

# 并行与分布式计算基础：第十九讲

杨超

chao\_yang@pku.edu.cn

2019 秋



# 内容提纲

## 1 CUDA 编程-4

- 基础知识 (简要回顾)
- 内存模型
- 程序举例：矩阵乘 (2)
- 线程簇优化

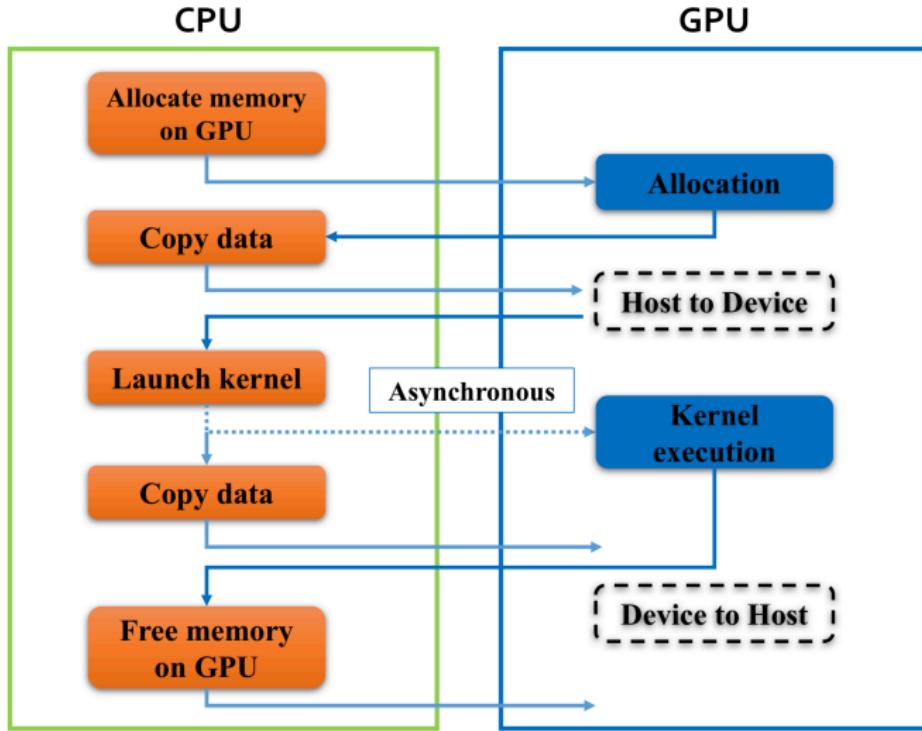
## 1 CUDA 编程-4

- 基础知识 (简要回顾)
- 内存模型
- 程序举例：矩阵乘 (2)
- 线程簇优化

# Nvidia GPU 总体架构简介

- NVIDIA 的 GPU 主要由多个 Streaming Multiprocessor (SM) 组成，SM 之间共享 L2 缓存；
- 每个 SM 由多个 Streaming Processor (SP) 组成，SP 之间共享控制逻辑和 L1 缓存；
- SP 主要包括单精度 (FP32) 核心、双精度 (FP64) 核心、特殊函数核心 (SFU) 等；
- 存储一般由 Graphics Double Data Rate (GDDR)，High Bandwidth Memory 2 (HBM2) 等组成；
- 与 CPU 传输数据一般使用 PCI-E (16-32 GB/s) 技术；
- 节点内 GPU 间数据传输可通过 NVLink (160-300 GB/s) 实现.

# CUDA 异构并行计算流程一览



# host 代码和 kernel 代码

一个典型的 CUDA 程序由两部分组成：

- host 代码——CPU 端执行：

- ▶ CPU 计算；
  - ▶ GPU 内存分配和释放；
  - ▶ CPU 数据与 GPU 数据的传输，包括常量和普通数据；
  - ▶ 检查错误信息、计时等.

- kernel 代码——GPU 端执行：

- ▶ 每段 kernel 代码可以作为多个执行实例 (execution instances)；
  - ▶ 每个执行实例被一个 SM 执行，每个 SM 可以对应多个实例；
  - ▶ 每个执行实例可以由多个线程并发执行；
  - ▶ 每个线程拥有私有的数据信息；
  - ▶ 同一执行实例的线程间可以通过共享存储进行交互.

# CUDA C 函数声明关键字

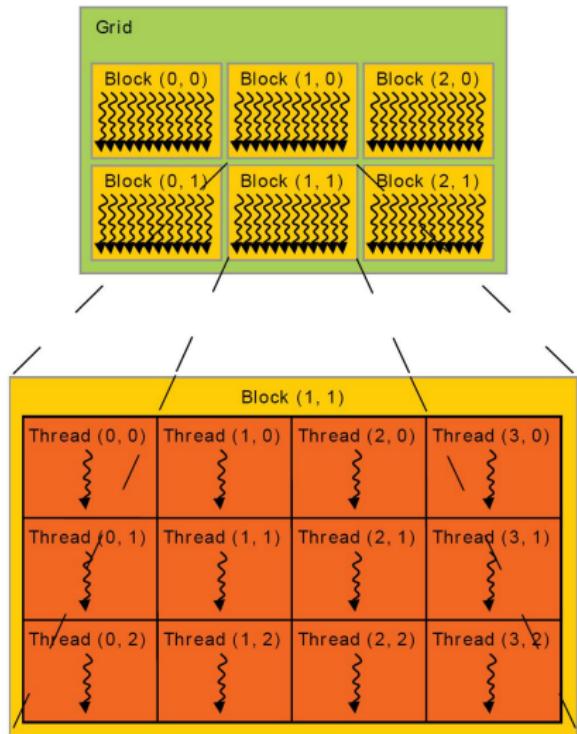
CUDA C 关键字	执行位置	调用位置
<code>--device__ type DeviceFunc()</code>	device	device
<code>--global__ void KernelFunc()</code>	device	host
<code>--host__ type HostFunc()</code>	host	host

