中山大學
SUN YAT-SEN UNIVERSITY

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# 并行程序设计与算法

## CUDA入门


陶钧

taoj23@mail.sysu.edu.cn

中山大学 计算机学院

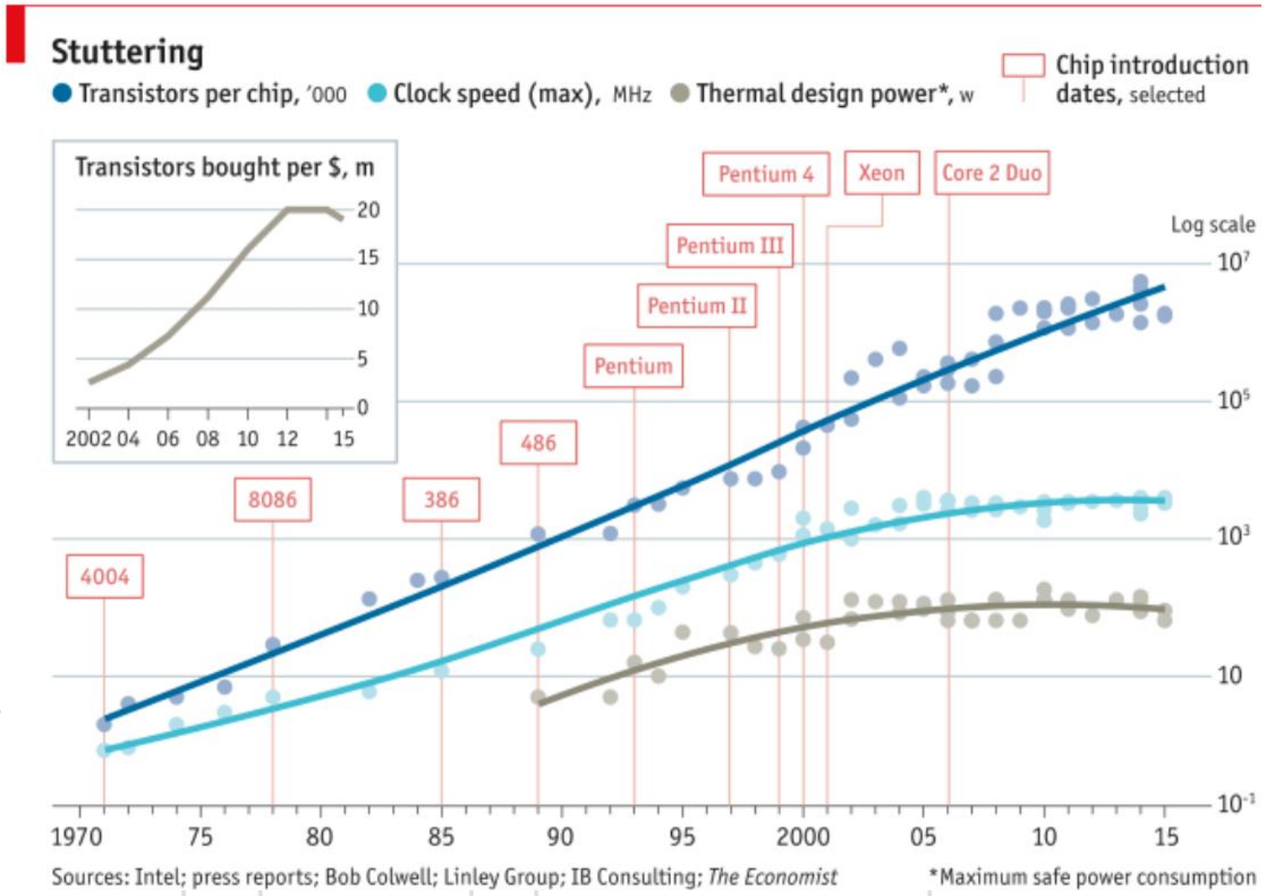
国家超级计算广州中心

# 课程概要

- 要点回顾
- CPU vs GPU
- NVIDIA GPU架构
- CUDA编程模型
- CUDA编程举例

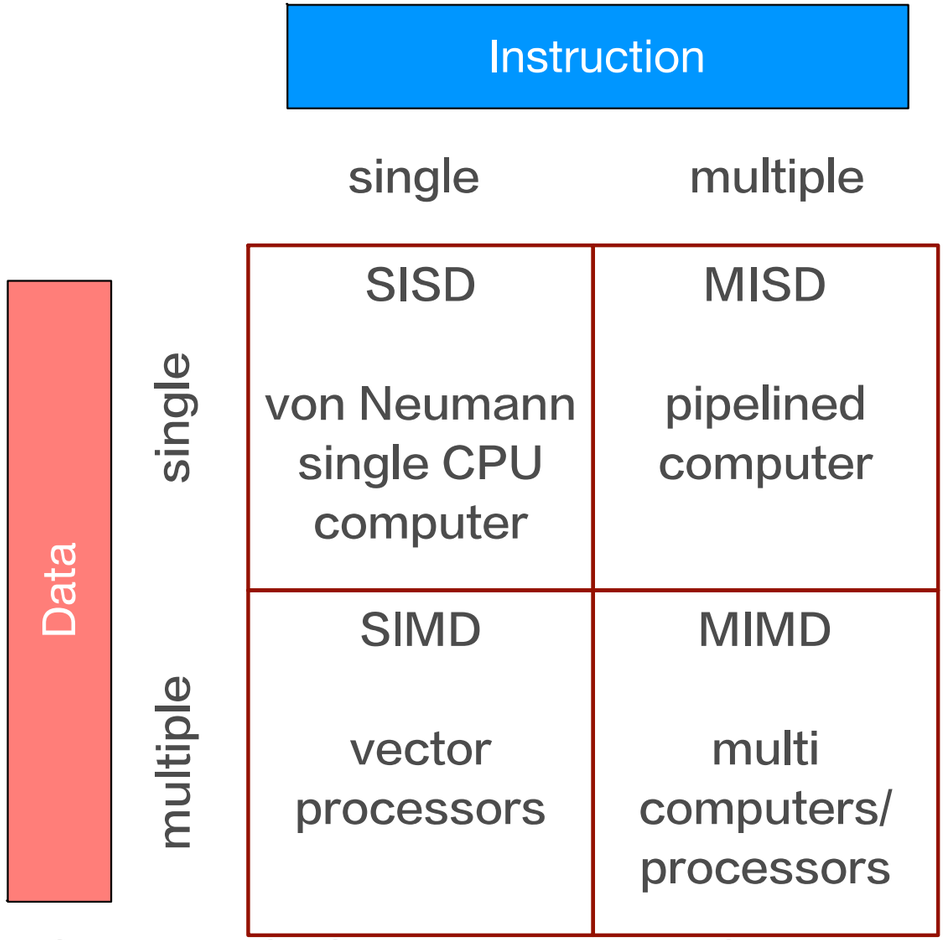
# 相关要点回顾

- 频率提升的时代已经基本告一段落
  - 晶体管数目≠频率
  - 缓存增大
  - 模块数量增加
    - 流水线
    - 多发射



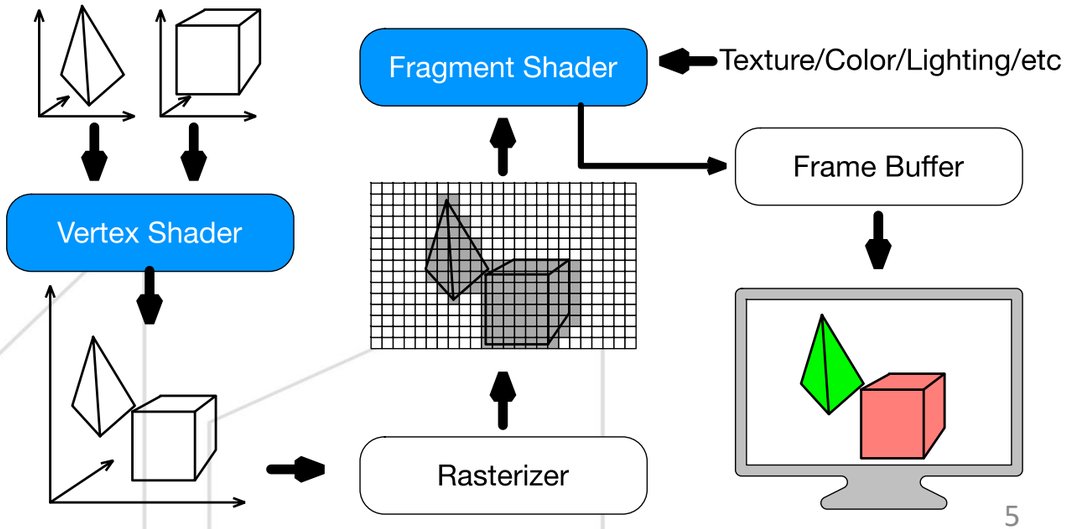
# 相关要点回顾

- 并行计算机架构：Flynn分类法
  - 根据指令和数据进入CPU的方式分类
    - SISD, SIMD, MISD, MIMD
    - 拓展分类：SPMD, MPMD
  - SIMD与GPU最为相似
    - 阵列计算机
    - 向量计算机



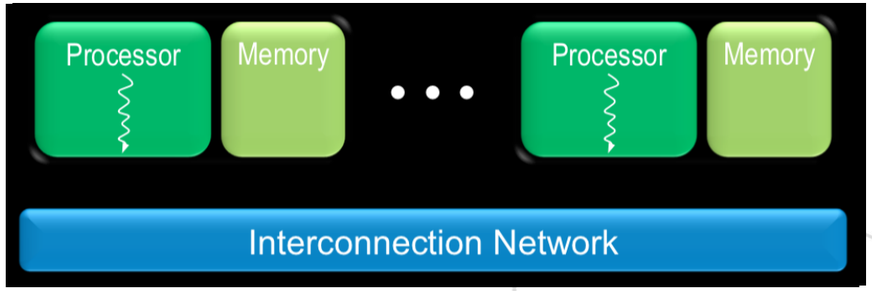
# 相关要点回顾

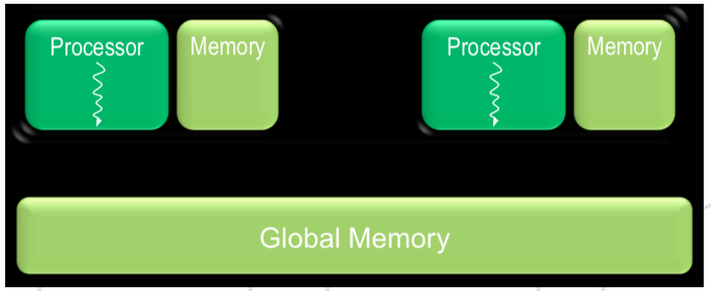
- 使用GPU进行并行计算
  - Vertex shader 和 fragment shader 可由用户自行通过shading language编程
  - 把封装数据封装成图形绘制库所需的数据形式传入
  - 绘制过程中通过修改shader执行用户编写的程序
    - 通常是fragment shader
  - 将结果绘制到texture中传出
  - 需要很强的图形学编程能力

