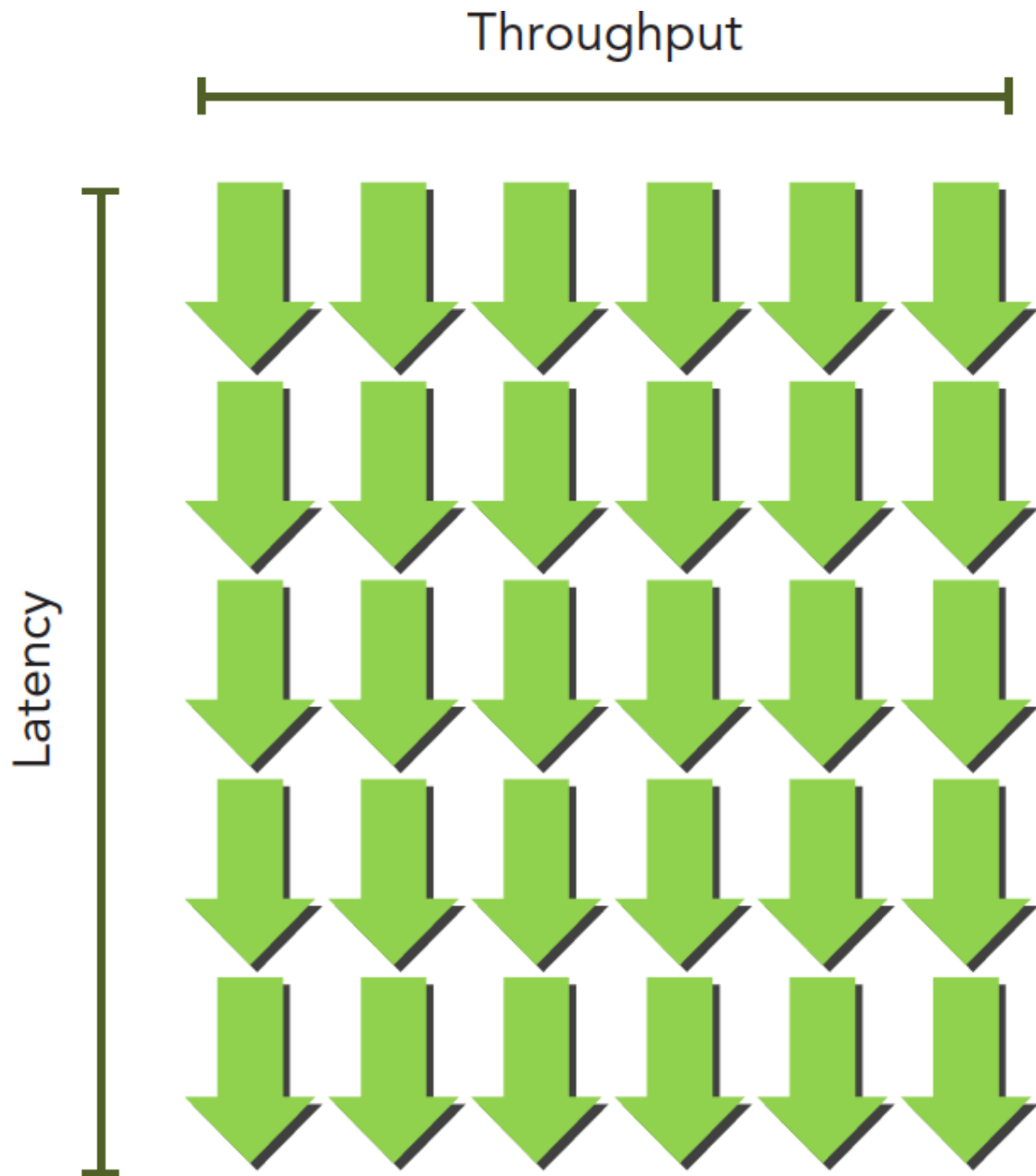


FIGURE 3-16

下面表格给出了Fermi 和Kepler执行某个简单计算时需要的并行操作数：

GPU MODEL	INSTRUCTION LATENCY (CYCLES)	THROUGHPUT (OPERATIONS/CYCLE)	PARALLELISM (OPERATIONS)
Fermi	20	32	640
Kepler	20	192	3,840

