一，GPU结构

1. GPU核心部分，有并行计算引擎，主要用于执行并行计算任务。
    - GPU核心通常由大量的流处理器（Streaming Processors）和寄存器文件（Register File）等组成
        - 计算单元（Compute Units）：GPU核心通常由多个计算单元组成，**每个计算单元包含多个流处理器（SM）和寄存器文件等。**
        - 流多处理器（Stream Multiprocessor）：可以同时执行多个线程块，并且具有自己的共享内存、寄存器和控制单元。一个 GPU 设备可以包含多个流多处理器，不同的 GPU 设备具有不同的流多处理器数量和组织形式。
            - 流处理器：是**执行计算任务和图形渲染任务的最基本组成单元**，每个流处理器都包含**一个算术逻辑单元（ALU）和寄存器文件**等，可以独立执行可编程的计算任务。流处理器数量即CUDA核的数量
        - 寄存器文件：用于存储计算任务和图形渲染任务所需的数据
        - 算术逻辑单元（ALU）：用于执行算术运算和逻辑运算（ALU可以通过编程进行控制和配置，以实现不同的运算和计算任务。）
    - 图形处理器核心（Graphic Processing Cores）
        - 图形处理管道（Graphics Pipeline）：GPU的工作流程，包括几何处理、光栅化、像素处理、输出等多个阶段，用于将图形数据转化为可视化的图像。
        - 纹理单元（Texture Units）：专用硬件单元，用于快速处理和压缩纹理数据。
        - 采样器（Sampler）：处理图像纹理的模块，主要包括过滤、抗锯齿和各种变换等功能。
        - 像素处理器（Pixel Processor）：对图像的颜色、亮度、对比度等进行处理，最终输出模拟的图像。
2. 内存控制器（Memory Controller）：负责管理内存的访问、读写和缓存等。其通常包括内存接口、内存控制器芯片、数据总线等（GPU内存主要用于存储代码、程序数据和其他GPU需要的数据；显存则是用于存储图像和视频数据）
    - 较大内存带宽（Memory Bandwidth）：指的是内存传输数据的速率，通常以每秒传输的数据量（单位为GB/s）来衡量，可以支持GPU高速的数据传输和访问。影响因素：
        - 内存类型：DDR，GDDR，HBM，HMC
        - 内存接口宽度（Memory Interface Width）：是指连接**GPU和内存之间的数据通道的宽度**，通常以位（bit）为单位来表示（如GTX 1080采用了256-bit）
        - 内存时钟频率（Memory Clock）：是指**内存芯片在单位时间内的运行速度**，通常以MHz为单位表示（如DDR4内存的标准频率为2133MHz）

        - 内存总线速率（Memory Bus Speed）：是指**内存控制器与内存之间的数据传输速度**，通常以MT/s（每秒传输的百万字节）为单位表示（内存总线速率的提升有可能会使内存延迟（Latency）提高，原因↓↓↓）
            - 时序限制：随着内存时钟频率的提升，内存访问所需的时序可能会更加严格。为保证内存正常工作，会限制预充电时间（Precharge Time）和恢复时间（Recovery Time）
            - 内存结构：内存的行地址和列地址通常需要多次转换才能得到实际的内存地址，内存总线速率的提升可能会使这些转换更加频繁，从而增加内存延迟。
            - 数据量增加：内存总线速率的提升可以传输更多的数据，同时也需要更多的时间来处理这些数据。
    - 划分内存区域：GPU内存控制器可以将GPU内存划分为不同的存储区域，以实现不同的数据访问需求，例如静态存储区、动态存储区等
        - 静态存储区
            - 全局内存（Global Memory）：是GPU上最大的内存池之一，可以存储大量的数据，并且可以被所有的线程访问。全局内存的读写速度相对较慢
            - 常量内存（Constant Memory）：常量内存的读取速度更快，并且具有只读的属性（常量内存适合存储不需要被修改的数据，如常量数组和常量参数等）
        - 动态存储区
            - 共享内存（Shared Memory）：是GPU内部用于在同一线程块内多个线程之间共享数据的一种存储区域（速度快，存量小）
            - 寄存器（Register）：用于存储线程执行时需要使用的数据和变量（个线程只有有限的寄存器数量）
    - 内存映射：将GPU内存映射到CPU的地址空间中，以便CPU可以直接访问GPU内存。

