### 实验报告

并行计算大实验

陈其华 PB16060171

1. **OpenMP实验 -- 并行归并排序**

**算法描述**

归并排序从上到下分割排序问题，一个问题产生多个复杂度相当的子问题，因此适合并行计算来提高效率。实验中探索了三种并行的方法，第二种取得了最好的效果。

**结果总体展示：(十次平均数，**5000000个随机数排序**)**

串行运行时间:1772ms

第一种并行:22765ms 1200%

第二种并行:426ms 24%

第三种并行:540ms 30%

**第一种：直接在递归时进行并行**

核心代码：

void merge\_sort\_parallel1(int a[], int start, int end)

{

        if (a == NULL || start >= end)

                return;

        int mid = (end + start) / 2;

        printf("id:%d\n", omp\_get\_thread\_num()); //检查是否并行

        #pragma omp task //与section相同

        {

                merge\_sort\_parallel1(a, start, mid);

        }

        #pragma omp task

        {

                merge\_sort\_parallel1(a, mid+1, end);

        }

        #pragma omp taskwait  //等待两个线程都排序完再合并

        merge(a, start, mid, end);

}

打印出一部分线程id可以看到，它确实是并行的，但是非常慢。

这是因为这种方法对划分的线程数没有限制，每次递归二分的时候都重新创建线程，但是由于线程总数有限（设置为10，因为如果创建太多线程会更慢），所以实际上是在几个线程中不断切换，大部分时间都浪费在进程的切换上了。

打印出的一部分线程id： id:6 id:7 id:3 id:5 id:1 id:4 id:0 id:0 id:9 id:3

**第二种：在递归时进行有上限的并行**

void merge\_sort\_parallel2(int a[], int start, int end)

{

        if (a == NULL || start >= end)

                return;

        int mid = (end + start) / 2;

        //printf("id:%d\n", omp\_get\_thread\_num());

        if (end - start < NUM\_SORT / 16) { //如果划分超过4次,则不再并行划分

                merge\_sort(a, start, mid);

                merge\_sort(a, mid + 1, end);

        }

        else {

        #pragma omp task

        {

                merge\_sort\_parallel2(a, start, mid);

        }

        #pragma omp task

        {

                merge\_sort\_parallel2(a, mid + 1, end);

        }

        #pragma omp taskwait

        }

        merge(a, start, mid, end);

}

这种方法取得了最好的效果，而且很简单，只是在递归时加了一个判断防止划分出过多的线程。

尝试过改变判断的阈值来改变线程上限，但是结果差别并不大（所以没有展示），理论上应该是等于8时效果最好，但是并不明显，可能是切换次数少，所以总的开销比起排序所需时间的方差相比较小。

**第三种：手动划分并行子任务**

核心代码：

void merge\_sort\_parallel3(int a[]) {

        int i;

#pragma omp parallel for

                for (i = 0; i < 8; i++) {

                        merge\_sortpl3(a, i \* NUM\_SORT/8, i \* NUM\_SORT/8 + NUM\_SORT/8 - 1);

                }

#pragma omp parallel for

                for(i = 0; i < 4; i++){

                        merge(a, i \* NUM\_SORT/4, i \* NUM\_SORT/4 + NUM\_SORT/8-1, i \* NUM\_SORT/4 + NUM\_SORT/4 - 1);

}

#pragma omp parallel for

                for(i = 0; i < 2; i++){

                        merge(a, i \* NUM\_SORT/2, i \* NUM\_SORT/2 + NUM\_SORT/4-1, i \* NUM\_SORT/2 + NUM\_SORT/2 - 1);

}

merge(a, 0, NUM\_SORT/2 -1, NUM\_SORT);

}

这种方法比较麻烦，因为是手动将数组划分成八块进行并行，效果也比第二种差一些。

想这么做的原因是，原本我认为不应该让并行的程序做判断语句，并且私有变量会快一些，写完之后才发现这两个性质是GPU的，我搞混了。把原数组分成八块复制到各线程的私有变量中反而增加了额外的时间和空间开销，因为公有私有都在主机内存中。

如果去掉不必要的复制（如上面的代码），效果仍然不如第二种（我原本以为会持平）。分析后认为这是由于手动划分后还要手动合并三次，而这前后四个过程是串行的，因为parallel for要等所有线程结束才结束，并行度降低了。而第二种方法不需要等待所有子任务都完成后再进行手动合并的步骤，并行度更高。

**实验中遇到的问题：**

1. 原本用的clock计时，但是发现并行程序输出的时间比我感觉的更长，因为clock使用cpu的滴答数计时，线程并行时同时计算滴答数，得到的时间会更长。改用omp\_get\_wtime之后问题解决。
2. VS不太会用，总是忘记重新生成解决方案。
3. **CUDA实验 -- 双目深度估计SAD算法**

