

# CUDA编程入门及上机练习

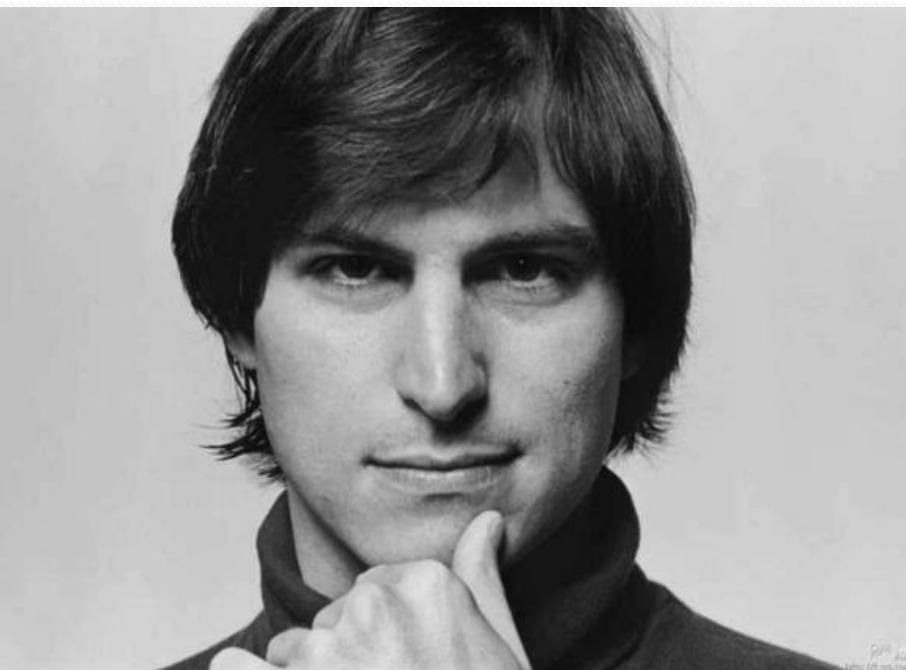
朱有亮

2019年10月2日

# 理解 > 记忆

编程不是简单的逻辑+计算

编程是任务划分，是一种解决问题的思维方式(结构化思维模式)



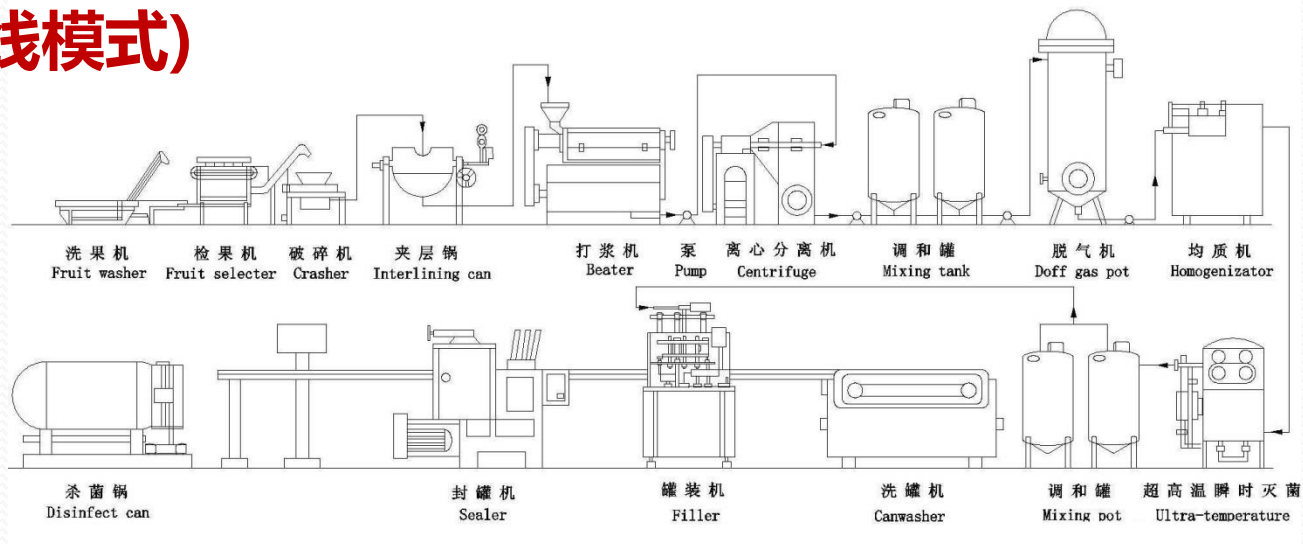
“这个国家的每个人都应该学习如何编程，因为它能教会你如何思考。”