另外有两种方法可以提高并行：

- 指令级并行(ILP): 一个线程中有很多独立的指令
- 线程级并行(TLP): 很多并发地符合条件的线程

同样，与指令周期隐藏延迟类似，内存隐藏延迟是靠内存读取的并发操作来完成的，需要注意的是，指令隐藏的关键目的是使用全部的计算资源，而内存读取的延迟隐藏是为了使用全部的内存带宽，内存延迟的时候，计算资源正在被别的线程束使用，所以我们不考虑内存读取延迟的时候计算资源在做了什么，这两种延迟我们看做两个不同的部门但是遵循相同的道理。

我们的根本目的是把计算资源，内存读取的带宽资源全部使用满，这样就能达到理论的最大效率。

同样下表根据Little 法则给出了需要多少线程束来最小化内存读取延迟，不过这里有个单位换算过程，机器的性能指标内存读取速度给出的是GB/s 的单位，而我们需要的是每个时钟周期读取字节数，所以要用这个速度除以频率，例如C 2070 的内存带宽是144 GB/s 化成时钟周期： $\frac{144GB/s}{1.566GHz} = 92B/t$,这样就能得到单位时间周期的内存带宽了。

得出下表的数据

TABLE 3-4: Device Parallelism Required to Maintain Full Memory Utilization

GPU MODEL	INSTRUCTION LATENCY (CYCLES)	BANDWIDTH (GB/SEC)	BANDWIDTH (B/CYCLE)	PARALLELISM (KB)
Fermi	800	144	92	74
Kepler	800	250	96	77