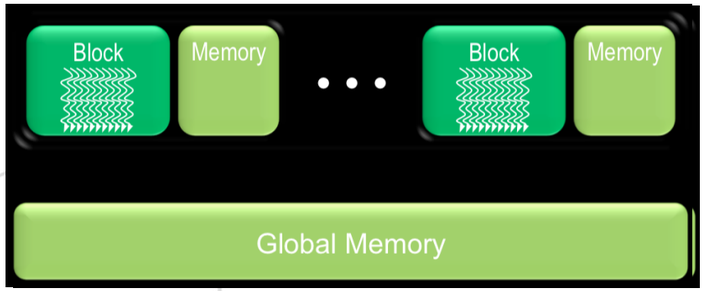


# 相关要点回顾

- 分布式 vs 共享内存架构
  - 分布式：MPI
    - 多进程，进程完全独立，创建销毁开销高，通过消息传递通信
  - 共享内存：Pthreads/OpenMP
    - 多CPU线程，线程完全独立，创建销毁开销较低，通过共享内存通信
  - 共享内存：CUDA
    - 多GPU线程，线程并不完全独立，创建销毁开销极低，多级存储



# 课程概要

- 要点回顾
- CPU vs GPU
- NVIDIA GPU架构
- CUDA编程模型
- CUDA编程举例

# CPU vs GPU

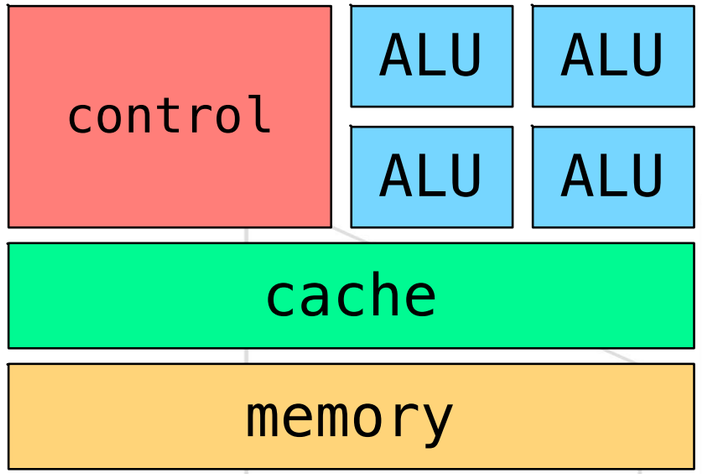
- 性能
  - 延迟：一个操作从开始到完成所需的时间（ms）
  - 带宽：单位时间内可处理的数据量（MB/s, GB/s）
  - 吞吐量：单位时间内成功可处理的运算量（gflops）
- CPU
  - 目标：降低延迟
  - 提高串行代码性能
  - 更适用于复杂单任务

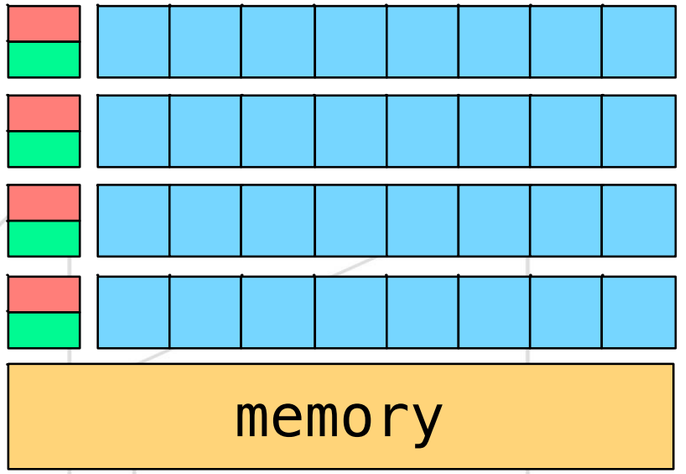
- GPU
  - 目标：提高吞吐量
  - 大规模并行架构
  - 更适用于大量相似任务

# CPU vs GPU

- CPU
  - 大缓存
    - 掩盖较长的存储器延迟
  - 强大的运算器
    - 降低运算延迟
  - 复杂的控制机制
    - 分支预测等
  - 线程上下文切换开销大
    - 降低一个或两个线程运行延迟

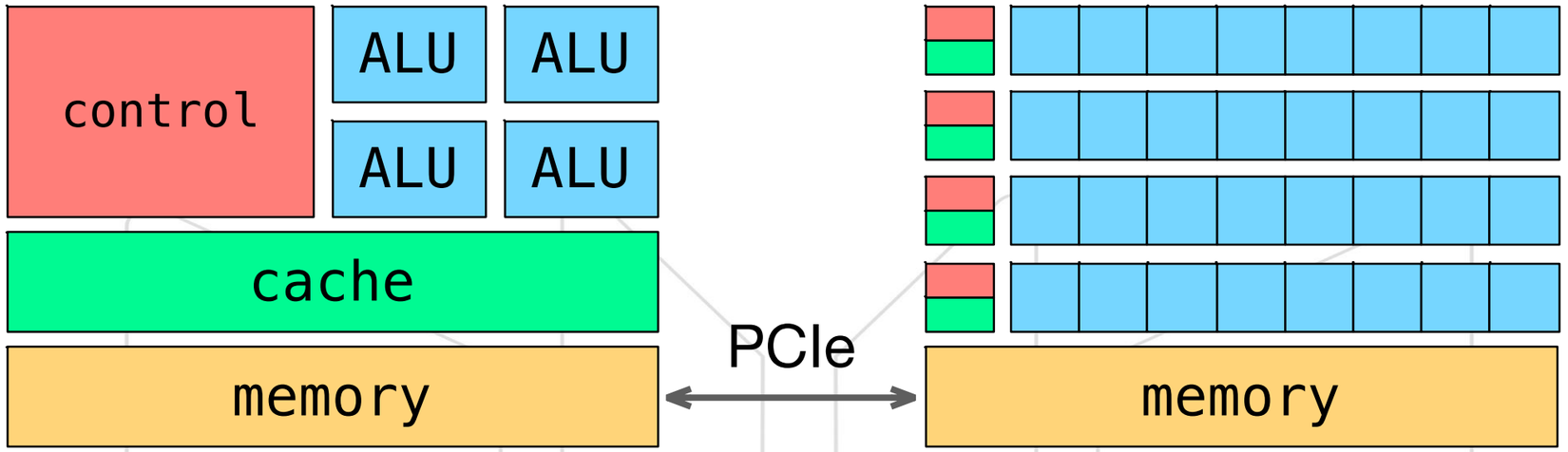
- GPU
  - 小缓存
    - 但通过更快的存储提高吞吐量
  - 更节能的运算器
    - 延迟长但总吞吐量更大
  - 简单的控制流机制
    - 无分支预测
  - 线程高度轻量级
    - 大量并发



# 异构架构

- CPU+GPU
  - 利用CPU处理复杂控制流
  - 利用GPU处理大规模运算
  - CPU与GPU之间通过PCIe总线通信
    - 新显卡支持通过NVLink直接连接（提供5-12倍于PCIe 3.0总线带宽）

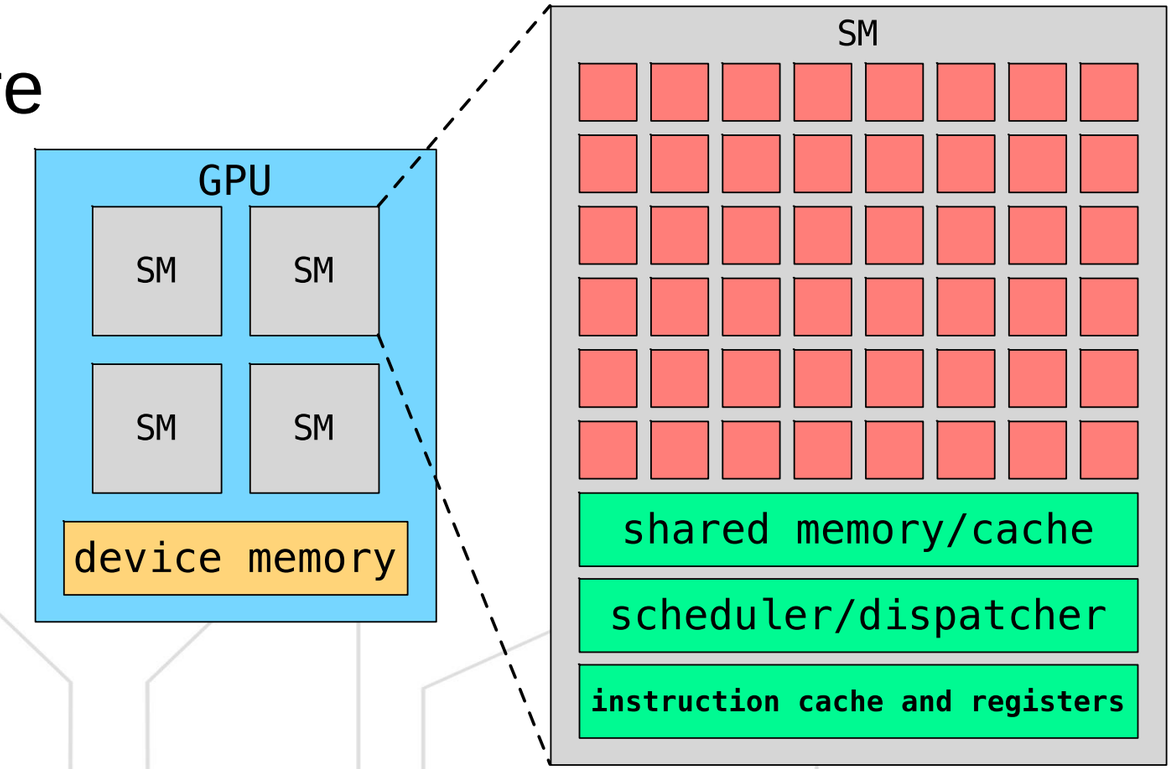


# 课程概要

- 要点回顾
- CPU vs GPU
- NVIDIA GPU架构
- CUDA编程模型
- CUDA编程举例

# NVIDIA GPU架构

- 2级架构
  - 每个GPU拥有多个Streaming Multiprocessor (SM)
    - 具体数目及设计因产品而异
    - SM共用显存
  - 每个SM拥有多个CUDA core
    - 数目因产品而异
    - Core共用调度器和指令缓存



# NVIDIA GPU架构

- 2级架构下的执行模式：线程束（warp）
  - CUDA线程以32个为一组在GPU上执行
    - 线程束以单指令多线程的方式运行（SIMT）
      - 所有线程在不同数据上执行相同的指令
    - SIMT，SIMD，SMT (simultaneous multithreading)
      - 灵活度：SIMD < SIMT <SMT
      - 性能：SIMD > SIMT > SMT
      - SIMT与SIMD相比：多个状态寄存器，多个地址，独立的执行路径
        - » http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html
  - SM负责调度并执行线程束
    - 线程束调度时会产生上下文切换
    - 调度方式因架构而异

