GPU 加速图像深度恢复

姓名：余海林　　学号：21721039

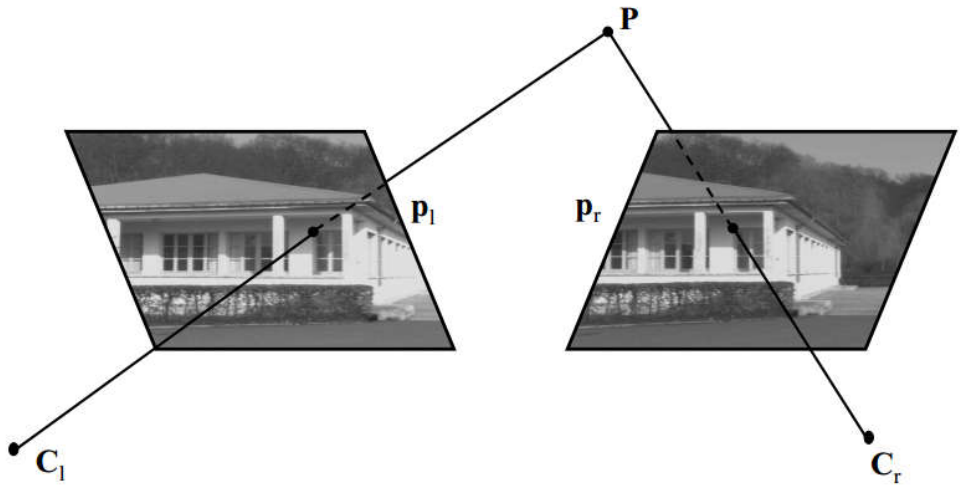
摘要

GPU（Graphics Processing Unit）图形处理单元，有很多的 ALU 组成，可以同时并行非常的多的线程， 相对于 CPU（Central Processing Unit）更适合做并行计算。CUDA(Compute Unified Device Architecture)，是显卡厂商 NVIDIA 推出的运算平台，支持 C、C++等编程语言。本文用 CUDC C 加速立体视觉中图像的深度恢复算法 SAD，分析常量内存、共享内存、流和动态并行带来的加速效果，以及通过组合不同的方式给出最大化的加速效果。

1 背景

为了完整性，在这里简要介绍一下图像的深度恢复，深度恢复是立体视觉中最关键的一部分。将三维物体投影的二维图像上，物体的深度信息丢失，图像的深度恢复就从二维图像上恢复物体的深度信息。深度恢复有很多算法，有基于传统的几何来做的，也有用最近很火的深度学习来做的，这里采用几何方法。几何方法需要用多幅图像，利用同一物体不同视角的图像以及采集这些不同图像时相机间的位置姿态关系来完成深度恢复任务。

现假设有同一场景的两幅图像，来自两个相机或同一相机不同拍摄点所拍摄，关系如下图[1]：



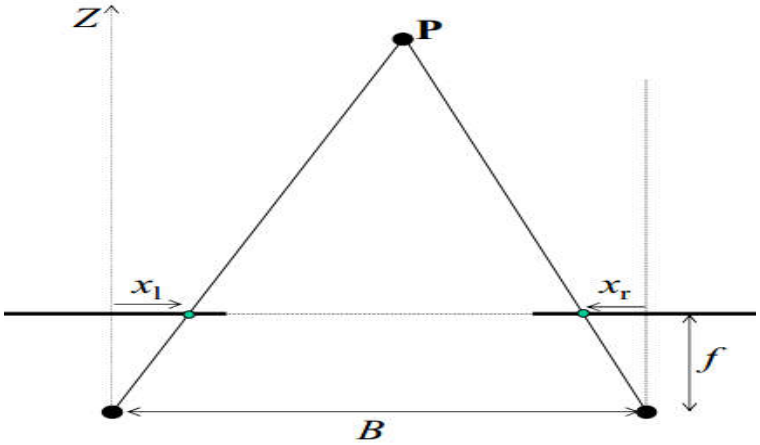
其中 C1 和 C2 为相机光心，pl 和 pr 是两幅图像上同一三维点 P 的像，在无误差情况下这五个点是处于同一平面上的。那么知道了 C1 和 C2 的距离 pl 和 pr 的距离再加上相机内参，三维点 P 的深度通过三角形的相似性质也就能计算出来了。C1 与 C2 的距离也可通过相机外参计算出来，那么在给定相机内外参的情况下，主要求 pl 和 pr 的距离。一般分两步：

