

并行与分布式计算基础：第十七讲

杨超

chao_yang@pku.edu.cn

2019 秋



内容提纲

1 CUDA 编程-3

- CUDA 基础知识 (含复习)
- 线程的组织与调度
- 程序举例：图片灰白化
- 程序举例：矩阵乘 (1)

CUDA 基础知识 (含复习)

1 CUDA 编程-3

- CUDA 基础知识 (含复习)
- 线程的组织与调度
- 程序举例：图片灰白化
- 程序举例：矩阵乘 (1)

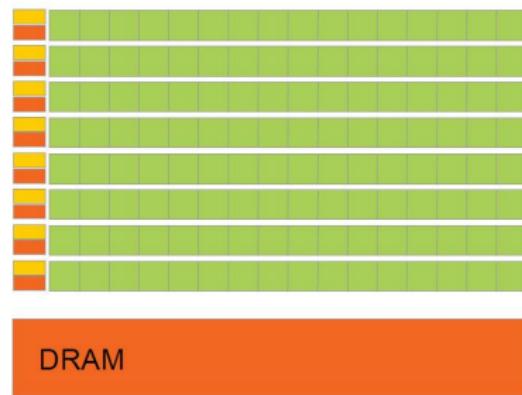
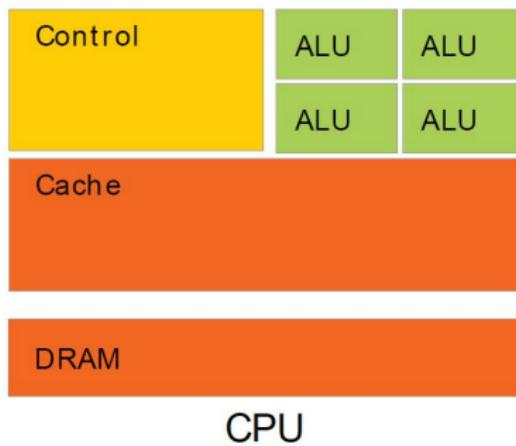
CPU 和 GPU 对比

CPU = Central Processing Unit:

- 面向延迟的设计；
- 非常大的 cache；
- 复杂的控制逻辑；
- 强大的 ALU；
- 不多的线程.

GPU = Graphic Processing Unit:

- 面向通量的设计；
- 小的 cache；
- 简单的控制逻辑；
- 高能效的 ALU；
- 大量的线程.



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.

每个 SM 中的核都是 SIMT (Single Instruction Multiple Threads):

- 每 32 个 (将来也许会增加) 线程一组，构成一个线程簇 (warp);
- 同一个线程簇中的线程同时执行同样的程序；
- 每个线程处理的数据不同.

为什么性能高？

- 大量的线程数，从而大幅提升了吞吐能力；
- 无需上下文切换 (context switching): 每个线程将数据存储在私有存储 (寄存器) 中；
- 硬件内置的线程簇调度器 (warp scheduler) 高效调度线程簇，保障多个活跃的线程簇被同时执行，线程簇非活跃往往是因为等待数据.

P100 的 SM 架构



FP32 核数 FP64 核数 寄存器大小 共享存储大小 线程簇调度器

64

32

256 KB

64 KB

2

P100 总体架构



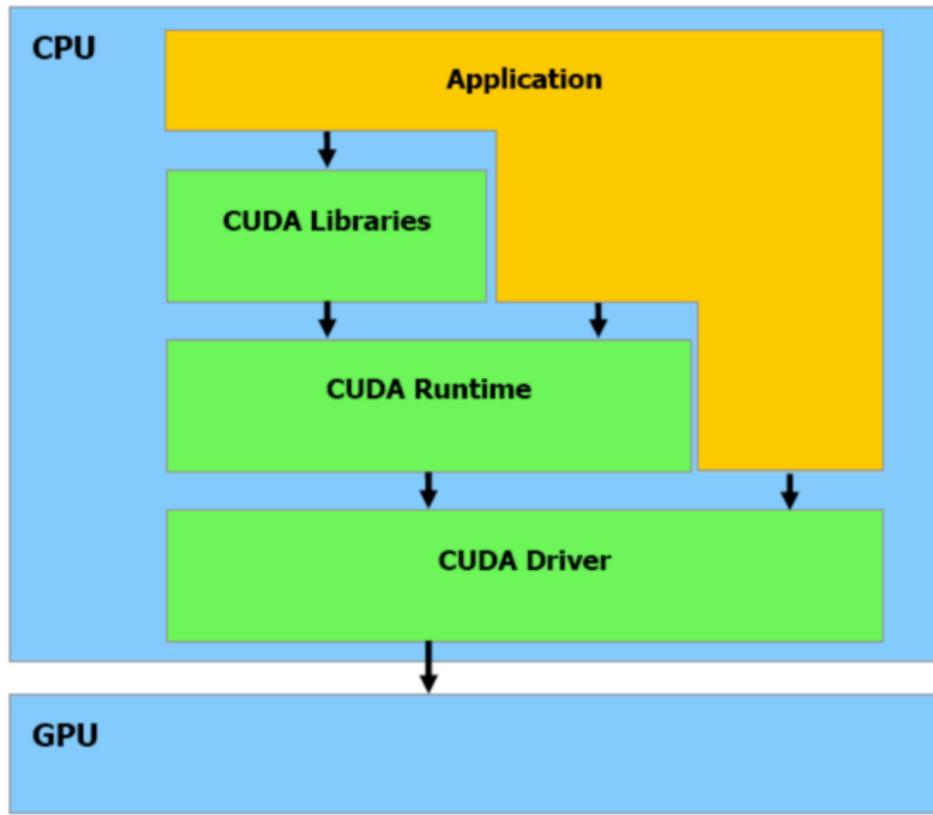
SM 个数	总 FP32 核数	总 FP64 核数	L2 Cache	带宽	内存大小
56	3584	1792	4096 KB	732 GB/s	16 GB HBM2

几代 NVIDIA GPU 硬件参数总结

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS ¹	5	6.8	10.6	15.7
Peak FP64 TFLOPS ¹	1.7	.21	5.3	7.8
Peak Tensor TFLOPS ¹	NA	NA	NA	125
Texture Units	240	192	224	320