——乔布斯

## 面向过程编程(生产线模式)

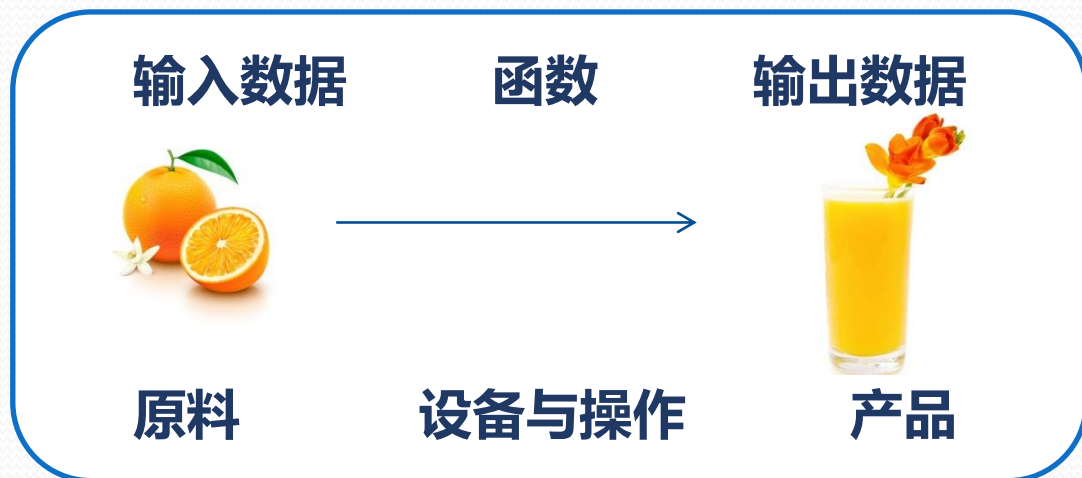
### C, Fortran

果汁生产线 JUICE PRODUCTS LINE



**数据：变量和数组**

**函数：逻辑，运算和操作**



## 面向对象编程 (工厂模式)

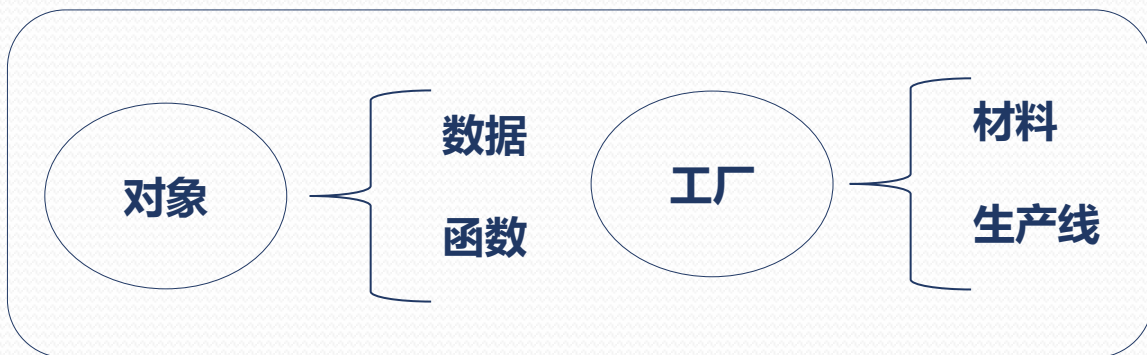
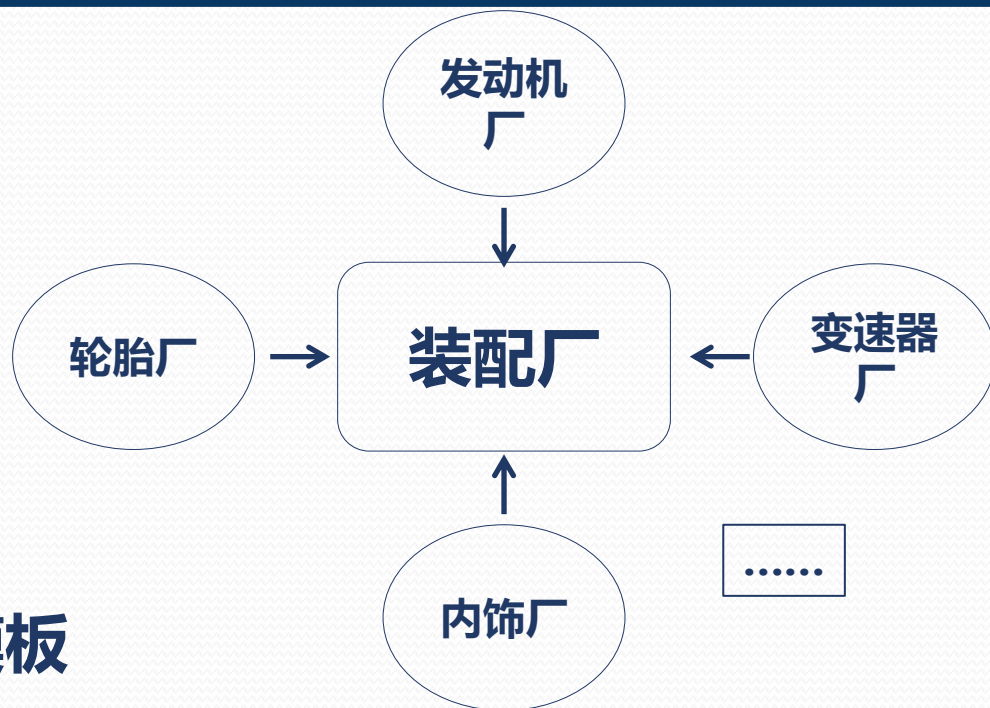
C++, Python