- 默认函数是 host，所以一般 `--host__` 省略；
- `--global__` 定义了 kernel，只能返回 void；
- `--host__` 和 `--device__` 可以一起使用，告诉编译器生成 CPU 和 GPU 两个版本。

## CUDA 线程与 kernel 程序的关系

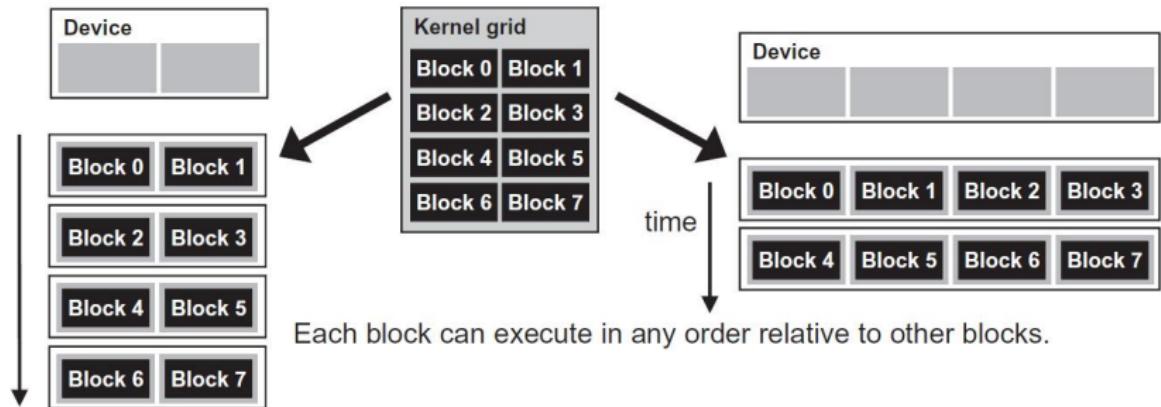
一个 CUDA kernel 程序被一个线程网格 (thread grid) 中的所有线程块 (thread block) 的所有线程执行：

- 线程网格中线程块的个数决定了 kernel 程序有多少个执行实例；
  - 线程块中线程的个数决定了每个执行实例由多少个线程执行；
  - 每个线程通过各自的索引来访问所需数据和控制程序执行流程。



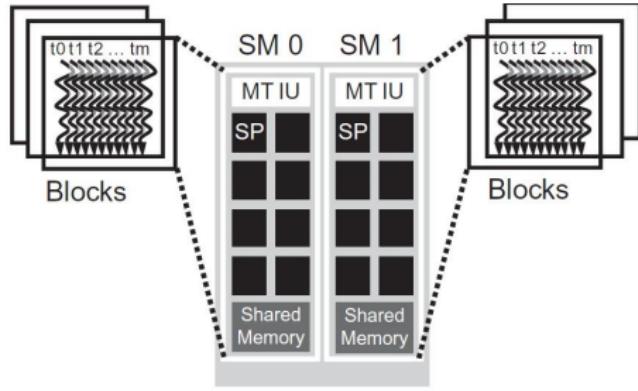
# 线程块间透明扩展

CUDA 的不同线程块间可以按照任何执行顺序进行计算，一般很难实现同步，之所以这样是为了能够在不同计算力的设备上进行透明扩展 (transparent scalability) .



# 线程块资源分配

- CUDA 程序按照 block-by-block 的方式进行调度，一个线程块只能在一个 SM 上执行，每个 SM 可以同时运行多个线程块（只要硬件资源允许）。
- 一般应设置线程块数多于硬件能同时运行的最大块数，CUDA 运行时会维护一个线程块列表，每当 SM 上的一个线程块计算完成后，就会分配新的线程块。



# 线程调度

- 被分配到 SM 上的线程块将按照 warp 为单元进行调度，而 warp 的大小跟设备计算能力相关，一般为 32 个线程.

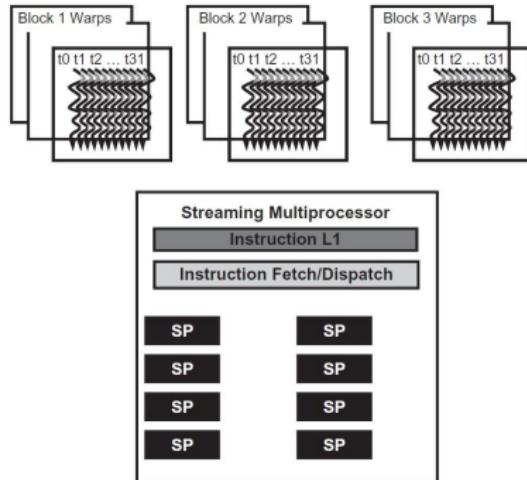


图: 每个 warp 中线程 threadIdx 值连续: 0 到 31 属于第一个 warp, 32 到 63 属于第二个 warp, 以此类推. 图中例子总共有三个线程块分配到当前 SM.

