lab2

讲解内容

本次只讲解基础的解法，即程序局部性的相关使用。

前置知识

程序局部性

计算机系统存储器层次结构

于。在较高档的计算机中，还可以根据具体的功能方上细划为寄存器、高
器、磁盘缓存、固定磁盘、可移动存储介质等 6 层。如图 4-1 所示，在存储
字储介质的访问速度越快，价格也越高，相对存储容量也越小。其中，寄
主存储器和磁盘缓存均属于操作系统存储管理的管辖范畴，掉电后它们有
生。固定磁盘和可移动存储介质属于设备管理的管辖范畴，它们存储的信

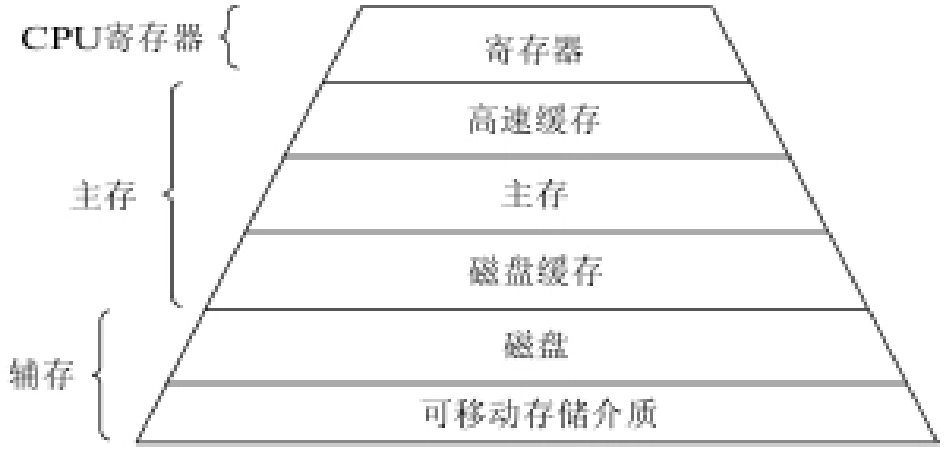


图 4-1　计算机系统存储层次示意

机系统存储层次中，寄存器和主存储器又被称为可执行存储器，存放于其
于辅存中的信息相比较而言，计算机所采用的访问机制是不同的，所需耗
同的。进程可以在很少的时钟周期内使用一条 load 或 store 指令对可执行在

程序局部性

- 时间局部性：如果一个信息项正在被访问，那么在不久的将来它很可能再次被访问。
- 空间局部性：如果一个信息项正在被访问，那么与它相邻的信息项很可能也会被访问。

```
root@fdbbae811068:/home/easy_lab# getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 49152
LEVEL1_DCACHE_ASSOC                12
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  1310720
LEVEL2_CACHE_ASSOC                 20
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  40894464
LEVEL3_CACHE_ASSOC                 12
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC
LEVEL4_CACHE_LINESIZE
```

平台相关

所有数据的单位均为Byte

一级缓存分为指令缓存和数据缓存两种，分别使用ICACHE、DCACHE表示

SIZE指该级缓存的总大小

LINESIZE表示该级缓存的cacheline大小，也就是该级缓存向低一级缓存访问时一次性抓取的数据量大小

ASSOC指该级缓存组相联的组数

测试大小

矩阵大小	Byte（总）	Byte（行/列）
512*512	2MB	4KB
1024*1024	8MB	8KB
2048*2048	32MB	16KB

缓存级数	缓存大小	缓存行大小	缓存组相联数
1	32KB	64B	12
2	1.25MB	64B	20
3	39MB	64B	12

矩阵计算算法及优化

矩阵基本乘法

根据定义直接写代码，用A矩阵第i行的每个元素乘以B矩阵第j列的每个元素，然后求和。先遍历A的行，再遍历B的列。A和B的每一个元素会被遍历n次。

很明显A的行是连续的，B的列是不连续的，所以这个算法的空间局部性不好。

```
void matrix_multiplication(double matrix1[N][M], double matrix2[M][P],
                           double result_matrix[N][P]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < P; j++) {
            result_matrix[i][j] = 0;
            for (int k = 0; k < M; k++) {
                result_matrix[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

交换循环顺序

先假设一行/列可以完整存在于缓存中。将第二层循环和第三层循环交换。

相当于对于A每一行，选取B的一行，对A这行的每个元素遍历相乘B这行的元素，把结果加到C的对应位置。

