并行与分布式计算基础：作业 3 报告

吕洋

2023 年 1 月 18 日

目录

1 问题的理论分析

我们的问题来自于求解 PDE

$$\Delta u = 0 \text{ in } \Omega = (0,1)^2$$

$$u = 1 \text{ on } \partial\Omega$$

其真实解为 $u = 1$.
我们使用 27 点差分法来逼近微分算子:

$$\Delta u(i,j,k) \approx u(i,j,k) - \frac{1}{27}\sum_{i_x=-1}^{1}\sum_{i_y=-1}^{1}\sum_{i_z=-1}^{1} u(i+i_x, j+i_y, k+i_z)$$

利用多元 Taylor 展开容易知道

$$u(i,j,k) - \frac{1}{27}\sum_{i_x=-1}^{1}\sum_{i_y=-1}^{1}\sum_{i_z=-1}^{1} u(i+i_x\Delta h, j+i_y\Delta h, k+i_z\Delta h) = o(\Delta h^2), \Delta h \to 0$$

因此该方法一定收敛, 且系数只有 26 和 -1 便于实现. 通过离散获得的稀疏矩阵 A 是正定对称矩阵, 问题即为求解 $Ax = b$.
使用 MINRES 算法求解, 其收敛速度为

$$\|r_k\| \le 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \|r_0\|$$

其中 $\kappa(A)$ 是矩阵 A 的条件数.

2 使用 PyCUDA 的背景及介绍

2.1 为什么使用 python 及 PyCUDA

本次作业我选择使用 python 及 pyCUDA 求解, 是因为本次作业涉及的情景更加复杂 (例如, 相比于稀疏矩阵乘法). 我之前在计算数学和数据科学的学习中，使用过 numpy 实现类似的问题, 开发速度很快, 但当问题规模比较大时 (例如 1024×1024 及以上的精度解微分方程) 运行极其缓慢, 让人难以忍受, 因此希望借助本次作业探索一个有效的利用 GPU 并行来进行加速的一般方法, 希望结合 python 的灵活性和 CUDA 的运行效率两方面优点.

2.2 PyCUDA 介绍及与 C/C++ CUDA 对比

PyCUDA 是 Andreas Kloeckner 等人开发的一套包装 CUDA driver API 的 python 拓展包, 它具有以下特点:

- 和 C/C++ CUDA 相似, 使用 C 语言风格的 CUDA kernel, 需要对 GPU 进行内存分配, 明确进行 CPU-GPU 双向内存拷贝, 以及在给定的 block size 和 grid size 上由 host 端发起 kernel. 需要指定 GPU 上变量类型, 例如 `numpy.float64` 对应 C/C++ 中的 `double`. 使用者能获得完全的 CUDA driver API 的使用权限.
- PyCUDA 无需显式声明清除 GPU 中的变量，由程序自动解决.

- PyCUDA 也有一些无需指定 block size 和 grid size 的高级 API, 比如 `gpuarray`, 但是不能使用 C/C++ 中的 CUDA runtime API/CUDA library, 比如 CuBLAS 或者 Thrust 这些便于使用的高级 API.
- 在 python 中, 还有一些其余的方式可以(间接)使用 CUDA, 例如通过 CuPy 拓展包, pytorch 调用等等, 但 PyCUDA 可以直接使用 CUDA driver API.

3 MINRES 的并行实现

我们将在本部分介绍 MINRES 并行的代码实现和实验结果。

3.1 代码展示

串行版本的 MINRES 函数实现十分简单 (需要提前生成稀疏矩阵 A):

```
def cpu_minres(A,x0,b,maxit):
    x = np.array(x0)
    r = b - A @ x0
    p0 = np.array(r)
    s0 = A @ p0
    p1 = np.array(p0)
    s1 = np.array(s0)
    for iter in range(1,maxit):
        p2 = np.ndarray.copy(p1)
        p1 = np.ndarray.copy(p0)
        s2 = np.ndarray.copy(s1)
        s1 = np.ndarray.copy(s0)
        alpha = np.dot(r,s1)/np.dot(s1,s1)
        x = x + alpha * p1
        r = r - alpha * s1
        p0 = np.ndarray.copy(s1)
        s0 = A @ s1
        beta1 = np.dot(s0,s1)/np.dot(s1,s1)
        p0 = p0 - beta1* p1
        s0 = s0 - beta1* s1
        if iter>1:
            beta2 = np.dot(s0,s2)/np.dot(s2,s2)
            p0 = p0 - beta2* p2
            s0 = s0 - beta2* s2
    return x, r
```

