CUDA 并行计算基础

> Computer Unifiied Device Architecture

CUDA C/C++

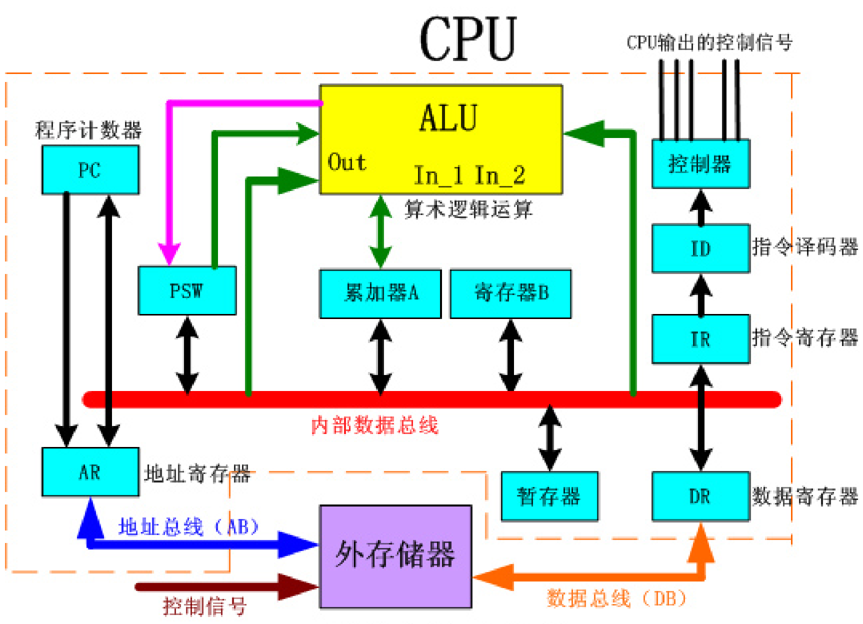
- 基于C/C++的编程方法（主要是C）
- 支持异构编程的扩展方法
- 简单明了API，能够轻松的管理存储系统

