计算机体系结构 lab 5

PB20111704 张宇昂

CPU

核心代码

核心代码部分：

```
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];
        }
    }
}

void gemm_avx(float *A, float *B, float *C){
    for(int i = 0; i < N; i++){
        for(int k = 0; k < N; k++){
            float temp = A[i * N + k];
            __m256 vec = _mm256_set1_ps(temp);
            for(int j = 0;j <= N - 8;j += 8){
                __m256 vec1 = _mm256_loadu_ps(B + k * N + j);
                __m256 vec2 = _mm256_loadu_ps(C + i * N + j);
                vec2 = _mm256_fmadd_ps(vec,vec1,vec2);_mm256_storeu_ps(C + i * N
+ j,vec2);
            }
        }
    }
}

void gemm_avx_block(float *A, float *B, float *C){
     for (int i = 0; i <= N - BLOCK_SIZE; i += BLOCK_SIZE) {
        for (int j = 0; j <= N - BLOCK_SIZE; j += BLOCK_SIZE) {
            for (int k = 0; k <= N - BLOCK_SIZE; k += BLOCK_SIZE) {
                for (int i1 = 0; i1 < BLOCK_SIZE; i1++) {
                    for (int k1 = 0; k1 < BLOCK_SIZE; k1++) {
                        int i2 = i + i1;
                        int k2 = k + k1;
                        float temp = A[i2 * N + k2]; // element blockA[i1][k1]
                        __m256 vec = _mm256_set1_ps(temp);
                        for (int j1 = 0; j1 <= BLOCK_SIZE-8; j1 += 8) {
                            int j2 = j + j1;
                            __m256 vec1 = _mm256_loadu_ps(B + k2 * N + j2);
                            __m256 vec2 = _mm256_loadu_ps(C + i2 * N + j2);
                            vec2 = _mm256_fmadd_ps(vec, vec1, vec2);
                            _mm256_storeu_ps(C + i2 * N + j2, vec2);
                        }
                    }
```

```
                }
            }
        }
    }
}

void gemm_verify(float *A, float *B, float *C){
    float *baseline = (float *)malloc(N * N * sizeof(float));
    for(int i = 0;i < N;i++){
        for(int j = 0;j < N;j++){
            baseline[i * N + j] = 0.0;
        }
    }
    clock_t start,end;
    start = clock();
    gemm_baseline(A,B,baseline);
    end = clock();
    printf("The running cycle of baseline is: %d",end - start);
    start = clock();
    gemm_avx_block(A,B,C);
    end = clock();
    printf("The running cycle of avx-block is: %d",end - start);
    FILE *f = fopen("../CPU-output/avx_block.txt","w");
    if (f == NULL) {
        printf("cannot open output file\n");
    }
    fprintf(f,"The baseline output matrix is: ");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            fprintf(f, "%-20f ", baseline[i * N + j]);
        }
        fprintf(f, "\n");
    }
    fprintf(f,"The avx output matrix is: ");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            fprintf(f, "%-20f ", C[i * N + j]);
        }
        fprintf(f, "\n");
    }
    for(int i = 0;i < N; i++) {
        for(int j = 0;j < N;j++){
            if(fabs(C[i * N + j] - baseline[i * N + j]) > 0.1){
                printf("Avx-block not correct!!!\n");
                free(baseline);
                fclose(f);
                return;
            }
        }
    }
    printf("Avx-block is verified as correct\n");
    fclose(f);
    free(baseline);
}
```

函数`gemm_baseline`给出了最基本的矩阵乘法的过程，就是按照矩阵乘法的定义进行计算

函数`gemm_avx`给出了通过将8个数组成向量后对矩阵乘法进行计算的过程，由于此处我们的矩阵维度都为2^n，所以向量不会存在不够组成一个向量的情况

函数`gemm_avx_block`给出了通过将8个数组成向量后对矩阵**分块**乘法进行计算的过程，由于此处我们的矩阵维度都为2^n，所以向量不会存在不够组成一个向量的情况，且**分块**也不会出现不够组成一个块的情况

结果分析

三种不同方法在不同输入规模的矩阵下的运行时间：

N\method	basic	avx	avx_block(block_size)
16	19	6	-
32	145	41	-
64	1145	333	-
128	9755	2316	2591(64)
256	79036	17183	19651(128)
512	1132972	131072	148438(256)
1024	10197696	1221800	1205193(512)

结果表明：

- avx的性能要远优于朴素的矩阵乘法，在规模增大到一定程度后可以减少一个数量级
- avx分块的性能优于朴素的矩阵乘法，但是大部分不如avx的性能

原因分析：

- 向量操作大幅减少了条件分支带来的开销，以及向量计算比朴素的计算要快很多
- 分块的函数调用带来的开销可能略大（实现问题）

不同`BLOCK_SIZE`下性能为：(我们固定$N = 512$)

block_size	time
16	199133
32	179085
64	162633
128	155407
256	148623
512	149872

分块的参数是需要仔细调整的重要参数，不同的块大小带来的性能区别很大，例如在上表中`block_size=16`所花的开销是`block_size=256`的1.5倍，经过我的实验，我认为在$N \leq 2048$时，`block_size`的最佳参数是$\frac{N}{2}$

其它优化手段

1. 使用Strassen算法降低矩阵乘法的复杂度
2. 使用循环展开技术就降低分支跳转的次数
3. 通过多线程并行来计算矩阵乘法
4. 通过矩阵转置来降低cache失效率