1）求出匹配点对

就是找出某三维点在两幅图像上的像素点对，由对极约束[2]可知，对应的像素点只需在对极线（epipolar line）上搜索即可，通常为简化操作先对图像做矫正（rectification）处理，处理后对极线平行于 x 轴。这里采用的是 MiddleBury 的数据集[3]，图像都是矫正过的。根据匹配方式不同，有点匹配和窗口匹配，这里采用窗口匹配。根据匹配代价函数不同，有 SSD(Sum of Square Difference)，SAD(Sum of Absolute Difference)，NC(Normalized Correlation)。SSD 对噪声敏感，NC 计算量大，这里采用 SAD,

$$\psi(I_l(x,y),I_r(x+d,y))=\left|I_l(x,y)-I_r(x-d,y)\right|$$

2）恢复点的三维结构



如上图所示，P 的深度 $Z=\frac{Bf}{d}$,其中 $d=x_l-x_r$,f 为相机焦距，B 为两相机中心距离。相机的内外参已知，则 Bf 确定，则深度 Z 与 d 成反比。三维点 P 的 X、Y 坐标也可根据相应公式求出，这里为了把重点放在 CUDA 的加速方面，最终只求出视差 d，输出视差图。

2 算法及加速分析

根据上面深度恢复的简单描述，最终得到一个简化版本的深度恢复算法，算法描述如下：

算法 1：深度恢复 SAD

```
Input: image1, image2
Output: deepImage
For y = 1 : image1.rows
    For x = 1 : image1.cols
    Sbest = MAX;
    For i = 1 : image2.cols
        S = SAD(image1(x, y), image(i, y));
        If(S < Sbest)       Sbest = S,    Xopt = i;
    deepImage(x, y) = Xopt-x;
```

从上面的算法可以看出，算法共有三个循环，最简单的思路就是利用 CUDA 并行处理第一幅图像上的每个点，即每个点开一条线程，但每条线程内的点还要处理一个循环--从一维对极线上搜索最优匹配点。

对于每条线程内的循环，为了进一步提高并行性，从而提高速度可以用动态并行来实现。除此之外，还可以借助常量设备内存、共享设备内存和流等手段进行加速。下面实验中具体分析。