## Kernel Launch

In its simplest form it looks like:

```
kernel_routine<<<gridDim, blockDim>>>(args);
```

- gridDim is the number of instances of the kernel (the “grid” size)
- blockDim is the number of threads within each instance (the “block” size)
- args is a limited number of arguments, usually mainly pointers to arrays in graphics memory, and some constants which get copied by value

The more general form allows gridDim and blockDim to be 2D or 3D to simplify application programs

# 内存模型

## 1 CUDA 编程-4

- 基础知识 (简要回顾)
- 内存模型
- 程序举例：矩阵乘 (2)
- 线程簇优化

# P100 外观

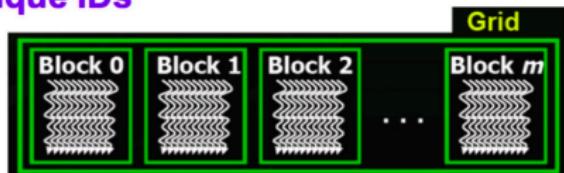
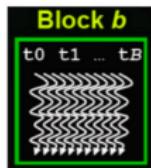
基于 Pascal GP100 架构的 NVIDIA Tesla P100 GPU 的板卡 (card) 示意图：

- 片上 (on-chip): SM、缓存、寄存器等；
- 片外 (off-chip): 设备内存 (device memory) 等.

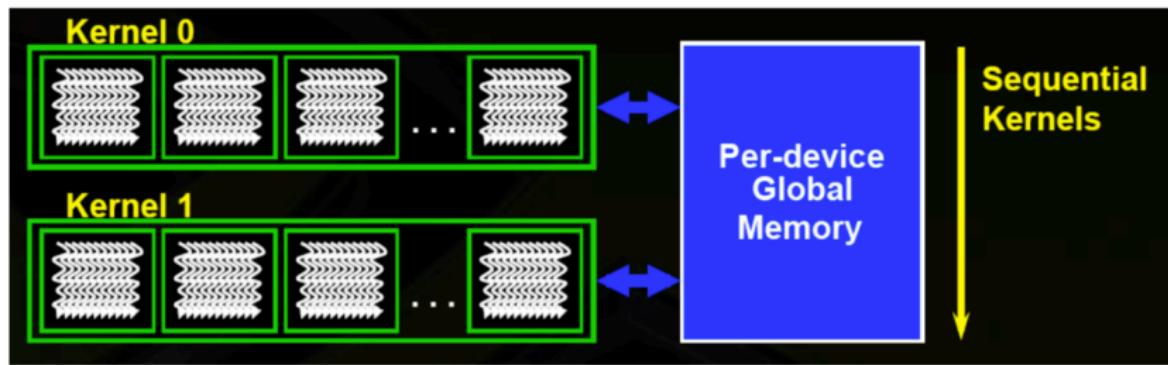


# 编程视角下的线程层次

- Parallel kernels composed of many **threads**
  - all threads execute same sequential program
  - use parallel threads rather than sequential loops
- Threads are grouped into **thread blocks**
  - threads in block can sync and share memory
- Blocks are grouped into **grids**
  - threads and blocks have unique IDs
    - threadIdx: 1D, 2D, or 3D
    - blockIdx: 1D, 2D, or 3D
  - simplifies addressing when processing multidimensional data



# 编程视角下的存储层次



# 私有变量：局部内存、寄存器、数据漫溢

- 编程视角：
  - ▶ kernel 代码中定义的变量，默认情况下全部为私有变量 (private variable)，存储在局部内存 (local memory) 中。
- 硬件视角：
  - ▶ 局部内存优先占用寄存器 (register) 资源，寄存器由每个 SM 内所有活跃线程共用，是最宝贵的存储资源 (P100: 约 256KB/SIMD、1cycle)；
  - ▶ 如果线程占用的局部内存大于可用寄存器大小，则触发寄存器漫溢 (register spilling)，部分私有数据被存储在缓存和设备内存 (device memory) 中，具体调度机制不明；
  - ▶ 由于设备内存性能 (P100: 约 16GB、500cycle) 远远低于寄存器，会严重影响性能，因此应设法避免寄存器漫溢。

# 共享变量：共享内存、擦板内存

- 编程视角：

- ▶ kernel 代码中定义的带有 `__device__ __shared__` 关键字（其中 `__device__` 关键字可省略）的变量为共享变量 (shared variable)，存储在共享内存 (shared memory) 中，由同一线程块内所有线程共享；
- ▶ 可以用 kernel 的第三个运行配置参数调节每个线程块共享内存的 byte 数。

- 硬件视角：

- ▶ 共享内存为擦板内存 (scratchpad memory)，由每个 SM 内所有活跃线程共用，是第二宝贵的存储资源 (P100: 约 64KB/SM、5cycle)；
- ▶ 在早期 CUDA GPU 上，共享内存和 L1 缓存的总大小为 64KB，用户可控制两者之间的比例，从 Maxwell 开始，两者相互独立。

# 全局/常数变量：全局/常量内存

- 编程视角：

- ▶ kernel 代码外定义带有 `__device__` 关键字的变量为全局变量，存储于全局内存 (global memory) 中，可被所有线程访问；
- ▶ 对经常使用的只读数据，可额外加上 `__constant__` 关键字（此时 `__device__` 关键字可省略），设置为常数变量 (constant variable)，存储于常量内存 (constant memory) 中.

- 硬件视角：

- ▶ 全局内存即设备内存 (device memory) (P100: 约 16GB、500cycle)，两种名称经常混用；
- ▶ 常量内存实际是设备内存的一个特殊部分被单独缓存，缓存后可以快速读取 (P100: 约 64KB、5cycle).