Class (类)  $\xrightarrow{\text{实例化}}$  Object (对象)  
 图纸 工厂

对象是类的实例 类是对象的模板

简单问题: 面向过程

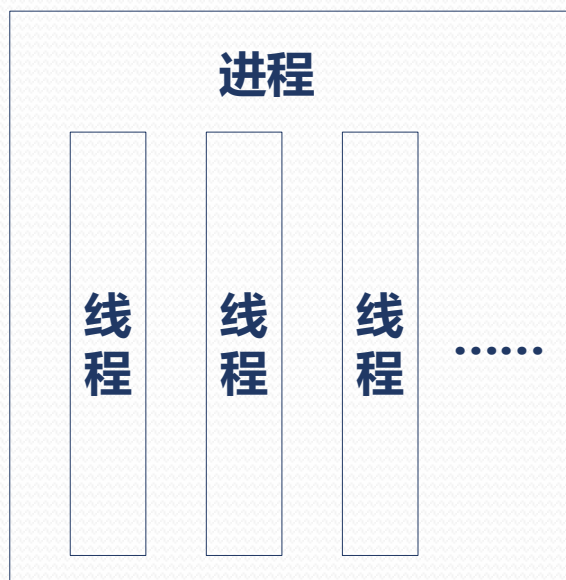
复杂问题: 面向对象



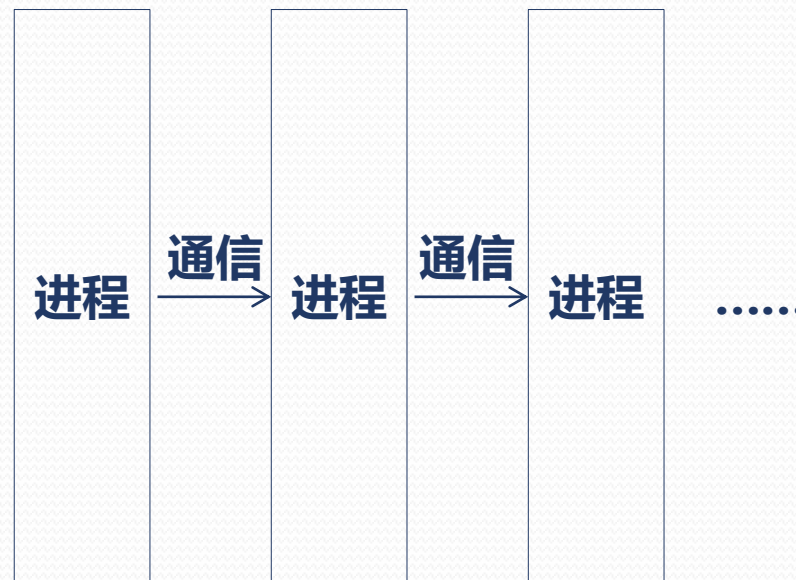
**Process: 一个进程就是一个任务，进程是资源分配的最小单位**

**Thread: 单一顺序的控制流，线程是程序执行的最小单位**

## CPU 编程模型

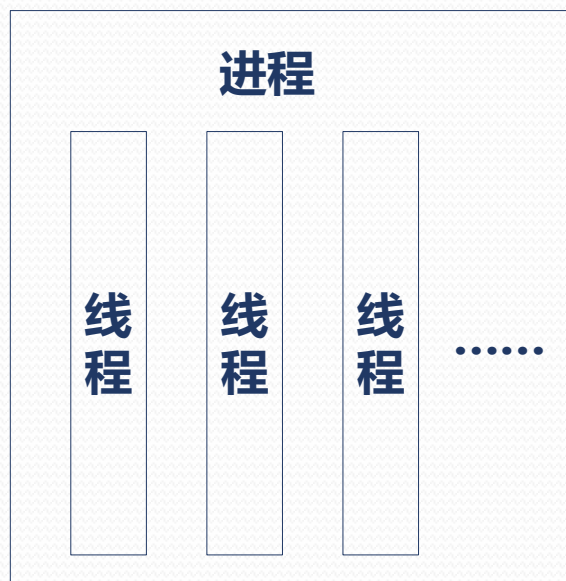


**OpenMP**



**MPI**

## GPU 编程模型 (NVIDIA GPU)



**CUDA/OpenACC**

**一个GPU一次运行一个任务 (进程)**

**(1) GPU的线程切换不耗时**

**(2) 线程数量不受GPU硬件的核数限制**

**(3) 所有线程执行同样的命令，但是处理不同的数据**

```
for (unsigned int i=0; i<N; i++)
    c[i] = a[i] + b[i]
```

CPU

```
c[thread_id] = a[thread_id] + b[thread_id]
```

GPU

```
for (unsigned int i=0; i<N; i+=64)
    if(thread_id+i<N)
        c[thread_id+i] = a[thread_id+i] + b[thread_id+i]
```

GPU

*Talk is cheap. Show me the code.*”, Linus Torvalds

## OpenMP

```
#pragma omp parallel num_threads(3)
{
  #pragma omp for
  for(int i=0; i<9; ++i)
  {
    nthreads = omp_get_num_threads();
    threads_id = omp_get_thread_num();
    printf("%d,%d,%d\n", nthreads, threads_id, i);
  }
}
```