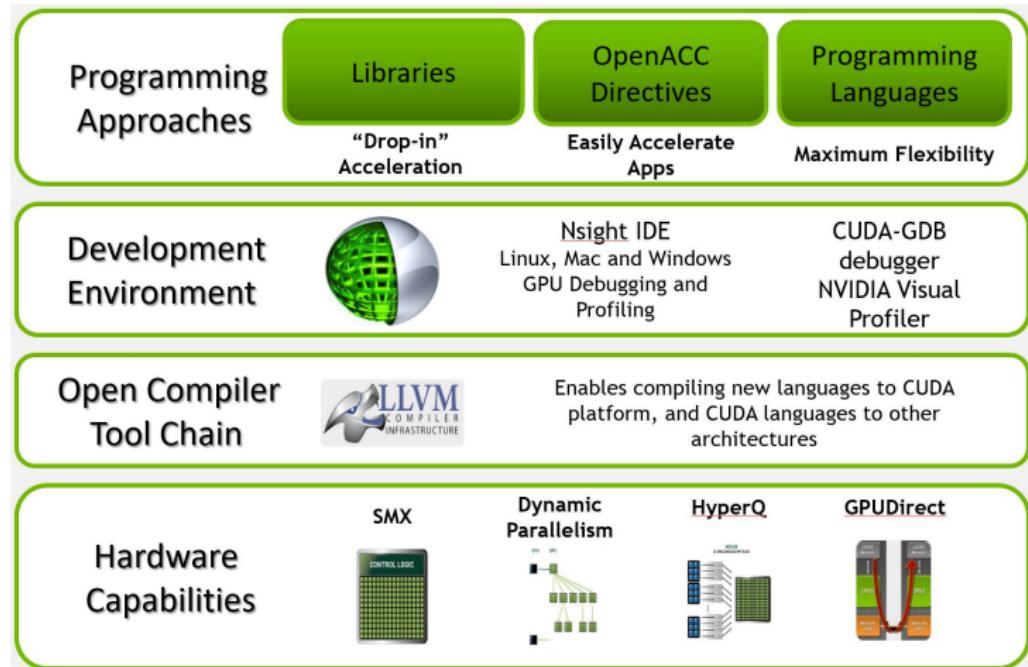
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm ²	601 mm ²	610 mm ²	815 mm ²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

CUDA 的软件架构



CUDA 并行编程平台

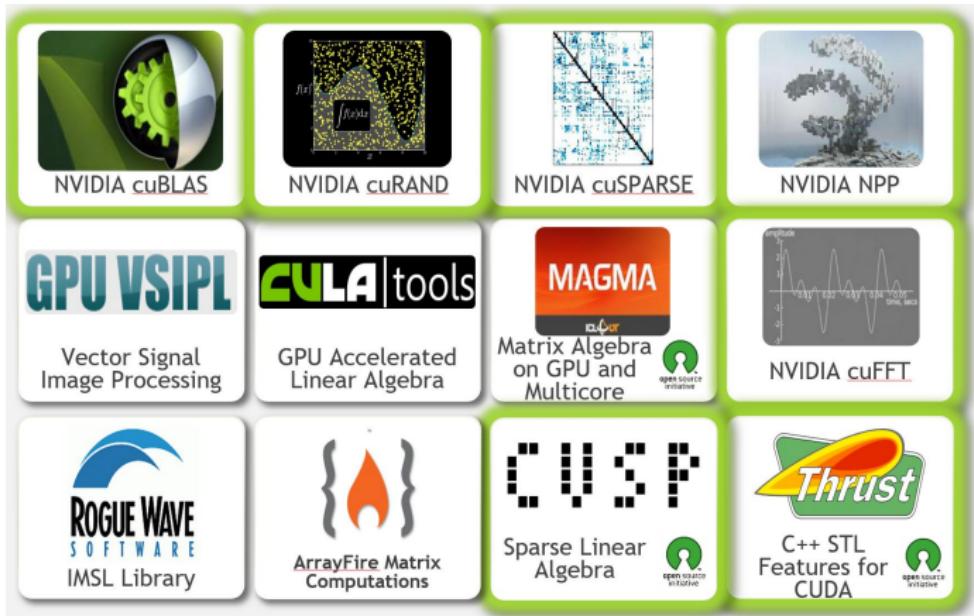
CUDA (Compute Unified Device Architecture): 由 NVIDIA 公司于 2007 年推出旨在进行通用 GPU 计算的并行计算平台和并行编程接口.



方式一：使用 CUDA 计算库

- 主要特点：

- ▶ 最简单，基本不需要了解 GPU 并行编程知识；
- ▶ 库具有很高的代码质量，性能经受过专业人士调优.



方式二：使用 OpenACC 编译指示

- 主要特点：

- ▶ 较简单，加入编译制导语句能实现自动并行；
- ▶ 较灵活，能够快速实现多 GPU 和 CPU 上并行执行.

Matrix-vector multiplication

```
#pragma acc data copyin(a[0:n*m])
{
    ...
    #pragma acc data copyin(v[0:n]) \
        copyout(x[0:n])
    {
        ...
        matvecmul( x, a, v, m, n );
        ...
    }
    ...
}
```

```
void matvecmul( float* x, float* a,
                float* v, int m, int n ){
#pragma acc parallel loop gang \
    pcopyin(a[0:n*m],v[0:n]) pcopyout(x[0:m])
    for( int i = 0; i < m; ++i ){
        float xx = 0.0;
        #pragma acc loop worker reduction(+:xx)
        for( int j = 0; j < n; ++j )
            xx += a[i*n+j]*v[j];
        x[i] = xx;
    }
}
```

方式三：使用 CUDA 编程语言

- 主要特点：

- ▶ 较复杂，需要深入了解 GPU 并行编程知识；
- ▶ 能提供最大的灵活性和性能需求.

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

C

```
__global__
void saxpy_parallel(int n,
                     float a,
                     float *x,
                     float *y)
{
    int i = blockIdx.x*blockDim.x +
            threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096, 256>>>(n, 2.0, x, y);
```

CUDA

CUDA 编程语言在已有编程语言基础上加入异构并行编程扩展.

Numerical analytics ➤

MATLAB, Mathematica, LabVIEW

Fortran ➤

OpenACC, CUDA Fortran

C ➤

OpenACC, CUDA C

C++ ➤

Thrust, CUDA C++

Python ➤

PyCUDA, Copperhead

F# ➤

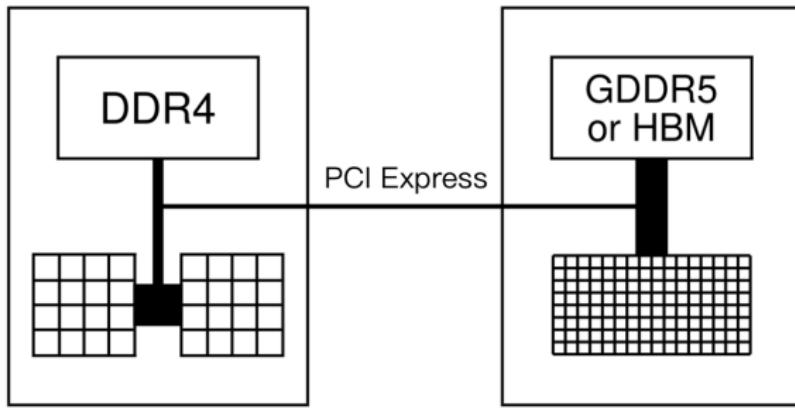
Alea.cuBase

硬件视角与编程视角

编程视角

主机 (host)

设备 (device)



