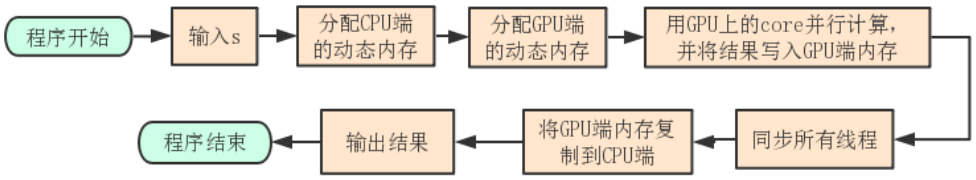
多核程序设计与实践 – 第 1 次作业实验报告

1. 程序设计逻辑

1.1 CUDA 加速计算二维高斯函数



对一个(6s+1)*(6s+1)的二维数组，每个点用一个 GPU core 计算其高斯函数，这样 GPU 中的多个 core 可以同时计算多个点的二维高斯函数。如果 core 不够，每个 core 则需要多次计算不同点的二维高斯函数。

1.2 OpenMP 加速计算二维高斯函数

对一个(6s+1)*(6s+1)的二维数组，有两层 for 循环，用#pragma omp parallel for 令多个线程同时计算多个点的二维高斯函数。

2. 参数对程序性能的影响

主要讨论 CUDA 参数对程序性能的影响。在测试中发现，影响整个程序效率的主要是数据传输而非计算过程(这一点将在第 3 部分讨论)。为了使实验效果更佳明显，我们只关注“GPU 计算”这个过程，代码如下：

```
    start = clock();
    gaussianDistribution <<<grid_dim, block_dim >>> (dev_arr,
matrix_length, pitch / sizeof(float));
    stop = clock();
    printf("RUNNING...%d\n", stop-start);
```

2.1 线程块数对程序性能的影响

把线程块数设为 GPU 上限，同时令每个线程块只包含一个线程。更具体地说，查询[4][5]知道，1 个 SM 最多容纳 32 个线程块，1 个 GPU 含 56 个 SM，因此设置 32*56*1 个线程块，每个线程块只有 1 个线程。

s=4000 时，测试结果如下：

```
jovyan@jupyter-16337153:~/hw1$ ./hw1_cuda 4000
Select Device... 24842
Allocate Cuda Memory... 259846
RUNNING...36
Synchronize Threads...206483
Copy from CUDA memory to CPU Memory...1602419
Total Time Used: 2095720
```

2.2 线程数对程序性能的影响

将每个线程块的线程数设为上限，并且整个 GPU 只设置一个线程块。更具体地说，每个线程块设置 32*32*1 个线程，grid 的大小设为 1*1*1。

s=4000，测试结果如下：

```
jovyan@jupyter-16337153:~/hw1$ ./hw1_cuda 4000
Select Device... 29217
Allocate Cuda Memory... 251086
RUNNING...48
Synchronize Threads...340014
Copy from CUDA memory to CPU Memory...1598116
Total Time Used: 2220709
```

2.3 综合考虑线程数和线程块数对程序效率的影响

将每个线程块的线程设为上限，同时设置对应的 grid 大小。具体地说，将线程块大小设为 32*32*1，grid 大小设为 16*7*1。

S=4000，测试结果如下：

```
jovyan@jupyter-16337153:~/hw1$ ./hw1_cuda 4000
Select Device... 26330
Allocate Cuda Memory... 247066
RUNNING...24
Synchronize Threads...6113
Copy from CUDA memory to CPU Memory...1612877
Total Time Used: 1894456
```

综合以上 3 个测试，可以得出结论，总线程数在一定程度上影响了计算花费的时间。GPU 里总线程数越多，计算时间相对越短。

3. 性能优化方法

主要讨论 CUDA 实现计算二维高斯函数的性能优化方法。测试代码样例如下

```
start = clock();
calculateGaussianWithCuda(arr, matrix_length);
stop = clock();
printf("Total Time Used: %d\n", stop-start);
```

3.1 cudaMalloc 和 cudaMemcpy 对性能的影响

实验过程中我们发现，影响程序运行时间的主要因素是 **GPU 内存申请**和**将数据从 GPU 内存拷贝至 CPU 内存**。二者加起来占总运行时间（不含输出）的 90%以上，**将数据从 GPU 内存拷贝至 CPU 内存**这项操作占了总运算时间（不含输出）的 80%以上。

查询资料发现[3]，如果直接使用 cudaMalloc 申请 GPU 上的一维数组来模拟二维数组，当数组特别大时，如果我们需要访问后面的某一行，就需要遍历前面的元素。“为了减小访问单行的代价，我们希望我们的每一行起始地址与第一行的地址是对齐的”[3]。同时，我们希望每次从内存载入的数据可以被更有效的使用（提高局部性），即可以被多个线程并行访问。“cudaMallocPitch 所做的事情就是：首先分配第一行的空间，并且检查它的总字节数是否是 128 的倍数，如果不是的话，就再多分配几个空余空间，使得总大小为 128 的倍数，这个一行的大小（包括补齐部分）就是一个 pitch，然后以此类推分配其他行。最后，分配的总内存要大于实际所需的内存。”

将 cudaMalloc 修改为 cudaMallocPitch，cudaMemcpy 修改为 cudaMemcpy2D 后，测试发现总运行时间在一定程度上减小，但是**GPU 内存申请**和**将数据从 GPU 内存拷贝至 CPU 内存仍然是影响程序性能的主要因素**。下图为修改后的运行时间，s=4000

```
jovyan@jupyter-16337153:~/hw1$ ./hw1_cuda 4000
Select Device... 26977
Allocate Cuda Memory... 257070
RUNNING...23
Synchronize Threads...6805
Copy from CUDA memory to CPU Memory...1662696
Total Time Used: 1956677
```

从而我们了解到，数据传输过程常常是决定程序性能的最主要因素。因此，如果需要继续提高程序性能，就需要更好地利用带宽，并且减少内存访问和数据传输的过程。

3.2 同时运行多个 GPU

在测试过程中发现，集群给每个节点分配了 4 张显卡。因此如果需要更好的程序性能，可以同时用 4 张显卡同时运算。但是处于对 3.1 的考虑(数据传输是影响程序效率的主要因素)，即使计算时间得到减少，由于 GPU-CPU 之间的带宽有限，数据传输时间不能得到有效减少，因此提交代码中未实现这个猜想。

4. 其他说明

如果有多块 GPU，cuda 程序默认使用第 0 块。若第 0 块 GPU 资源不足则报错退出。

5. 参考资料

[1] 使用 Padding（cudaMallocPitch）的二维数组
https://blog.csdn.net/fb_help/article/details/79806889
[2] 浅谈 CPU 内存访问要求对齐的原因
https://yangwang.hk/?p=773
[3] cuda 二维数组内存分配和数据拷贝
https://blog.csdn.net/yu132563/article/details/52658080
[4] NVIDIA Tesla P100 White Paper
https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
[5] CUDA-wiki https://en.wikipedia.org/wiki/CUDA