CUDA 支持语言：C/Cpp/Python/Java

硬件层次：

- CPU/GPU 本质区别：CPU顺序执行，GPU并行执行
- 术语：
  - Host CPU和内存
  - Device GPU和显存
- CPU结构：

CPU内部结构图

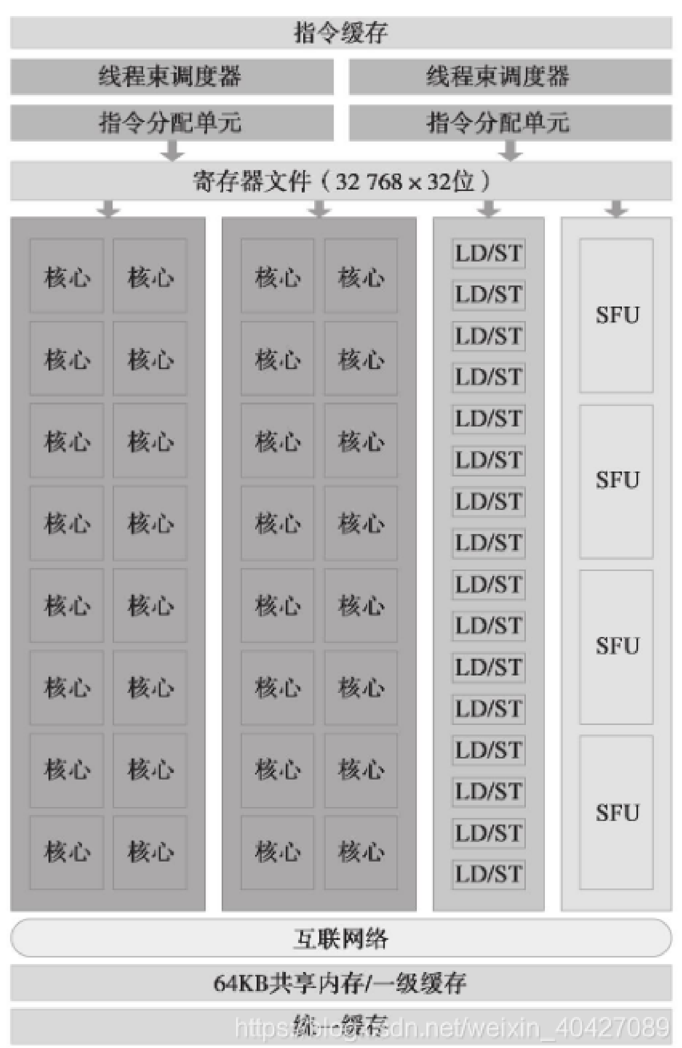


CPU内部结构框图

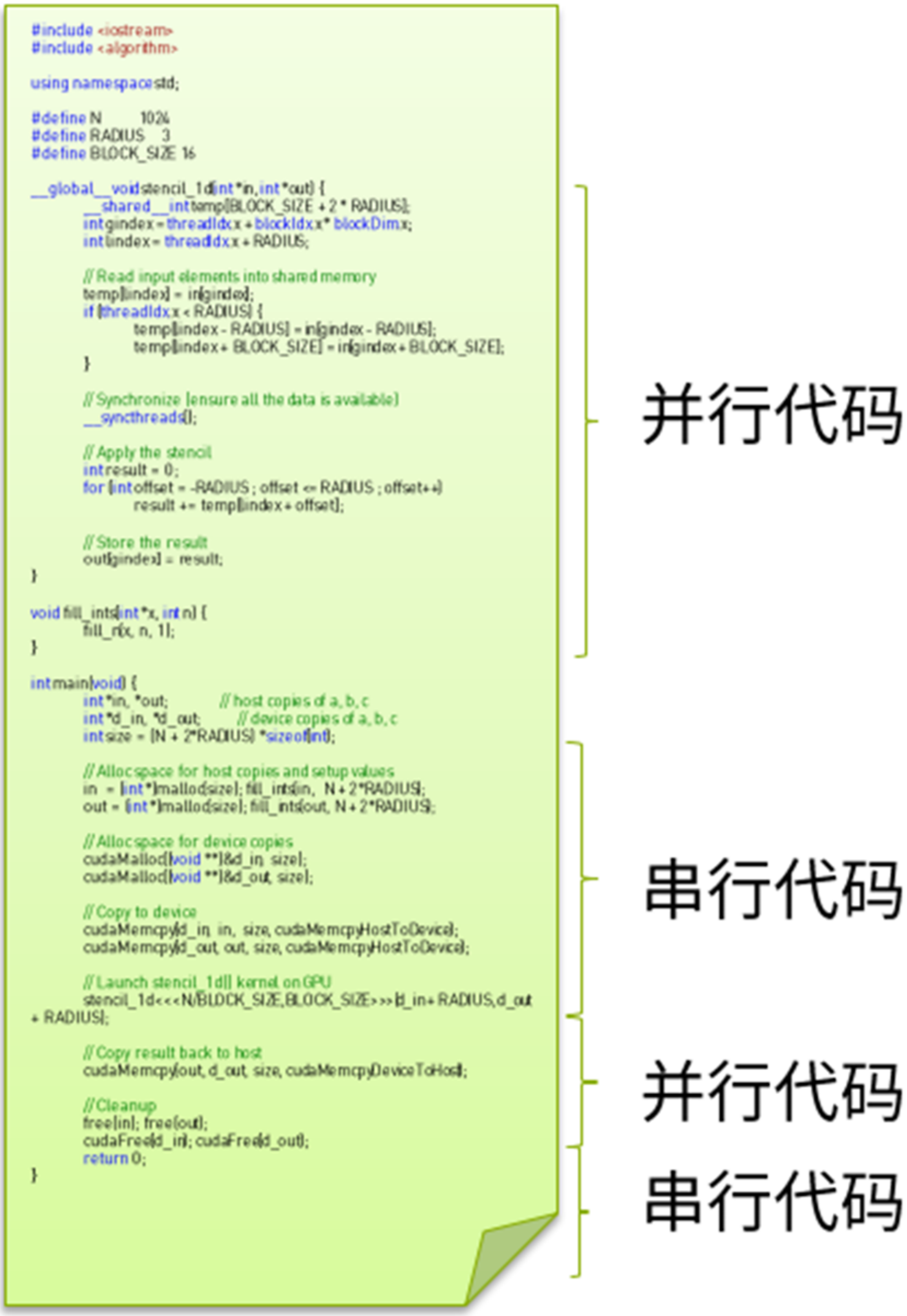
> 存储单元占用大多位置，核只占用两边，剩下的就是控制器

- GPU结构：

> 绿色的部分（CUDA core）用于执行计算，蓝色的部分表示各级存储空间
>
> 每个存储单元（SM: stream multiprocessor 流多处理器）包含四个结构（每个计算核心处理不同任务），寄存器、调度器、缓存
>
> 在硬件底层执行时是 32thread = 1warp



CUDA 异构计算及处理流程



```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;           // host copies of a, b, c
        int *d_in, *d_out;       // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,d_out
+ RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```


> kernel函数（并行代码）
>
> main函数中有串行代码（初始化、申请内存、数据传输、设置）：调用核函数并执行

代码执行顺序：

1. 把输入数据CPU内存复制到GPU显存
    - 在CPU上初始化程序（初始化数据、申请空间）
    - 复制传输
2. 在执行芯片上缓存数据，加载GPU程序并执行（并行执行）
    - GPU较慢显存—>较快memory多核计算
3. 将结果从GPU显存中复制到CPU内存（返回）

CUDA 线程层次

Grid – 一维或多维线程块(block)

　　一维 二维 或 三维

Block – 一组线程

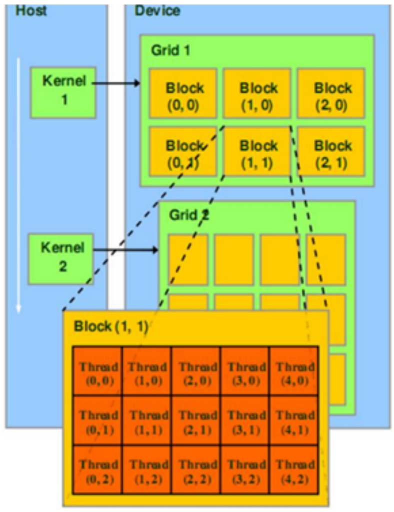
　　一维，二维或三维

　　一个Grid里面的每个Block的线程数是一样的

　　block内部的每个线程可以:

　　　　同步　synchronize

　　　　访问共享存储器 shared memory



> 每段线程执行完要同步，否则CPU、GPU速度不并行

如何将线程编号对应到每份数据中:

每8个thread对应1个block

Idx ~ index

- **声明**
  - global, device, shared, local, constant
- **关键词**
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads
- **运行期API**
  - Memory, symbol, execution management
- **函数调用**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

关键字:

```
#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_tsize = 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);
    floot *h_C = (floot *)malloc(size);
    //初始化
    for(int i=0; i < numElements; ++i){
        h_A[i] = rand()/(floot)RAND_MAX;
        h_B[i] = rand()/(flout)RAND_MAX;
    }

    //在 GPU 当中给 ABC 三个向量申请存储空间
    float *d_A = NULL;
    floot *d_B = NULL;
    floot *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);
}
```