硬件视角

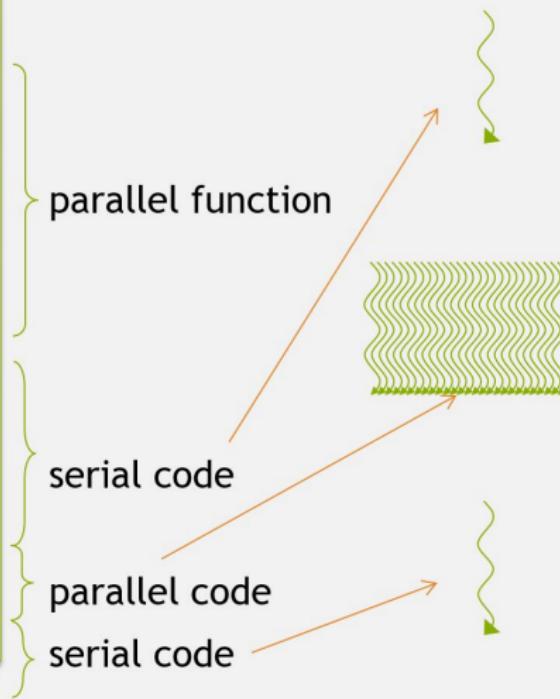
motherboard

graphics card

可以是
多个

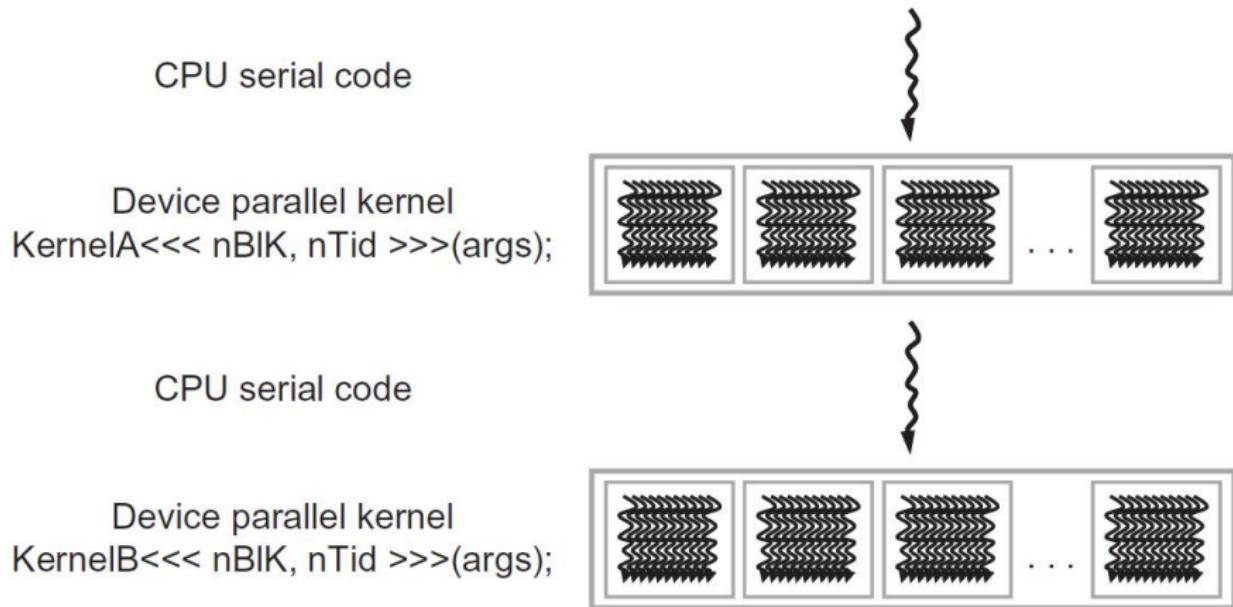
CUDA 异构编程方式

```
...  
#include <sys/types.h>  
#include <algorithm>  
  
using namespace std;  
  
#define N 1024  
#define RADIUS 3  
#define BLOCK_SIZE 16  
  
__global__ void stencil_1d<int>(int *out) {  
    __shared__ int temp[BLOCK_SIZE * 2 * RADIUS];  
    int index = blockDim.x * blockIdx.x * blockDim.x  
    int offset = blockDim.x * RADIUS;  
  
    // Read input elements into shared memory  
    temp[offset] = in[index];  
    if (offset <= RADIUS) {  
        temp[index - RADIUS] = in[index - RADIUS];  
        temp[index + RADIUS] = in[index + RADIUS];  
    }  
  
    // Synchronize (ensure all the data is available)  
    __syncthreads();  
  
    // Apply the stencil  
    int result = 0;  
    for (int offset = -RADIUS; offset <= RADIUS; offset++)  
        result += temp[index + offset];  
  
    // Store the result  
    out[index] = result;  
}  
  
void fill_1d<int>(int n) {  
    for (int i = 0; i < n; i++)  
        in[i] = 0;  
}  
  
int main(void) {  
    fill_1d<int>(N);  
    // host copies of a, b, c  
    int *d_in, *d_out;  
    int size = (N + 2*RADIUS) * sizeof(int);  
  
    // Alloc space for host copies and setup values  
    in = (int *)malloc(size);  
    d_in = (int *)malloc(size);  
    out = (int *)malloc(size);  
    d_out = (int *)malloc(size);  
  
    // Copy to device  
    cudaMemCpy(d_in, in, size, cudaMemcpyHostToDevice);  
    cudaMemCpy(d_out, out, size, cudaMemcpyHostToDevice);  
  
    // Launch stencil_1d(16) kernel on GPU  
    stencil_1d<int>(N, BLOCK_SIZE, BLOCK_SIZE) >> (d_in + RADIUS, d_out - RADIUS);  
  
    // Copy result back to host  
    cudaMemCpy(out, d_out, size, cudaMemcpyDeviceToHost);  
  
    // Clean up  
    free(in); free(d_in);  
    cudaFree(d_out);  
    return 0;  
}
```

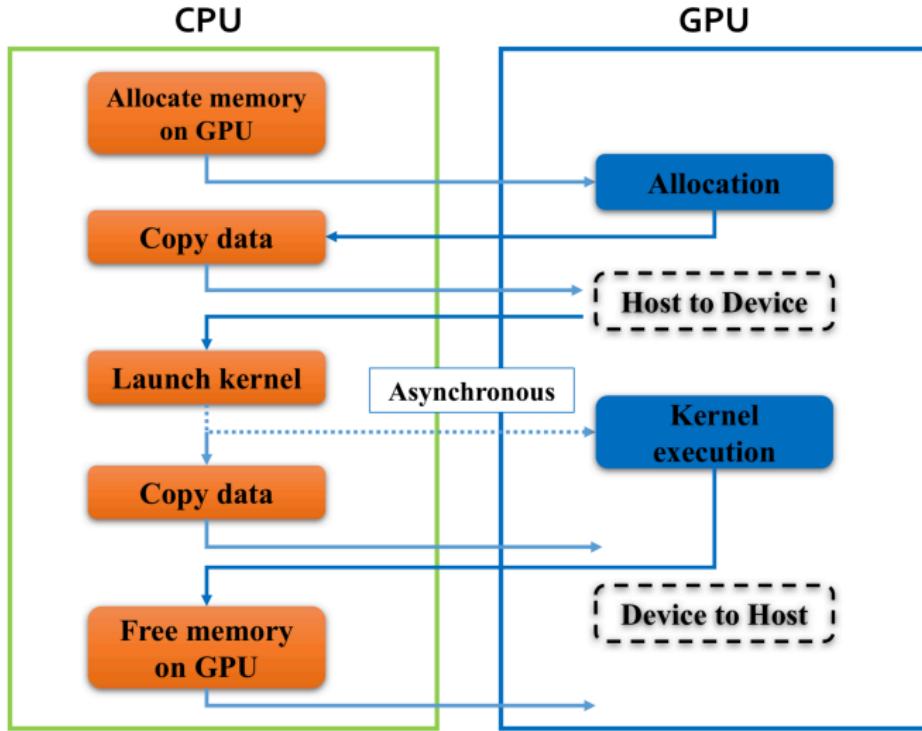


CUDA 程序执行模型

基本模型： host (串行或少量) + device (大量并行).



