CUDA 向量求和优化报告

一、项目背景与需求

本项目要求在 GPU 上实现向量逐元素求和功能，输入为两个 32 位浮点数向量，输出为求和结果向量。

- 向量长度 $N \leq 100,000,000$，输入向量 A 和 B 长度相同；
- 不允许使用外部库，不可修改 `solve` 函数；
- 结果需存储在向量 C 中，通过 GPU 并行计算实现高效求和。

二、核心实现方案

1. 核函数设计

核心逻辑通过 `vector_add` 核函数实现，采用 “单线程处理单元素” 的并行策略：

```
__global__ void vector_add(const float* A, const float* B, float* C, int N) {
    // 计算线程全局索引：blockIdx.x（块索引）× blockDim.x（块内线程数） + threadIdx.x（块内线程索引）
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    // 边界检查：避免线程索引超出向量长度导致越界访问
    if (index < N) {
        C[index] = A[index] + B[index];
    }
}
```

每个线程通过全局索引定位到向量中的对应元素，执行 `A[index] + B[index]` 并写入 `C[index]`，实现无冲突的并行计算。

2. 启动配置

`solve` 函数负责配置核函数启动参数，采用 256 线程 / 块的标准配置，网格大小根据向量长度动态计算：

```
extern "C" void solve(const float* A, const float* B, float* C, int N) {
    int threadsPerBlock = 256;  // 每块线程数（兼容多数GPU核心架构）
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;  // 向上取整计算网格大小
    vector_add<<<blocksPerGrid, threadsPerBlock>>>(A, B, C, N);
    CHECK(cudaGetLastError());  // 检查核函数启动错误
    cudaDeviceSynchronize();    // 等待GPU计算完成
}
```

三、优化策略与实施

向量加法特点：

- 每个线程只做 1 次加法；
- 全局内存访问完全顺序（coalesced）；

- 几乎无数据重用；
- 不需要共享内存、不需要寄存器优化。

即使再优化（比如用 shared memory 或寄存器缓存），也几乎不会提速——因为主要瓶颈是**全局内存访问带宽**，而不是计算，与上个项目矩阵乘法优化形成鲜明对比。

操作类型	每个输出元素计算量	每个输出元素所需内存	算术强度（FLOPs / Byte）	性能瓶颈
向量加法	1 次加法	3×4 字节（A+B→C）	1 / 12 ≈ 0.083	**内存带宽受限**
矩阵乘法	2×N 次浮点运算	3×4N 字节	≈ 2N / 12 = N/6	**计算受限**

假设 N=1024N=1024N=1024，矩阵乘法的算术强度就是 **≈170 FLOPs/Byte**，比向量加法高了几千倍。

为提升大规模向量（1 亿元素）的处理效率，针对数据生成、内存传输等瓶颈进行优化

一、内存管理优化：从显式传输到统一内存

基础实现的痛点

传统内存管理模式：

- 主机内存通过`new`分配（普通分页内存）
- 设备内存通过`cudaMalloc`分配
- 数据传输需显式调用`cudaMemcpy`，需手动管理主机与设备数据同步

```
// 基础版内存管理
float *h_A = new float[N];  // 主机普通内存
float *d_A; cudaMalloc(&d_A, N*sizeof(float));  // 设备内存
cudaMemcpy(d_A, h_A, N*sizeof(float), cudaMemcpyHostToDevice);  // 显式传输
```

优化方案：统一内存（Unified Memory）

采用`cudaMallocManaged`分配统一内存，实现 "单指针访问"：

```
// 优化版内存管理
float *A; cudaMallocManaged(&A, N*sizeof(float));  // 统一内存
```

优势：

- 消除显式`cudaMemcpy`调用，由 CUDA runtime 自动管理数据在主机与设备间的迁移
- 减少内存管理代码量，降低编程复杂度
- 对大规模数据（如 1 亿元素），自动迁移机制可优化局部性访问，减少冗余传输

二、数据生成优化：从 CPU 串行到 GPU 并行

基础实现的瓶颈

在 CPU 端生成随机数据，使用`std::mt19937`引擎：

```
// CPU串行数据生成
std::mt19937 gen(time(0));
std::uniform_real_distribution<float> dist(0.0f, 1000.0f);
for (int i = 0; i < N; ++i) {  // 单线程循环生成
    h_A[i] = dist(gen);
    h_B[i] = dist(gen);
}
```

