并行与分布式计算基础 IV：CUDA 编程与实践

杨超
chao_yang@pku.edu.cn

2020 秋

内容提纲

内容提纲

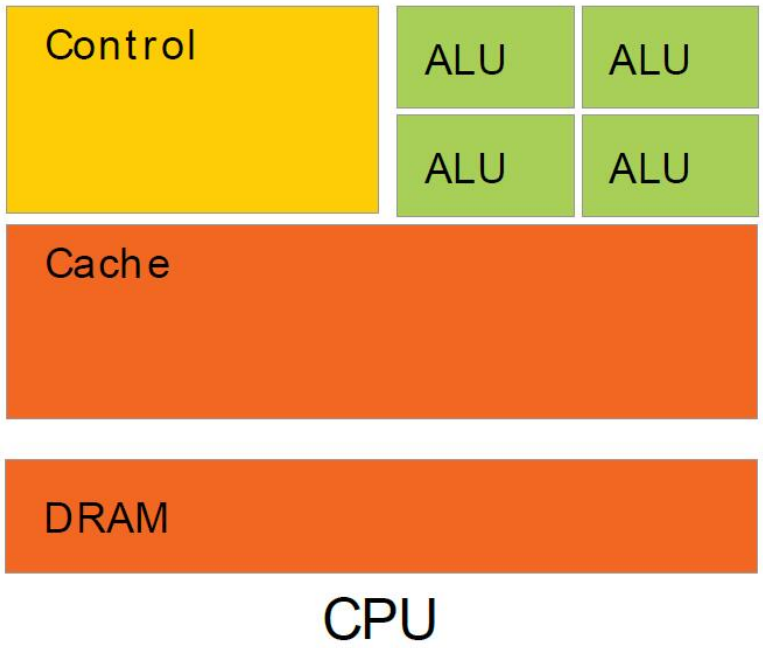
CPU 和 GPU 对比

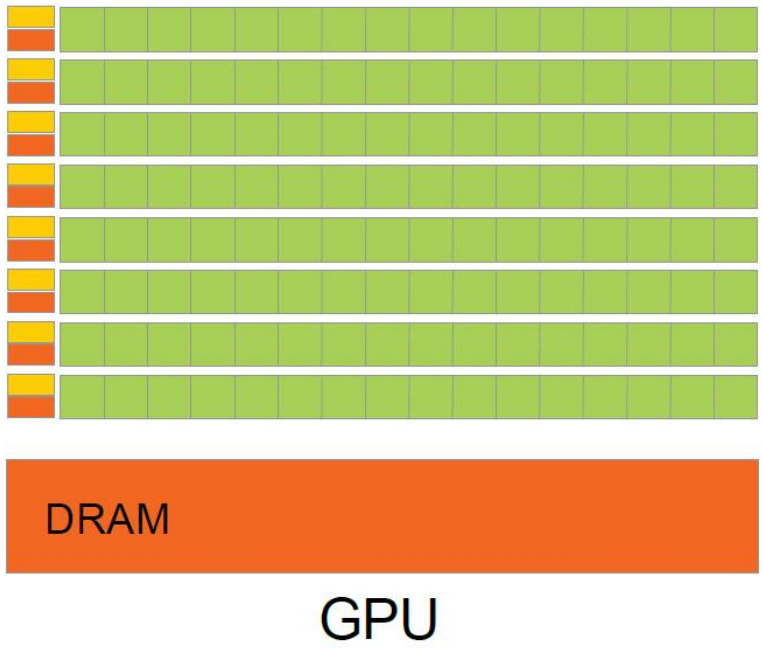
CPU = Central Processing Unit:

- 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.





等一下！GPU 不是打游戏的吗?

从 2005 年左右说起：GPGPU (General Purpose GPU)

Stream-based programming model

Express algorithms in terms of graphics operations

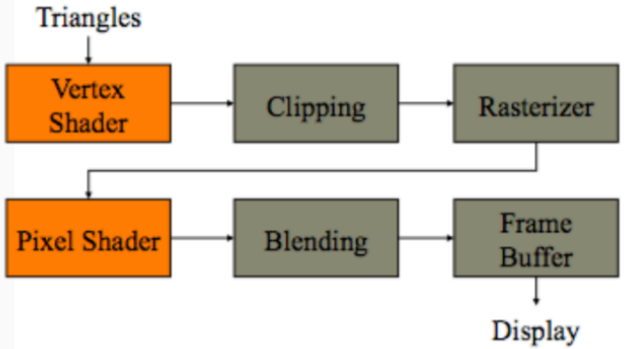
—use GPU pixel shaders as general-purpose SP floating point units

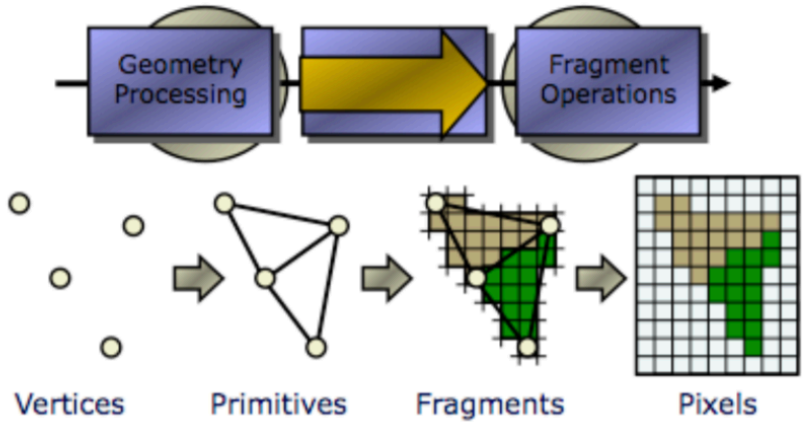
Directly exploit

—pixel shaders

—vertex shaders

—video memory

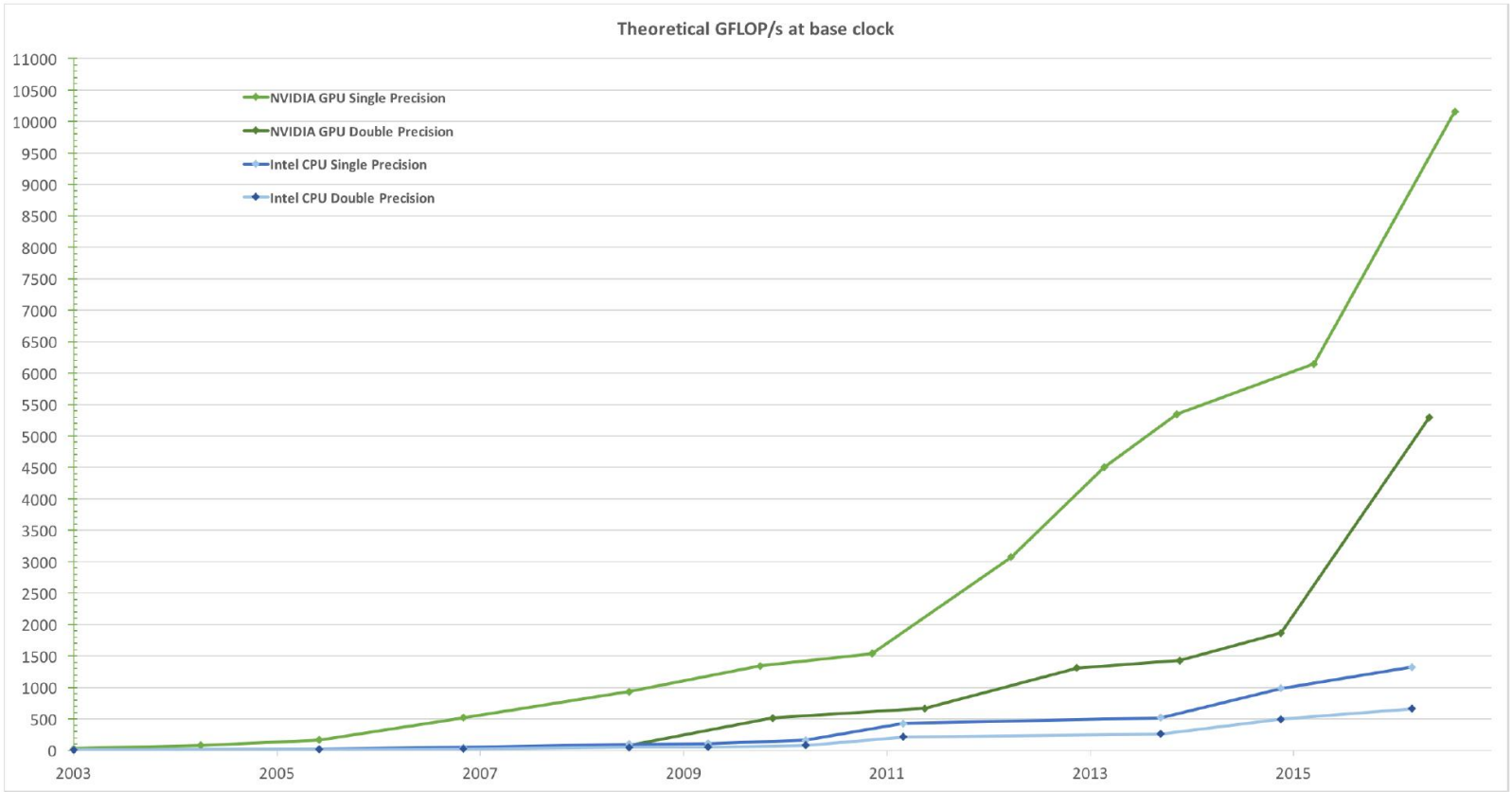




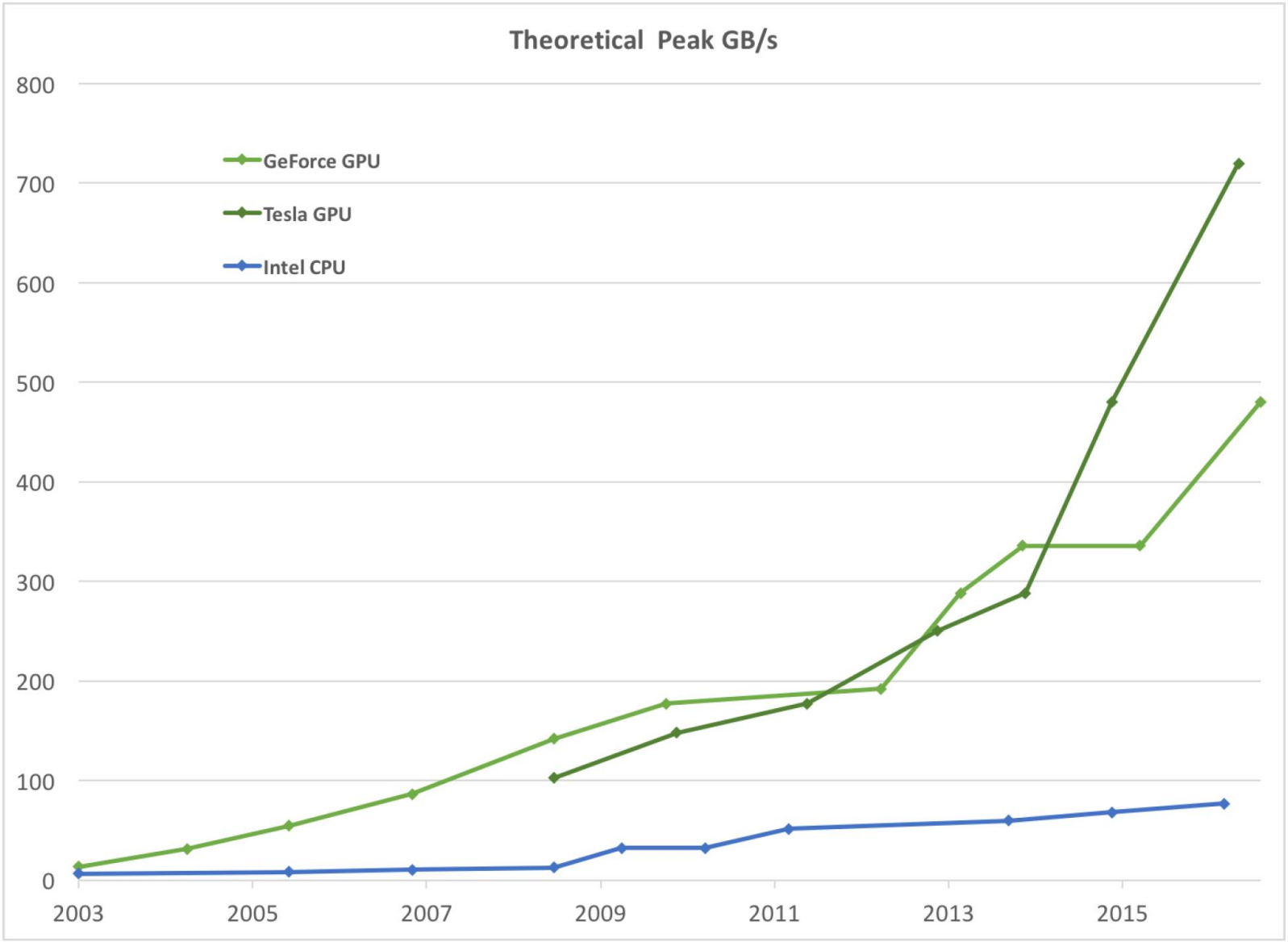
threads interact through off-chip video memory

Example: GPUSort (Govindaraju, Manocha; 2005)

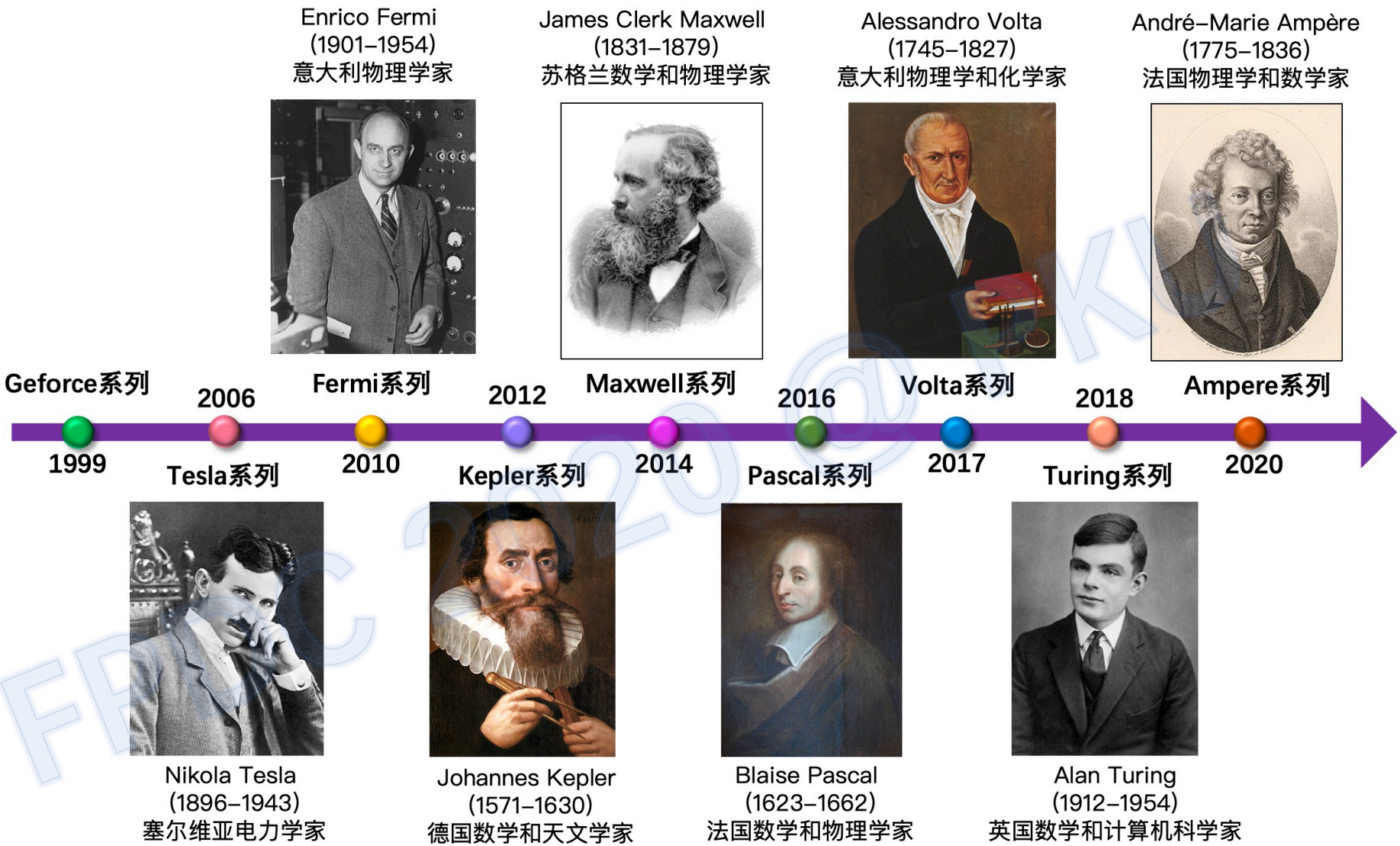
GPU 浮点计算能力的优势



GPU 访存能力的优势


Theoretical Peak GB/s
800
700
600
500
400
300
200
100
0
GeForce GPU
Tesla GPU
Intel CPU
2003
2005
2007
2009
2011
2013
2015

NVIDIA 的 GPU 微架构分类



NVIDIA 的 GPU 产品分类

桌面 (desktop)
GeForce系列

例：GeForce GTX 780
发布：2012.03
架构：Kepler GK110

移动 (mobile)
GeForce M系列

例：GeForce GTX 970M
发布：2014.10
架构：Maxwell GM204

工作站 (workstation)
Quadro系列

例：Quadro FX 5800
发布：2008.11
架构：Tesla GT200GL

高性能计算 (HPC)
Tesla系列

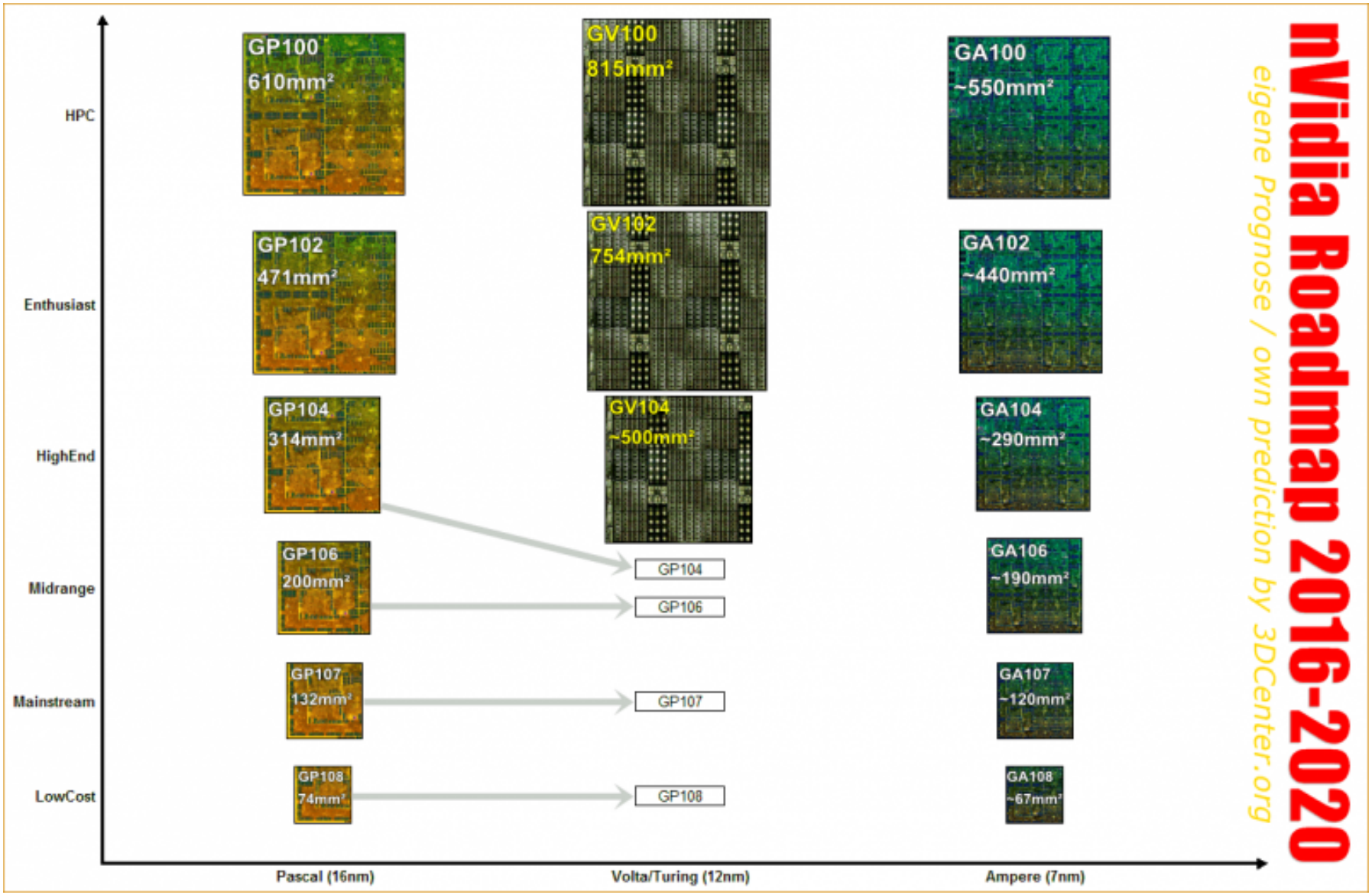
例：Tesla V100
发布：2017.06
架构：Volta GV100

Nvidia早期产品：Riva, Vanta, TNT, Tegra, ...

此外，还有Titan等融合了不同特色的系列产品

更多信息： https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

NVIDIA 的 2016-2020 路线图 (旧)



几台典型的 GPU 超级计算机

- 中国天河 1 号 A (2010): 7,168 Tesla M2050 GPUs (架构：GF100)；
- 美国 Titan (2012): 18,688 Tesla K20x GPUs (架构：GK110)；
- 日本 TSUBAME 3.0 (2017): 2,160 Tesla P100 GPUs (架构：GP100)；
- 瑞士 Piz Daint (2017): 5,704 Tesla P100 GPUs (架构：GP100)；
- 美国 Summit (2018): 27,648 Tesla V100 GPUs (架构：GV100)；
- ...
- 数院机器 (2018): 40 Titan Xp (GeForce) GPUs (架构：GP102).

Tesla P100 的计算性能

- 双精度峰值（FP64）5.3 TFLOPS
- 单精度峰值（FP32）10.6 TFLOPS
- 半精度峰值（FP16）21.2 TFLOPS
- 相对于 CPU，对于 DNN 的不同应用来说，加速 10 到 20 倍

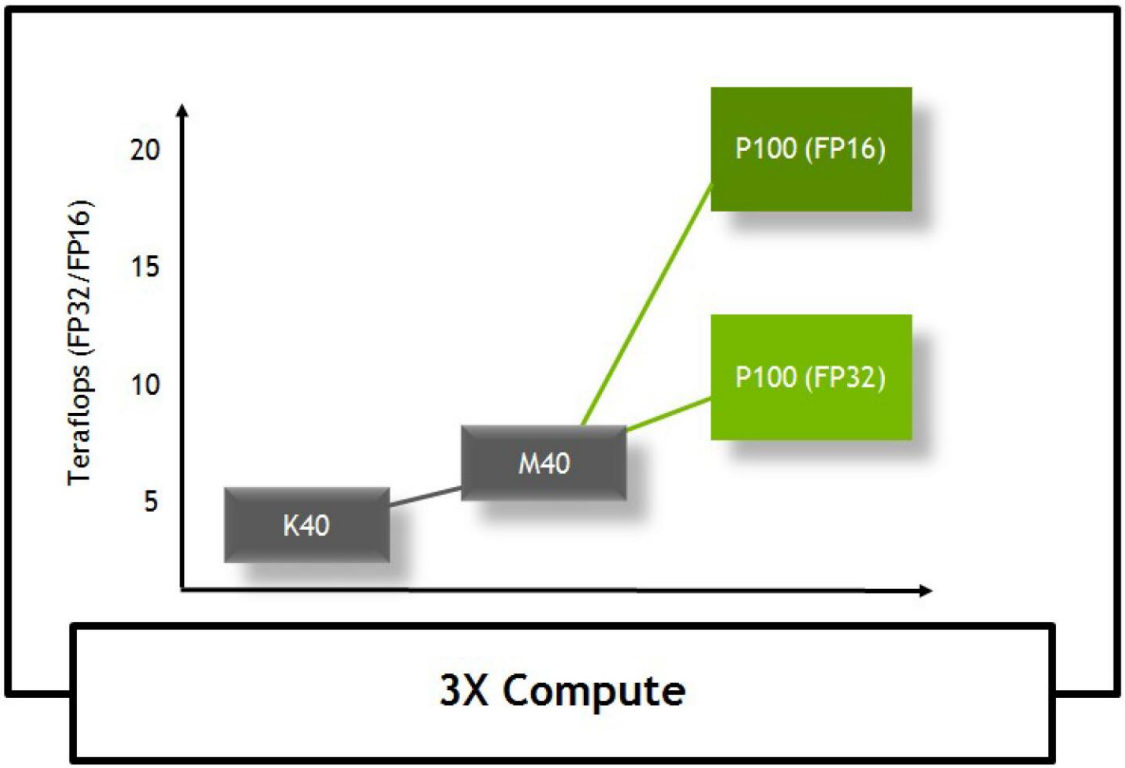
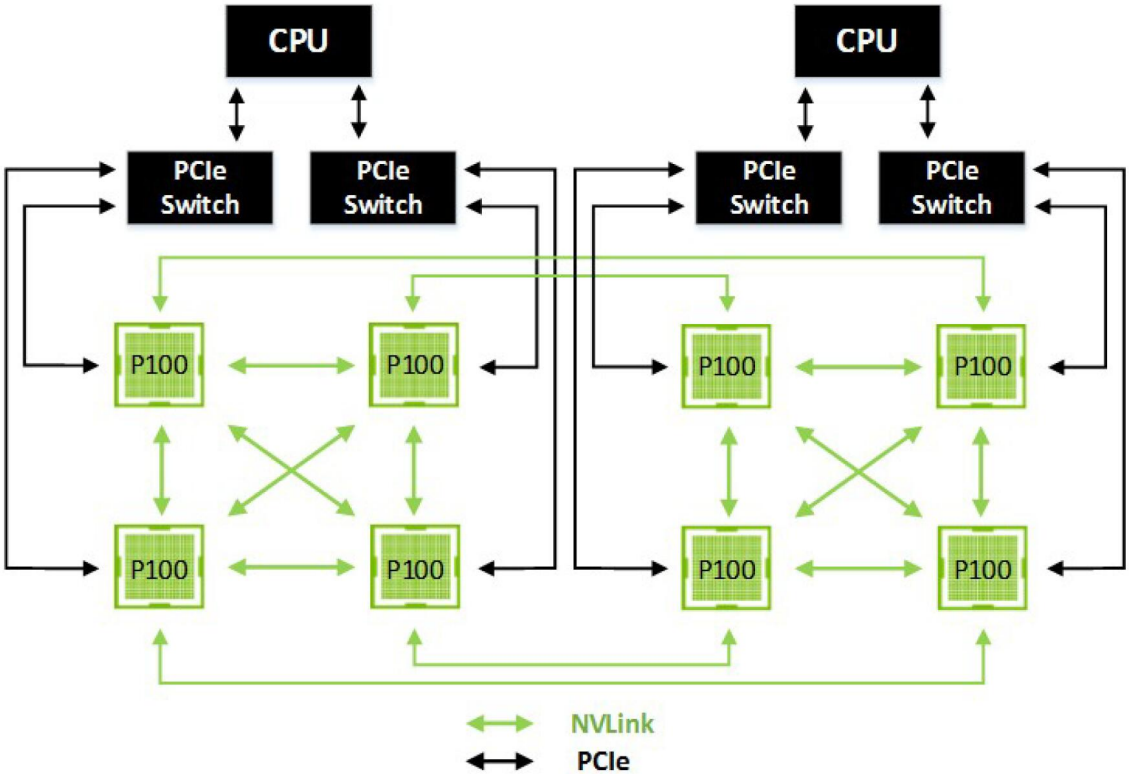


图: P100 和过去两代 GPU 的计算性能对比

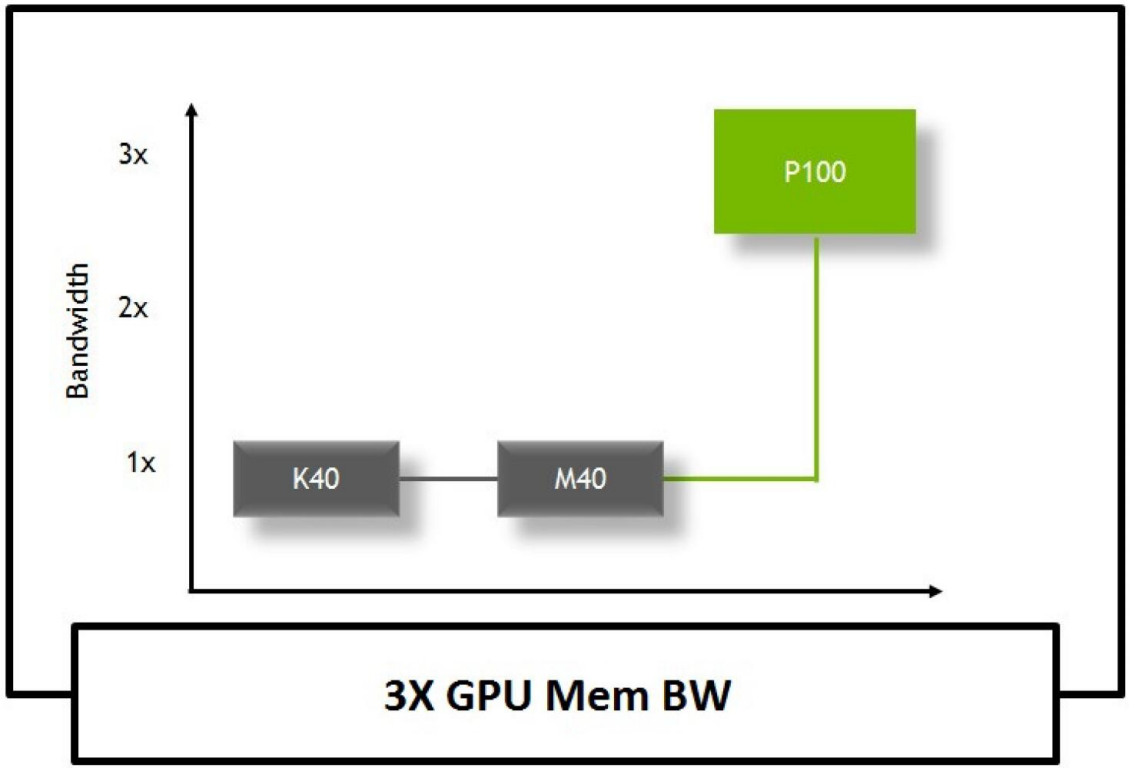
Tesla P100 的芯片互连

- NVLINK 能够在 GPU-to-GPU 间提供 160 Gigabytes/second 的双向带宽，相对于 PCIe Gen 3 x16 来说，数据传输能力提升 5 倍.