对于并行版本: 首先定义一些之后要发起的 CUDA kernel:

```
mod12 = SourceModule("""
__global__
void subtract_alpha_vector(double *x,  double *y, double *alpha)
{
    int R1 = blockIdx.x * blockDim.x + threadIdx.x;
    int ind = R1;
    x[ind] = x[ind] - alpha[0]*y[ind];

}
""")
sub_alpha_gpu = mod12.get_function("subtract_alpha_vector")
```

接下来定义`gpu_minres()`加速版函数: 首先在 GPU 上分配内存空间:

```
def gpu_minres(x0,b,maxit):
   x0_gpu = cuda.mem_alloc(x0.nbytes)
    x_gpu = cuda.mem_alloc(x0.nbytes)
    b_gpu = cuda.mem_alloc(x0.nbytes)
    r_gpu = cuda.mem_alloc(x0.nbytes)
    p0_gpu = cuda.mem_alloc(x0.nbytes)
    p1_gpu = cuda.mem_alloc(x0.nbytes)
    p2_gpu = cuda.mem_alloc(x0.nbytes)
...
```

将数据从 host 拷贝至 device

```
def gpu_minres(x0,b,maxit):
...
    cuda.memcpy_htod(x0_gpu, x0)
    cuda.memcpy_htod(b_gpu, b)

    cuda.memcpy_htod(one_gpu, cst_one)
    cuda.memcpy_htod(neone_gpu, cst_neone)
...
```

之后分别调用 kernel 执行，为了节省篇幅，只展示部分代码片段：

```
def gpu_minres(x0,b,maxit):
...
        #p2 = np.ndarray.copy(p1)
        cpy_gpu(p1_gpu,p2_gpu,block=(512,1,1),grid=(16*16*16,1,1))

        #p1 = np.ndarray.copy(p0)
        cpy_gpu(p0_gpu,p1_gpu,block=(512,1,1),grid=(16*16*16,1,1))

        #s2 = np.ndarray.copy(s1)
        cpy_gpu(s1_gpu,s2_gpu,block=(512,1,1),grid=(16*16*16,1,1))

        #s1 = np.ndarray.copy(s0)
        cpy_gpu(s0_gpu,s1_gpu,block=(512,1,1),grid=(16*16*16,1,1))


        #alpha = np.dot(r,s1)/np.dot(s1,s1)
        cpu_buffer_3[0] = cal_dpdt(r_gpu,s1_gpu,gpu_buffer1,gpu_buffer2,cpu_buffer_1,cpu_buffer_2f
    )
        cuda.memcpy_htod(alpha_gpu, cpu_buffer_3)

        #x = x + alpha * p1
        plus_alpha_gpu(x_gpu,p1_gpu,alpha_gpu,block=(512,1,1),grid=(4096,1,1))

        #r = r - alpha * s1
        sub_alpha_gpu(r_gpu,s1_gpu,alpha_gpu,block=(512,1,1),grid=(4096,1,1))


        #p0 = np.ndarray.copy(s1)
        cpy_gpu(s1_gpu,p0_gpu,block=(512,1,1),grid=(16*16*16,1,1))


        #s0 = A @ s1
        spmv_gpu_64(s1_gpu,s0_gpu,block=(8,8,8), grid=(16,16,16))
...
```

其中 # 注释部分为与并行对应的串行语句。最后将结果拷贝回 host 输出。

```
def gpu_minres(x0,b,maxit):
...
    cuda.memcpy_dtoh(x,x_gpu)
    cuda.memcpy_dtoh(r,r_gpu)
    return x,r
```

3.2 并行算法设计思路及具体参数

设计并行算法时候遵循以下思路：

- 除了在循环开始前和循环结束后，最小化 host 和 device 之间的数据传递，因为 host 和 device 之间的数据传递远远慢于 register 和 gpu global memory 之间的速度.
- 尽量多使用缓冲（device）变量，减少不必要的 kernel call.
- 全部 kernel 有：SpMV_kernel_64,apxby_kernel_1d,copy_kernel,reduce0,Dot_Prod,plus sub_alpha_gpu. 具体功能与实现请参见后面部-分和源代码 `gpu_minres_v3.py`.
- 具体 kernel 发起参数为：除 spmv kernel 为 block size=$(8,8,8)$, grid size =$(16,16,16)$ 外，其余 kernel 均为 block size=$(512,1,1)$, grid size=$(4096,1,1)$.

