09020328-王亮-数据结构实践报告1

Warm up

1.问题描述

模拟设计外存情况下的矩阵乘法，并对比验证Cache的miss和理论情况。

2.算法思想

如果将整个矩阵当做一维向量看待，例如对于 i，j 对应位置为 i * n + j（n为矩阵大小），那么Cache只需要存储左右边界两个值，若不在左右边界内，则每次不命中miss计数++，并更新Cache边界为 [i * n + j + 1, i * n + j + 1 + w)（w为Cache大小）（注：代码实现上从1开始计数）。

3.功能模块设计

文件模拟：

二维指针模拟存储文件

```
int** data1;
int** data2;

int** dataGenerator() {
    int** data = new int*[n];
    for (int i = 0; i < n; i++) {
        data[i] = new int[n];
        for (int j = 0; j < n; j++) {
            data[i][j] = i * n + j + 1;
        }
    }
    return data;
}
```

Cache设计：

其实需求实现并不真的需要设计一个能存储矩阵数据的Cache，只需要两个int存储左右边界判断即可。

```
int* cache1;
int* cache2;
int* cache3;

cache1 = new int[2]{0};
cache2 = new int[2]{0};
cache3 = new int[2]{0};
```

矩阵乘法与Cache计数：

```
void matrixMultiply(int** a, int** b) {
    int** c = dataGenerator();
    for (int i = 0; i < n; i++) {
```

```
        for (int j = 0; j < n; j++) {
            c[i][j] = 0;
        }
    }
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < n; k++) {
            for (int j = 0; j < n; j++) {
                res += cacheHandler(i, j, k);
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

```
void matrixMultiply2(int** a, int** b) {
    int** c = dataGenerator();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            c[i][j] = 0;
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                res += cacheHandler(i, j, k);
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

处理Cache miss计数

不命中则计数++并更新Cache

```
int cacheHandler(int i, int j, int k) {
    int ans = 0;
    if (!isHit(i, k, cache1)) {
        ans++;
        update(i, k, cache1);
        //cout << "A: " << " " << i << " " << k << endl;
    }
    if (!isHit(k, j, cache2)) {
        ans++;
        update(k, j, cache2);
        //cout << "B: " << " " << k << " " << j << endl;
    }
    if (!isHit(i, j, cache3)) {
        ans++;
        update(i, j, cache3);
        //cout << "C: " << " " << i << " " << j << endl;
    }
    return ans;
}
```

```
bool isHit(int i, int j, int* cache) {
    int pos = i * n + j + 1;
    // for (int k = 0; k < w; k++) {
    //  if (pos == cache[k]) {
    //      return true;
    //  }
    // }
    // return false;
    return pos >= cache[0] && pos < cache[1];
}
```

```
void update(int i, int j, int* cache) {
    int base = i * n + j + 1;
    // for (int i = 0; i < w; i++) {
    //  cache[i] = base + i;
    // }
    cache[0] = base;
    cache[1] = base + w;
}
```

在输入n w 后：

输出结果与各Cache miss理论计数，比较ACB三者之和与结果

```
cout << "input n: " << endl;
cin >> n;
cout << "input w: " << endl;
cin >> w;

cout << "----------ikj: ----------" << endl;
matrixMultiply(data1, data2);
cout << "Result: " << res << endl;

res = 0;

cout << "C: " << ceil(n * n * n * 1.0 / w) << endl;
cout << "A: " << ceil(n * n * 1.0 / w) << endl;
cout << "B: " << ceil(n * n * n * 1.0 / w) << endl;


cout << "----------ijk: ----------" << endl;
matrixMultiply2(data1, data2);
cout << "Result: " << res << endl;

cout << "A: " << ceil(n * n * n * 1.0 / w) << endl;
cout << "C: " << ceil(n * n * 1.0 / w) << endl;
cout << "B: " << ceil(n * n * n) << endl;

clear();
```