**g++ filename.cc -fopenmp**

**OpenMP把循环计算分配到多个线程上进行执行**

## CUDA

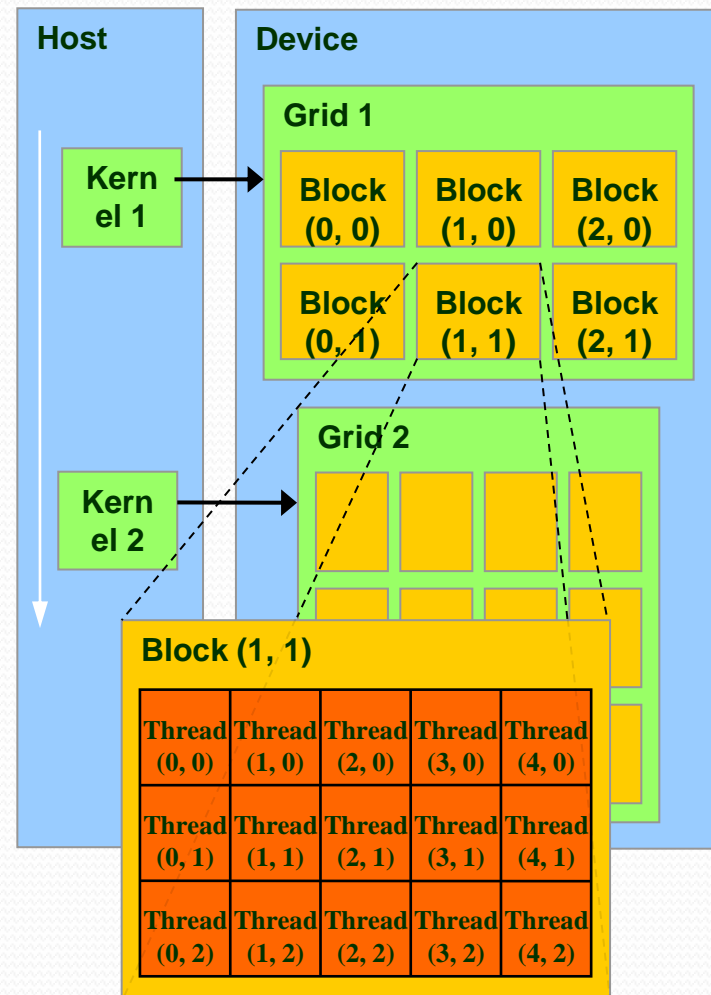
```
__global__ void myFirstKernel()
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if(idx<9)
    printf("thread %d \n",idx);
}

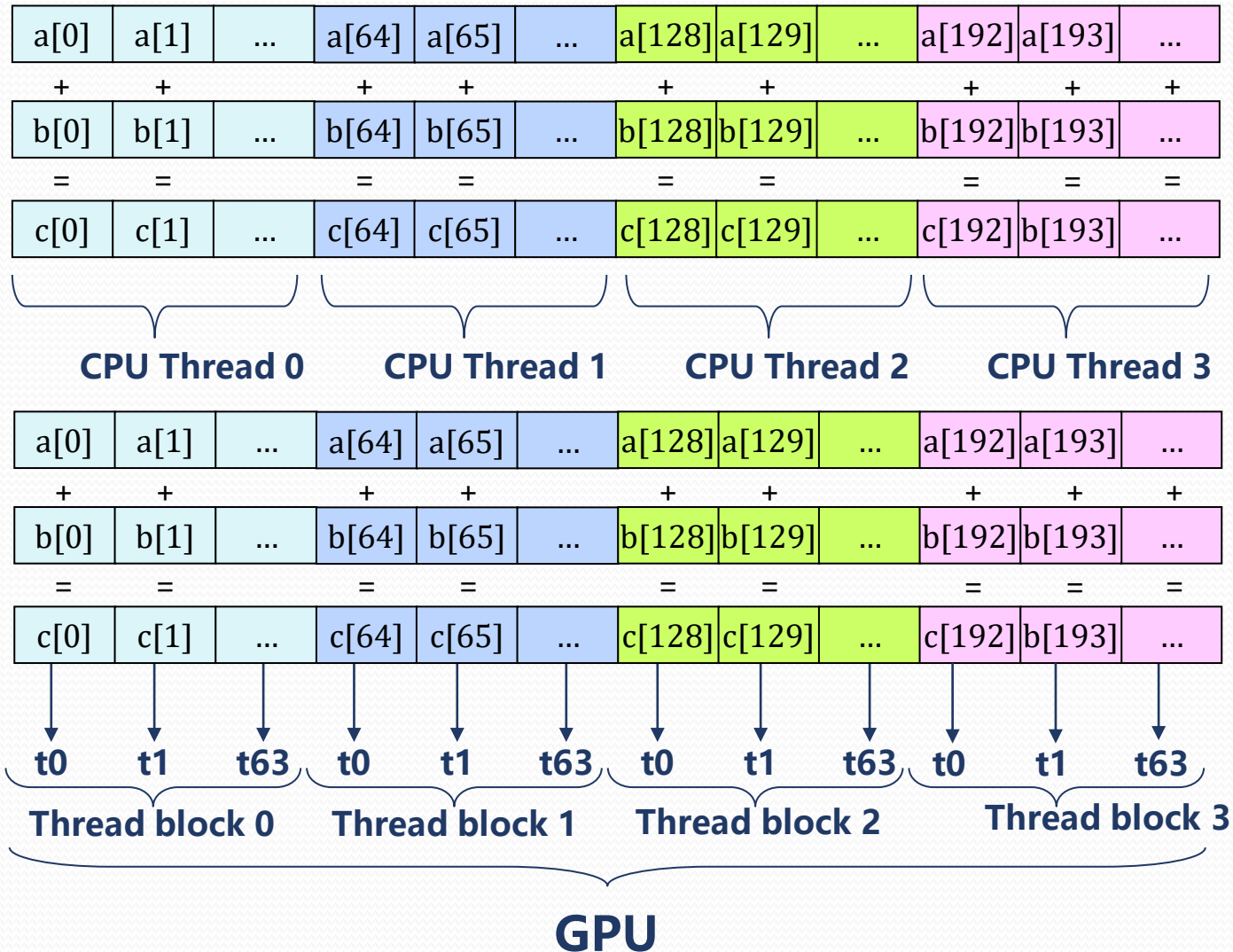
dim3 dimGrid(1);
dim3 dimBlock(32);
myFirstKernel<<< dimGrid, dimBlock >>>();
cudaThreadSynchronize();
```