二，CUDA编程模型

> 一种通用并行计算平台和编程模型，它将GPU架构建模为多核结构（是指将GPU核心中的流处理器组织成多个处理核心，并具有统一的内存空间和调度器，以支持更高的并行度和计算性能），将并行线性抽象为层次化的线性结构。

CUDA把GPU分成多个线程块（block），每个线程块包含多个线程（thread）。每个线程块运行在一个多处理器（multiprocessor）上，每个多处理器可以同时执行多个线程块。线程块可以通过共享内存（shared memory）来协作计算，加速计算效率。

基本流程

1. 分配GPU内存

```
float *d_a, *d_b, *d_c;
int size = N * sizeof(float);
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
```

3. 将数据从CPU内存拷贝到GPU内存中
4. 定义线程块和线程的数量
5. 编写CUDA核函数，并在核函数中定义线程的计算任务
6. 调用核函数

7. 将计算结果从GPU内存中拷贝到CPU内存中
8. 释放GPU内存

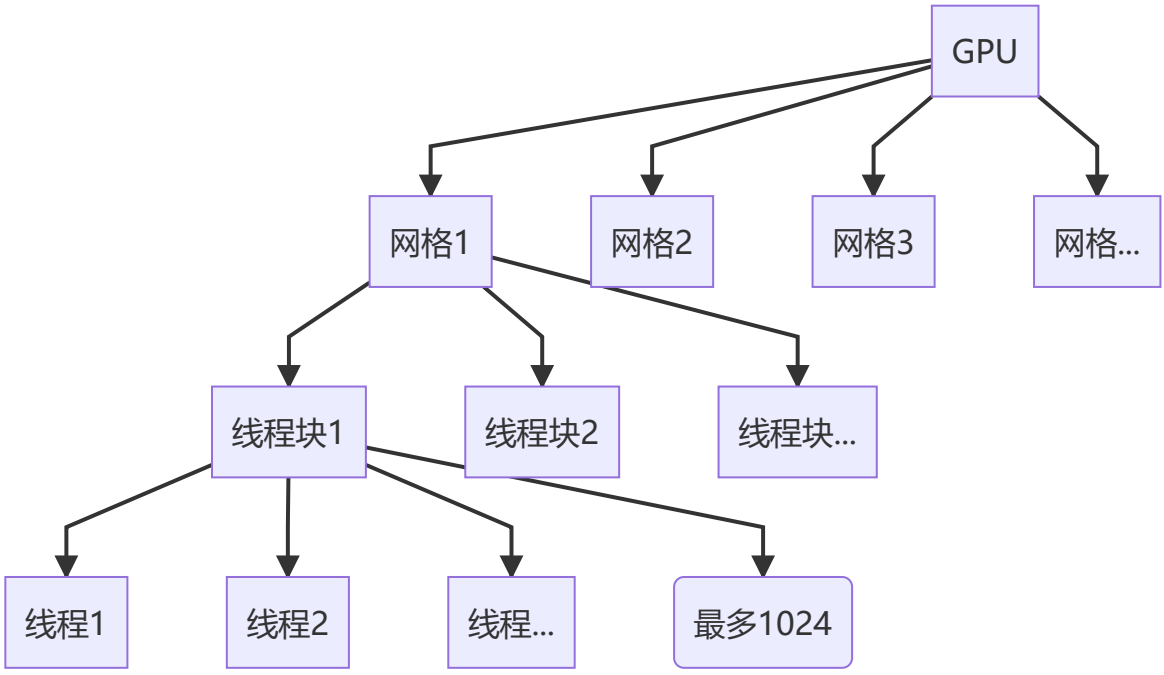
```
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

核函数及调用

- 线程块和线程是指GPU上运行的并行执行的最小单位，它们是独立的计算单元，每个线程可以执行不同的任务。**线程块和线程的数量可以在启动核函数时进行设置。**
- 核函数是在GPU上执行的函数，由__global__关键字定义。核函数是GPU上执行的**计算任务**，每个线程都执行相同的核函数，但是使用**不同的线程ID**来区分不同的线程。
- gridDim.x（网格块数）blockDim.x（当前块线程数）blockIdx.x（当前块索引）threadIdx（当前块中的线索引）