需要说明的是这个速度不是单个SM的而是整个GPU设备的，以内们用的内存带宽是GPU设备的而不是针对一个SM的。

Fermi 需要并行的读取74的数据才能让GPU带宽满载，如果每个线程读取4个字节，我们大约需要18500个线程，大约579个线程束才能达到这个峰值。

所以，延迟的隐藏取决于活动的线程束的数量，数量越多，隐藏的越好，但是线程束的数量又受到上面的说的资源影响。所以这里就需要寻找最优的执行配置来达到最优的延迟隐藏。

那么我们怎么样确定一个线程束的下界呢，使得当高于这个数字时SM的延迟能充分的隐藏，其实这个公式很简单，也很好理解，就是SM的计算核心数乘以单条指令的延迟，

比如32个单精度浮点计算器，每次计算延迟20个时钟周期，那么我需要最少 $32 \times 20 = 640$ 个线程使设备处于忙碌状态。

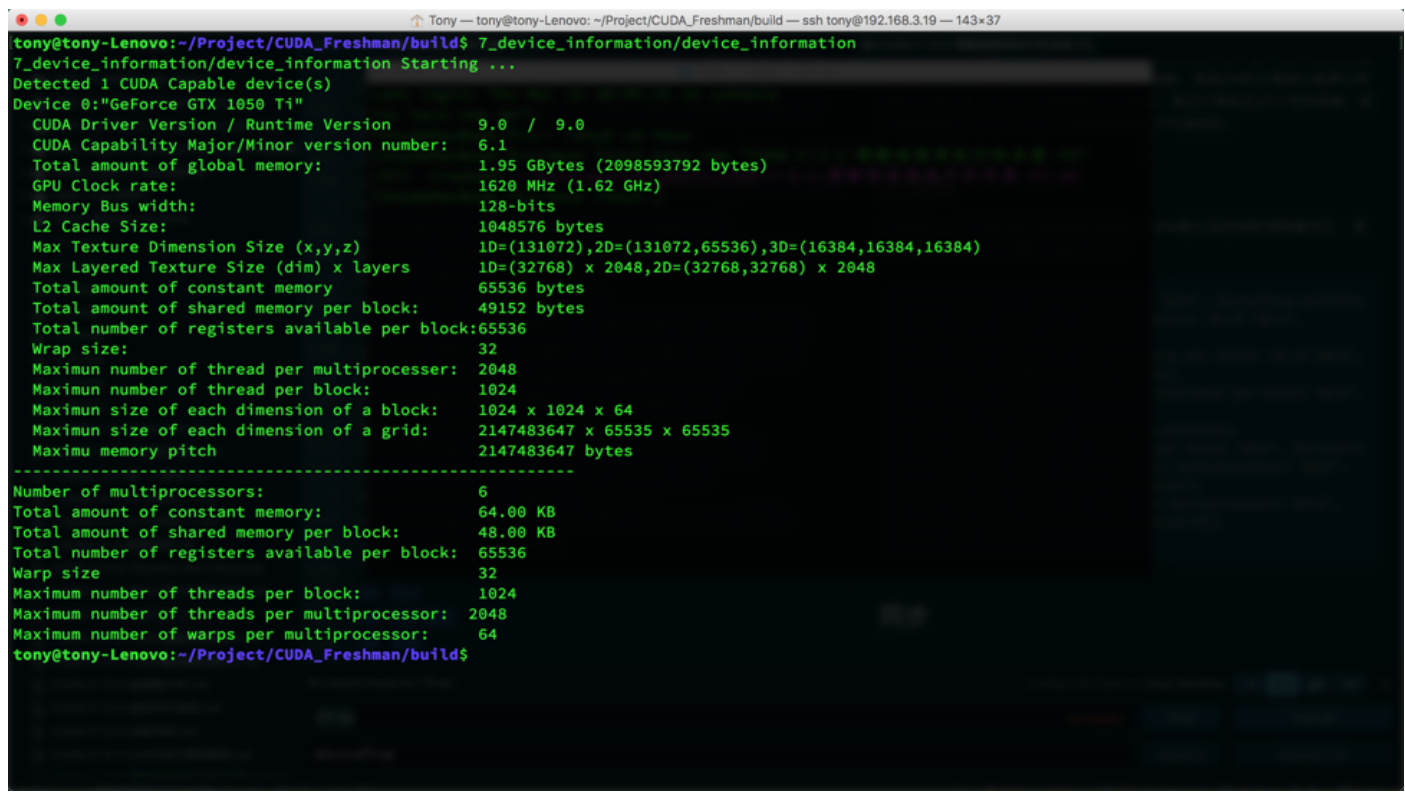
占用率是一个SM种活跃的线程束的数量，占SM最大支持线程束数量的比，

我们前面写的程序7_deviceInformation 中添加几个成员的查询就可以帮我们找到这个值：

完整代码：https://github.com/Tony-Tan/CUDA_Freshman

```
1 // 省略了上半部分
2 printf("-----\n");
3 printf("Number of multiprocessors:           %d\n", deviceProp.multiProc
4 printf("Total amount of constant memory:      %4.2f KB\n",
5 deviceProp.totalConstMem/1024.0);
6 printf("Total amount of shared memory per block:  %4.2f KB\n",
7 deviceProp.sharedMemPerBlock/1024.0);
8 printf("Total number of registers available per block:  %d\n",
9 deviceProp.regsPerBlock);
10 printf("Warp size                               %d\n", deviceProp.warpSize)
11 printf("Maximum number of threads per block:      %d\n", deviceProp.maxThreac
12 printf("Maximum number of threads per multiprocessor: %d\n",
13 deviceProp.maxThreadsPerMultiProcessor);
14 printf("Maximum number of warps per multiprocessor:  %d\n",
15 deviceProp.maxThreadsPerMultiProcessor/32);
16 return EXIT_SUCCESS;
```

