CUDA 并行归约

```
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

  sdata[tid] = g_idata[i];
  //__syncthreads();  // 移除第一次规约的同步

  for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  if (tid == 0)
    g_odata[blockIdx.x] = sdata[0];
}

int main() {
  const int N = 1024;  // 数组大小
  const int block_size = 256;  // 每个块的线程数量

  // 在主机端分配和初始化输入数组
  int h_idata[N];
  for (int i = 0; i < N; ++i) {
    h_idata[i] = i;
  }

  // 在设备端分配和初始化输入和输出数组
  int *d_idata, *d_odata;
  cudaMalloc((void**)&d_idata, N * sizeof(int));
  cudaMalloc((void**)&d_odata, (N / block_size) * sizeof(int));
  cudaMemcpy(d_idata, h_idata, N * sizeof(int), cudaMemcpyHostToDevice);

  // 第一次规约
  reduce0<<<N / block_size, block_size, block_size * sizeof(int)>>>(d_idata,
d_odata);

  // 同步设备
  cudaDeviceSynchronize();

  // 第二次规约（将输入和输出数组都设置为d_odata）
  reduce0<<<1, N / block_size, (N / block_size) * sizeof(int)>>>(d_odata,
d_odata);

  // 同步设备
  cudaDeviceSynchronize();
```

```
    // 在设备端将结果传输回主机端
    int h_odata;
    cudaMemcpy(&h_odata, d_odata, sizeof(int), cudaMemcpyDeviceToHost);

    // 打印结果
    printf("Final reduction result: %d\n", h_odata);

    // 释放设备端内存
    cudaFree(d_idata);
    cudaFree(d_odata);

    return 0;
}
```

Halve the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Reduction #5: Unroll the Last Warp

```
__device__ void warpReduce(volatile int* sdata, int tid) {
      sdata[tid] += sdata[tid + 32];
      sdata[tid] += sdata[tid + 16];
      sdata[tid] += sdata[tid +  8];
      sdata[tid] += sdata[tid +  4];
      sdata[tid] += sdata[tid +  2];
      sdata[tid] += sdata[tid +  1];
 }
```

IMPORTANT:
For this to be correct, we must use the "volatile" keyword!

```
// later…
 for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
      if (tid < s)
         sdata[tid] += sdata[tid + s];
      __syncthreads();
 }

 if (tid < 32) warpReduce(sdata, tid);
```

Reduction #6: Completely Unrolled

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
      if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
      if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
      if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
      if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
      if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
      if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
 }
```

```
   if (blockSize >= 512) {
      if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
   if (blockSize >= 256) {
      if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
   if (blockSize >= 128) {
      if (tid <  64)  { sdata[tid] += sdata[tid +   64]; } __syncthreads(); }

   if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

Don’t we still need block size at compile time?