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.

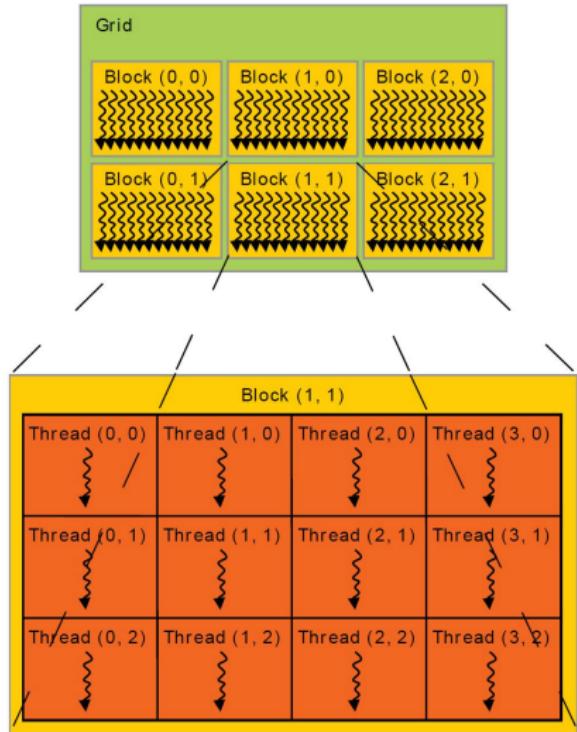
- kernel 代码——GPU 端执行：

- ▶ 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 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。



CUDA 并行网格组织层次

CUDA 并行线程网格分两层进行组织：

- 第一层：多个 thread 按照 1、2 或 3 维方式组成一个 block，同一 block 的线程可以通过共享内存、原子操作、栅栏同步来协作。
 - ▶ Block 的 x,y,z 轴大小通过内部结构体变量 blockDim 的 x,y,z 成员变量表示；
 - ▶ 每个线程的 x,y,z 轴坐标通过内部结构体变量 threadIdx 的 x,y,z 成员变量定位。
- 第二层：多个 block 按照 1、2 或 3 维方式组成一个 grid，不同 block 内线程相对独立。
 - ▶ Grid 的 x,y,z 轴大小通过内部结构体变量 gridDim 的 x,y,z 成员变量表示；
 - ▶ 每个 block 的 x,y,z 轴坐标通过内部结构体变量 blockIdx 的 x,y,z 成员变量定位。

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

kernel launch 的一般形式

- 假设 kernel 函数定义：`_global_ void Func(float* input)`
- 那么在 host 上发起时的执行配置：

```
Func<<<grid_dim, blk_dim, shared_size, stream_id>>>(input)
```

- ▶ `grid_dim`: 整型或 `dim3` 类型, 定义 grid 的大小, 总线程块数量等于 `grid_dim.x × grid_dim.y × grid_dim.z`;
- ▶ `blk_dim`: 整型或 `dim3` 类型, 定义每个 block 的大小, 总线程数量等于 `blk_dim.x × blk_dim.y × blk_dim.z`;
- ▶ `shared_size`: 可选参数, 默认为 0, `size_t` 类型, 确定为每个线程块动态分配的 shared memory 大小 (暂时忽略);
- ▶ `stream_id`: 可选参数, 默认为 0, `cudaStream_t` 类型, 定义执行关联的 stream (暂时忽略).

具体到每个线程

At the lower level, when one instance of the kernel is started on a SM it is executed by a number of threads, each of which knows about:

- some variables passed as arguments
- pointers to arrays in device memory (also arguments)
- global constants in device memory
- shared memory and private registers/local variables
- some special variables:
 - `gridDim` size (or dimensions) of grid of blocks
 - `blockDim` size (or dimensions) of each block
 - `blockIdx` index (or 2D/3D indices) of block
 - `threadIdx` index (or 2D/3D indices) of thread
 - `warpSize` always 32 so far, but could change

线程网格组织层次举例：1D

1D grid with 4 blocks, each with 64 threads:

- gridDim = 4
- blockDim = 64
- blockIdx ranges from 0 to 3
- threadIdx ranges from 0 to 63



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。

内存分配

- `cudaError_t cudaMalloc(void** devPtr, size_t size)`

分配设备内存 (global memory):

- ▶ `devPtr`: 存放所分配设备内存地址;
- ▶ `size`: 分配内存的大小, 字节为单位;
- ▶ `cudaError_t`: 如果调用成功返回 `cudaSuccess`, 否则返回相应错误码.

- `cudaError_t cudaFree(void** devPtr)`

释放已经分配的设备内存:

- ▶ `devPtr`: 对应上述分配内存函数中的设备内存地址;
- ▶ `cudaError_t`: 如果调用成功返回 `cudaSuccess`, 否则返回相应错误码.

数据传输

- `cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)`
 - ▶ `dst`: 传输目的内存地址;
 - ▶ `src`: 数据源内存地址;
 - ▶ `count`: 传输的数据大小, 以字节为单位;
 - ▶ `cudaMemcpyKind`: 数据传输方式, 有:
 - ★ Host → Host: `cudaMemcpyHostToHost`;
 - ★ Host → Device: `cudaMemcpyHostToDevice`;
 - ★ Device → Host: `cudaMemcpyDeviceToHost`;
 - ★ Device → Device: `cudaMemcpyDeviceToDevice`;
 - ▶ `cudaError_t`: 如果调用成功返回 `cudaSuccess`, 否则返回相应错误码.

统计 kernel 运行时间

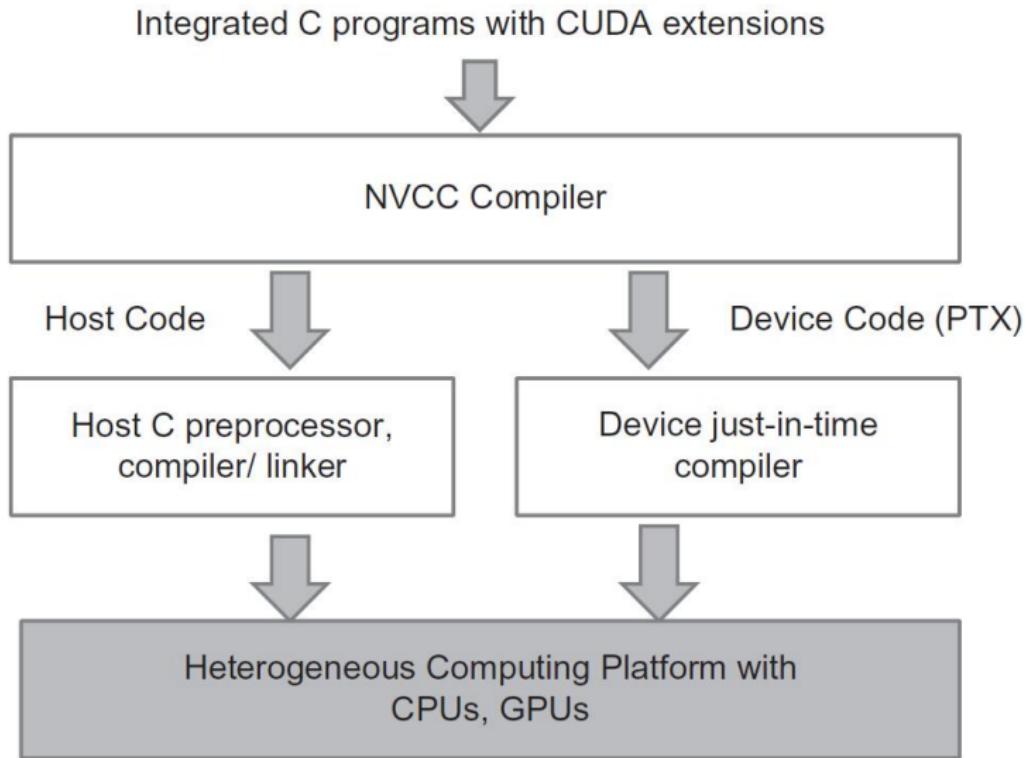
- 创建和销毁 CUDA event:

```
1  cudaEvent_t start, stop;  
2  cudaEventCreate(&start);  
3  cudaEventCreate(&stop);  
4  ...  
5  cudaEventDestroy(start);  
6  cudaEventDestroy(stop);
```

