Computer Architecture Lab5

PB19071501 李平治

实验目的

- 在CPU和GPU平台上基于矩阵乘法，开展数据级并行实验，探讨不同矩阵规模与划分参数的性能

实验环境

- CPU平台
  - macOS12.3.1
  - 2 GHz 4vIntel Core i5
  - G++(Apple clang version 13.1.6)
- GPU平台
  - Cuda 10.2.89
  - GeForce RTX 2080
  - g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
- 为了消除编译器优化的影响，实验代码均添加了`#pragma G++ optimize(0)`优化指示预处理

实验内容

I. 阶段一 CPU

1. 基础矩阵乘法

```
void gemm_baseline(float *A, float *B, float *C) {
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            for(int k=0; k<N; k++){
                C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

2. AVX矩阵乘法

```
void gemm_avx(float *A, float *B, float *C) {
    __m256 a, b, c;
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j+=8){
            for(int k=0; k<N; k++){
                c = _mm256_load_ps(C+i*N+j);
                a = _mm256_broadcast_ss(A+i*N+k);
                b = _mm256_load_ps(B+k*N+j);
                c = _mm256_fmadd_ps(a, b, c);
                _mm256_store_ps(C+i*N+j, c);
            }
        }
    }
}
```

3. AVX分块矩阵乘法

```
void gemm_avx_block(float *A, float *B, float *C){
    for(int i=0; i<N; i+=BLOCK_SIZE){
        for(int j=0; j<N; j+=BLOCK_SIZE){
            for(int k=0; k<N; k+=BLOCK_SIZE){
                for(int m=i; m<i+BLOCK_SIZE; m+=8*UNROLL){
                    for(int n=j; n<j+BLOCK_SIZE; n++){
                        __m256 c[UNROLL];
                        for(int x=0; x<UNROLL; x++){
                            c[x] = _mm256_load_ps(C+8*x+m+n*N); // c[x] = C[m][n]
                        }
                        for(int p=k; p<k+BLOCK_SIZE; p++){
                            __m256 b = _mm256_broadcast_ss(B+p+n*N); // b = B[p][n]
                            for(int x=0; x<UNROLL; x++){
                                c[x] = _mm256_add_ps(c[x],
_mm256_mul_ps(_mm256_load_ps(A+N*p+x*8+m), b)); // A[m][p]
                            }
                        }
                        for(int x=0; x<UNROLL; x++){
                            _mm256_store_ps(C+m+8*x+n*N, c[x]);
                        }
                    }
                }
            }
        }
    }
}
```

4. 三种算法性能对比

矩阵规模$\log_2 N$	Baseline运行时间	AVX运行时间	AVX分块运行时间
6	0.00167s	0.00044s	0.00027s
7	0.01174s	0.00429s	0.00147s
8	0.07758s	0.02561s	0.01273s
9	0.71398s	0.19027s	0.09678s
10	12.4893s	2.29080s	0.76403s

可以看出，在相同输入规模下，AVX分块性能最优，基础矩阵乘法性能最差。原因是AVX向量化运算起到了并行加速的效果；而AVX分块则进一步利用cache局部性，保证循环中使用的数据保留在缓存中，优化了访存运行时间。

5. AVX分块参数与性能

AVX分块算法有两个可调参数，`BLOCK_SIZE`表示AVX分块大小，`UNROLL`表示AVX循环展开大小

取$N = 2^9$,

BLOCK_SIZE	UNROLL	运行时间
32	4	0.246458s
64	4	0.089727s
128	4	0.101501s
256	4	0.099313s
256	8	0.097088s
256	16	0.086633s
256	32	0.081855s

可以看出，在`UNROLL`大小相同的情况下，`BLOCK_SIZE`和性能并不是严格的单调关系，这与缓存局部性和矩阵规模有密切联系；在`BLOCK_SIZE`大小相同的情况下，`UNROLL`越大，性能越好，这是因为展开次数越高，缓存局部性利用越好。

6. CPU平台上的其他矩阵乘法优化方法

- 基于OpenMP等框架的并行计算方法，通过多线程/多进程加速计算
- 使用循环排列技术，改变矩阵内嵌套循环顺序，或改变二维矩阵在一位数组上的展平顺序，使得矩阵访问对缓存更友好

II. 阶段二 GPU

1. 基础矩阵乘法

