



中山大學  
SUN YAT-SEN UNIVERSITY



# 多核程序设计与实践

## CUDA编程模型

陶钧

[taoj23@mail.sysu.edu.cn](mailto:taoj23@mail.sysu.edu.cn)

中山大学 数据科学与计算机学院  
国家超级计算广州中心

- 计算机架构分类
- CPU vs GPU
- NVIDIA GPU架构
- CUDA编程模型
- CUDA编程举例



## ● Flynn's taxonomy

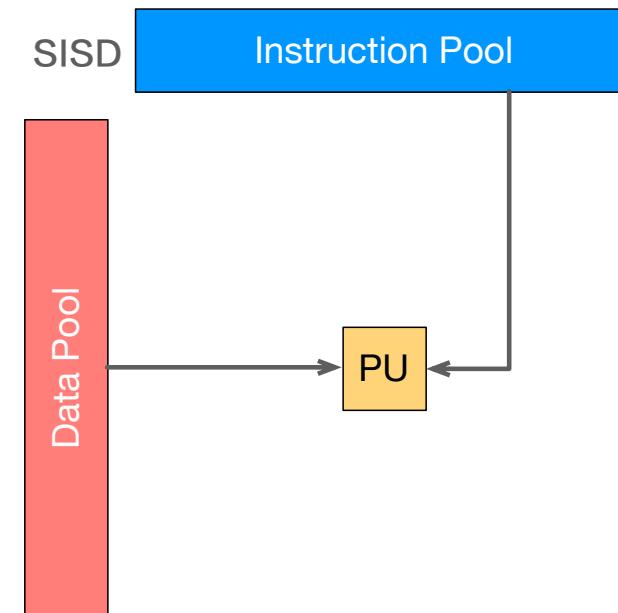
- 1972年由费林（Michael J. Flynn）提出
- 根据指令和数据进入CPU的方式分类
  - SISD, SIMD, MISD, MIMD
- 拓展分类
  - SPMD, MPMD

		Instruction	
		single	multiple
Data	single	SISD  von Neumann single CPU computer	MISD  pipelined computer
	multiple	SIMD  vector processors	MIMD  multi computers/ processors

## ● Flynn's taxonomy

### – SISD

- 经典冯诺依曼架构
- PU：处理单元



### – SIMD

- 多个处理单元使用同样指令处理不同数据

– 早期向量超级计算机

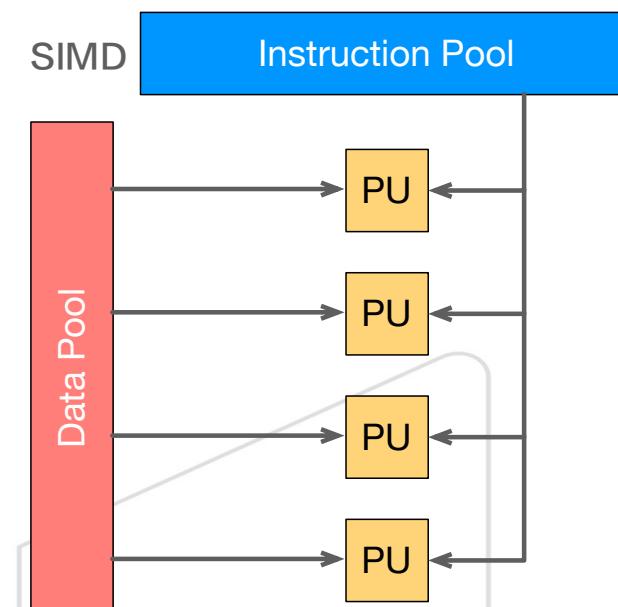
» 70年代–90年代

– 向量处理器

» GPU?

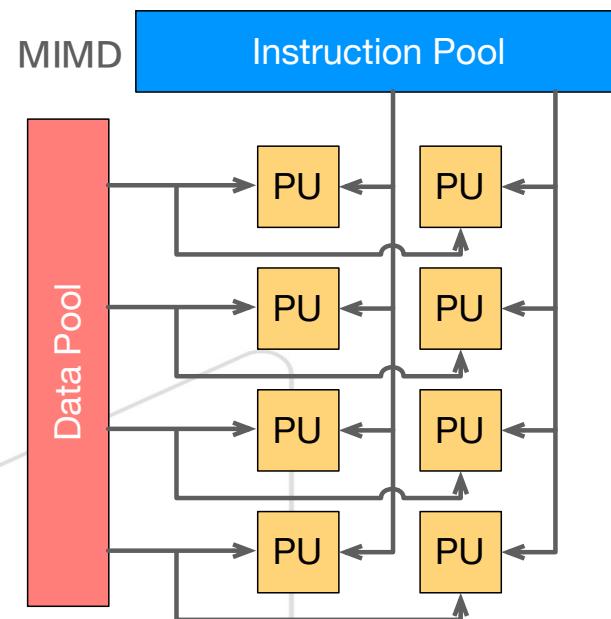
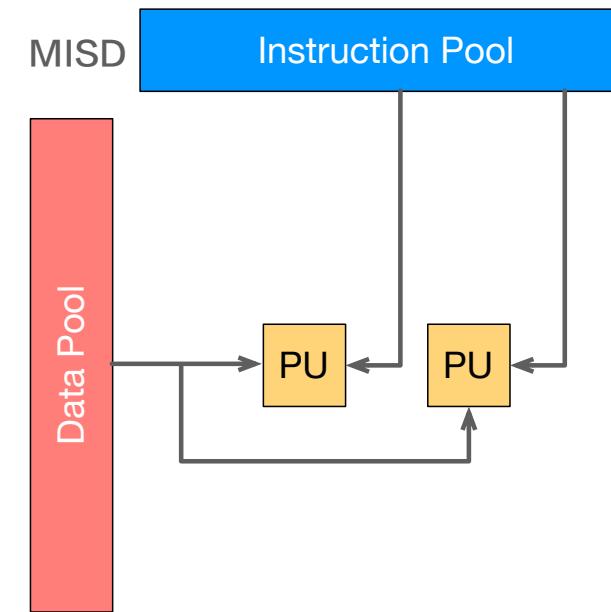
– 现代CPU具有SIMD指令

