CUDA编程模型

汤善江 副教授
天津大学智能与计算学部
tashj@tju.edu.cn
http://cic.tju.edu.cn/faculty/tangshanjiang/

Outline

- CUDA编程环境概述
  - NIVIDIA GPU体系架构
  - CUDA软件开发环境
  - CUDA异构计算
- CUDA程序设计
  - 线程创建与管理
  - 线程调度与执行

Outline

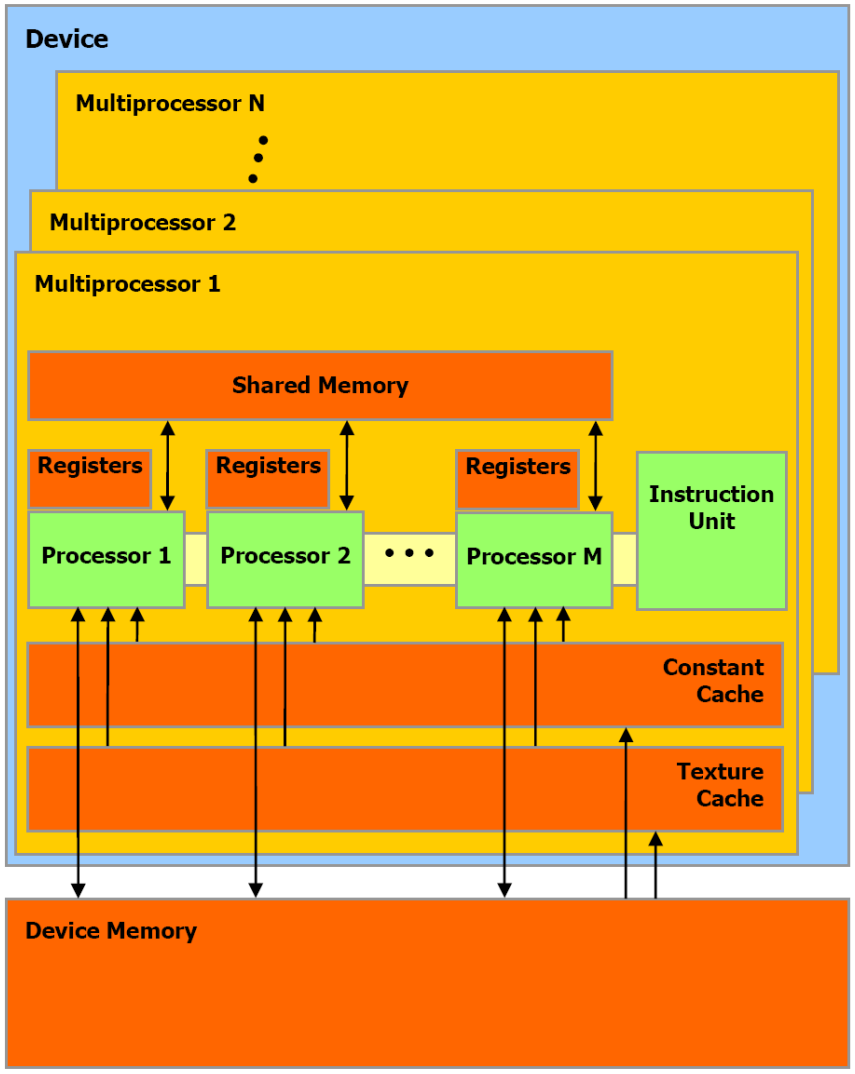
- **CUDA编程模型概述**
  - **NIVIDIA GPU体系架构**
  - CUDA软件开发环境
  - CUDA异构计算
- CUDA程序设计
  - 线程创建与管理
  - 线程调度与执行

GPU架构系列

- GPU架构
  - Fermi (2010)
  - Kepler (2012)
  - Maxwell (2014)
  - Pascal (2016)
  - Volta (2017)
  - Turing (2018)
  - Ampere(2021)
  - Hopper(2022)
- 芯片型号
  - GT200
  - GK210
  - GM104
  - GF104
- 显卡系列
  - GeForce
  - Quadro
  - Tesla

GPU架构

- 基本硬件架构：
- 流多处理器（Stream Multiprocessor）
- 流处理器（Stream Processor）
- 共享内存（Shared Memory）
- 板载显存（Device Memory）

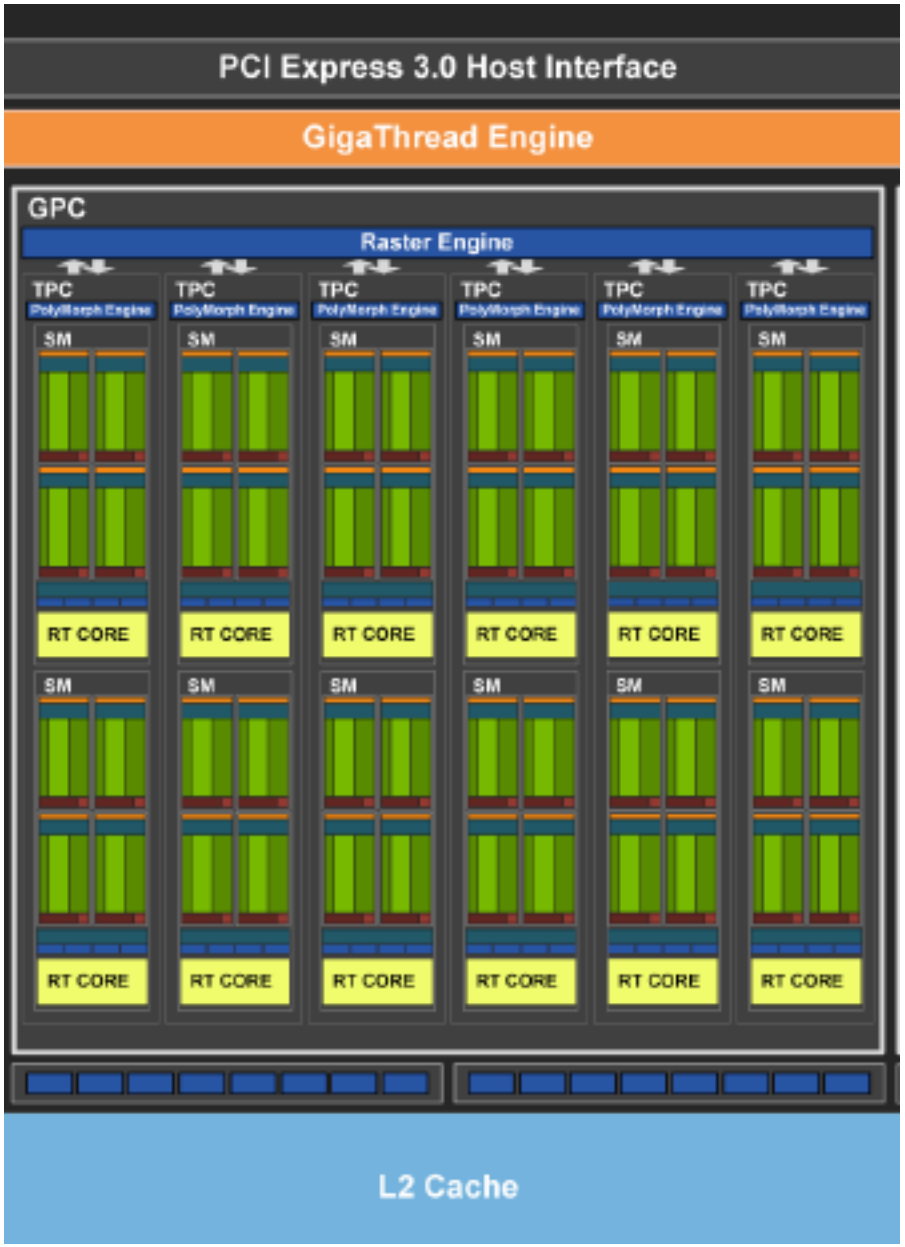


GPU逻辑架构(示例：TU102)

- Turing 架构
- 每个 GPU 封装6个 **GPC** (Graphics Processing Cluster)

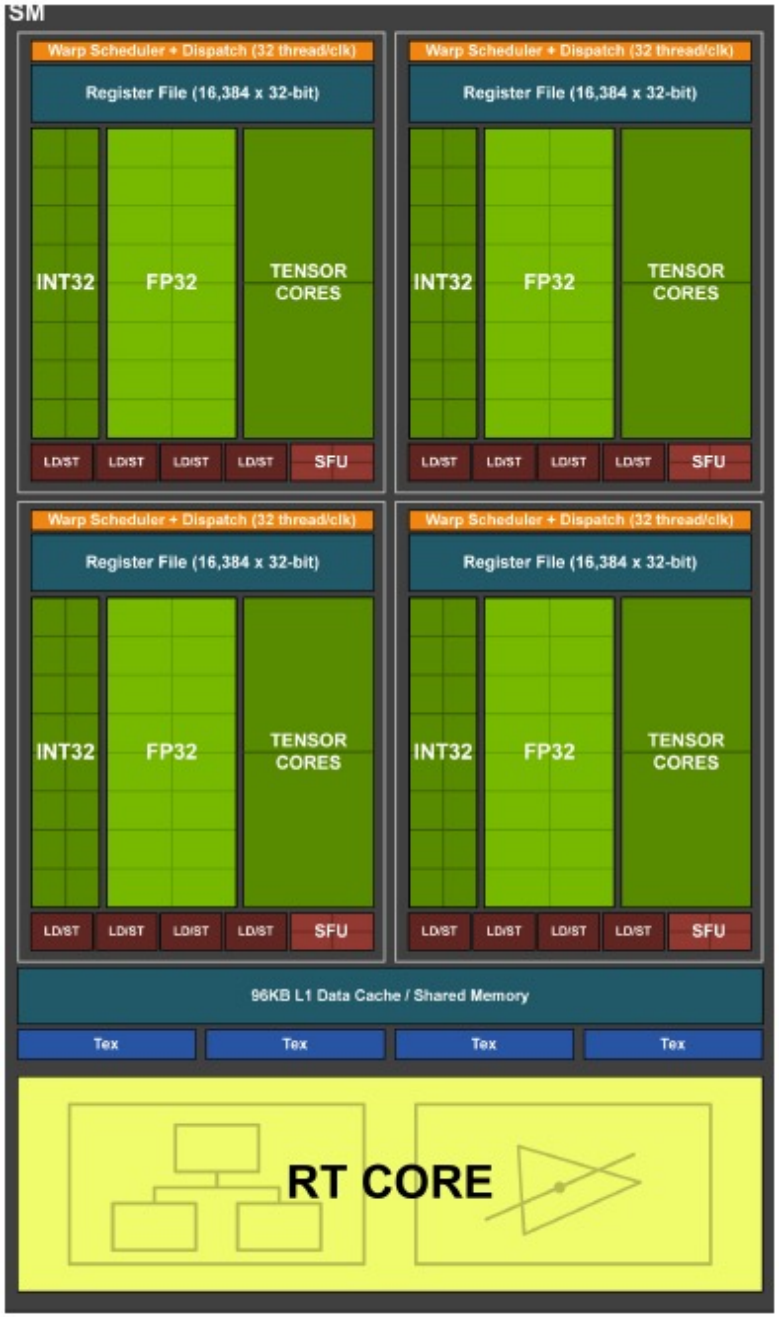
GPC 架构

- 每个 GPC 封装6个 **TPC**（Texture Processing Clusters）
- 每个 TPC 封装2个 **SM**（Streaming Multiprocessor）