- 利用 CUDA event 统计 kernel 耗时:

```
1  cudaEventRecord(start, 0);  
2  ...  
3  cudaEventRecord(stop, 0);  
4  cudaEventSynchronize(stop);  
5  float elapsedTime;  
6  cudaEventElapsedTime(&elapsedTime, start, stop);
```

CUDA 程序编译



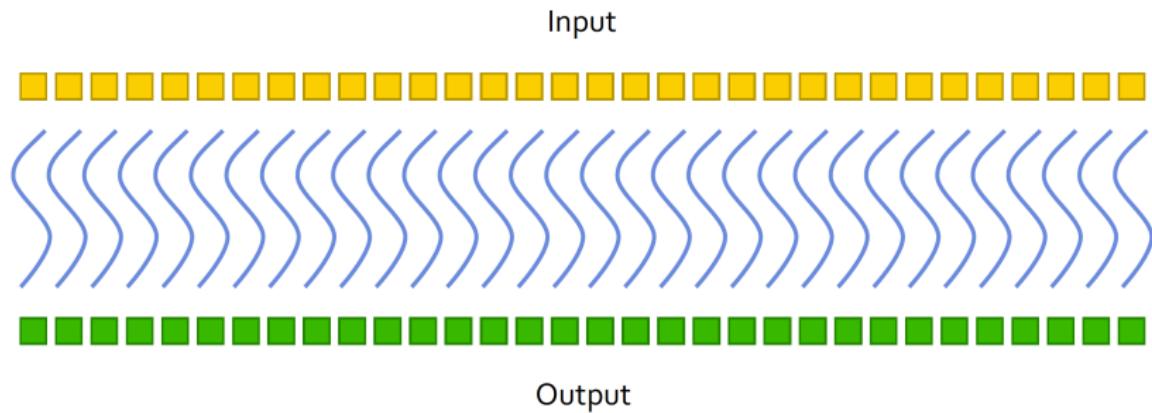
线程的组织与调度

1 CUDA 编程-3

- CUDA 基础知识 (含复习)
- 线程的组织与调度
- 程序举例：图片灰白化
- 程序举例：矩阵乘 (1)

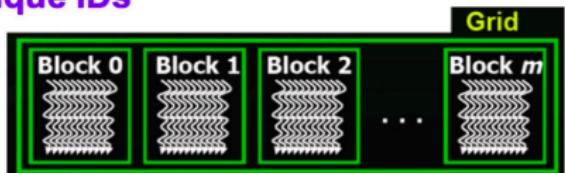
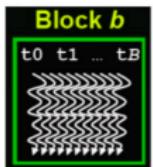
GPU 的执行模式

- Calculations on a GPU always follow the same model.
- This is very constrained.
- GPUs run kernels that are designed to execute calculations in the following way.

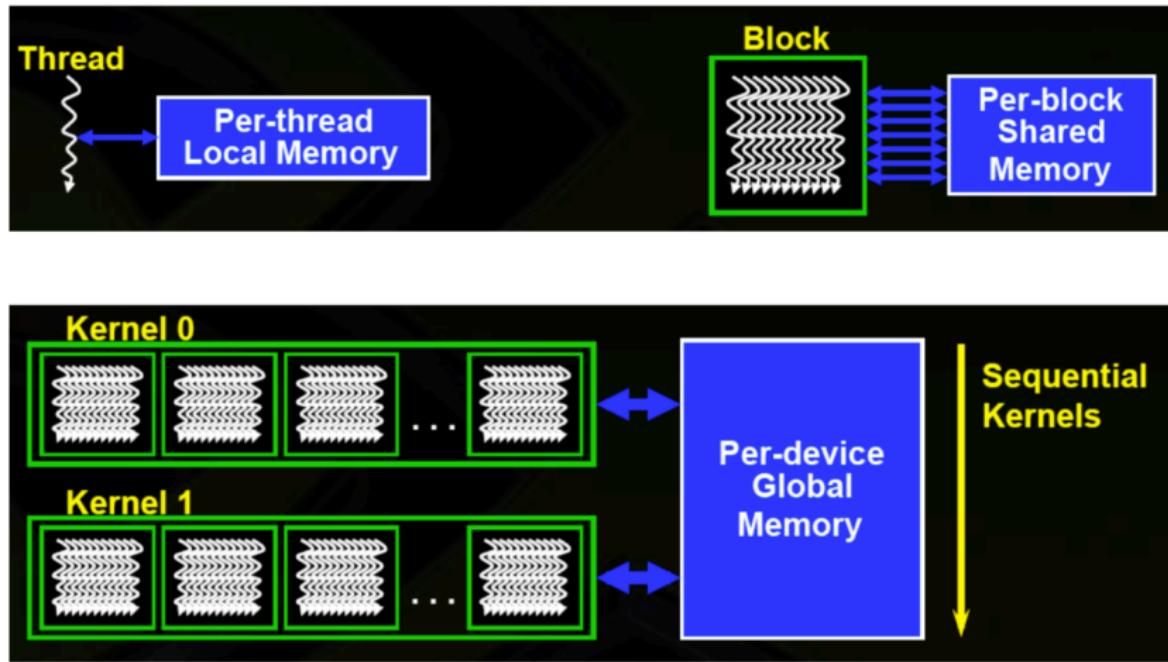


线程的层次结构

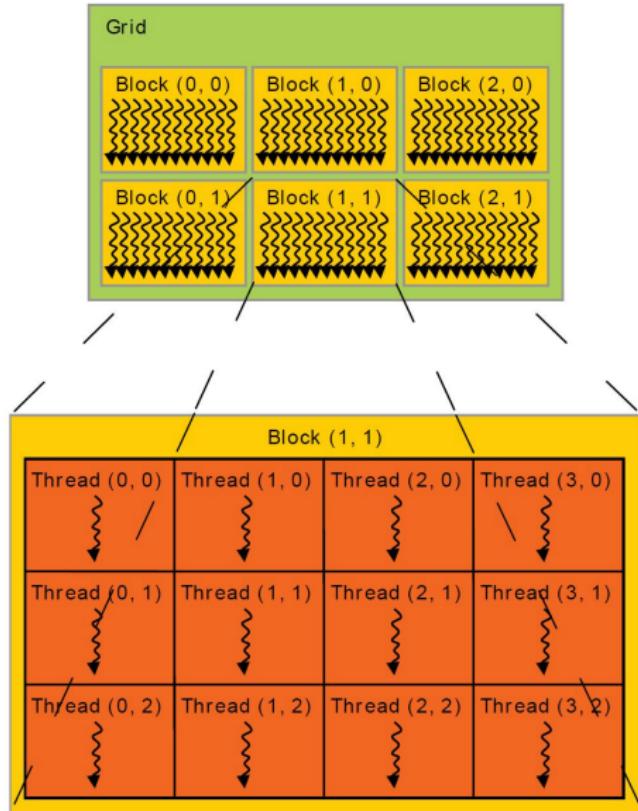
- Parallel kernels composed of many **threads**
 - all threads execute same sequential program
 - use parallel threads rather than sequential loops
- Threads are grouped into **thread blocks**
 - threads in block can sync and share memory
- Blocks are grouped into **grids**
 - threads and blocks have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