» 如Intel MMX, SSE指令集等



## ● Flynn's taxonomy

- MISD
  - 流水线架构
- MIMD
  - 多个处理单元使用不同指令处理不同数据
    - 处理单元之间可完全独立
    - 从2006年起, 前10的超级计算机以及Top 500榜单上大多数超级计算机都采用此架构
    - OpenMP编程模型也基于此架构
  - SPMD与MPMD也属于此分类
    - P: Program



- 计算机架构分类
- CPU vs GPU
- NVIDIA GPU架构
- CUDA编程模型
- CUDA编程举例



## ● 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

- 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

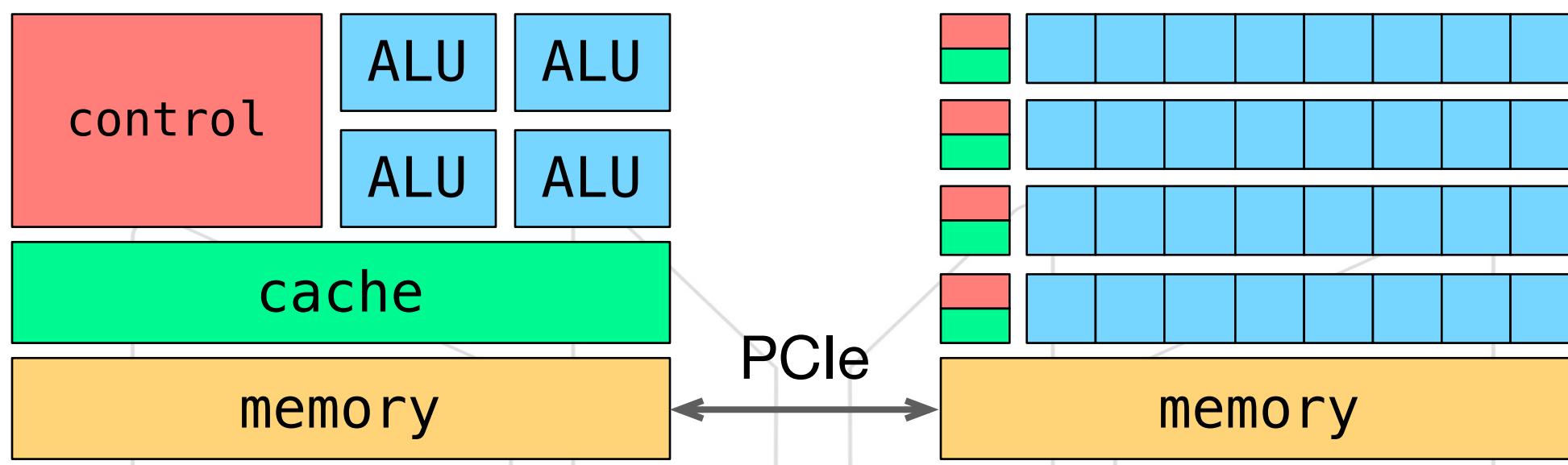
## ● GPU

- 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



- 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
  - 最新显卡支持NVLink（提供5-12倍于PCIe 3.0总线带宽）