SM（TU102）

- 分为4个 block，96KB 共享缓存
- 每个 block
  - FP32核：16个
  - INT32核：16个
  - Tensor核：2个
  - 寄存器：16384个，32位
- RT 核
  - 用于 Ray Tracing
- 另有2个 FP64核
  - FP32计算速度的1/32

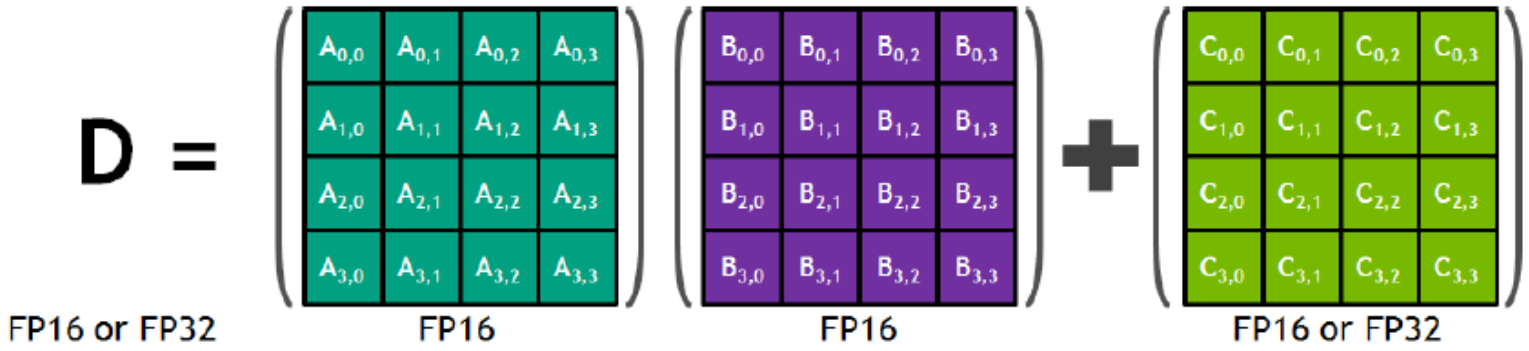


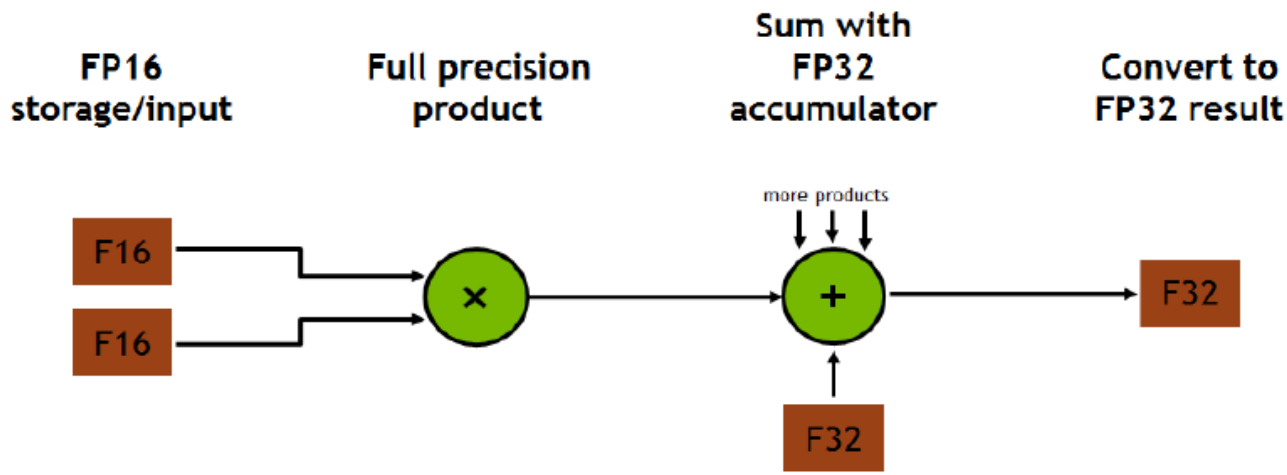
SM （V100）

- 分为4个 block，**128KB** 共享缓存
- 每个 block
  - **FP64核：8个**
  - FP32核：16个
  - INT32核：16个
  - Tensor核：2个
  - 寄存器：16384个，32位

Tensor Core

- 混合精度矩阵计算





Tensor Core


腾讯视频
不 负 好 时 光
Tensor Cores承担混合精度训练的苦力任务
最开始是由Volta架构引入，现在图灵架构的所有产品都具备

NVIDIA Tesla

共84个，只使用80个

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

SPECIFICATIONS

GPU Architecture	**NVIDIA Turing**
NVIDIA Turing Tensor Cores	**320**
NVIDIA CUDA® Cores	**2,560**
Single-Precision	**8.1 TFLOPS**
Mixed-Precision (FP16/FP32)	**65 TFLOPS**
INT8	**130 TOPS**
INT4	**260 TOPS**
GPU Memory	**16 GB GDDR6 300 GB/s**
ECC	**Yes**
Interconnect Bandwidth	**32 GB/sec**
System Interface	**x16 PCIe Gen3**
Form Factor	**Low-Profile PCIe**
Thermal Solution	**Passive**
Compute APIs	**CUDA, NVIDIA TensorRT™, ONNX**

NVIDIA Tesla

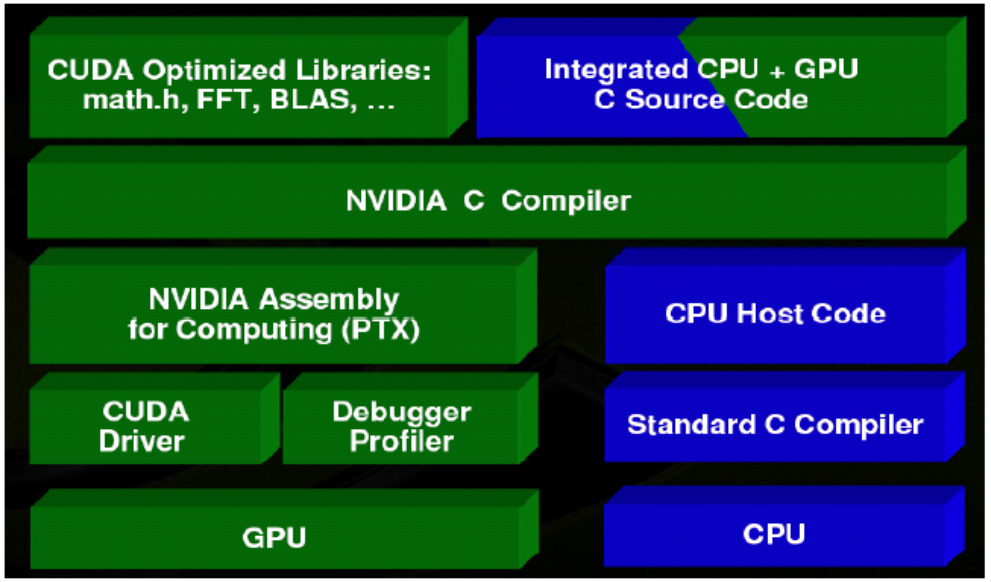
	"Fermi"	"Fermi"	"Kepler"	"Kepler"	"Maxwell"	"Pascal"	"Volta"	"Turing"	"Ampere"	"Hopper"
Tesla GPU	**GF100**	**GF104**	**GK104**	**GK110**	**GM200**	**GP100**	**GV100**	**TU104**	**GA100**	**GH100**
Compute Capability	2.0	2.1	3.0	3.5	5.3	6.0	7.0	7.0	8.0	9.0
Streaming Multiprocessors	16	16	8	15	24	56	84	72	128	144
FP32 Cores / SM	32	32	192	192	128	64	64	64	64	128
FP32 Cores	512	512	1,536	2,880	3,072	3,584	5,376	4,608	8,192	18,432
FP32 Cores / SM			64	64	4	32	32	-	32	64
FP64 Cores	-	-	512	960	96	1,792	2,688	-	4,096	9,216
Tensor Core Units							672	576	512	576
Threads / Warp	32	32	32	32	32	32	32	32	32	32
Max Warps / SM	48	48	64	64	64	64	64	64	64	64
Max Threads / SM	1,536	1,536	2,048	2,048	2,048	2,048	2,048	2,048	2,048	2,048
Max Thread Blocks / SM	8	8	16	16	32	32	32	32	32	32
32-bit Registers / SM	32,768	32,768	65,536	65,536	65,536	65,536	65,536	65,536	65,536	65,536
Max Registers / Thread	63	63	63	255	255	255	255	255	255	255
Max Threads / Thread Block	1,024	1,024	1,024	1,024	1,024	1,024	1,024	1,024	1,024	1,024
Shared Memory Size Configs	16 KB	16 KB	16 KB	16 KB	96 KB	64 KB	Config	Config	Config	Config
	48 KB	48 KB	32 KB	32 KB			Up To	Up To	Up To	Up To
			48 KB	48 KB			96 KB	96 KB	164 KB	228 KB
Hyper-Q	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic Parallelism	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Unified Memory	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes
Pre-Emption	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes
Sparse Matrix	No	No	No	No	No	No	No	No	No	Yes
Dynamic Programming	No	No	No	No	No	No	No	No	No	Yes