```
__global__ void gemm_baseline(float* A, float * B, float* C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    float temp = 0;
    if((i < N) && (j < N)){
        for(int k=0; k<N; k++){
            temp += A[i * N + k] * B[k * N + j];
        }
        C[i * N + j] = temp;
    }
}
```

2. 分块矩阵乘法

```
__global__ void gemm_cuda_block(float* A, float * B, float* C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    float temp = 0;
    if(i < (N - N%BLOCK) && j < (N - N%BLOCK)) {
        __shared__ float A_sub[BLOCK][BLOCK];
        __shared__ float B_sub[BLOCK][BLOCK];
        for(int k=0; k<N/blockDim.x; k++) {
            A_sub[threadIdx.x][threadIdx.y] = A[blockIdx.x*BLOCK*N + threadIdx.x*N + k*BLOCK + threadIdx.y];
            B_sub[threadIdx.x][threadIdx.y] = B[blockIdx.y*BLOCK + threadIdx.x*N + k*BLOCK*N + threadIdx.y];
            __syncthreads();
            for(int l=0; l<blockDim.x; l++) {
                temp += A_sub[threadIdx.x][l] * B_sub[l][threadIdx.y];
            }
            __syncthreads();
        }
        C[i*N+j] = temp;
    }else if(i < N && j < N){
        for(int k=0; k<N; k++){
            temp += A[i * N + k] * B[k * N + j];
        }
        C[i*N+j] = temp;
    }
}
```

3. 两种算法性能对比

其中分块CUDA算法取`BLOCK = 8`，

矩阵规模$\log_2 N$	基础CUDA运行时间	分块CUDA运行时间
9	6.50189ms	0.57580ms
10	18.1184ms	4.35043ms
11	59.0665ms	34.6267ms
12	340.949ms	252.301ms
13	4473.96ms	1636.05ms
14	77956.6ms	13509.6ms

分块CUDA的性能显著优于基础CUDA，这是由于前者利用了CUDA的`shared memory`缓存，该储存层次访存更快，因此性能更好。

4. 基础CUDA程序参数与性能

取$N = 2^{12}$

GRID_SIZE	BLOCK_SIZE	运行时间
128	32	1107.12ms
256	32	1126.25ms
512	32	1132.99ms
1024	32	1122.59ms
1024	16	633.674ms
1024	8	331.721ms
1024	4	412.600ms

可以看出，`GRID_SIZE`对性能的影响非常小，而`BLOCK_SIZE`对性能影响很大，这是由于矩阵使用行优先储存方式；与此同时，这两个参数与性能之间并没有显著的单调关系。当分块小而并行数大时，可能会由于高并行数性能更好，但也有可能由于线程数过多导致通信消耗过多，降低性能。

5. 分块CUDA程序参数与性能

BLOCK_SIZE	$\log_2 N$	运行时间
8	10	4.41888ms
8	11	34.7813ms
8	12	240.618ms
8	13	1679.32ms
2	10	56.9374ms
4	10	9.07997ms
16	10	8.984ms

可以看出，随着分块减小，运行时间先减少再升高，这是由于当分块变小时，并行线程增多，提高效率；而当分块过小，线程数太多而导致线程同步消耗时间过大，性能降低。

实验总结

- 本次试验通过动手进行AVX、CUDA等不同版本的矩阵乘法实现，探讨了不同矩阵规模与划分参数的性能
- 本次试验用时6h：
    - CPU部分：2h
    - GPU部分：6h
    - 实验报告：2h

附录

`./src`下各文件对应实验及编译方式：

- base_matmul.cpp 基础矩阵乘法: `g++-11 -o base_matmul base_matmul.cpp`
- avx_matmul.cpp AVX矩阵乘法: `g++-11 -mavx2 -march=native -o avx_matmul avx_matmul.cpp`
- avx_block_matmul.cpp AVX基础矩阵乘法: `g++-11 -mavx2 -march=native -o avx_block_matmul avx_block_matmul.cpp`
- cuda_matmul.cu CUDA矩阵乘法: `nvcc -o cuda_matmul cuda_matmul.cu`
- cuda_block_matmul.cu CUDA分块矩阵乘法`nvcc -o cuda_block_matmul cuda_block_matmul.cu`