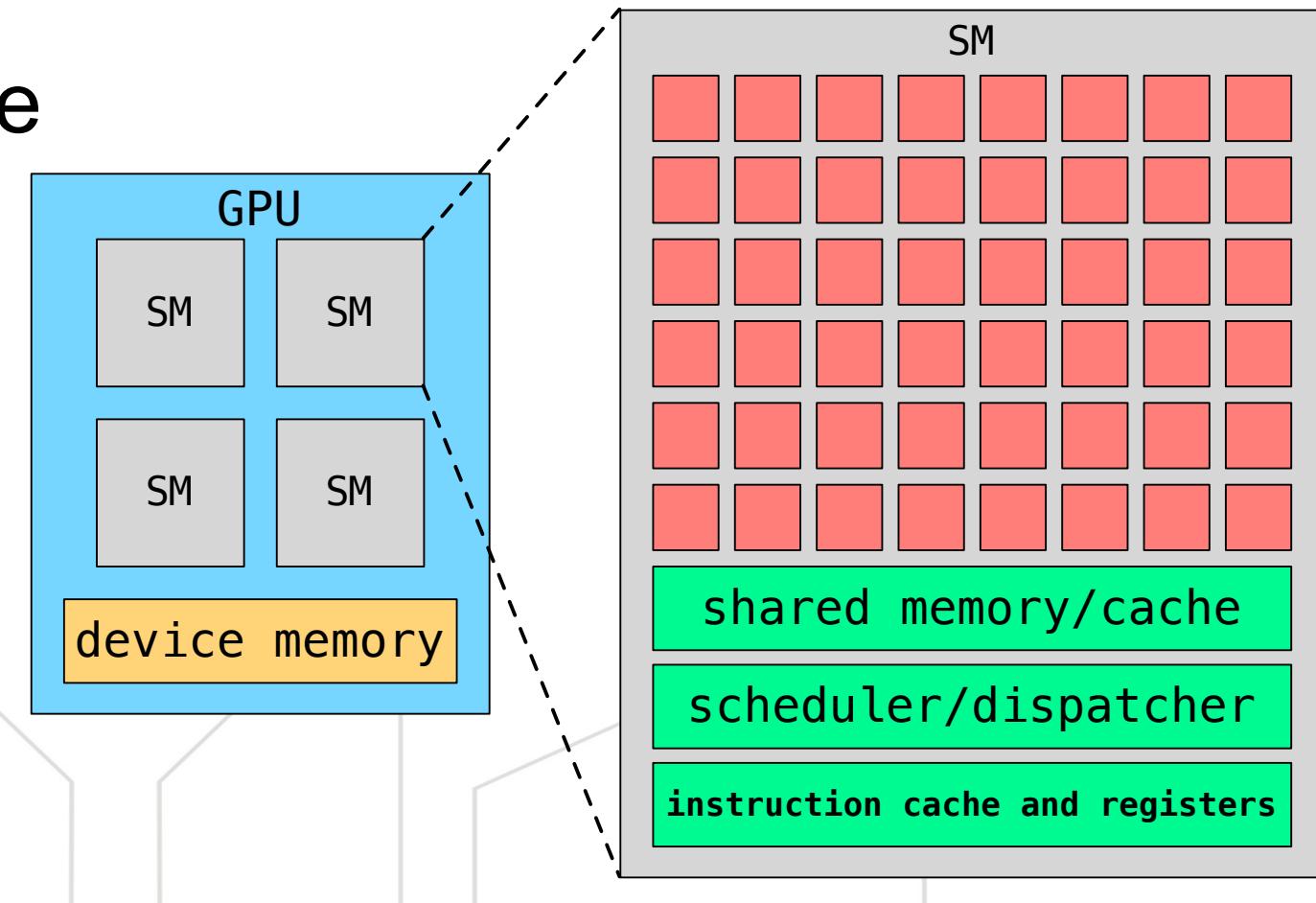


- 计算机架构分类
- CPU vs GPU
- NVIDIA GPU架构
- CUDA编程模型
- CUDA编程举例



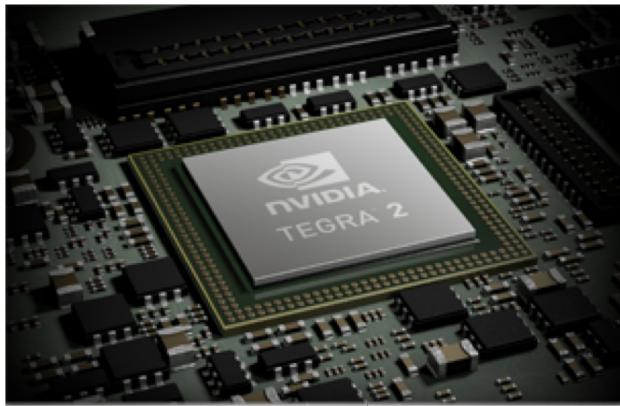
## ● 2级架构

- 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



## ● 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
- 更强大的双精度运算能力
- 更大的内存带宽



## ● 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
- 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



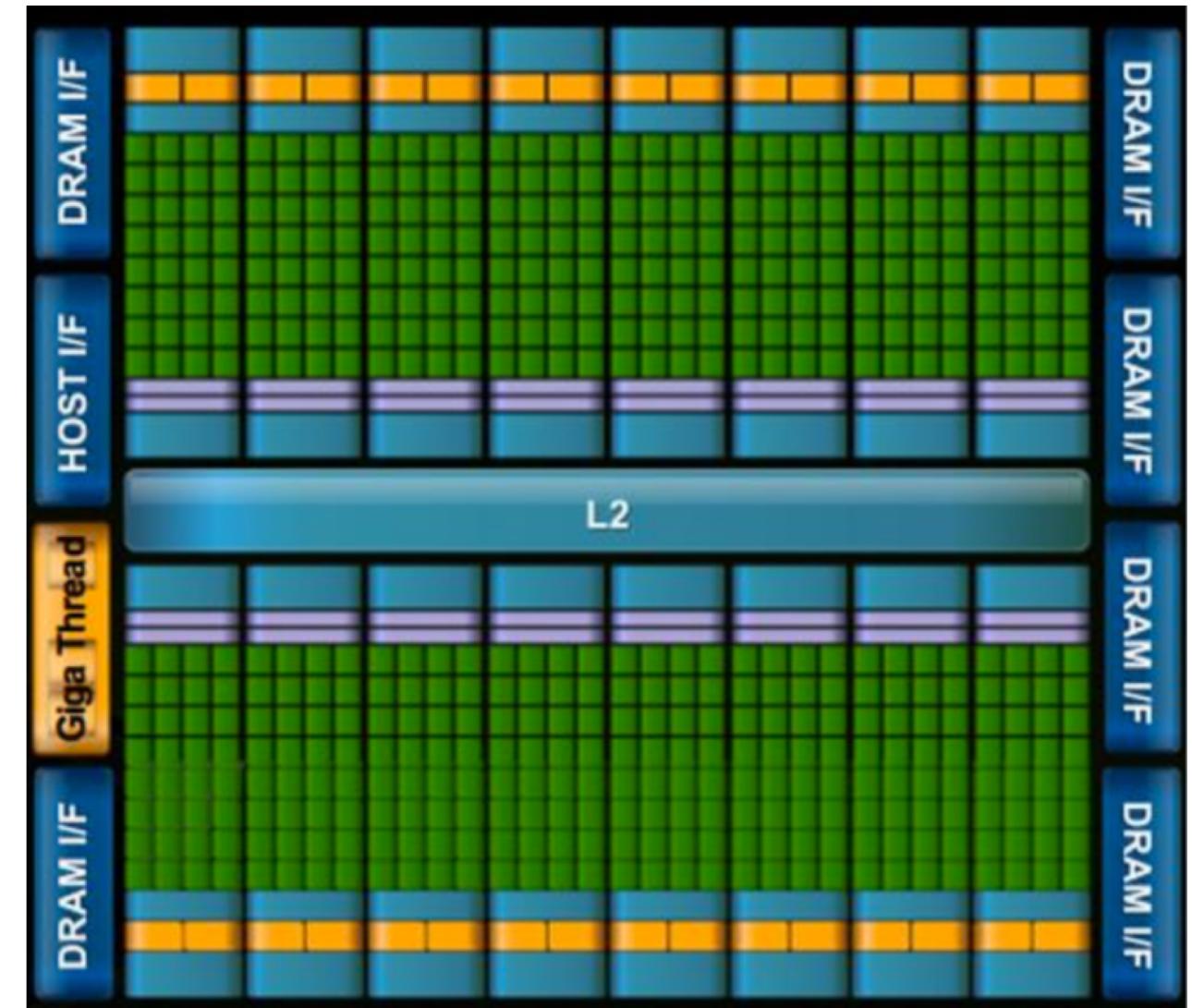
- Tesla (2006) → Fermi (2010) → Kepler (2012) → Maxwell (2014)  
→ Pascal (2016) → Volta (2017) → Turing (2018)