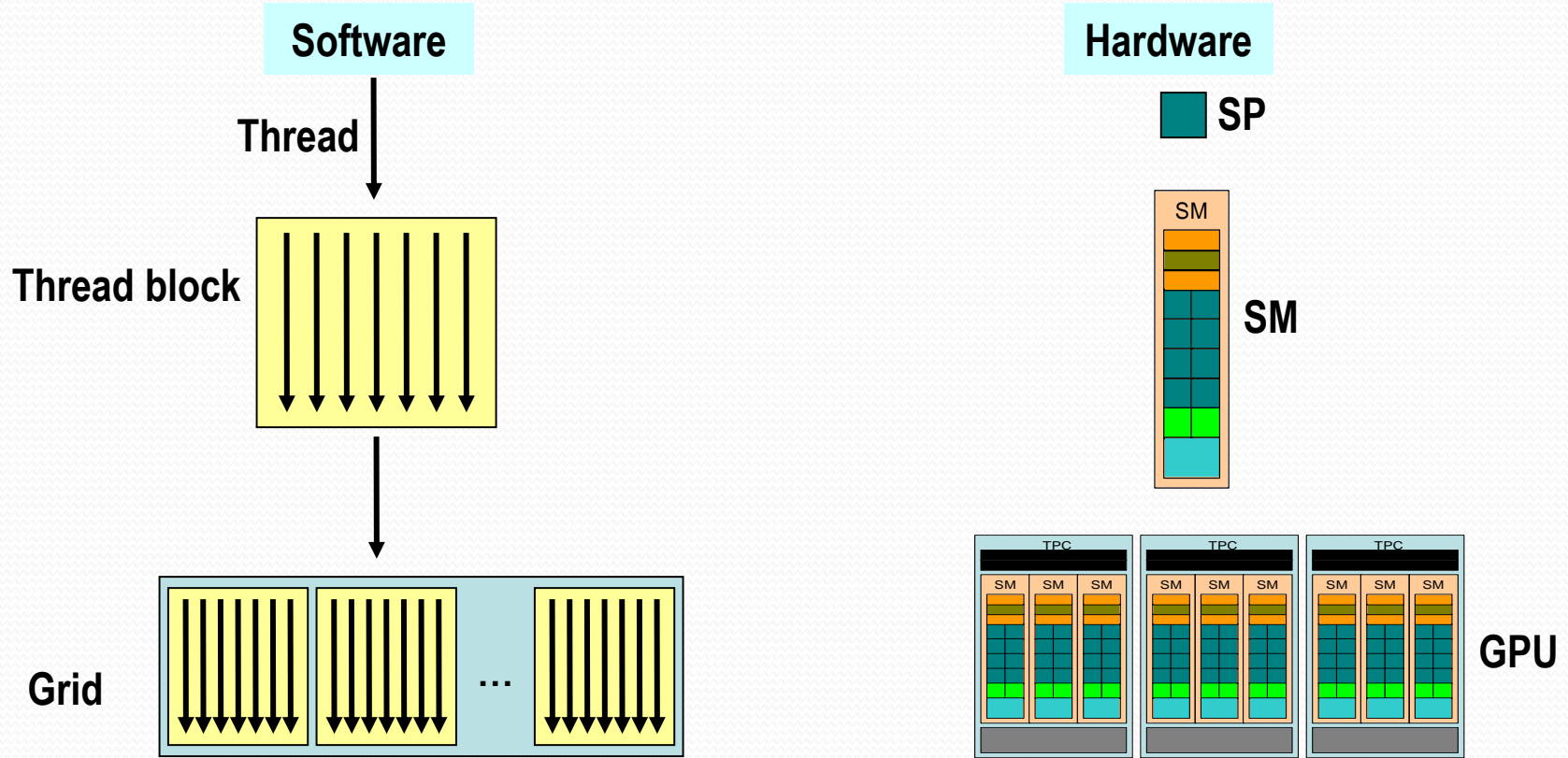
**nvcc filename.cu**

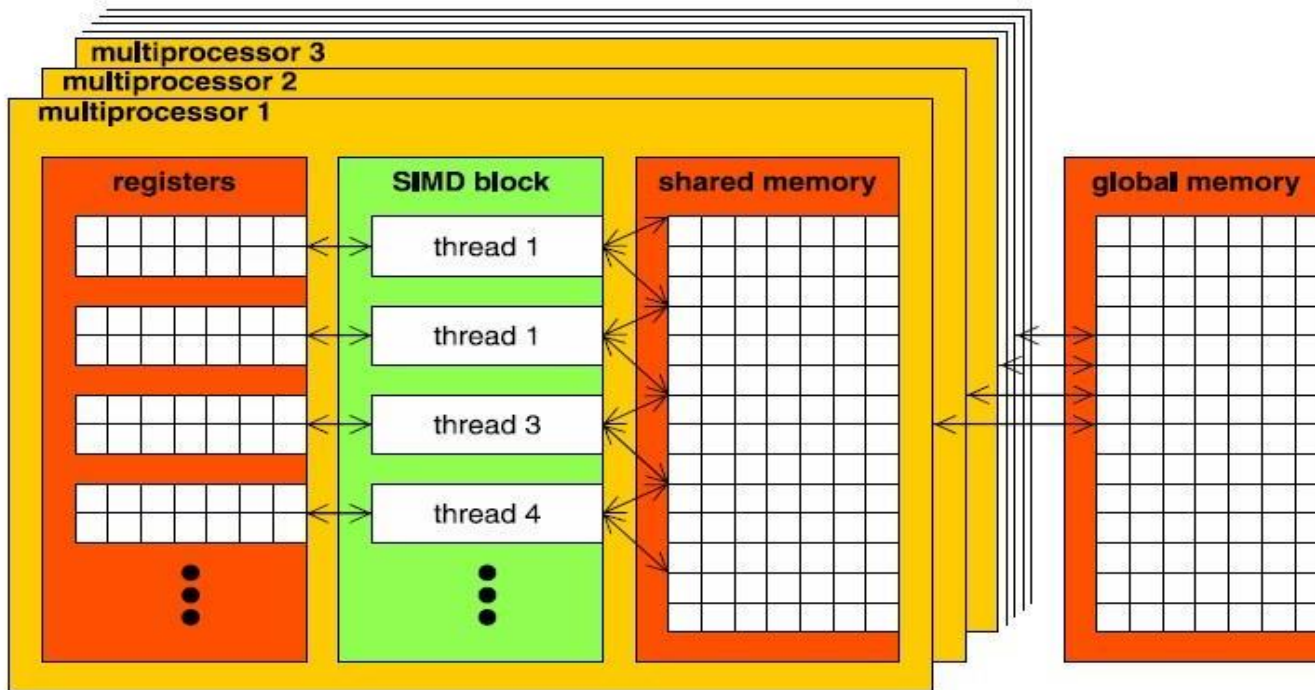
**CUDA 是SIMD，即单指令多数据  
每个线程执行同样的计算流程**

- **Thread:** 并行的基本单位
- **Thread block:** 互相合作的线程组
  - Cooperative Thread Array (CTA)
  - 允许彼此同步
  - 通过快速共享内存交换数据
  - 以1维、2维或3维组织
  - 最多包含1024个线程
- **Grid:** 一组thread block
  - 以1维或2维组织
  - 共享全局内存
- **Kernel:** 在GPU上执行的核心程序
  - One kernel  $\leftrightarrow$  one grid









还有两种可以被所有线程访问的只读存储器：常数存储器（constant memory）和纹理存储器（Texture memory）

- 常数存储器中的数据位于显存，但拥有缓存加速。常数存储器的空间较小（只有64KB）
- 纹理存储器不是一块专门的存储器，而是涉及到显存，两级纹理缓存，纹理拾取单元的纹理流水线

## CUDA 内核和线程

- 定义: 设备=GPU; 主机= CPU
- 内核= 运行在设备上的函数
- 程序的并行部分在设备上执行,这称为内核
- 一次执行一个内核, 内核以Grid执行
- 一组线程阵列执行同一CUDA 内核
- 所有线程运行相同的代码, 每个线程都有自己的ID, 用来计算内存地址和做控制决策

## 函数必须用限定词声明

- `__host__`: 只能被CPU执行, 从主机调用
- `__global__`: 在主机 (CPU) 代码中调用, 不能从设备 (GPU) 代码中调用必须返空void
- `__device__`:在其他GPU代码中调用,不能从主机CPU代码中调用

## GPU显存管理

- `cudaMalloc()`: 显存上分配数组空间
- `cudaMemcpy()`: host 和 device 之间数据拷贝

```
int main( int argc, char** argv)
{
int *h_a; // pointer for host memory
int *d_a; // pointer for device memory
int numBlocks = 8; // define grid size