Tesla P100 的访存带宽

- 在 P100 中，首次采用了 High Bandwidth Memory 2 (HBM2) 堆叠式存储，带宽相对于 Maxwell GM200 GPU 提升了 3 倍.



Titan Xp (GP102) vs Tesla P100 (GP100)

Model	NVIDIA Tesla P100	NVIDIA Titan Xp
Core name	GP100	GP102
Transisitor Count	15.3B	12B
Die Size	610mm^2	471mm^2
Manufacturing Process	TSMC 16nm FF+	TSMC 16nm FF+
Stream Processors	3584	3840
SM or Cus	56 SM	60 SM
FP32 Cores	3584	3840
FP64 Cores	1792（1：2）	120（1：32）
Tensor Cores	N/A	N/A
TA/TF	224	240
ROP/RBE	?	96

Boost Clock	1480MHz	1582MHz
FP16 TFLOPS	21TFLOPS	24TFLOPS
FP32 TFLOPS	10.6TFLOPS	12.1TFLOPS
FP64 TFLOPS	5.3TFLOPS	0.38TFLOPS
Tensor TFLOPS	N/A	N/A
L2 Cache	4096KB	4096KB
Memory Size	16GB	12GB
Memory Clock	1.44Gbps	11.4Gbps
Memory Interface	4096bit HBM2	384bit GDDR5X
Memory Bandwidth	720GB/s	547.7GB/s
PCI-E/NVLink	NVLink/PCI-E	PCI-E
TDP	300W	250W

Tesla V100 的计算性能

- 双精度峰值（FP64）7.8 TFLOPS (P100: 5.3 TFLOPS)
- 单精度峰值（FP32）15.7 TFLOPS (P100: 10.6 TFLOPS)
- 半精度峰值（FP16*）125 TFLOPS (P100: 21.2 TFLOPS)

  *: 基于 Tensor Core 张量计算核心的混合精度实现

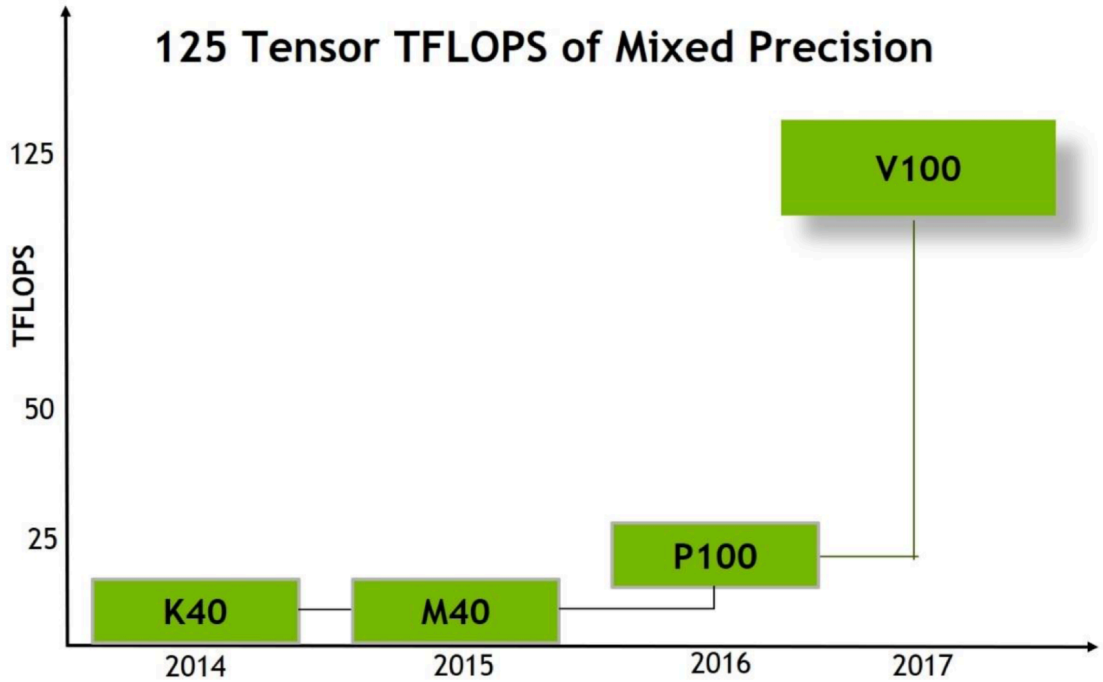
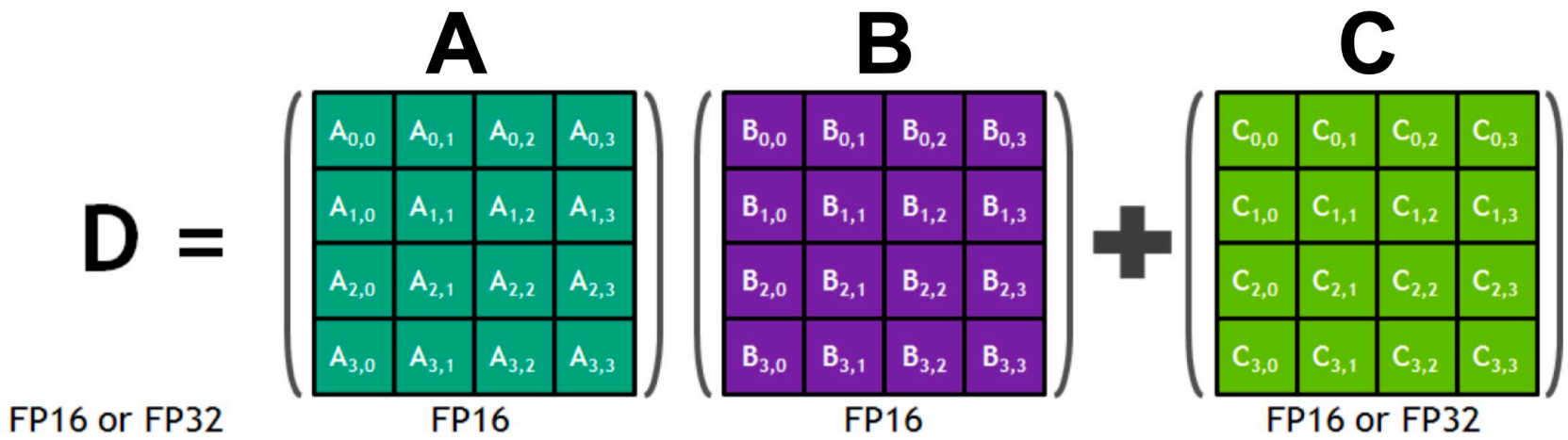
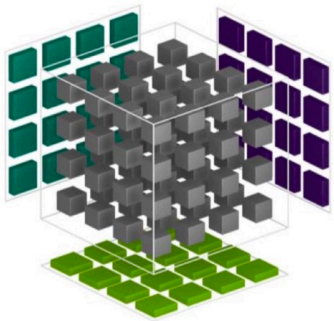


图: V100 和过去三代 GPU 的计算性能对比

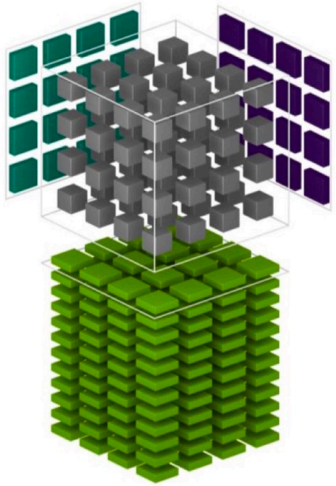
Tesla V100 的 Tensor Core



Pascal



Volta Tensor Core



Tensor Core GEMM

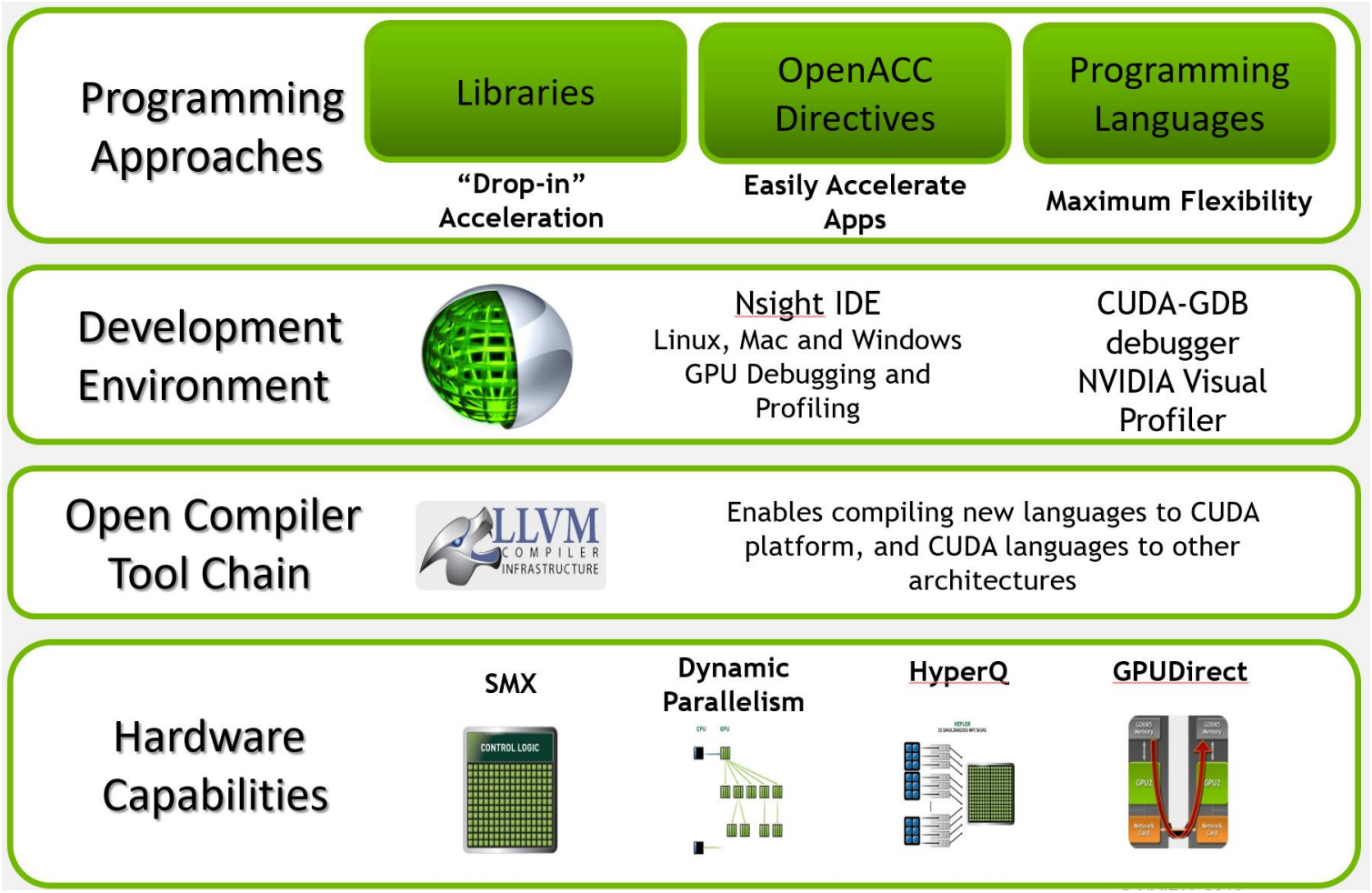
- A, B, C, D are 4x4 matrices
- A and B are FP16
- C and D are FP16 or FP32
- Products are FP64
- Additions are FP32

Note: GEMM=General Matrix Multiplication

内容提纲

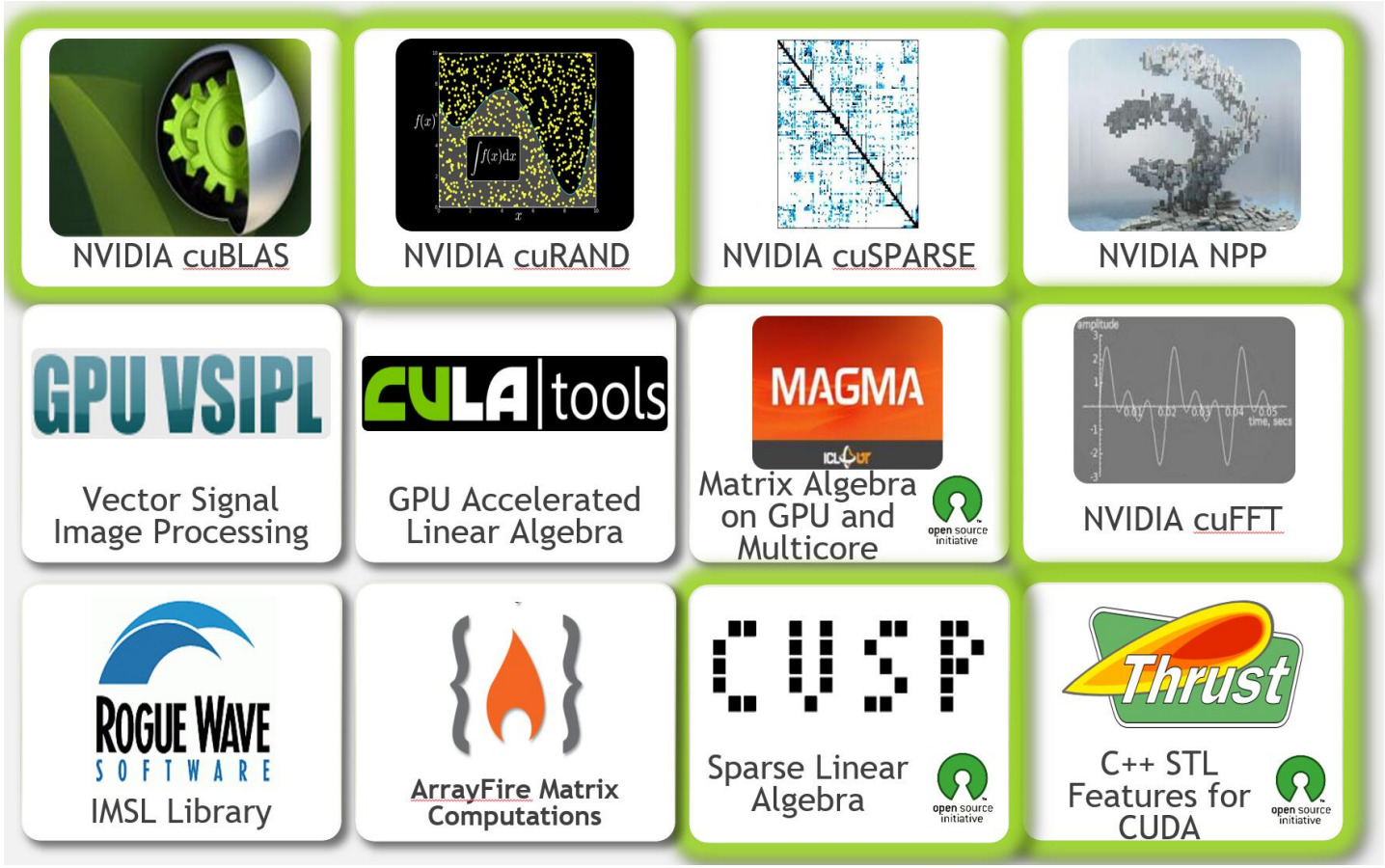
CUDA 并行编程平台

CUDA (Compute Unified Device Architecture)：由 NVIDIA 公司于 2006 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.

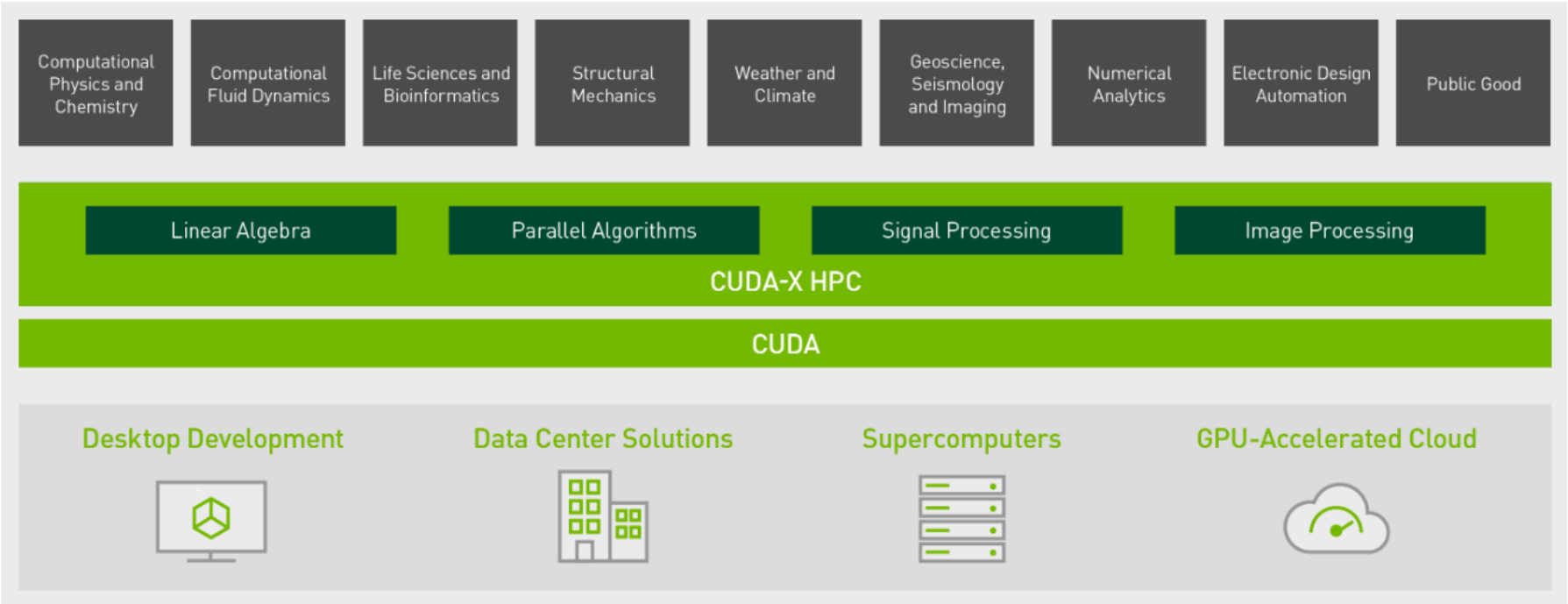


方式一：使用 CUDA 计算库

- 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.

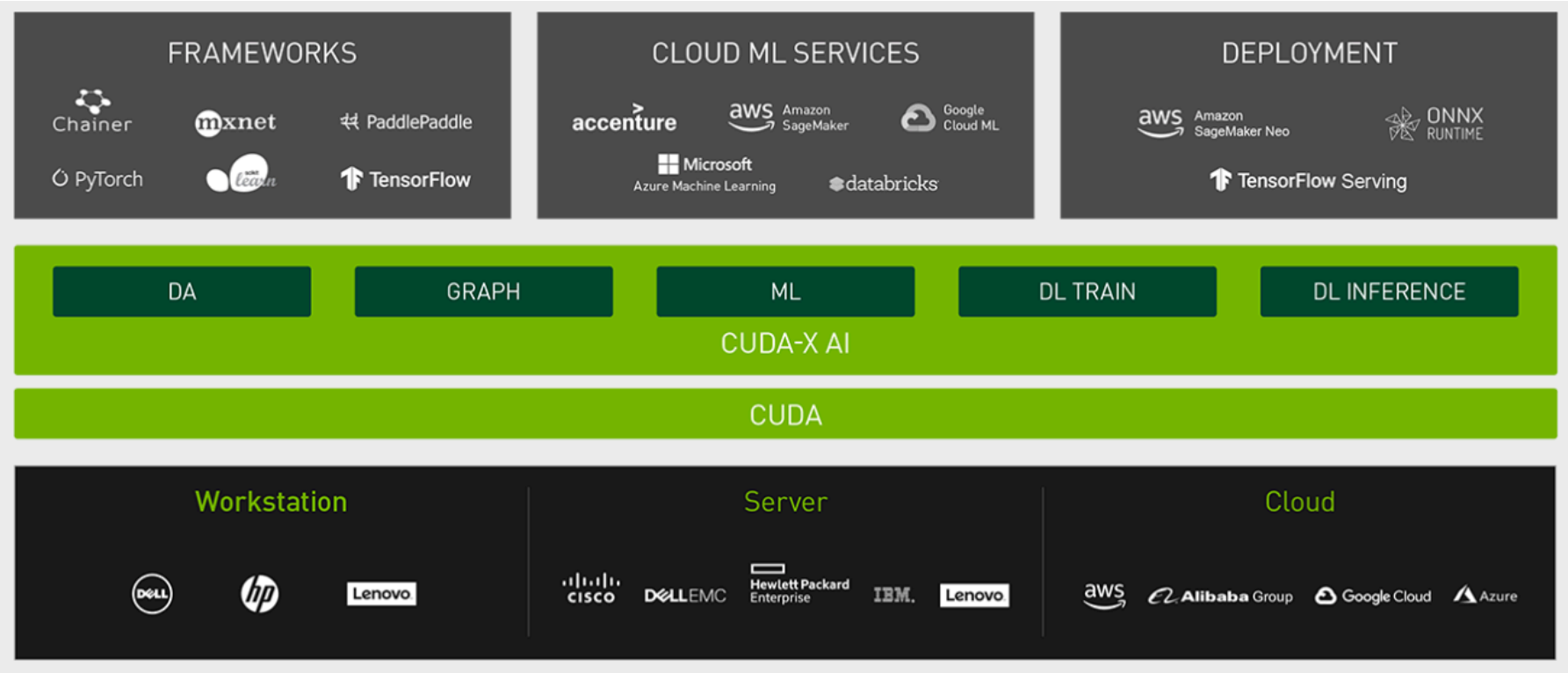


CUDA-X：GPU-accelerated libraries for AI and HPC.



图：CUDA-X HPC.

CUDA-X：GPU-accelerated libraries for AI and HPC.



图：CUDA-X AI.

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.

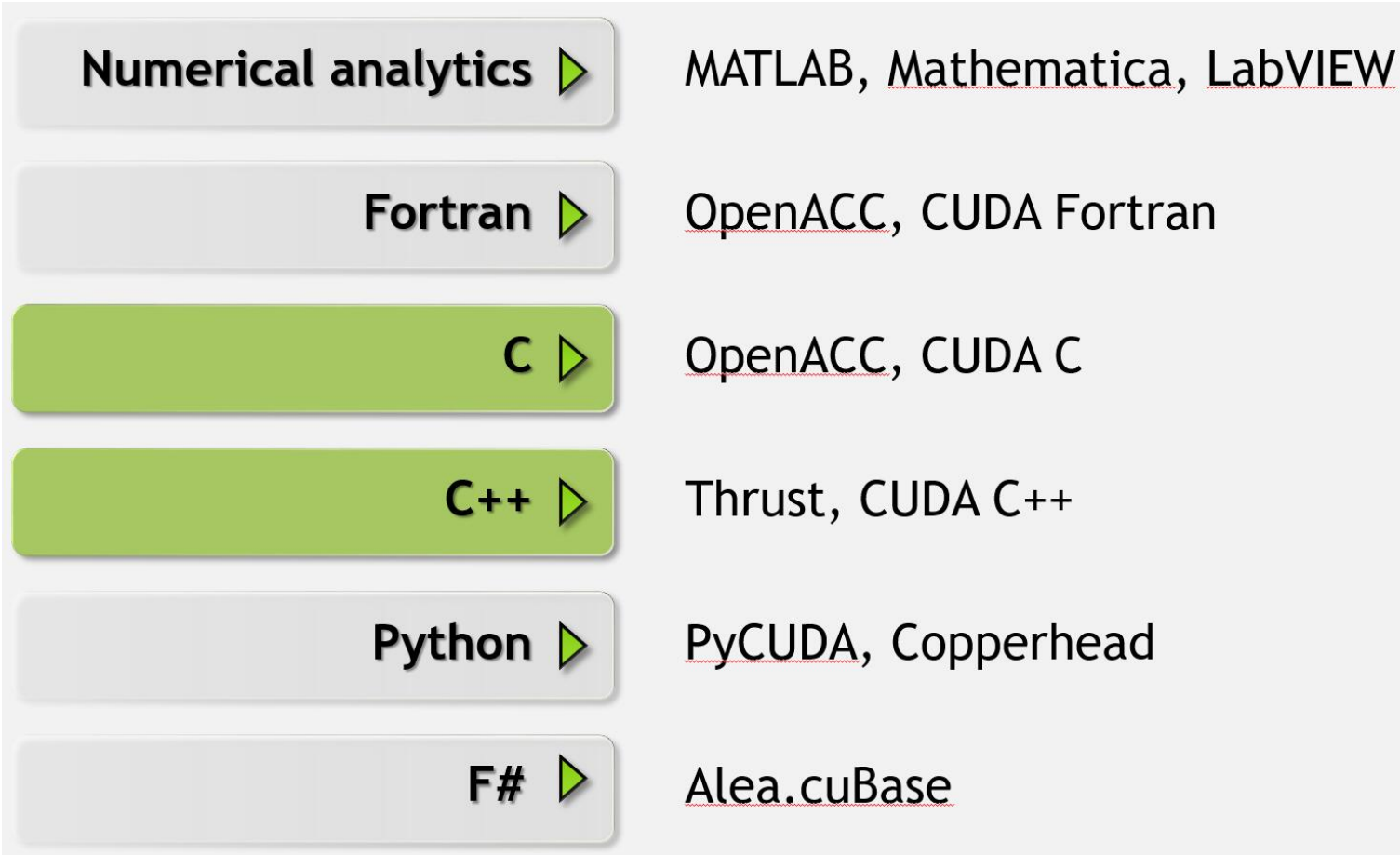
```c
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{

  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