- 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

## ● 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

## ● 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

## ● Maxwell

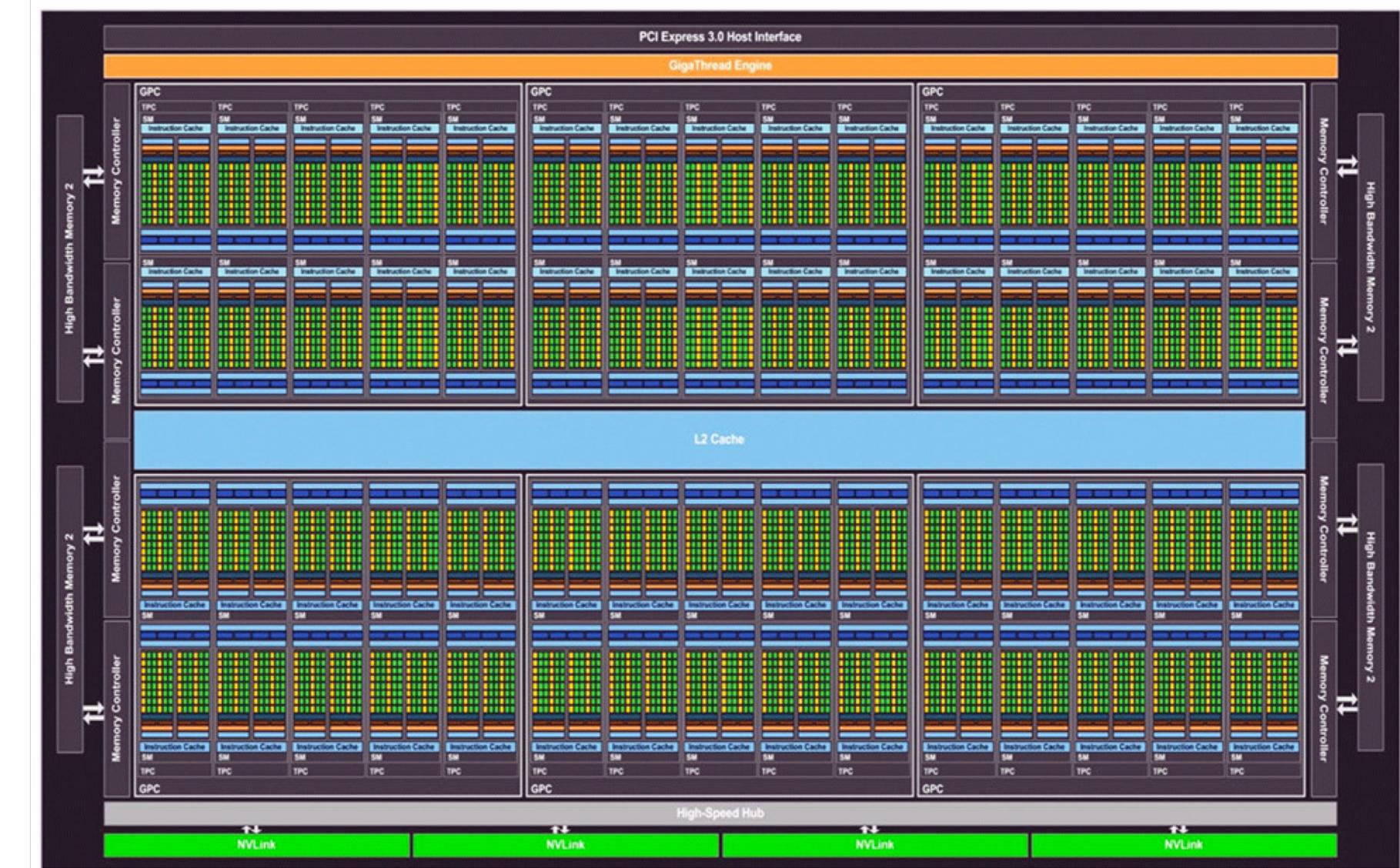
- 芯片划分为四个GPC (quadrant)
- 每个GPC上有多个 Streaming Multiprocessor Module (SMM)
- SMM减少了core数目但提高了能源效率
  - $192 \rightarrow 128$
  - 50%能耗
  - 90%运算能力



图片来自NVIDIA

## ● Pascal

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



图片来自NVIDIA

## ● Turing

- 左图为单个SM
- 增加Tensor cores
- 增加Ray Tracing Cores



图片来自NVIDIA

## ● 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											Yes
Integer atomic functions operating on 32-bit words in shared memory	No										Yes
atomicExch() operating on 32-bit floating point values in shared memory											Yes
Integer atomic functions operating on 64-bit words in global memory	No										Yes
Warp vote functions											
Double-precision floating-point operations	No										Yes

## ● 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				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				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		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															