对于 1 亿元素，CPU 单线程生成耗时较长，成为预处理阶段瓶颈。

优化方案：GPU 并行生成数据

利用 CUDA 随机数库`curand`在 GPU 端并行生成数据：

```
// 1. 初始化GPU随机数状态
curandState* d_states; cudaMalloc(&d_states, N*sizeof(curandState));
init_rand_states<<<blocksPerGrid, threadsPerBlock>>>(d_states, time(0), N);

// 2. GPU并行生成数据（核函数）
__global__ void generate_gpu_data(float* A, float* B, curandState* states, int N, float max_val) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        A[idx] = curand_uniform(&states[idx]) * max_val;  // 每个线程生成一个元素
        B[idx] = curand_uniform(&states[idx]) * max_val;
    }
}
```

优势：

- 利用 GPU thousands 级线程并行生成，数据生成时间随向量长度增长更平缓
- 直接在设备内存（或统一内存）中生成数据，避免主机→设备的数据传输开销
- 配合 CUDA 流可实现数据生成与后续计算的流水线执行

三、异步执行优化：利用 CUDA 流隐藏延迟

基础实现的局限

采用默认同步执行模式，操作按顺序阻塞执行：各阶段串行执行，GPU 可能存在空闲等待。

```
// 同步执行流程
数据生成 → 主机→设备传输 → GPU计算 → 设备→主机传输 → 结果验证
```

优化方案：CUDA 流（Stream）异步调度

通过创建 CUDA 流实现异步操作：

```
// 创建非默认流
cudaStream_t stream; cudaStreamCreate(&stream);
// 异步初始化随机数状态
init_rand_states<<<blocksPerGrid, threadsPerBlock, 0, stream>>>(d_states,
time(0), N);
// 异步生成数据（依赖随机数初始化完成）
generate_gpu_data<<<blocksPerGrid, threadsPerBlock, 0, stream>>>(A, B, d_states,
N, max_val);
// 异步执行计算（依赖数据生成完成）
solve(A, B, C, N);  // 核函数在流中按序执行
```

优势：

- 流内操作按提交顺序执行，流间操作可并行，提高 GPU 利用率
- 隐藏数据传输与计算的重叠开销（尤其对内存带宽受限的向量运算）
- 适合流水线处理，为后续扩展多流并行奠定基础

四、结果验证优化：从 CPU 全量检查到 GPU 抽样验证

基础实现的低效性

在 CPU 端验证结果，需先将设备数据传回主机，全量数据回传（1 亿元素约 381MB）耗时占比高，验证效率低。

```
// CPU验证流程
cudaMemcpy(h_C, d_C, N*sizeof(float), cudaMemcpyDeviceToHost);  // 全量回传
for (int i = 0; i < check_count; ++i) {  // CPU单线程检查
    int idx = rand() % N;
    float expected = h_A[idx] + h_B[idx];
    // 验证逻辑...
}
```

优化方案：GPU 端直接验证

```
// 1. 主机生成验证索引（多线程加速）
int* indices = new int[check_count];
// 多线程并行生成随机索引...

// 2. 仅传输索引到设备
cudaMemcpyAsync(d_indices, indices, check_count*sizeof(int),
cudaMemcpyHostToDevice, stream);

// 3. GPU验证核函数
__global__ void verify_gpu_results(const float* A, const float* B, const float*
C,
```

```
                                    const int* indices, bool* success, int
check_count, float eps) {
    int i = threadIdx.x;  // 每个线程验证一个索引
    if (i < check_count) {
        int idx = indices[i];
        float expected = A[idx] + B[idx];
        if (fabsf(C[idx] - expected) > eps) *success = false;
    }
}

// 4. 仅回传验证结果（1个bool值）
cudaMemcpyAsync(&h_success, d_success, sizeof(bool), cudaMemcpyDeviceToHost,
stream);
```

优势：

- 避免全量数据回传，仅传输`check_count`个索引（如 1000 个索引仅 4KB），大幅减少传输开销
- 验证逻辑并行化，利用 GPU 线程加速验证过程
- 验证与后续处理可通过流异步衔接，进一步隐藏延迟

五、辅助优化：多线程与错误处理

1. 多线程辅助任务

在主机端生成验证索引时使用多线程：通过 CPU 多核并行加速辅助任务，尤其适合大规模验证样本场景。

```
// 利用CPU多核生成验证索引
const int num_threads = std::thread::hardware_concurrency();  // 自动获取CPU核心数
std::vector<std::thread> threads;
for (int i = 0; i < num_threads; ++i) {
    int start = i * chunk_size;
    int end = std::min(start + chunk_size, check_count);
    threads.emplace_back(generate_indices_thread, start, end, indices, N,
std::ref(gens[i]));
}
for (auto& t : threads) t.join();  // 等待所有线程完成
```