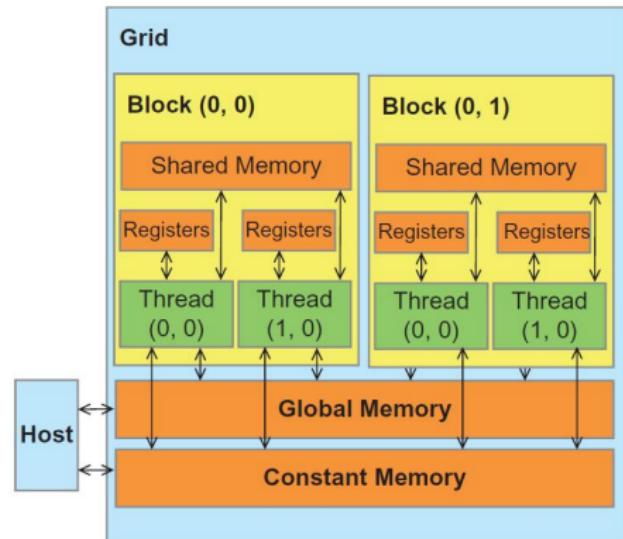
# 不同硬件层次的存储特性

Device 端代码：

- 访问寄存器 ( $\approx 1$  cycle)
- 访问共享内存 ( $\approx 5$  cycle)
- 访问全局内存 ( $\approx 500$  cycle)
- 访问常量存储 (缓存时  $\approx 5$  cycle)

Host 端代码：

- 在主存和设备全局内存间传递数据
- 在主存和设备常量存储间传递数据



# 程序举例：矩阵乘 (2)

## 1 CUDA 编程-4

- 基础知识 (简要回顾)
- 内存模型
- 程序举例：矩阵乘 (2)
- 线程簇优化

# GPU 上的矩阵乘

假设  $M, N$  均为宽度 width 的方阵，记  $P = MN$ ，则：

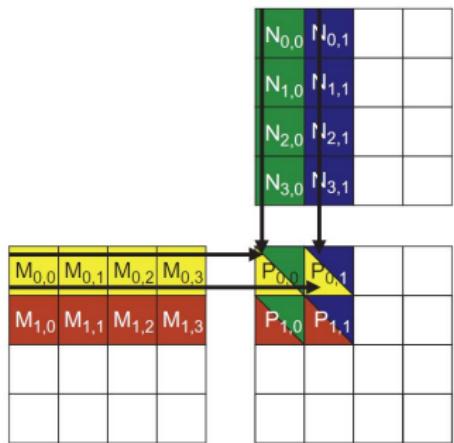
$$P_{i,j} = \sum_{k=0}^{width-1} M_{i,k}N_{k,j}.$$

```
1 __global__ void
2 gpu_mat_mul_kernel(float* M, float* N, float* P, int width) {
3     int Row = blockIdx.y * blockDim.y + threadIdx.y;
4     int Col = blockIdx.x * blockDim.x + threadIdx.x;
5     float sum = 0;
6     for (int k = 0; k < width; k++) {
7         sum += M[Row * width + k] * N[k * width + Col];
8     }
9     P[Row * width + Col] = sum;
10 }
```

# 矩阵乘的访存分析

Access order →

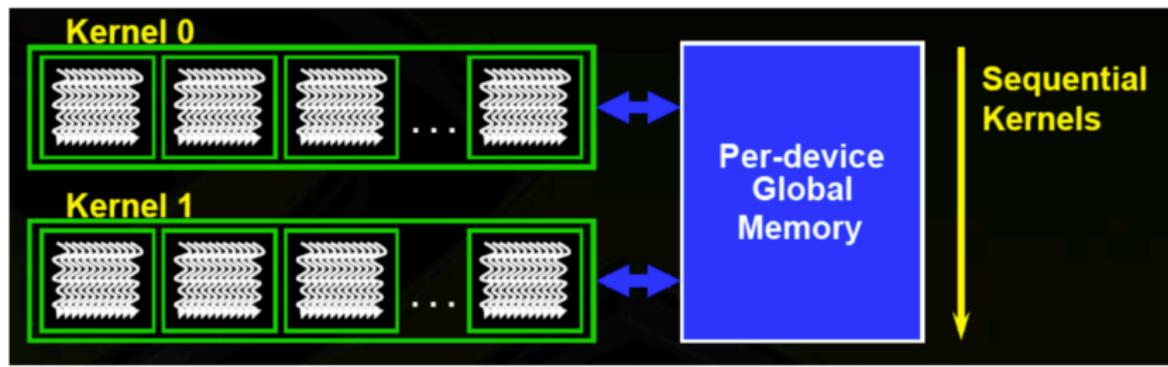
thread <sub>0,0</sub>	M <sub>0,0</sub> * N <sub>0,0</sub>	M <sub>0,1</sub> * N <sub>1,0</sub>	M <sub>0,2</sub> * N <sub>2,0</sub>	M <sub>0,3</sub> * N <sub>3,0</sub>
thread <sub>0,1</sub>	M <sub>0,0</sub> * N <sub>0,1</sub>	M <sub>0,1</sub> * N <sub>1,1</sub>	M <sub>0,2</sub> * N <sub>2,1</sub>	M <sub>0,3</sub> * N <sub>3,1</sub>
thread <sub>1,0</sub>	M <sub>1,0</sub> * N <sub>0,0</sub>	M <sub>1,1</sub> * N <sub>1,0</sub>	M <sub>1,2</sub> * N <sub>2,0</sub>	M <sub>1,3</sub> * N <sub>3,0</sub>
thread <sub>1,1</sub>	M <sub>1,0</sub> * N <sub>0,1</sub>	M <sub>1,1</sub> * N <sub>1,1</sub>	M <sub>1,2</sub> * N <sub>2,1</sub>	M <sub>1,3</sub> * N <sub>3,1</sub>



思考：

- (1) 如果线程块大小为 BLOCK\_WIDTH × BLOCK\_WIDTH，理论可以减少多少倍的全局内存访问？
- (2) 如何实现？

# CUDA 存储层次一览



# 矩阵乘的数据复用策略

将数据进行分块 (tiling)，并利用共享内存 (shared memory) 实现复用：

- ① 同一线程块内的线程协同加载一个子数据块到共享内存中；
- ② 同一线程块内的线程重复利用共享内存中数据进行计算；
- ③ 完成本块计算后，将共享内存中的计算结果拷贝到全局内存中，然后计算下个块.