## ● 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

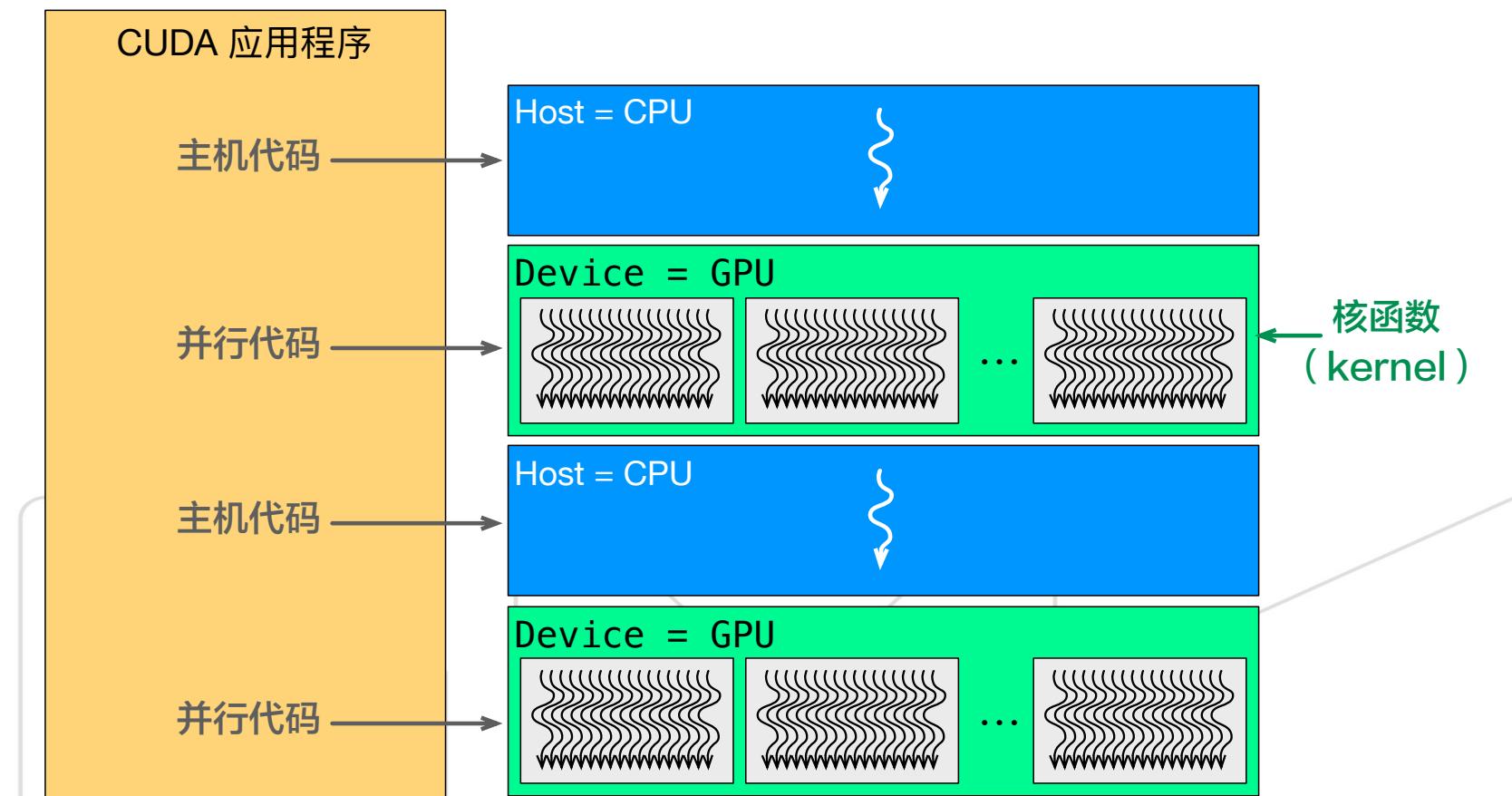


- 计算机架构分类
- CPU vs GPU
- NVIDIA GPU架构
- CUDA编程模型
- CUDA编程举例



## ● 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



- 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
- 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!

