PicCombine 实验报告

【最终实验结果】

合成结果：

时间对比：

（CPU 上）530.69412 秒

```
894 of 1024 done...
889 of 1024 done...
Rescaning done...: 530.69412 s
```

（GPU 上）使用和 CPU 上同样的查询函数时

282.910 秒 为什么它这么慢的猜测（点击跳转）

```
Start loading...
Loading done...: 0.12029 s
Start build_bvh...
build_bvh done...: 1.86424 s
Start querying...
Querying done... elapsed time: 282910.000000 ms
end
```

最终版本

117.26115 秒

```
Start querying...
Querying done... elapsed time: 117261.156250 ms
end
```

（加速倍数）4.5 倍

530.69412 ÷ 117.26115 =

4.525745483478543

【实验过程】

因为重要部分都已经在 CPU 版本的代码中写完了，所以只需要把计算要用的数据，getColor() 以及它调用到的函数搬到 GPU 上即可。

先搬数据：

```
//把线性BVH树复制给GPU
t_linearBvhNode* _root_gpu;
checkCudaErrors(cudaMalloc((void**)&_root_gpu, sizeof(t_linearBvhNode) * (1024 * 1024 * 2 + 1)));
checkCudaErrors(cudaMemcpy(_root_gpu, lTree->_root, sizeof(t_linearBvhNode) * (1024 * 1024 * 2 + 1), cudaMemcpyHostToDevice));
lTree->_root = _root_gpu;

t_linearBVH* dev_lTree = 0;
checkCudaErrors(cudaMalloc((void**)&dev_lTree, sizeof(t_linearBVH)));
checkCudaErrors(cudaMemcpy(dev_lTree, lTree, sizeof(t_linearBVH), cudaMemcpyHostToDevice));

//把颜色和深度信息复制给GPU
int N = target._cx * target._cy * 3;
float* dev_xyzs = 0;
float* dev_rgbsNew = 0;
float* dev_nearest = 0;

checkCudaErrors(cudaMalloc((void**)&dev_xyzs, N * sizeof(float)));
checkCudaErrors(cudaMalloc((void**)&dev_rgbsNew, N * sizeof(float)));
checkCudaErrors(cudaMalloc((void**)&dev_nearest, N / 3 * sizeof(float)));

checkCudaErrors(cudaMemcpy(dev_xyzs, target._xyzs, N * sizeof(float), cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(dev_rgbsNew, target._rgbsNew, N * sizeof(float), cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(dev_nearest, target._nearest, N / 3 * sizeof(float), cudaMemcpyHostToDevice));
```

接着再搬函数。一步步给函数前面加上__device__ __host__，到最后发现其实要做的只是把这个函数改成 GPU 版本。

```
__host__ __device__ void query(t_linearBvhNode *root, kBOX &bx, xyz2rgb &dst, xyz2rgb &minPt, double &minDist) {

    if (!_box.overlaps(bx)) {
        return;
    }

    if (isLeaf()) {
        if (bx.inside(_item.xyz())) {
            double dist = vdistance(_item.xyz(), dst.xyz());
            if (dist < minDist) {
                minDist = dist;
                minPt =_item;
            }
        }
        return;
    }
    root[_left].query(root, bx, dst, minPt, minDist);
    root[_right].query(root, bx, dst, minPt, minDist);
}
```

查资料得知目前 GPU 不支持递归，因此最后要做的工作其实就是把这个查找函数变成非递归版本。思路也比较明显，把一棵二叉树的查找从递归变成遍历，只要用一个 stack 来辅助就可以了。

于是我的想法是：

记送进来找最近点的那个点为 dst，建一个 stack，从 IBVH 的根节点开始查找。

伪代码如下：

```
while(该节点不是叶子节点且stack不为空){
    if(dst在该节点的包围盒中){
        if(该节点是叶子节点){
            if(dst扩张后的包围盒包含了该叶子节点中的点){
                更新距离最小值;
                记录该距离最小的点;
            }
            if(stack不为空){
                从stack中pop一个节点出来，去查它的右儿子;
            }
        }
        else{
            把当前节点压到stack中，接着查它的左儿子;
        }
    }
    else{
        if(stack不为空){
            从stack中pop一个节点出来，去查它的右儿子;
        }
        else return;
    }
    return;
}
```

最后完成的代码如下：

```
__host__ __device__ void query2(t_linearBvhNode *root, kBOX &bx, xyz2rgb &dst, xyz2rgb& minPt, double& minDist,int tid) {

        t_linearBvhNode _stack[32];

        float dist;
        int num_stack = 0;//元素个数

        t_linearBvhNode pCur = root[0];//从层次包围盒顶端开始搜

        while (!(pCur.isLeaf() && (num_stack == 0))) {
            //如果在这个范围内，判断一下是不是叶子节点，是就操作一下
            if (pCur._box.overlaps(bx)) {
                if (pCur.isLeaf()) {
                    if (bx.inside(pCur._item.xyz())) {
                        dist = vdistance(pCur._item.xyz(), dst.xyz());
                        if (dist < minDist) {
                            minDist = dist;
                            minPt = pCur._item;
                        }
                    }
                    //去找找右儿子们
                    if (!(num_stack == 0)) {
                        pCur = _stack[num_stack - 1];
                        num_stack--;
                        pCur = root[pCur._right];
                    }
                    else return;
                }//如果不是叶子节点就要继续查找它的儿子
                else {
                    _stack[num_stack] = pCur;
                    num_stack++;
                    pCur = root[pCur._left];
                }
            }
            else {
                if (!(num_stack == 0)) {
                    pCur = _stack[num_stack - 1];
                    num_stack--;
                    pCur = root[pCur._right];
                }
                else return;
            }
        }
    return;
}
```