```cuda
__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);
```

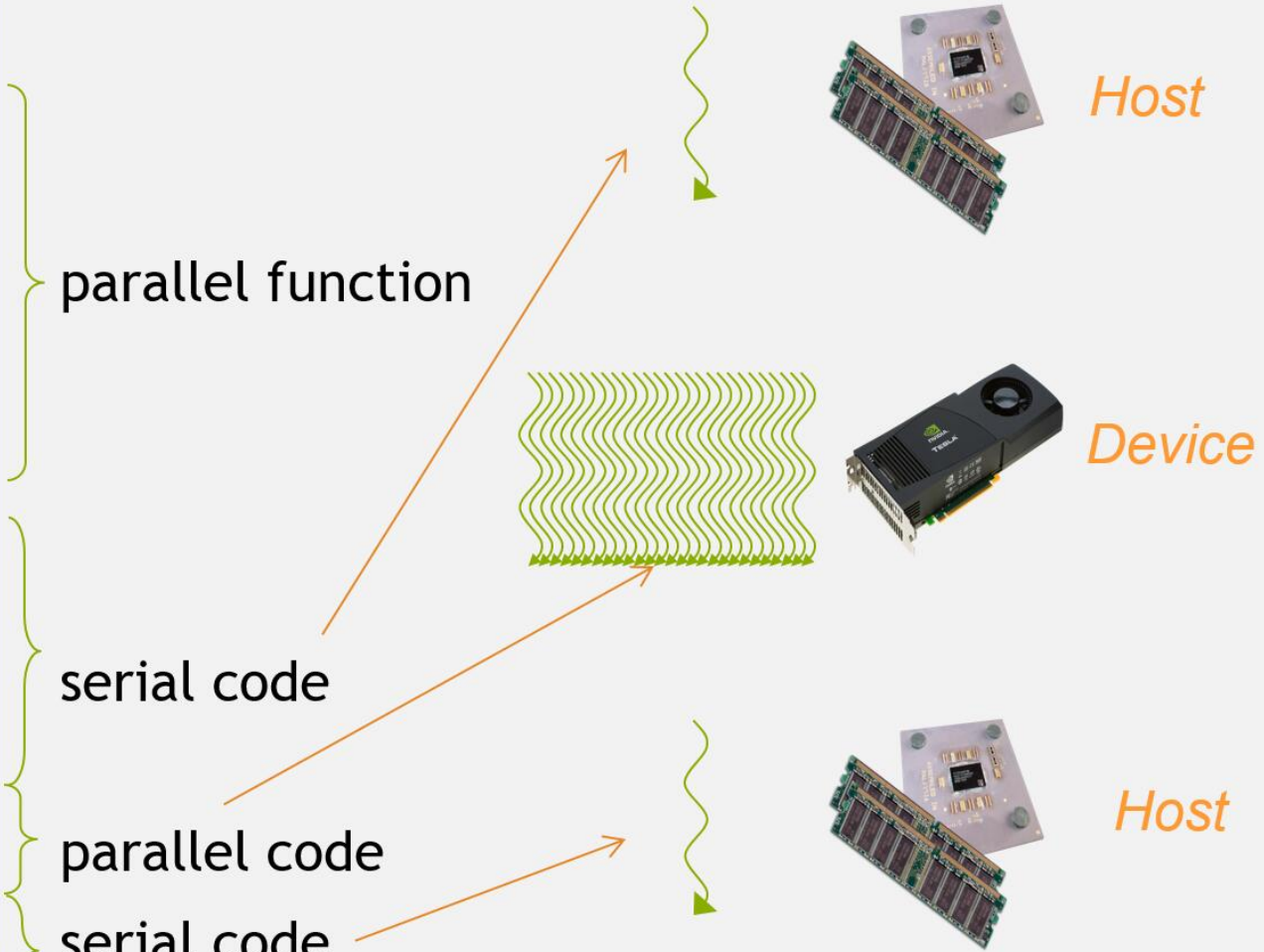
CUDA

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.

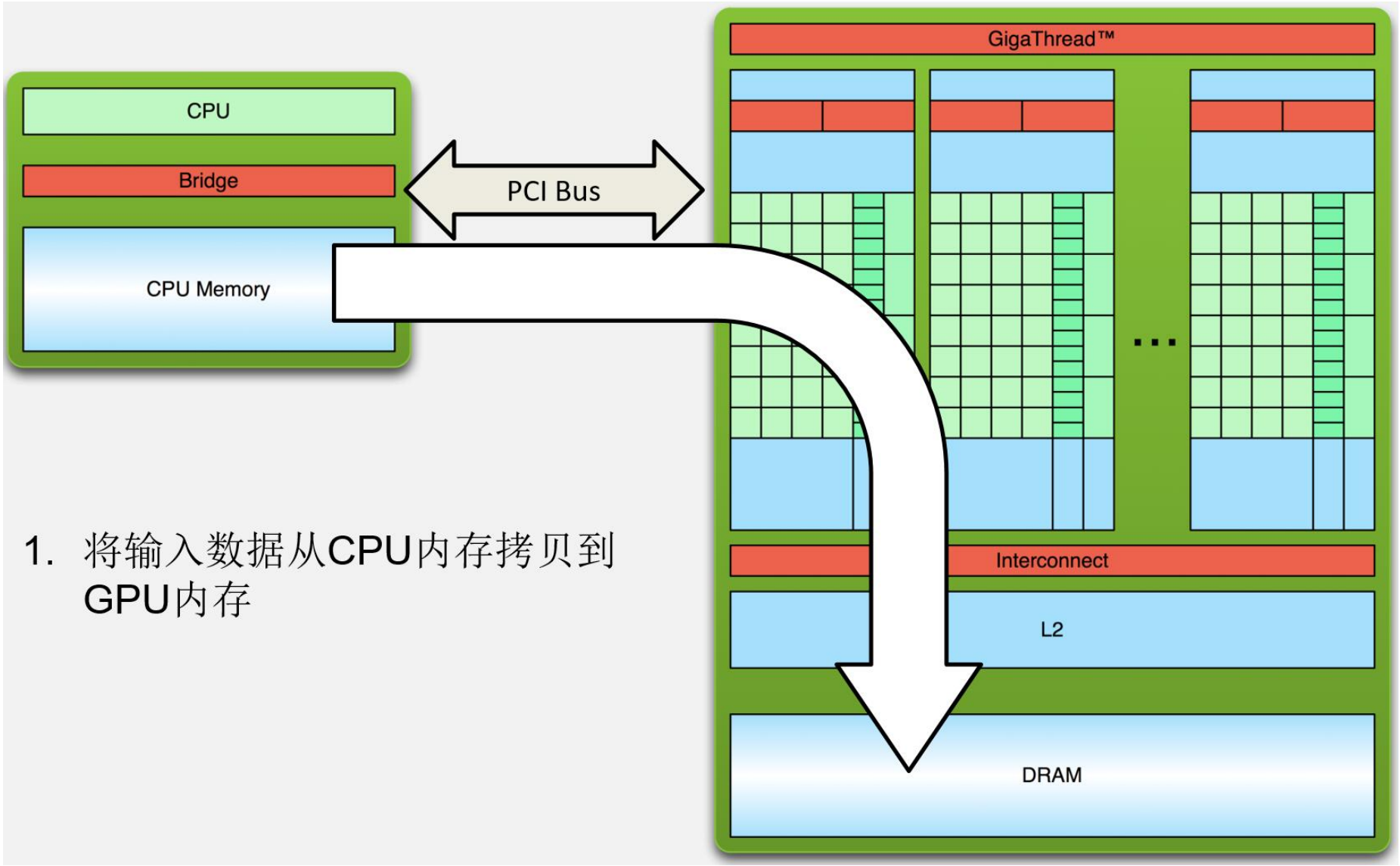


CUDA 异构编程方式

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

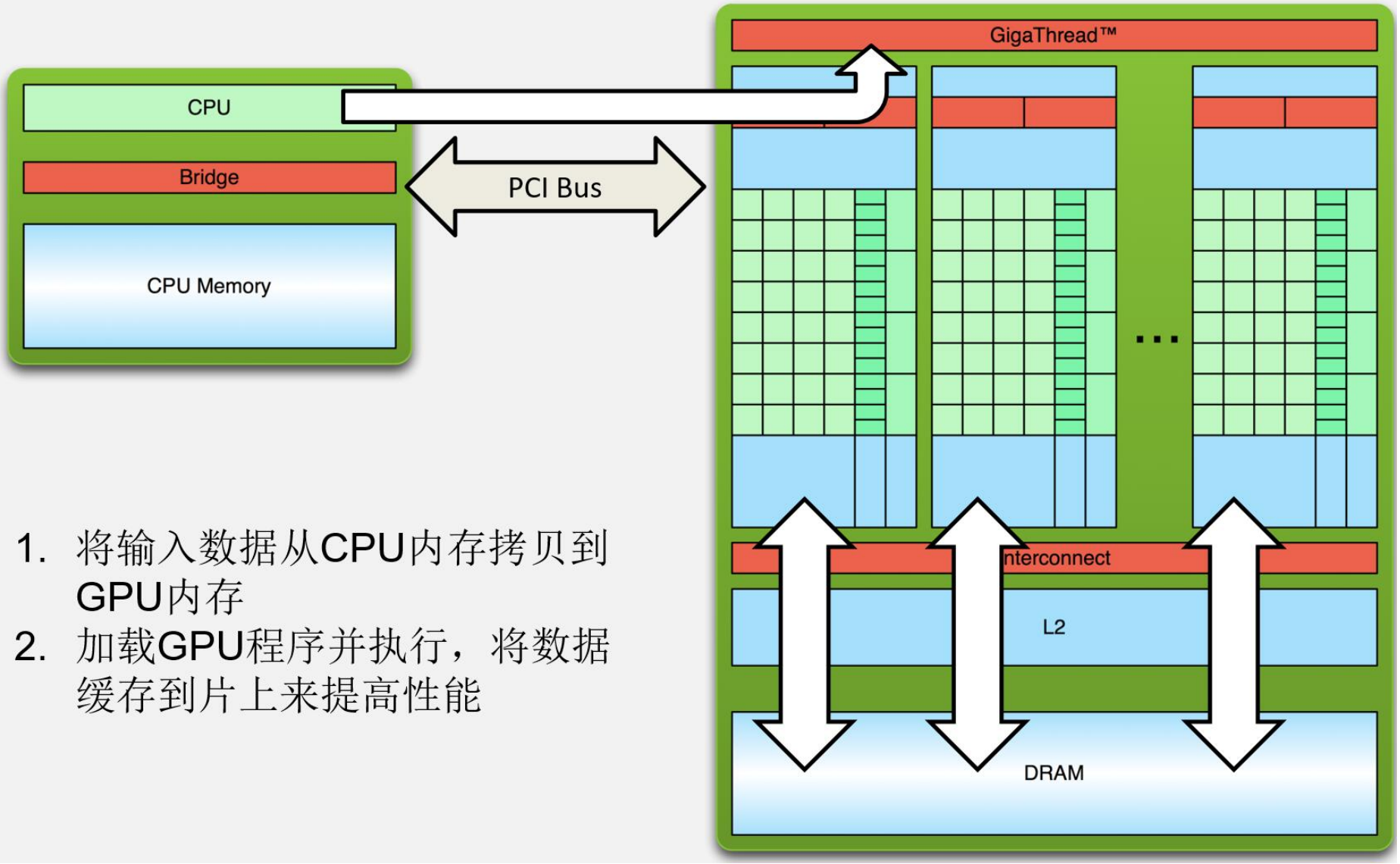


CUDA 异构并行计算流程



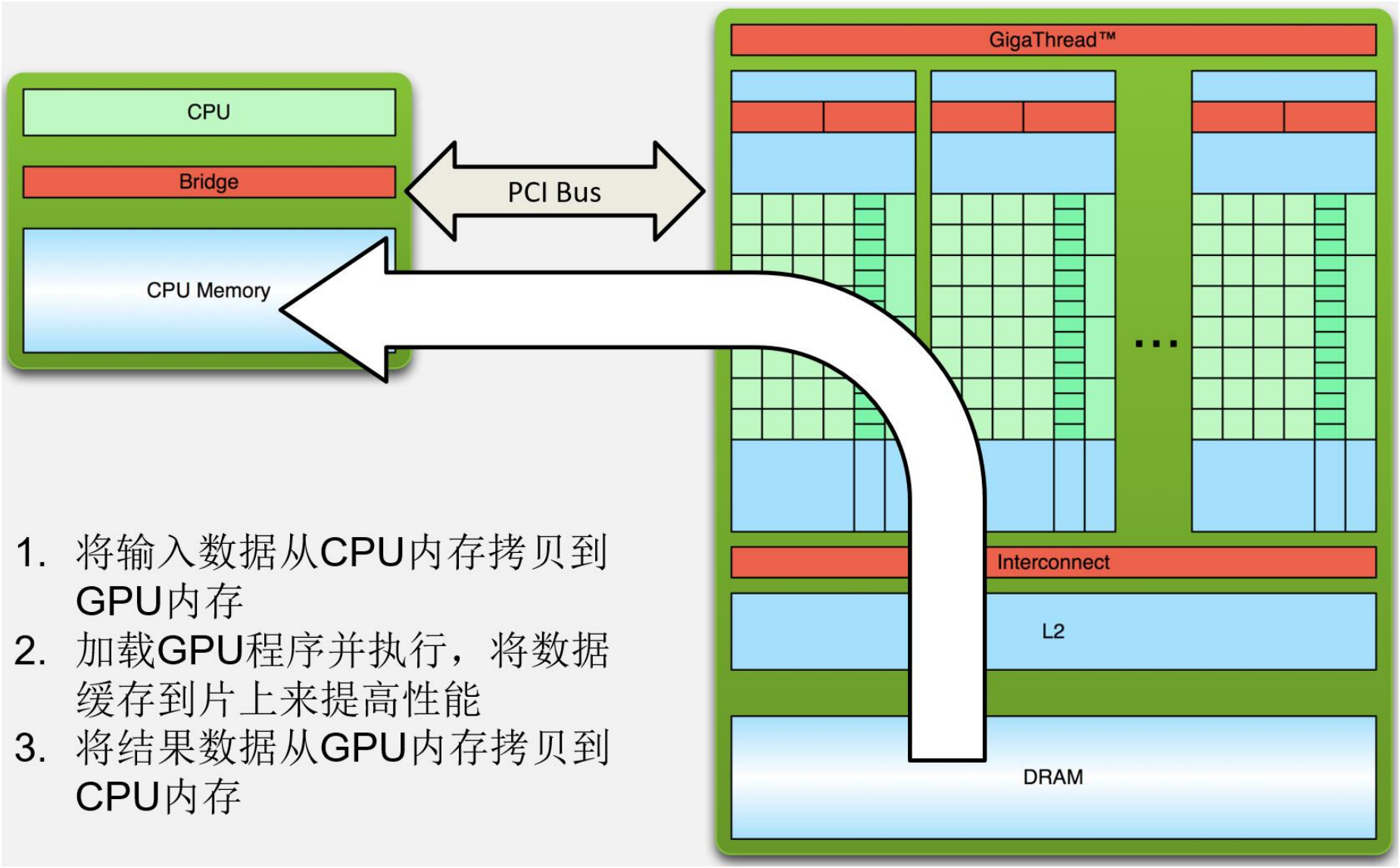
1. 将输入数据从CPU内存拷贝到GPU内存

CUDA 异构并行计算流程



1. 将输入数据从CPU内存拷贝到GPU内存
2. 加载GPU程序并执行，将数据缓存到片上来提高性能

CUDA 异构并行计算流程



1. 将输入数据从CPU内存拷贝到GPU内存
2. 加载GPU程序并执行，将数据缓存到片上来提高性能
3. 将结果数据从GPU内存拷贝到CPU内存

第一个 CUDA 程序：hello world!

cuda_hello.cu

```
1  __global__ void mykernel(void) {
2    printf("Hello World from GPU by thread %d!\n", threadIdx.x);
3  }
4  int main(void) {
5    // Use 1 thread
6    mykernel<<<1,1>>>();
7    // Flush the output
8    cudaDeviceSynchronize();
9    // Use 4 threads
10   mykernel<<<1,4>>>();
11   // Flush the output
12   cudaDeviceSynchronize();
13   return 0;
14 }
```

程序的编译与运行

- 加载编译器：