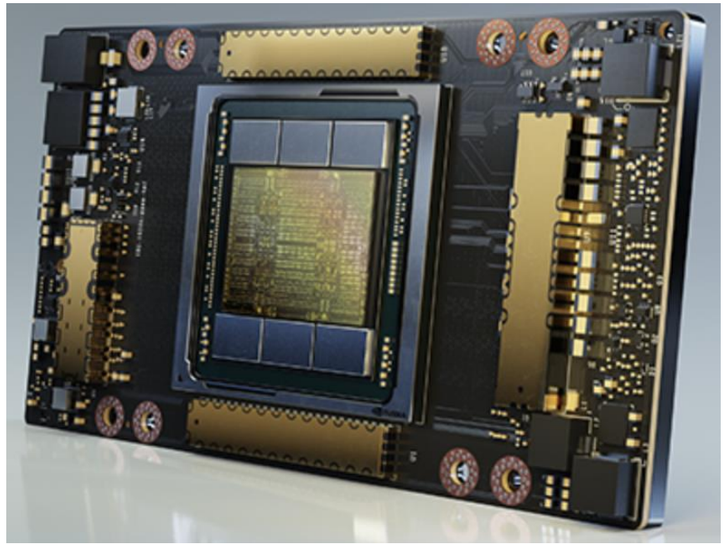
# NVIDIA GPU架构

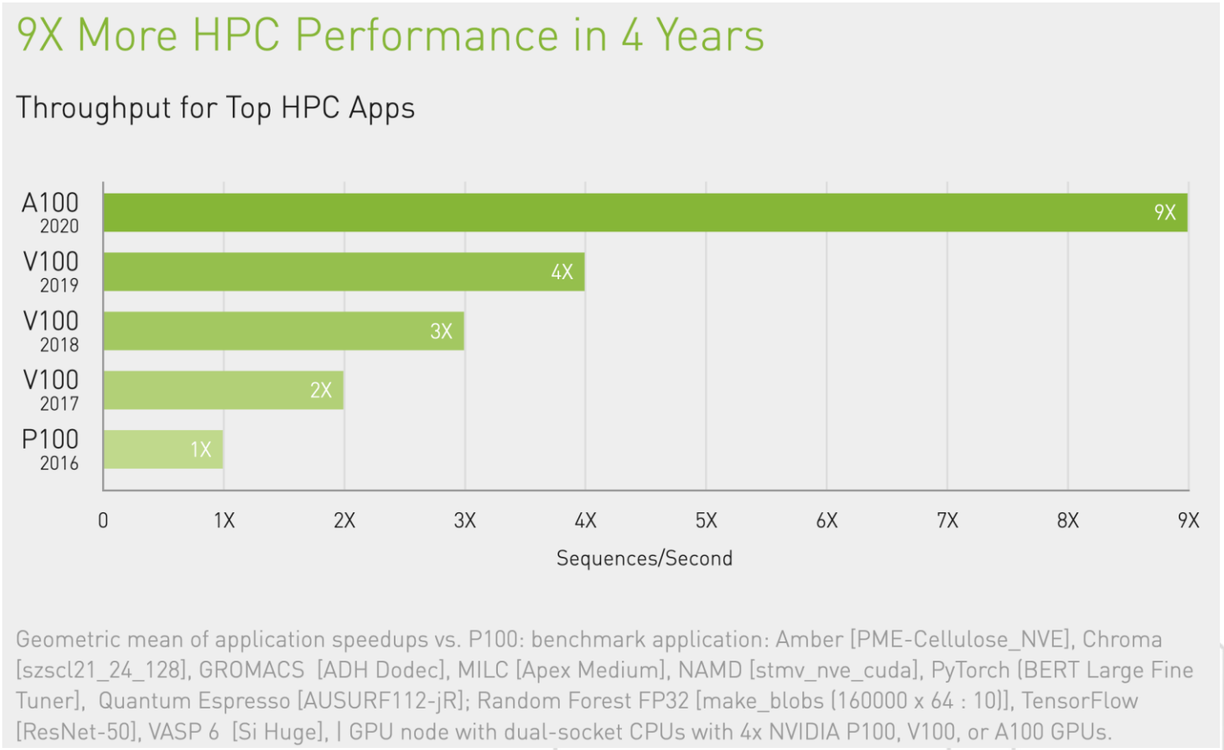
- Tegra
  - 面向移动和嵌入式设备
    - 平板电脑、手机等
- GeForce
  - 面向一般图形用户
    - 家用电脑，优化游戏性能
- Quadro
  - 面向专业绘图设计需求
    - 图形工作站，优化专业绘图软件性能
- Tesla
  - 面向大规模计算
    - 没有显示输出
    - 此前深度学习神器v100即属于此系列
  - 更强大的双精度运算能力
  - 更大的内存带宽

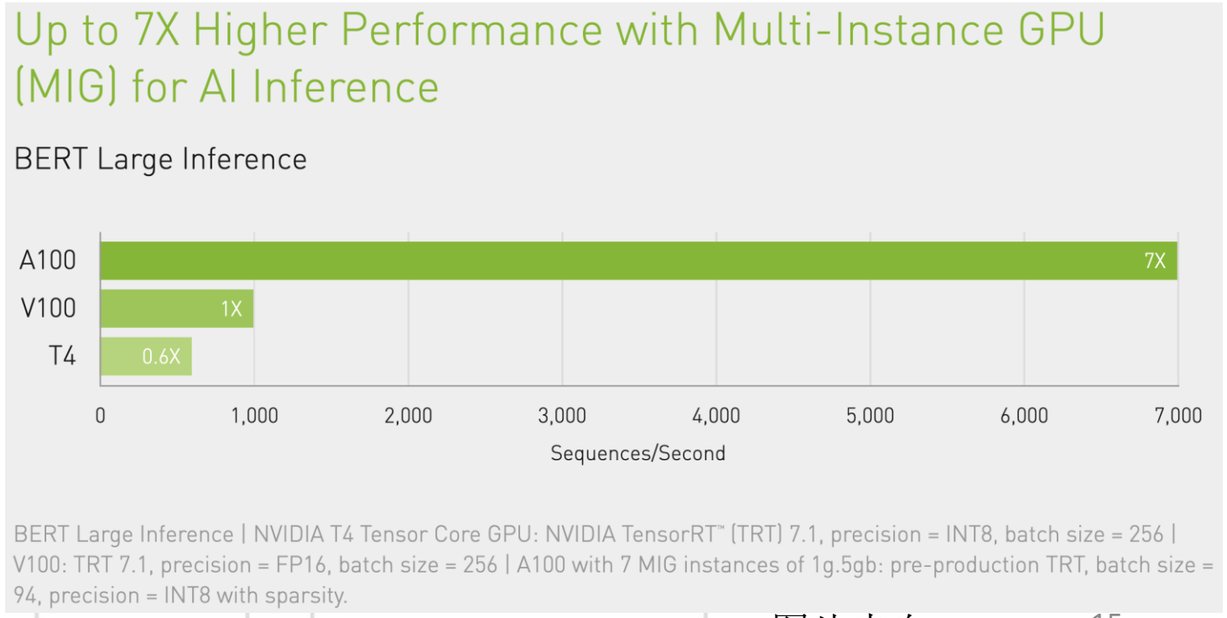



# NVIDIA GPU架构

- NVIDIA A100 Tensor Core GPU
  - “Engine of the NVIDIA data center platform”
  - “A100 can efficiently scale to thousands of GPUs”
  - “Multi-Instance GPU (MIG) technology allows multiple networks to operate simultaneously on a single A100 GPU for optimal utilization of compute resources.”





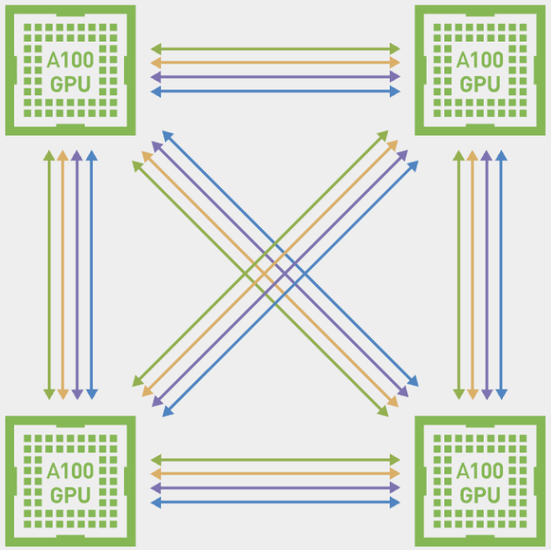




图片来自NVIDIA

# NVIDIA GPU架构

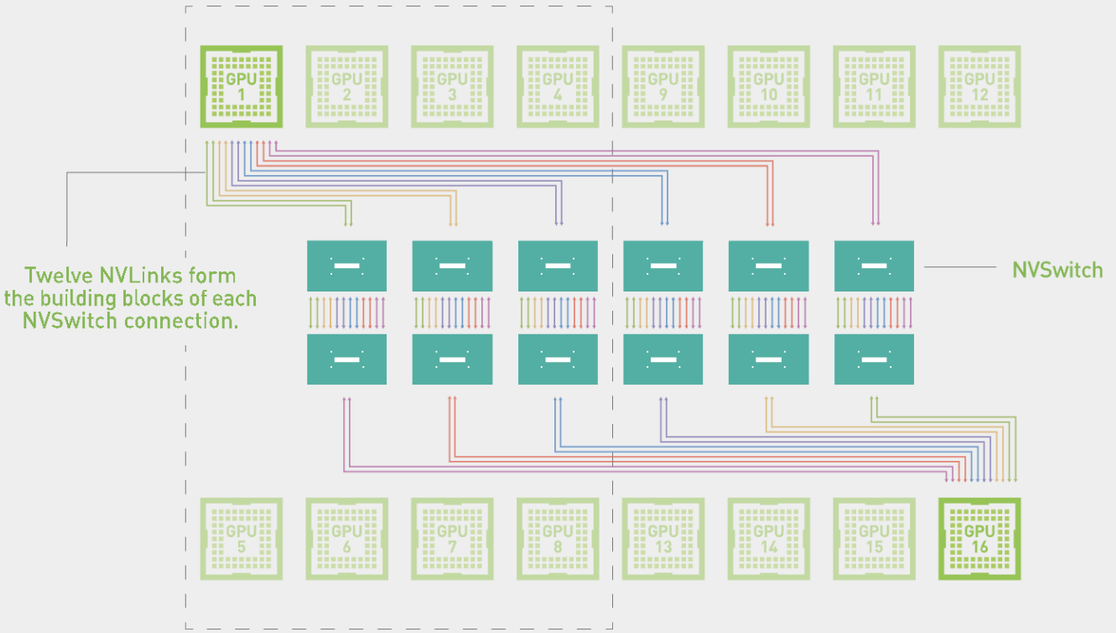
- NVSwitch 与NVIDIA HGX A100
  - NVSwitch: NVLink组成的全连接架构
  - HGX A100: 在一张主板上由NVSwitch连接多张A100计算卡
    - 4x/8x/16x-A100：人工智能也要进入军备竞赛阶段了？

NVIDIA NVLink




NVIDIA A100 with NVLink GPU-to-GPU connections

NVIDIA NVSwitch




The NVSwitch topology diagram shows the connection of two GPUs for simplicity. Eight or 16 GPUs connect all-to-all through NVSwitch in the same way.