**算法的基本步骤：**

输入：两幅不同视角图像，一幅Left-Image，一幅Right-Image且两幅图像已经校正实现行对准

对左图，依次扫描，选定一个锚点：

1.构造一个小窗口,类似于卷积核；

2.用窗口覆盖左边的图像,选择出窗口覆盖区域内的所有像素点；

3.同样用窗口覆盖右边的图像并选择出覆盖区域的像素点；

4.左边覆盖区域减去右边覆盖区域，并求出所有像素点灰度差的绝对值之和；

5.移动右边图像的窗口，重复（3）-（4）的处理（这里有个搜索范围,超过这个范围跳出）；

6.找到这个范围内SAD值最小的窗口,即找到了左图锚点的最佳匹配的像素块。

由于该算法的扫描涉及到大量循环，而且行与行直接并不相关，串行计算非常慢，所以非常适合用GPU做并行来提速。之前用matlab写过这个代码，非常慢，相同参数设置需要超过6分钟。

**实验环境：**

信院GPU集群G101节点

**CPU**：Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz，核心数12

**GPU：**GTX1080Ti，3328个核心**，**1623MHz

**实验结果（最快的）：**

**算法参数**：512\*512像素，窗口大小12\*12，扫描距离40；GPU设置：512-12个block，每个block512个thread，即每个线程计算一个像素点视差；

**纯计算加速比（不包括数组初始化以及复制到GPU）：**

**2435（接近核心数量\*时钟频率比）**

**包括数组初始化以及复制到GPU加速比：**

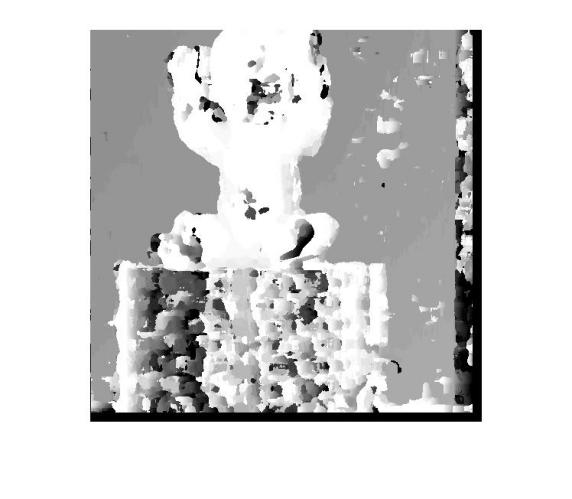
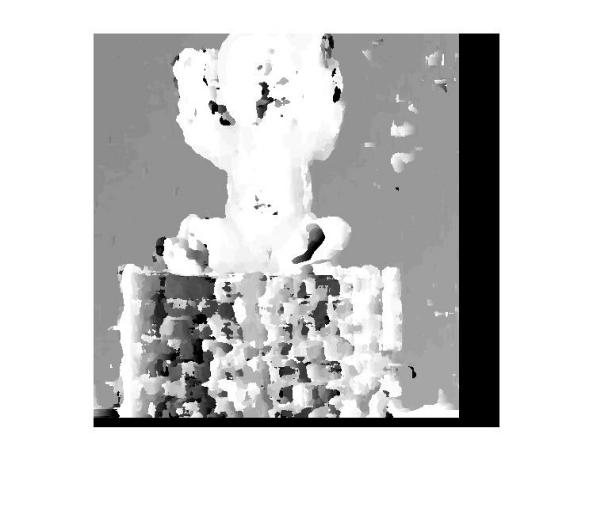
**8.08**

**详细结果统计：**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **平均** |
| **CPU** | **5.465s** | **5.864s** | **5.470s** | **5.462s** | **5.560s** | **5.778s** | **5.414s** | **5.975s** | **5.624s** |
| **GPU（纯计算）** | **2.30ms** | **2.31ms** | **2.32ms** | **2.32ms** | **2.30ms** | **2.29ms** | **2.32ms** | **2.31ms** | **2.31ms** |
| **GPU（包括分配）** | **0.698s** | **0.674s** | **0.745s** | **0.631s** | **0.719s** | **0.686s** | **0.708s** | **0.711s** | **0.696s** |
| **GPU（未使用共享存储纯计算）** | **15.2ms** | **15.1ms** | **15.1ms** | **15.0ms** | **15.1ms** | **15.2ms** | **15.1ms** | **15.0ms** | **15.1ms** |

**结果图：**

CPU，GPU结果 MATLAB结果

可以看到，GPU结果和MATLAB做出的视觉上并没有太大差别，但是大大加速了计算。

因为算法本身比较直接简单，而且图片分辨率较低并且是灰度图像，效果不是很好，在有大量重复纹理的时候效果很差，也不能处理遮挡但情况。但该实验主要讨论GPU并行计算的加速能力，算法本身不在讨论范围内。