- Nope, just a switch statement for 10 possible block sizes:

```
switch (threads)
    {
    case 512:
      reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
      reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
      reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
      reduce5< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
      reduce5< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
      reduce5< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  8:
      reduce5<  8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  4:
      reduce5<  4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  2:
      reduce5<  2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  1:
      reduce5<  1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    }
```



```
__device__ void warpReduce(volatile int *sdata, int tid) {
  if (blockSize >= 64)
    sdata[tid] += sdata[tid + 32];
  if (blockSize >= 32)
    sdata[tid] += sdata[tid + 16];
  if (blockSize >= 16)
    sdata[tid] += sdata[tid + 8];
  if (blockSize >= 8)
    sdata[tid] += sdata[tid + 4];
  if (blockSize >= 4)
    sdata[tid] += sdata[tid + 2];
  if (blockSize >= 2)
    sdata[tid] += sdata[tid + 1];
}

template<unsigned int blockSize>
__global__ void reduce(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];
  unsigned int tid = threadIdx.x;
  // unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

  //  sdata[tid] = g_idata[i];
  //  with two loads and first add of the reduction
  //  perform first level of reduction
  unsigned int i = blockIdx.x * blockSize * 2 + threadIdx.x;
  unsigned int gridSize = blockSize * 2 * gridDim.x;

  sdata[tid] = 0;

  while(i < n){
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
```

```
  }
  __syncthreads();

  if(blockSize >= 512){
    if(tid < 256){
      sdata[tid] += sdata[tid+256];
      __syncthreads();
    }
  }

  if(blockSize >= 256){
    if(tid < 128){
      sdata[tid] += sdata[tid+128];
      __syncthreads();
    }
  }

  if(blockSize >= 128){
    if(tid < 64){
      sdata[tid] += sdata[tid+64];
      __syncthreads();
    }
  }

  if (tid < 32)
    warpReduce(sdata, tid);
  //  if(tid < 32){
  //      volatile int *smem = sdata;
  //      #pragma unroll 8
  //      for(unsigned innt s=16; s>=1; s>>=1){
  //          smem[tid] += smem[tid+s];
  //      }
  //  }

  if (tid == 0)
    g_odata[blockIdx.x] = sdata[0];
}
```

```
// 两遍规约
template<unsigned int numThreads>
__global__ void reduction1_kernel(int *out, const int *in, size_t N)
{
    // lenght = threads (BlockDim.x)
    extern __shared__ int sPartials[];
    int sum = 0;
    const int tid = threadIdx.x;
    for (size_t i = blockIdx.x * numThreads+ tid; i < N; i += numThreads *
gridDim.x)
    {
        sum += in[i];
    }
    sPartials[tid] = sum;
    __syncthreads();

    if (numThreads >= 1024)
    {
```

```
        if (tid < 512) sPartials[tid] += sPartials[tid + 512];
        __syncthreads();
    }
    if (numThreads >= 512)
    {
        if (tid < 256) sPartials[tid] += sPartials[tid + 256];
        __syncthreads();
    }
    if (numThreads >= 256)
    {
        if (tid < 128) sPartials[tid] += sPartials[tid + 128];
        __syncthreads();
    }
    if (numThreads >= 128)
    {
        if (tid < 64) sPartials[tid] += sPartials[tid + 64];
        __syncthreads();
    }

    if (tid < 32)
    {
        volatile int *wsSum = sPartials;
        if (numThreads >= 64) wsSum[tid] += wsSum[tid + 32];
        if (numThreads >= 32) wsSum[tid] += wsSum[tid + 16];
        if (numThreads >= 16) wsSum[tid] += wsSum[tid + 8];
        if (numThreads >= 8) wsSum[tid] += wsSum[tid + 4];
        if (numThreads >= 4) wsSum[tid] += wsSum[tid + 2];
        if (numThreads >= 2) wsSum[tid] += wsSum[tid + 1];

        if (tid == 0)
        {
            out[blockIdx.x] = wsSum[0];
        }
    }
}
```

```
template<unsigned int numThreads>
void reduction1_template(int *answer, int *partial, const int *in, const size_t
N, const int numBlocks)
{
    unsigned int sharedSize = numThreads * sizeof(int);

    // kernel execution
    reduction1_kernel<numThreads><<<numBlocks, numThreads, sharedSize>>>(partial,
in, N);
    reduction1_kernel<numThreads><<<1, numThreads, sharedSize>>>(answer, partial,
numBlocks);
}

void reduction1t(int *answer, int *partial, const int *in, const size_t N, const
int numBlocks, int numThreads)
{
    switch (numThreads)
    {
        case 1: reduction1_template<1>(answer, partial, in, N, numBlocks); break;
        case 2: reduction1_template<2>(answer, partial, in, N, numBlocks); break;
```

```
        case 4: reduction1_template<4>(answer, partial, in, N, numBlocks); break;
        case 8: reduction1_template<8>(answer, partial, in, N, numBlocks); break;
        case 16: reduction1_template<16>(answer, partial, in, N, numBlocks);
break;
        case 32: reduction1_template<32>(answer, partial, in, N, numBlocks);
break;
        case 64: reduction1_template<64>(answer, partial, in, N, numBlocks);
break;
        case 128: reduction1_template<128>(answer, partial, in, N, numBlocks);
break;
        case 256: reduction1_template<256>(answer, partial, in, N, numBlocks);
break;
        case 512: reduction1_template<512>(answer, partial, in, N, numBlocks);
break;
        case 1024: reduction1_template<1024>(answer, partial, in, N, numBlocks);
break;
    }
}
```

```
 // lenght = threads (BlockDim.x)
    extern __shared__ int sPartials[];
    int sum = 0;
    const int tid = threadIdx.x;
    for (size_t i = blockIdx.x * blockDim.x + tid; i < N; i += blockDim.x *
gridDim.x)
    {
        sum += in[i];
    }
    sPartials[tid] = sum;
    __syncthreads();

    // 调整为2^n
    unsigned int floowPow2 = blockDim.x;
    if (floowPow2 & (floowPow2 - 1))
    {
        while(floowPow2 & (floowPow2 - 1))
        {
            floowPow2 &= (floowPow2 - 1);
        }
        if (tid >= floowPow2)
        {
            sPartials[tid - floowPow2] += sPartials[tid];
        }
        __syncthreads();
    }
```