在这里，我们取分块大小  $\text{TILE\_WIDTH} = \text{BLOCK\_WIDTH}$ .

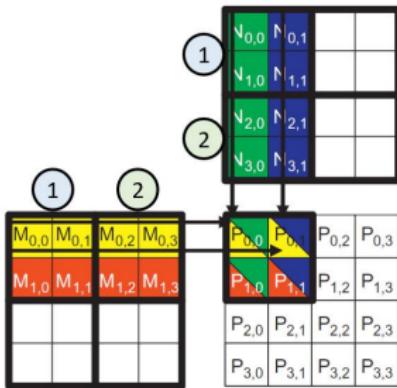
# GPU 上变量类型的设置

类型	位置	是否缓存	访问方式	可见域	生存期
Register	On-chip	No	R/W	1 thread	Thread
Shared	On-chip	No	R/W	All threads in block	Block
Global	Off-chip	Yes	R/W	All threads + host	Host allocation
Constant	Off-chip	Yes	R	All threads + host	Host allocation

- Register 变量 (在 kernel 里声明): `int register_var;`
  - Shared 变量 (在 kernel 里声明):  
`__device__ __shared__ int shared_var;`
  - Global 变量 (在 kernel 外声明): `__device__ int global_var;`
  - Constant 变量 (在 kernel 外声明):  
`__device__ __constant__ int const_var;`
- 其中 Shared 和 Constant 中的 `__device__` 一般可以省略.

# 矩阵乘的数据分块

- 按照全局访存进行线程分块，同时也将数据也进行对应分块。
- 计算分多个阶段执行，在每个阶段计算前，同一个线程块内线程先协同将需要数据块加载到 shared memory 中。



图：例子中线程块  $Block(0,0)$  的计算，分两个阶段，将两个阶段结果累加就是最终计算结果。

# 矩阵乘的分阶段计算

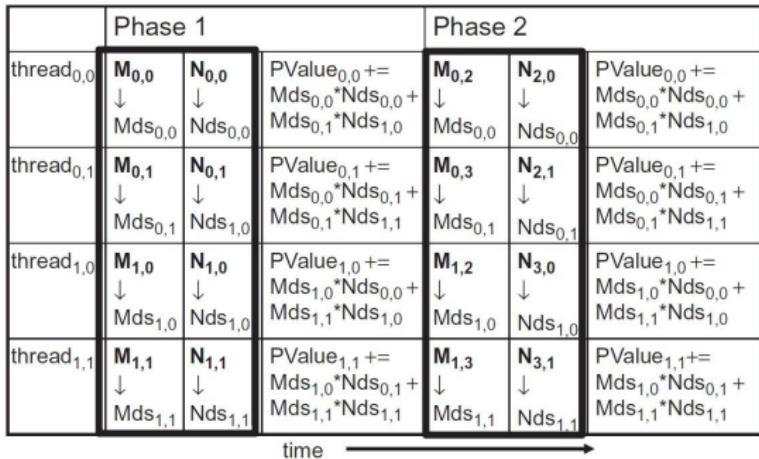
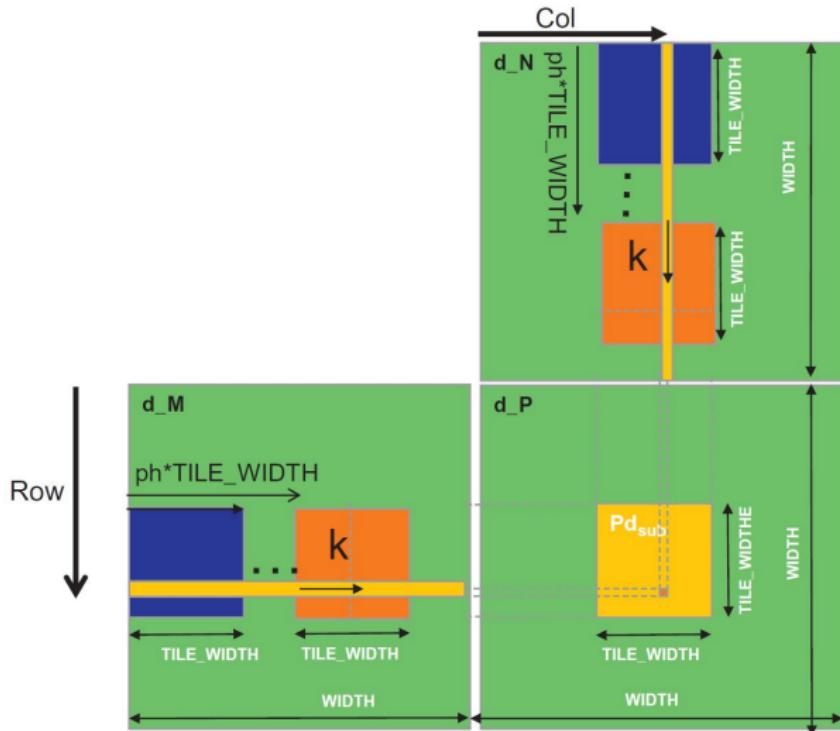


图: Mds 代表 M 矩阵的共享内存, Nds 表示 N 矩阵的共享内存.