图片来自NVIDIA

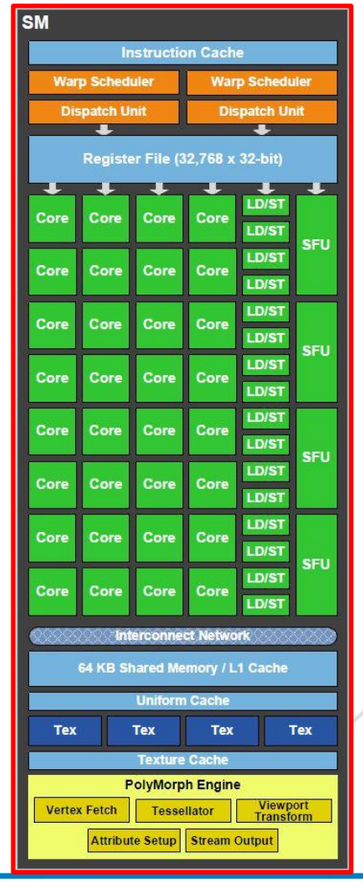
# NVIDIA GPU架构

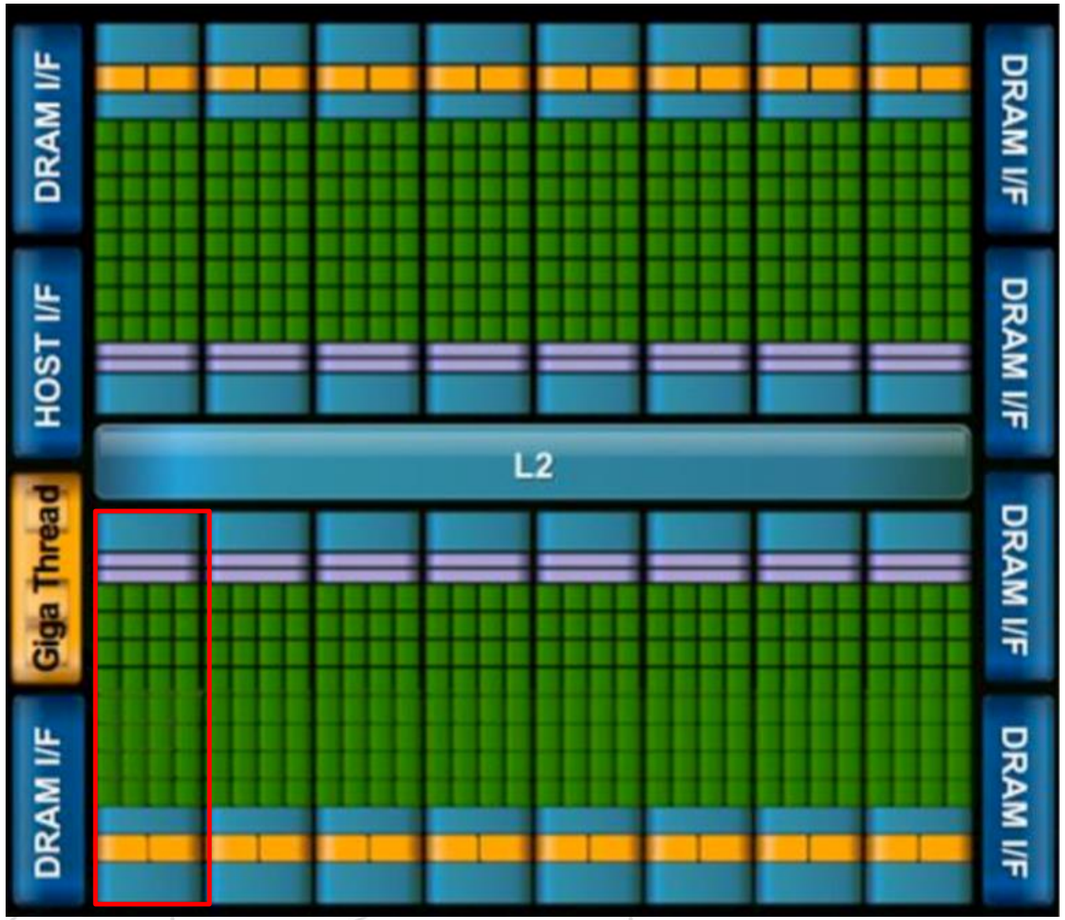
- Tesla (2006)→Fermi (2010)→Kepler (2012)→Maxwell (2014)→Pascal (2016)→Volta (2017)→Turing (2018)→Ampere(2020)
  - 更新速度加快
  - CUDA核心数量及内存大小增加
  - 峰值计算性能和内存带宽提高

| | “Kepler” K20 | “Kepler” K40 | “Maxwell” M40 | Pascal P100 | Volta V100 |
|---|---|---|---|---|---|
| CUDA cores | 2496 | 2880 | 3072 | 3584 | 5120 |
| Cores per SM | 192 | 192 | 128 | 64 | 64 |
| Single Precision | 3.52 Tflops | 4.29 Tflops | 7.0 Tflops | 9.5 Tflops | 15 Tflops |
| Double Precision | 1.17 TFlops | 1.43 Tflops | 0.21 Tflops | 4.7 Tflops | 7.5Tflops |
| Memory Bandwidth | 208 GB/s | 288 GB/s | 288 GB/s | 720 GB/s | 900 GB/s |
| Memory | 5 GB | 12 GB | 12 GB | 12/16 GB | 16 GB |

# NVIDIA GPU架构

- Fermi
  - 芯片划分为多个SM
  - 每个SM上32个CUDA core
  - 无缓存一致性
    - SM之间没有通信




图片来自NVIDIA

# NVIDIA GPU架构

- Kepler
  - 芯片划分为多个SMX
    - Streaming Multiprocessor Extreme
  - SMX上core数目大大增加（192）
  - 提供二级缓存一致性


图片来自NVIDIA

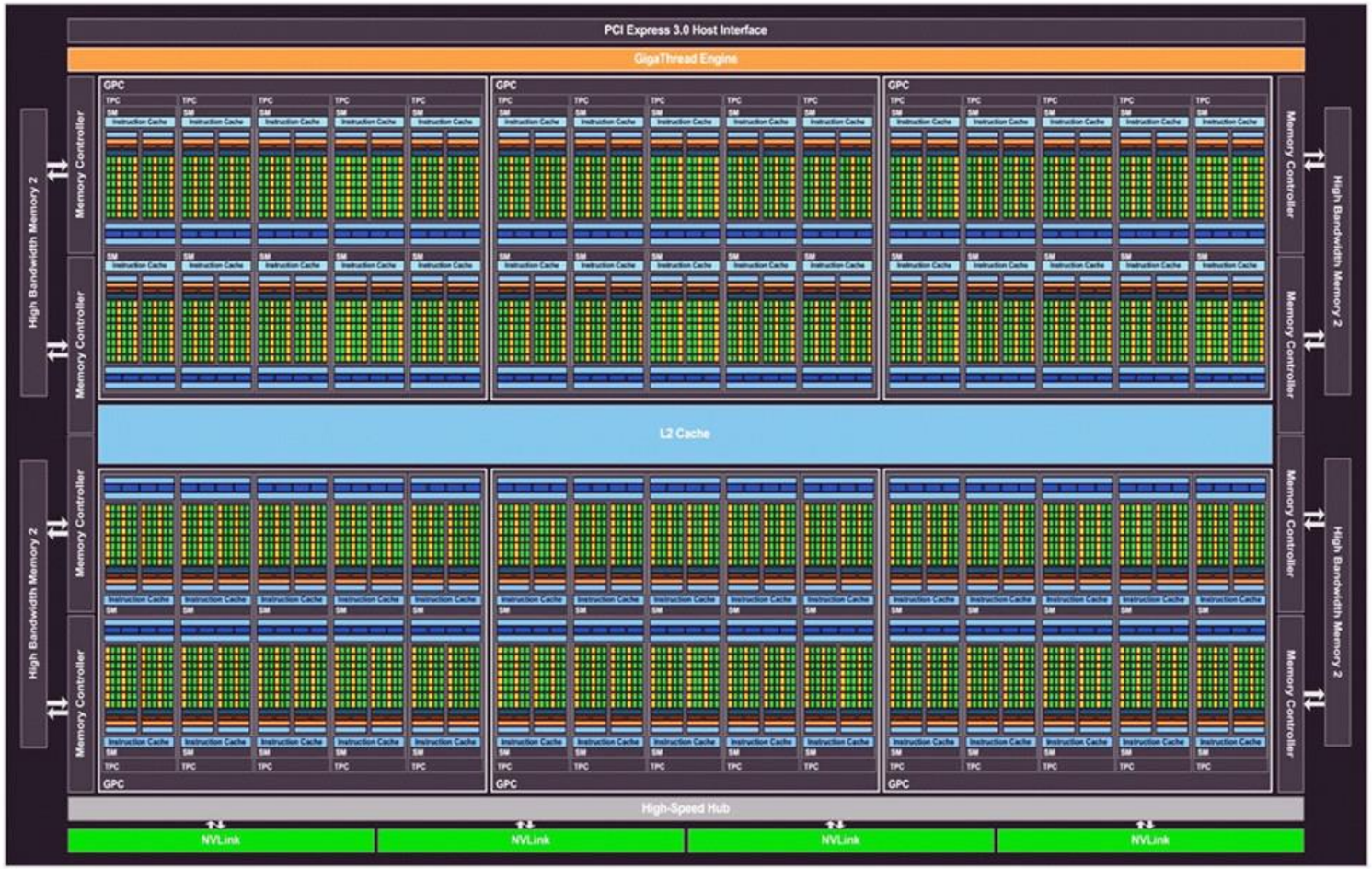
# NVIDIA GPU架构

- Maxwell
  - 芯片划分为四个GPC（quadrant）
  - 每个GPC上有多个Streaming Multiprocessor Module (SMM)
  - SMM减少了core数目但提高了能源效率
    - 192→128
    - 50%能耗
    - 90%运算能力


图片来自NVIDIA

# NVIDIA GPU架构

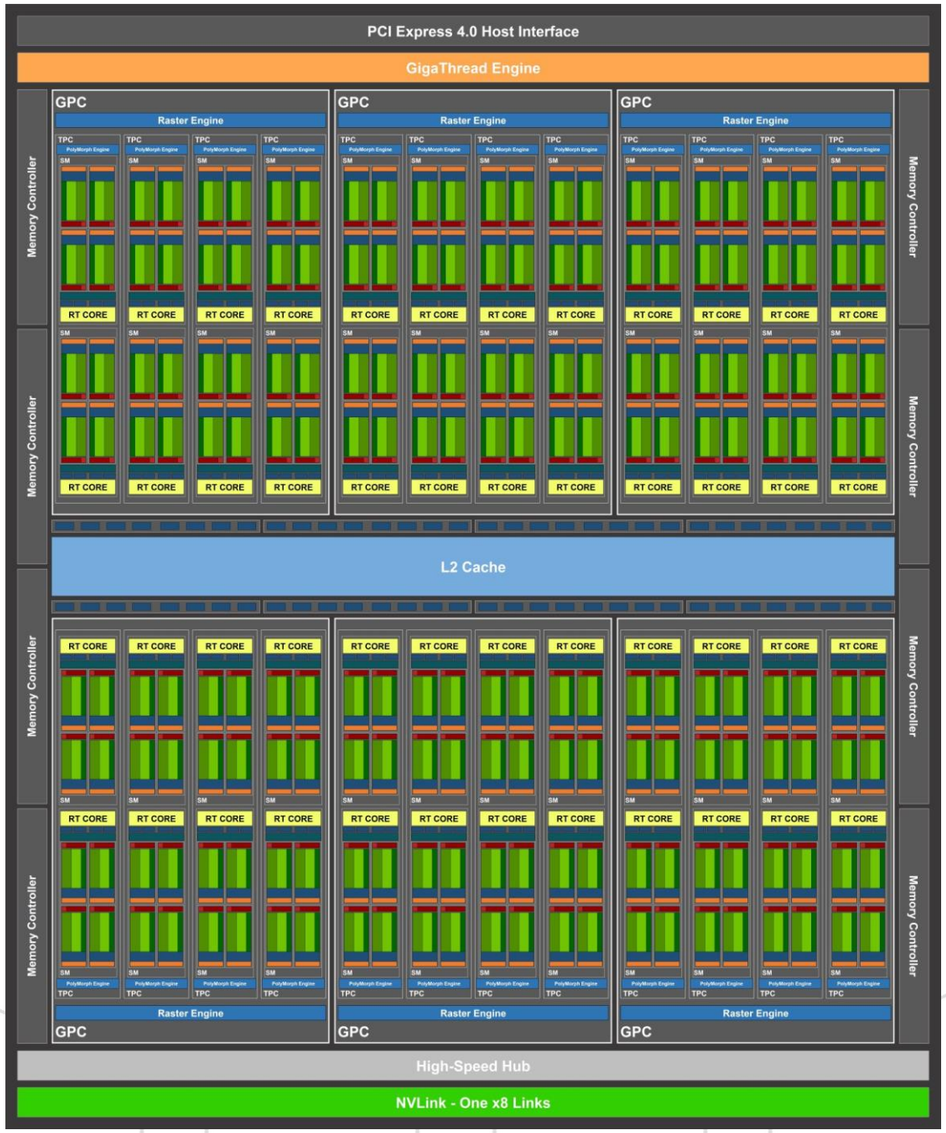
- Pascal
  - 更多GPC、SM、CUDA cores
  - 支持NVLink
    - 极大地增加带宽