Outline

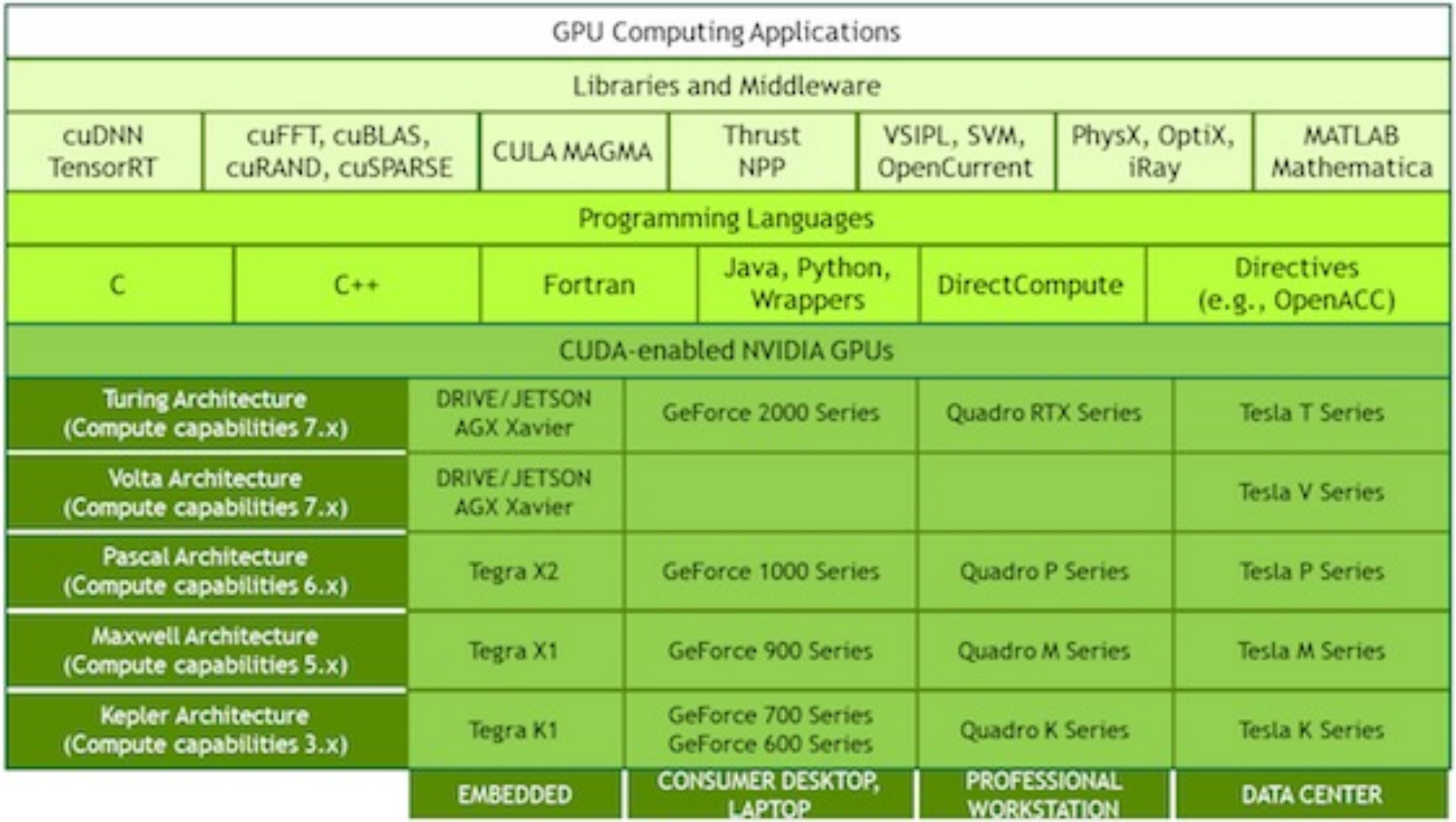
- **CUDA编程模型概述**
  - NIVIDIA GPU体系架构
  - **CUDA软件开发环境**
  - CUDA异构计算
- CUDA程序设计
  - 线程创建与管理
  - 线程调度与执行

CUDA-Compute Unified Device Architecture

- 2006年，NVIDIA 推出
- 支持多种编程语言
- SIMT 模式
  - 单指令多线程
- 参考文档
  - https://docs.nvidia.com/cuda/
- 软件工具
  - https://developer.nvidia.com/cuda-downloads
- 编程指南
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

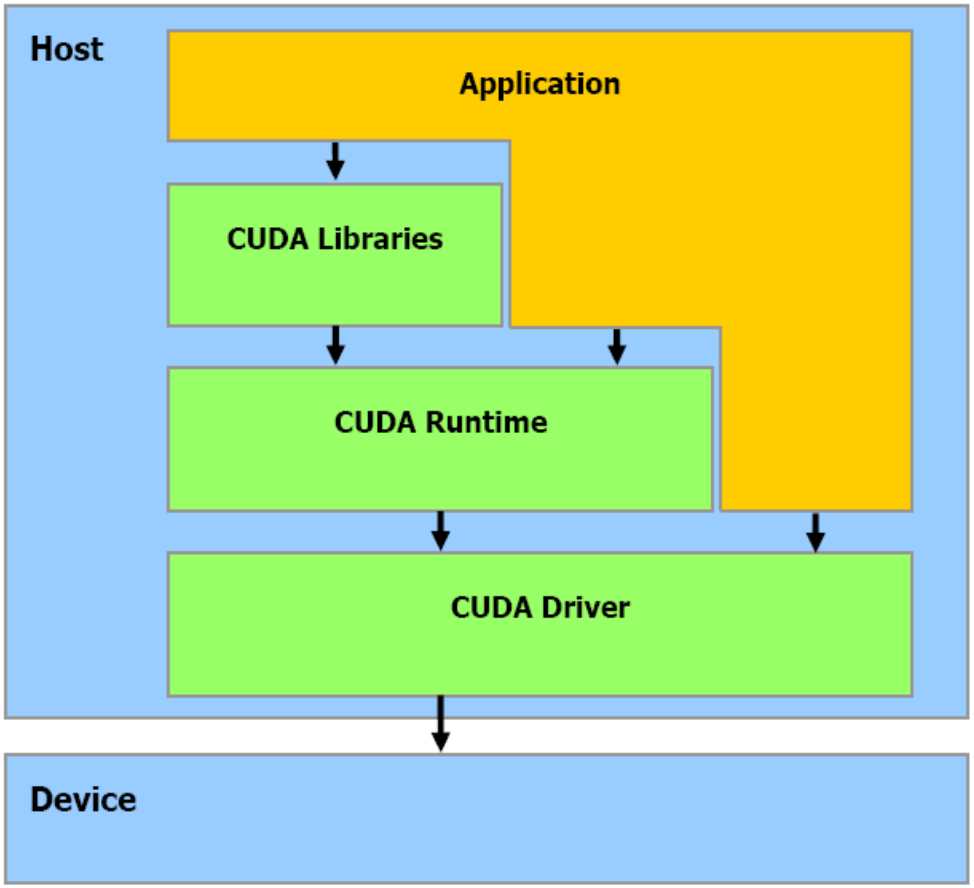


CUDA-Compute Unified Device Architecture



CUDA软件栈

- CUDA 软件栈包含多个层：
  - 设备驱动程序（CUDA Driver）
  - 应用程序编程接口（API）及其运行时（CUDA Runtime）
  - 两个较高级别的通用数学库
    - CUFFT（离散快速傅立叶变换）
    - CUBLAS（离散基本线性计算）



CUDA软件开发包

cuda是免费使用的，各种操作系统下的cuda安装包均可以在 **http://www.nvidia.cn/object/cuda_get_cn.html** 上免费下载。

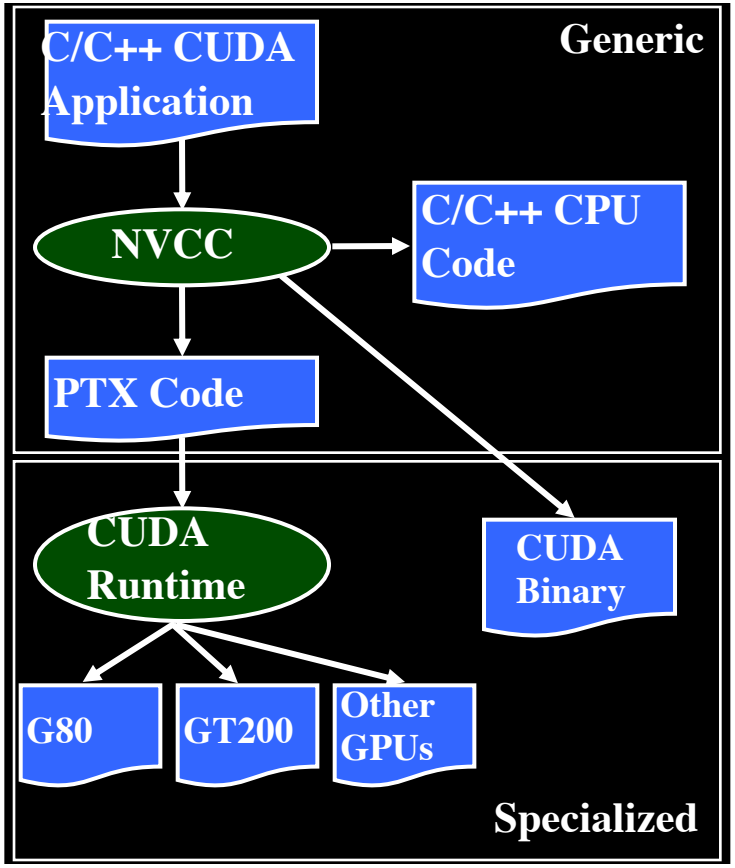
CUDA安装

- 安装 CUDA Display
  - 对于没有安装NV显卡的计算机，不需要安装Display安装包，程序也可以在模拟模式下运行。
- 安装cuda toolkit
  - 安装过程中toolkit自动设定了3个环境变量：CUDA_BIN_PATH、CUDA_INC_PATH和CUDA_LIB_PATH分别对应工具程序库、头文件库和程序库，预设路径为当前安装文件夹下的bin、include 和lib三个文件夹。
  - 并将bin文件夹目录加入环境变量path中。
- 安装CUDA SDK
  - SDK中的许多例子程序和函数库非常有用。

具体操作可参考“CUDA easy start up”视频和“在windows下安装CUDA”文件。

NVCC 编译器

- 生成三种不同的输出：PTX，CUDA二进制序列和标准C



NVCC 编译器 PTX

- PTX(Parallel Thread eXecution)作用类似于汇编，是为动态编译器(包含在标准的Nvidia 驱动中)设计的输入指令序列。这样，不同的显卡使用不同的机器语言，而动态编译器却可以运行相同的PTX。这样做使PTX成为了一个稳定的接口，带来了很多好处：后向兼容性，更长的寿命，更好的可扩展性和更高的性能，但在一定程度上也限制了工程上的自由发挥。这种技术保证了兼容型，但也使新一代的产品必须拥有上代产品的所有能力，这样才能让今天的PTX代码在未来的系统上仍然可以运行。

NVCC 编译器 CUBIN

- 虽然PTX和JIT编译器提供了很高的性能，但也不是在所有的场合都适用。某些独立软件开发商倾向于牺牲性能，以获得更好的可确定性和可验证性。JIT编译器的输出随着目标硬件和一些其他因素会发生变化。对于需要能够确定的代码的独立软件开发商(比如很多财经软件开发商)，它们可以将代码直接编译成CUDA二进制代码，这样就能避免JIT过程的不确定性。直接编译得到的CUDA二进制代码是与特定的硬件和驱动相关的。