```
$ module load cuda
```

- 编译：

```
$ nvcc -o hello cuda_hello.cu
```

- 运行 (请正确使用 sbatch 或者 salloc)：

```
$ ./hello
```

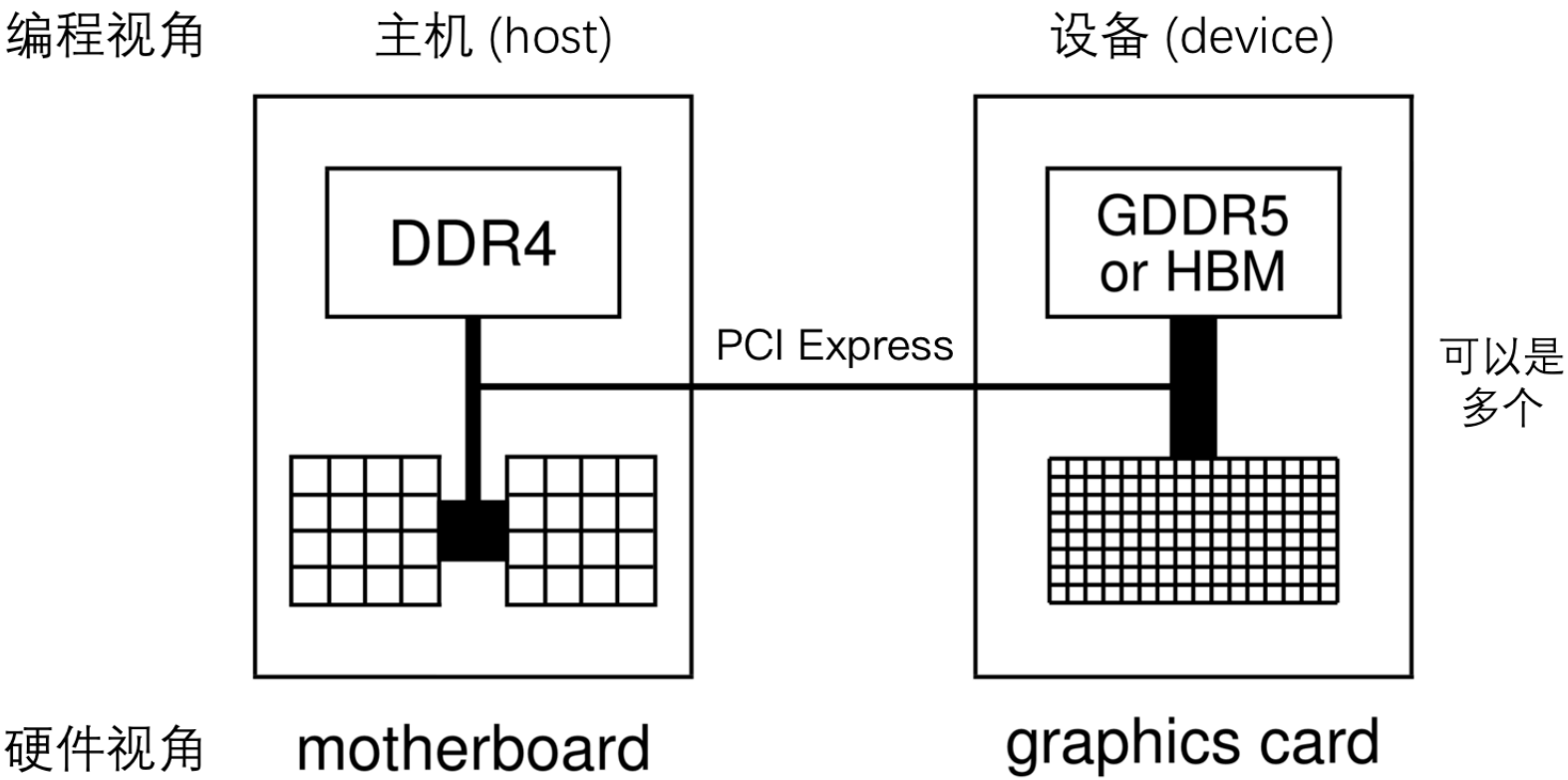
运行结果

- 运行结果：

```
Hello World from GPU by thread 0!
Hello World from GPU by thread 0!
Hello World from GPU by thread 1!
Hello World from GPU by thread 2!
Hello World from GPU by thread 3!
```

内容提纲

硬件视角与编程视角


编程视角
主机 (host)
设备 (device)
DDR4
GDDR5
or HBM
PCI Express
可以是
多个
硬件视角
motherboard
graphics card

Nvidia GPU 硬件总体架构简介

- 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.

SIMT 与线程簇

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.

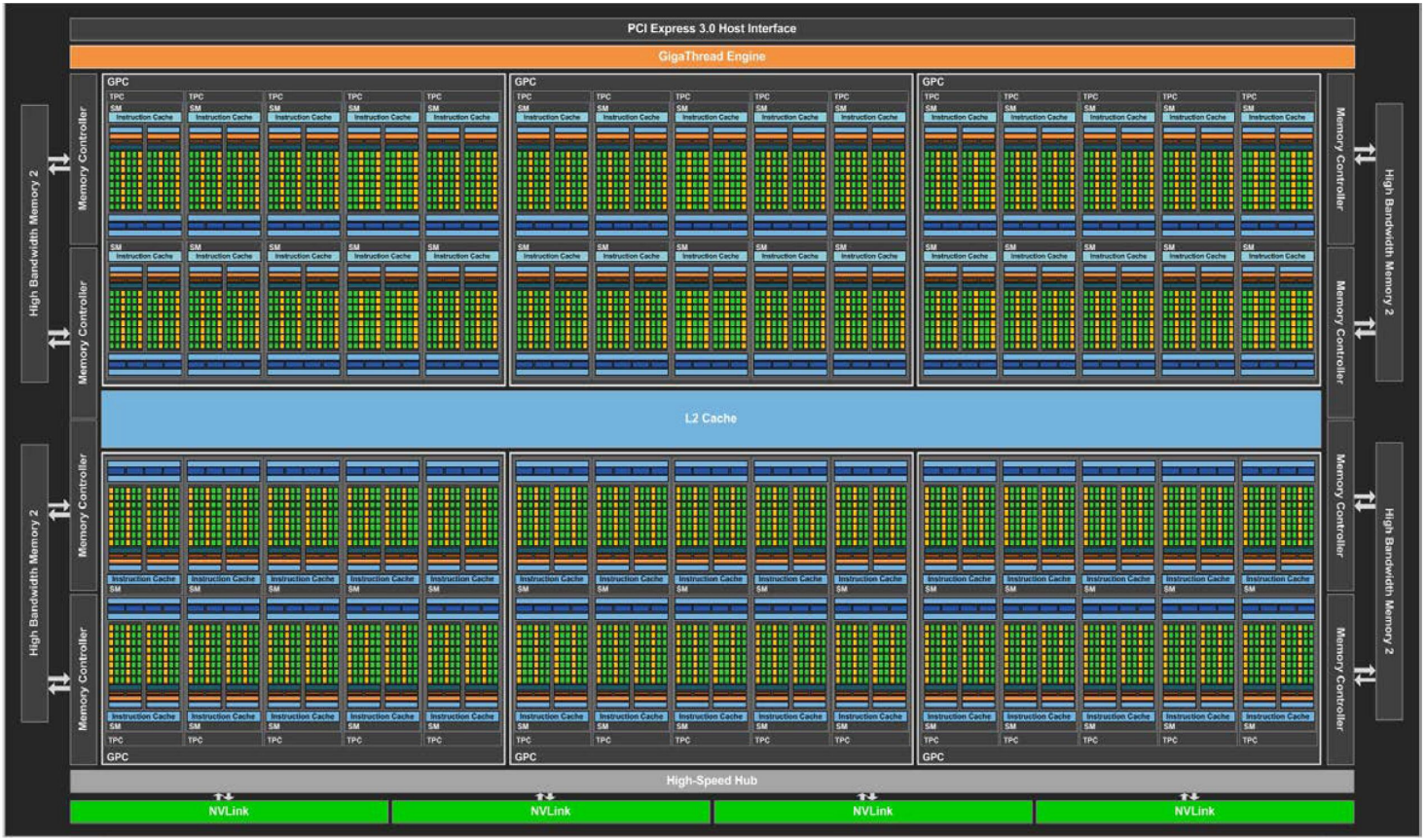
为什么性能高?

- 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

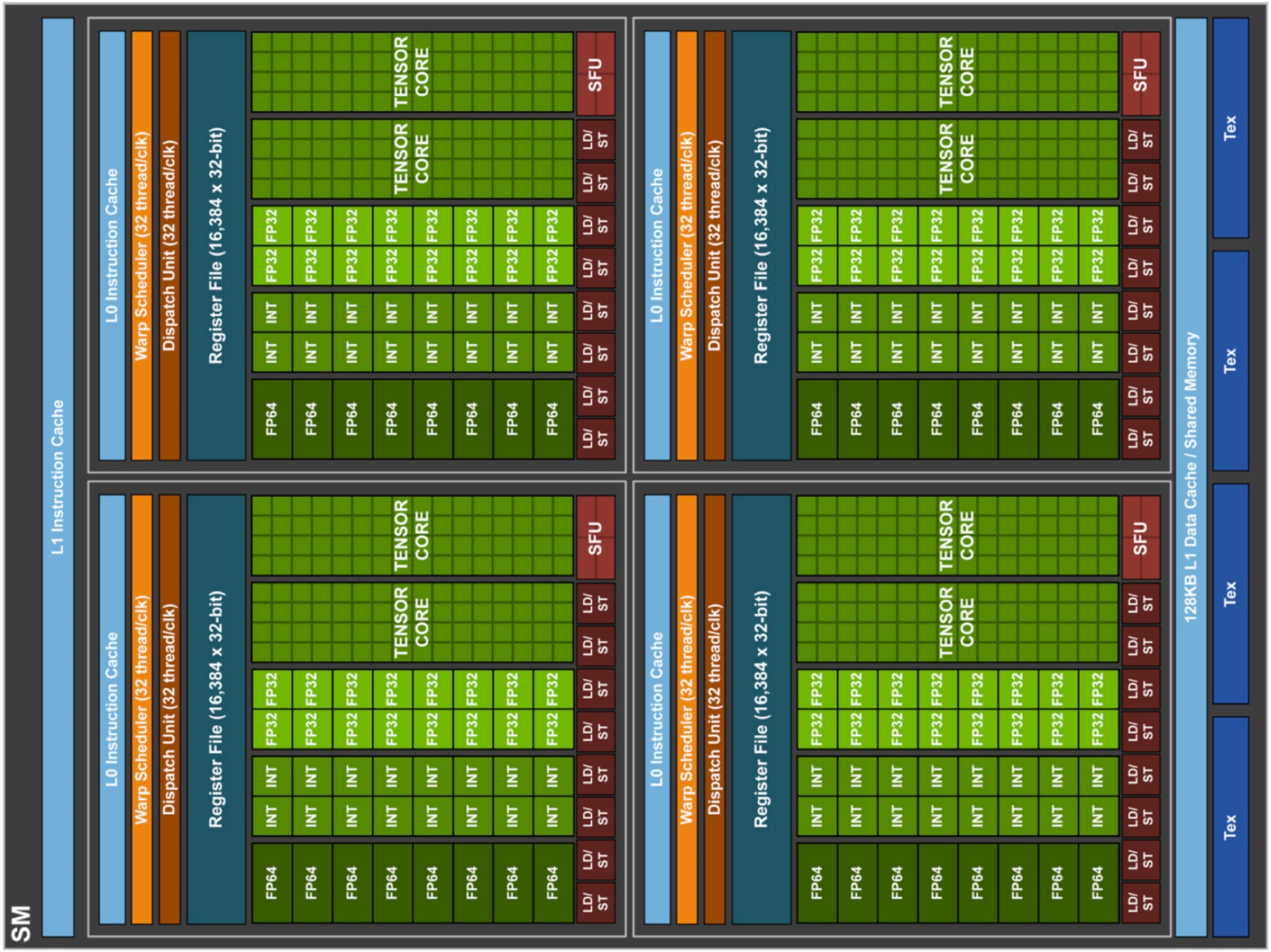
FP32 核数	FP64 核数	寄存器大小	共享存储大小	线程簇调度器
64	32	256 KB	64 KB	2

P100 总体架构

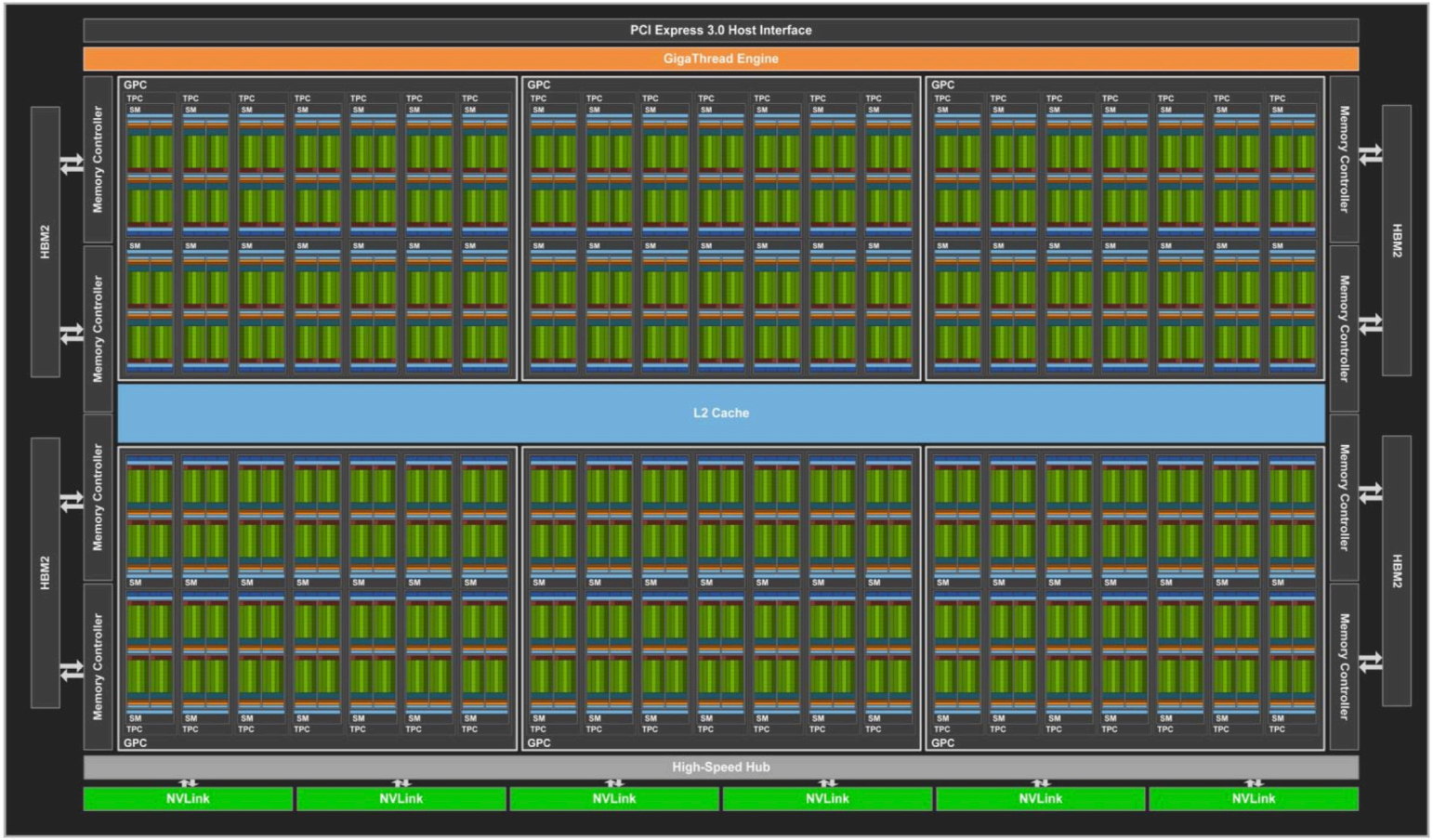


SM 个数	总 FP32 核数	总 FP64 核数	L2 Cache	带宽	内存大小
56	3584	1792	4096 KB	732 GB/s	16 GB HBM2

Tesla V100 的 SM 架构



Tesla V100 总体架构



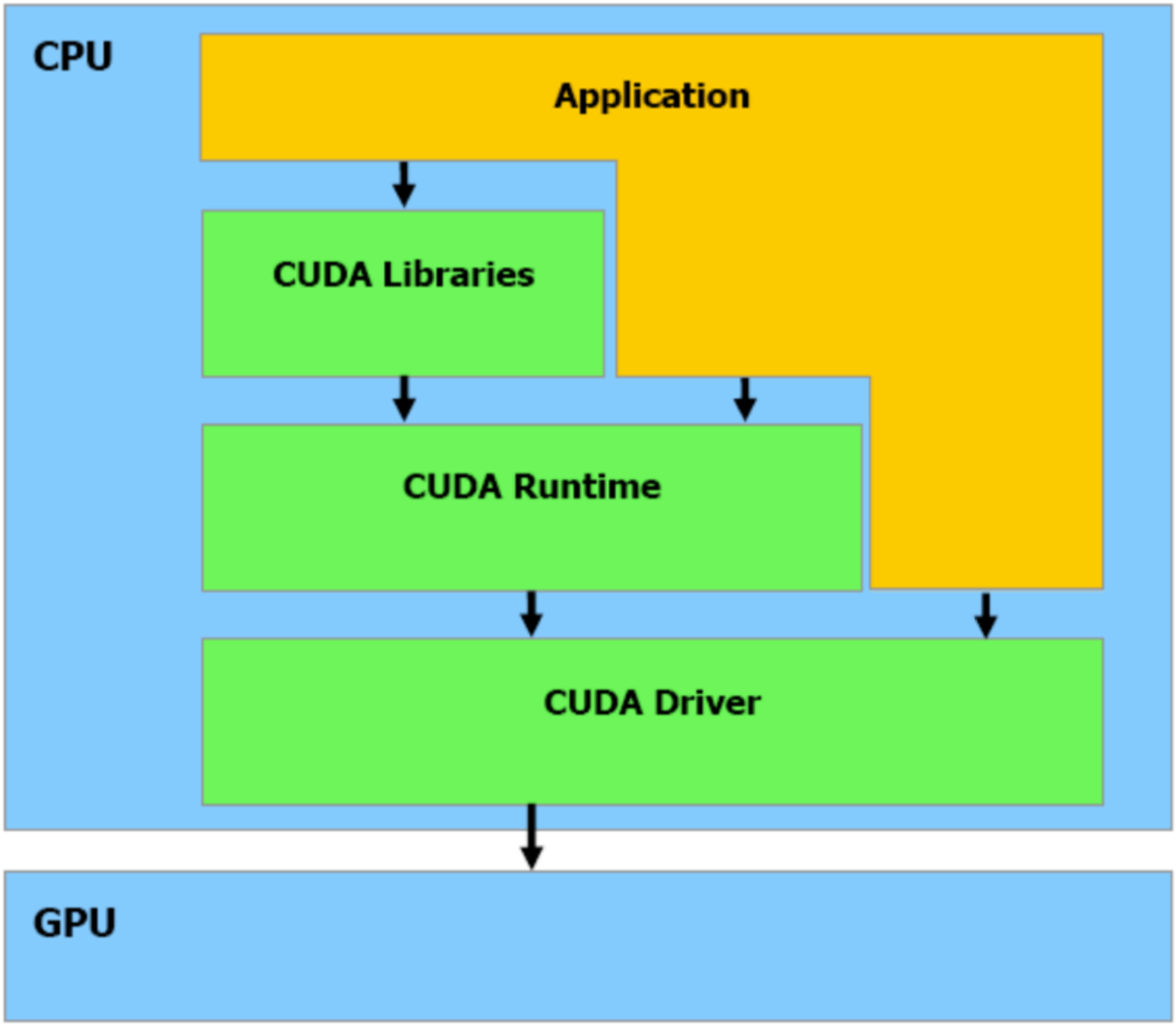
SM 个数	总 FP32 核数	总 FP64 核数	总 Tensor 核数	L2 Cache	带宽	内存大小
80	5120	2560	640	6144 KB	900 GB/s	16 GB HBM2

几代 NVIDIA GPU 硬件参数总结

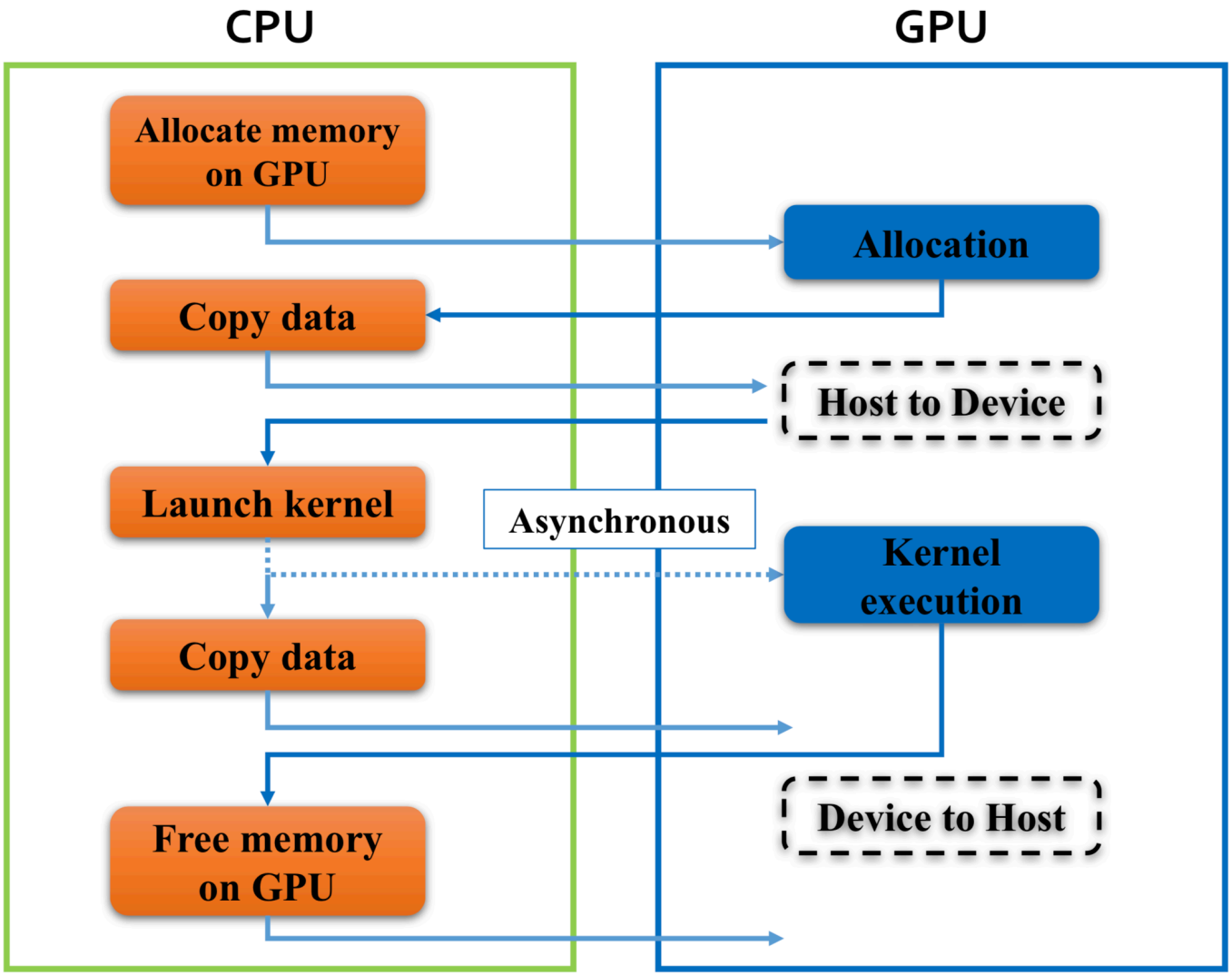
Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	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

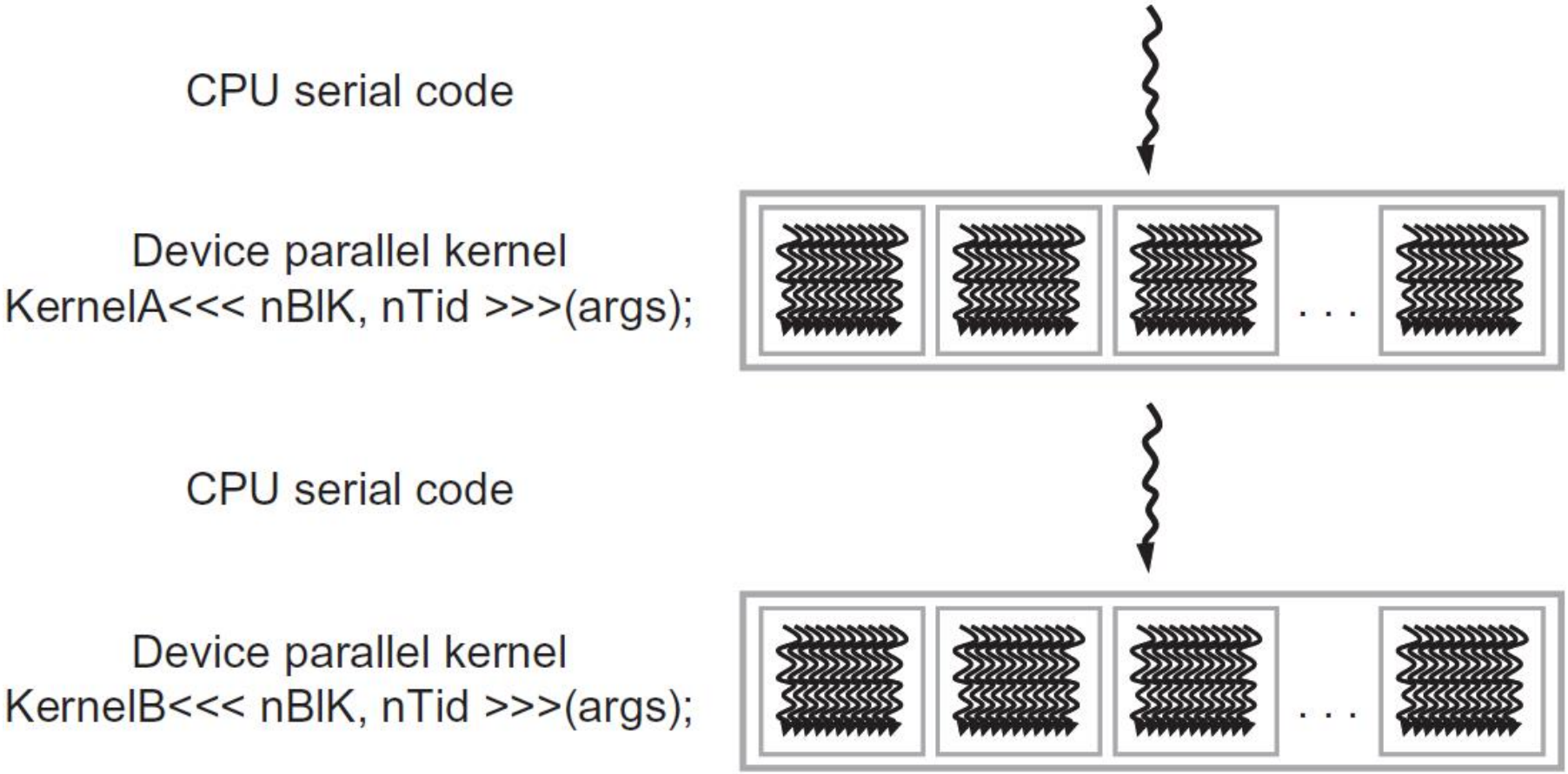

CPU
Application
CUDA Libraries
CUDA Runtime
CUDA Driver
GPU

CUDA 异构并行计算流程一览



CUDA 程序执行模型

基本模型：host (串行或少量) + device (大量并行).



host 代码和 kernel 代码

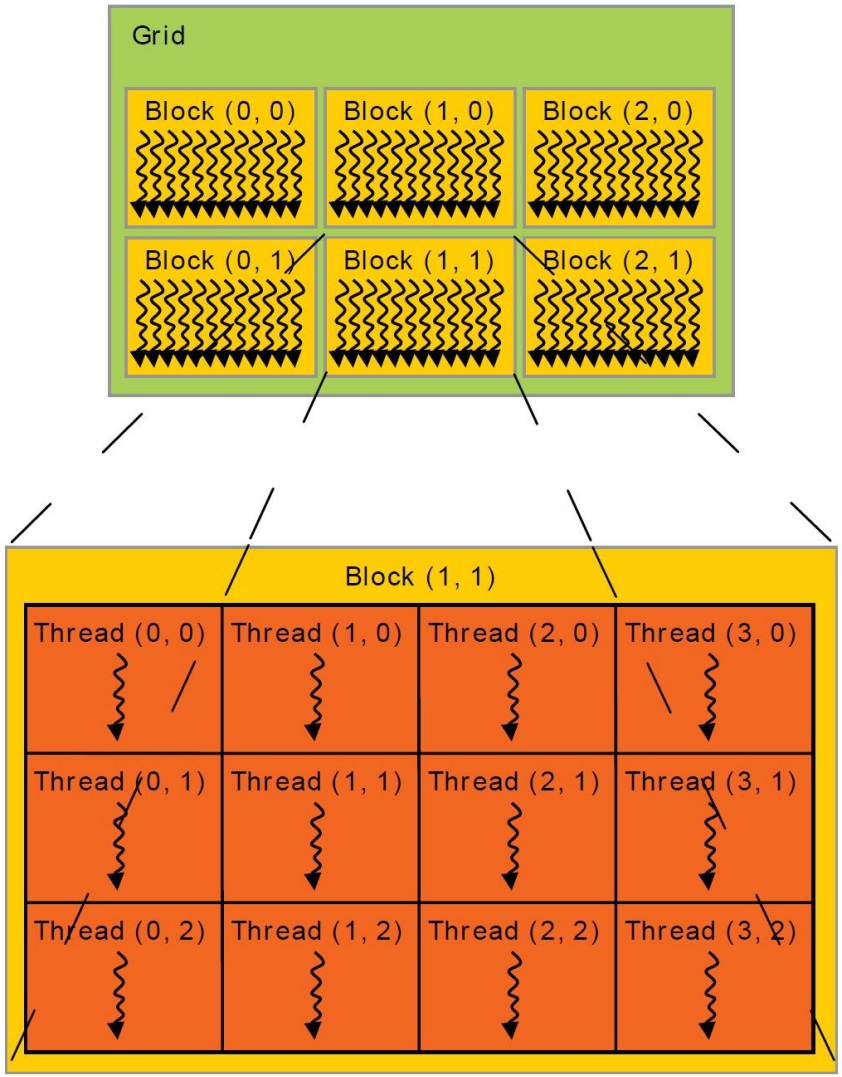
一个典型的 CUDA 程序由两部分组成：

- 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.

内容提纲

hello world!

cuda_hello.cu

```
1  __global__ void mykernel(void) {
2    printf("Hello World from GPU by thread %d!\n", threadIdx.x);
3  }
4  int main(void) {
5    // Use 1 thread
6    mykernel<<<1,1>>>();
7    // Flush the output
8    cudaDeviceSynchronize();
9    // Use 4 threads
10   mykernel<<<1,4>>>();
11   // Flush the output
12   cudaDeviceSynchronize();
13   return 0;
14 }
```

代码分析

```
1 __global__ void mykernel(void) {
2   printf("Hello World from GPU by thread %d!\n", threadIdx.x);
3 }
```

- CUDA C/C++ 关键字`__global__`表示两点：
  - 函数在 host 端被调用；
  - 函数在 device 端运行.
- `threadIdx.x`是线程索引.

```
1  ...
2  mykernel<<<1, 1>>>();
3  ...
4  mykernel<<<1, 4>>>();
5  ...
```

- <<<...>>>表示在 host 调用 device 代码，又称 kernel launch
  - 第一个参数代表线程网格大小，可为整数或 dim3 类型；
  - 第二个参数代表线程块大小，可为整数或 dim3 类型.

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

具体到每个线程

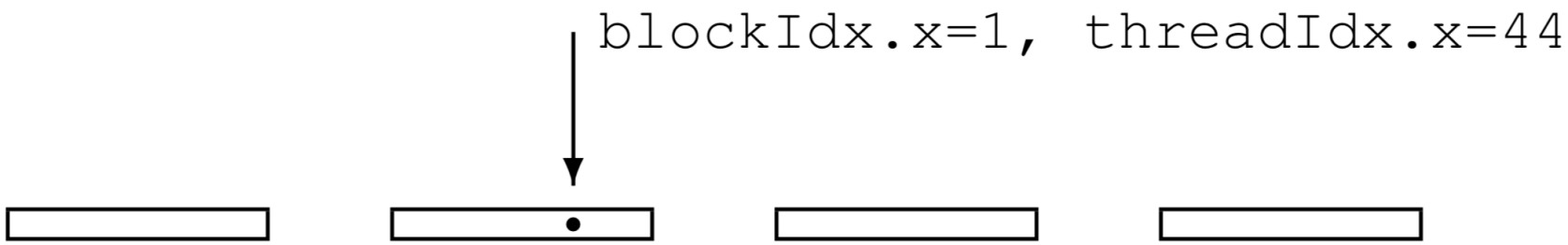
At the lower level, when one instance of the kernel is started on a SM it is executed by a number of threads, each of which knows about:

- 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`



编译与运行

- 编译:

```
$ module load cuda
$ nvcc -o hello cuda_hello.cu
```

- `nvcc -o hello hello_world.cu`在 host 和 device 端分别编译:
  - host 代码 (例如 main) 被 gcc 等 host 端编译器编译;
  - device 代码 (例如 mykernel) 被 NVIDIA 编译器编译.
- 运行结果:

```
$ ./hello
Hello World from GPU by thread 0!
Hello World from GPU by thread 0!
Hello World from GPU by thread 1!
Hello World from GPU by thread 2!
Hello World from GPU by thread 3!
```

内容提纲

CPU 上的向量加法

```
1  // Compute vector sum h_C = h_A + h_B
2  void cpu_vec_add(float *h_A, float *h_B, float *h_C, int n) {
3    for (int i = 0; i < n; i++)
4      h_C[i] = h_A[i] + h_B[i];
5  }
6
7  int main() {
8  // Memory allocation for h_A, h_B, and h_C
9    ...
10   cpu_vec_add(h_A, h_B, h_C, N);
11 }
```

利用 GPU 实现向量加法的三个步骤

```
1  #include <cuda.h>
2  ...
3  void vec_add(float *h_A, float *h_B, float *h_C, int n) {
4    int size = n* sizeof(float);
5    float *d_A, *d_B, *d_C;
6    ...
7    // 1. Allocate device memory for A, B, and C
8    // Transfer A and B to device memory
9
10   // 2. Kernel launch code to have the device
11   // to perform the actual vector addition
12
13   // 3. Transfer C from device to host
14   // Free device memory for A, B, C
15 }
```

步骤一和步骤三的代码

```
1  void vec_add(float* h_A, float* h_B, float* h_C, int n) {
2    ...
3    // 1. Allocate device memory for A, B, and C
4    cudaMalloc((void **) &d_A, size);
5    cudaMalloc((void **) &d_B, size);
6    cudaMalloc((void **) &d_C, size);
7    // Transfer A and B to device memory
8    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
9    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
10
11   // 2. Kernel invocation code to be shown later
12   ...
13
14   // 3. Transfer C from device to host
15   cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
16   // Free device memory for A, B, C
17   cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
18 }
```

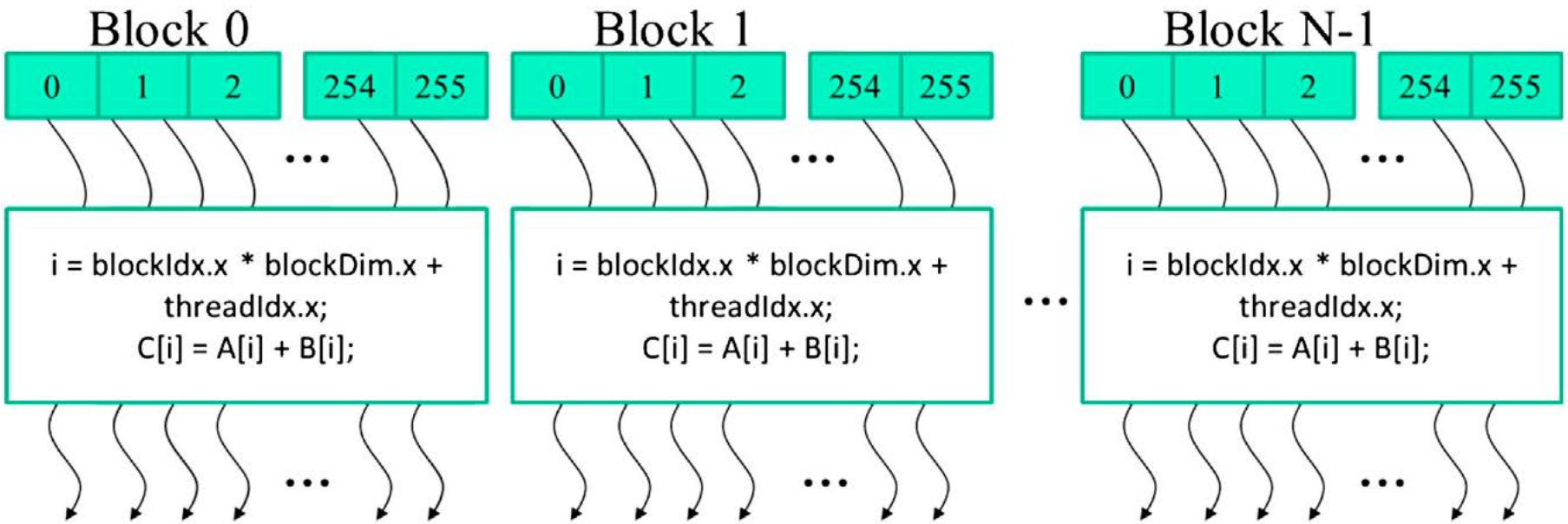
内存分配

- `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.

步骤二的线程组织



- Grid 中的 block 按照一维方式组织, 每个 block 中的 thread 也是按照一维方式组织;
- 可以看出向量加法中，线程组织与数据的处理相对应.

步骤二：device 端的 kernel 代码

```
1 // Compute vector sum C = A + B
2 // Each thread performs one pair-wise addition
3 __global__
4 void vec_add_kernel(float* d_A, float* d_B, float* d_C, int n){
5   int i = blockDim.x * blockIdx.x + threadIdx.x;
6   if (i < n) d_C[i] = d_A[i] + d_B[i];
7 }
```

- 思考：为什么加上 `if (i < n)` 判断?

CUDA C 函数声明关键字

CUDA C 关键字	执行位置	调用位置
`__device__ type DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ type HostFunc()`	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.

步骤二：host 上的 kernel launch

```
1  __host__
2  void vec_add(float* h_A, float* h_B, float* h_C, int n) {
3      ...
4    // d_A, d_B, d_C allocations and copies omitted
5    // Run ceil(n/256.0) blocks of 256 threads each
6    dim3 grid_dim(ceil(n/256.0),1,1);
7    dim3 block_dim(256,1,1);
8    vec_add_kernel<<<grid_dim, block_dim>>>(d_A, d_B, d_C, n);
9    ...
10 }
```

- 注意：此处 `__host__` 关键字可以省略；
- 思考：为什么写成 `ceil(n/256.0)`？

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).

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);
```

完整 host 端和 devide 端代码

host 端代码

```
1  void vec_add(float *h_A, float *h_B, float *h_C, int n) {
2    int size = n * sizeof(float);
3    float *d_A, *d_B, *d_C;
4    cudaEvent_t start, stop;
5    float elapsed_time = 0.0;
6    cudaEventCreate(&start);
7    cudaEventCreate(&stop);
8
9    //1. Allocate device memory and transfer data to device
10   cudaMalloc((void **) &d_A, size);
11   cudaMalloc((void **) &d_B, size);
12   cudaMalloc((void **) &d_C, size);
13   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
14   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
15
16   // 2. Configure execution and launch kernel
17   dim3 grid_dim(ceil(n/256.0), 1, 1);
18   dim3 block_dim(256, 1, 1);
```

```
19    cudaEventRecord(start, 0);
20    vec_add_kernel<<<grid_dim, block_dim>>>(d_A, d_B, d_C, n);
21    cudaEventRecord(stop, 0);
22    cudaEventSynchronize(stop);
23    cudaEventElapsedTime(&elapsed_time, start, stop);
24
25    // 3. Transfer result back to host and free device memory
26    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
27    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
28  }
```

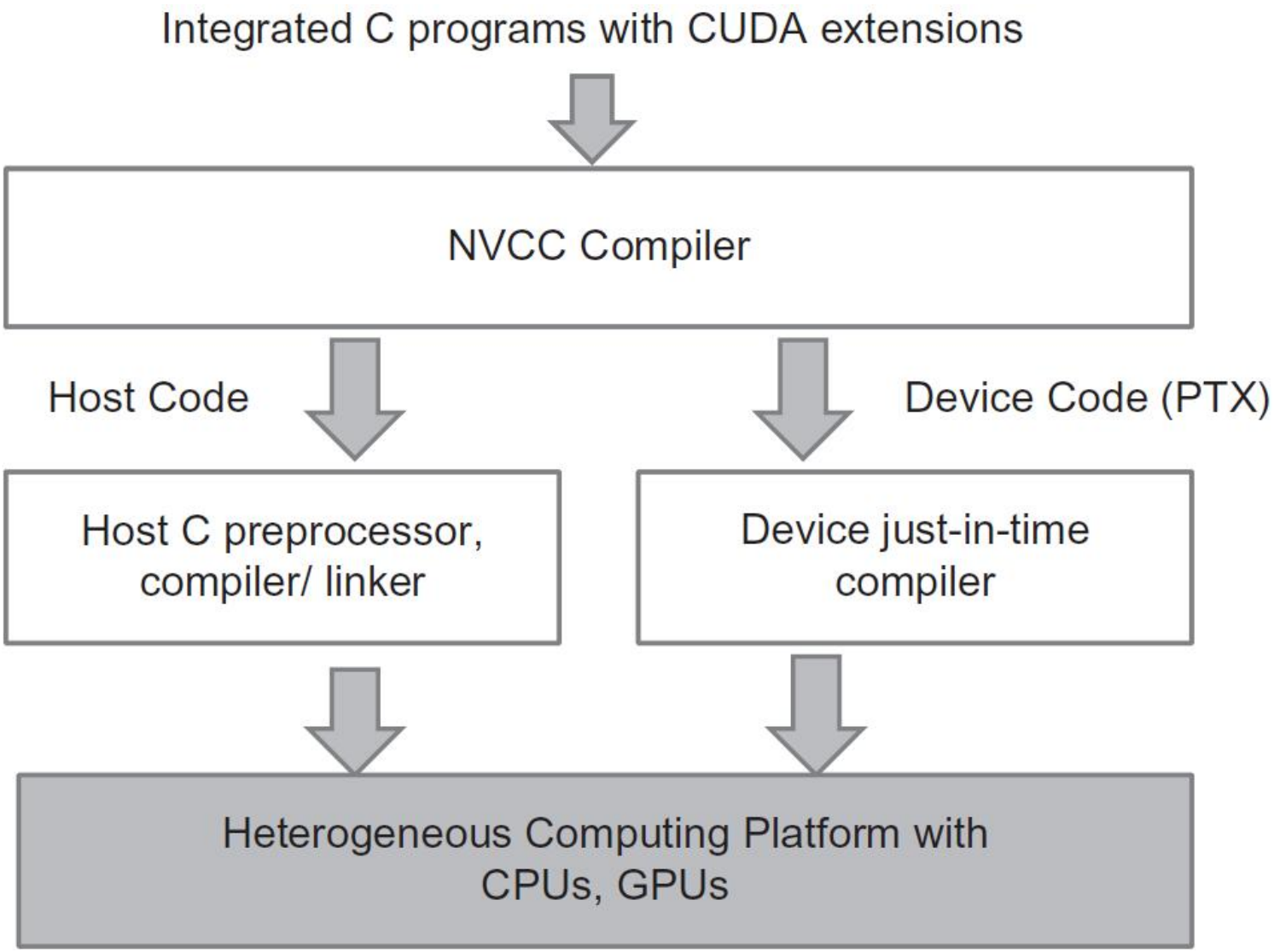
device 端代码

```
1  // Compute vector sum C = A + B
2  // Each thread performs one pair-wise addition
3  __global__
4  void vec_add_kernel(float* d_A, float* d_B, float* d_C, int n){
5    int i = blockDim.x * blockIdx.x + threadIdx.x;
6    if (i < n) d_C[i] = d_A[i] + d_B[i];
7  }
```

main 函数中的调用代码

```
1  int main(void) {
2
3    int N = 1024000;
4    ...
5    // Memory allocation for vectors A, B and C
6    float *h_A = rand_vec(N);
7    float *h_A = rand_vec(N);
8    float *h_C = raw_vec(N);
9    ...
10   vec_add(h_A, h_B, h_C, N);
11   ...
12 }
```