3 实验

设备基本信息

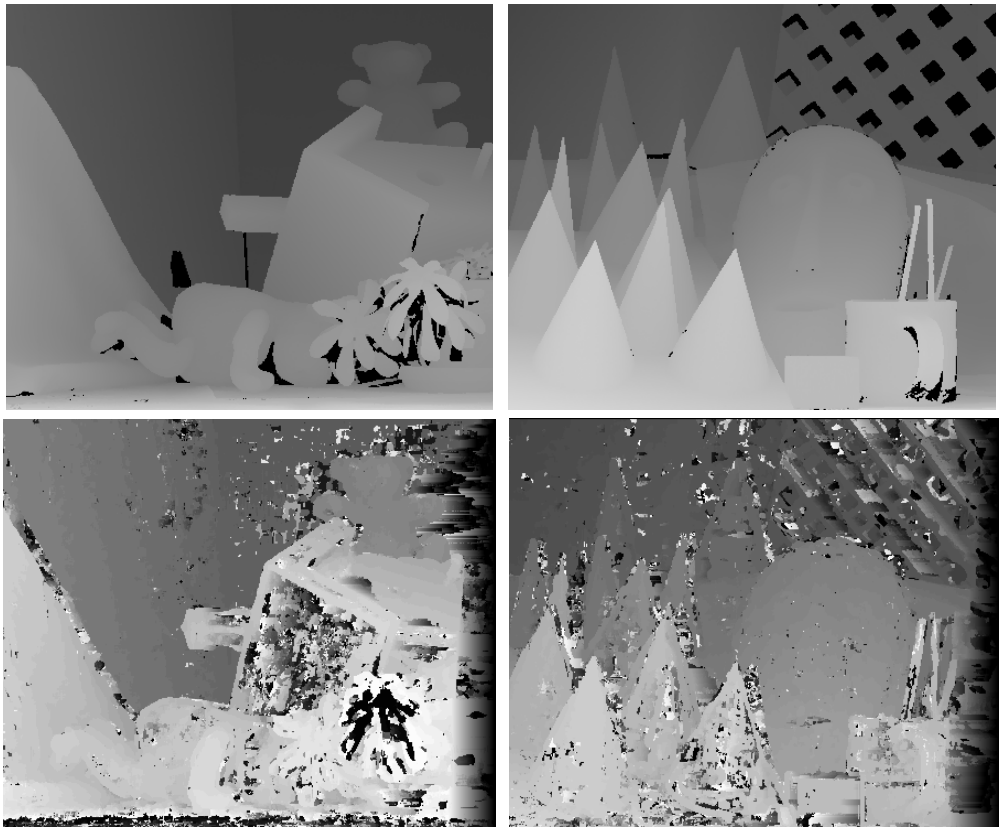
- CPU : Intel(R) Core(TM) i7-2600
- GPU : NVIDIA GeForce GTX 970
- OS : Windows 10 专业版
- 开发工具：Visual Studio 2015 企业版

```
CUDA Driver Version / Runtime Version          9.1 / 8.0
CUDA Capability Major/Minor version number:    5.2
Total amount of global memory:                 4096 MBytes (4294967296 bytes)
(13) Multiprocessors, (128) CUDA Cores/MP:     1664 CUDA Cores
GPU Max Clock rate:                            1266 MHz (1.27 GHz)
Memory Clock rate:                             3505 Mhz
Memory Bus Width:                              256-bit
L2 Cache Size:                                 1835008 bytes
Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 65536
Warp size:                                     32
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch:                          2147483647 bytes
Texture alignment:                             512 bytes
Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
Run time limit on kernels:                     Yes
Integrated GPU sharing Host Memory:            No
Support host page-locked memory mapping:       Yes
Alignment requirement for Surfaces:            Yes
Device has ECC support:                        Disabled
CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
Device supports Unified Addressing (UVA):      Yes
Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
```

3.1 只用 CPU

在只用 CPU 的版本里，release 版本的程序跑了 1785ms。方便对比，后面的默认都是基于 release 版本的跑出的时间。实验跑了两组图像，大小为 375*450，分别如下：

上面分别是实验用的两组图像，左边是场景的左视图，右边是场景的右视图，下面给出 ground truth 和实验结果：



上面两幅是 ground truth，下面两幅是实验结果，所用窗口大小为 5*5，后面用 GPU 加速后的结果和上面的结果一样，所以不再展示。

3.1 最简单的方法

最简单的方法，由于每个像素点的深度的计算互补影响，故将其中一幅图像的每个像素点放在 GPU 中并行计算其深度。根据 block 和 thread 的不同大小，获得数据如下：

Block	Thread	Time(ms)
512*512	1*1	313.278687
256*256	2*2	149.412292
128*128	4*4	39.278305
64*64	8*8	16.886047
32*32	16*16	10.226144
16*16	32*32	7.451360

从表中可以看出，利用 GPU 这种最简单的方式加速最终能达到 7.45ms，相比于只用 CPU 的 1700 多 ms, 加速将近 250 倍。并且可以看出随着线程数量的增加，时间在加快。但当线程数超过 1024 就会出现错误。

3.2 Shared Memory & Constant Memory

上面的方法没有利用任何存储方面的特性进行加速，考虑 shared memory 是芯片内部的存储器，速度非常快，对那些频繁访问的数据可以放入 shared memory 进行进一步的加速，分析上面算法一，可以看见对于第一幅图像中的每个像素点，都要在第二幅图像中对应的一行像素上进行 SAD 运算。所以考虑把第二幅图像加入 shared memory，把线程块和线程都设置为一维，如下所示：

```
getDepthMap_shared <<<375, 450>>> (devData[0], devData[1], devData[2], imL.cols, imL.rows);
```

其中 375 为图像的高度，450 为图像的宽度，这样做对每个线程块中的每个线程只需拷贝自己所在位置的上下若干个像素点（依据窗口大小而定）至 shared memory 即可，并且整个块在拷贝完后其所有的计算都在 shared memory 中进行。