NVCC 编译器 C

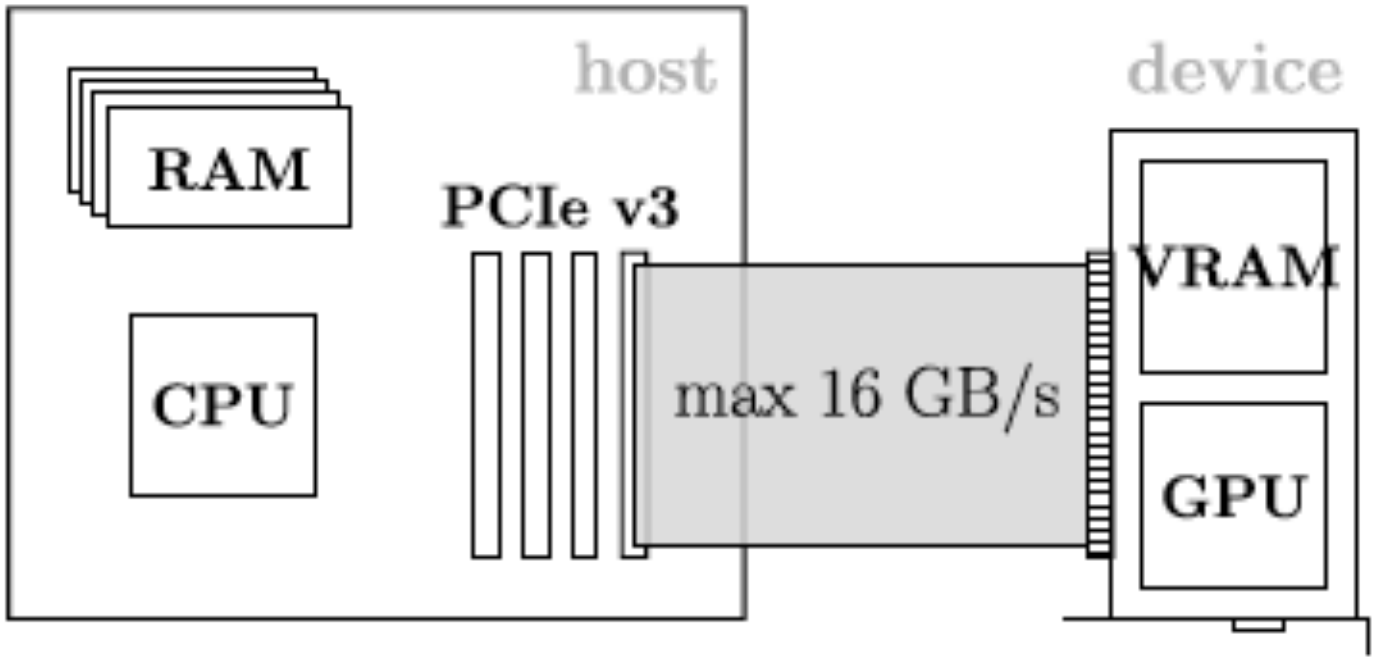
- Nvcc的输出还包括标准C。由nvcc生成的C代码将被重定向到其他编译器进行编译，比如ICC，GCC或者其他合适的高性能编译器。CUDA中明确的表示了程序中的并行度不仅在用于编写运行在Nvidia GPU上的代码时非常有效，而且为多核CPU生成高性能代码。在某些应用中，CUDA生成的代码比标准的x86编译器生成的代码的性能提高了4倍。

Outline

- **CUDA编程模型概述**
  - NIVIDIA GPU体系架构
  - CUDA软件开发环境
  - **CUDA异构计算**
- CUDA程序设计
  - 线程创建与管理
  - 线程调度与执行

CUDA异构计算

- 可用于计算的硬件分为两部分
  - Host（主机，CPU），负责控制/指挥 GPU 工作
  - Device（GPU），协处理



CUDA-异构计算

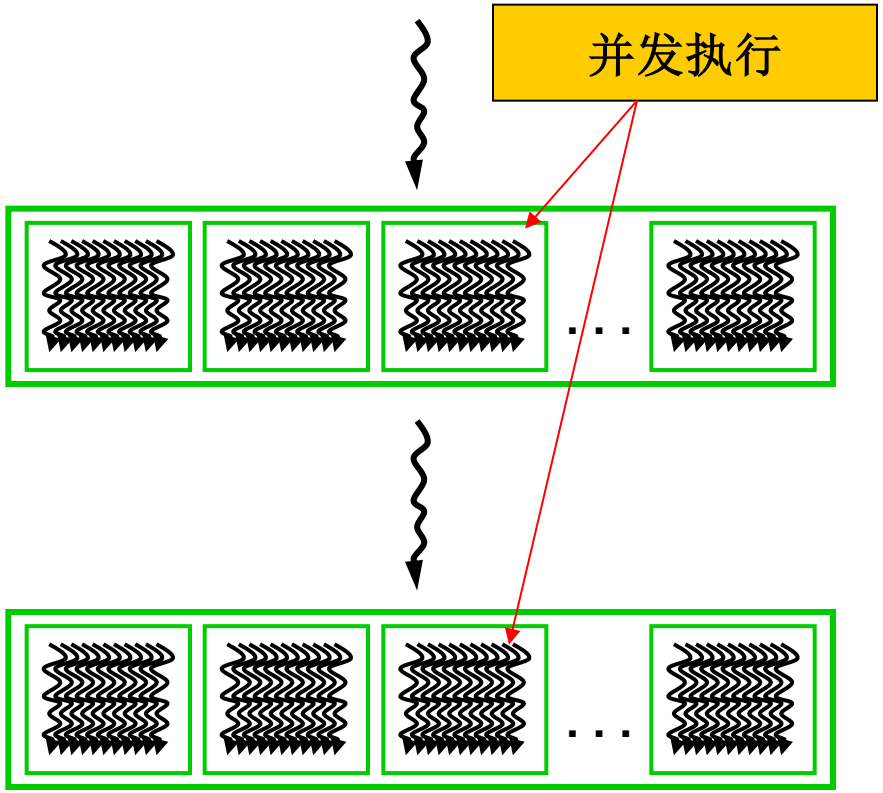
- **Host+device**异构并行C应用程序
  - **Host**端串行C代码
  - **Device**端SPMD并行化**kernel**（内核）C代码

CPU串行代码 (host)

GPU并行内核 (device)
KernelA<<< nBlk, nTid >>>(args);

CPU串行代码 (host)

GPU并行内核 (device)
KernelB<<< nBlk, nTid >>>(args);



CUDA函数定义

	Executed on the:	Only callable from the:
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

- __global__定义kernel函数
  - 必须返回void
- __device__和__host__ 可以组合使用
  - 则被定义的函数在CPU和GPU上都被编译

CUDA函数定义

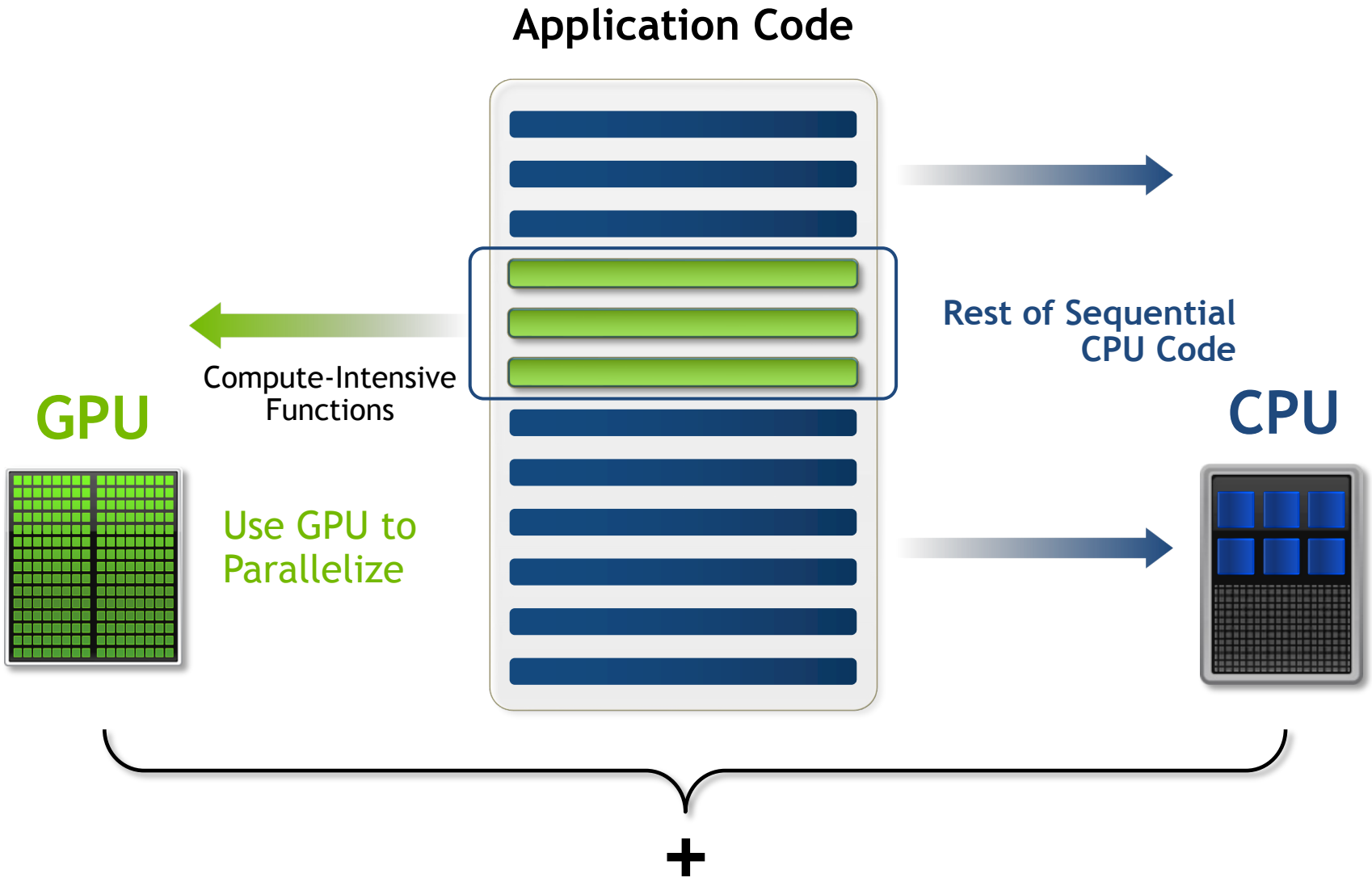
- __device__ 函数不能用&运算符取地址
- 限制
  - 不支持递归调用
  - 不支持静态变量(static variable)
  - 不支持可变长度参数函数调用
    - type va_list(stdarg.h)
    - double average(int count, ...)