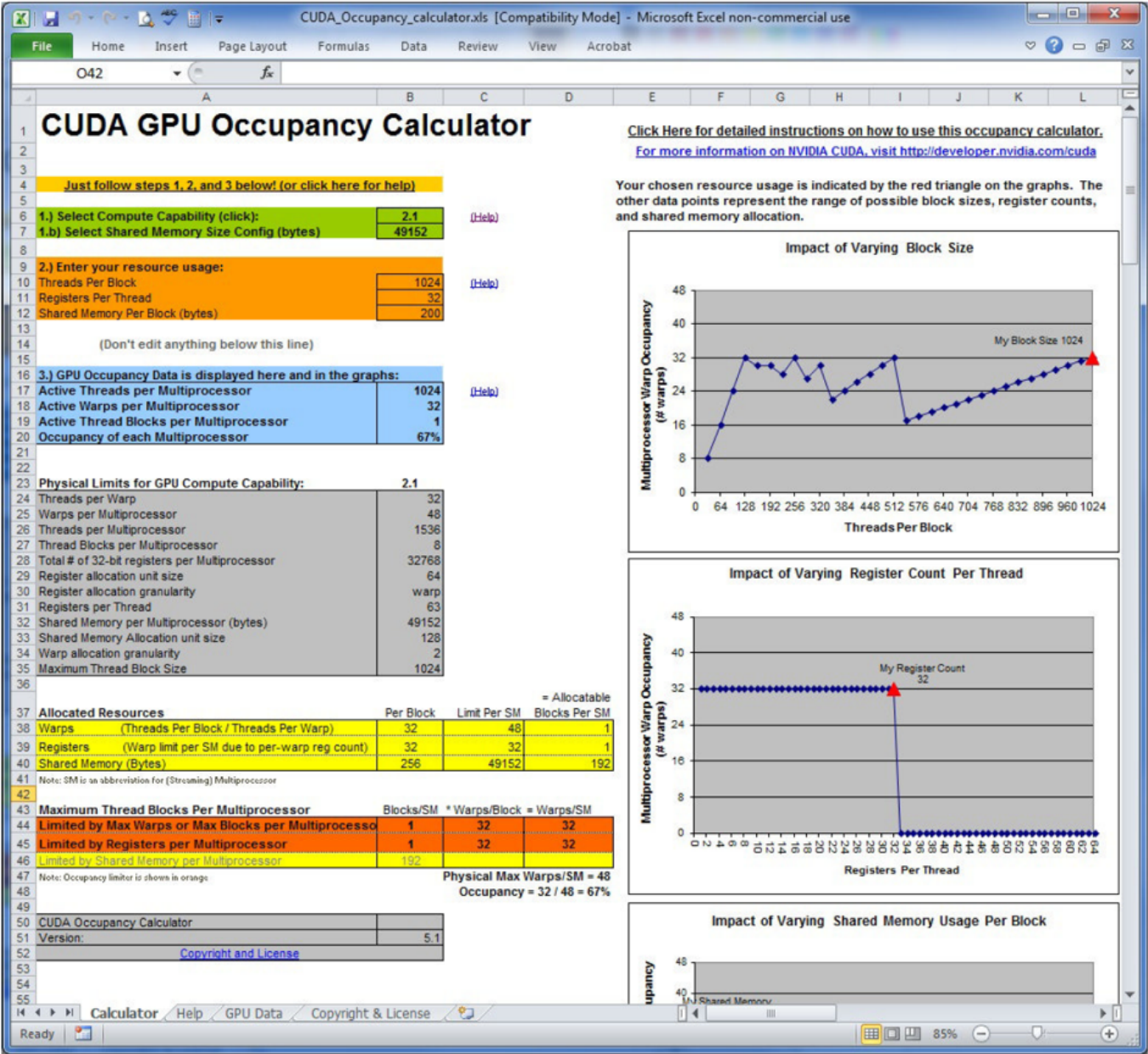
结果



```
tony@tony-Lenovo:~/Project/CUDA_Freshman/build$ 7_device_information/device_information
7_device_information/device_information Starting ...
Detected 1 CUDA Capable device(s)
Device 0:"GeForce GTX 1050 Ti"
  CUDA Driver Version / Runtime Version      9.0 / 9.0
  CUDA Capability Major/Minor version number: 6.1
  Total amount of global memory:              1.95 GBytes (2098593792 bytes)
  GPU Clock rate:                            1620 MHz (1.62 GHz)
  Memory Bus width:                          128-bits
  L2 Cache Size:                             1048576 bytes
  Max Texture Dimension Size (x,y,z)         1D=(131072),2D=(131072,65536),3D=(16384,16384,16384)
  Max Layered Texture Size (dim) x layers    1D=(32768) x 2048,2D=(32768,32768) x 2048
  Total amount of constant memory             65536 bytes
  Total amount of shared memory per block:    49152 bytes
  Total number of registers available per block:65536
  Warp size:                                 32
  Maximum number of thread per multiprocessor: 2048
  Maximum number of thread per block:         1024
  Maximum size of each dimension of a block:  1024 x 1024 x 64
  Maximum size of each dimension of a grid:   2147483647 x 65535 x 65535
  Maximum memory pitch                        2147483647 bytes
-----
Number of multiprocessors:                    6
Total amount of constant memory:              64.00 KB
Total amount of shared memory per block:      48.00 KB
Total number of registers available per block: 65536
Warp size                                     32
Maximum number of threads per block:          1024
Maximum number of threads per multiprocessor: 2048
Maximum number of warps per multiprocessor:   64
tony@tony-Lenovo:~/Project/CUDA_Freshman/build$
```