运行以后发现，虽然可以成功运行得到图片，然而运行速度却很慢。这种情况下要得到结果需要 282 秒，才比 CPU 上快了一倍而已。

```
Start loading...
Loading done...: 0.12029 s
Start build_bvh...
build_bvh done...: 1.86424 s
Start querying...
Querying done... elapsed time: 282910.000000 ms
end
```

【关于为什么这么慢的猜测】

① 目前的 block 数量为 2048，每个 block 中的线程数量为 512。我觉得可能跟这个数据有关系，试图改成 1024 个 block 中各有 1024 个线程。

↓

结果程序就退出了……猜测是空间不够。

② 程序运行的时候我用任务管理器监测电脑性能，发现 GTX1060 那块显卡的利用率一直是 0。于是我到显卡面板里设置，强制让 VS 使用高性能显卡。

↓

运行还是没有加快，显卡占用率也没有改变。

③ 问了同学，说有可能是函数调用太深的问题。于是我把好多层的 query 函数都合到了一起，直接在一个函数里做查询，不让它一层一层调用下去。

↓

根本看不出来到底有没有加快……耗时数量级依旧差不多。

④ 最后在同学的帮忙下重构了查询部分的代码，把前面用的结构体数组换成了指针数组（我也尝试过把存储 Node 结构体变为存储 int，结果从 282 秒优化到了 160 秒），具体的查询逻辑没有修改。修改后的代码如下：

```
__device__ vec3f traverseRecursive(t_linearBVH *bvh,xyz2rgb input){

    t_linearBvhNode* stack[32];
    t_linearBvhNode** stackPtr = stack;
    *stackPtr++ = NULL; // push
    vec3f input_pos = input._xyz;

    t_linearBvhNode* root = bvh->_root;


    //t_linearBvhNode* bvh_start = bvh - numObject;
    kBOX point(input._xyz);
    point.dilate(1.5f);

    // Traverse nodes starting from the root.
    t_linearBvhNode* node = root;
    float Nearest = 2000;
    int Nearest_id = -1;
    vec3f rgb = vec3f(1, 0, 0);

    do
    {
```

```
int childL = node->_left;
int childR = node->_right;
t_linearBvhNode left = root[childL];
t_linearBvhNode right = root[childR];
bool overlapL = (left._box.overlaps(point));
bool overlapR = (right._box.overlaps(point));

if (overlapL && left.isLeaf())
{
    if (point.inside(left._item.xyz())) {
        float dist = vdistance(left._item.xyz(), input_pos);

        if (dist < Nearest) {
            Nearest = dist;
            Nearest_id = childL;
        }
    }
}
```

```
    if (overlapR && right.isLeaf())
    {
        if (point.inside(right._item.xyz())) {
            float dist = vdistance(right._item.xyz(), input_pos);
            if (dist < Nearest) {
                Nearest = dist;
                Nearest_id = childR;
            }
        }
    }

    bool traverseL = (overlapL && !left.isLeaf());
    bool traverseR = (overlapR && !right.isLeaf());

    if (!traverseL && !traverseR)
        node = *--stackPtr; // pop
    else
    {
        if (childL == 0 || childR == 0) {
            printf("error\n");
        }
        int id= (traverseL) ? childL : childR;
        node = root + id;;
        if (traverseL && traverseR)
            *stackPtr++ = root+childR; // push
    }
} while (node != NULL);
```

```
    if (Nearest_id > 0) {
        rgb = root[Nearest_id]._item.rgb();
    }
    return rgb;
}
```

↓

结果运行确实变快了，现在的时间如下：

```
Start querying...
Querying done... elapsed time: 117261.156250 ms
end
```

虽然还是没有达到加速十倍的效果，但是我询问了一下那个加速十倍的同学台式上的显卡是 2070，所以我觉得这个计算能力的差距可能是还可以接受的吧。

附：老师的 GPU 版本的运行时间如下：


C:\Users\root\Desktop\gpuComb-release-pkg>C:\Users\root\Desktop\gpuComb-release-pkg\gpuComb.exe
delta = 1.500000
Start loading...
Loading done...: 0.10530 s
Start build_bvh...
cpu size = 208, 144, 48
gpu size = 208, 144, 48
build_bvh done...: 2.86315 s
Start rescanning...
kernelUpdate: 131.33036 s (131329.95313 ms)
Rescaning done...: 131.34404 s