## ● 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();
}
```

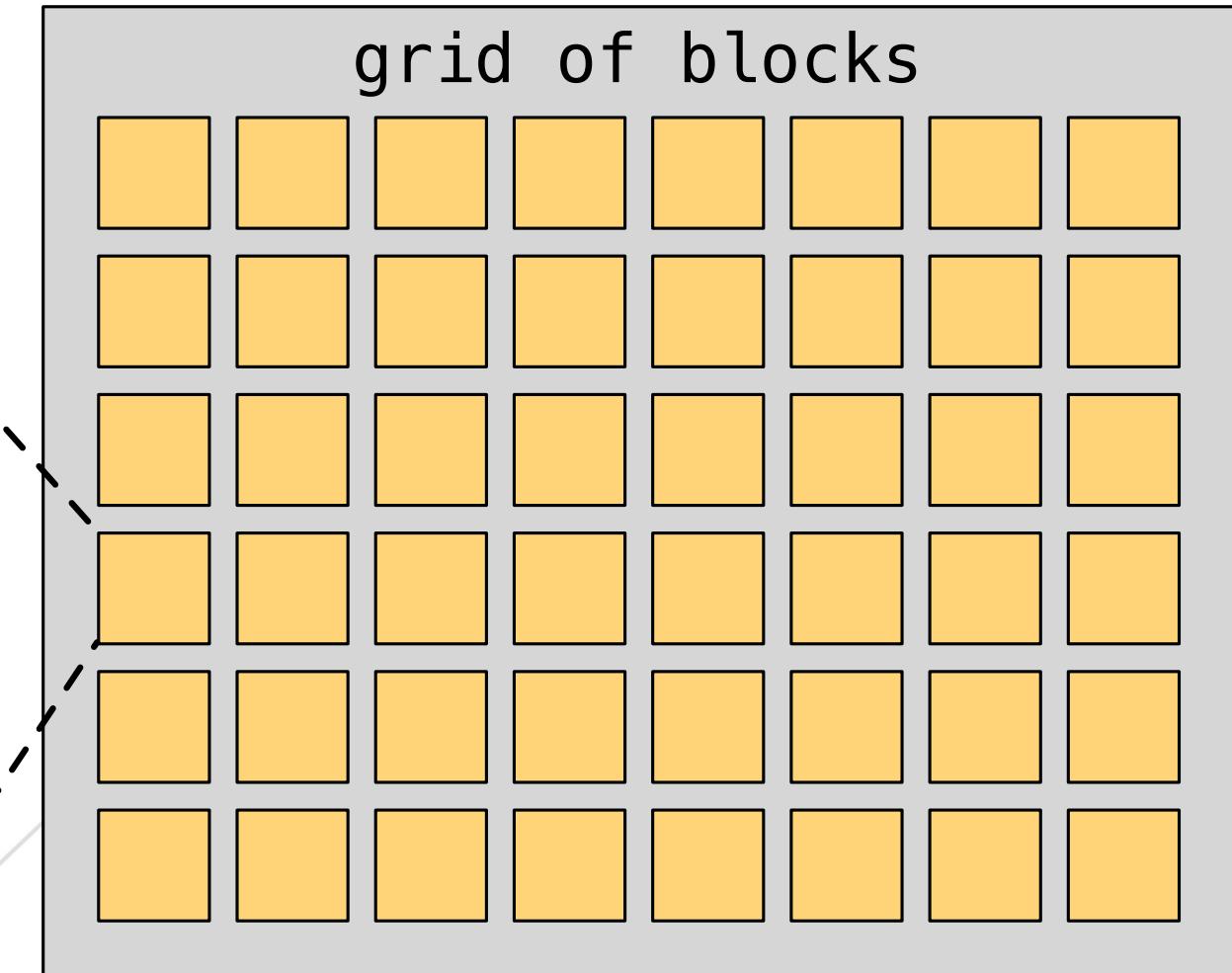
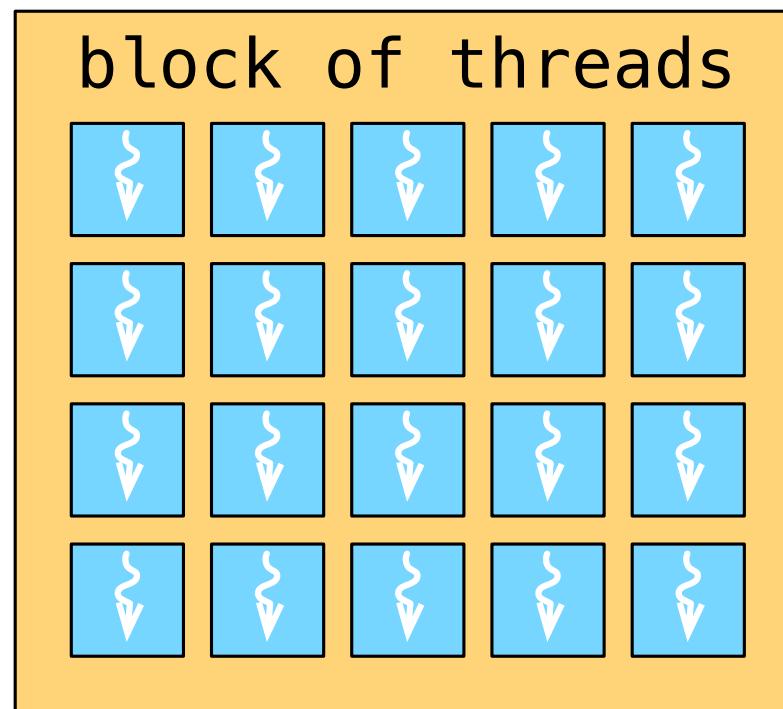
输出：  
Hello World from CPU!  
Hello World from GPU!  
Hello World from GPU!  
Hello World from GPU!  
Hello World from GPU!

- 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



## ● 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

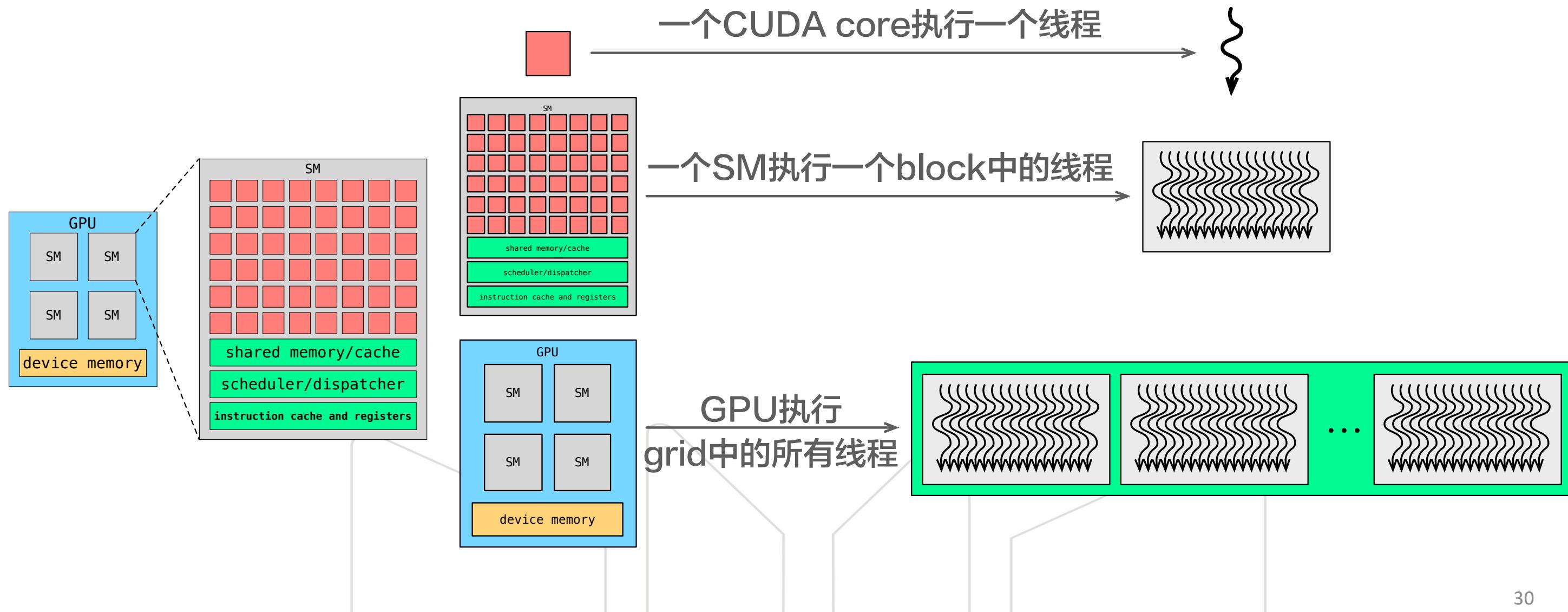


## ● 执行配置

- 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

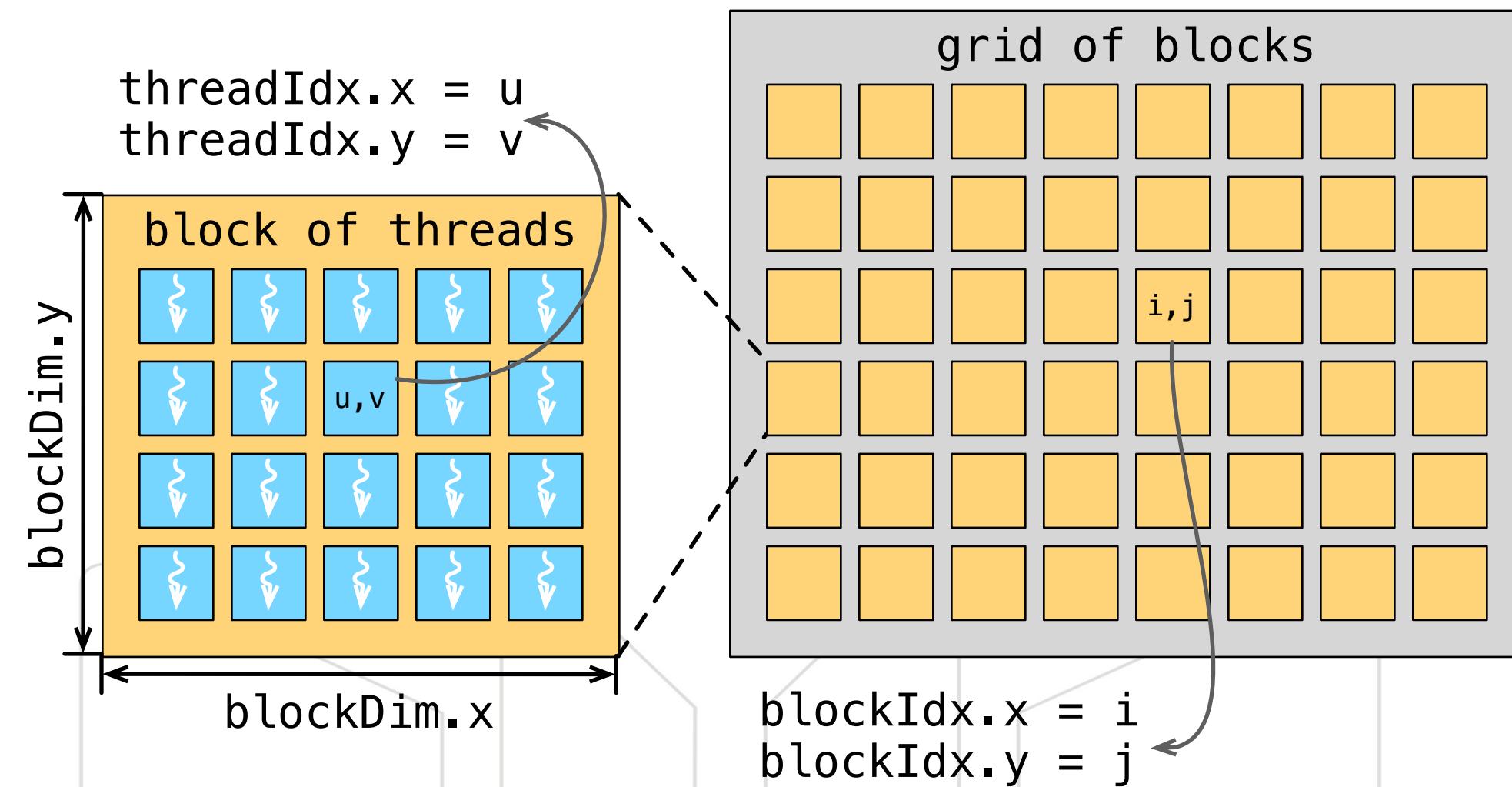
Technical specifications	Compute capability (version)																
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0 (7.2?)	7.5
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16	4			32			16	128	32	16		128
Maximum dimensionality of grid of thread 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						

## ● GPU架构与线程组织



## ● 确定线程编号

- 使用内置变量`threadIdx`, `blockIdx`, `blockDim`



## ● 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();
}
```

