体系结构实验五

梁峻滔 PB19051175

源码说明

为了可以亲眼确认计算结果, 我在每个任务的源码中都加入了把计算结果和使用baseline验证的结果写到文件中. 因此在运行前需要确认运行时的目录结构满足要求.

```
.
├── output # 存放输出文件，输出文件为计算出来的矩阵C
│   ├── CPU-part
│   └── GPU-part
└── src
    ├── CPU-part
    │   ├── MatMul_AVX.c            # AVX普通矩阵乘法
    │   ├── MatMul_AVX_Blocking.c   # AVX分块矩阵乘法
    │   └── MatMul_basic.c          # 基础矩阵乘法
    ├── GPU-part
    │   ├── MatMul_gpu.cu           # 使用gpu的普通矩阵乘法
    │   └── MatMul_gpu_blocking.cu # 使用gpu的分块矩阵乘法
    └── make.sh  # 代替Makefile的bash脚本
```

CPU-part

考察基础乘法和AVX乘法的性能差异

这部分**不考察 AVX 分块矩阵乘法**, 因为 AVX 分块矩阵乘法的性能与分块参数有关. 分块参数不同, AVX 分块乘法有可能比 AVX 普通乘法快, 也有可能更慢, 因此 AVX 分块乘法用不同分块参数来跟 AVX 普通乘法比较更有参考价值, 这个在下一部分讨论.

矩阵规模为N×N. 表项为计算矩阵乘法过程的CPU clock数. 以下数据为执行`MatMul_AVX.c`所生成的程序得到, 在verify阶段测试使用baseline乘法的运行周期数, 保证两种方法是对同一组数据(A和B)进行计算.

N	baseline	AVX	加速比
64	1731	555	3.12
128	8749	1944	4.50
256	76656	16242	4.72
512	908629	125866	7.22
1024	16549629	1050499	15.75
2048	250311519	9114599	27.46

观察上表可以发现:

- 使用 AVX 计算矩阵乘法的性能显著优于基础矩阵乘法.
- 随着矩阵规模增加, 呈现出加速比越来越大的趋势, 即对于大规模矩阵乘法, AVX的性能改进更明显.

原因分析: 矩阵乘法是不冲突数据的简单加、乘运算, 显然向量操作的速度会比单个元素串行计算要快.

考察不同分块参数对性能的影响

矩阵规模为N×N, 分块大小为BLOCK_SIZE×BLOCK_SIZE. 加速比 $= \frac{AVX\ clocks}{Blocking\ clocks}$.

查看本机上的cache信息:

L1 cache所能容纳的矩阵数组长度为 $\frac{128\times1024}{4\times3} = 10922 = 104^2$, 为了体现分块的效果, 理论上适合分块的情况是**矩阵大于L1 cache, 分块小于L1 cache**. 以下我们主要测试矩阵大小$\geq 128^2$的情况.

N	BLOCK_SIZE	AVX	AVX_blocking	加速比
128	32	2188	2644	<1
128	64	4903	5120	<1
256	32	17025	20048	<1
256	64	19164	20915	<1
256	128	17405	19268	<1
512	32	139926	168135	<1
512	64	128884	146486	<1
512	128	133127	141366	<1
512	256	125667	139385	<1
1024	64	1071623	1180856	<1
1024	128	1113672	1159519	<1
1024	256	1010443	1070298	<1
1024	512	1010649	1047317	<1
2048	128	10532895	12900076	<1
2048	256	9142851	8623160	1.06
2048	512	10066513	10030414	1.004
2048	1024	11248180	10709375	1.05
4096	256	73827760	82928072	<1
4096	512	85124145	81892235	1.04
4096	1024	81639772	108096934	<1
4096	2048	71899968	77834769	<1

大部分情况下分块都比不分块还要慢. 只有当矩阵规模足够大且分块不太小又不太大时才有会微小的加速.

原因分析:

- 可能代码哪里写得不好.
- 函数调用开销覆盖了分块的优势.

CPU平台上其他矩阵乘法的优化手段

- 矩阵转置: 访问B矩阵时数据不连续(C语言按行优先来存储), 转置后可以改善数据局部性
- 循环展开: 减少分支开销
- CPU的多线程方法

GPU-part

相关说明

```
使用GPU device 0: NVIDIA GeForce MX250
设备全局内存总量:  2047MB
SM的数量: 3
每个线程块的共享内存大小: 48 KB
每个线程块的最大线程数: 1024
设备上一个线程块（Block）种可用的32位寄存器数量:  65536
每个EM的最大线程数: 2048
每个EM的最大线程束数: 64
设备上多处理器的数量:  3
```

以下两个task均采用一个线程计算矩阵C的一个元素的做法. 每个线程块大小不超过$1024 = 32^2$.