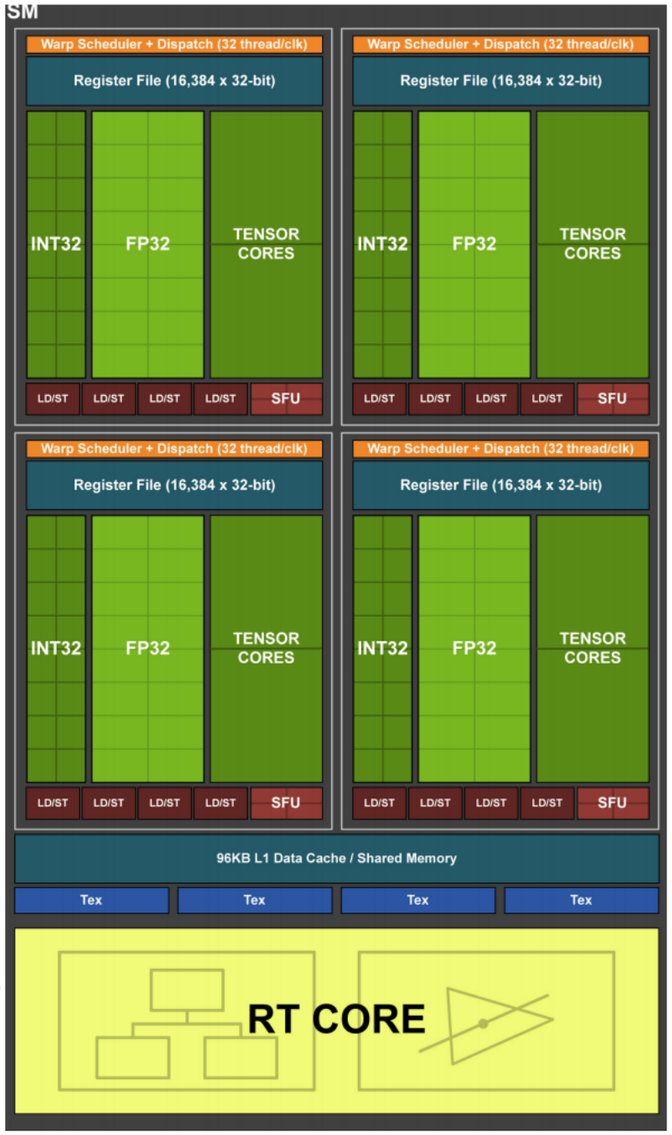



图片来自NVIDIA

# NVIDIA GPU架构

- Turing & Ampere
  - 右图为单个SM
    - 增加Tensor cores
    - 增加Ray Tracing cores




图片来自NVIDIA

# NVIDIA GPU架构

- 计算能力（compute capability）
  - 不同架构的NVIDIA GPU拥有不同计算能力
    - 由硬件决定，不同于CUDA版本
    - 计算能力≠运算性能
    - 参考https://en.wikipedia.org/wiki/CUDA

| Feature support (unlisted features are supported for all compute abilities) | Compute capability (version) | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1.0 | 1.1 | 1.2 | 1.3 | 2.x | 3.0 | 3.2 | 3.5, 3.7, 5.0, 5.2 | 5.3 | 6.x | 7.0/2 (Volta) | 7.5 (Turing) |
| Integer atomic functions operating on 32-bit words in global memory | No | Yes | | | | | | | | | | |
| atomicExch() operating on 32-bit floating point values in global memory | | | | | | | | | | | | |
| Integer atomic functions operating on 32-bit words in shared memory | No | | Yes | | | | | | | | | |
| atomicExch() operating on 32-bit floating point values in shared memory | | | | | | | | | | | | |
| Integer atomic functions operating on 64-bit words in global memory | | | | | | | | | | | | |
| Warp vote functions | | | | | | | | | | | | |
| Double-precision floating-point operations | No | | | Yes | | | | | | | | |

图片截取自https://en.wikipedia.org/wiki/CUDA

# NVIDIA GPU架构

- 计算能力（compute capability）

| Technical specifications | Compute capability (version) | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1.0 | 1.1 | 1.2 | 1.3 | 2.x | 3.0 | 3.2 | 3.5 | 3.7 | 5.0 | 5.2 | 5.3 | 6.0 | 6.1 | 6.2 | 7.0 (7.2?) | 7.5 |
| Maximum number of resident grids per device (concurrent kernel execution) | t.b.d. | | | | 16 | | 4 | 32 | | | | 16 | 128 | 32 | 16 | 128 | |
| Maximum dimensionality of grid of thread blocks | 2 | | | | 3 | | | | | | | | | | | | |
| Maximum x-dimension of a grid of thread blocks | 65535 | | | | $2^{31} - 1$ | | | | | | | | | | | | |
| 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 | 512 | | | | 1024 | | | | | | | | | | | | |
| Maximum z-dimension of a block | 64 | | | | | | | | | | | | | | | | |
| Maximum number of threads per block | 512 | | | | 1024 | | | | | | | | | | | | |
| Warp size | 32 | | | | | | | | | | | | | | | | |
| Maximum number of resident blocks per multiprocessor | 8 | | | | 16 | | | | | 32 | | | | | | | 16 |
| Maximum number of resident warps per multiprocessor | 24 | | 32 | | 48 | 64 | | | | | | | | | | | 32 |
| Maximum number of resident threads per multiprocessor | 768 | | 1024 | | 1536 | 2048 | | | | | | | | | | | 1024 |
| Number of 32-bit registers per multiprocessor | 8 K | | 16 K | | 32 K | 64 K | | | 128 K | 64 K | | | | | | | |
| Maximum number of 32-bit registers per thread block | N/A | | | | 32 K | 64 K | 32 K | 64 K | | | | 32 K | 64 K | | 32 K | 64 K | |
| Maximum number of 32-bit registers per thread | 124 | | | | 63 | 255 | | | | | | | | | | | |
| Maximum amount of shared memory per multiprocessor | 16 KB | | | | 48 KB | | | | 112 KB | 64 KB | 96 KB | 64 KB | | 96 KB | 64 KB | 96 KB (of 128) | 64 KB (of 96) |
| Maximum amount of shared memory per thread block | 48 KB | | | | | | | | | | | | | | | 48/96 KB | 64 KB |

图片截取自https://en.wikipedia.org/wiki/CUDA

# NVIDIA GPU架构

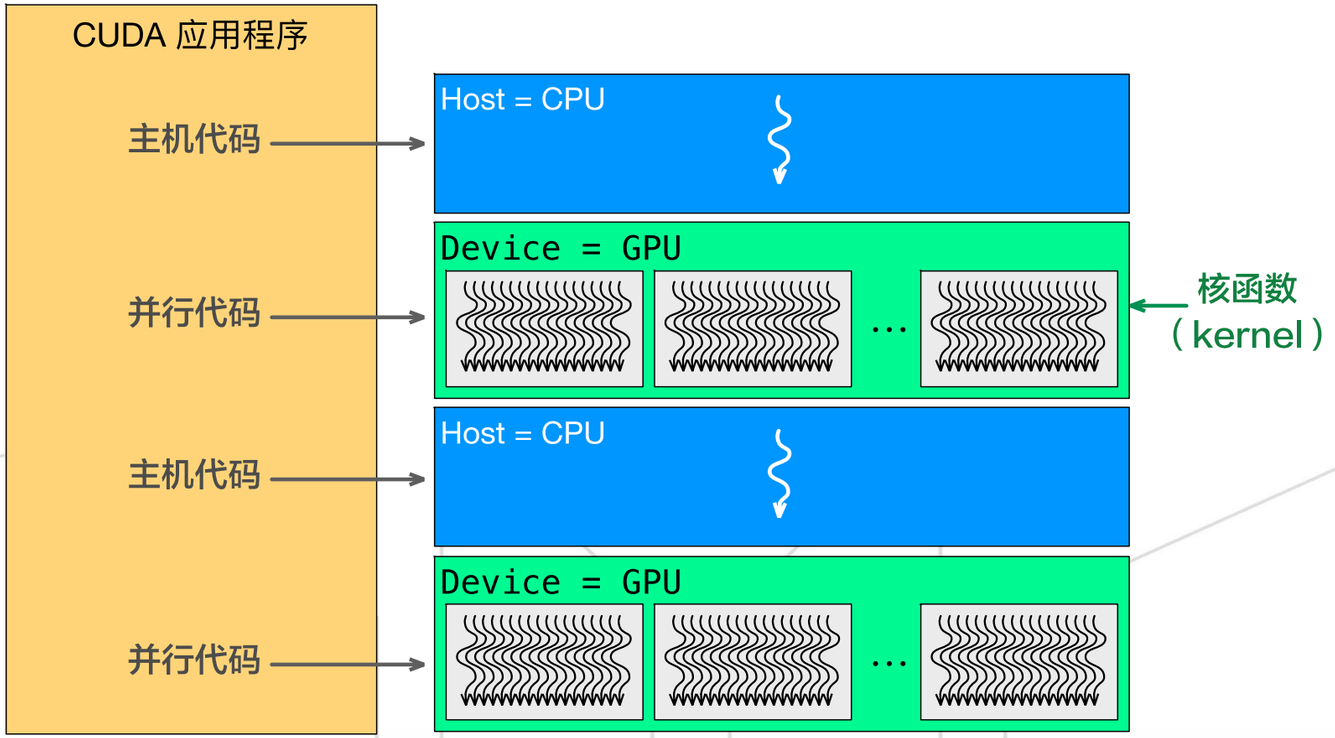
- CUDA的架构分类
  - 异构
    - CPU+GPU
      - CPU执行host代码
      - GPU执行device代码
  - SIMD？
    - 线程束以SIMD方式执行
  - SPMD？
    - SM之间拥有一定的独立性