CUDA矩阵乘法

```
#include <iostream>
#include <cuda_runtime.h>

// CUDA kernel function for matrix multiplication
```

```cpp
__global__ void matrixMul(float* A, float* B, float* C, int width) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < width && col < width) {
        float sum = 0.0f;
        for (int i = 0; i < width; i++) {
            sum += A[row * width + i] * B[i * width + col];
        }
        C[row * width + col] = sum;
    }
}

int main() {
    const int width = 1024;
    size_t size = width * width * sizeof(float);

    // Allocate memory on the host
    float* h_A = new float[width * width];
    float* h_B = new float[width * width];
    float* h_C = new float[width * width];

    // Initialize matrix A and B
    // ... (Omitted for brevity)

    // Allocate memory on the device
    float* d_A, * d_B, * d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // Copy data from host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Launch the CUDA kernel
    dim3 block(16, 16);
    dim3 grid((width + block.x - 1) / block.x, (width + block.y - 1) / block.y);
    matrixMul<<<grid, block>>>(d_A, d_B, d_C, width);

    // Copy the result from device to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free memory
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

```cpp
// 加上了共享内存的方式
// CUDA kernel function for matrix multiplication
```

```
__global__ void matrixMulKernel(float* A, float* B, float* C, int width) {
    // Allocate shared memory for matrix tiles
    __shared__ float Asub[16][16];
    __shared__ float Bsub[16][16];

    // Calculate the row and column index of the element
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Initialize the accumulator
    float sum = 0.0f;

    // Loop over the tiles
    for (int t = 0; t < (width + 15) / 16; ++t) {
        // Load the matrices from global memory to shared memory
        Asub[threadIdx.y][threadIdx.x] = A[row * width + (t * 16 + threadIdx.x)];
        Bsub[threadIdx.y][threadIdx.x] = B[(t * 16 + threadIdx.y) * width + col];

        // Synchronize to make sure the matrix tiles are loaded
        __syncthreads();

        // Multiply the two matrices together
        for (int i = 0; i < 16; ++i) {
            sum += Asub[threadIdx.y][i] * Bsub[i][threadIdx.x];
        }

        // Synchronize to make sure all the partial results have been computed
        __syncthreads();
    }

    // Write the result to global memory
    C[row * width + col] = sum;
}
```

矩阵乘法

```
// 矩阵乘法
vector<vector<int>> matrixMultiply(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
    int m = A.size();
    int n = B[0].size();
    int p = A[0].size();

    vector<vector<int>> C(m, vector<int>(n, 0));

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < p; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return C;
}
```