Kernel函数调用

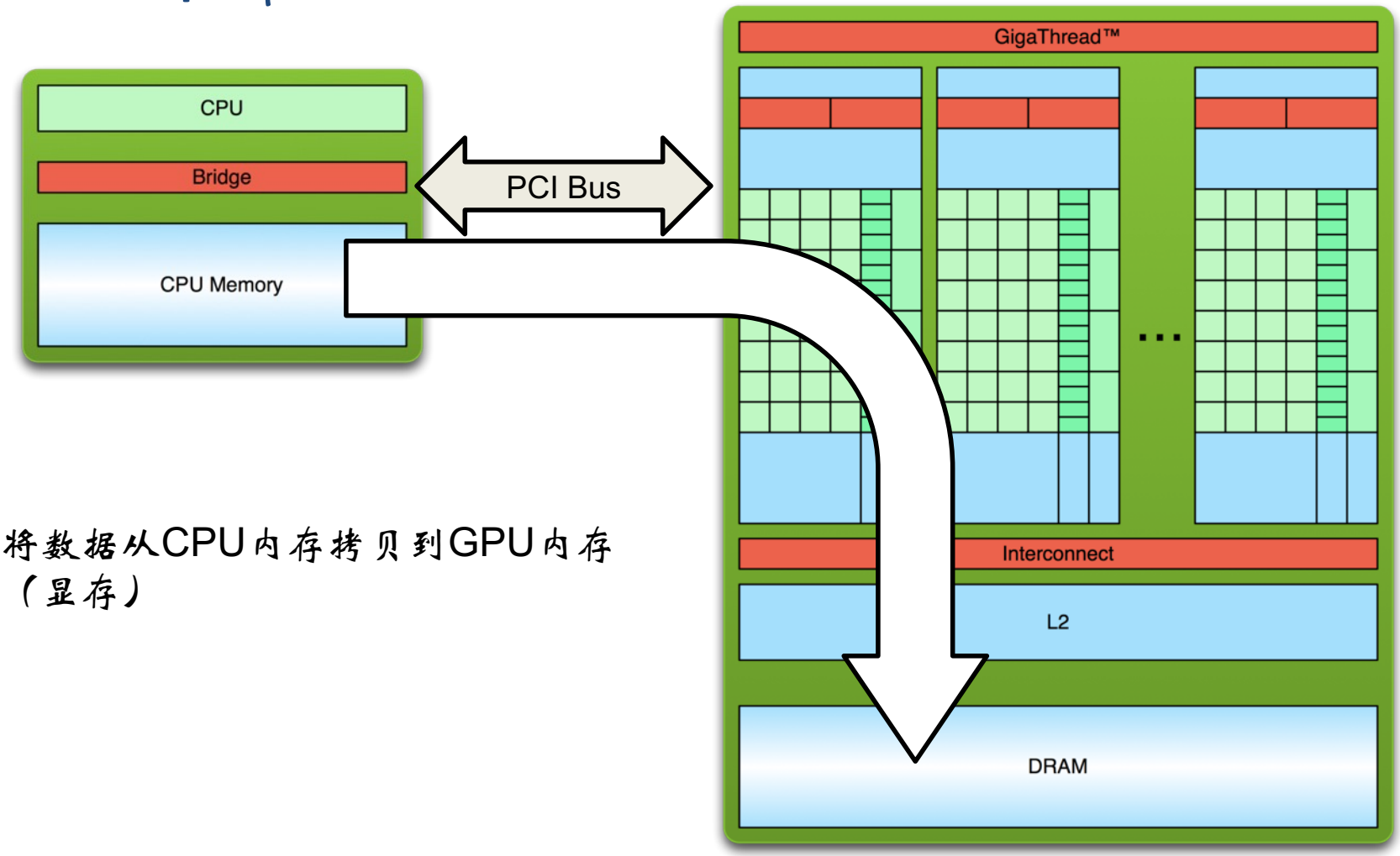
- 调用时必须给出线程配置方式

```
__global__ void KernelFunc(...);
dim3 DimGrid(100, 50); // 5000 thread blocks
dim3 DimBlock(4, 8, 8); // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

GPU异构计算编程模型

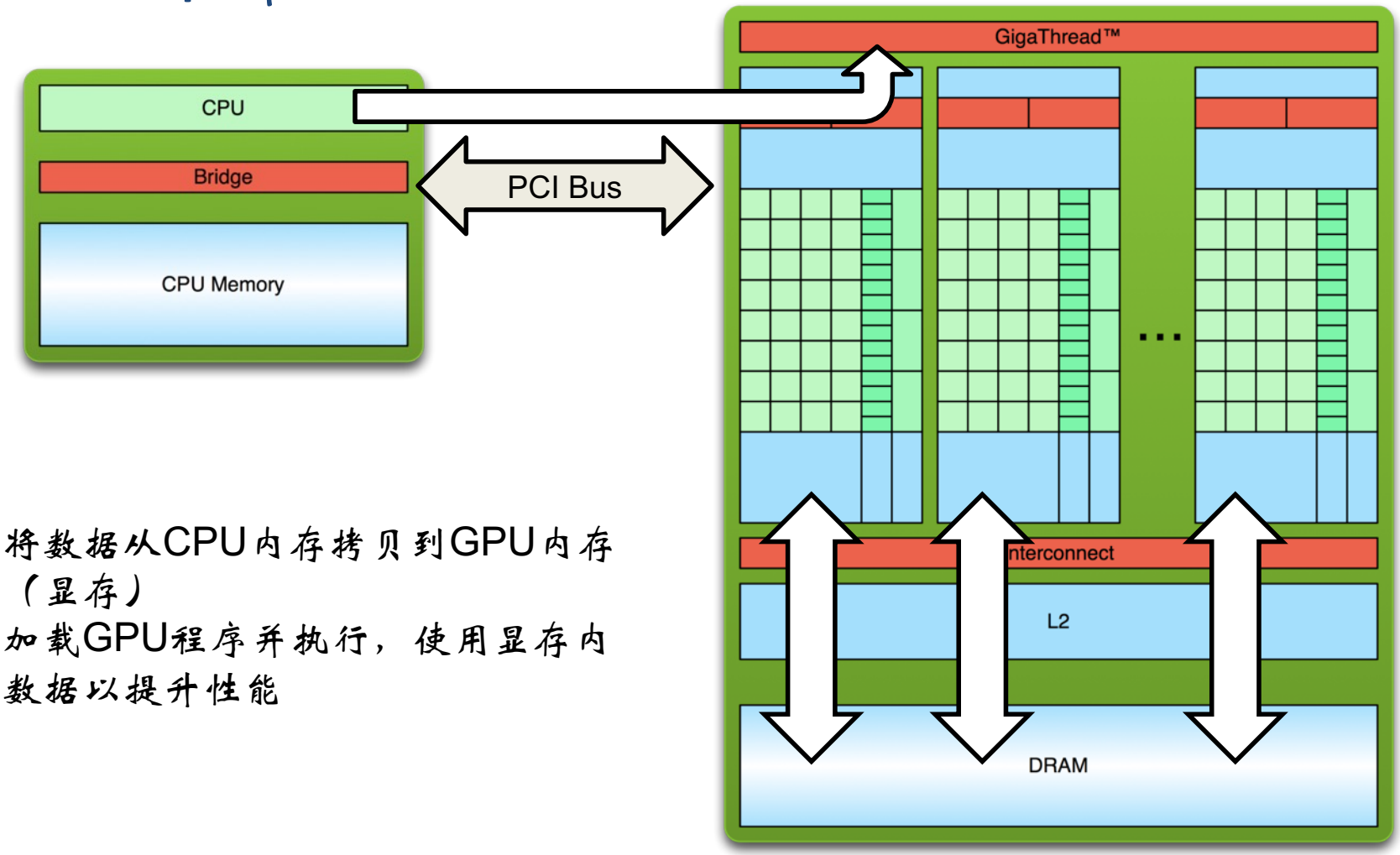


GPU计算过程



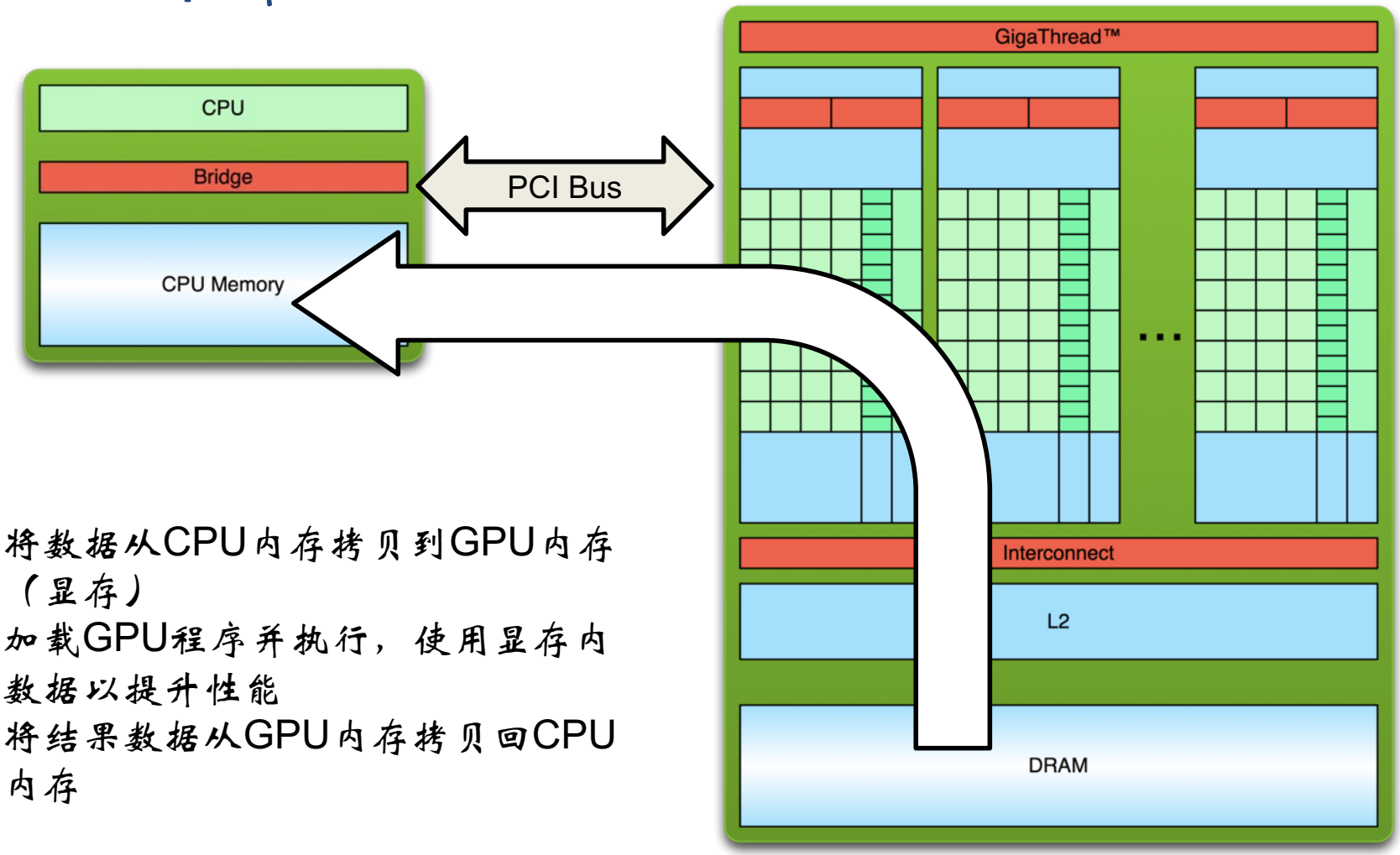
1. 将数据从CPU内存拷贝到GPU内存（显存）

GPU计算过程



1. 将数据从CPU内存拷贝到GPU内存（显存）
2. 加载GPU程序并执行，使用显存内数据以提升性能

GPU计算过程



1. 将数据从CPU内存拷贝到GPU内存（显存）
2. 加载GPU程序并执行，使用显存内数据以提升性能
3. 将结果数据从GPU内存拷贝回CPU内存

Hello World!

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

编译运行:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
```