最大64个线程束每个SM。

CUDA工具包中提供一个叫做UCDA占用率计算器的电子表格，填上相关数据可以帮你自动计算网格参数：



上图是书上的截图，吐个槽，这些人居然写了个表格，为啥不写个程序？

上面我们已经明确内核使用寄存器的数量会影响SM内线程束的数量，nvcc的编译选项也有手动控制寄存器的使用。

也可以通过调整线程块内线程的多少来提高占用率，当然要合理不能太极端：

- 小的线程块：每个线程块中线程太少，会在所有资源没用完就达到了线程束的最大要求
- 大的线程块：每个线程块中太多线程，会导致每个SM中每个线程可用的硬件资源较少。

同步

并发程序对同步非常有用，比如pthread中的锁，openmp中的同步机制，这没做的主要目的是避免内存竞争

CUDA同步这里只讲两种：

- 线程块内同步
- 系统级别

块级别的就是同一个块内的线程会同时停止在某个设定的位置，用

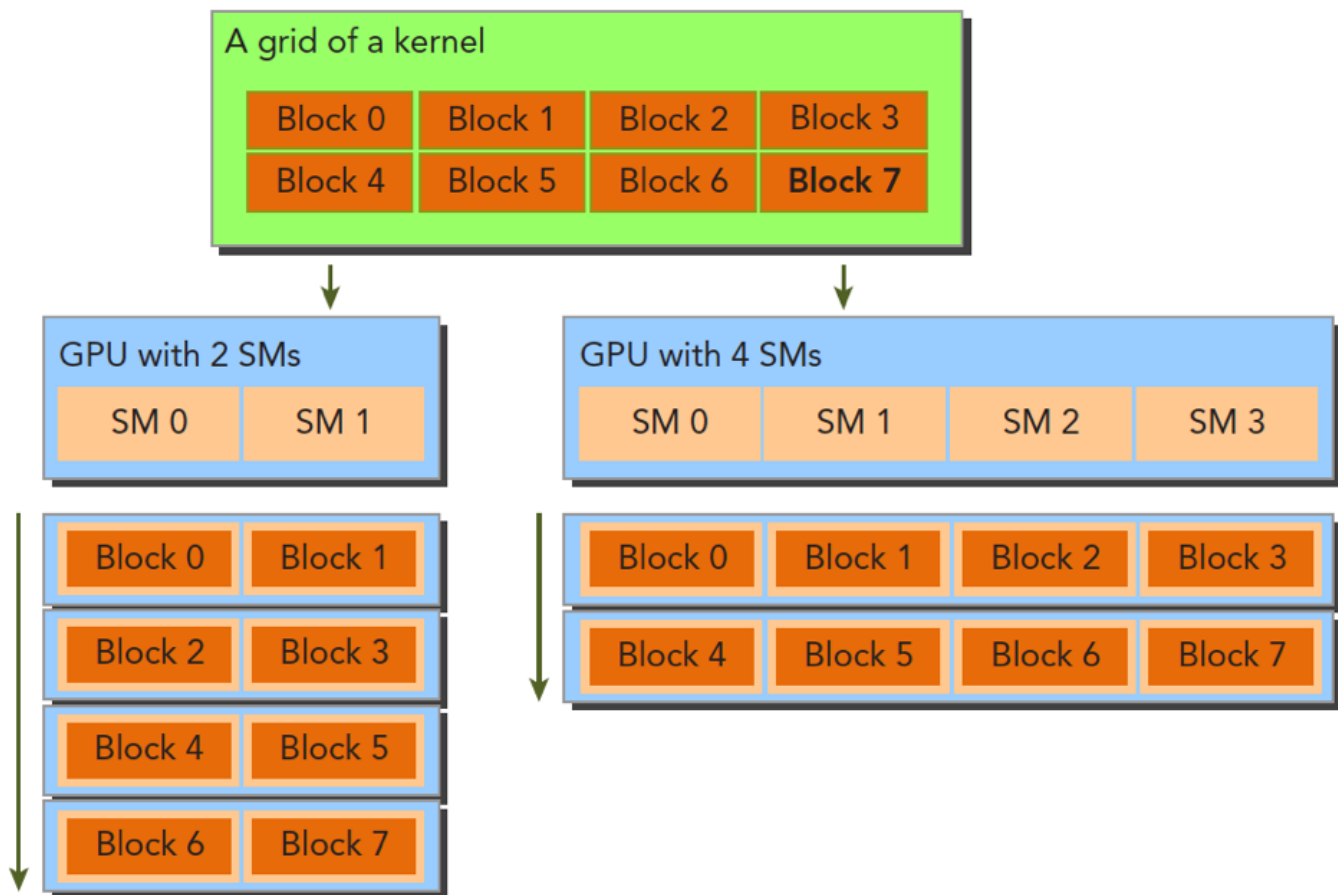
```
1 __syncthread();
```

这个函数完成，这个函数只能同步同一个块内的线程，不能同步不同块内的线程，想要同步不同块内的线程，就只能让核函数执行完成，控制程序交换主机，这种方式来同步所有线程。

内存竞争是非常危险的，一定要非常小心，这里经常出错。

可扩展性

可扩展性其实是相对于不同硬件的，当某个程序在设备1上执行的时候时间消耗是T当我们使用设备2时，其资源是设备1的两倍，我们希望得到T/2的运行速度，这种性质是CUDA驱动部分提供的特性，目前来说Nvidia正在致力于这方面的优化，如下图：



总结

今天效率很高，主要是这个部分之前已经研究透彻了，第三章是Freshman阶段的最核心部分，需要大家多查资料，多思考，多练习，待续。。。

本文作者： 谭升

本文链接：<https://face2ai.com/CUDA-F-3-2-理解线程束执行的本质-P2/>

版权声明： 本博客所有文章除特别声明外，均采用 [CC BY-NC-SA 4.0](#) 许可协议。转载请注明出处！

相关文章

- [【CUDA 基础】4.4 核函数可达到的带宽](#)
- [【Julia】整型和浮点型数字](#)
- [【Julia】变量](#)
- [【Julia】开始使用Julia](#)