[CUDA学习入门（一）（包含CUDA安装和相关基础知识）-CSDN博客](https://blog.csdn.net/weixin_44222088/article/details/135694581?ops_request_misc=&request_id=&biz_id=102&utm_term=cuda%E5%AD%A6%E4%B9%A0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-0-135694581.142^v100^pc_search_result_base1&spm=1018.2226.3001.4187)

#include <iostream>

#include <algorithm>

using namespace std;

#define N 1024

#define RADIUS 3

#define BLOCK\_SIZE 16

\_\_global\_\_ void idnt(int \*in, int \*out) {

\_\_shared\_\_ int temp[BLOCK\_SIZE + 2 \* RADIUS]; // 共享内存，大小为 BLOCK\_SIZE 加上两个半径

int index = threadIdx.x + blockIdx.x \* blockDim.x; // 计算当前线程输送的全局索引

int tempIndex = threadIdx.x + RADIUS; // 计算临时数组的索引

// 将元素读入共享内存

temp[tempIndex] = in[index];

// 边界条件处理

if (threadIdx.x < RADIUS) {

temp[tempIndex - RADIUS] = (index >= RADIUS) ? in[index - RADIUS] : 0; // 处理左边界

temp[tempIndex + BLOCK\_SIZE] = (index + RADIUS < N) ? in[index + RADIUS] : 0; // 处理右边界

}

// 同步以确保数据读取完成

\_\_syncthreads();

// 应用模板

int result = 0;

for (int offset = -RADIUS; offset <= RADIUS; offset++)

result += temp[tempIndex + offset]; // 计算结果

// 存储结果

out[index] = result;

}

void fill\_ints(int \*x, int n) {

fill\_n(x, n, 1); // 初始化数组为1

}

int main(void) {

int \*in, \*out; // 主机上的输入和输出指针

int \*d\_in, \*d\_out; // 设备上的输入和输出指针

int size = N + 2 \* RADIUS; // 计算大小

// 为主机分配内存并设置值

in = (int \*)malloc(size \* sizeof(int));

fill\_ints(in, N + 2\*RADIUS);

out = (int \*)malloc(size \* sizeof(int));

// 为设备拷贝分配内存

cudaMalloc((void \*\*)&d\_in, size \* sizeof(int));

cudaMalloc((void \*\*)&d\_out, size \* sizeof(int));

// 拷贝到设备

cudaMemcpy(d\_in, in, size \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_out, out, size \* sizeof(int), cudaMemcpyHostToDevice);

// 在 GPU 上启动内核

idnt<<<N/BLOCK\_SIZE, BLOCK\_SIZE>>>(d\_in, d\_out);

// 拷贝结果回主机

cudaMemcpy(out, d\_out, size \* sizeof(int), cudaMemcpyDeviceToHost);

// 清理

free(in);

free(out);

cudaFree(d\_in);

cudaFree(d\_out);

return 0;

}

这段 CUDA 程序的主要功能是实现一个简单的模板操作，针对输入数组的每个元素，将其周围的值（受到一个指定半径的影响）进行求和，并将结果存储到输出数组中。这种处理类似于图像处理中的卷积运算。

### 整体功能

程序的整体功能是对一个一维数组进行扩展窗口的求和操作，窗口的大小由 `RADIUS` 控制。这种操作常见于信号处理、图像处理等领域。

### 模块作用

1. \*\*头文件和常量定义\*\*：

```cpp

#include <iostream>

#include <algorithm>

using namespace std;

#define N 1024

#define RADIUS 3

#define BLOCK\_SIZE 16

```

- 导入标准库并定义了一些常量，用于设置数据的大小和计算块的大小。

2. \*\*CUDA 内核 `idnt`\*\*：

```cpp

\_\_global\_\_ void idnt(int \*in, int \*out) {

// ...

}

```

- \*\*作用\*\*：该内核函数执行主要的计算逻辑。

- \*\*实现\*\*：

- 在共享内存中存储输入数组部分和边界数据。

- 使用同步机制确保所有线程都完成数据读入。

- 对于每个线程，计算其索引处的值与周围 `RADIUS` 的区域的和，存储在输出数组中。

3. \*\*填充数组的函数 `fill\_ints`\*\*：

```cpp

void fill\_ints(int \*x, int n) {

fill\_n(x, n, 1);

}

```

- \*\*作用\*\*：初始化输入数组，将其所有元素置为 1。

- \*\*实现\*\*：使用 `fill\_n` 函数简化赋值操作。

4. \*\*主函数 `main`\*\*：

```cpp

int main(void) {

// ...

}

```

- \*\*作用\*\*：程序的入口，负责输入输出数据的管理，包括内存分配、数据初始化、内核调用等。

- \*\*实现\*\*：

- 分配主机和设备的内存。

- 将主机内存数据复制到设备内存。

- 调用 CUDA 内核进行计算。

- 将结果从设备复制回主机。

- 释放所有分配的内存，确保没有泄漏。

### 实现流程

1. \*\*内存分配\*\*：

- 为主机和设备分配内存，在主机上分配输入和输出数组的内存。

2. \*\*数据初始化\*\*：

- 使用 `fill\_ints` 函数初始化输入数组的内容，使其每个元素都为 1。

3. \*\*设备内存拷贝\*\*：

- 将初始化后的输入数组复制到 GPU 的设备内存中，同时也分配设备的输出数组内存。

4. \*\*内核计算\*\*：

- 在 GPU 上启动 `idnt` 内核，按照块和线程的划分执行计算。

5. \*\*结果拷贝\*\*：

- 将 GPU 计算的结果从设备内存复制回主机内存。

6. \*\*内存清理\*\*：

- 释放所有在主机和设备上分配的内存，以防止内存泄漏。

### 总结

该程序利用 CUDA 的并行计算能力，针对一维数组实现了基于窗口求和的操作。通过合理的内存管理和线程同步，确保了在 GPU 上的高效计算。这种处理方式在数据分析、图像处理中有着广泛的应用。