- 计算机架构分类
- CPU vs GPU
- NVIDIA GPU架构
- CUDA编程模型
- 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];  
    }  
}
```



## ● 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[i] = a[i] + b[i];  
}
```

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

## ● 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[tid] = a[tid] + b[tid];  
}
```

```
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

## ● 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 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
```

## ● 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));**

## ● 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);
```

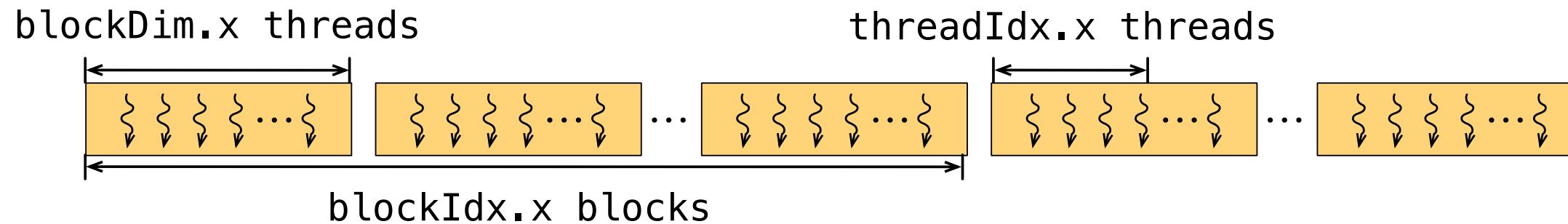
- 存在问题：
  - block中有最大线程数限制：n必须不大于1024
  - 同一个block只在一个SM上执行：没有充分利用GPU计算资源

## ● 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);
```

## ● 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  $n$

## ● 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 (if(n%m)?(n/m+1):(n/m));  
}  
  
vector_add<<< divup(n,m), m>>>(a, b, c);
```

## ● 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 (if(n%m)?(n/m+1):(n/m));  
}  
  
vector_add<<< divup(n,m), m>>>(a, b, c);
```

## ● 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？