GPU

核心代码

核心代码部分：

```
__global__ void gemm_gpu(float *A, float *B, float *C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i >= N || j >= N) {
        return;
    }

    float temp = 0.0f;
    for (int k = 0; k < N; k++) {
        temp += A[i * N + k] * B[k * N + j];
    }
    C[i * N + j] = temp;
}

__global__ void gemm_blocking(float *A, float *B, float *C, int N) {
    __shared__ float sharedA[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float sharedB[BLOCK_SIZE][BLOCK_SIZE];

    int tx = threadIdx.x, ty = threadIdx.y;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int bi = blockIdx.x;
    int bj = blockIdx.y;


    if (i >= N || j >= N) {
        return;
    }

    float sum = 0.0;
    for (int bk = 0; bk < N / BLOCK_SIZE; bk++) {
        int Ai = bi * BLOCK_SIZE + tx;
        int Aj = bk * BLOCK_SIZE + ty;
        int Bi = bk * BLOCK_SIZE + tx;
        int Bj = bj * BLOCK_SIZE + ty;
        sharedA[tx][ty] = A[Ai * N + Aj];
```

```
        sharedB[tx][ty] = B[Bi * N + Bj];
        __syncthreads();
        for (int tk = 0; tk < BLOCK_SIZE; tk++) {
            sum += sharedA[tx][tk] * sharedB[tk][ty];
        }
        __syncthreads();
    }

    C[i * N + j] = sum;
}
```

上面两段代码是在cuda并行处理下进行朴素矩阵乘法以及分块矩阵乘法的部分

```
    float *CudaA, *CudaB, *CudaC;
    cudaMalloc((void **)&CudaA, sizeof(float) * N * N);
    cudaMalloc((void **)&CudaB, sizeof(float) * N * N);
    cudaMalloc((void **)&CudaC, sizeof(float) * N * N);
    cudaMemcpy(CudaA, A, sizeof(float) * N * N, cudaMemcpyHostToDevice);
    cudaMemcpy(CudaB, B, sizeof(float) * N * N, cudaMemcpyHostToDevice);
    cudaMemcpy(CudaC, C, sizeof(float) * N * N, cudaMemcpyHostToDevice);
    int bN = BLOCK_SIZE;
    dim3 thread_per_block(bN, bN);
    dim3 block_per_grid(N / bN, N / bN);

    cudaEvent_t start_, end_;
    float running_time;
    cudaEventCreate(&start_);
    cudaEventCreate(&end_);
    cudaEventRecord(start_, 0);

    gemm_blocking<<<block_per_grid, thread_per_block>>>(CudaA, CudaB, CudaC, N);

    cudaEventRecord(end_, 0);
    cudaEventSynchronize(end_);
    cudaEventElapsedTime(&running_time, start_, end_);
    cudaEventDestroy(start_);
    cudaEventDestroy(end_);
    printf("gpu-case running time: %lf ms\n", running_time);
    cudaMemcpy(C, CudaC, sizeof(float) * N * N, cudaMemcpyDeviceToHost);
    cudaFree(CudaA);
    cudaFree(CudaB);
    cudaFree(CudaC);
    cudaDeviceSynchronize();
    cudaProfilerStop();
```

上面的代码片段创建了三个cuda float一维数组，通过拷贝随机生成的A，B，C数组到三个cuda数组后进行计算，并用cudaEventCreate创建计时的两个变量并计算时间，最后再将结果数组拷贝回原数组并检查

结果分析

朴素的多线程矩阵乘法结果如下：

N	gridsize	blocksize	time(ms)
256	32	8	0.140320
256	16	16	0.249952
256	8	32	0.449568
512	32	16	1.621248
512	16	32	3.220928
512	64	8	0.868256
1024	128	8	6.679744
1024	64	16	12.612608
1024	32	32	24.941248

对固定规模的矩阵，时间开销会随`blocksize`的增大而增大，我认为这是因为同一个block内不同线程的访问依然比不同block之间的访问局部性更优，能明显减少访存时间。值得一提的是，当我对大规模矩阵的`blocksize`增大到1024时，矩阵会变成全0，根据查询的结果，这可能是因为超过了GPU允许的最大`blocksize`

我用如下代码测试得到结果：

```
#include <iostream>
#include <cuda_runtime_api.h>

int main() {
    int max_block_size;
    int device_id = 0;

    cudaDeviceProp device_prop;
    cudaGetDeviceProperties(&device_prop, device_id);
    cudaDeviceGetAttribute(&max_block_size, cudaDevAttrMaxBlockDimX, device_id);

    std::cout << "Max block size: " << max_block_size << std::endl;

    return 0;
}
```

```
tf-docker ~/work/GPU-files > ./a.out
Max block size: 1024
```

在保持矩阵分块大小和线程块一致的情况下，分块多线程矩阵乘法结果如下：

N	BLOCK	blocksize	time(ms)
256	32	8	0.113920
256	16	16	0.129472
256	8	32	0.191744
512	32	16	0.776096
512	16	32	1.297216
512	64	8	0.570496
1024	128	8	4.373920
1024	64	16	5.995232
1024	32	32	9.974880

随着子矩阵的大小增大，GPU分块矩阵乘法的性能单调提升。这可能是因为每个分块更充分地利用了自己的shared memory，从而减少对global memory的访问次数。