- host上运行的是标准C语言代码
- 使用NVIDIA编译器 (nvcc)可以编译没有device代码的程序

Hello World! with Device Code

```
__global__ void mykernel(void){
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

- __global__ 表示此函数在device上执行，自host调用。
- mykernel() 在此没有进行任何计算。

输出：

```
$ nvcc hello.cu
$ a.out
Hello World!
$
```

Outline

- CUDA编程模型概述
  - NIVIDIA GPU体系架构
  - CUDA软件开发环境
  - CUDA异构计算
- **CUDA程序设计**
  - **线程创建与管理**
  - 线程调度与执行

CUDA程序基本结构

```
int main(int argc, char** argv)
{
  // Allocate memory on the host for input data - malloc()
  // Initialize input data from file, user input, etc.

  // Allocate memory on the device - cudaMalloc()
  // Send input data to the device - cudaMemcpy()

  // Set up grid and block dimensions - dim3 variables
  // Invoke the kernel on the device (GPU) -
  kernelName<<<gridSize, blockSize>>>(input_params);

  // Copy results from device to host - cudaMemcpy()
  // Free up device memory - cudaFree()

  // Print results at the host, because device can't.
  // printf() from kernel only works in emulation mode
}
```

GPU求和计算：kernel

- 两个整数求和的kernel

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- add() 在device运行，所以 a, b 和 c 需要指向device内存
- 所以需要在GPU上分配内存空间

GPU求和计算：main()

```
int main(void) {
        int a, b, c;                    // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c, 分配内存
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values, 设置输入数据
        a = 2;
        b = 7;
```

GPU求和计算：main()

```
    // Copy inputs to device, 拷贝数据到 GPU 的显存
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU, 执行 kernel
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host, 数据拷贝回 host 内存
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup, 释放内存
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

GPU并行计算

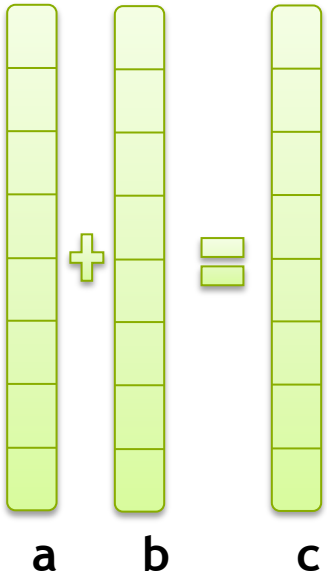
- GPU的优势是大规模并行
  - 如何在device上并行运行代码？

```
add<<< 1, 1 >>>();
```

```
add<<< N, 1 >>>();
```

- 这样，add() 不再是只执行了一次，而是并行执行了N次

GPU向量加



- 并行执行 add() 可以实现向量加法
- 定义: add() 的每个调用定义为一个 block
  - 一组 blocks 定义为 grid
  - 各个调用通过 blockIdx.x 标识

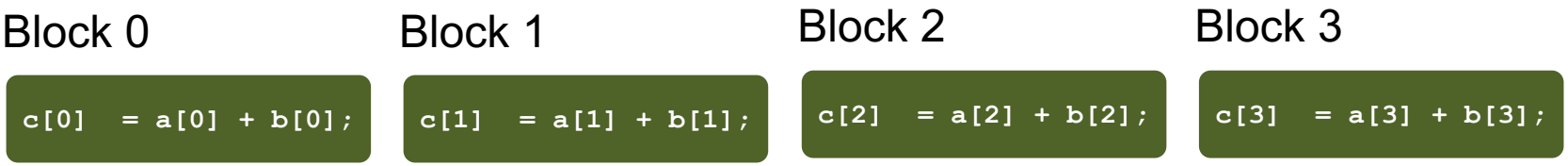
```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- 使用 blockIdx.x 作为数组的索引，每个block处理各自的数据

GPU向量加

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- 在device中，各个block可以并行执行：



GPU向量加：main()

```
#define N 512
int main(void) {
    int *a  *b  *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

GPU向量加：main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

CUDA 线程

- 定义: 每个block 可以拆分为并行的 threads

- 将add() 改为使用并行 *threads*，不再使用并行 *blocks*

```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- 使用 `threadIdx.x` 替换 `blockIdx.x`

- **main()...中只需要修改一处**

使用Threads进行向量加：main()

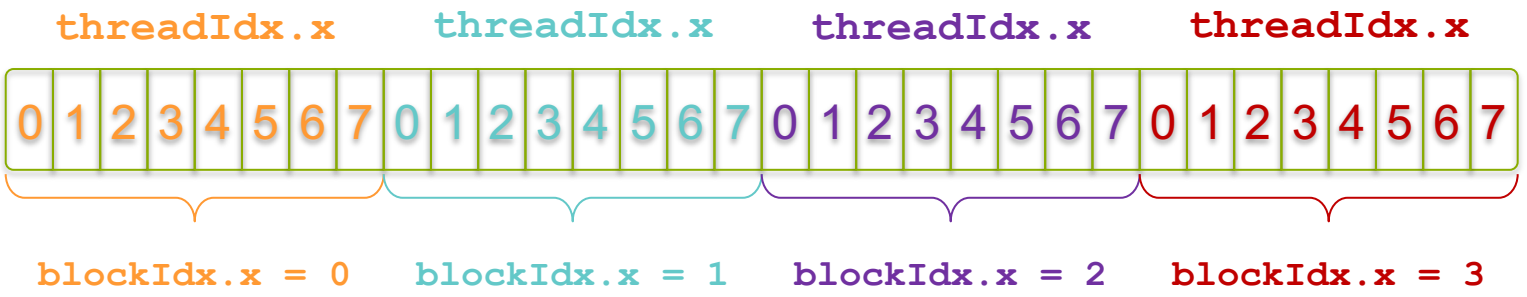
```
#define N 512
int main(void) {
    int *a, *b, *c;                         // host copies of a, b, c
    int *d_a, *d_b, *d_c;           // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

使用Threads进行向量加：main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Blocks + Threads的索引数组

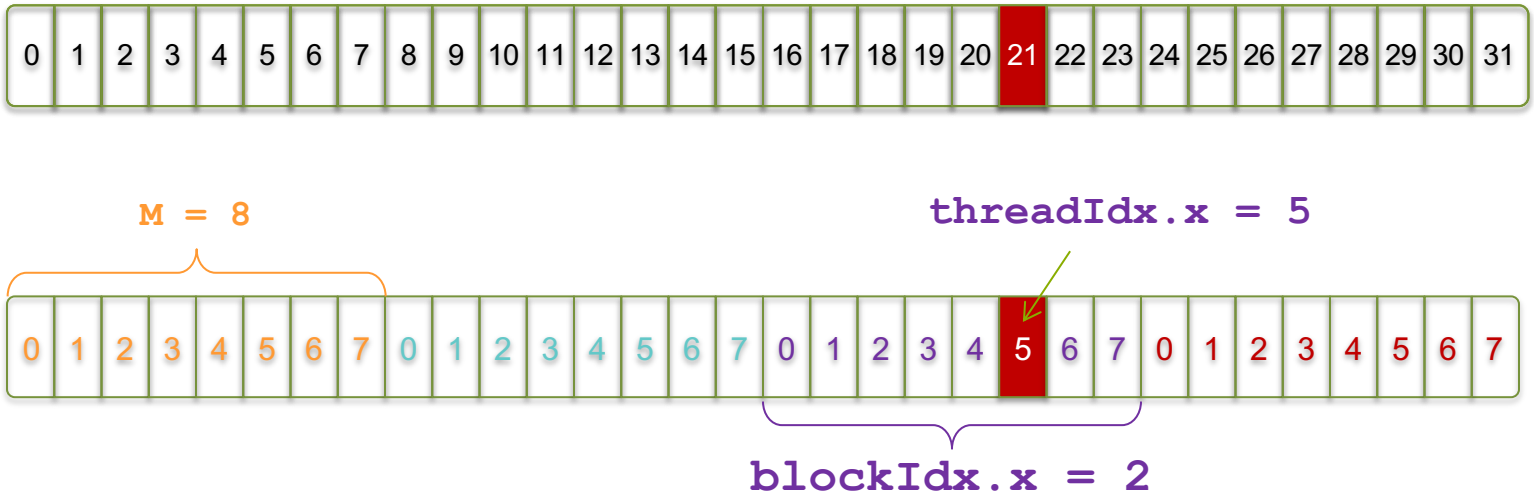
- 比单独使用`blockIdx.x` 和 `threadIdx.x`更复杂
  - 每个线程需要全局唯一标识



- M为每个block中的thread数量，则：

```
int index = threadIdx.x + blockIdx.x * M;
```

索引数组示例




```
int index = threadIdx.x + blockIdx.x * M;
          =       5       +      2       * 8;
          = 21;
```

使用Blocks + Threads进行向量加

- 使用内置变量 blockDim.x （每个block的thread数量）

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- 组合使用并行 thread 和并行 block 的add()

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- main()函数也需要修改。

使用Blocks + Threads进行向量加：main()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                         // host copies of a, b, c
    int *d_a, *d_b, *d_c;           // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

使用Blocks + Threads进行向量加：main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

处理任意大小的向量

- 向量的长度并不能保证恰好是 blockDim.x 的整数倍

- 避免数组访问越界（最后一个block可能会发生）：

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- kernel调用方式修订（保证block数量足够）：

```
add<<<(N + M-1) / M, M>>>(d_a, d_b, d_c, N);
```