4.测试结果与分析

ACB分别表示在当前组合下对应的Cache miss理论值 Result代表总的实际值 只要Result = A + C + B，就代表实验结果正确

分析：

当 n （矩阵大小） 不变时，更大的 w（Cache大小）会减少miss数

当 w （Cache大小）不变时，更大的 n（矩阵大小）会提高miss数

相同 n 与 w 情况下 ikj 方式 miss 数显著小于 ijk 从理论结果计算式很容易看出这一点，二者miss数和的其他两项相同，主要区别在于：

$$ikj:\frac{n^3}{w}\qquad ijk:n^3$$

```
input n:
100
input w:
5
----------ikj: ----------
Result: 402000
C: 200000
A: 2000
B: 200000
----------ijk: ----------
Result: 1202000
A: 200000
C: 2000
B: 1e+06
```

```
input n:
100
input w:
20
----------ikj: ----------
Result: 100500
C: 50000
A: 500
B: 50000
----------ijk: ----------
Result: 1050500
A: 50000
C: 500
B: 1e+06
```

```
input n:
200
input w:
20
----------ikj: ----------
Result: 802000
C: 400000
A: 2000
B: 400000
```

```
----------ijk: ----------
Result: 8402000
A: 400000
C: 2000
B: 8e+06
```

5.实验总结

计算顺序的改变极大地减少了Cache的miss，进而减少程序运行中相当耗时的Cache IO，体现出在程序设计中对于相关方面考量的重要性。

6.源代码

```
#include<iostream>
#include<cmath>
using namespace std;
int n = 3; // size of matrix
int w = 3; // size of cache
int res = 0;
int** data1;
int** data2;
int* cache1;
int* cache2;
int* cache3;
int** dataGenerator() {
    int** data = new int*[n];
    for (int i = 0; i < n; i++) {
        data[i] = new int[n];
        for (int j = 0; j < n; j++) {
            data[i][j] = i * n + j + 1;
        }
    }
    return data;
}

bool isHit(int i, int j, int* cache) {
    int pos = i * n + j + 1;
    // for (int k = 0; k < w; k++) {
    //  if (pos == cache[k]) {
    //      return true;
    //  }
    // }
    // return false;
    return pos >= cache[0] && pos < cache[1];
}
void update(int i, int j, int* cache) {
    int base = i * n + j + 1;
    // for (int i = 0; i < w; i++) {
    //  cache[i] = base + i;
    // }
    cache[0] = base;
    cache[1] = base + w;
```

```
}
int cacheHandler(int i, int j, int k) {
    int ans = 0;
    if (!isHit(i, k, cache1)) {
        ans++;
        update(i, k, cache1);
        //cout << "A: " << " " << i << " " << k << endl;
    }
    if (!isHit(k, j, cache2)) {
        ans++;
        update(k, j, cache2);
        //cout << "B: " << " " << k << " " << j << endl;
    }
    if (!isHit(i, j, cache3)) {
        ans++;
        update(i, j, cache3);
        //cout << "C: " << " " << i << " " << j << endl;
    }
    return ans;
}
void clear() {
    for (int i = 0; i < n; i++) {
        delete[] data1[i];
    }
    delete[] data1;
    for (int i = 0; i < n; i++) {
        delete[] data2[i];
    }
    delete[] data2;
    delete[] cache1;
    delete[] cache2;
    delete[] cache3;

}
void matrixMultiply(int** a, int** b) {
    int** c = dataGenerator();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            c[i][j] = 0;
        }
    }
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < n; k++) {
            for (int j = 0; j < n; j++) {
                res += cacheHandler(i, j, k);
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
void matrixMultiply2(int** a, int** b) {
    int** c = dataGenerator();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            c[i][j] = 0;
```

```
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                res += cacheHandler(i, j, k);
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}

int main() {
  cout << "input n: " << endl;
    cin >> n;
    cout << "input w: " << endl;
    cin >> w;
    cache1 = new int[2]{0};
    cache2 = new int[2]{0};
    cache3 = new int[2]{0};
    data1 = dataGenerator();
    data2 = dataGenerator();

    cout << "----------ikj: ----------" << endl;
    matrixMultiply(data1, data2);
    cout << "Result: " << res << endl;

    res = 0;

    cout << "C: " << ceil(n * n * n * 1.0 / w) << endl;
    cout << "A: " << ceil(n * n * 1.0 / w) << endl;
    cout << "B: " << ceil(n * n * n * 1.0 / w) << endl;


    cout << "----------ijk: ----------" << endl;
    matrixMultiply2(data1, data2);
    cout << "Result: " << res << endl;

    cout << "A: " << ceil(n * n * n * 1.0 / w) << endl;
    cout << "C: " << ceil(n * n * 1.0 / w) << endl;
    cout << "B: " << ceil(n * n * n) << endl;

    clear();

}
```