# 课程概要

- 要点回顾
- CPU vs GPU
- NVIDIA GPU架构
- CUDA编程模型
- CUDA编程举例

# CUDA编程模型

- Host与device
  - Host（CPU相关）：运行在CPU上的代码及主机内存
  - Device（GPU相关）：运行在GPU上的代码及显存（设备内存）
  - 通过在主机上调用核函数（kernel）执行并行代码



# CUDA编程模型

- 指明host与device代码
  - __host__ 从主机端调用，在主机端执行
  - __global__ 从主机端调用，在设备端执行
    - 计算能力5.0后设备已经可以在设备端调用
  - __device__ 从设备端调用，在设备端执行
  - __host__与__device__限定符可以一起使用
    - 函数可以从主机端和设备端调用

```
__global__ void hello_d(){
    printf("Hello World from GPU!");
}

__host__ void hello_h(){
    printf("Hello World from CPU!")
    hello_d<<<1,4>>>();
    cudaDeviceSynchronize();
}
```

输出：

```
Hello World from CPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
```

# CUDA编程模型

- 指明host与device代码
  - <<<1,4>>>为执行配置
    - 指明网格中有1个块
    - 每块中有4个线程
  - cudaDeviceSynchronize();
    - 与OpenMP不同，CUDA核函数为异步执行
    - 调用完成后，控制权立即返回给CPU

```
__global__ void hello_d(){
    printf("Hello World from GPU!");
}

__host__ void hello_h(){
    printf("Hello World from CPU!")
    hello_d<<<1,4>>>();
    cudaDeviceSynchronize();
}
```

输出：

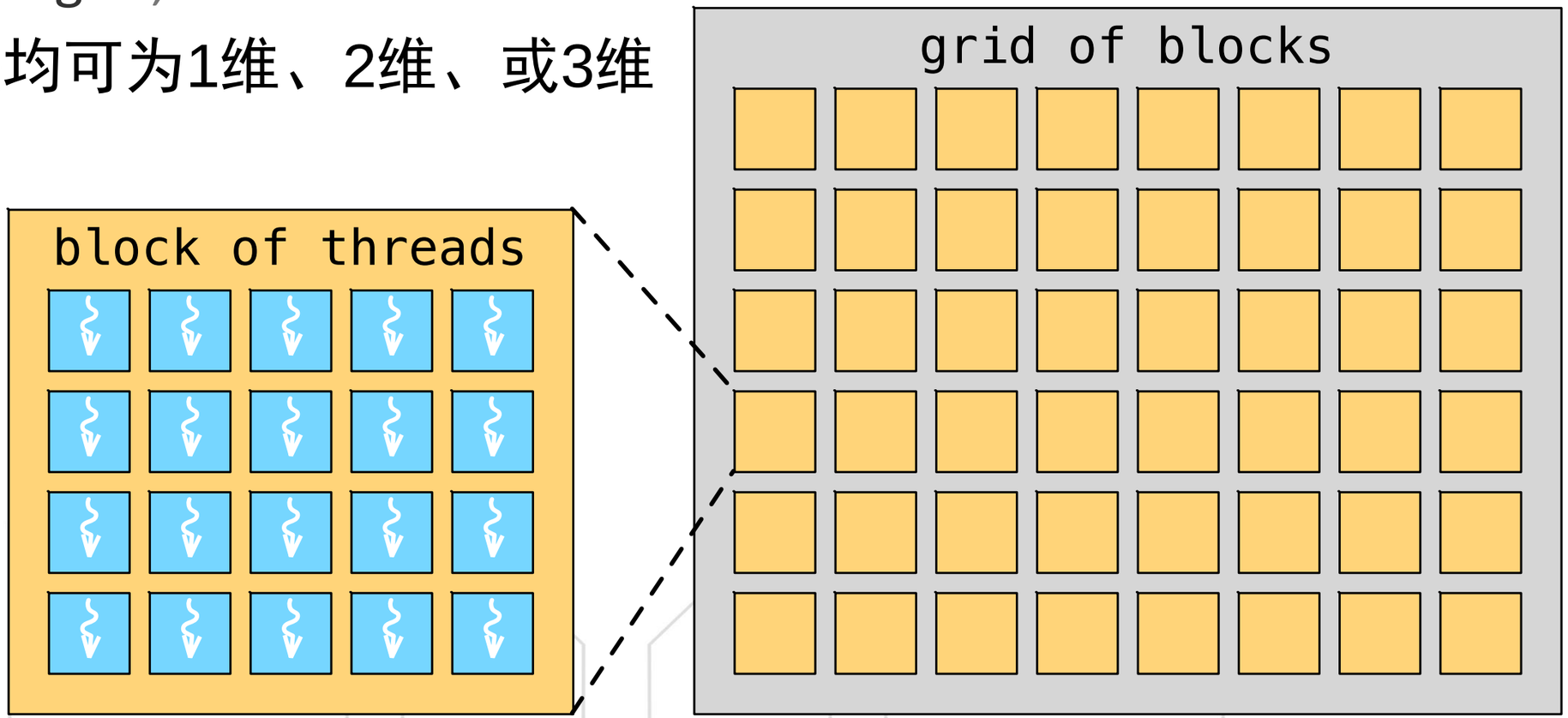
```
Hello World from CPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
```

# CUDA编程模型

- 指明host与device代码
  - 核函数限制条件（__global__函数）
    - 只能访问设备内存
    - 必须返回void
    - 不支持可变数量的参数
      - int func_name(int n_args, ...)
    - 参数不可为引用类型
    - 不支持静态变量

# CUDA编程模型

- 执行配置
  - 指明网格及块的维度
  - 形式为<<< grid, block >>>
  - 网格与块均可为1维、2维、或3维



# CUDA编程模型

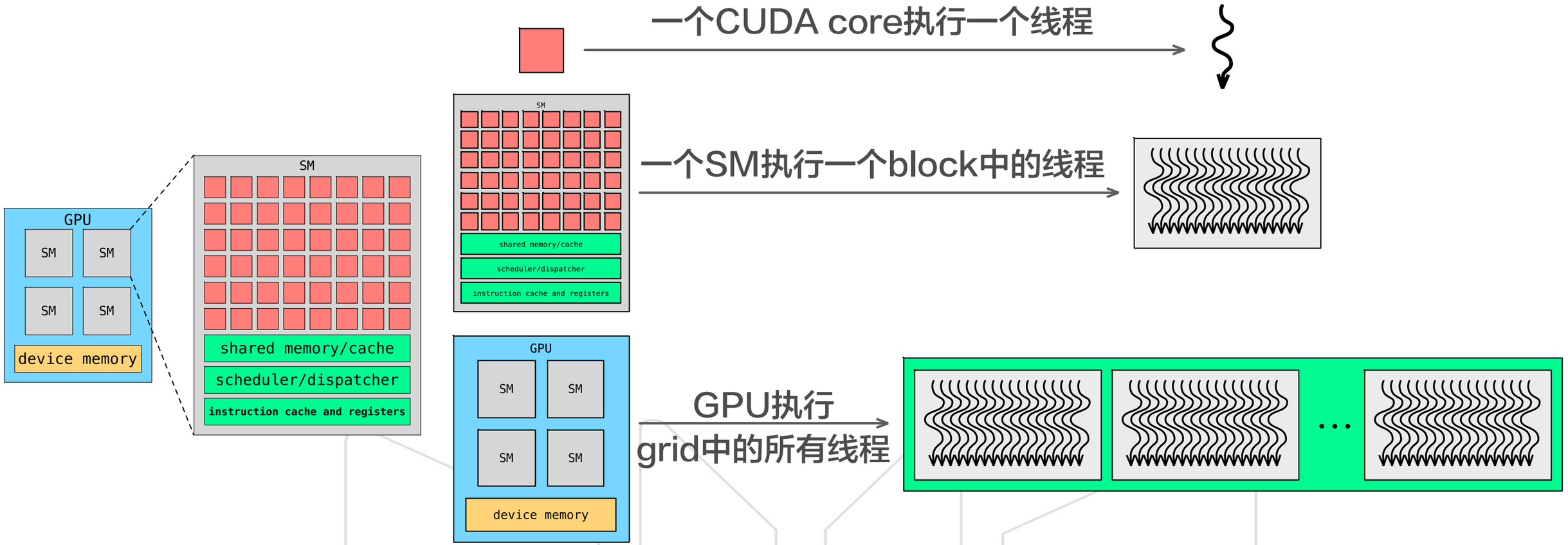
- 执行配置
  - 指明网格及块的维度
  - 形式为<<< grid, block>>>
    - grid与block为dim3类型（三个分量为x, y, z）
    - grid与block的大小受到计算能力的限制

| Technical specifications | Compute capability (version) | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1.0 | 1.1 | 1.2 | 1.3 | 2.x | 3.0 | 3.2 | 3.5 | 3.7 | 5.0 | 5.2 | 5.3 | 6.0 | 6.1 | 6.2 | 7.0 (7.2?) | 7.5 |
| Maximum number of resident grids per device (concurrent kernel execution) | t.b.d. | | | | 16 | | 4 | 32 | | | | 16 | 128 | 32 | 16 | 128 | |
| Maximum dimensionality of grid of thread blocks | 2 | | | | 3 | | | | | | | | | | | | |
| Maximum x-dimension of a grid of thread blocks | 65535 | | | | $2^{31} - 1$ | | | | | | | | | | | | |
| 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 | 512 | | | | 1024 | | | | | | | | | | | | |
| Maximum z-dimension of a block | 64 | | | | | | | | | | | | | | | | |
| Maximum number of threads per block | 512 | | | | 1024 | | | | | | | | | | | | |