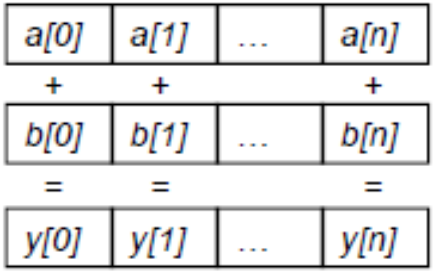
并行线程组织

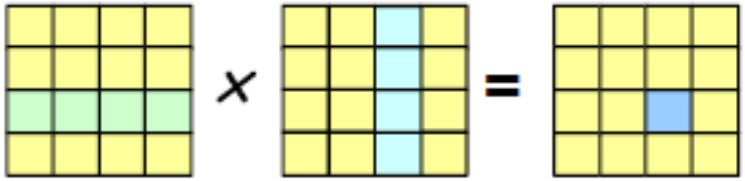
- 并行性的维度
  - 一维

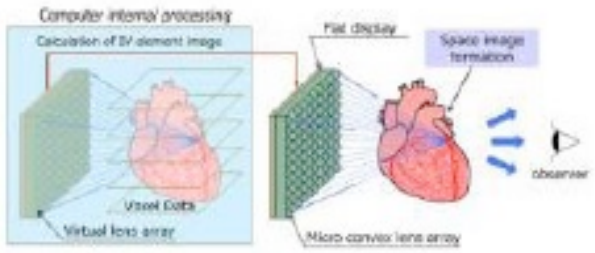
    y = a + b
  - 二维

    P = M × N
  - 三维

    CT or MRI

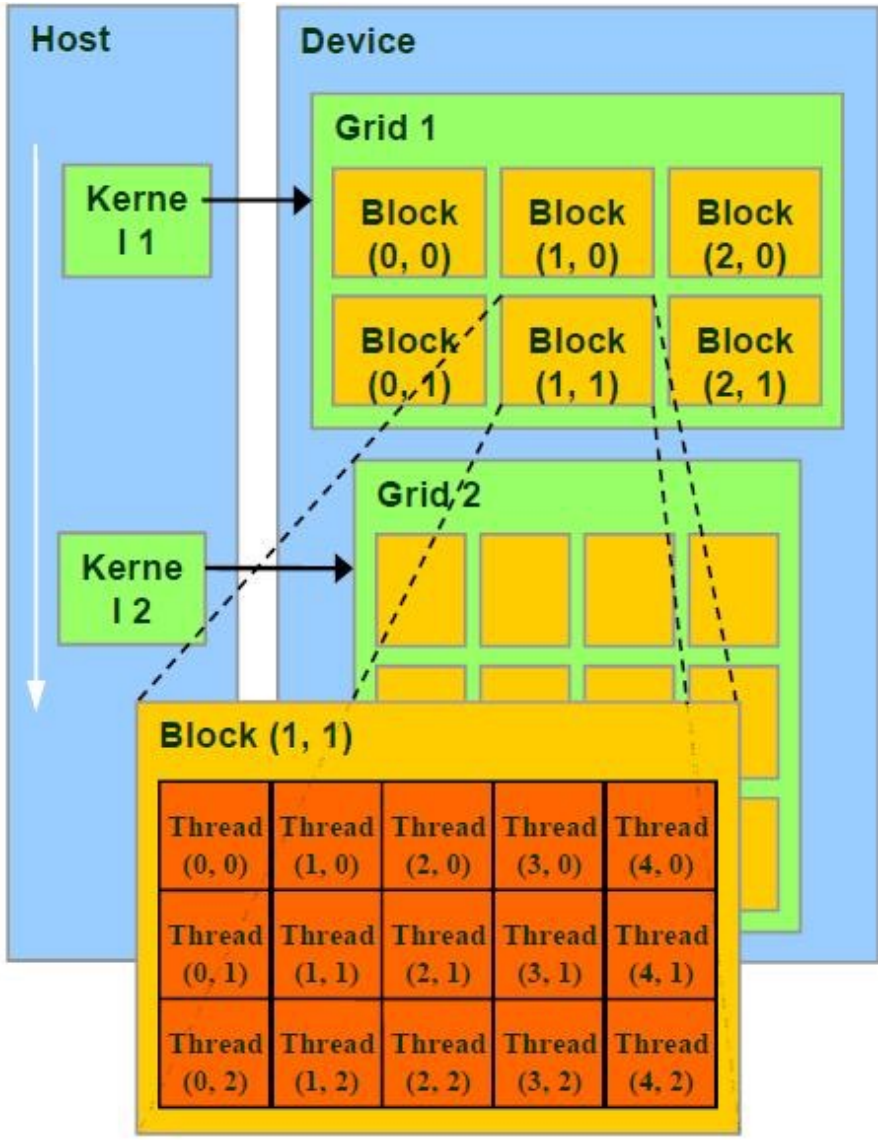






CUDA线程组织结构

- Thread: 并行的基本单位
- Thread block: 互相合作的线程组
  - 每个block包含多个线程
  - 可同步
  - 高效数据交换
- 每个grid对应一组block
  - 共享全局内存
- 每个kernel对应一个grid



内置变量

- grid 与 block 的维度，block 与thread的坐标
  - gridDim
    - dim3类型，grid的维度
  - blockIdx
    - uint3类型，block在grid中的位置坐标
  - blockDim
    - dim3类型，block的维度
  - threadIdx
    - uint3类型，thread在block中的位置坐标
  - warpSize
    - int类型，warp的大小是固定的（32）

参数与约束

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	255[1]
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/ 48 KB	96 KB	64 KB	Configurable up to 96 KB

CUDA 线程层次（示例1，1个 block）

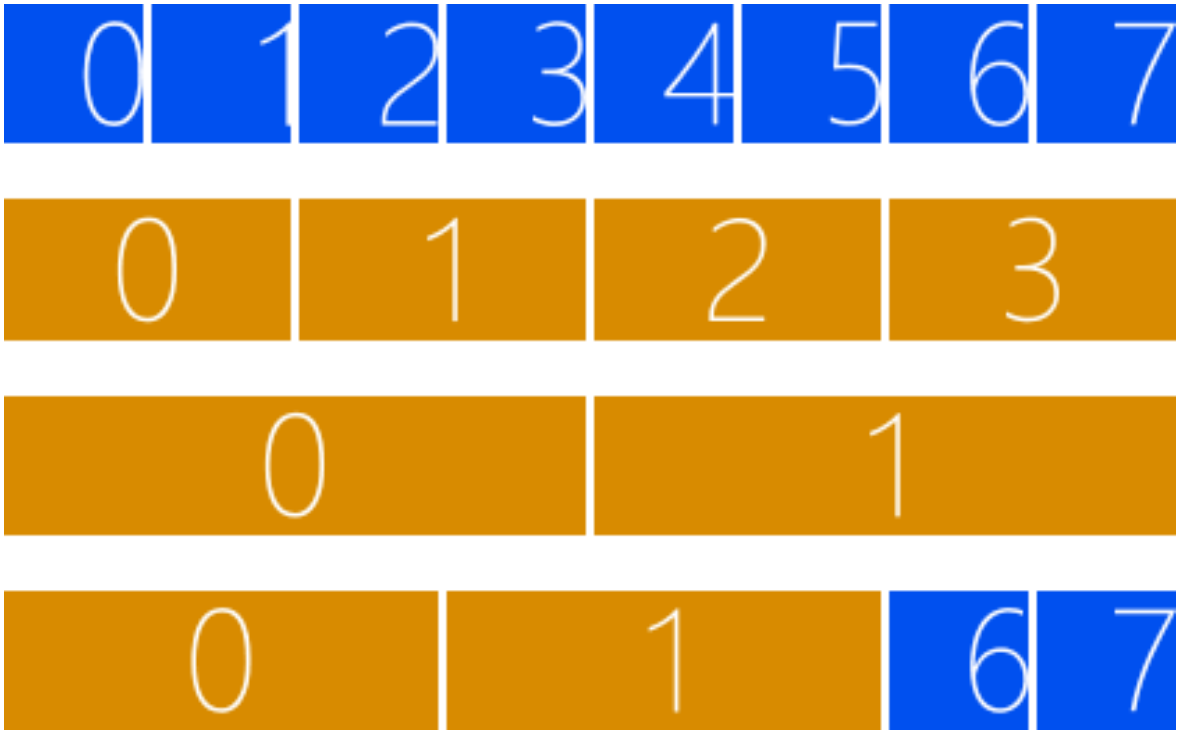
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    …
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    …
}
```

CUDA 线程层次（示例2，多个 block）

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
   …
    // 先定义每个 block 中线程“形状”，再计算 block 的“形状”
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
   …
}
```

线程分组

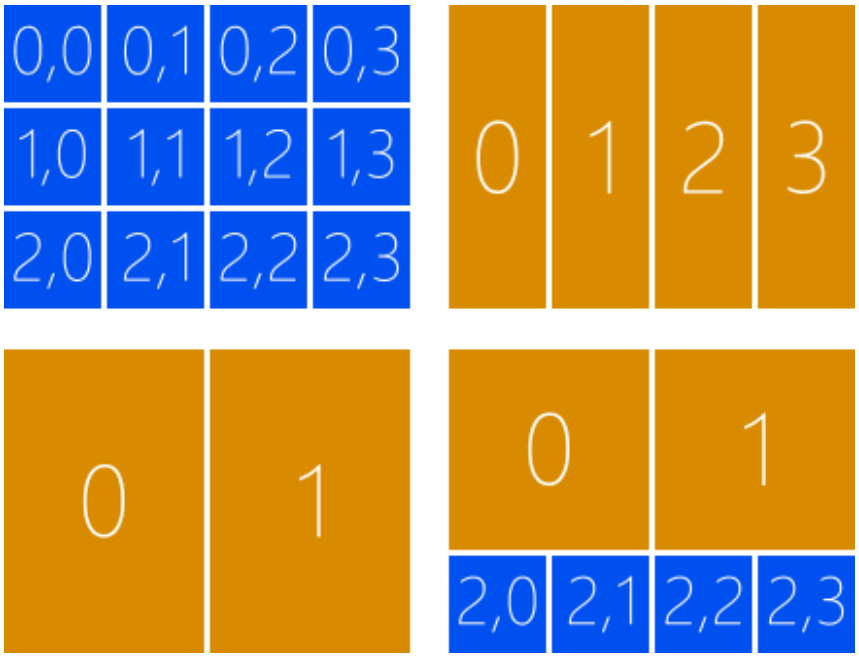
- 对应的维度必须能被整除，分组的大小不能超过1024。
- 假设GPU线程是一维的，共8个，则可以选择每2个GPU线程为1组或者每4个GPU线程为1组，但不能选择每3个GPU线程为1组，因为剩下2个GPU线程不足1组。



线程分组

- 假设GPU线程是二维的，3 x 4，共12个，则可以选择3 x 1或者3 x 2作为分组方案，但不能选择2 x 2作为分组的规格，因为剩下的4个GPU线程虽然满足分组的大小，分组形状不同。每个分组必须完全相同，包括大小和形状。