思考: 按照这种计算方式, 如果分块大小为 TILE\_WIDTH, 矩阵宽度为 Width, 那么共需要多少个阶段? 数据能够复用多少次?

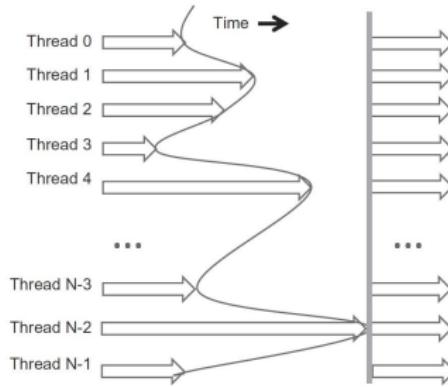
# 矩阵乘的计算过程

计算过程示意图：



# 栅栏同步

- CUDA 使用 `__syncthreads()` 语句可以保证同一个线程块内的所有线程同步.



- 在矩阵乘矩阵算法中的应用：
  - ▶ 确保每个 tile 所需的数据加载到 shared memory 中.
  - ▶ 确保每个 tile 计算的结果从 shared memory 存回到全局内存中.

# GPU 矩阵乘优化版程序

```
1  __global__ void
2  gpu_mat_mul_kernel(float* M, float* N, float* P, int width){
3
4      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
5      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
6      int bx = blockIdx.x; int by = blockIdx.y;
7      int tx = threadIdx.x; int ty = threadIdx.y;
8
9      // Identify the row and column of the P element to work on
10     // Each thread works on an element of P
11     int Row = by * TILE_WIDTH + ty;
12     int Col = bx * TILE_WIDTH + tx;
13
14     float sum = 0;
15     int phase_num = width/TILE_WIDTH;
16
17     // Each thread loads 'Row'th row of M and 'Col'th column of N
18     for (int ph = 0; ph < phase_num; ph++) {
19
```

```
20 // Collaborative loading data into shared memory
21 Mds[ty][tx] = M[Row * width + ph * TILE_WIDTH + tx];
22 Nds[ty][tx] = N[(ph * TILE_WIDTH + ty) * width + Col];
23
24 __syncthreads();
25 for (int k = 0; k < TILE_WIDTH; ++k) {
26     sum += Mds[ty][k] * Nds[k][tx];
27 }
28 __syncthreads();
29 }
30 P[Row * width + Col] = sum;
31 }
```

# 程序结果分析

- 程序测试结果：

```
Matrix width: 512.  
CPU: 0.84830 sec  
grid dim: 32, 32, 1.  
block dim: 16, 16, 1.  
kernel time: 0.00020 sec  
GPU: 0.14626 sec  
GPU all values correct
```

```
Matrix width: 1024.  
CPU: 8.15976 sec  
grid dim: 64, 64, 1.  
block dim: 16, 16, 1.  
kernel time: 0.00156 sec  
GPU: 0.17366 sec  
GPU all values correct
```

- 思考：相比于上一个版本快了多少，为什么？

# 作业

目前矩阵乘的计算程序 ex5 包含了两条基本假设：

- 假设一：矩阵宽度刚好是线程块宽度的整数倍.
- 假设二：矩阵是方阵.

放宽假设：

- 如果矩阵宽度不是线程块宽度的整数倍怎么办？
- 如果矩阵不是方阵怎么办？

作业：修改 ex5 中的程序，使程序可以满足上述两个条件.

提交时间：2019 年 12 月 23 日 24 点前，发邮件给助教.

# 编译提示

- 可采用--ptxas-options=-v 选项打开编译提示

```
$ nvcc --ptxas-options=-v --gpu-architecture=sm_50 -std=c++11  
-O3 -Wno-deprecated-gpu-targets -c mat_mul.cu -o  
gpu_mat_mul.o  
ptxas info      : 0 bytes gmem  
ptxas info      : Compiling entry function '  
_Z18gpu_mat_mul_kernelPfS_S_i' for 'sm_50'  
ptxas info      : Function properties for  
_Z18gpu_mat_mul_kernelPfS_S_i  
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill  
loads  
ptxas info      : Used 32 registers, 348 bytes cmem[0]
```

- 进一步，可用 cuda-memcheck 命令检查内存，包括初始化、读写冲突、同步、泄漏等等。

# 性能统计与分析

- 可采用 nvprof 命令实现性能统计

```
$ nvprof ./mat_mul 512
==69662== NVPROF is profiling process 69662, command: ./mat_mul 512
==69662== Profiling application: ./mat_mul 512
==69662== Profiling result:
      Type  Time (%)    Time   Calls       Avg        Min        Max     Name
GPU activities: 43.84%  199.43us    1  199.43us  199.43us  199.43us  gpu_mat_mul_kernel
                  38.32%  174.31us    2  87.155us  87.011us  87.299us  [CUDA memcpy HtoD]
                  17.83%  81.123us    1  81.123us  81.123us  81.123us  [CUDA memcpy DtoH]
      API calls: 98.77%  283.98ms    3  94.660ms  6.4140us  283.80ms  cudaMalloc
                  0.51%  1.4803ms    3  493.45us  265.24us  880.36us  cudaMemcpy
                  0.33%  955.06us   94  10.160us   174ns  446.55us  cuDeviceGet...
                  0.13%  384.60us    3  128.20us  16.027us  201.86us  cudaFree
                  0.13%  362.15us    1  362.15us  362.15us  362.15us  cuDevice...
                  0.07%  202.13us    1  202.13us  202.13us  202.13us  cudaEvent...
                  0.02%  70.719us    1  70.719us  70.719us  70.719us  cuDevice...
                  0.02%  46.598us    1  46.598us  46.598us  46.598us  cudaLaunch
      ...
...
```

- 进一步，可采用 nvpp 等工具进行性能分析.