图片截取自https://en.wikipedia.org/wiki/CUDA

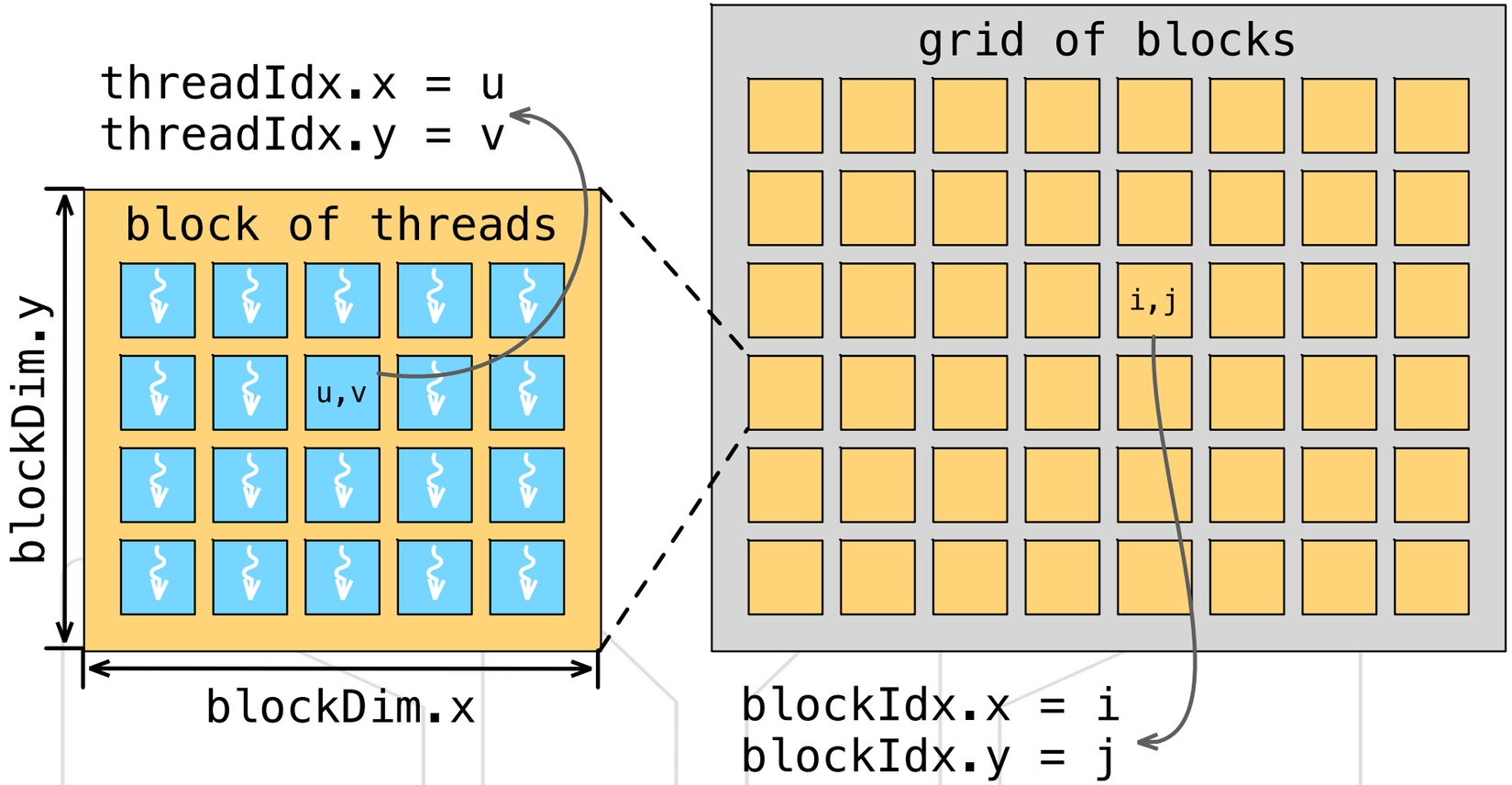
# CUDA编程模型

## GPU架构与线程组织



# CUDA编程模型

- 确定线程编号
  - 使用内置变量threadIdx, blockIdx, blockDim



# CUDA编程模型

- 确定线程编号
  - 使用内置变量threadIdx, blockIdx, blockDim
    - dim3类型
    - 只可用于设备代码上

```
__global__ void hello_d(){
    int bid = blockIdx.x;
    int tid = threadIdx.x;
    printf("Hello World from thread (%d, %d)", bid, tid);
}

__host__ void hello_h(){
    hello_d<<<2,4>>>();
    cudaDeviceSynchronize();
}
```

# 课程概要

- 要点回顾
- CPU vs GPU
- NVIDIA GPU架构
- CUDA编程模型
- CUDA编程举例

# CUDA编程举例

- 向量相加
  - 已知长度为n的两个向量a与b，求向量c，使c[i]=a[i]+b[i]
    - 数据依赖性：非循环迭代相关
    - 串行代码（循环）

```
void vector_add(int *a, int* b, int* c, int n){
    for(int i = 0; i < n; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

# CUDA编程举例

- 例：向量相加
  - 已知长度为n的两个向量a与b，求向量c，使c[i]=a[i]+b[i]
    - 串行代码（循环）

```
void vector_add(int *a, int* b, int* c, int n){
    for(int i = 0; i < n; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

    - CUDA（使用1个block）

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

vector_add<<< 1, n >>>(a, b, c);
```

# CUDA编程举例

- 例：向量相加
  - 已知长度为n的两个向量a与b，求向量c，使c[i]=a[i]+b[i]
    - CUDA（使用1个block）

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

vector_add<<< 1, n >>>(a, b, c);
```

    - 存在问题：
      - a, b, c为主内存地址，GPU无法访问

# CUDA编程举例

- GPU内存管理
  - 创建：cudaMalloc
  - 拷贝：cudaMemcpy
    - 使用cudaMemcpyHostToDevice与cudaMemcpyDeviceToHost指明拷贝方向
  - 释放：cudaFree

```
int *a_h, *b_h, *c_h; //_h常用来表明主机内存
int *a_d, *b_d, *c_d; //_d常用来表明设备内存
int n_bytes = sizeof(int)*n;

cudaMalloc((void**)&a_d, sizeof(int)*n);
cudaMemcpy(a_d, a_h, n_bytes, cudaMemcpyHostToDevice);
... //same for b and cudaMalloc for c

vector_add<<< 1, n >>>(a_d, b_d, c_d);
cudaDeviceSynchronize();
cudaMemcpy(c_h, c_d, n_bytes, cudaMemcpyDeviceToHost);
cudaFree(a_d);
... //same for b and c
```

# 处理错误

- 使用宏定义
  - 来自Grossman and McKercher, “Professional CUDA C Programming”

```
#define CHECK(call)
\{
\    const cudaError_t error = call;
\    if (error != cudaSuccess){
\        printf("Error: %s:%d, ", __FILE__, __LINE__);
\        printf("code:%d, reason: %s \n",
\            error, cudaGetErrorString(error));
\        exit(1);
\    }
\}
```

用法举例：CHECK(cudaMalloc((void**)&a, n_bytes));

# CUDA编程举例

- 例：向量相加
  - 已知长度为n的两个向量a与b，求向量c，使c[i]=a[i]+b[i]
    - CUDA（使用1个block）
      - 假设 a, b, c 均为设备内存

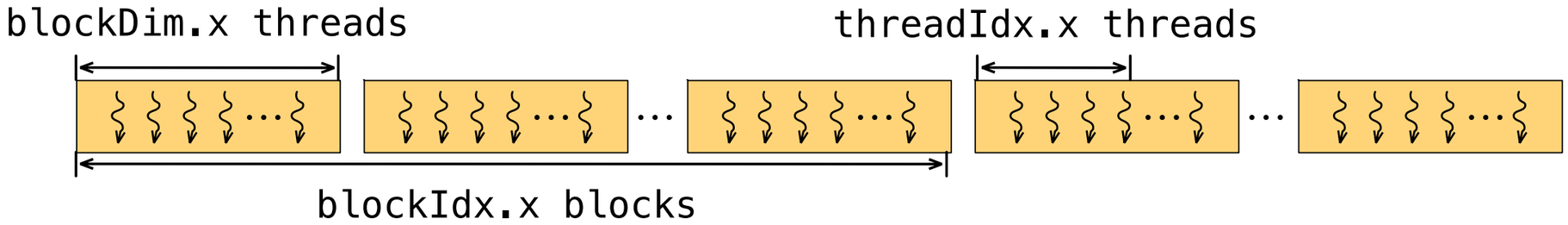
```
__global__ void vector_add(int *a, int* b, int* c){
    int tid = threadIdx.x;
    c[tid] = a[tid] + b[tid];
}

vector_add<<< 1, n >>>(a, b, c);
```

    - 存在问题：
      - block中有最大线程数限制：n必须不大于1024
      - 同一个block只在一个SM上执行：没有充分利用GPU计算资源

# CUDA编程举例

- 例：向量相加
  - 已知长度为n的两个向量a与b，求向量c，使c[i]=a[i]+b[i]
    - CUDA（使用多个block）
      - 每个block使用m个thread（如 m = 32）
      - 确定thread的全局编号




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

vector_add<<< n/m, m>>>(a, b, c);
```

# CUDA编程举例

- 例：向量相加
  - 已知长度为n的两个向量a与b，求向量c，使c[i]=a[i]+b[i]
    - CUDA（使用多个block）

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

vector_add<<< n/m, m>>>(a, b, c);
```

    - 存在问题：n无法被m整除
      - 需对 n/m 向上取整
      - 需判断 tid 是否超过 n

# CUDA编程举例

- 例：向量相加
  - 已知长度为n的两个向量a与b，求向量c，使c[i]=a[i]+b[i]
    - CUDA（使用多个block）

```
__global__ void vector_add(int *a, int* b, int* c, int n){
    int tid = blockDim.x * blockIdx.x + threadIdx.x;

    if (tid < n){
        c[tid] = a[tid] + b[tid];
    }
}

int divup(int n, int m){
    return ((n%m)?(n/m+1):(n/m));
}

vector_add<<< divup(n,m), m>>>(a, b, c, n);
```

# CUDA编程举例

- 例：向量相加
  - 已知长度为n的两个向量a与b，求向量c，使c[i]=a[i]+b[i]
    - CUDA（使用多个block）

可否使用const int& n?

```
__global__ void vector_add(int *a, int* b, int* c, int n){
    int tid = blockDim.x * blockIdx.x + threadIdx.x;

    if (tid < n){
        c[tid] = a[tid] + b[tid];
    }
}

int divup(int n, int m){
    return ((n%m)?(n/m+1):(n/m));
}

vector_add<<< divup(n,m), m>>>(a, b, c, n);
```

# CUDA编程举例

- 例：向量相加
  - 已知长度为n的两个向量a与b，求向量c，使c[i]=a[i]+b[i]
    - CUDA（每个block使用 m 个线程, 每个thread处理 k 个数据）
      - 优缺点？
      - 注意：以下代码并非最优数据访问方式，我们会在性能优化部分重新回顾

```
__global__ void vector_add(int *a, int* b, int* c, int n, int k){
    int tid = blockDim.x * blockIdx.x + threadIdx.x;

    for (int i=tid*k; i<(tid+1)*k && i<n; ++i){
        c[i] = a[i] + b[i];
    }
}

vector_add<<< divup(n,m*k), m >>>(a, b, c, n, k);
```