CUDA 程序编译


Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C preprocessor, compiler/ linker
Device just-in-time compiler
Heterogeneous Computing Platform with CPUs, GPUs

- 部分运行结果：

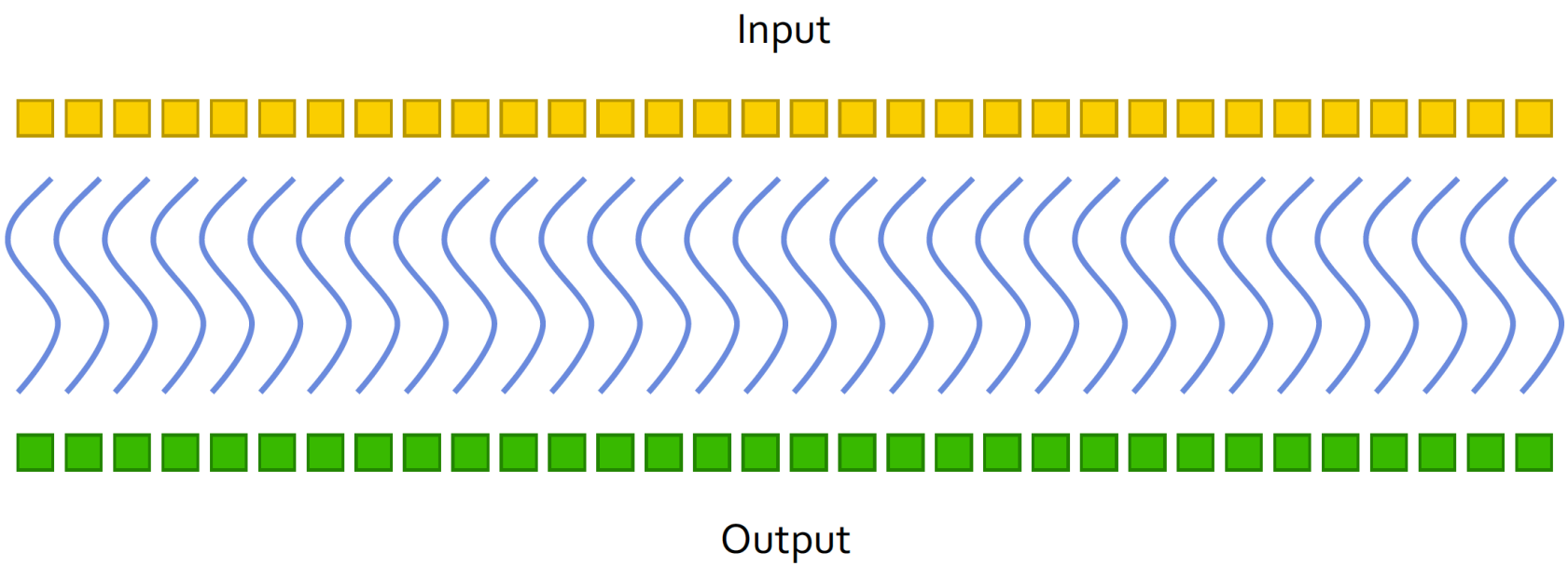
```
Vector length: 102400.
CPU: 0.00183 sec
  grid  dim:  400, 1, 1.
  block dim: 256, 1, 1.
  kernel time: 0.00009 sec.
GPU: 0.37654 sec
All values correct.
...
Vector length: 1024000.
CPU: 0.01484 sec
  grid  dim:  4000, 1, 1.
  block dim: 256, 1, 1.
  kernel time: 0.00011 sec.
GPU: 0.35207 sec
All values correct.
```

- 思考 1：为什么 GPU 总时间比 CPU 还慢?
- 思考 2：为什么规模增加 10 倍，但是 kernel 计算时间增加不多?

内容提纲

GPU 的执行模式

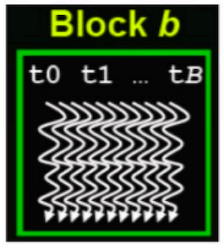
- **Calculations on a GPU always follow the same model.**
- **This is very constrained.**
- **GPUs run kernels that are designed to execute calculations in the following way.**



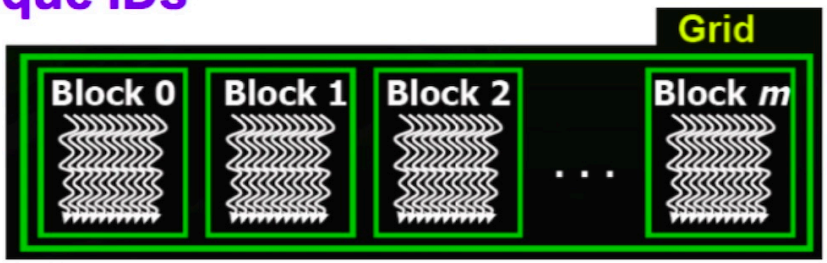
线程的层次结构

- Parallel kernels composed of many threads
  - —all threads execute same sequential program
  - —use parallel threads rather than sequential loops

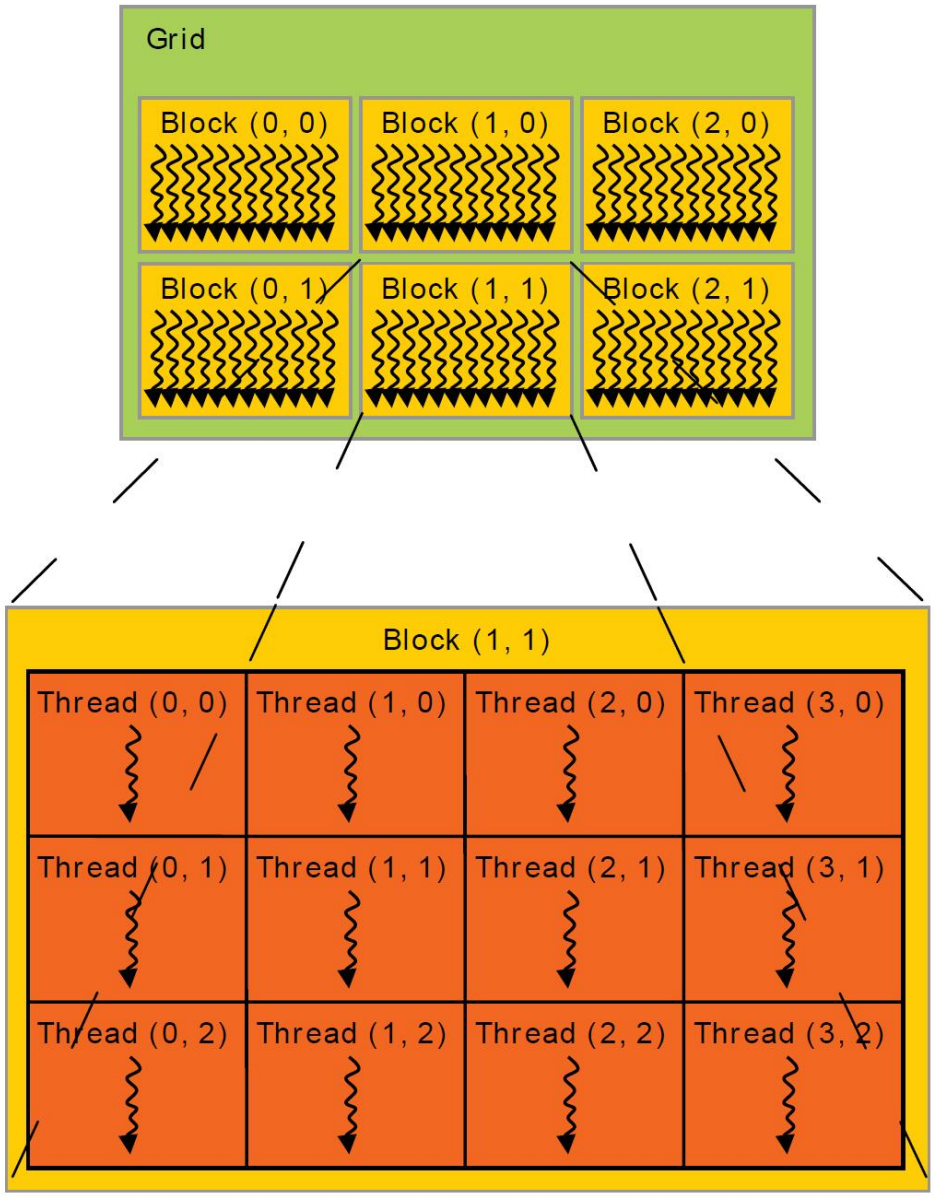
- Threads are grouped into thread blocks
  - —threads in block can sync and share memory



- 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



线程网格组织层次举例：2D

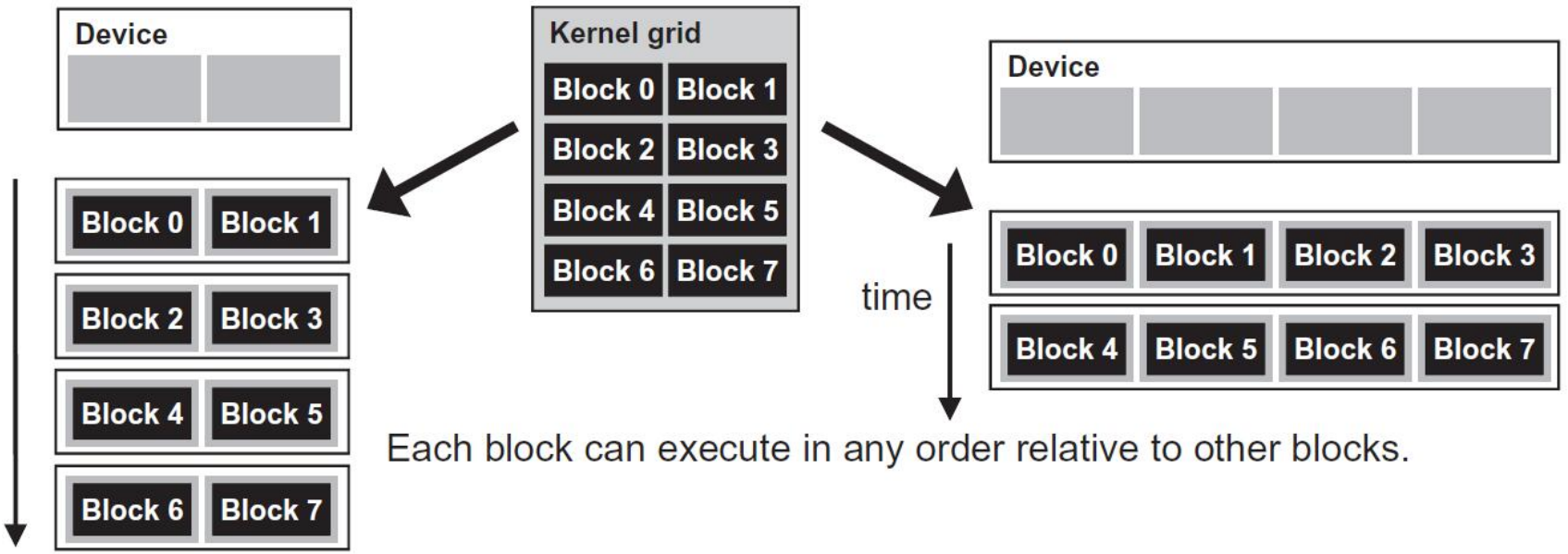



```
dim3 mygrid(3,2,1);
dim3 myblock(4,3,1);
mykernel<<<mygrid, myblock>>>;
```

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

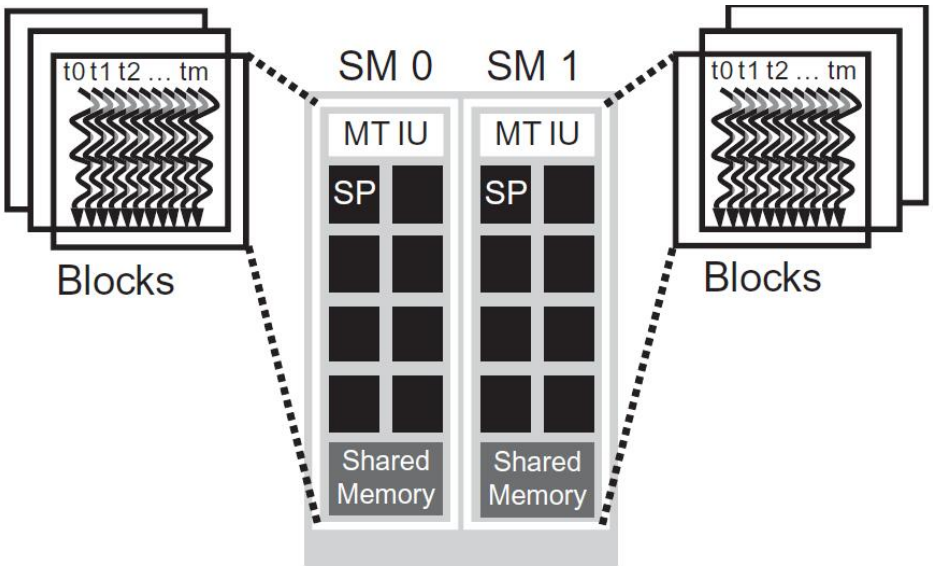
线程块间透明扩展

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.

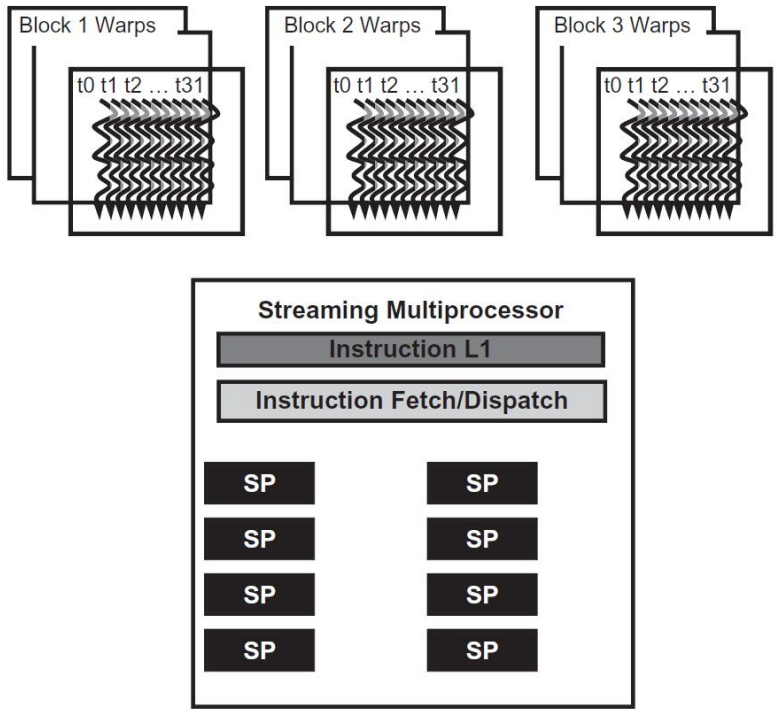


图: 每个 warp 中线程 threadIdx 值连续：0 到 31 属于第一个 warp，32 到 63 属于第二个 warp，以此类推. 图中例子总共有三个线程块分配到当前 SM.

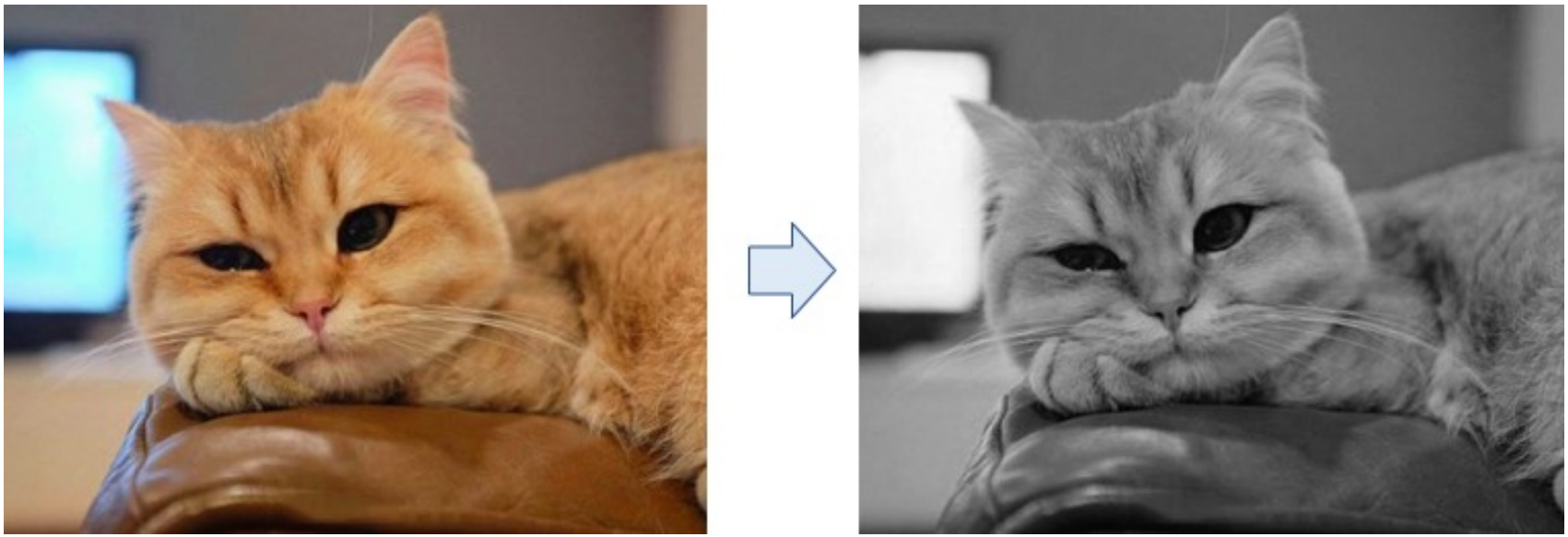
内容提纲

图片灰白化处理

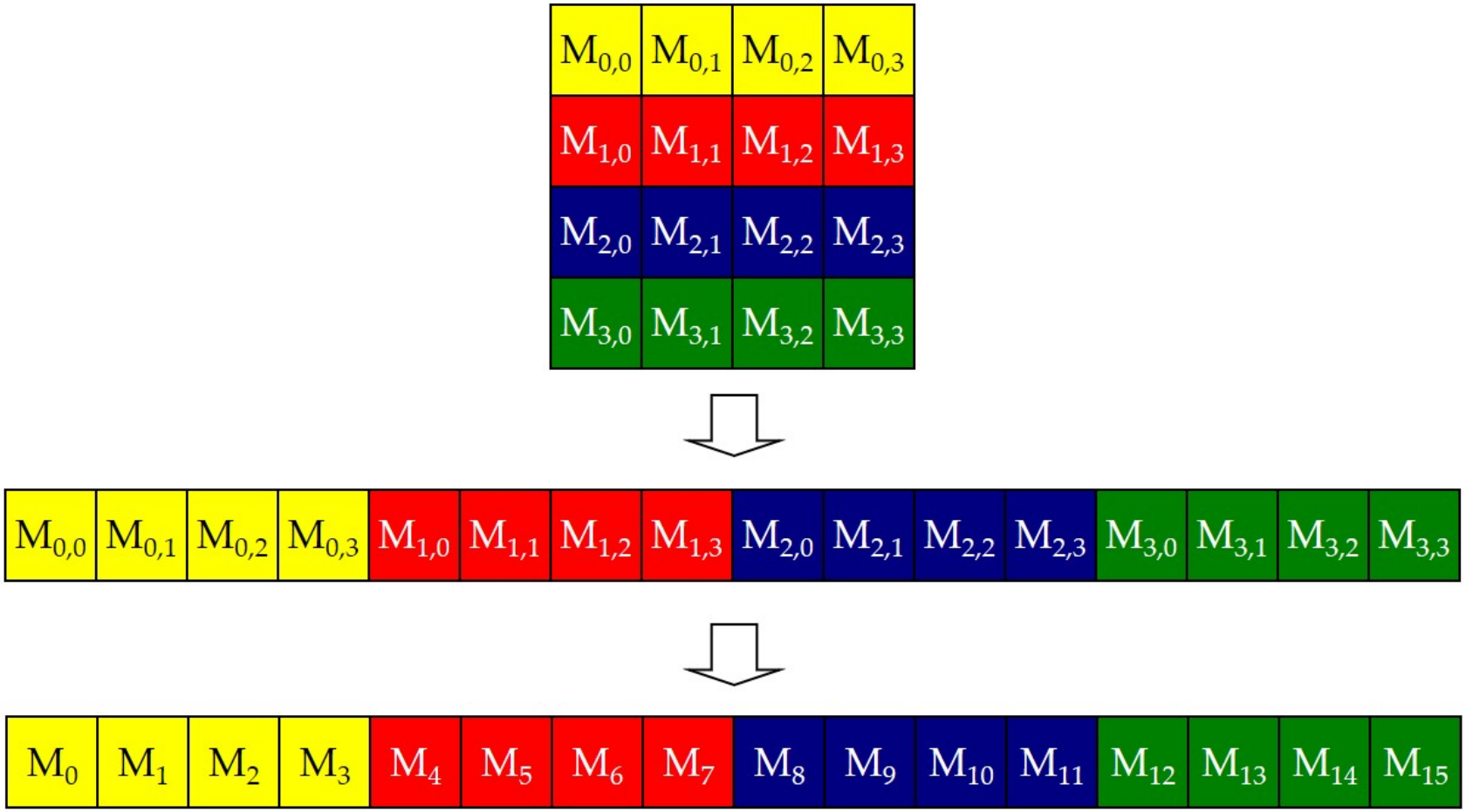
- 图片灰白化处理的计算公式：

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

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


M0,0 M0,1 M0,2 M0,3
M1,0 M1,1 M1,2 M1,3
M2,0 M2,1 M2,2 M2,3
M3,0 M3,1 M3,2 M3,3
M0,0 M0,1 M0,2 M0,3 M1,0 M1,1 M1,2 M1,3 M2,0 M2,1 M2,2 M2,3 M3,0 M3,1 M3,2 M3,3
M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11 M12 M13 M14 M15