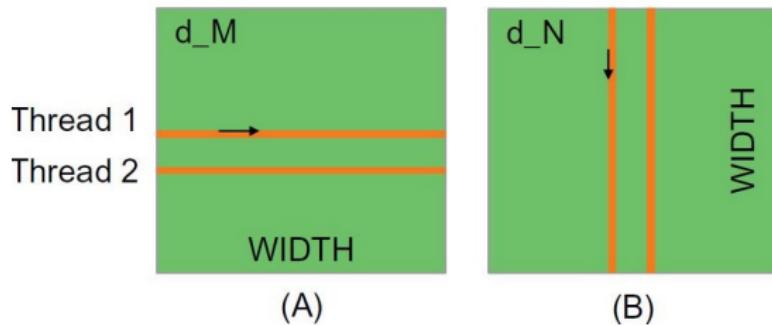
# 线程簇优化

## 1 CUDA 编程-4

- 基础知识 (简要回顾)
- 内存模型
- 程序举例：矩阵乘 (2)
- 线程簇优化

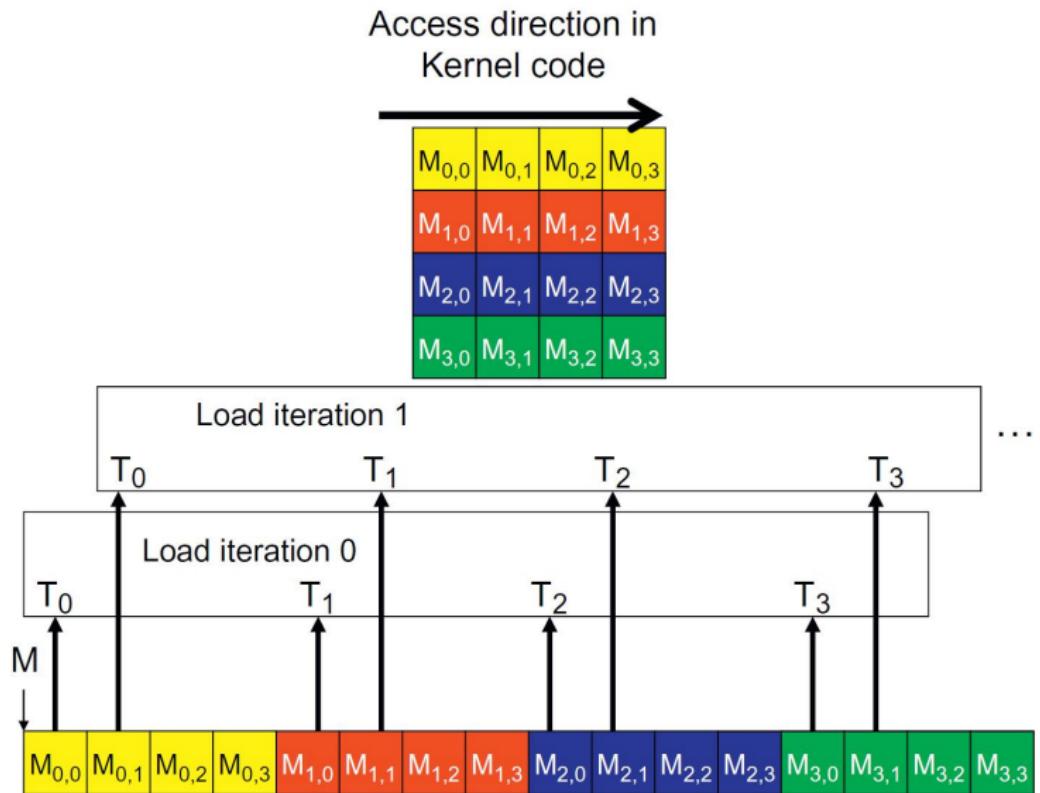
# 全局内存的合并访问

- 如果同一个线程块中多个线程同时访问全局内存的相邻位置，硬件上会触发合并访问 (coalesced access) 机制提高性能；
- 为了实现合并访存，需要保证数据内存对齐 (memory alignment)，位于同一个 cache line 中 (P100: L1 cache line=128Byte).