```
__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);
```

## ● 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



## ● CUDA架构特征

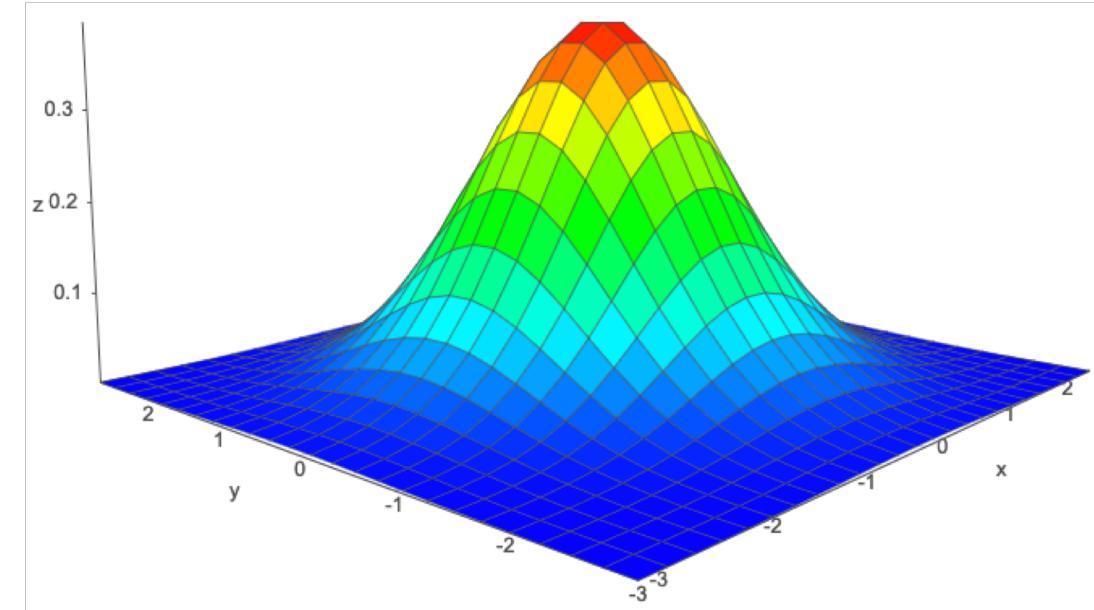
- 异构: CPU+GPU
- 兼具SIMD与SPMD的特征
- 大量超轻量级线程提高吞吐量

## ● 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)

## ● 计算二维高斯函数

$$g(x, y) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



## ● 输入输出

- 输入：整数  $s = \sigma$
- 输出： $(6s+1) \times (6s+1)$  的二维单精度浮点型数组 arr
  - $g(0, 0)$  位于二维数组中心  $\text{arr}[3s][3s]$  处
  - $g(-3s, -3s)$  位于二维数组左下角  $\text{arr}[0][0]$  处
  - 输出代码示范：

```
for(int i = 0; i < 6*s+1; i++)  
    for(int j = 0; j < 6*s+1; j++)  
        printf("%5.4f ", arr[i][j]);
```

## ● 输出样例

- $s=2$
- 见 github 项目中 `hw1-sample-output-s-2.txt`
- 注意不要输出额外内容

```
0.0000 0.0001 0.0003 0.0007 0.0013 0.0020 0.0022 0.0020 0.0013 0.0007 0.0003 0.0001 0.0000
0.0001 0.0004 0.0012 0.0028 0.0053 0.0077 0.0088 0.0077 0.0053 0.0028 0.0012 0.0004 0.0001
0.0003 0.0012 0.0037 0.0088 0.0164 0.0238 0.0270 0.0238 0.0164 0.0088 0.0037 0.0012 0.0003
0.0007 0.0028 0.0088 0.0210 0.0393 0.0571 0.0648 0.0571 0.0393 0.0210 0.0088 0.0028 0.0007
0.0013 0.0053 0.0164 0.0393 0.0734 0.1068 0.1210 0.1068 0.0734 0.0393 0.0164 0.0053 0.0013
0.0020 0.0077 0.0238 0.0571 0.1068 0.1553 0.1760 0.1553 0.1068 0.0571 0.0238 0.0077 0.0020
0.0022 0.0088 0.0270 0.0648 0.1210 0.1760 0.1995 0.1760 0.1210 0.0648 0.0270 0.0088 0.0022
0.0020 0.0077 0.0238 0.0571 0.1068 0.1553 0.1760 0.1553 0.1068 0.0571 0.0238 0.0077 0.0020
0.0013 0.0053 0.0164 0.0393 0.0734 0.1068 0.1210 0.1068 0.0734 0.0393 0.0164 0.0053 0.0013
0.0007 0.0028 0.0088 0.0210 0.0393 0.0571 0.0648 0.0571 0.0393 0.0210 0.0088 0.0028 0.0007
0.0003 0.0012 0.0037 0.0088 0.0164 0.0238 0.0270 0.0238 0.0164 0.0088 0.0037 0.0012 0.0003
0.0001 0.0004 0.0012 0.0028 0.0053 0.0077 0.0088 0.0077 0.0053 0.0028 0.0012 0.0004 0.0001
0.0000 0.0001 0.0003 0.0007 0.0013 0.0020 0.0022 0.0020 0.0013 0.0007 0.0003 0.0001 0.0000
```

● 正确性			● 书面报告	
– CUDA	30		– 解释程序设计逻辑	10
– OpenMP	10		– 讨论参数对程序性能的影响	10
● 性能			• Block的大小	
– CUDA	30		• 每个线程需要完成的任务数	
– 根据实际搜集的运行时间分布决定			– 讨论性能优化方法	+10
– 标准将与成绩一并公布				
● 编程规范			● 提交作业	
– 初始分	10		– 邮箱: MulticoreSYSU@163. com	
– 缺少文件头	-5		– 截止时间	
– 缺少函数头	-5		• 4月8日晚23:59	
– 换行没有正确缩进	-5		• 如需使用slip days, 请于截止	
– 函数过长	-5		时间前将需要使用的天数发送至	
			提交作业邮箱	

## ● 编程规范举例

- 应组织到适当文件中
- 应根据功能适当划分成函数

文件头

```
#####
## 姓名: XXX
## 文件说明: *****
## *****
## *****
## *****
## *****
#####
#include <abc.h>

int main(){
    ...
}
```

函数头

```
#####
## 函数: do_something
## 函数描述: *****
## *****
## 参数描述:
## par1: *****
## par2: *****
#####
void do_something(par1, par2){
    for( ... ){
        if( ... ){
            ...
        }
    }
}
```

## ● 提交文件结构说明

- your name-your ID
  - README (实验报告)
  - sources
    - all sources files
    - Makefile (样本见课程github项目 )
- 保证TA能在集群上使用如下代码测试你的程序
  - make编译应产生名为hw1\_cuda与hw1\_omp的可执行文件
  - s为输入

```
cd "your name-your ID/sources"  
make  
./hw1_cuda s  
./hw1_omp s
```



- 课件
  - [https://github.com/multicoresysu/multicore\\_sysu\\_slides](https://github.com/multicoresysu/multicore_sysu_slides)
- 学院集群网址（新）
  - <http://222.200.180.181:28377/>
  - 关于编译
    - Makefile见课程github项目
    - 在集群上使用nvcc编译需先安装gcc-6，安装步骤见课程github项目



# Questions?