另外, 我编写的程序使用`nvprof`会出现以下情况:

```
junesnow@LAPTOP-JIKPBC87:GPU-part$ nvprof ./baseline
==4055== NVPROF is profiling process 4055, command: ./baseline
======== Profiling result:
No kernels were profiled.
No API activities were profiled.
======== Error: incompatible CUDA driver version.
junesnow@LAPTOP-JIKPBC87:GPU-part$ nvprof ./blocking
==4121== NVPROF is profiling process 4121, command: ./blocking
======== Profiling result:
No kernels were profiled.
No API activities were profiled.
======== Error: incompatible CUDA driver version.
```

说是CUDA驱动版本不对, 但是我在这方面折腾得已经够久了, 暂时不想继续在环境上面debug, `nvcc`可以正常编译, 程序也可以执行, 计算结果也经过CPU baseline验证为正确, 就暂时搁置. 下面给出相关信息:

```
junesnow@LAPTOP-JIKPBC87:GPU-part$ nvidia-smi
Sun May 29 11:02:23 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.73.05    Driver Version: 512.77       CUDA Version: 11.6     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  On   | 00000000:02:00.0 Off |                  N/A |
| N/A   46C    P0    N/A /  N/A |    248MiB /  2048MiB |      6%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

最后运行时间信息通过使用`cudaEvent`相关数据结构和API来获取.

Task1 基础矩阵乘法

编译和参数相关说明

编译:

```
$ nvcc MatMul_gpu.cu -o baseline
$ ./baseline
```

参数: 最多可输入三个参数, 必须按顺序指定

- N: 指定矩阵规模为 N × N, 默认值为256
- blocksize: 指定线程块大小为 blocksize × blocksize, 默认值为16
- bound: 随机生成A, B矩阵元素时元素的取值范围为[-bound, bound], 默认值为100.0

example: 指定矩阵规模为1024×1024, 线程块大小为32×32

```
$ ./baseline 1024 32
```

不同gridsize和blocksize对性能的影响

对于一个规模为N×N的矩阵, 我的实现下是一个GPU线程计算C矩阵的一个元素, 所以指定blocksize后, gridsize也就确定了, 即$gridsize = \frac{N}{blocksize}$.

N	gridsize	blocksize	time/ms
256	32	8	2.2272
256	16	16	2.1798
256	8	32	2.1704
512	64	8	17.2749
512	32	16	17.2408
512	16	32	17.2239
1024	128	8	137.3453
1024	64	16	137.2066
1024	32	32	137.1572
2048	256	8	970.8777
2048	128	16	962.8637
2048	64	32	962.9438

实际测试过程中, 每次执行程序都会随机生成新的矩阵元素, 并没有控制测试数据完全一样, 得到的运行时间中也有blocksize更大运行得更慢的. 得出的结论是: 不同gridsize和blocksize的影响并不显著.

我们对比CPU和GPU情形下的运行时间(单位为ms):

这里`CLOCKS_PER_SEC`的值为1000000, 因此一个clock_t即为1ms. 把CPU-part的数据搬下来得到

N	CPU baseline	CPU AVX	GPU baseline(32 blocksize)
256	76656	16242	2.17
512	908629	125866	17.22
1024	16549629	1050499	137.16
2048	250311519	9114599	962.94

从上面测试数据看来, 使用GPU计算所花费的时间远远少于仅使用CPU计算所花费的时间. 原因是显然的: GPU可以多个线程同时执行简单的运算操作.

Task2 分块矩阵乘法

保持矩阵分块大小与线程块大小一致, 即blocksize. 时间单位为ms.

> 注意:
>
> 由于在`gemm_blocking`中分配共享内存时需要用到静态的块大小参数, 所以代码中是采用宏定义指定块大小, 意味着每次修改块大小进行测试时都需要重新编译.

N	blocksize	time(blocking)	time(baseline)
256	8	0.523136	1.317888
256	16	0.784768	2.257952
256	32	1.410432	3.201536
512	8	4.842304	10.558272
512	16	5.951808	17.875328
512	32	10.806624	32.257343
1024	8	30.554144	80.245697
1024	16	42.683743	128.164352
1024	32	85.229149	219.352005

同一矩阵规模下, 分块乘法都比普通乘法快, 但是块大小增大会导致运行时间增多. 同一矩阵规模、同一分块规模下, 分块乘法比普通乘法快是合理的, 主要是因为分块乘法利用了shared memory减少了访存时间开销. 但是为什么分块规模增大会导致运行时间增多我就不是很确定了, 可能是上面我的理论分析有误.