3.3 实验环境

基于方便性和服务器占用率等原因，本次实验没有选择数院服务器，而选择了 Tesla P100 @kaggle 云服务器. 其 GPU, CPU 信息为：GPU 处理单浮点数性能为 9.3TFLOPS, GPU-CPU 数据传输速度为 32 GB/s, GPU 读取 off-chip 内存速度为 549 GB/s. CPU 为 Intel(R) Xeon(R) CPU @ 2.00GHz. 软件环境见 `README.ME`.

3.4 实验结果：加速求解

CPU: relative error=0.4569910937141935
residue=12.604018126018175
elapsed time(in seconds):10.46273946762085

GPU: relative error=0.45699109371422114
residue=12.604018126016776
elapsed time(in seconds)=0.20324681499994313
加速比为 **51.48x**
误差之间差距为 10^{-11} 左右，这是可以接受的，因为 double 数据类型在交换顺序时不满足结合律.

3.5 实验结果：准备数据

我们不能忽视进行迭代求解之前，首先需要准备数据：在 CPU 上这可以是用 A 和 x_exact 作稀疏矩阵乘向量得到 b. （需要先生成 A, 比较花时间，利用提供的生成算法（6 重 for 循环），大约 1-3 分钟才能生成）。这里我们在 CPU 上不用 matrix-free 的方式生成 b 的原因是 python 中 6 重 for 循环是极其缓慢的，而迭代中经常需要用 A 作稀疏矩阵乘法，因此准备 A 是必要的。

在 GPU 中我们直接调用 spmv64_kernel 即可得到数据 b.

CPU:0.07263612099995953s

GPU:0.007991972000127134s

若不考虑 A 的生成，加速比为 **9.089x**

若考虑 A 的生成，加速比至少 **7500x**, 即代码

```
mod2 = SourceModule("""
__global__
void SpMV_kernel_64(double *x, double *y)
{
    int R1 = blockIdx.x * blockDim.x + threadIdx.x;
    int R2 = blockIdx.y * blockDim.y + threadIdx.y;
    int R3 = blockIdx.z * blockDim.z + threadIdx.z;
    int NX = 128;


    double sum = 0.0;
    for (int bz = -1; bz < 2; bz++) {
        for (int by = -1; by < 2; by++) {
            for (int bx = -1; bx < 2; bx++) {
                if(R1 + bz >= 0 && R1+bz < NX && R2+by >= 0 && R2+by < NX && R3+bx >= 0 && R3+bx < NX)
                sum += x[(R1+bz)*NX*NX+(R2+by)*NX+(R3+bx)];
            }
        }
    }
    y[(R1) * (NX) * (NX) + (R2) * (NX) + R3] = -sum + 27.0 * x[(R1) * (NX) * (NX) + (R2) * (NX) + R3];
}

""")
```

```
def spmv(x):
    y = np.zeros_like(x)
    for i in range(N):
        for j in range(N):
            for k in range(N):
                summ = 0.0
                for bz in range(-1,2):
                    for by in range(-1,2):
                        for bx in range(-1,2):
                            if i+bz>=0 and i+bz < N and j + by>=0 and j+by<N and k+bx>=0 and k+bx<N:
                                summ = summ + x[N*N*i+N*j+k]
                y[i*N*N+j*N+k] = -summ + 27*x[i*N*N+ j*N+k]
    return y
```

