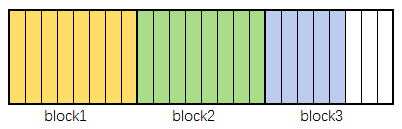
# Reduce——使用并行算法计算数组的和

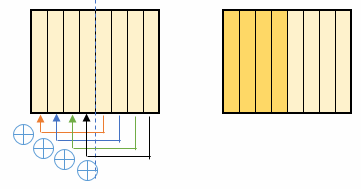
## 原理

有一个数组，对其求和，一串行编程时很简单，在并行编程时，可采用这里介绍的方法。

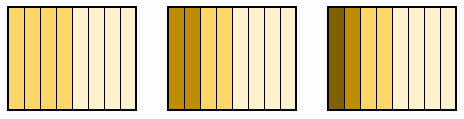
假如有一个数组长度为21，如果我们使用3个block去映射它，每个block的线程数为8（当然，这里只是举个例子，实际情况下，每个block可用的线程数很大，如1024），如下图所示：



我们采取这样的策略，对于每一个block，我们将block分前后两个部分，将前后两个部分对应元素相加，结果保存到前一部分的位置中，这样，前半部分保存了当前block对应的数组元素的和，如下图所示：



再将以上的前半部分的block分为两个部分，重复同样的操作，结果是前1/4的block保存了当前block对应的数组元素的和，继续以上步骤，直到最后第一个元素保存了当前block对应的数组元素的和：



每一个block都执行以上步骤，将每一个block的第一个元素的值保存到一个新的数组中：



现在的问题转为对这个新的数组求和，使用以上的策略，只是这里block的数目为1，最后直接读取第一个元素的值，即为新数组的元素和，也即为原21个元素的数组元素和。

## 代码

首先，这里使用共享内存来实现，这里要注意的是，Device的变量要使用cudaMalloc来分配内存，如例程中的Device的数组，更要注意的是，例程中的d\_out是一个指针，也需要使用cudaMalloc来分配内存，不可以用一个简单的int变量d\_out，然后取地址做为指针传入到cudaMemcpy函数中去。因为Device的变量使用cudaMalloc来分配内存，Device的变量只能用指针，而不能用简单变量。

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <iostream>

#include <stdlib.h>

#include <iostream>

\_\_global\_\_ void global\_reduce\_kernel(float \* d\_out, float \* d\_in)

{

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

int tid = threadIdx.x;

for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)

{

if (tid < s)

{

d\_in[myId] += d\_in[myId + s];

}

\_\_syncthreads();

}

if (tid == 0)

{

d\_out[blockIdx.x] = d\_in[myId];

}

}

void reduce(float \* d\_out, float \* d\_intermediate, float \* d\_in,

int size)

{

const int maxThreadsPerBlock = 8;

int threads = maxThreadsPerBlock;

int blocks = ceil(double(size) / maxThreadsPerBlock);

global\_reduce\_kernel <<<blocks, threads >>>

(d\_intermediate, d\_in);

threads = blocks;

blocks = 1;

global\_reduce\_kernel <<<blocks, threads >>>(d\_out, d\_intermediate);

}

int main(int argc, char \*\*argv)

{

int dev = 0;

cudaSetDevice(0);

const int ARRAY\_SIZE = 21;

const int ARRAY\_BYTES = ARRAY\_SIZE \* sizeof(float);

float h\_in[ARRAY\_SIZE];

float sum = 0.0f;

for (int i = 0; i < ARRAY\_SIZE; i++) {

h\_in[i] = 1.0f;

sum += h\_in[i];

}

float \* d\_in, \*d\_intermediate, \*d\_out;

cudaMalloc((void \*\*)&d\_in, ARRAY\_BYTES);

cudaMalloc((void \*\*)&d\_intermediate, ARRAY\_BYTES); // overallocated

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

cudaMemcpy(d\_in, h\_in, ARRAY\_BYTES, cudaMemcpyHostToDevice);

reduce(d\_out, d\_intermediate, d\_in, ARRAY\_SIZE);

float h\_out;

cudaMemcpy(&h\_out, d\_out, sizeof(float), cudaMemcpyDeviceToHost);

std::cout << "result1: " << h\_out << std::endl;

std::cout << "result2: " << sum << std::endl;

cudaFree(d\_in);

cudaFree(d\_intermediate);

cudaFree(d\_out);

system("pause");

return 0;

}

这里使用共享内存实现，与以上程序基本相似，值得注意的是共享内存的核方法的定义和调用。

\_\_global\_\_ void shmem\_reduce\_kernel(float \* d\_out, const float \* d\_in)

{

extern \_\_shared\_\_ float sdata[];

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

int tid = threadIdx.x;

sdata[tid] = d\_in[myId];

\_\_syncthreads();

for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)

{

if (tid < s)

{

sdata[tid] += sdata[tid + s];

}

\_\_syncthreads();

}

if (tid == 0)

{

d\_out[blockIdx.x] = sdata[0];

}

}

\_\_global\_\_ void global\_reduce\_kernel(float \* d\_out, float \* d\_in)

{

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

int tid = threadIdx.x;

for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)

{

if (tid < s)

{

d\_in[myId] += d\_in[myId + s];

}

\_\_syncthreads();

}

if (tid == 0)

{

d\_out[blockIdx.x] = d\_in[myId];

}

}

void reduce(float \* d\_out, float \* d\_intermediate, float \* d\_in,

int size)

{

const int maxThreadsPerBlock = 8;

int threads = maxThreadsPerBlock;

int blocks = ceil(double(size) / maxThreadsPerBlock);

shmem\_reduce\_kernel <<<blocks, threads, threads \* sizeof(float)>>>(d\_intermediate, d\_in);

blocks = 1;

shmem\_reduce\_kernel <<<blocks, threads, threads \* sizeof(float)>>>(d\_out, d\_intermediate);

}

int main(int argc, char \*\*argv)

{

int dev = 0;

cudaSetDevice(0);

const int ARRAY\_SIZE = 21;

const int ARRAY\_BYTES = ARRAY\_SIZE \* sizeof(float);

float h\_in[ARRAY\_SIZE];

float sum = 0.0f;

for (int i = 0; i < ARRAY\_SIZE; i++) {

h\_in[i] = 1.0f;

sum += h\_in[i];

}

float \* d\_in, \*d\_intermediate, \*d\_out;

cudaMalloc((void \*\*)&d\_in, ARRAY\_BYTES);

cudaMalloc((void \*\*)&d\_intermediate, ARRAY\_BYTES); // overallocated

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

cudaMemcpy(d\_in, h\_in, ARRAY\_BYTES, cudaMemcpyHostToDevice);

reduce(d\_out, d\_intermediate, d\_in, ARRAY\_SIZE);

float h\_out;

cudaMemcpy(&h\_out, d\_out, sizeof(float), cudaMemcpyDeviceToHost);

std::cout << "result1: " << h\_out << std::endl;

std::cout << "result2: " << sum << std::endl;

cudaFree(d\_in);

cudaFree(d\_intermediate);

cudaFree(d\_out);

system("pause");

return 0;

}