CPU 程序

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

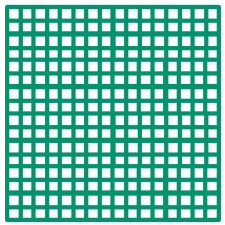
GPU 线程设置

- 设线程块大小为 16×16，执行配置：

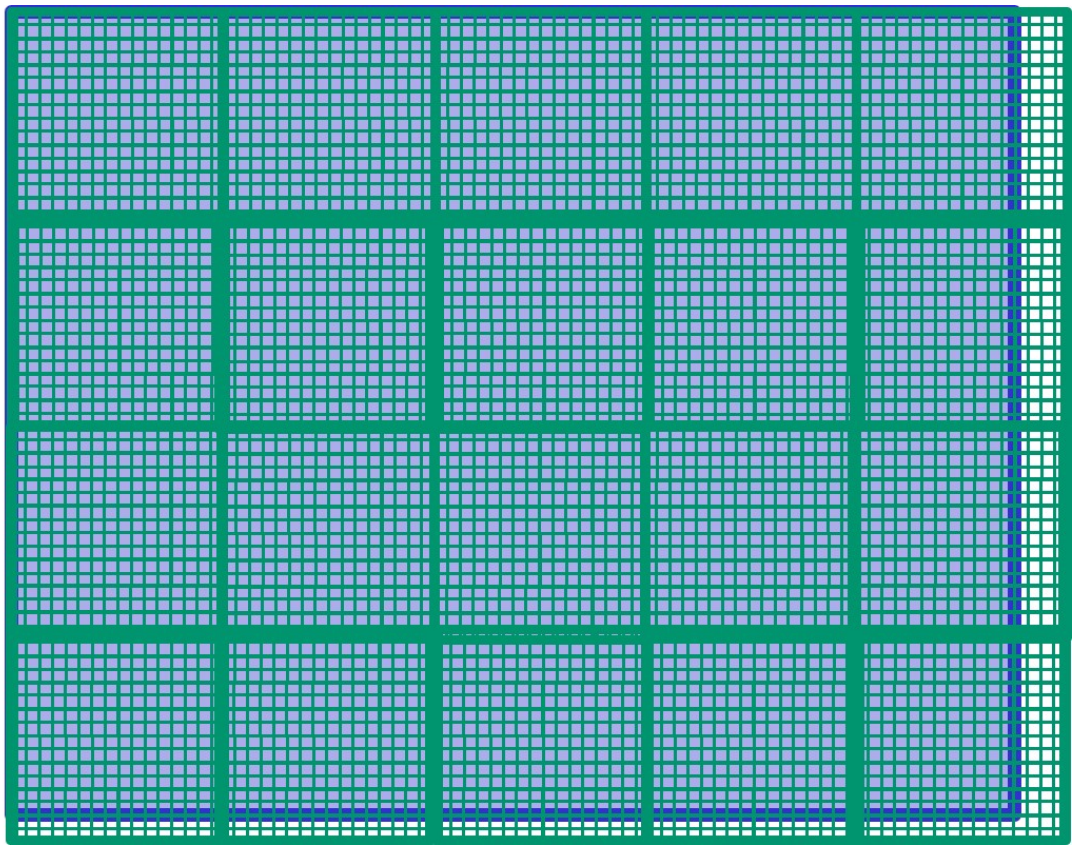
```
1  ...
2  // Allocate and copy data
3  cudaMalloc((void **)&d_out, size_out);
4  cudaMalloc((void **)&d_in, size_in);
5  cudaMemcpy(d_in, h_in, size_in, cudaMemcpyHostToDevice);
6  // Process the m × n input image
7  dim3 grid_dim(ceil(m/16.0), ceil(n/16.0), 1);
8  dim3 block_dim(16, 16, 1);
9  gpu_pic2grey_kernel<<<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.

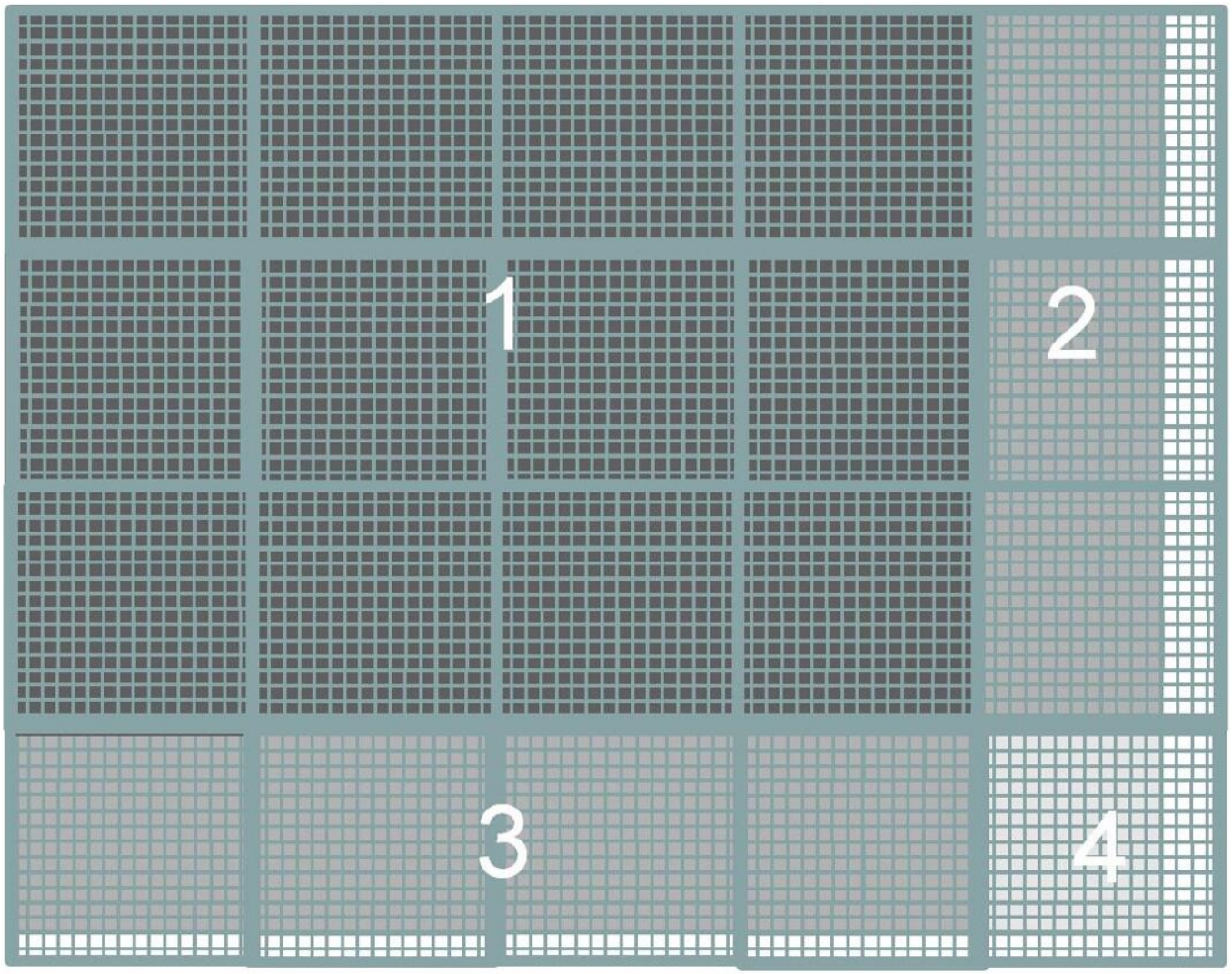


16×16



四种不同的任务负载

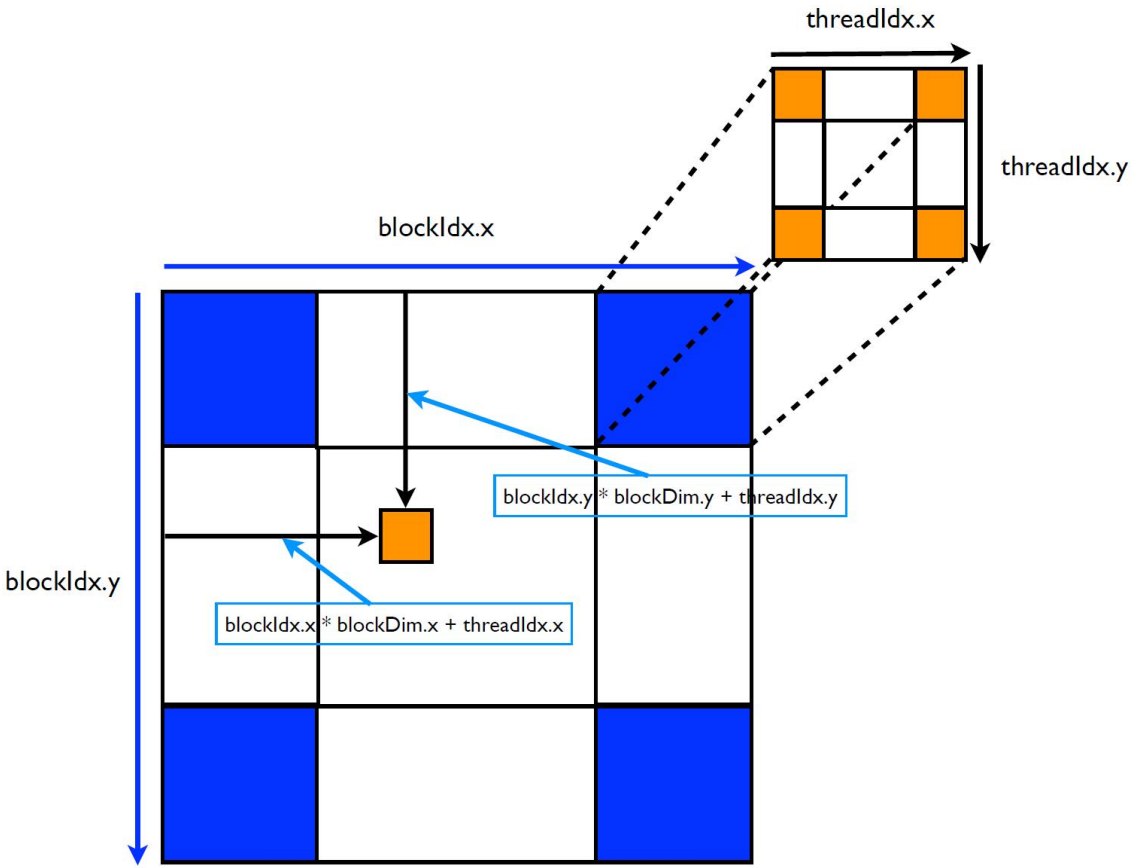
根据图片灰白化处理的计算过程，任务负载有四种不同情形：



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：

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



GPU 端的 kernel 程序

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

内容提纲

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 × BLOCK_WIDTH 二维方式组织；
- 网格按照 (width/BLOCK_WIDTH) × (width/BLOCK_WIDTH) 二维方式组织；.

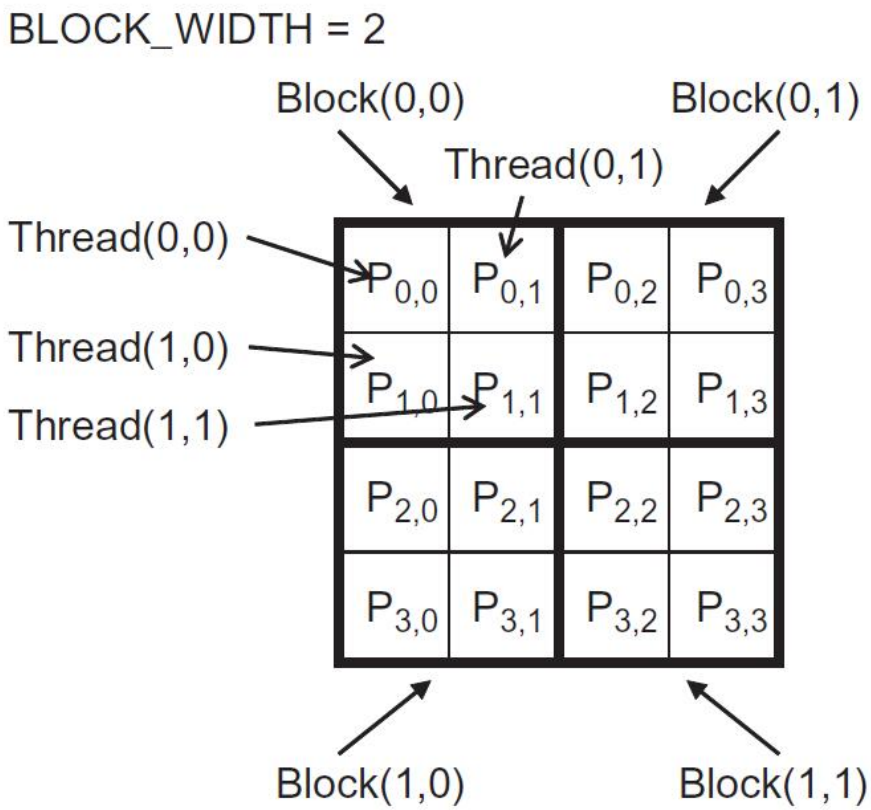


图: 矩阵宽为 4，线程块宽为 2.

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;
```

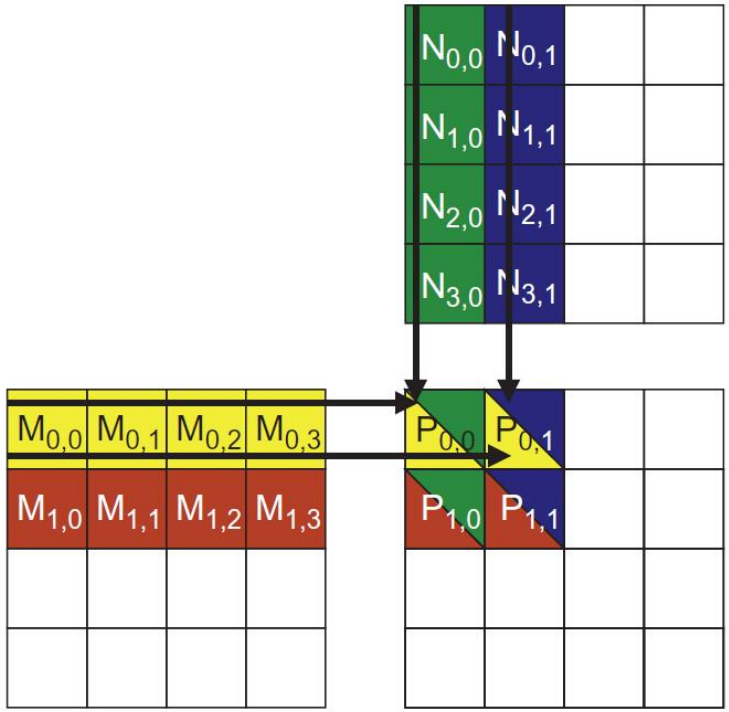
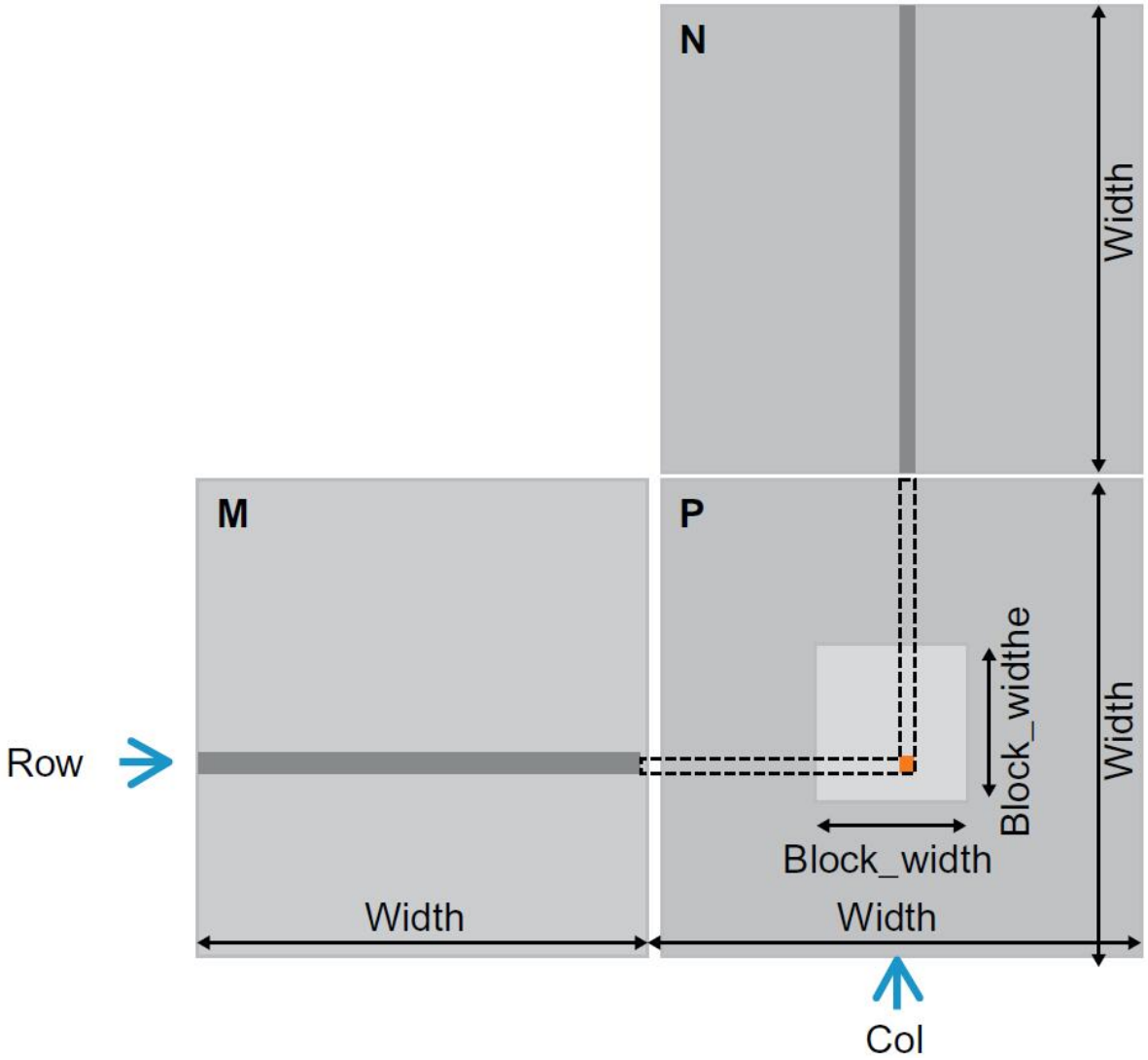


图: 矩阵宽为 4、线程块宽为 2 时，线程块 (0, 0) 的计算过程.

分块方式


N
Width
M
P
Row
Block_widthe
Block_width
Width
Width
Width
Col

GPU 矩阵乘 kernel 程序

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

程序结果分析

- 测试结果:

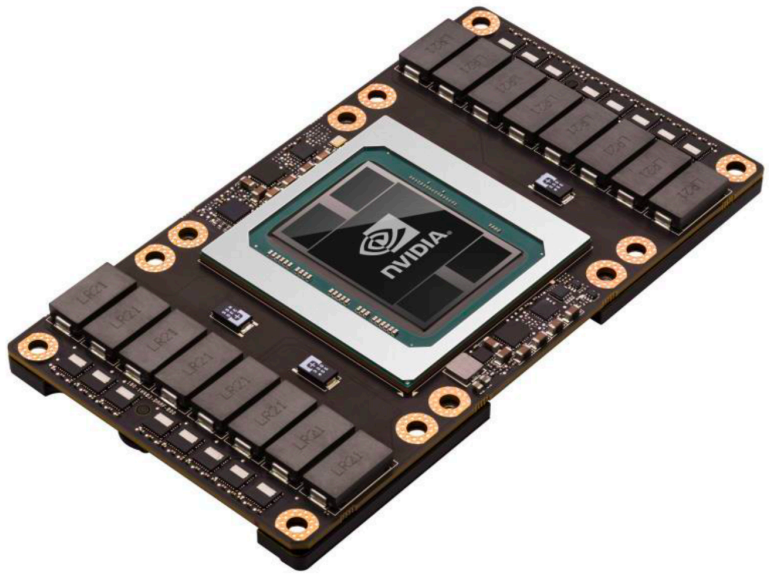
```
Matrix width: 512.
CPU: 0.89709 sec
  grid  dim:  32, 32, 1.
  block dim: 16, 16, 1.
  kernel time: 0.00056 sec
GPU: 0.15642 sec
GPU all values correct

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
```

内容提纲

P100 外观

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

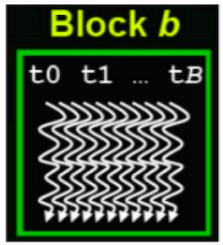
- 片上 (on-chip)：SM、缓存、寄存器等；
- 片外 (off-chip)：设备内存 (device memory) 等.



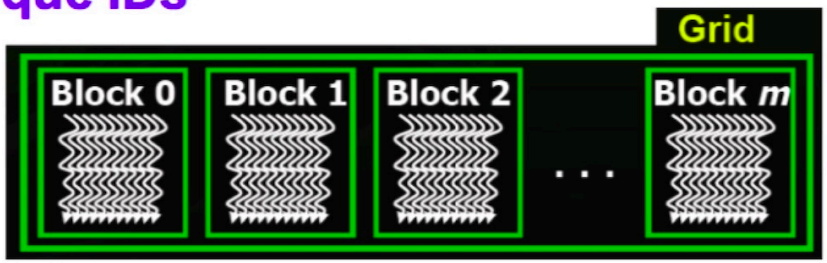
编程视角下的线程层次

- Parallel kernels composed of many threads
  - —all threads execute same sequential program
  - —use parallel threads rather than sequential loops

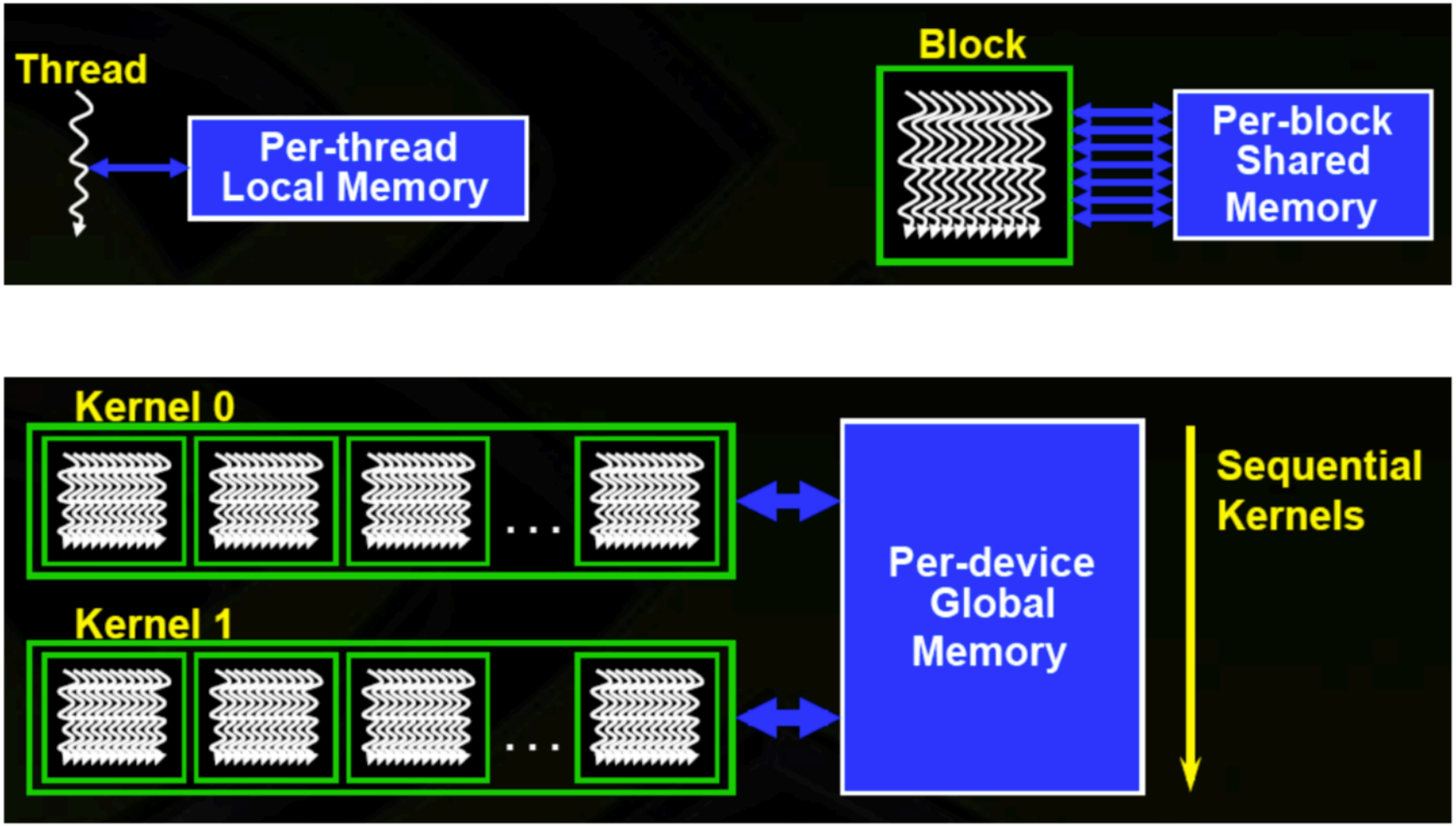
- Threads are grouped into thread blocks
  - —threads in block can sync and share memory



- 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


Thread
Per-thread
Local Memory
Block
Per-block
Shared
Memory
Kernel 0
Kernel 1
Per-device
Global
Memory
Sequential
Kernels

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/SM、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).