```
__global__ void kernel_name(argument_list){
    //__global__关键字表示该函数在GPU上执行并可全局调用（只能返回void类型），kernel_name表示函数名字
    // kernel code here
    int i = blockDim.x * blockIdx.x +threadIdx.x;
    //索引当前线程数=此线程块前的线程数+当前线程块的线程索引
}

int main(){
    kernel_name<<<gridDim.x, blockDim.x>>>();//配置GPU（初始化），使用<<<...>>>语法来启动核函数
    cudaDeviceSynchronize();//核函数启动方式为异步，是GPU代码执行完，再在CPU上恢复执行
}
```



> 核(块)能够并行运算，线程不行，即<<<一次运算， 次数为线程数>>>

矩阵加法

矩阵一的每个元素加矩阵二的每个元素 = 矩阵三（相当于c = numpy.add(a, b)）

```
//单个元素加法
__global__ void add_matrix(float *a, float *b, float *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}
```

分配显存

CPU内存分配(动态分配内存时不能写成函数)

```
const int N = 2 << 20;
int *a;
cudaMallocManaged(&a, size);//分配统一内存，CPU、GPU都能用，自动判断
for(int i = 0; i < N; i++){
    a[i] = i;
}
size_t size = N * size(int);
a = (int*)malloc(size);
free(a);
//`size_t`是一个无符号整数类型，通常用来表示内存块的大小、数组下标、循环计数器等。能够容纳系统中最大的内存块大小。
//`%zu`是C语言中用来格式化`size_t`类型变量的占位符
如: int numElements = num;
    size_t size = numElements * sizeof(datatype);
```

```
//在CPU端给ABC三个向量申请存储空间(用指针申请)
float *h_A = (floot *)malloc(size);
float *h_B = (floot *)malloc(size);
float *h_C = (floot *)malloc(size);
//初始化
for(int i=0; i < numElements; ++i){
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
}
//`rand()` 函数生成伪随机数（每次运行程序生成的随机数序列都是一样的）
//`RAND_MAX` 是 C 语言标准库中定义的一个常量，它是一个整数，代表 `rand()` 函数返回的随机整数的范围上限
```

GPU内存分配

```
int *b;
int i = blockIdx.x * blockDim.x + threadIdx.x;
if(i < N){
    b[i] *= 2
}
cudaMallocManaged(&b, size);
kernelname<<<dimGrid, dimBlock>>>(b, N);
cudaFree(b);
```

`cudaMalloc` 和 `cudaMallocManaged` 都是 CUDA 库中用于申请内存的函数

- cudaMalloc，用于在 GPU 的全局内存中分配指定大小的内存空间，并返回该内存空间的地址(仅是在 GPU 全局内存)

```
cudaError_t cudaMalloc(void** devPtr, size_t size);
//cudaError_t 类型的错误码，如果分配成功则返回 cudaSuccess
//devPtr 是一个指向指针的指针，用于存储分配的内存空间的地址
```

- cudaMallocManaged，用于在 GPU 的全局内存中分配指定大小的内存空间（ CPU 和 GPU 可以共享这段内存空间)