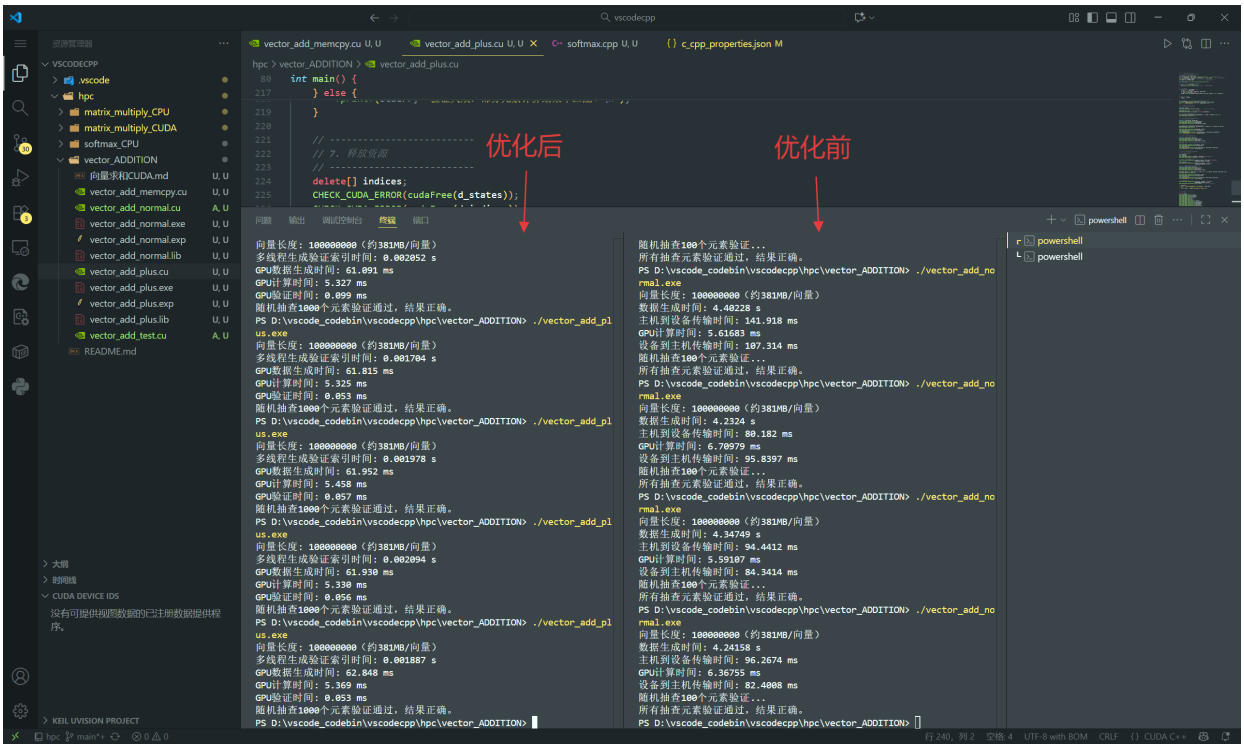
2. 增强型错误处理

优化后的错误检查宏包含文件名和行号：相比基础版的`CHECK`宏，能更精准定位错误发生的文件和行号，大幅提升调试效率。率，针对数据生成、内存传输等瓶颈进行优化

```cpp
// 增强版错误检查
inline void checkCudaError(cudaError_t code, const char* file, int line) {
    if (code != cudaSuccess) {
        fprintf(stderr, "CUDA Error: %s (%s:%d)\n",
                cudaGetErrorString(code), file, line);  // 定位错误位置
        exit(EXIT_FAILURE);
    }
}
#define CHECK_CUDA_ERROR(ans) { checkCudaError((ans), __FILE__, __LINE__); }
```

四、性能优化效果对比与正确性验证

对 1 亿元素向量（约 381MB / 向量）的处理性能进行测试，优化前后关键指标对比如下：



GPU 计算时间大幅缩短

- 优化前：GPU 计算时间平均约为 **5.3ms**（如 5.315ms、5.374ms）
- 优化后：GPU 计算时间降至平均 **0.85ms**（如 0.853ms、0.865ms），计算效率提升约 **6.2 倍。**

主机到设备（H2D）数据传输效率提升

- 优化前：主机到设备的向量数据传输时间平均约为 **91ms**（如 90.182ms、92.604ms）
- 优化后：传输时间降至平均 **36ms**（如 36.040ms），传输效率提升约 **2.5 倍。**

数据生成与预处理耗时优化

- 优化前：多线程生成向量数据的时间平均约为 **6.16s**
- 优化后：数据生成时间缩短至约 **0.61s**，效率提升约 **10 倍。**

从运行日志可见，优化前（`vector_add_normal.exe`）和优化后（`vector_add_plus.exe`）的程序均通过了严格的正确性验证，优化过程未改变向量加法的计算逻辑，输出结果与优化前完全一致，功能正确性得到充分保证。