int numThreadsPerBlock = 8; // define block size

// Part 1 of 5: allocate host and device memory
size_t memSize = numBlocks * numThreadsPerBlock * sizeof(int);

h_a = (int *) malloc(memSize);
cudaMalloc( (void **) &d_a, memSize );

// Part 2 of 5: launch kernel
dim3 dimGrid(numBlocks);
dim3 dimBlock(numThreadsPerBlock);
myFirstKernel<<< dimGrid, dimBlock >>>( d_a );

cudaThreadSynchronize(); // block until the device has completed
```

```
// check if kernel execution generated an error
checkCUDAError("kernel execution");

// Part 4 of 5: device to host copy
cudaMemcpy( h_a, d_a, memSize, cudaMemcpyDeviceToHost );
// Check for any CUDA errors
checkCUDAError("cudaMemcpy");

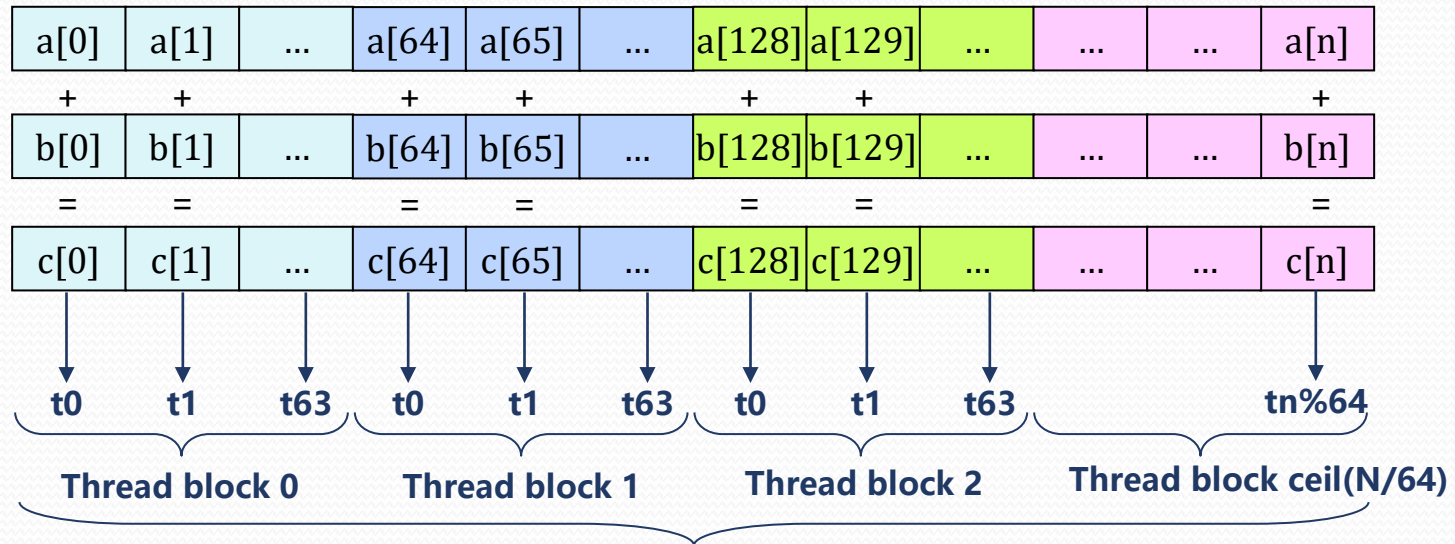
// free device memory
cudaFree(d_a);
// free host memory
free(h_a);
// If the program makes it this far, then the results are correct and
// there are no run-time errors. Good work!
printf("Correct!\n");
return 0;
}

__global__ void myFirstKernel(int *d_a)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    d_a[idx] = 1000*blockIdx.x + threadIdx.x;
}
```