看不懂的一些名词



[CUDA学习入门（二）（CUDA编译过程 & CUDA 线程层次）\_【cuda编程(2)】线程层级-CSDN博客](https://blog.csdn.net/weixin_44222088/article/details/135716596)

对实验项目的实战体验

根据您的需求，下面是一个简单的 C++ 项目代码示例，旨在探索异构计算的加速方式。这个项目将展示如何使用 CPU 和 GPU 进行并行计算，并进行简单的性能比较。我们将使用 CUDA 进行 GPU 编程，并通过 C++ 实现 CPU 计算。

### 项目结构

```

Heterogeneous\_Computing/

├── CMakeLists.txt

├── main.cpp

├── cpu\_computation.cpp

├── cpu\_computation.h

└── gpu\_computation.cu

```

### 1. `CMakeLists.txt`

```cmake

cmake\_minimum\_required(VERSION 3.10)

project(HeterogeneousComputing)

set(CMAKE\_CXX\_STANDARD 11)

set(CMAKE\_CUDA\_STANDARD 11)

# 添加 CPU 计算源文件

add\_library(cpu\_computation cpu\_computation.cpp)

# 添加 GPU 计算源文件

add\_library(gpu\_computation gpu\_computation.cu)

# 主程序

add\_executable(main main.cpp)

# 链接库

target\_link\_libraries(main cpu\_computation gpu\_computation)

```

### 2. `main.cpp`

```cpp

#include <iostream>

#include "cpu\_computation.h"

#include "gpu\_computation.h"

int main() {

const int N = 1024;

float \*input = new float[N];

float \*output\_cpu = new float[N];

float \*output\_gpu = new float[N];

// 初始化输入数据

for (int i = 0; i < N; ++i) {

input[i] = static\_cast<float>(i);

}

// CPU 计算

std::cout << "Running CPU computation..." << std::endl;

cpu\_computation(input, output\_cpu, N);

// GPU 计算

std::cout << "Running GPU computation..." << std::endl;

gpu\_computation(input, output\_gpu, N);

// 比较结果

for (int i = 0; i < N; ++i) {

if (output\_cpu[i] != output\_gpu[i]) {

std::cerr << "Mismatch at index " << i << ": CPU = " << output\_cpu[i] << ", GPU = " << output\_gpu[i] << std::endl;

break;

}

}

// 清理内存

delete[] input;

delete[] output\_cpu;

delete[] output\_gpu;

return 0;

}

```

### 3. `cpu\_computation.h`

```cpp

#ifndef CPU\_COMPUTATION\_H

#define CPU\_COMPUTATION\_H

void cpu\_computation(float \*input, float \*output, int size);

#endif // CPU\_COMPUTATION\_H

```

### 4. `cpu\_computation.cpp`

```cpp

#include "cpu\_computation.h"

#include <cmath>

void cpu\_computation(float \*input, float \*output, int size) {

for (int i = 0; i < size; ++i) {

output[i] = std::sin(input[i]); // 示例计算：正弦函数

}

}

```

### 5. `gpu\_computation.cu`

```cpp

#include <cuda\_runtime.h>

#include "cpu\_computation.h"

\_\_global\_\_ void gpu\_kernel(float \*input, float \*output, int size) {

int index = blockIdx.x \* blockDim.x + threadIdx.x;

if (index < size) {

output[index] = sinf(input[index]); // 示例计算：正弦函数

}

}

void gpu\_computation(float \*input, float \*output, int size) {

float \*d\_input, \*d\_output;

cudaMalloc((void\*\*)&d\_input, size \* sizeof(float));

cudaMalloc((void\*\*)&d\_output, size \* sizeof(float));

cudaMemcpy(d\_input, input, size \* sizeof(float), cudaMemcpyHostToDevice);

int blockSize = 256;

int numBlocks = (size + blockSize - 1) / blockSize;

gpu\_kernel<<<numBlocks, blockSize>>>(d\_input, d\_output, size);

cudaMemcpy(output, d\_output, size \* sizeof(float), cudaMemcpyDeviceToHost);

cudaFree(d\_input);

cudaFree(d\_output);

}

```

### 项目说明

- \*\*项目结构\*\*：项目包含 CPU 和 GPU 的计算模块，使用 CMake 构建。

- \*\*CPU 计算\*\*：在 `cpu\_computation.cpp` 中实现，计算输入数据的正弦值。

- \*\*GPU 计算\*\*：在 `gpu\_computation.cu` 中实现，使用 CUDA 核心函数并行计算正弦值。

- \*\*结果比较\*\*：在 `main.cpp` 中比较 CPU 和 GPU 的计算结果，确保它们相同。

- \*\*内存管理\*\*：程序中动态分配和释放内存，确保没有内存泄漏。

### 编译和运行

1. 确保安装了 CUDA 和 CMake。

2. 在项目目录下运行以下命令：

```bash

mkdir build

cd build

cmake ..

make

./main

```

### 扩展

- \*\*任务调度\*\*：可以根据任务的特性（计算密集型、内存带宽等）动态选择使用 CPU 或 GPU。

- \*\*性能测试\*\*：可以在不同的硬件上测试性能，并记录功耗等数据。

- \*\*混合调度策略\*\*：可以根据任务类型，将不同的计算任务分配给不同的计算单元。

这个项目是一个基础的示例，您可以根据需要进行扩展和优化。希望这能帮助您探索异构计算的加速方式！如果有其他问题或需要进一步的帮助，请随时询问！