中山大学计算机学院

本科生实验报告

（2025学年春季学期）

课程名称: 并行程序设计

实验	0-环境配置与串行矩阵乘法	专业(方向)	计算机科学与技术
学号	--	姓名	--
Email	--	完成日期	2025.03.18

实验目的

- 配置环境
- 实现串行矩阵乘法
- 优化串行矩阵乘法
- 对比及初步性能分析

实验过程和核心代码

Python 实现矩阵乘法

核心代码如下:

```python
for i in range(m):
    mat1_row = mat1[i][:]
    for j in range(k):
        mat2_col = mat2[:][j]
        ans[i][j] = sum([ x*y for x,y in zip(mat1_row, mat2_col)])
```

利用python的列表切片性质直接取mat1的一行和mat2的一列进行相乘并求和, 得到ans一个位置的值.

C/C++ 实现矩阵乘法

核心代码如下:

```cpp
for(int i = 0; i < m; ++i) {
    vector<double> mat1_row = mat1[i];
    for(int j = 0; j < k; ++j) {
        for(int l = 0; l < n; ++l) {
            ans[i][j] += mat1_row[l] * mat2[l][j];
        }
    }
}
```

循环按照m, k, n的顺序, 可以得出mat1(m*n), ans(m*k)是横向访问, 而mat2(n*k)是竖向访问.

调整循环顺序

```cpp
for(int i = 0; i < m; ++i) {
    for(int l = 0; l < n; ++l) {
        double a = mat1[i][l];
        for(int j = 0; j < k; ++j) {
            ans[i][j] += a * mat2[l][j];
        }
    }
}
```

调整循环按照m, n, k的顺序进行, 使三个矩阵都横向访问, 充分利用空间局部性. 同时, 将`mat[i][l]`的值存储到局部变量中, 减少mat1的访问次数.

编译优化

编译时加上`-O3`启用3级优化

```
g++ matrixMul.cpp -O3 -o matrixMul
```

循环展开

```
int i, j, l;
for(i = 0; i < m; ++i) {
    for(l = 0; l < n; ++l) {
        double a = mat1[i][l];
        for(j = 0; j < ((k)&(~3)); j += 4) {
            ans[i][j] += a * mat2[l][j];
            ans[i][j+1] += a * mat2[l][j+1];
            ans[i][j+2] += a * mat2[l][j+2];
            ans[i][j+3] += a * mat2[l][j+3];
        }
        // 不足四列的部分
        for(; j < k; ++j) {
            ans[i][j] += a * mat2[l][j];
        }
    }
}
```

循环展开后, 可以减少循环的开销, 同时由于展开项之间没有数据依赖, 可以充分利用流水线, 多发射等指令集的加速.

Intel MKL

```
double *mat1, *mat2, *ans;
int m, n, k, i, j;
double alpha, beta;

m = 1024, n = 1024, k = 1024;
cout << "input m, n, k:" << endl;
cin >> m >> n >> k;
alpha = 1.0; beta = 0.0;
mat1 = (double *)mkl_malloc( m*n*sizeof( double ), 64 );
mat2 = (double *)mkl_malloc( n*k*sizeof( double ), 64 );
ans = (double *)mkl_malloc( m*k*sizeof( double ), 64 );
// mat1, mat2 随机赋值
// ans 初始化为0
auto begin = chrono::high_resolution_clock::now();
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            m, n, k, alpha, mat1, k, mat2, n, beta, ans, n);
auto end = chrono::high_resolution_clock::now();
auto elapsed = chrono::duration_cast<chrono::nanoseconds>(end - begin);

printf("running time: %.4f s\n", elapsed * 1e-9);
mkl_free(mat1);
mkl_free(mat2);
mkl_free(ans);
```

实验结果

令 m, k, n 都为1024, 结果如下:
浮点运算次数都为: $m \times k \times (n + n - 1) = 2.146435072 \times 10^9$

使用以下脚本粗略测试系统的峰值性能:

```
# 设置测试参数
NUM_OPERATIONS=100000000  # 进行的浮点运算次数
START_TIME=$(date +%s.%N)  # 记录开始时间

# 执行浮点运算
echo "scale=10; for (i=0; i<$NUM_OPERATIONS; ++i) {sqrt(i^2) * atan(1)}" | bc -l > /dev/null

# 计算总时间
END_TIME=$(date +%s.%N)
ELAPSED_TIME=$(echo "$END_TIME - $START_TIME" | bc)

# 计算FLOPS并转换为GFLOPS
FLOPS=$(echo "scale=10; $NUM_OPERATIONS / $ELAPSED_TIME" | bc)
GFLOPS=$(echo "scale=10; $FLOPS / 1000000000" | bc)

echo "总共执行了 $NUM_OPERATIONS 次浮点运算"
echo "总耗时: $ELAPSED_TIME 秒"
echo "平均每秒浮点运算次数: $FLOPS FLOPS"
echo "峰值性能: $GFLOPS GFLOPS"
```

峰值性能约为10.654 GFLOPS

版本	实现描述	运行时间(sec.)	相对加速比	绝对加速比	浮点性能(GFLOPS)	峰值性能百分比
1	Python	47.3629	1	1	0.045	0.42%
2	C/C++	18.5693	2.55	2.55	0.116	1.09%
3	调整循环顺序	6.3590	2.92	7.45	0.338	3.17%
4	编译优化(O3)	2.0719	3.07	22.86	1.036	9.72%
5	循环展开	5.9611	0.35	7.95	0.360	3.38%
6	Intel MKL	0.4077	14.62	116.17	5.265	49.42%

观察实验结果可以得到以下结论:

- C/C++ 比Python更高效
- 循环顺序对矩阵乘法的性能有很大影响
- 编译优化能大大提高程序性能
- 循环展开只有微小的性能提升, 主要在于减少循环开销以及利用指令级的并行
- Intel MKL库矩阵乘法API的性能非常好

实验感想

python语言的计算能力与C/C++有显著的差距. 通过空间局部性原理安排循环顺序能大大减少执行时间, 通过合理编写代码可以利用到指令级的并行优化(流水线, 多发射). 想要提升程序性能时, 可以多观察能否利用空间局部性以及指令级并行, 在编译时期可以根据需求考虑是否使用编译优化来提高程序的性能. 此外, 我们还可以多了解一些高性能的计算API, 使用这些高效的API也能够提升程序的性能.