- 假设GPU线程是640 x 480，那么16 x 48、32 x 16和32 x 32都可以选择，它们分别产生40 x 10、20 x 30和20 x 15个分组，但32 x 48不能选择，因为大小超过1024。

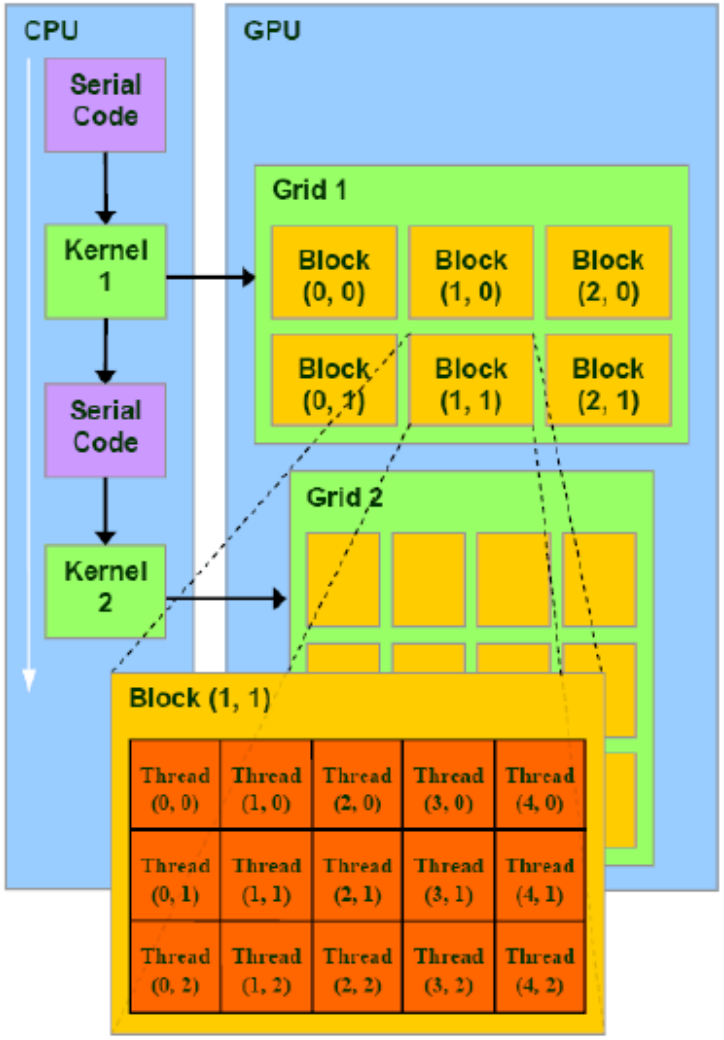


Outline

- CUDA编程模型概述
  - NIVIDIA GPU体系架构
  - CUDA软件开发环境
  - CUDA异构计算
- **CUDA程序设计**
  - 线程创建与管理
  - **线程调度与执行**

CUDA执行模型

- 调用核程序时CPU调用API将显卡端程序的二进制代码传到GPU
- grid运行在SPA上
- block运行在SM上
- thread运行在SP上



grid block thread

- Kernel不是一个完整的程序，而只是其中的一个关键并行计算步
- Kernel以一个网格(Grid)的形式执行，每个网格由若干个线程块（block）组成，每一个线程块又由最多512个线程(thread)组成。
- 一个grid最多可以有65535 * 65535个block
- 一个block总共最多可以有512个thread，在三个维度上的最大值分别为512, 512和64

grid block thread

- grid之间通过global memory交换数据
- block之间不能相互通信,只能通过global memory共享数据,不要让多个block写同一区段内容（不保证数据一致性和顺序一致性）
- 同一block内的thread可以通过shared memory和同步实现通信
- block间粗粒度并行，block内thread细粒度并行

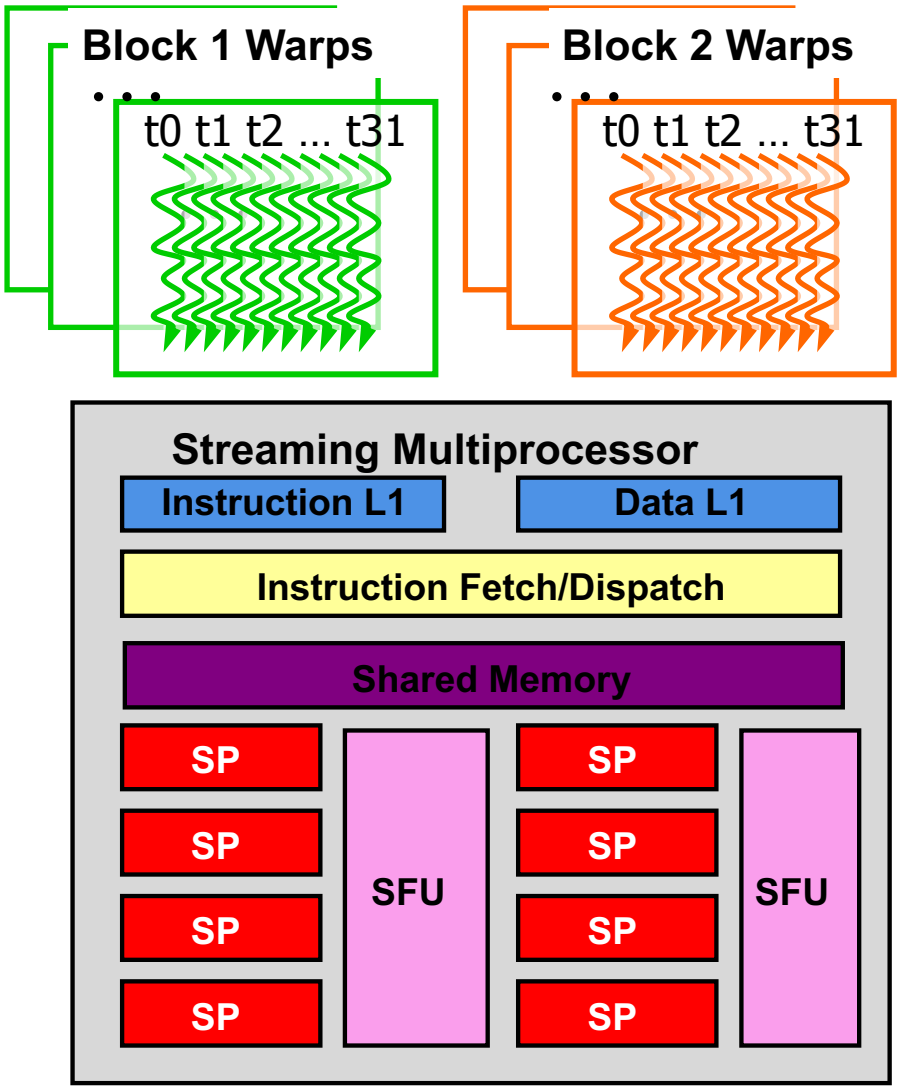
warp

- Warp是硬件特性带来的概念，在CUDA C语言中是透明的（除vote函数），但应用中不能忽略
- 一个warp中有32个线程，这是因为SM中有8个SP，执行一条指令的延迟是4个周期，使用了流水线技术
- 一个half warp中有16个线程，这是因为执行单元的频率是其他单元的两倍，每两个周期才进行一次数据传输

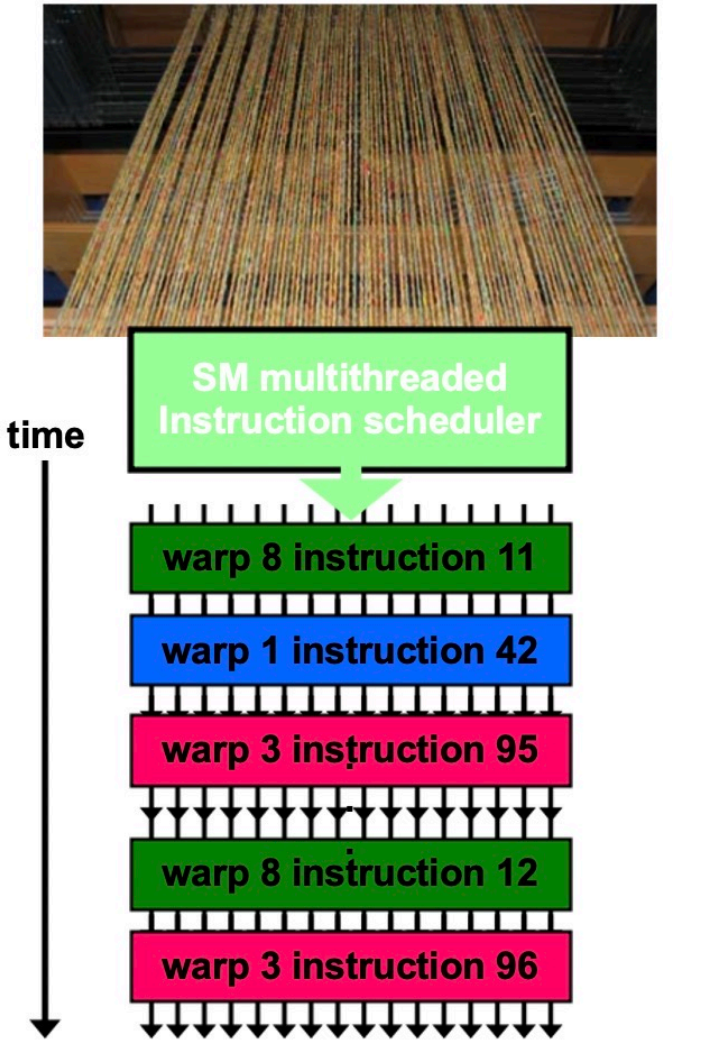
SIMT编程模型

- SIMT是对SIMD（Single Instruction, Multiple Data，单指令多数据）的一种变形。
- 两者的区别在于：SIMD的向量宽度是显式的，固定的，数据必须打包成向量才能进行处理；而SIMT中，执行宽度则完全由硬件自动处理了。（每个block中的thread数量不一定是32）
- 而SIMT中的warp中的每个线程的寄存器都是私有的，它们只能通过shared memory来进行通信。

Thread Scheduling/Execution



SIMT Wrap Scheduling



分支性能

- 与现代的微处理器不同，Nvidia的SM没有预测执行机制-没有分支预测单元(Branch Predicator)。
- 在需要分支时，只有当warp中所有的线程都计算出各自的分支的地址，并且完成取指以后，warp才能继续往下执行。
- 如果一个warp内需要执行N个分支，那么SM就需要把每一个分支的指令发射到每一个SP上，再由SP根据线程的逻辑决定需不需要执行。这是一个串行过程，此时SIMT完成分支的时间是多个分支时间之和。