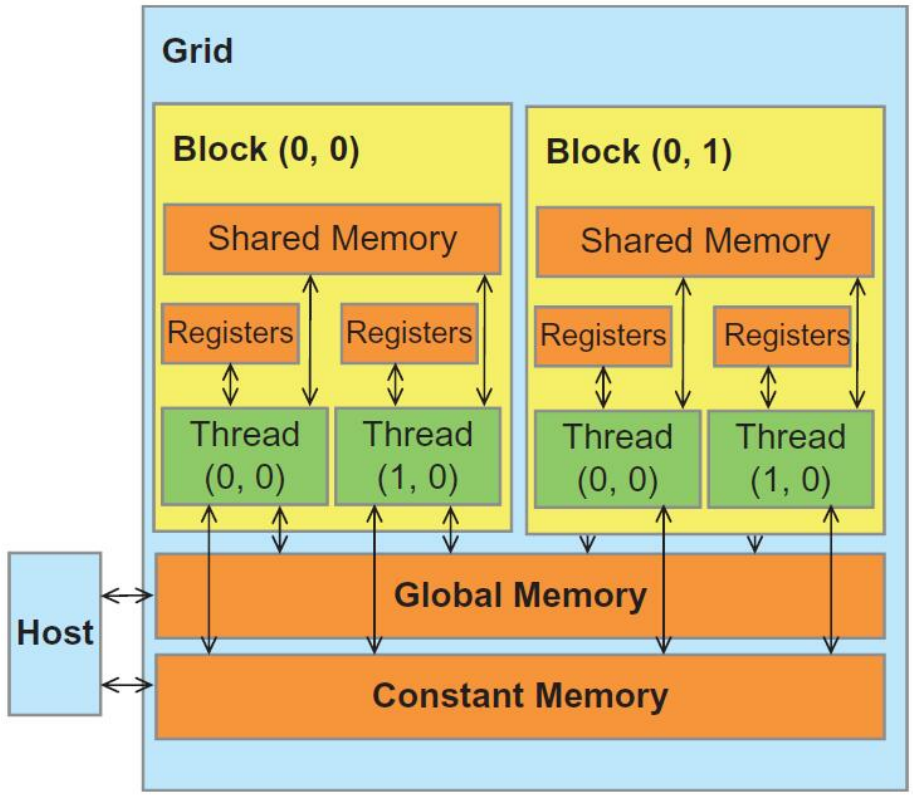
不同硬件层次的存储特性

Device 端代码：

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



内容提纲

GPU 上的矩阵乘

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

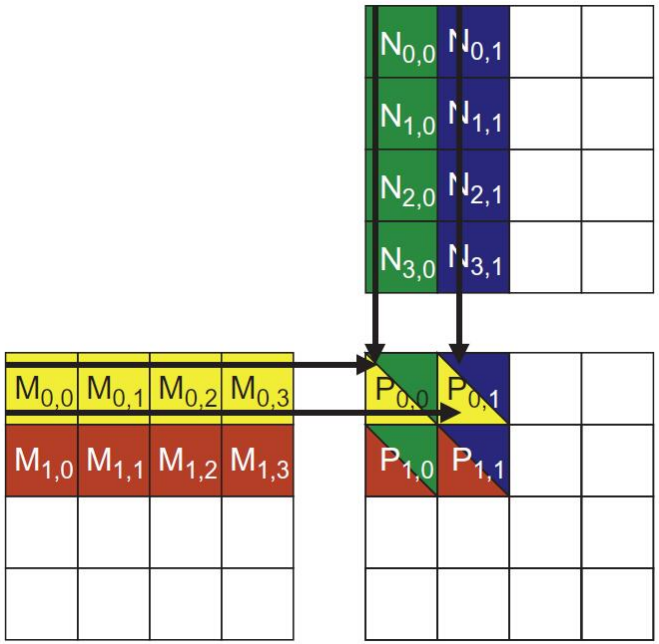
$$P_{i,j} = \sum_{k=0}^{width-1} M_{i,k} N_{k,j}.$$

```
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

$thread_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

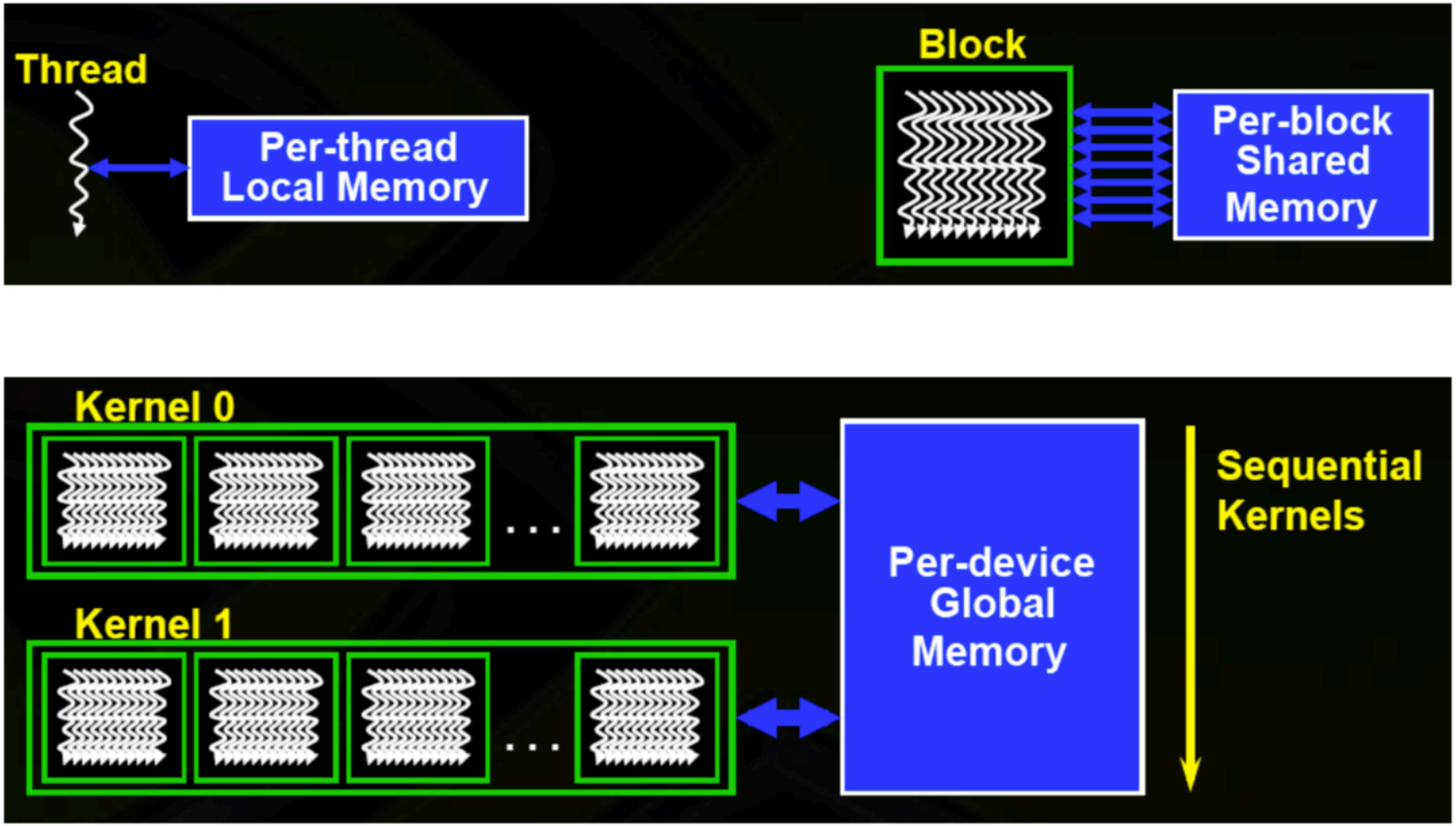


思考：

(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


Thread
Per-thread
Local Memory
Block
Per-block
Shared
Memory
Kernel 0
Kernel 1
Per-device
Global
Memory
Sequential
Kernels

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 `TILE_WIDTH` = `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.

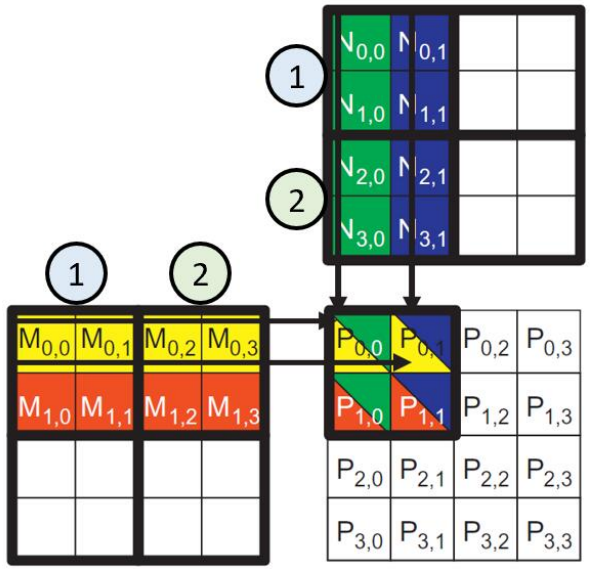


图: 例子中线程块 $Block(0,0)$ 的计算，分两个阶段，将两个阶段结果累加就是最终计算结果.

矩阵乘的分阶段计算

	Phase 1			Phase 2		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$+= $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

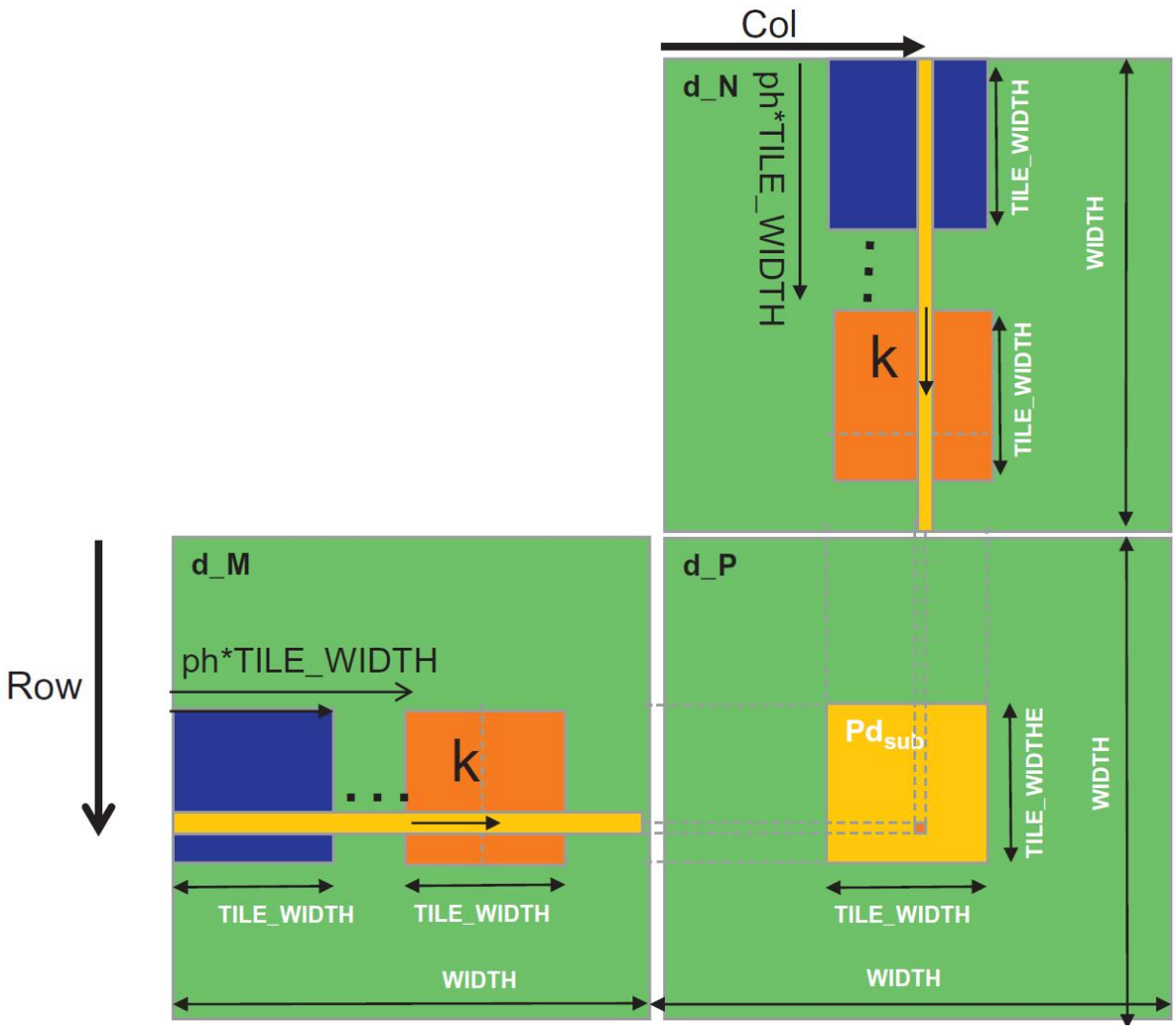
time →

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？

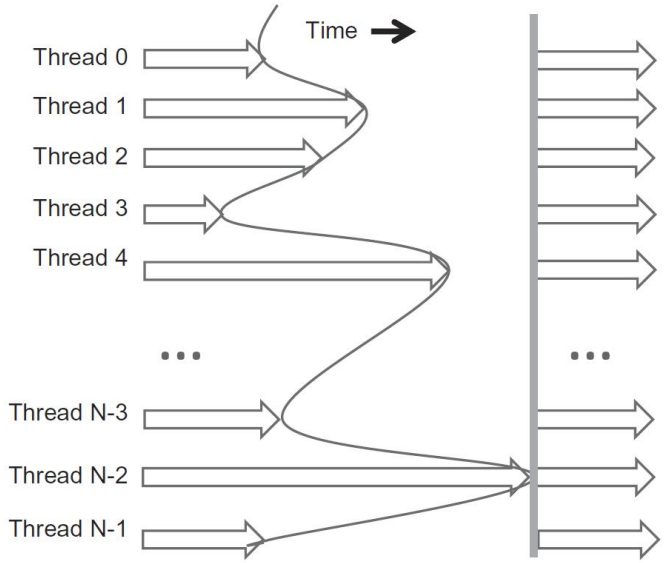
矩阵乘的计算过程

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？

内容提纲

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

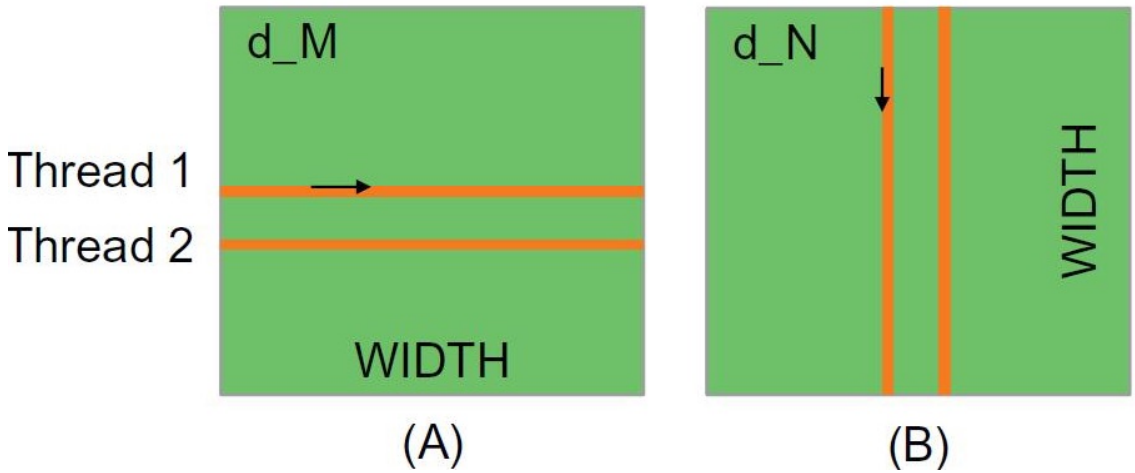
- 可采用 nvprof 命令实现性能统计

```
$ 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.

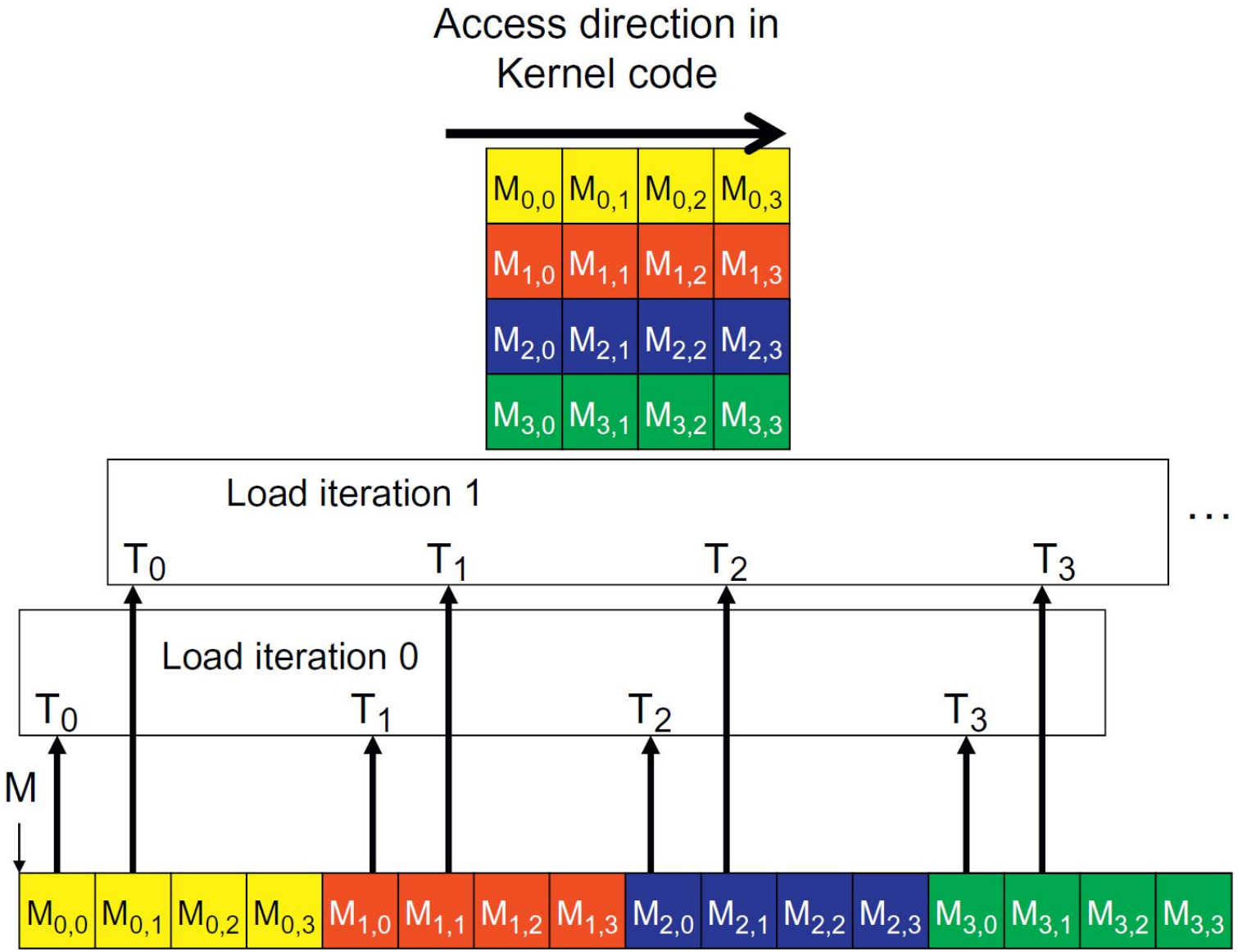
全局内存的合并访问

- 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).

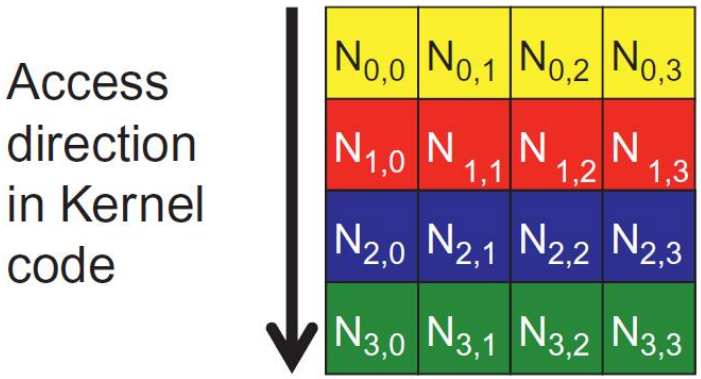


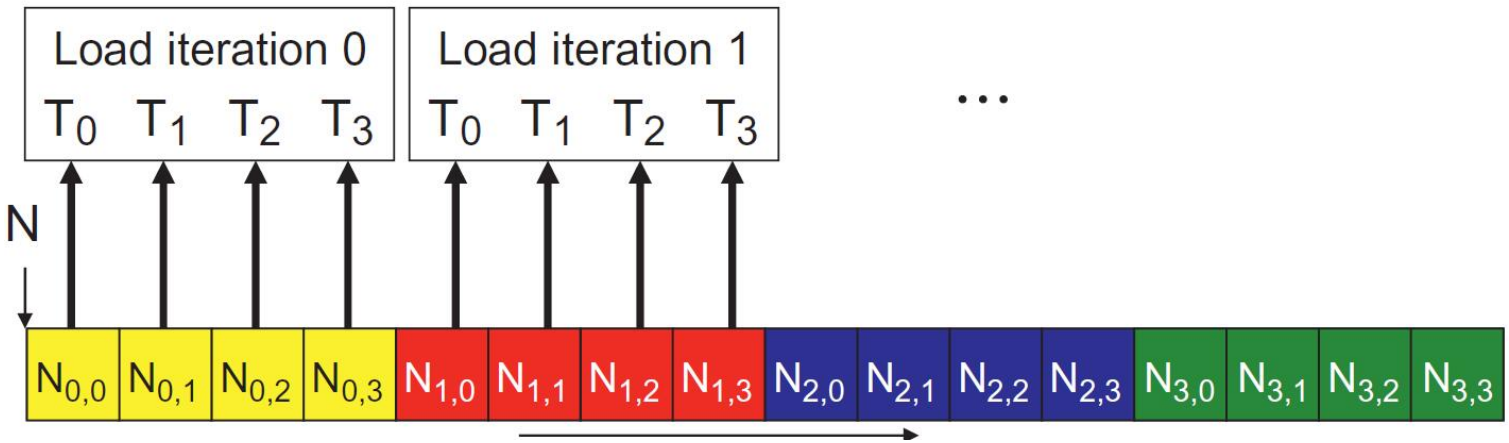
思考：矩阵乘 ex4 算例中，矩阵 M 和 N 的访问是合并访问吗?

矩阵 M 的访存意图：


Access direction in
Kernel code
M0,0 M0,1 M0,2 M0,3
M1,0 M1,1 M1,2 M1,3
M2,0 M2,1 M2,2 M2,3
M3,0 M3,1 M3,2 M3,3
Load iteration 1
T0 T1 T2 T3
...
Load iteration 0
T0 T1 T2 T3
M
M0,0 M0,1 M0,2 M0,3 M1,0 M1,1 M1,2 M1,3 M2,0 M2,1 M2,2 M2,3 M3,0 M3,1 M3,2 M3,3

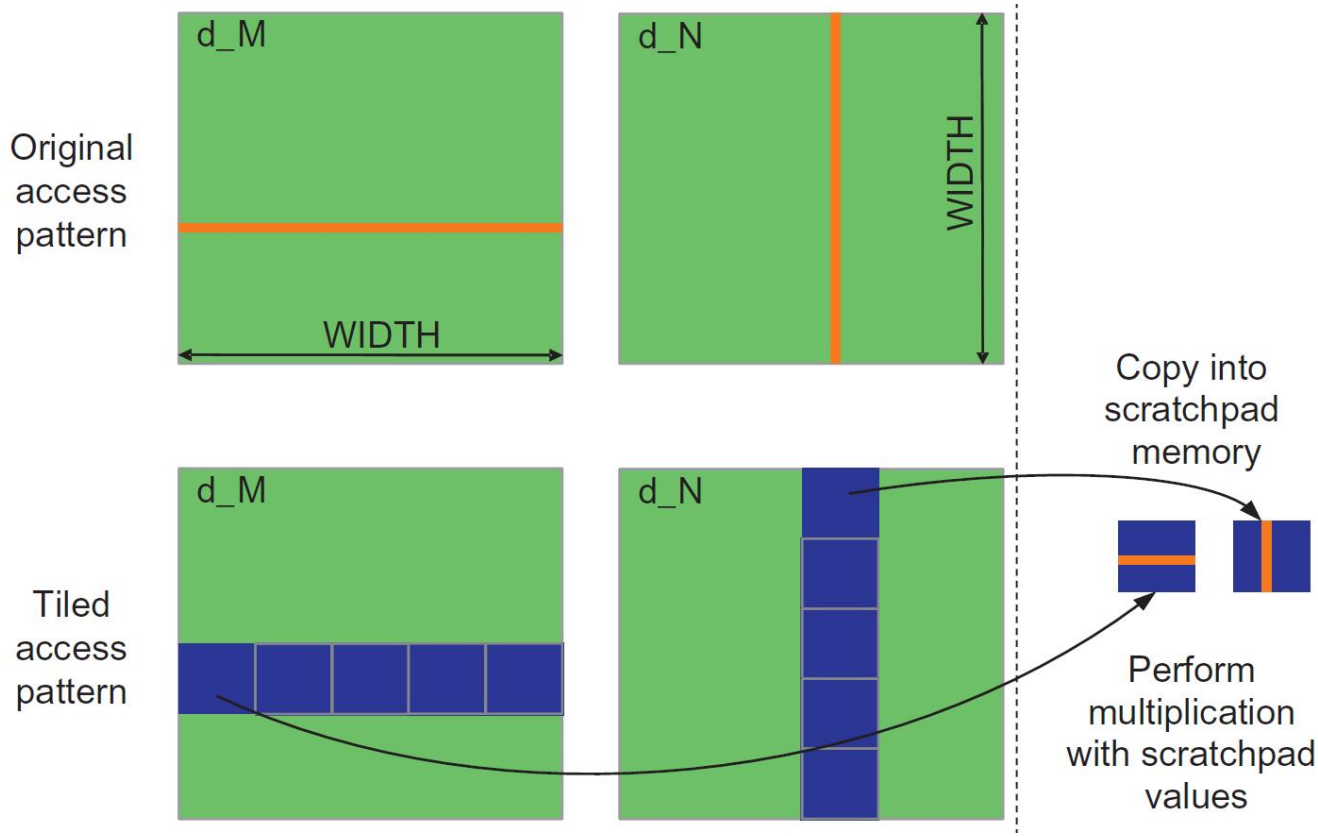
矩阵 N 的访存示意图：


Access direction in Kernel code
N0,0 N0,1 N0,2 N0,3
N1,0 N 1,1 N 1,2 N 1,3
N2,0 N2,1 N2,2 N2,3
N3,0 N3,1 N3,2 N3,3


Load iteration 0
T0 T1 T2 T3
Load iteration 1
T0 T1 T2 T3
...
N
N0,0 N0,1 N0,2 N0,3 N1,0 N1,1 N1,2 N1,3 N2,0 N2,1 N2,2 N2,3 N3,0 N3,1 N3,2 N3,3

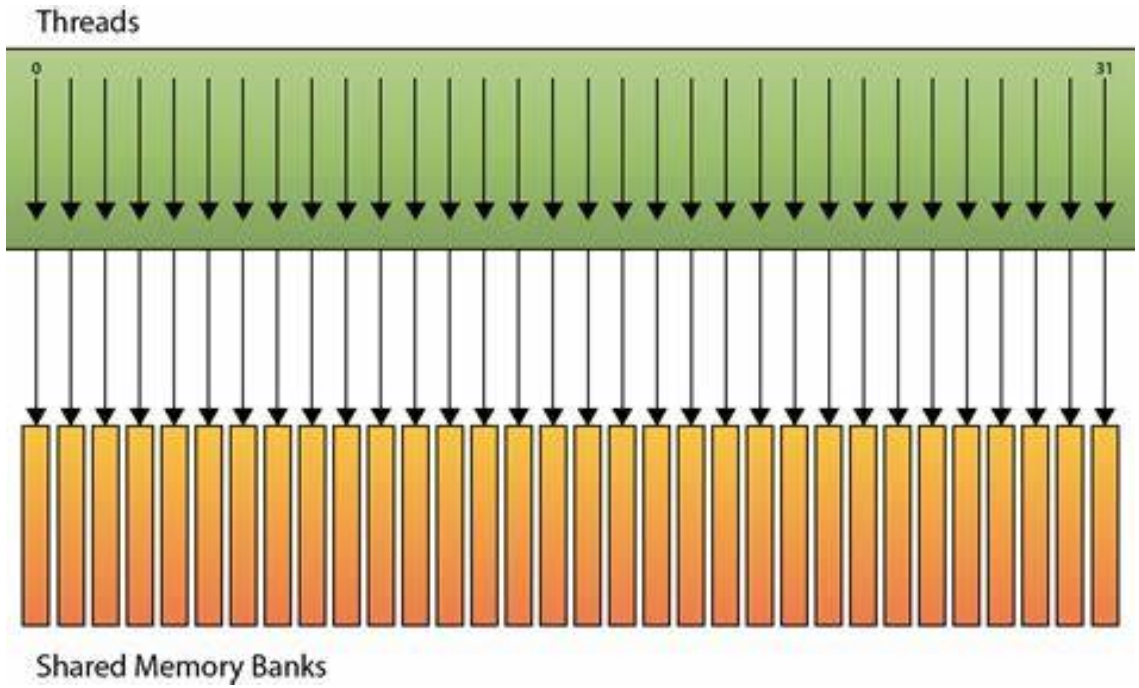
利用数据分块实现合并访存

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）.



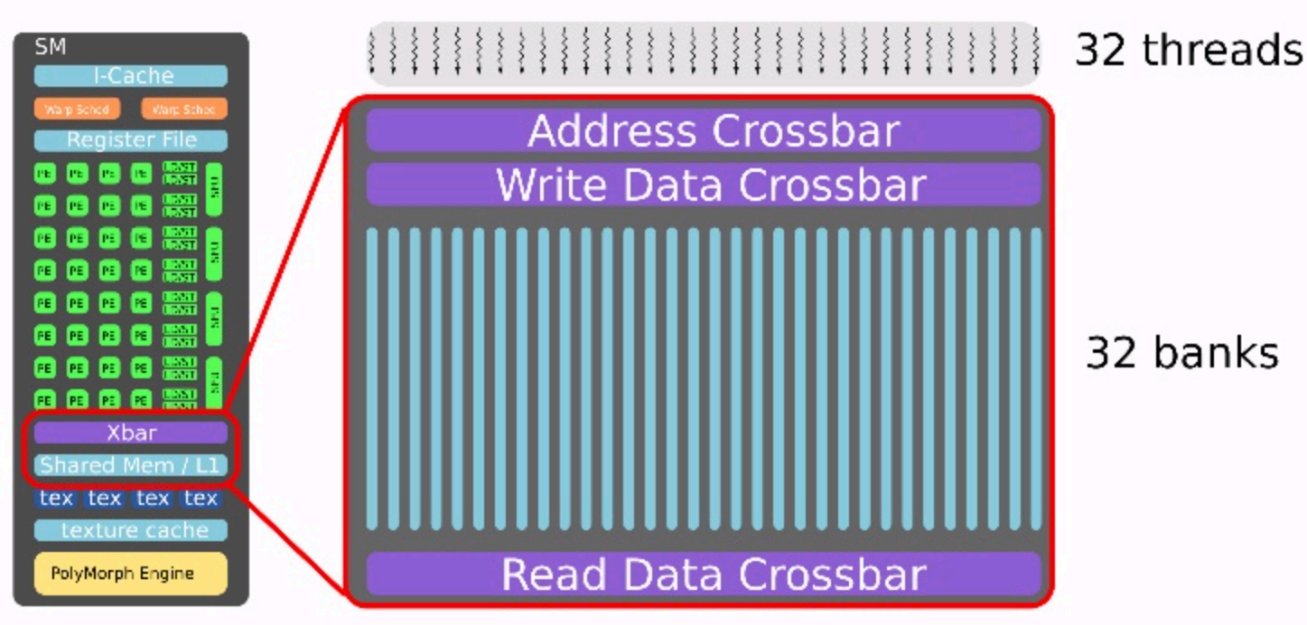
共享内存的组织

- 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.



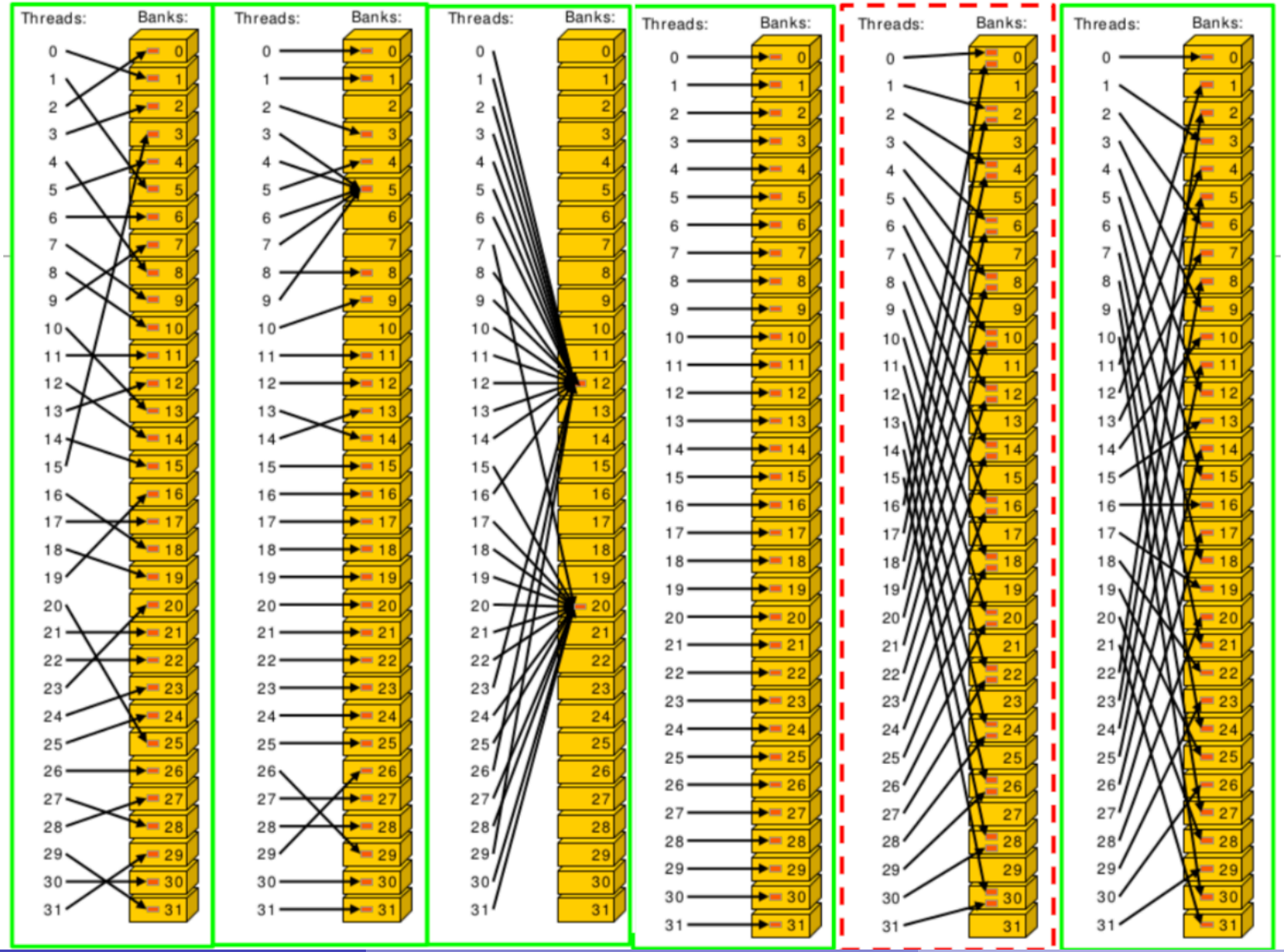
Bank Conflict

- 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

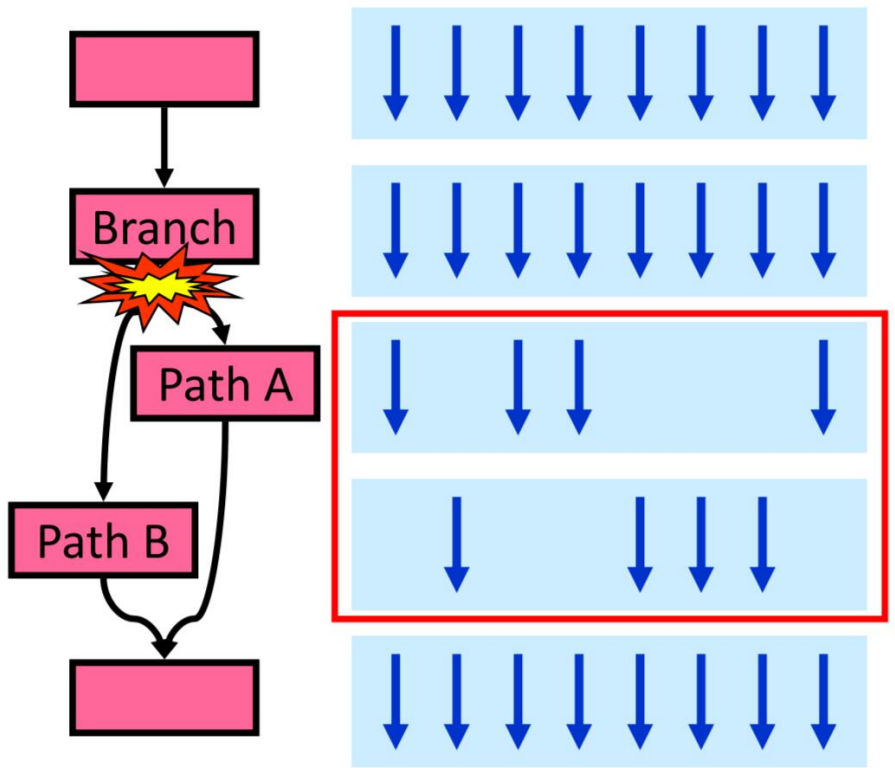
- 一般可通过调整访存跨步、偏置和补零等手段避免 bank conflict.