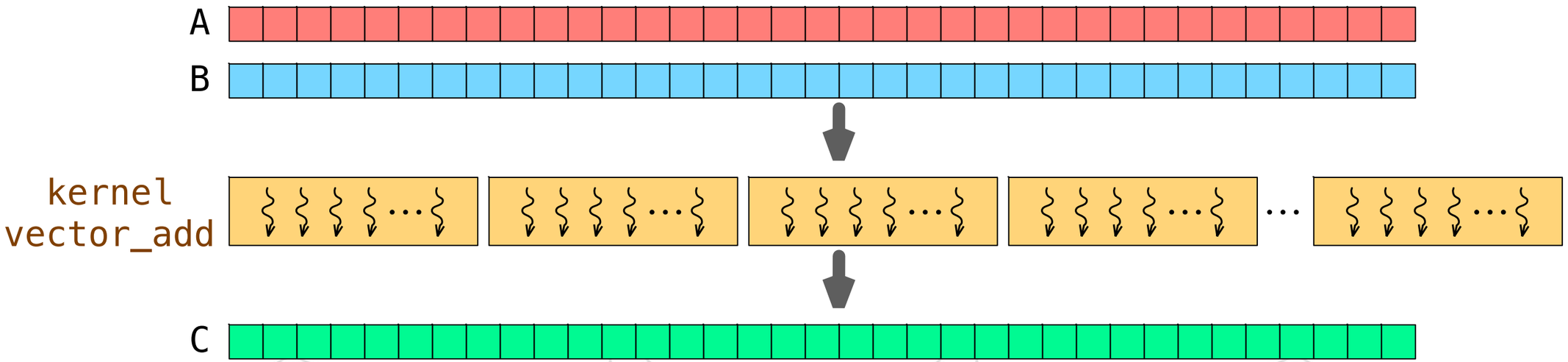
# CUDA vs OpenMP

- CUDA
  - 由核函数指明并行代码
  - 主机代码调用核函数产生设备线程
  - 用户决定每个线程处理的任务
  - 异步执行

- OpenMP
  - 由预处理指令与{}指明并行区域
  - 主线程产生从线程
  - 由调度算法将任务分配到线程上
  - 默认在并行区域结束时同步

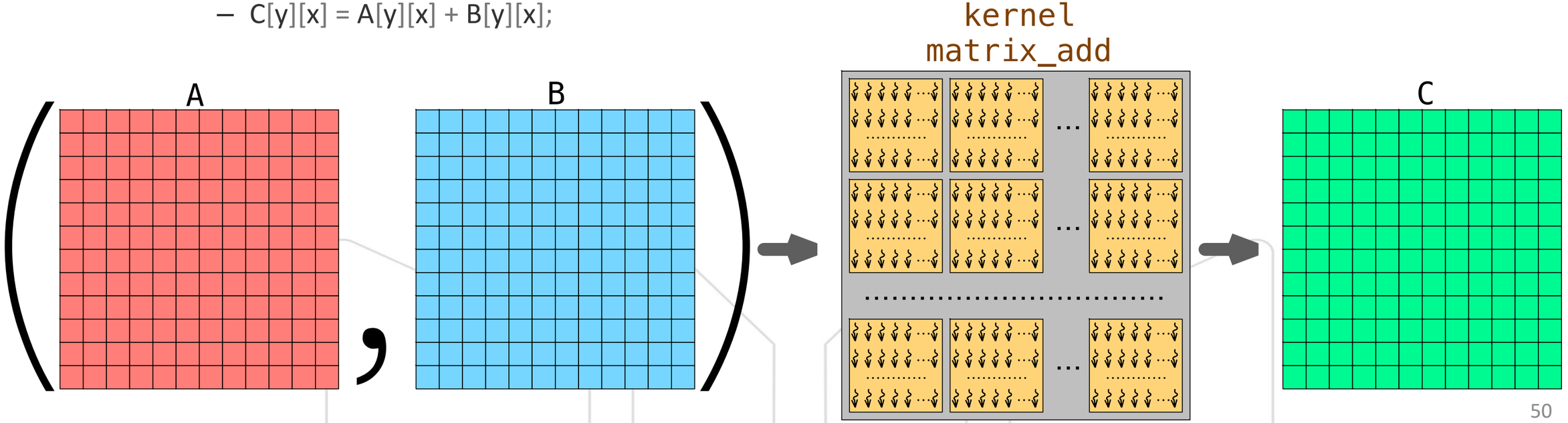
# 二维线程组织

- CUDA向量相加中，数据与线程组织都是一维的
- 如果数据是二维（矩阵）呢？



# 二维线程组织

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 每个线程负责一个位置上的数字相加
    - x、y方向上分别求全局坐标
      - int x = blockDim.x * blockIdx.x + threadIdx.x;
      - int y = blockDim.y * blockIdx.y + threadIdx.y;
      - C[y][x] = A[y][x] + B[y][x];



# 二维线程组织

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维

```
__global__ void matrix_add(int **A, int **B, int **C, int n, int m){
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;
    if (y<n && x<m ){
        C[y][x] = A[y][x] + B[y][x];
    }
}

dim3 block(w, h, 1);
dim3 grid(divup(m, w), divup(n, h), 1);
matrix_add<<< grid, block >>>(A, B, C, n, m);
```

# 二维线程组织

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 存在问题：需要二级指针结构创建二维数组
    - CPU

```
int *A_data = new int[n*m];
int **A = new int*[n];

for (int i=0; i<n; ++i)
    A[i] = &A_data[i*m];
```

    - GPU

```
__global__ void init_matrix(int **A, int *A_data, int n, int m){
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid<n){
        A[tid] = &A_data[tid*m];
    }
}

int *A_data, **A;
cudaMalloc((void*)&A_data, sizeof(int)*n*m);
cudaMalloc((void*)&A, sizeof(int*)*n);
```

# 二维线程组织

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 存在问题：需要二级指针结构创建二维数组
    - 解决方案：使用一维数组存放矩阵
      - 即使在前一个版本中，数据在内存中也是一维连续（int *A_data）
      - int **A只提供访问数据的方式（ A[y][x]增加内存访问次数！）

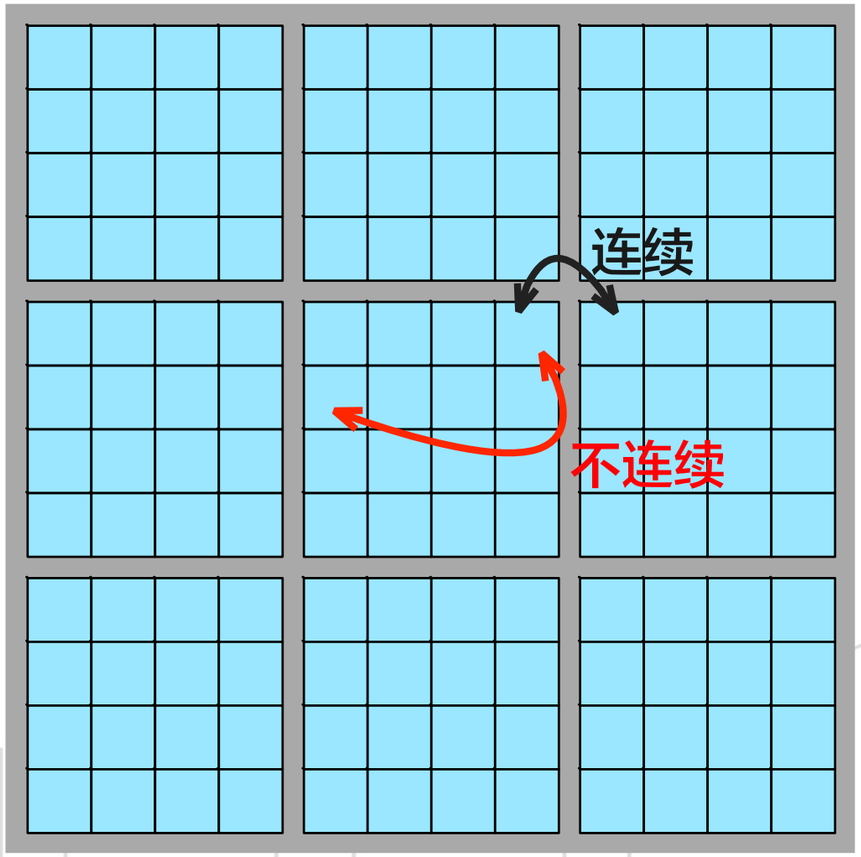
```
__global__ void matrix_add(int *A, int *B, int *C, int n, int m){
   int x = blockDim.x * blockIdx.x + threadIdx.x;
   int y = blockDim.y * blockIdx.y + threadIdx.y;

   if (y<n && x<m ){
      C[y*m+x] = A[y*m+x] + B[y*m+x];
   }
}

dim3 block(w, h, 1);
dim3 grid(divup(m, w), divup(n, h), 1);
matrix_add<<< grid, block >>>(A, B, C , n , m);
```

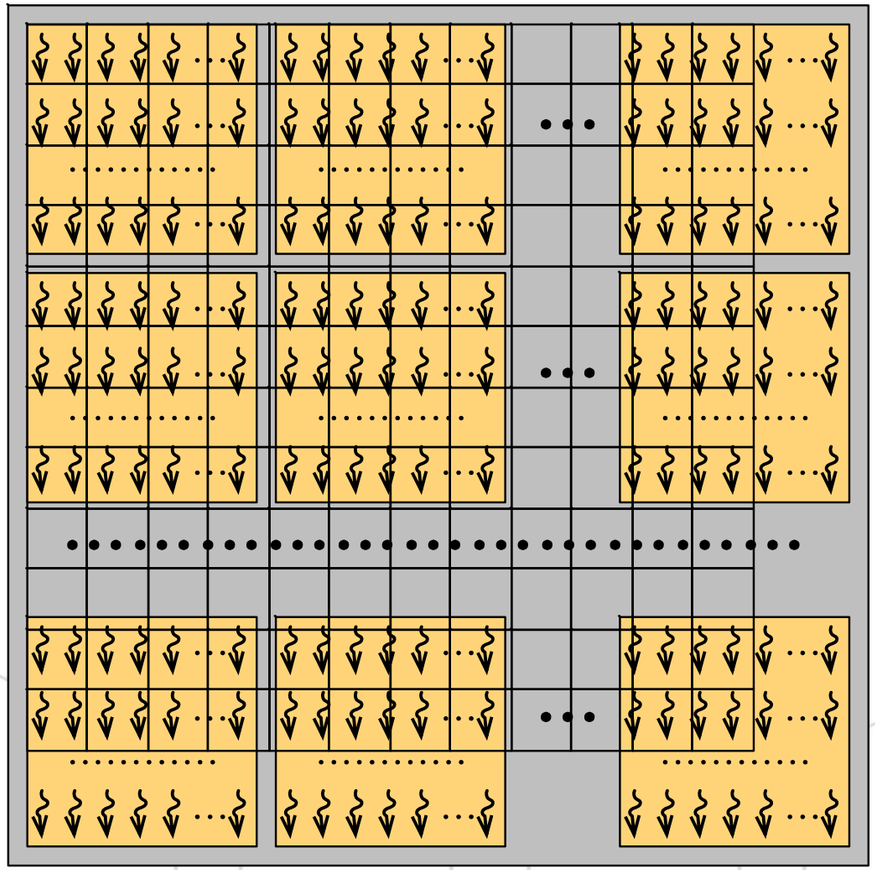
# 二维线程组织

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 存在问题：block中线程访问的内存空间不连续
      - 内存访问的局部性差



# 二维线程组织

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 存在问题：在x、y维度上都可能出现余数
      - 线程利用率低



# 二维线程组织

- 例子：矩阵相加
  - 直接将一维gird与block扩展至二维
    - 存在问题：block中线程访问的内存空间不连续
      在x、y维度上都可能出现余数
    - 解决方案：使用一维grid与block
      - 与**vector_add**(A, B, C, n*m)等价!

```
__global__ void matrix_add(int *A, int *B, int *C, int n, int m){
   int tid = blockDim.x * blockIdx.x + threadIdx.x;

   if ( tid<n*m ){
      C[tid] = A[tid] + B[tid];
   }
}

matrix_add<<< divup(n*m, block_size), block_size >>>(A, B, C);
```

# 二维线程组织

- 例子：矩阵相加
  - 线程组织方式和内存中数据的组织方式可以互相独立
    - 二维grid与block也可以用于一维数据，反之亦然
  - 相同的组织方式在编程中显得更直观
    - 但效率不一定最优
    - 效率归根结底由硬件结构决定

# 小结

- CUDA架构特征
  - 异构：CPU+GPU
  - 兼具SIMD与SPMD的特征
  - 大量超轻量级线程提高吞吐量
- CUDA编程结构
  - Host与device
    - __host__, __global__, __device__
    - 内存管理
  - 线程组织
    - 网格（grid）与块（block）
    - 组织方式可以是一维、二维、三维

# Questions?