**加速手段：**

1. **直接并行化**

**思路：**从算法步骤可以看到，每个像素的视差计算只依赖于两张视角图的一部分视野，所以可以每行由一个block负责，而block中每个线程负责一个像素的计算，不需要大循环，每个线程处理滑窗求差值，进行比较。

**代码：**

\_\_global\_\_ void sad\_gpu\_without\_sm(int\* dis, int\* v1, int\* v5, int width){

        int bi = blockIdx.x;

        int tj = threadIdx.x;

        if(tj < WIDTH-WINSIZE){

                int min = 2550000;

                int min\_index = 0;

                for(int k = 0; k < LIMIT; k++){

                        unsigned int result = 0;

                        for(int m = 0; m < WINSIZE; m++){

                                for(int l = 0; l < WINSIZE; l++ ){

                                        result = \_\_sad(v5[(bi+m)\*WIDTH+tj+l], v1[(bi+m)\*WIDTH+tj+k+l], result);

                                }

                        }

                        if(result < min){

                                min = result;

                                min\_index = k;

                        }

                }

                dis[bi\*WIDTH+tj] = min\_index;

        }

}

**结果：**平均计算时间为15.1ms，和CPU相比加速了372倍，虽然加速了很多，但还是远远小于GPU核心数，所以还有加速的空间。猜测额外的开销是由于大量访问GPU主存造成的。

**2.使用共享存储**

**思路：**由于个像素在匹配的时候窗口会水平滑动，每个像素会被多个线程使用（不考虑边界，会被用扫描距离数次，在本次实验中为40），因此可以每行的线程设置共享存储来减少对主存内容的存取。

所以设置每个block内的共享存储为对应行在计算时涉及的WIDTH\*WINSIZE\*2个像素。步骤为：先每个线程取它对应的像素列（1\*WINSIZE）到共享存储，同步；再每个像素按原方法计算对应视差。

**代码：**

\_\_global\_\_ void sad\_gpu(int\* dis, int\* v1, int\* v5, int width){

        //每行共享存储，为该行滑窗时需要的所有像素

        \_\_shared\_\_ int v1\_c[WINSIZE][WIDTH];

        \_\_shared\_\_ int v5\_c[WINSIZE][WIDTH];

        //同一行用一个block中的线程计算

        int bi = blockIdx.x;

        //一行中的每个像素对应视差用不同thread计算

        int tj = threadIdx.x;

        //每个线程取它对应的像素列

        for(int k = 0; k < WINSIZE; k++){

                v1\_c[k][tj] = v1[(bi+k)\*WIDTH + tj];

                v5\_c[k][tj] = v5[(bi+k)\*WIDTH + tj];

        }

        \_\_syncthreads();        //等所有都取完再计算

        //每不越界的线程计算滑窗像素和，找出最接近的

        if(tj < WIDTH-WINSIZE){

                int min = 2550000;

                int min\_index = 0;

                for(int k = 0; k < LIMIT; k++){ //滑窗

                        unsigned int result = 0;

                        for(int m = 0; m < WINSIZE; m++){

                                for(int l = 0; l < WINSIZE; l++ ){

                                        result = \_\_sad(v5\_c[m][tj+l], v1\_c[m][tj+k+l], result);

                                }

                        }

                        if(result < min){

                                min = result;

                                min\_index = k;

                        }

                }

                dis[bi\*WIDTH+tj] = min\_index;

        }

}

**结果：**平均计算时间为2..31ms，和CPU相比加速了2435倍，与核心数3328\*GPU时钟频率1623/CPU时钟频率2220得到的理论极限加速比2455相差很小，算法设计比较成功。相比不使用共享存储快6.6倍，说明不使用共享存储时，GPU确实把大部分时间花在访问主存上了。

但是存在共享存储使用过多的问题，如果再加大窗口大小或者图片大小，或者增加通道，编译就不通过。

**3.尝试简化计算（效果不好）**

**思路**：主要是为了解决共享存储不够的问题。能想到的是用每列的和计算窗口中的图像差别，而不是对每个像素。但是问题也非常明显，在滑窗时，需要计算每个对应像素的差的绝对值，如果用列和取代就损失了一些信息。

**代码：**

\_\_global\_\_ void sad\_gpu2(int\* dis, int\* v1, int\* v5, int width){

//每行共享存储

\_\_shared\_\_ int v1\_c[WIDTH];

\_\_shared\_\_ int v5\_c[WIDTH];

//同一行用一个block中的线程计算

int bi = blockIdx.x;

// printf("id: %d", bi);

//一行中的每个像素对应视差用不同thread计算

int tj = threadIdx.x;