```cpp
// 分块矩阵乘法
vector<vector<int>> matrixMultiplyBlocked(const vector<vector<int>>& A, const
vector<vector<int>>& B, int block_size) {
    int m = A.size();
    int n = B[0].size();
    int p = A[0].size();

    vector<vector<int>> C(m, vector<int>(n, 0));

    for (int i = 0; i < m; i += block_size) {
        for (int j = 0; j < n; j += block_size) {
            for (int k = 0; k < p; k += block_size) {
                // 计算当前块的结果
                for (int ii = i; ii < min(i + block_size, m); ii++) {
                    for (int jj = j; jj < min(j + block_size, n); jj++) {
                        for (int kk = k; kk < min(k + block_size, p); kk++) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }

    return C;
}
```

反转链表

```cpp
ListNode* reverseList(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* curr = head;
    while (curr != nullptr) {
        ListNode* next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

快速排序

```cpp
int partition(int* a,int low,int high){
    int pivot = a[low];
    while(low < high){
        while(low<high && a[high] >= pivot) --high;
        a[low] = a[high];
        while(low<high && a[low] <= pivot) ++low;
        a[high] = a[low];
    }
    a[low] = pivot;
    return low;
```

```
}

void qsort(int* a, int low, int high){
    if(low < high){
        int pivot = partition(a,low,high);
        qsort(a,low,pivot-1);
        qsort(a,pivot+1,high);
    }
}
```

归并排序

```
const int N = 100010;
int a[N];
int *b = (int *)malloc(sizeof(int)*N);

void merge(int *a,int low,int mid,int high){
    int i,j,k;
    for(k=low;k<=high;k++){
        b[k] = a[k];
    }
    for(i=low,k=i,j=mid+1;i<=mid&&j<=high;k++){
        if(b[i] <= b[j]){
            a[k] = b[i++];
        }
        else{
            a[k] = b[j++];
        }
    }
    while(i<=mid) a[k++] = b[i++];
    while(j<=high) a[k++] = b[j++];
}

void mergesort(int *a,int low,int high){
    if(low < high){
        int mid = (low+high)/2;
        mergesort(a,low,mid);
        mergesort(a,mid+1,high);
        merge(a,low,mid,high);
    }
}
```

树

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
};
```

```
void preorderTraversal(TreeNode* root) {
    if (!root) return;
    cout << root->val << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
```

链表

```
typedef struct Node{
    int data;
    Node *next;
}node;

node* create(int data){
    node *n = (node *)malloc(sizeof(node));
    n->data = data;
    n->next = NULL;
    return n;
}

void print(node *head){
    node* p;
    p = head;
    while(p->next != NULL){
        cout << p->next->data << " ";
        p = p->next;
    }
    cout << endl;
}

// 反转
void reverse(node *head){
    node *p;
    p = head->next;
    head->next = NULL;
    while(p != NULL){
        node *t = p->next;
        p->next = head->next;
        head->next = p;
        p = t;
    }
}

// 快慢指针
void change(node *head){
    node *q, *p;
    // 1. 快慢指針 來确定中间位置
    q = head;
    p = head;
    while(q->next != NULL){
        p = p->next;
        q = q->next;
        if(q->next != NULL) q = q->next;
    }
```

```
    // 2. 反转链表
    // 头插法 头就是 p
    reverse(p);
    // 3. 按题目要求进行插入
    node *f = head->next;
    node *r = p->next;
    p->next = NULL;

    while(r!=NULL){
        node* t2 = r->next;
        r->next = f->next;
        f->next = r;
        f = r->next;
        r = t2;
    }
}
```

二叉树的中序遍历