```
MallocManaged(void** devPtr, size_t size, unsigned int flags =
cudaMemAttachGlobal);
//flags 表示内存分配的标志
//cudaMemAttachGlobal 是 CUDA 中定义的一种枚举类型，用于设置 CUDA 统一内存（Unified
Memory）分配的策略：
1. cudaMemAttachGlobal: 指定内存空间将同时在 CPU 和 GPU 中可见，且该内存空间会被所有 GPUs
（它是一个 `int` 型数组，用于存储当前系统中所有可用 GPU 设备的设备号。使用 gpus 数组，可以获
取当前系统中可用的 GPU 设备数量以及设备号） 共享，可以通过应用程序中的任何一个 GPU 访问
2.cudaMemAttachHost: 指定内存空间在分配时只分配于 CPU 系统内存空间，而不在 GPU 设备内存空间
中分配，该内存空间只能被 CPU 访问
```

> cudaMallocManaged 函数申请的内存空间的管理存在一定的开销，通常不适用于大型数据的处理

```
//在 GPU 当中给 ABC 三个向量申请存储空间
float *d_A = NULL;
float *d_B = NULL;
float *d_B = NULL;
cudaMalloc((void **)&d_A, size);
cudaMalloc((void **)&d_B, size);
cudaMalloc((void **)&d_C, size);

//把数据 AB 从 CPU 内存当中复制到 GPU 显存当中
printf("Copy input data from the host memory\n ");
cudaMemcpy(d_A, h_A, size cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size cudaMemcpyHostToDevice);
```

```
//cudaMemcpy用于在 CPU 内存和 GPU 内存之间进行数据的拷贝操作
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind
kind);
//dst 表示目标内存地址，src 表示源内存地址，count 表示需要拷贝的数据大小（以字节为单位），
kind 表示数据传输方向，它是一个枚举值包括：
- `cudaMemcpyHostToHost`：表示从 CPU 内存复制到 CPU 内存。
- `cudaMemcpyHostToDevice`：表示从 CPU 内存复制到 GPU 内存。
- `cudaMemcpyDeviceToHost`：表示从 GPU 内存复制到 CPU 内存。
- `cudaMemcpyDeviceToDevice`：表示从 GPU 内存复制到 GPU 内存。
```

三，原函数

```
#include<stdio.h>
#include<cuda_runtime.h> //导入CUDA的运行时库

//A+B+C
__global__ void vectorAdd(const float *A, const float *B, float *C, int
numElements){
    int i = blockDim.x * blockIdx.x +threadIdx.x;
    if(i < numElements){
        c[i] = A[i] + B[i];
    }
}


int main(void){
    //A/B/C元素总数
    int numElements = 50000;
    size_t size = numElements * sizeof(float);
    printf("Vector addition of %d elements.\n", numElements );

    //在CPU端给ABC三个向量申请存储空间
    float *h_A = (floot *)malloc(size);
    float *h_B = (floot *)malloc(size);
    float *h_C = (floot *)malloc(size);
    //初始化
    for(int i=0; i < numElements; ++i){
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }

    //在 GPU 当中给 ABC 三个向量申请存储空间
    float *d_A = NULL;
    float *d_B = NULL;
    float *d_B = NULL;
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    //把数据 AB 从 CPU 内存当中复制到 GPU 显存当中
    printf("Copy input data from the host memory\n ");
    cudaMemcpy(d_A, h_A, size cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(d_B, h_B, size cudaMemcpyHostToDevice);

    //执行GPUkernel函数
    int threadsPerBlock = 256;
    int blockPerGrid = (numElements + threadsPerBlock -1)/threadsPerBlock;
    vectorAdd <<< blockPerGrid, threadsPerBlock >>> (d_A, d_B, d_C,
numElements);
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    for(int i = 0; icnumElements; ++i){
        if(fabs(h_A[i] + h_B[i] - h_C[i]) > 1e - S){
            fprintf(stderr, "Result verification failed at element %d!\n", i);
            exit(EXIT_FAILURE);
        }
    }
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
}
```