对应的存储层次结构



线程网格组织层次举例：2D

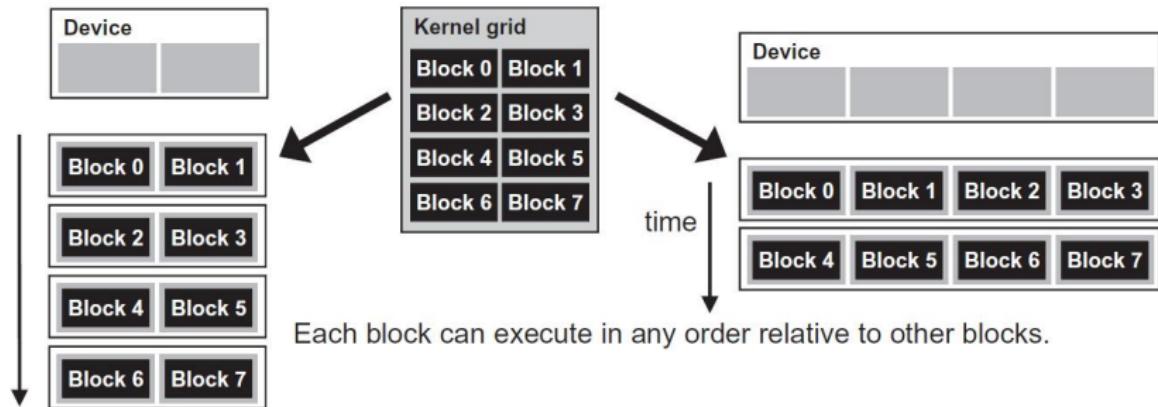


```
dim3 mygrid(3,2,1);  
dim3 myblock(4,3,1);  
mykernel<<<mygrid, myblock>>>;
```

- 左图为二维 grid，包括 3×2 个 block，所以 $\text{gridDim.x} = 3$, $\text{gridDim.y} = 2$, $\text{gridDim.z} = 1$
- 每个 block 也是二维，包含 4×3 个 thread，所以 $\text{blockDim.x} = 4$, $\text{blockDim.y} = 3$, $\text{blockDim.z} = 1$
- 注意：索引序号从 0 开始，维度顺序为 x 、 y 、 z .

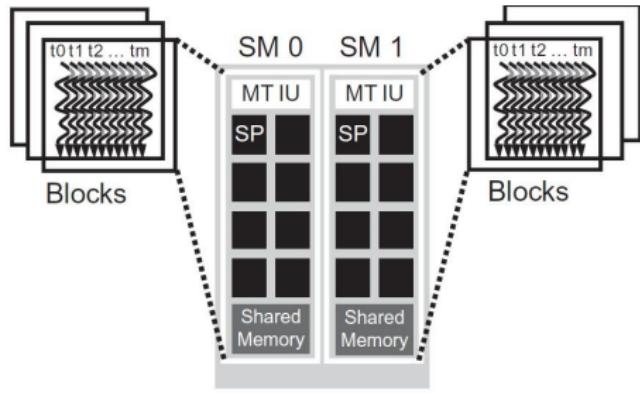
线程块间透明扩展

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.

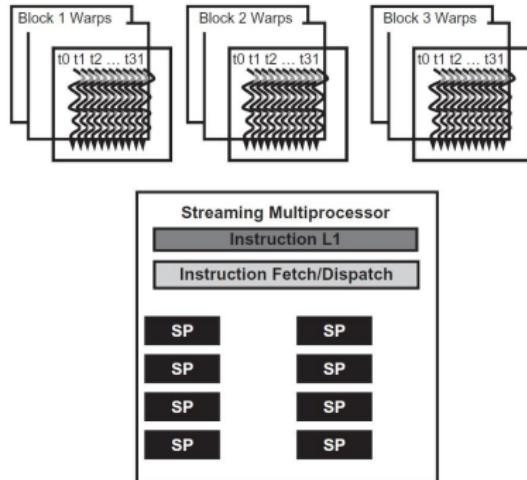


图: 每个 warp 中线程 threadIdx 值连续: 0 到 31 属于第一个 warp, 32 到 63 属于第二个 warp, 以此类推. 图中例子总共有三个线程块分配到当前 SM.

程序举例：图片灰白化

1 CUDA 编程-3

- CUDA 基础知识 (含复习)
- 线程的组织与调度
- 程序举例：图片灰白化
- 程序举例：矩阵乘 (1)

图片灰白化处理

- 图片灰白化处理的计算公式:

$$L = r \times 0.21 + g \times 0.72 + b \times 0.07$$

即对输入图片的每个像素的 RGB 三个通道的值加权平均.



C/C++ 行优先数据存储

M _{0,0}	M _{0,1}	M _{0,2}	M _{0,3}
M _{1,0}	M _{1,1}	M _{1,2}	M _{1,3}
M _{2,0}	M _{2,1}	M _{2,2}	M _{2,3}
M _{3,0}	M _{3,1}	M _{3,2}	M _{3,3}



M _{0,0}	M _{0,1}	M _{0,2}	M _{0,3}	M _{1,0}	M _{1,1}	M _{1,2}	M _{1,3}	M _{2,0}	M _{2,1}	M _{2,2}	M _{2,3}	M _{3,0}	M _{3,1}	M _{3,2}	M _{3,3}
------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------



M ₀	M ₁	M ₂	M ₃	M ₄	M ₅	M ₆	M ₇	M ₈	M ₉	M ₁₀	M ₁₁	M ₁₂	M ₁₃	M ₁₄	M ₁₅
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

CPU 程序

```
1 #define CHANNELS 3
2 void cpu_pic2grey(unsigned char * Pout, unsigned char * Pin,
3                     int width, int height) {
4     for (int i = 0; i < height; ++i) {
5         for (int j = 0; j < width; ++j) {
6             int grey_offset = i * width + j;
7             int rgb_offset = grey_offset * CHANNELS;
8             unsigned char r = Pin[rgb_offset + 0];
9             unsigned char g = Pin[rgb_offset + 1];
10            unsigned char b = Pin[rgb_offset + 2];
11            Pout[grey_offset] = 0.21f * r + 0.71f * g + 0.07f * b;
12        }
13    }
```