线程簇分歧

- 回顾：线程块将按照线程簇 (一般为 32 个线程) 为单元在 SM 上调度，同一线程簇中所有线程采用 SIMD (或称 SIMT) 方式执行.
- 线程簇分歧 (warp divergence)：当同一线程簇中线程执行不同程序路径时，会触发串行执行，导致程序性能下降.

```
if (...) {
    // Path A
} else {
    // Path B
}
```



例如，在向量加法中的边界检查：

```
if (i < n)d_C[i] = d_A[i] + d_B[i];
```

- 向量长度为 100 时，总共 4 个 warp，其中有 1 个 warp 产生分歧，占 25%.
- 向量长度为 1000 时，总共 32 个 warp，其中有 1 个 warp 产生分歧，占 3%.

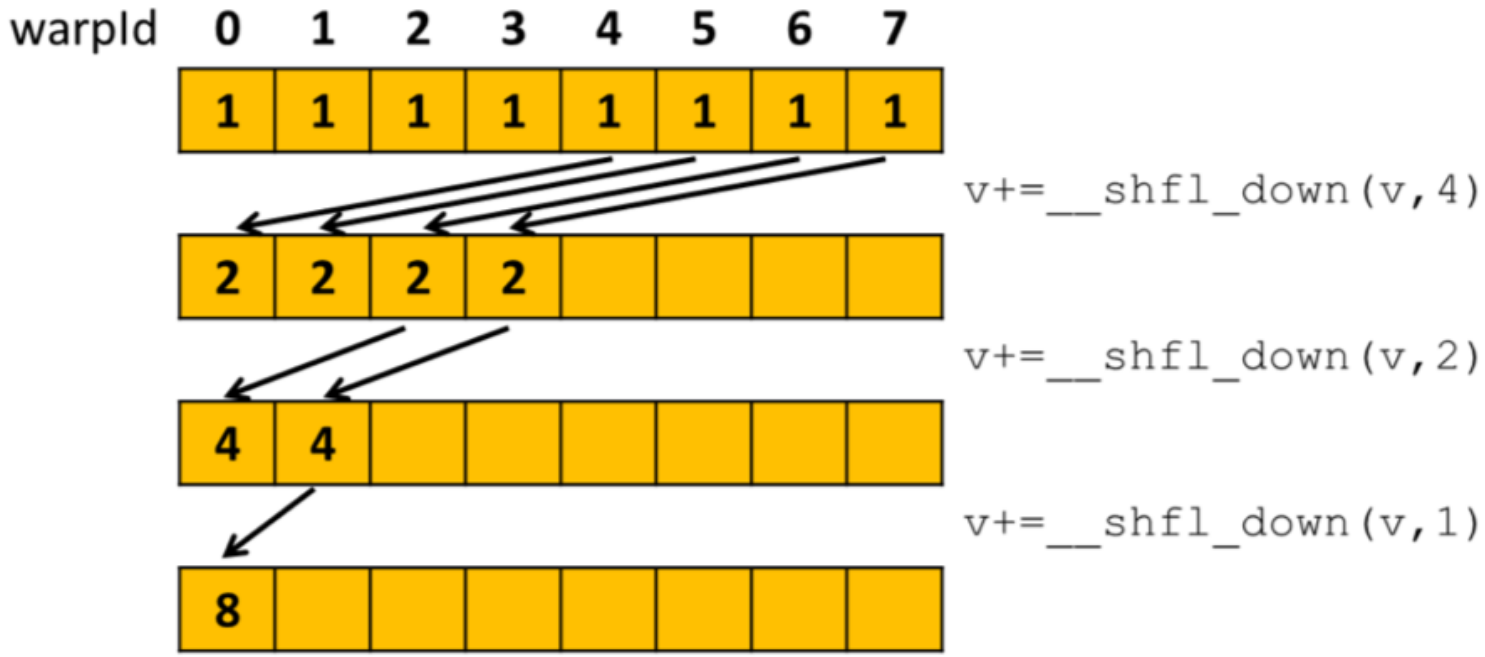
可以看出，对于边界检查，一般来说随着规模增加，分歧的影响会降低.

常见分歧的处理策略

- 线程分支：比如，在代码`if (threadIdx.x > 2)...` 中，线程 0，1，2 和线程 3-31 执行路径不同，程序分支尽量以线程簇大小作为粒度，设法改为`if (threadIdx.x / WARP_SIZE < 2)...`;
- 边界检查：例如向量加法中`if (i < n)d_C[i] = d_A[i] + d_B[i];`，如果开销太大，可以考虑使用两个 kernel，一个处理边界内的计算，一个处理边界情况.
- 线程改变：一些并行算法如 reduction 等，随着时间推移，参与的线程数目发生改变，可以通过设计新的并行算法来减少线程簇分歧.

线程簇混洗

- 线程簇混洗 (warp shuffle)：允许某线程直接读取同一个 warp 中其它线程寄存器中的值，延迟低且不占用额外的存储资源.



原子操作

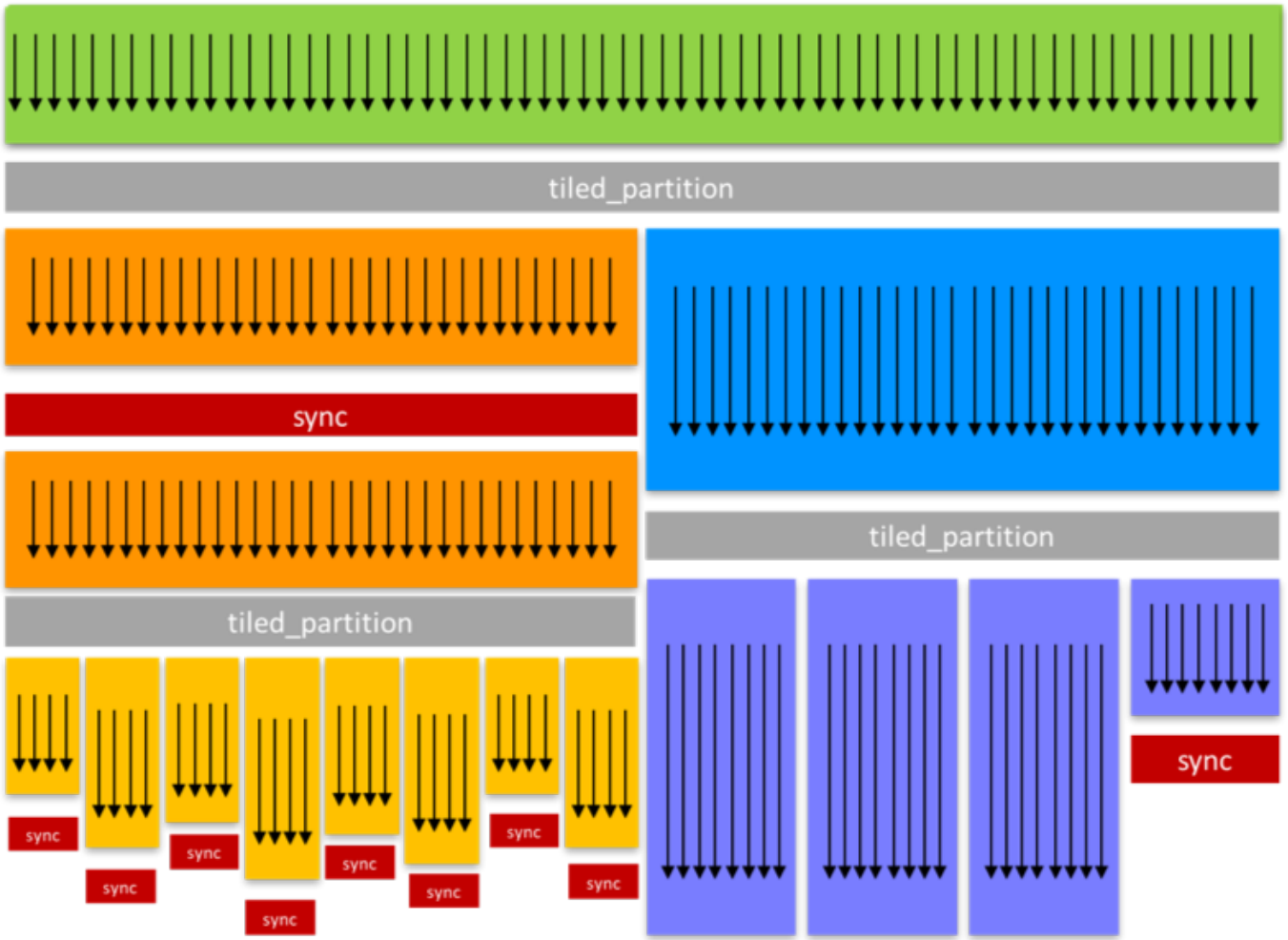
- 原子操作 (atomic operations)：当所有线程同时修改某个全局变量时，需要加锁后访问，保证结果的正确性，常见的原子操作有：

```
atomicAdd, atomicSub, atomicMin, atomicMax,
atomicInc, atomicDec, atomicExch, atomicCAS,
atomicAnd, atomicOr, atomicXor
```

- 线程簇聚合 (warp aggregation)：多个线程原子累加到单个计数器以提高性能，线程簇中的线程首先计算它们之间的总增量，然后选择单个线程将增量原子地添加到全局计数器中；
- CUDA9.0 以上的 NVCC 编译器已官方支持自动地为原子操作执行线程簇聚合.

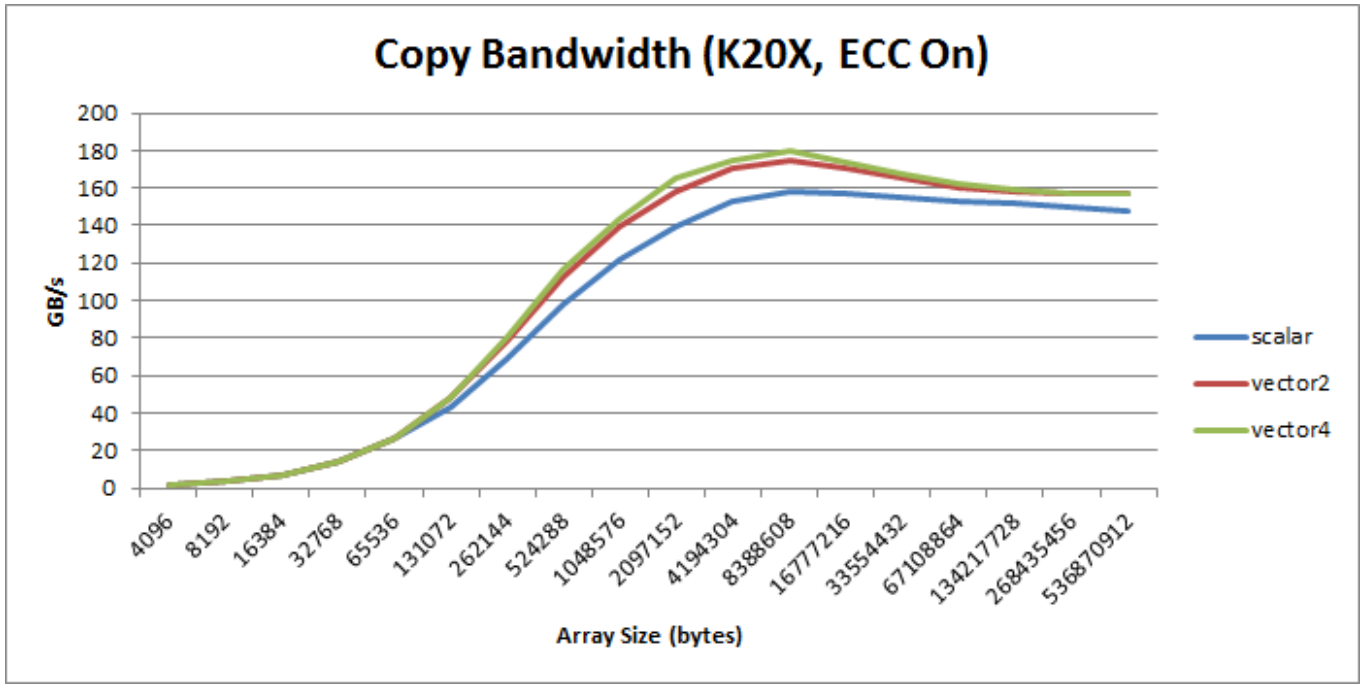
协同分组

- 协同分组 (cooperative groups)：CUDA9.0 以上，支持更为灵活的线程组合方式，从而可以在不同粒度上进行线程间协作.



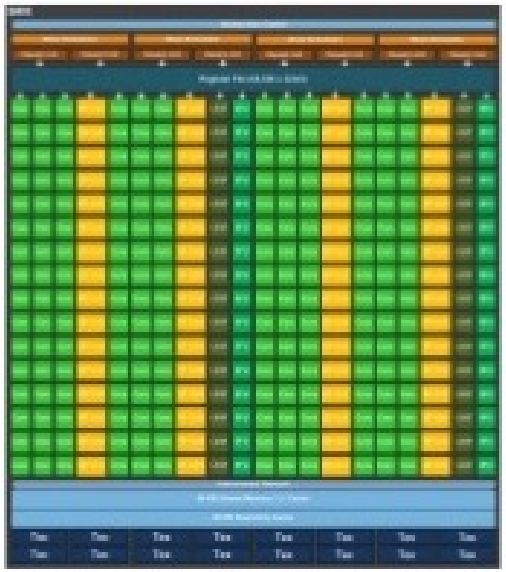
向量化访存

- 向量化访存 (vectorized memory access)：对于访存受限的问题，在保证数据对齐的前提下，通过使用例如 `float2` 等向量化的数据类型，并结合 `reinterpret_cast` 对指针进行强制转换，可以帮助编译器实现访存的向量化，从而提升性能.

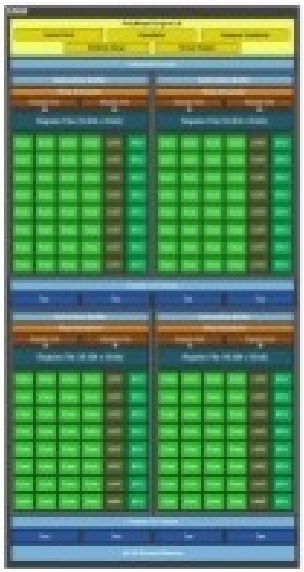


NVIDIA GPU 的计算能力 (compute capability)

- 计算能力用于反映 CUDA 设备所支持的不断更迭的功能和特性；
- 计算能力以 `X.Y` 表示，两个数字分别为主版本和从版本号；
- 计算能力的版本之间向后兼容，越新表示设备的功能越强大；

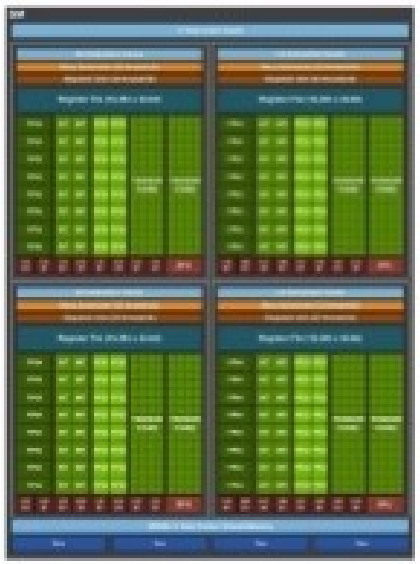


Kepler
CC 3.5
192 cores / SMX



Maxwell
CC 5.0
128 cores / SMM

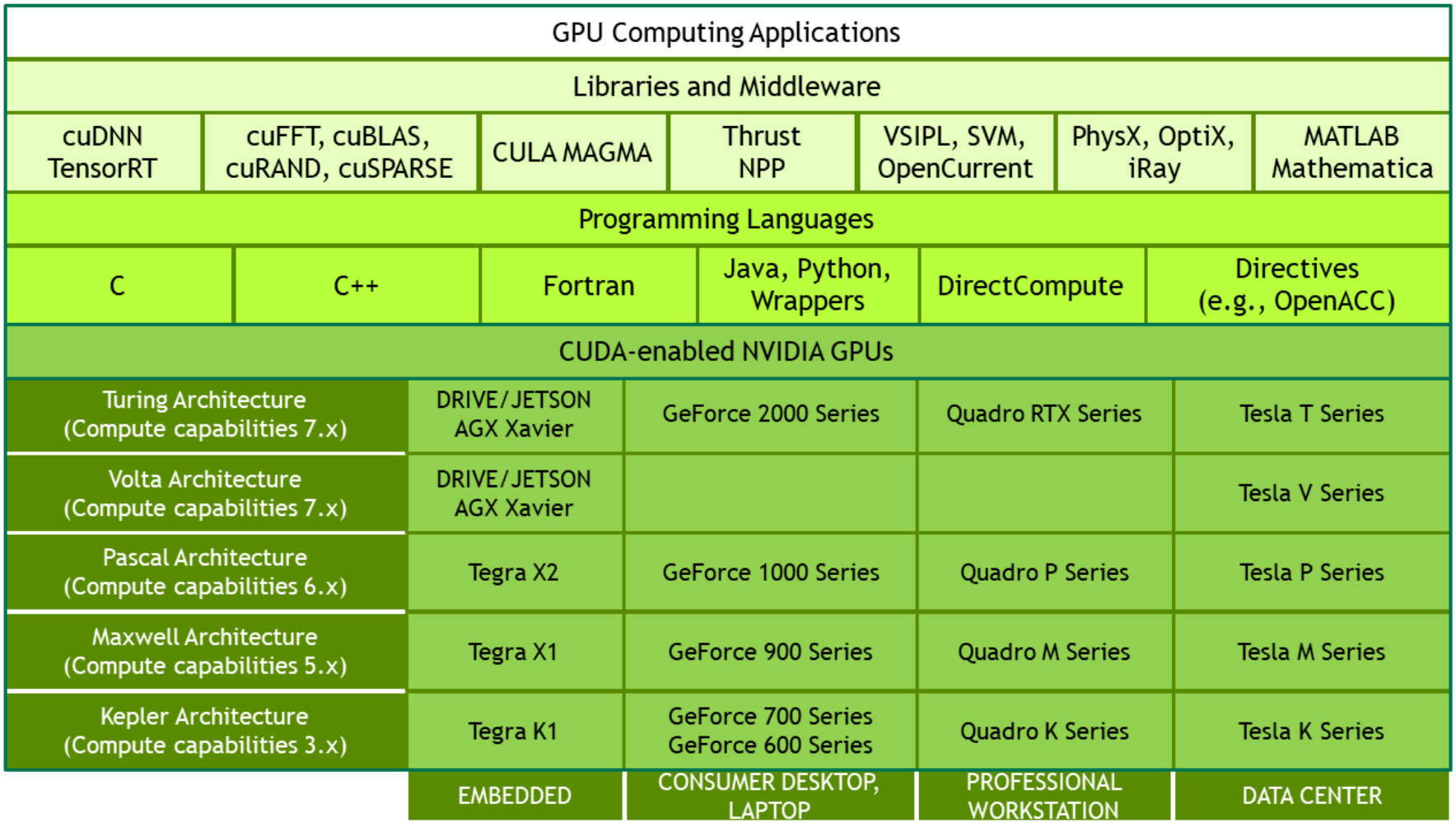
Pascal
CC 6.0
64 cores / SMM



Volta
CC 7.0
64 cores / SMM

https://developer.nvidia.com/cuda-gpus

- 不同计算能力、不同产品线的设备构成了庞大的生态体系.



- CUDA 提供了 `deviceQuery` 样例程序用于检查设备的计算能力.

```
$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Quadro GV100"
  CUDA Driver Version / Runtime Version          10.1 / 10.1
  CUDA Capability Major/Minor version number:    7.0
  Total amount of global memory:                 32508 MBytes (34087305216 bytes)
  (80) Multiprocessors, ( 64) CUDA Cores/MP:     5120 CUDA Cores
  GPU Max Clock rate:                            1627 MHz (1.63 GHz)
  Memory Clock rate:                             850 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  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 threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 4 copy engine(s)
```

Feature Support	Compute Capability					
(Unlisted features are supported for all compute capabilities)	**3.0**	**3.2**	**3.5, 3.7, 5.0, 5.2**	**5.3**	**6.x**	**7.x**
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	Yes					
atomicExch() operating on 32-bit floating point values in global memory (atomicExch())	Yes					
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	Yes					
atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())	Yes					
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)	Yes					
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	Yes					
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())	Yes					
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No				Yes	
Warp vote and ballot functions (Warp Vote Functions)	Yes					
__threadfence_system() (Memory Fence Functions)						
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)						
Surface functions (Surface Functions)						
3D grid of thread blocks						
Unified Memory Programming						
Funnel shift (see reference manual)	No	Yes				
Dynamic Parallelism	No		Yes			
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No			Yes		
Tensor Core	No					Yes

	Compute Capability											
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.5**
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4	32				16	128	32	16	128	
Maximum dimensionality of grid of thread blocks	3											
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$											
Maximum y- or z-dimension of a grid of thread blocks	65535											
Maximum dimensionality of thread block	3											
Maximum x- or y-dimension of a block	1024											
Maximum z-dimension of a block	64											
Maximum number of threads per block	1024											
Warp size	32											
Maximum number of resident blocks per multiprocessor	16				32							16
Maximum number of resident warps per multiprocessor	64											32
Maximum number of resident threads per multiprocessor	2048											1024
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K							
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	63	255										
Maximum amount of shared memory per multiprocessor	48 KB			112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB	64 KB
Maximum amount of shared memory per thread block [27]	48 KB										96 KB	64 KB
Number of shared memory banks	32											
Amount of local memory per thread	512 KB											
Constant memory size	64 KB											
Cache working set per multiprocessor for constant memory	8 KB							4 KB	8 KB			

不同计算能力的 GPU 所支持的最大资源数

GPU	Kepler GK110	Maxwell GM200	Pascal GP100
Compute Capability	3.5	5.2	6.0
Threads / Warp	32	32	32
Max Warps / Multiprocessor	64	64	64
Max Threads / Multiprocessor	2048	2048	2048
Max Thread Blocks / Multiprocessor	16	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	32768	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB

占有率

- 占有率 (occupancy)：每个 SM 中活跃线程簇数与最大线程簇数的比值，越接近 100% 一般越好.
- 实际占有率往往受限于 kernel 的硬件资源消耗，主要硬件资源有：
  - 每个 SM 寄存器的容量；
  - 每个 SM 共享内存的容量；
  - 每个 SM 允许的最大线程块数；
  - 每个 SM 允许的最大线程数；
  - 每个线程块允许的最大线程数.
- 上述因素与具体硬件的计算能力密切相关.

几个例子

P100 每个 SM 寄存器容量为 256KB、共享内存容量为 64KB、最大活跃线程簇数 64 (即 2048 线程)，最大活跃线程块数 32.

例子 1：寄存器限制

若每个线程使用 32 个单精度寄存器，则活跃线程为 256KB÷(4B×32) = 2048，占有率为 1；若每个线程使用 64 个单精度寄存器，占有率则降低为 0.5.

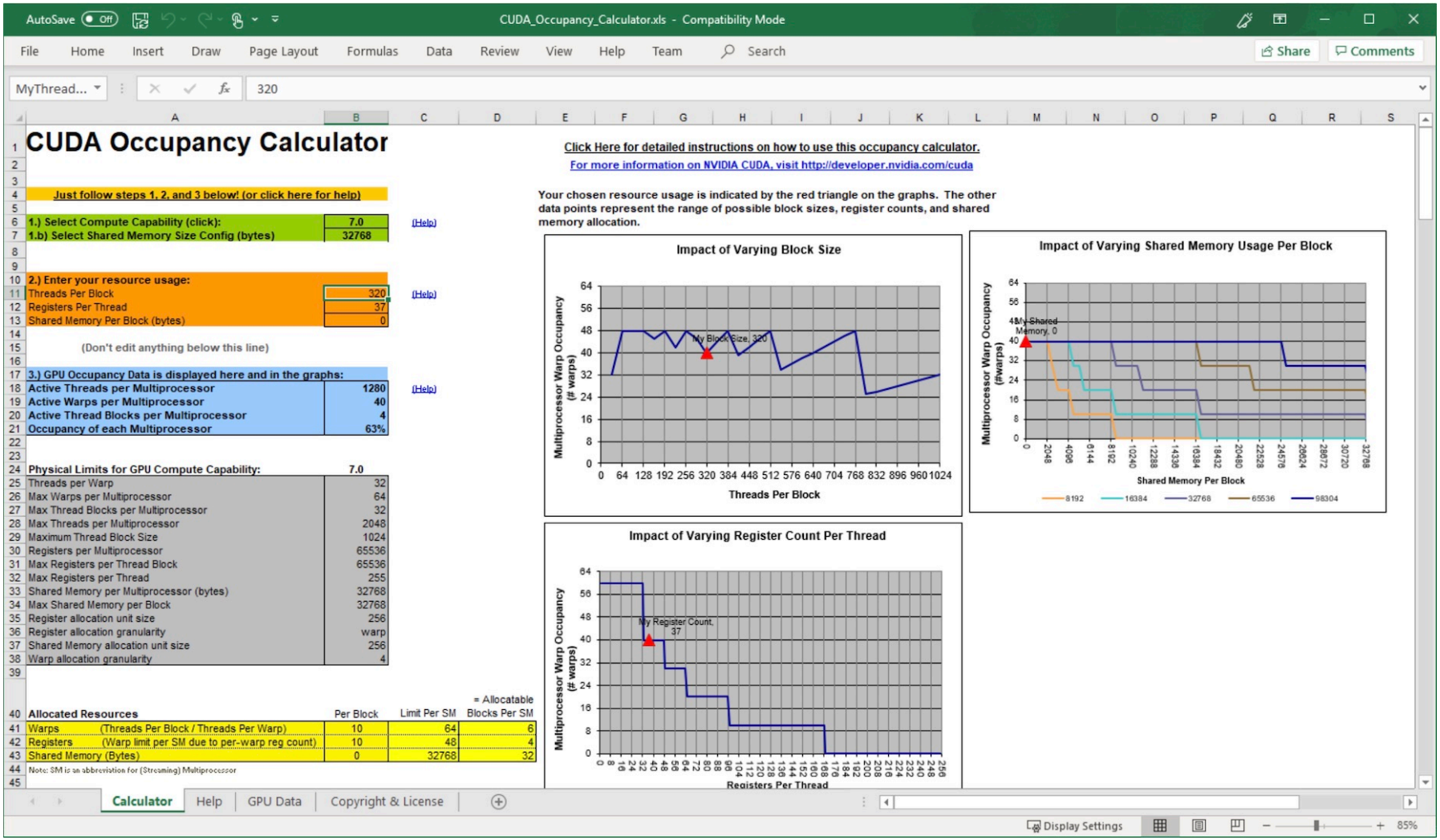
例子 2：共享内存限制

若每个线程平均使用 32Byte 共享内存，则活跃线程为 64KB/32B=2048，占有率为 1；若每个线程平均使用 64B 共享内存，占有率则降低为 0.5.

例子 3：线程块大小限制

若每个线程块有 32 个线程，占有率则不会高于 32×32÷2048 = 0.5.

CUDA 占有率计算工具



CUDA 占有率计算函数

- 根据 kernel 预估块大小：

```
1 CUresult cuOccupancyMaxPotentialBlockSize(
2     int* minGridSize, int* blockSize, CUfunction func,
3     CUoccupancyB2DSize blockSizeToDynamicSMemSize,
4     size_t dynamicSMemSize, int  blockSizeLimit )
```

- 返回 kernel 实际运行块数：

```
1 CUresult cuOccupancyMaxActiveBlocksPerMultiprocessor(
2     int* numBlocks, CUfunction func,
3     int blockSize, size_t dynamicSMemSize )
```