Constant memory 主要利用广播和 cache 来减少访存次数和提高速度的。Constant memory 大小为 64KB， 由于所使用的图像大小 450*375B 约 163KB，一次装入 constant memory 不行，所以分三次来做，这里只做第一幅图像。

结合 shared memory 和 constant memory，即第一幅图像放 shared memory，第二副图像放 global memory。运行时间如下表所示：

Image1	Image2	Time(ms)
global	shared	4.214176
constant	global	6.421354
constant	shared	3.845235

从上表可以看出，只用 constant memory 或只用 shared memory，相对于只用 global memory 速度都会有提升，shared memory 提升的效果明显要比 constant memory 好。同时用 shared memory 和 constant memory，速度提升至 3.8ms，相对于只用 global memory 时最好情况 7.4ms，速度提升了近两倍。

3.3 流

除了运用内存特性，算法一还可以用流来进一步来提高并行性，下面是通过不同数量的流加速的效果。最简单的思路就是按照图片的行进行划分，每个流处理若干行，不同流的数量程序运行的时间如下表所示：

stream	1	2	3	4	5	6
time（ms）	6.061472	6.048448	5.006784	5.032512	5.030656	5.132672

如上表所示，随着流的数目的增加时间有所加快，但最后趋于平稳，当流数为 15 时（表中未给出），时间也在 5ms 多。

3.4 动态并行

从算法可以看出，利用上面那种方式，每条线程还是会处理一个比较大的循环，即图像宽度这么大的循环。这里主要是求同一横坐标不同纵坐标下的窗口与当前窗口的 SAD 值，然后求出使得该值最小的那个横坐标，即

$$x^{*} = \arg\min_{x} \sum_{x=0}^{img.cols} |I(X,Y) - I(x,Y)|$$

要想进一步的提高并行性，可以通过 CUDA 的并行性来并行处理这个循环，即在核函数里继续调用核函数。即对每个线程再开 img.cols 个线程用来计算 SAD 值。

计算完 SAD 值外，还要从所有值中选取最小的一个。可以值安排一条线程串行的选出值最小的那个，也可以采用归约的方法来并行的选出值最小的那个横坐标。下面比较两种方式的加速效果。

Type	Time(ms)
serialization	6.855230
reduction	3.154260

上表是只用动态并行时，若采用串行求最小值下标，则并没有获得多少加速。当采用归约的方式时，有比较明显的加速，但需要额外设置一个下标数组用来记录较小值的下标。

```
int step = 450;
do {
    step = (step + 1) / 2;
    if (x + step < 450 && S[x] > S[x + step]) {
        S[x] = S[x + step];
        idx[x] = idx[x + step];
    }
} while (step != 1);
if (idx == 0) *xopt = idx[0];
```

最后 idx[0]中存储的就是最小 SAD 值的小标。注意，使用动态并行时，必须要在命令行编译，不能在 VS 下直接编译（可能是我没有找到合适的方法）。编译命令如下：

nvcc -arch=sm_35 -rdc=true main.cu

4 总结

合理的使用 GPU 的特性，能使偏于计算的程序在速度上得到很大的提升。在这个的程序中，最终相对于 CPU 的加速达到了近 700 倍，原本恢复一幅图像的深度需要将近 1800ms，即 1.8 秒，这对实时的恢复视频序列中各帧的深度显然是不行的，但经过 GPU 的加速，每帧的处理只需要 3ms，这完全可以满足实时的要求。但这只是一个简单的算法，恢复的效果并不是很好，像一些基于全局的算法，如 graph-cut、belief propagation 等算法效果上要比这好很多，并且也可以用 GPU 进行加速。

除此以外，上面的动态并行也可以和 shared memory 做进一步的结合以提升速度，除此之外还可以将它们与流和 constant memory 做结合。

参考文献

[1] cuda by example

[2] Scharstein, Daniel, and Richard Szeliski.“High-Accuracy Stereo Depth Maps Using Structured Light.” 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings., vol. 1, 2003, pp. 195–202.

[3] Multi-viem Geometry in Computer Version