的并行版本相对于串行版本的加速比。

4 CUDA kernel 的实现分析

我们对几个典型的用到的 kernel 详细分析不同实现的性能差异。

4.1 copy

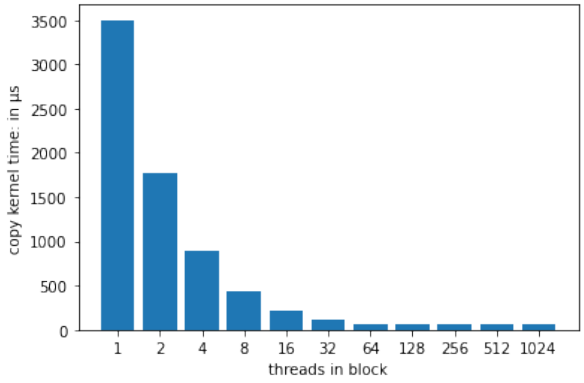

图 1: copykernel 1d 实现时间

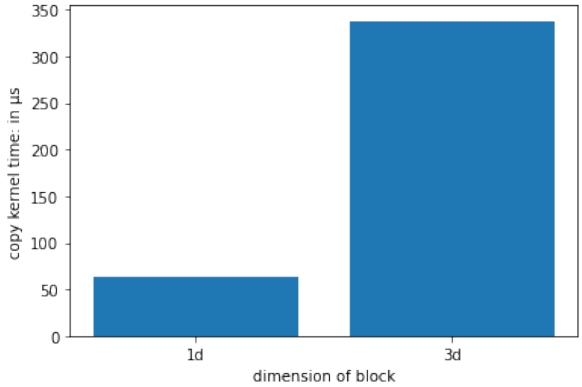

图 2: copykernel 1d vs 3d 实现时间

```
mod7 = SourceModule("""
__global__
void copy_kernel(double *src, double *dst)
{
    int R1 = blockIdx.x * blockDim.x + threadIdx.x;
    dst[R1] = src[R1];


}
""")
cpy_gpu = mod7.get_function("copy_kernel")
```

copykernel 即把数组从 GPU 一个位置搬运到另一个位置，可以说是最简单的 kernel 之一了。我们来看一下 block 和 grid 都是一维的时候，这个 kernel 执行的时间：当 block size 大于等于 64 时，时间没有显著差异，都是 64µs 左右。我们再来看看 block size 控制为 512 的情况下，使用 block 形状为 3d 数组和 1d 数组之间的时间差异：

这可以从使用 3d 数组会增加 kernel 运算量来解释。

4.2 spmv

```
mod2 = SourceModule("""
__global__
void SpMV_kernel_64(double *x, double *y)
```

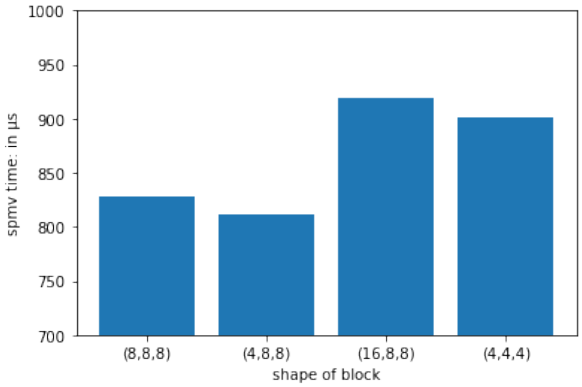

图 3: spmv 3d 形状与时间

```
{
    int R1 = blockIdx.x * blockDim.x + threadIdx.x;
    int R2 = blockIdx.y * blockDim.y + threadIdx.y;
    int R3 = blockIdx.z * blockDim.z + threadIdx.z;
    int NX = 128;


    double sum = 0.0;
    for (int bz = -1; bz < 2; bz++) {
        for (int by = -1; by < 2; by++) {
            for (int bx = -1; bx < 2; bx++) {
                if(R1 + bz >= 0 && R1+bz < NX && R2+by >= 0 && R2+by < NX && R3+bx >= 0 && R3+bx <
     NX)
                sum += x[(R1+bz)*NX*NX+(R2+by)*NX+(R3+bx)];
            }
        }
    }
    y[(R1) * (NX) * (NX) + (R2) * (NX) + R3] = -sum + 27.0 * x[(R1) * (NX) * (NX) + (R2) * (NX) +
     R3];
}

""")
spmv_gpu_64 = mod2.get_function("SpMV_kernel_64")
```

spmv kernel 负责执行稀疏矩阵乘法。这个 kernel 由于几何结构的原因，使用 3 维 block size 最合适。我们来探索 block 的形状对执行时间的影响：(注意 blocksize.x * gridsize.x = blocksize.y *gridsize.y = blocksize. z * gridsize.z) 发现形状对时间影响比较小，使用 block=(4,8,8),grid=(32,16,16) 似乎能达到最好的效果。另外这里也能看出稀疏矩阵乘法是十分耗时间的 kernel.

4.3 reduce

这里我们希望来使用 GPU 计算 $\frac{(\alpha,\beta)}{(\beta,\beta)}$ 的值。用 CUDA 计算向量内积不是一件很方便的事情，因为 CUDA 不支持不同 block 之间的同步，而 block 最大不能超过 1024. 因此一般采用的策略是使用 reduction 操作，将需要求和的数组分多次（本问题中是 2 次）reduce 成很小的数组，利用 CPU 求和再将结果传回 GPU.reduce 作为 CUDA kernel 操作实现方式是通过使用 GPU shared memory 进行树状规约：

```
mod10 = SourceModule("""
```