- 数组的显存空间分配
- 初始化显存中的数组的数据，比如从内存拷贝到显存
- 定义grid，调用内核函数
- 拷贝显存数据到内存，进行进一步的处理，比如输出到文件或屏幕

## Example 1: Addition (N elements)

- 1 thread for one addition
- 64 threads per block
- $\text{ceil}(N/64)$  thread blocks



Addition (CPU code)

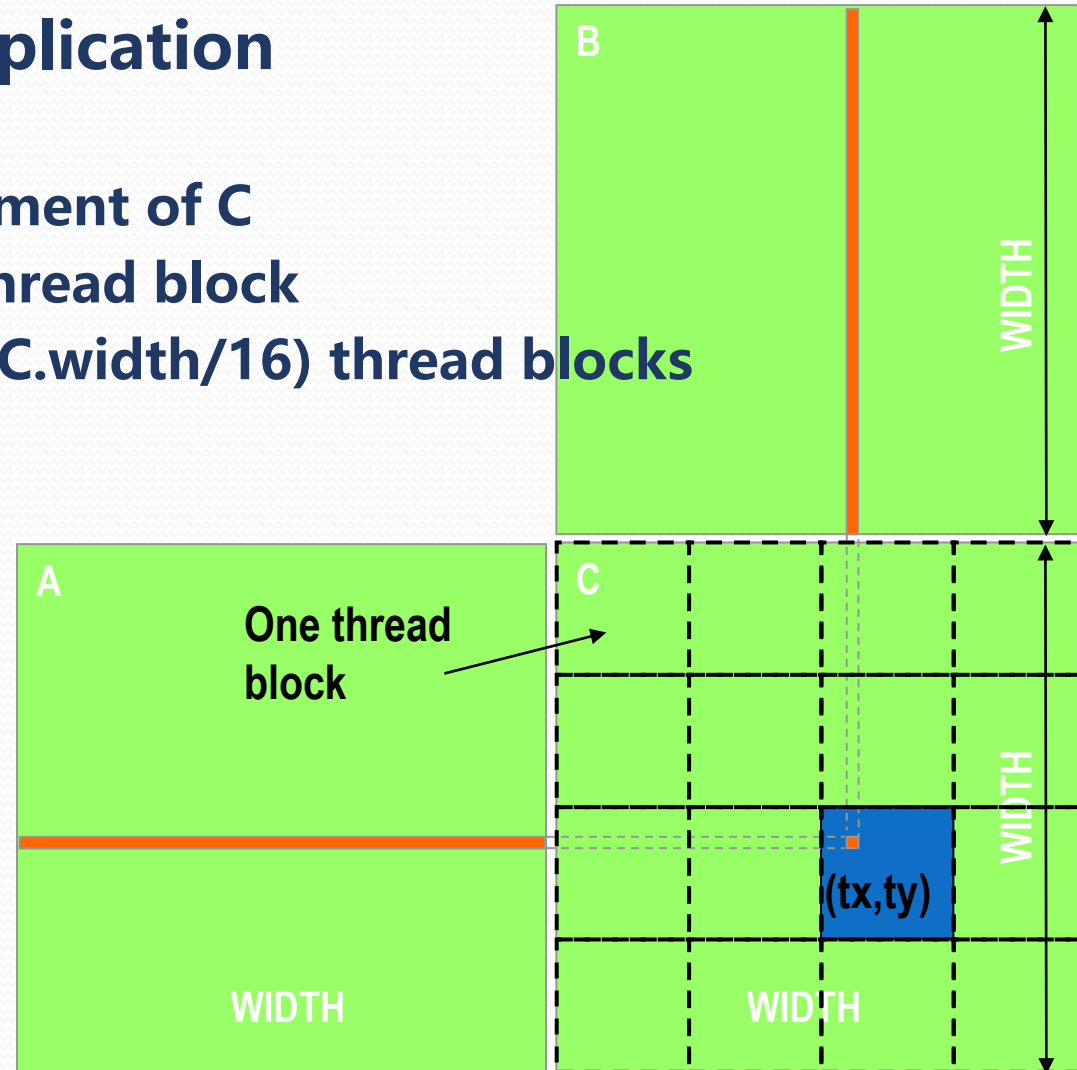
```
for (i=0;i<N;i++){  
    h_c[i] = h_a[i] + h_b[i];}
```