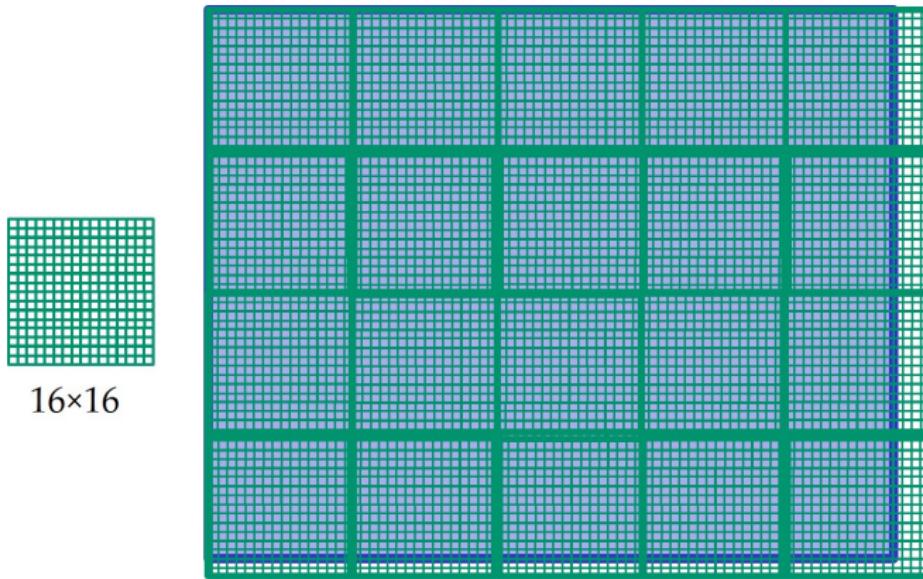
GPU 线程设置

- 设线程块大小为 16×16 , 执行配置:

```
1 ...
2 // Allocate and copy data
3 cudaMalloc((void **)&d_out, size_out);
4 cudaMalloc((void **)&d_in, size_in);
5 cudaMemcpy(d_in, h_in, size_in, cudaMemcpyHostToDevice);
6 // Process the  $m \times n$  input image
7 dim3 grid_dim(ceil(m/16.0), ceil(n/16.0), 1);
8 dim3 block_dim(16, 16, 1);
9 gpu_pic2grey_kernel<<<grid_dim,block_dim>>>(d_out,d_in,m,n);
10 // Transfer back and free data
11 cudaMemcpy(h_out, d_out, size_out, cudaMemcpyDeviceToHost);
12 cudaFree(d_out); cudaFree(d_in);
13 ...
```

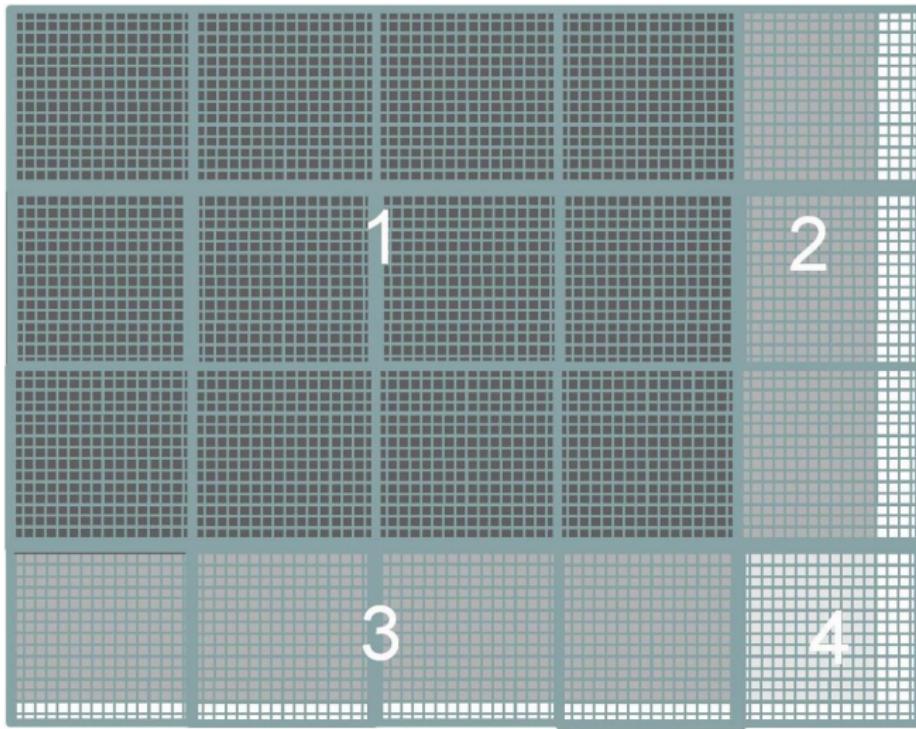
边界处理

用 16×16 的线程块处理 76×62 像素的图片数据。每个线程处理一个像素，为了处理所有像素，线程网格为 80×64 ，导致 x 方向多出 4 个额外线程， y 方向多处 2 个额外线程，需要做判断防止越界。



四种不同的任务负载

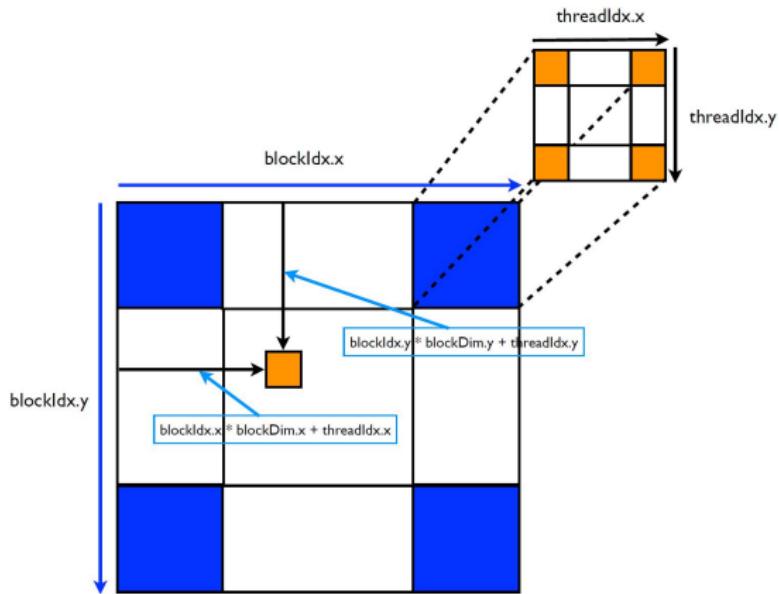
根据图片灰白化处理的计算过程，任务负载有四种不同情形：



计算线程的全局坐标

以 2D grid 和 2D block 为例，线程全局坐标计算公式：

- $x = \text{threadIdx.x} + \text{blockIdx.x} \times \text{blockDim.x}$
- $y = \text{threadIdx.y} + \text{blockIdx.y} \times \text{blockDim.y}$



GPU 端的 kernel 程序

```
1 #define CHANNELS 3
2 __global__
3 void gpu_pic2grey_kernel(unsigned char * Pout, unsigned char *
4     Pin, int width, int height) {
5     int Col = threadIdx.x + blockIdx.x * blockDim.x;
6     int Row = threadIdx.y + blockIdx.y * blockDim.y;
7     if (Col < width && Row < height) {
8         int grey_offset = Row * width + Col;
9         int rgb_offset = grey_offset * CHANNELS;
10        unsigned char r = Pin[rgb_offset + 0];
11        unsigned char g = Pin[rgb_offset + 1];
12        unsigned char b = Pin[rgb_offset + 2];
13        Pout[grey_offset] = 0.21f * r + 0.71f * g + 0.07f * b;
14    }
}
```