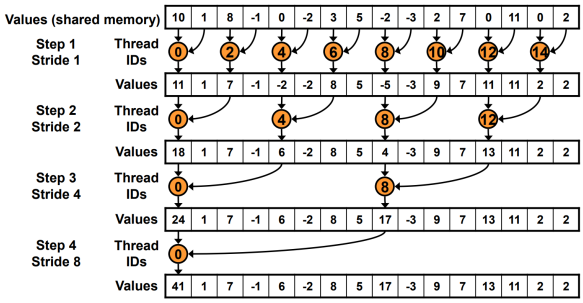

图 4: reduce 操作示意图

```
__global__ void reduce0(double *g_in, double *g_odata) {
 __shared__ double sdata[512];
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_in[i];
__syncthreads();
// do reduction in shared mem
for(unsigned int s=1; s < blockDim.x; s *= 2) {
if (tid % (2*s) == 0) {
sdata[tid] += sdata[tid + s];
}
__syncthreads();
}
// write result for this block to global mem
if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
""")
```

其实施逻辑可以参见示意图（取自 [1]）。不过 [1] 中指出了, 这样实施 reduce 操作计算密度较低, 还有很大改进空间, 其最终版本的 reduce kernel 可以将效率提高 30 倍. 这是未来值得改进的可能方向之一.

5 代码分析与继续优化

5.1 编译选项

pyCUDA 使用的是即时编译 (Just-in-time Complilation) 技术, 例如在用 python 解释器运行一个 kernel 时,实际上执行了以下指令: `nvcc --cubin -I/opt/conda/lib/python3.7/site-packages/pycuda/` 那么加入一些编译选项, 例如`-arch=sm_60 -O3`等进行优化, 对 kernel 运行时间有显著影响吗? 经过测试, 发现没有显著影响。

5.2 初步统计三部分时间

PyCUDA 对命令行 profiling(比如`nv`) 的支持目前有一些问题,因此使用程序内部的`cuda.Event()`和`time.perf_counter()`来进行分析。我们首先统计三部分时间的占比:第一部分实在 CPU 和 GPU 上分配内存空间的时间, 第二部分是将数据从 host 传给 device 和从 device 传给 host 的时间, 第三部分是运行 kernel 的时间（包含了数据从 gpu register 到 offchip global memory 之间传输的

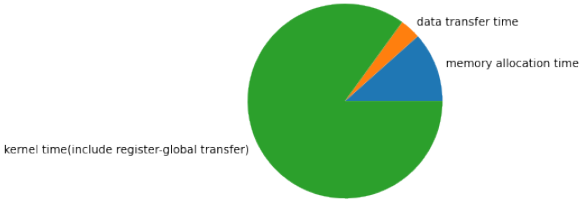

图 5: 3 部分时间占比分析

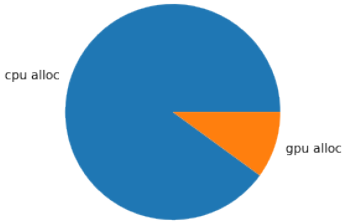

图 6: 1 部分时间 cpu:gpu

时间)。 发现第一部分时间比想象中多一些，后来发现这时间主要花费在 cpu 上开辟内存空间了(时间复杂度是 $\mathcal{O}(N)$)，因此想方法减少 cpu 上开内存。由于我们的设计，cpu-gpu 数据传输占的时间很少。接下来我们分析 kernel 调用的时间。

5.3 kernel 调用时间统计分析

这个表格总结了 iter> 1 之后，每一次循环的 CUDA kernel 调用的次数以及单次调用的时间，可以发现需要重点优化 spmv 和 dot product.

5.4 优化

结合上述的分析，我们制定了一个优化后的版本，放在`gpu_minres_v4.py`中。总结一下比较有效的优化策略有：

1. 减少将数据从 host 拷贝到 device
2. 减少在 CPU 上开辟内存空间，尽量都开在 GPU 上
3. 根据特定问题指定 kernel size 和 block size

kernel	call	time(s)
copy	5	64
cal_dpdt	3	527
plus_alpha_vector	6	43
spmv	1	800

表 1: kernel 时间分析

4. 优化耗时或调用最多的 kernel

这样可以让效率提升 10%, 如果要再提升需要在合并 kernel 的方向考虑. 但这种做法的缺点是降低了程序的灵活性.

6 总结

我们成功实现了用 cuda driver API 对 python 进行加速的目标. 对于实际应用中, 每一步循环需要比较大计算量的应用程序, 使用 cuda 能达到非常明显的加速效果. 我们还进行了广泛的分析, 计时和优化的尝试.

参考文献

[1] *Optimizing Parallel Reduction in CUDA*, Mark Harris, NVIDIA Developer Technology. Link