//每个线程取它对应的像素列计算和

for(int k = 0; k < WINSIZE; k++){

v1\_c[tj] += v1[(bi+k)\*WIDTH + tj];

v5\_c[tj] += v5[(bi+k)\*WIDTH + tj];

}

\_\_syncthreads(); //等所有都取完再计算

//每不越界的线程计算滑窗像素和，找出最接近的

if(tj < WIDTH-WINSIZE){

int min = 2550000;

int min\_index = 0;

for(int k = 0; k < LIMIT; k++){ //滑窗

unsigned int result = 0;

for(int l = 0; l < WINSIZE; l++ ){

result = \_\_sad(v5\_c[tj+l], v1\_c[tj+k+l], result);

}

if(result < min){

min = result;

min\_index = k;

}

}

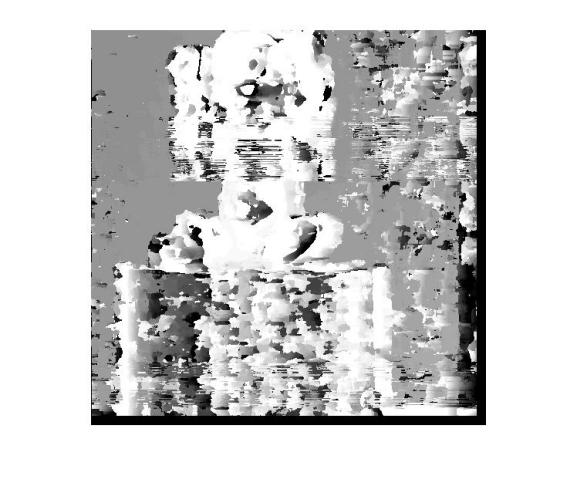
// printf("%d ",min\_index);

dis[bi\*WIDTH+tj] = min\_index;

}

}

**结果：**速度又提高了大约2000倍，并且使用的共享存储为原来的1/WINSIZE，但是效果也变差了：很明显损失了列方向的信息，引入了噪声，如果做平滑处理一下可能会好一些。



**其他可行的减少共享存储的方法是**将图片竖直方向切条来做，但是在切的时候各条需要有重合，因为视差计算需要用到水平方向的别的像素。这样会增加总存储和拷贝存储的时间。但是对这个算法而言，窗口大小和图片大小还有通道数都是非常影响结果的，这种改进是值得的。

**部分其他代码（参数定义以及kernel函数调用准备工作）**

#define WIDTH 512

#define LIMIT 40

#define WINSIZE 12

**main函数中：**

        //分配设备内存，非纯计算计时起点

        int\* d\_1;

        cudaMalloc((void\*\*)&d\_1, size\_im);

        int\* d\_5;

        cudaMalloc((void\*\*)&d\_5, size\_im);

        //拷贝数据到设备

        cudaMemcpy(d\_1, h\_1, size\_im, cudaMemcpyHostToDevice);

        cudaMemcpy(d\_5, h\_5, size\_im, cudaMemcpyHostToDevice);

        //分配结果所需空间

        int\* d\_dis;

        cudaMalloc((void\*\*)&d\_dis, size\_im);

        int\* h\_dis = (int\*)malloc(size\_im);

        //计时

        startTime = clock(); //纯计算计时起点

        //设置block，thread数

        dim3 threads(WIDTH);

        dim3 grid(WIDTH-WINSIZE);

        printf("begin gpu computation\n");

        sad\_gpu<<< grid, threads>>>(d\_dis, d\_1, d\_5, WIDTH);

        cudaThreadSynchronize(); //这个同步很重要

        //停止计时

        endTime = clock();

        double GPUtime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

        printf("GPU:%f\n", GPUtime);

        //把结果拷贝到主机

        cudaMemcpy(h\_dis, d\_dis, size\_im, cudaMemcpyDeviceToHost);

**遇到的问题：**

1. 该算法的每个线程在处理过程中都会遇到判断，而GPU并不擅长判断，它会把每个分支都做了之后，再取判断成立的那个分支。所以在选定题目的时候，有些怀疑这个算法的并行效果。但是所幸涉及的每个判断都是要么做处理要么直接跳过，所以其实并不影响，实验结果的加速比还是不错的。
2. GPU在资源不够的时候并不会报错，而是直接运行完毕，结果全为0。在集群上运行之前，要先查看有哪些空闲GPU，再指定GPU运行。

3. 从统计数据中明显可以看出，大部分时间都用于分配空间和复制主机存储到GPU主存了，所以实际上共享存储那一步优化虽然对计算速度的提升很多，但是对总体时间还是很微小。主要的瓶颈在复制存储。但是这一步操作是完全无法优化的，除非增大数据量，会使得计算占比更多，加速比才会提升。