思考：矩阵乘 ex4 算例中，矩阵 M 和 N 的访问是合并访问吗？

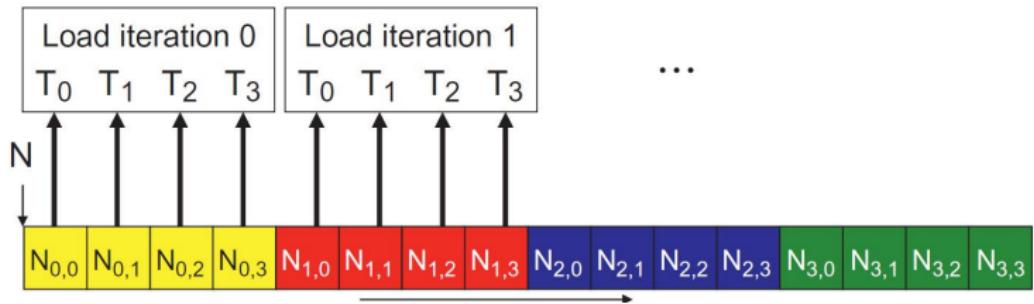
# 矩阵 M 的访存意图：



## 矩阵 N 的访存示意图：

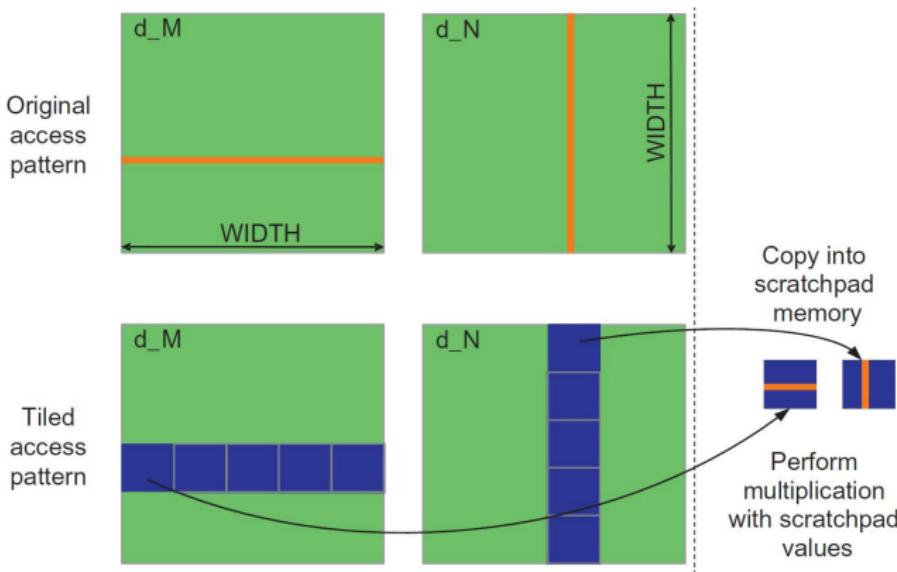
Access  
direction  
in Kernel  
code

N <sub>0,0</sub>	N <sub>0,1</sub>	N <sub>0,2</sub>	N <sub>0,3</sub>
N <sub>1,0</sub>	N <sub>1,1</sub>	N <sub>1,2</sub>	N <sub>1,3</sub>
N <sub>2,0</sub>	N <sub>2,1</sub>	N <sub>2,2</sub>	N <sub>2,3</sub>
N <sub>3,0</sub>	N <sub>3,1</sub>	N <sub>3,2</sub>	N <sub>3,3</sub>



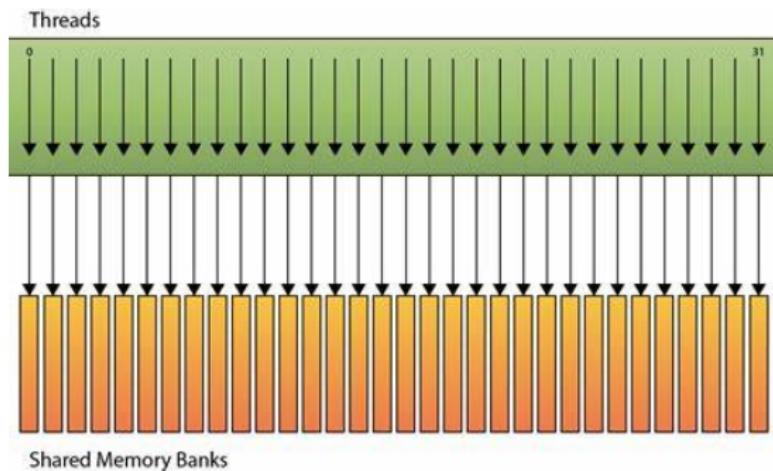
# 利用数据分块实现合并访存

在矩阵乘 ex5 算例中，数据分块实现了矩阵 M 的全局内存合并访问（可参考代码验证）。



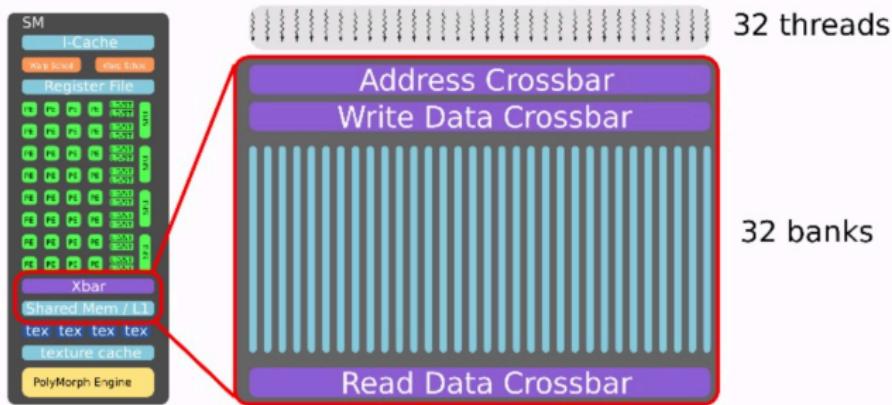
# 共享内存的组织

- 共享内存被分为 32 个大小相同 (一般 1word=32bit) 且地址连续的内存模组，称为 bank，这些数据可以同时在一个时钟周期被访问；
- 如何判断地址与 bank 的对应关系：把地址按照 32word 取模，范围 0-31，对应于不同的 bank，分配方式类似于 round-robin.



# Bank Conflict

- 在同一时刻，如果同一 warp 中的 32 个线程访问不同的 bank，则能够达到最佳性能，否则会出现 bank conflict；
- 出现 bank conflict 时，系统会将访存分解为多步 conflict-free 操作；
- 例外，若不同线程访问同一地址，系统会用广播避免 conflict.



# 避免 Bank Conflict

- 一般可通过调整访存跨步、偏置和补零等手段避免 bank conflict.