```
typedef struct Node{
    char data;
    struct Node *left, *right;
}node;

void visit(node *t){
    cout << t->data << " ";
}

void bianli(node *t,int h){
    if(t != NULL){
    // 根 和 叶子
        if(h > 0 && t->left!=NULL && t->right!=NULL) cout << "(";
        bianli(t->left,h+1);
        visit(t);
        bianli(t->right,h+1);
        if(h > 0 && t->left!=NULL && t->right!=NULL) cout << ")";
    }
}

node* create(char data){
    node* s;
    s = (node *)malloc(sizeof(node));
    s->data = data;
    s->right = NULL;
    s->left = NULL;
    return s;
}
```

STL

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
```

```
    // 入队
    q.push(1);
    q.push(2);
    q.push(3);

    // 打印队列
    std::cout << "Queue: ";
    while (!q.empty()) {
        std::cout << q.front() << " ";
        q.pop();
    }
    std::cout << std::endl;

    // 再次入队
    q.push(4);
    q.push(5);

    // 访问队头元素
    std::cout << "Front element: " << q.front() << std::endl;

    // 出队
    q.pop();
    std::cout << "After pop, front element: " << q.front() << std::endl;

    return 0;
}
```

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;

    // 压栈
    s.push(1);
    s.push(2);
    s.push(3);

    // 打印栈
    std::cout << "Stack: ";
    while (!s.empty()) {
        std::cout << s.top() << " ";
        s.pop();
    }
    std::cout << std::endl;

    // 再次压栈
    s.push(4);
    s.push(5);

    // 访问栈顶元素
    std::cout << "Top element: " << s.top() << std::endl;

    // 出栈
    s.pop();
```

```
    std::cout << "After pop, top element: " << s.top() << std::endl;

    return 0;
}
```

vector

```
// 初始化一个有 5 个元素的 vector, 每个元素初始化为 0
std::vector<int> v2(5, 0);

// 从数组初始化 vector
int arr[] = {1, 2, 3, 4, 5};
std::vector<int> v3(arr, arr + sizeof(arr)/sizeof(int));

// 使用 initializer_list 初始化 vector
std::vector<int> v4 = {1, 2, 3, 4, 5};

std::vector<int> v;
v.push_back(1);  // 在末尾添加元素 1
v.emplace_back(2);  // 在末尾添加元素 2
v.insert(v.begin() + 1, 3);  // 在第二个位置插入元素 3
v.erase(v.begin() + 2);  // 删除第三个元素
v.clear();  // 清空 vector

std::vector<int> v = {1, 2, 3, 4, 5};
int first = v[0];  // 访问第一个元素
int last = v.back();  // 访问最后一个元素
for (int i = 0; i < v.size(); i++) {
    std::cout << v[i] << " ";
}
for (int x : v) {
    std::cout << x << " ";  // 使用范围 for 循环遍历
}

std::vector<int> v1 = {1, 2, 3};
std::vector<int> v2 = {4, 5, 6};
v1.swap(v2);  // 交换 v1 和 v2 的内容
```

```
std::vector<std::vector<int>> grid(m, std::vector<int>(n));
```

生产者和消费者

```
std::mutex mtx;
std::condition_variable cv;
std::queue<int> q;

void producer() {
    mtx.lock();
    q.push(item);
    cv.notify_one(); // notify_one() 向一个条件变量发送通知,以唤醒正在等待这个条件变量的一个线程
    mtx.unlock();
}
```

```
void consumer() {
    mtx.lock();
    while (q.empty()) {
        cv.wait(mtx);
    }
    int item = q.front();
    q.pop();
    mtx.unlock();
}
```

`cv.notify_all();`，它会唤醒所有正在等待该条件变量的线程。

map

```
// 创建一个空的 map
std::map<std::string, int> myMap;

// 使用初始化列表创建并初始化 map
std::map<std::string, int> capitals = {
    {"USA", Washington},
    {"France", Paris},
    {"Japan", Tokyo}
};

// 插入新元素
myMap["London"] = 8942;
myMap.insert(std::make_pair("Berlin", 3670));

// 访问元素
int usaCapital = myMap["USA"];  // 获取值
auto it = myMap.find("France"); // 查找元素
if (it != myMap.end()) {
    std::cout << it->second; // 访问值
}

// 删除指定键的元素
myMap.erase("Japan");

// 删除迭代器指向的元素
auto it = myMap.find("France");
if (it != myMap.end()) {
    myMap.erase(it);
}

// 清空整个 map
myMap.clear();

// 查找元素是否存在
bool found = (myMap.find("USA") != myMap.end());