Addition(GPU code)

```
i = blockIdx.x*blockDim.x + threadIdx.x;  
h_c[i] = h_a[i] + h_b[i];
```

## Example 2: Matrix multiplication

- $C = A \times B$
- One thread for one element of  $C$
- $16 \times 16$  threads in one thread block
- $\text{ceil}(C.\text{width}/16) \times \text{ceil}(C.\text{width}/16)$  thread blocks



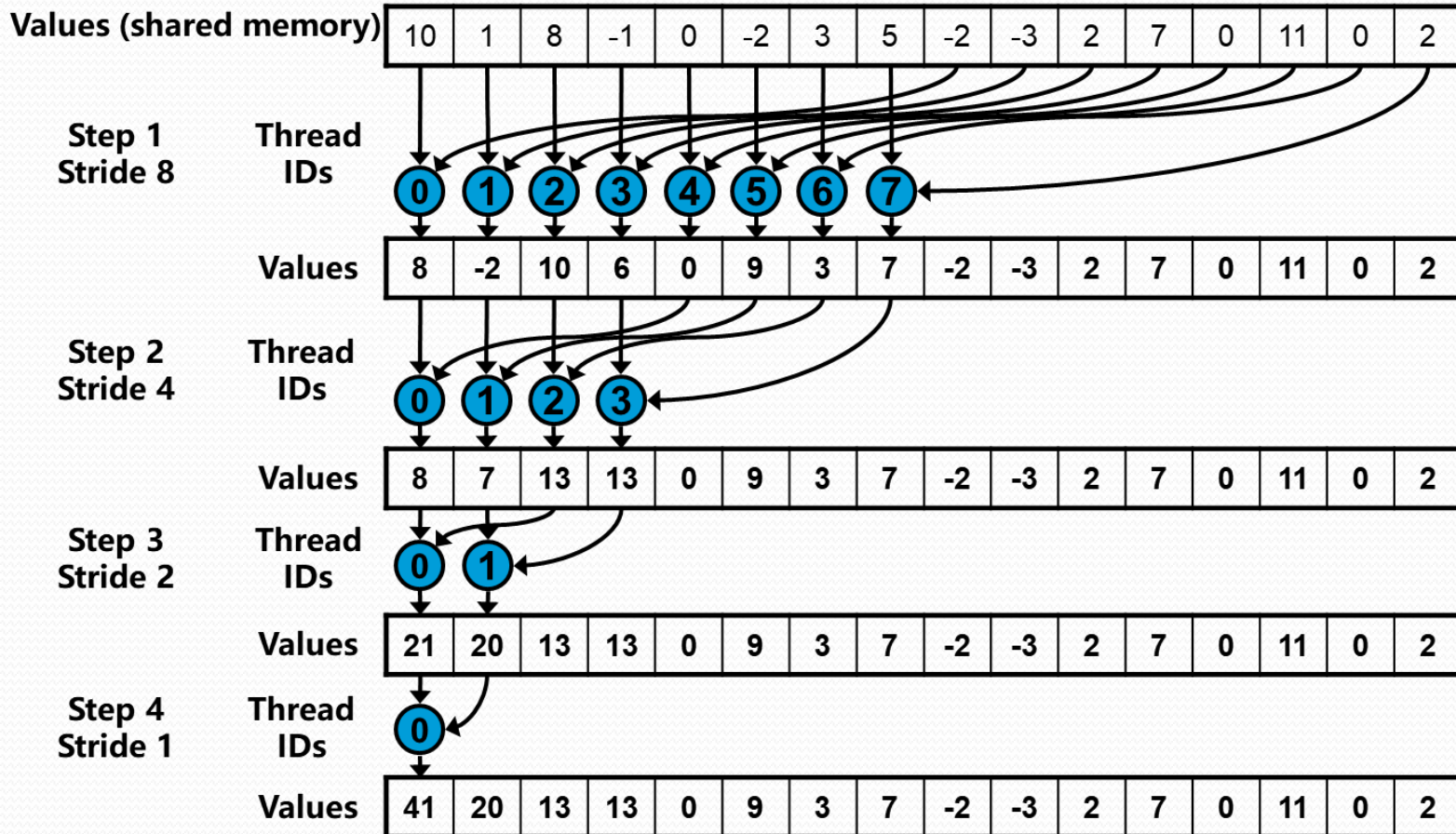
Matrix multiplication (CPU code)

```
for (i = 0; i < ha; ++i){
    for (j = 0; j < wb; ++j){
        sum = 0;
        for (k = 0; k < wa; ++k){
            sum += (float)h_a[i*wa + k]*(float)h_b[k*wb + j];}
        h_c[i*wb+j] = (float)sum;}}
```

Matrix multiplication (GPU code)

```
row = blockIdx.x*blockDim.x + threadIdx.x;
col = blockIdx.y*blockDim.y + threadIdx.y;
if(row<ha&&col<wb){
    for (i = 0; i < wa; i++) {
        sum += d_a[row*wa + i]*d_b[i*wb + col];}
    d_c[row*wb + col] = sum;}
```

## Example 3: Parallel Reduction



## Parallel Reduction (CPU code)

```
collection = 0.0;
for (i=0;i<N;i++){
    collection = collection + h_a[i] }
```

## Parallel Reduction (GPU code)

```
int offs = blockDim.x >> 1;
while (offs > 0) {
    if (threadIdx.x < offs) {
        d_a[threadIdx.x] += d_a[threadIdx.x + offs];
        offs >>= 1;
        __syncthreads(); }
collection = d_a[0];
```

**Thank you for your  
attention !**