程序结果分析

- 编译：

```
module load gcc/6.5.0
module load cuda/9.0
module load ffmpeg/4.1
module load opencv/4.0.0
make
```

- 运行：

```
height: 716 width: 1080
CPU: 0.00731 sec
grid dim: 68, 45, 1.
block dim: 16, 16, 1.
kernel time: 0.00004 sec.
GPU: 0.16426 sec
```

程序举例：矩阵乘 (1)

1 CUDA 编程-3

- CUDA 基础知识 (含复习)
- 线程的组织与调度
- 程序举例：图片灰白化
- 程序举例：矩阵乘 (1)

CPU 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 __host__
2 void cpu_mat_mul(float* M, float* N, float* P, int width) {
3     for (int i = 0; i < width; i++) {
4         for (int j = 0; j < width; j++) {
5             float sum = 0.0;
6             for (int k = 0; k < width; k++) {
7                 sum += M[i * width + k] * N[k * width + j];
8             }
9             P[i * width + j] = sum;
10        }
11    }
12 }
```

GPU 矩阵乘的并行策略

- 每个线程计算 P 矩阵一个元素；
- 线程块按照 $BLOCK_WIDTH \times BLOCK_WIDTH$ 二维方式组织；
- 网格按照 $(width/BLOCK_WIDTH) \times (width/BLOCK_WIDTH)$ 二维方式组织； .

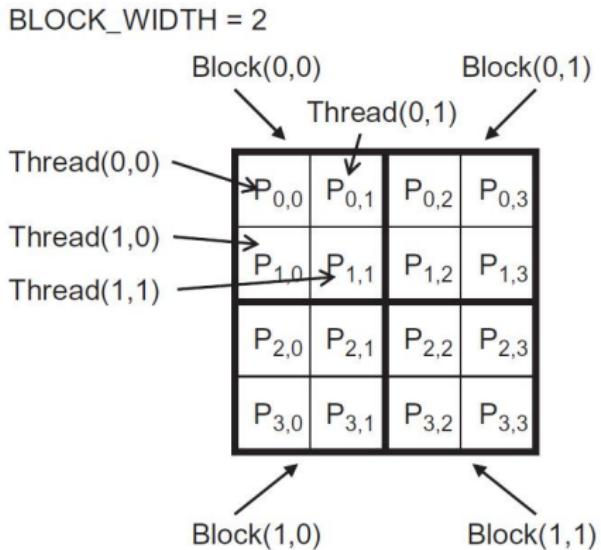
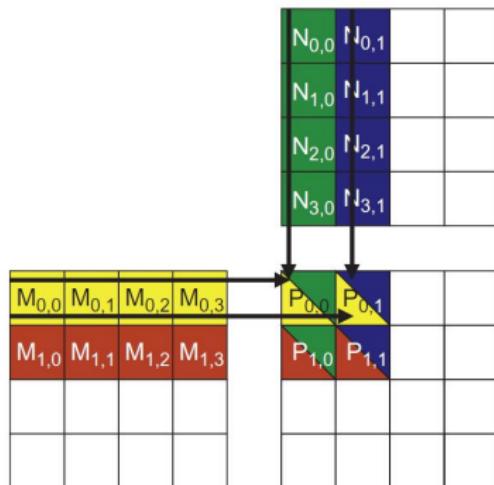


图: 矩阵宽为 4, 线程块宽为 2.

线程索引

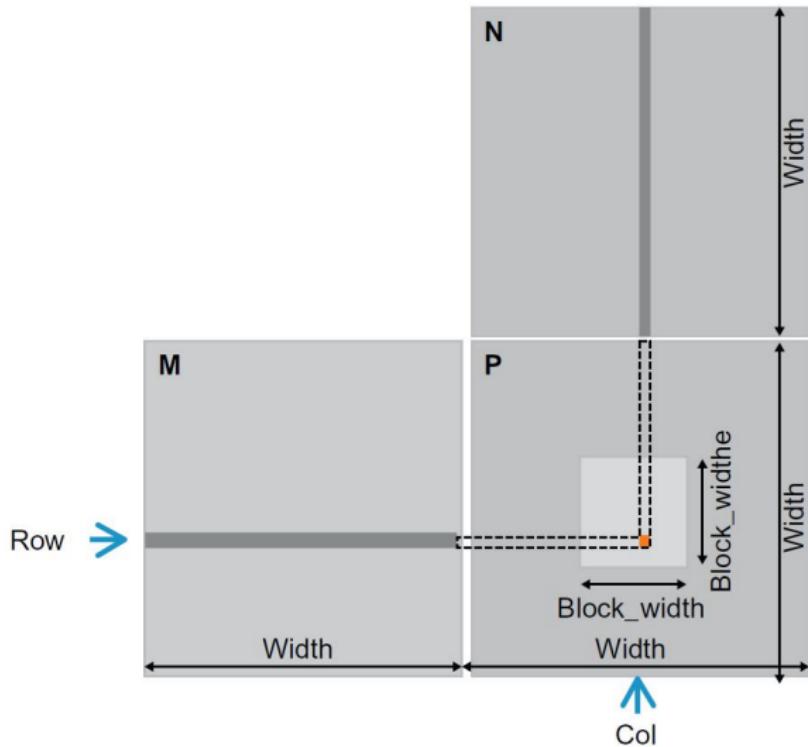
单个线程所计算的行列位置通用公式：

```
Row = blockIdx.y * blockDim.y + threadIdx.y;  
Col = blockIdx.x * blockDim.x + threadIdx.x;
```



图：矩阵宽为 4、线程块宽为 2 时，线程块 (0, 0) 的计算过程.

分块方式



GPU 矩阵乘 kernel 程序

```
1  __global__ void gpu_mat_mul_kernel(float* M, float* N, float* P
2      , int width) {
3
4      int Row = blockIdx.y * blockDim.y + threadIdx.y;
5      int Col = blockIdx.x * blockDim.x + threadIdx.x;
6
7      float sum = 0;
8      for (int k = 0; k < width; k++) {
9          sum += M[Row * width + k] * N[k * width + Col];
10     }
11
12 }
```

GPU 执行配置

```
1 #define BLOCK_WIDTH 16
2 ...
3 // Setup the execution configuration
4 dim3 grid_dim(width/BLOCK_WIDTH, width/BLOCK_WIDTH, 1);
5 dim3 block_dim(BLOCK_WIDTH, BLOCK_WIDTH, 1);
6 // Launch the device computation threads
7 gpu_mat_mul_kernel<<<grid_dim, block_dim>>>(d_M, d_N, d_P,
8 width);
9 ...
```

程序结果分析

- 测试结果：

```
Matrix width: 512.  
CPU: 0.89709 sec  
grid dim: 32, 32, 1.  
block dim: 16, 16, 1.  
kernel time: 0.00056 sec  
GPU: 0.15642 sec  
GPU all values correct
```

```
Matrix width: 1024.  
CPU: 8.21264 sec  
grid dim: 64, 64, 1.  
block dim: 16, 16, 1.  
kernel time: 0.00500 sec  
GPU: 0.16927 sec  
GPU all values correct
```