// 自定义比较函数
struct CustomComparator {
    bool operator()(const std::string& a, const std::string& b) {
        return a.length() < b.length();
    }
```

```
};

std::map<std::string, int, CustomComparator> myMap;
```

模板

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

int result = max(5, 10); // 调用 max<int>(5, 10)
```

匿名函数

```
   // 使用匿名函数作为比较器
    std::sort(numbers.begin(), numbers.end(), [](int a, int b) {
        return a > b; // 按照降序排序
    });

   // 使用 Lambda 函数查找最大值
    auto max_element = *std::max_element(numbers.begin(), numbers.end(),
        [](int a, int b) -> bool {
            return a < b;
        });

    // 使用 Lambda 函数计算平方和
    int square_sum = std::accumulate(numbers.begin(), numbers.end(), 0,
        [](int acc, int x) -> int {
            return acc + x * x;
        });
```

lambda 表达式的语法如下:

```
[capture clause](parameters) -> return_type { body }
```

- `capture clause`: 指定 lambda 函数可以访问的外部变量。可以为空。
- `parameters`: 函数参数列表。可以为空。
- `return_type`: 函数返回类型。可以省略,由编译器自动推断。
- `body`: 函数体。

Lambda 函数的 `capture clause` 用于指定 Lambda 函数可以访问的外部变量。可以使用以下几种方式:

1. **值捕获**:
    - 使用 `[x, y]` 捕获变量 `x` 和 `y` 的值。这会创建一个新的副本,在 Lambda 函数内部对这些副本进行操作,不会影响原始变量。
    - 使用 `[&x, &y]` 捕获变量 `x` 和 `y` 的引用。这样可以在 Lambda 函数内部直接操作原始变量。
2. **按值捕获全部**:
    - 使用 `[=]` 捕获所有外部变量的值。
    - 使用 `[&]` 捕获所有外部变量的引用。

3. **按引用捕获全部**:
    - 使用 `[=, &z]` 捕获所有外部变量的值,但变量 `z` 的引用。
    - 使用 `[&, x]` 捕获所有外部变量的引用,但变量 `x` 的值。
4. **隐式捕获**:
    - 使用 `[this]` 捕获当前类的 `this` 指针。这在成员函数中很常见。

```
int x = 10, y = 20;

// 值捕获 x 和 y
auto f1 = [x, y]() {
    std::cout << "x = " << x << ", y = " << y << std::endl;
    // 这里的 x 和 y 是副本,不会影响原始变量
};

// 引用捕获 x 和 y
auto f2 = [&x, &y]() {
    x = 30;
    y = 40;
    std::cout << "x = " << x << ", y = " << y << std::endl;
    // 这里直接修改了原始变量
};

// 按值捕获全部
auto f3 = [=]() {
    std::cout << "x = " << x << ", y = " << y << std::endl;
};

// 按引用捕获全部,但 x 按值捕获
auto f4 = [&, x]() {
    x = 50; // 这里修改的是按值捕获的 x 副本
    std::cout << "x = " << x << ", y = " << y << std::endl;
};
```

智能指针

```
#include <iostream>
#include <memory>

class MyClass {
public:
    MyClass() { std::cout << "MyClass constructor called." << std::endl; }
    ~MyClass() { std::cout << "MyClass destructor called." << std::endl; }
};

int main() {
    // 创建一个 unique_ptr
    std::unique_ptr<MyClass> ptr1(new MyClass());

    // 不允许拷贝 unique_ptr,但可以移动
    std::unique_ptr<MyClass> ptr2 = std::move(ptr1);

    // 输出 ptr1 和 ptr2 的状态
    std::cout << "ptr1 is " << (ptr1 ? "not null" : "null") << std::endl; // 输出:
ptr1 is null
```

```
    std::cout << "ptr2 is " << (ptr2 ? "not null" : "null") << std::endl; // 输出: ptr2 is not null

    // 当 ptr2 离开作用域时,对象将被自动释放
    return 0;
}
```