```
void matrix_multiplication(double matrix1[N][M], double matrix2[M][P],
                           double result_matrix[N][P]) {
    for (int i = 0; i < N; i++) {
        for (int k = 0; k < M; k++) {
            for (int j = 0; j < P; j++) {
                result_matrix[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

优化-一次计算4组元素

B的每一列不止与A的一行相乘，第一层循环步长为4，这样B的每一列可以与A的4行相乘。

同时尽可能的将循环次数减少，减少循环带来的开销。

```
void matrix_multiplication(double matrix1[N][M], double matrix2[M][P],
                           double result_matrix[N][P]) {
    for (int i = 0; i < N; i+=4) {
        for (int j = 0; j < P; j++) {
            for (int k = 0; k < M; k+=4) {
                result_matrix[i][j] += matrix1[i][k] * matrix2[k][j];
                result_matrix[i+1][j] += matrix1[i+1][k] * matrix2[k][j];
                result_matrix[i+2][j] += matrix1[i+2][k] * matrix2[k][j];
                result_matrix[i+3][j] += matrix1[i+3][k] * matrix2[k][j];

                result_matrix[i][j] += matrix1[i][k+1] * matrix2[k+1][j];
                result_matrix[i+1][j] += matrix1[i+1][k+1] * matrix2[k+1][j];
                result_matrix[i+2][j] += matrix1[i+2][k+1] * matrix2[k+1][j];
                result_matrix[i+3][j] += matrix1[i+3][k+1] * matrix2[k+1][j];
                ......
            }
        }
    }
}
```

优化-使用寄存器

寄存器是比缓存更快的存储器，一般可以用来存储单个变量，以此获得更快的变量访问速度,使用寄存器作为中间变量，将累和结果加入数组指定位置。可以使用访问寄存器将内存访问减少到最小。

现代编译器会自动优化变量的寄存器分配，无需手动指定，但寄存器风格的声明和使用变量可以让寄存器的自动分配更加明显。

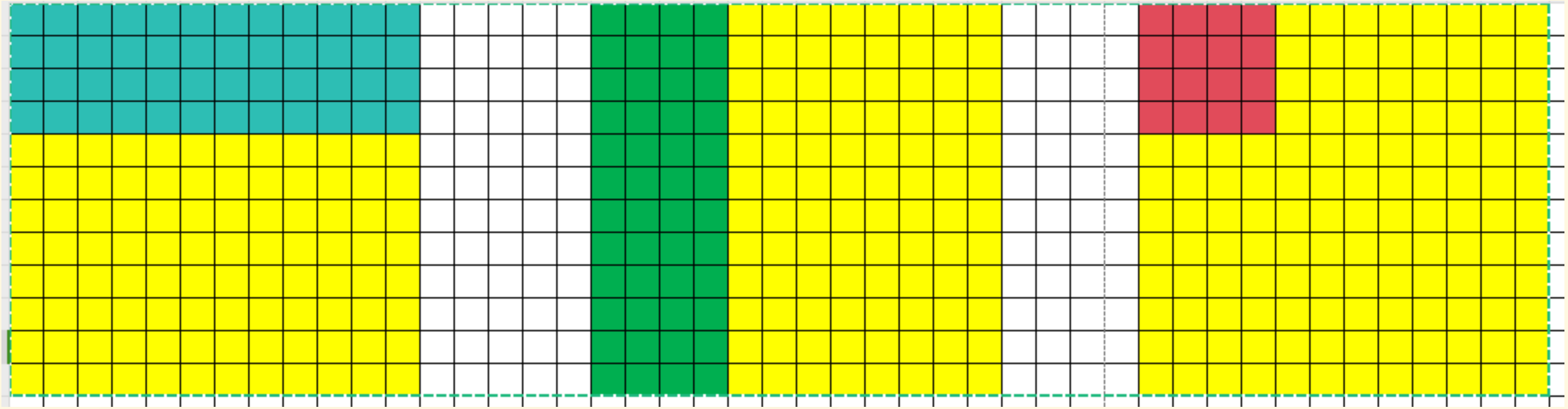
现代计算机的寄存器数目比较多，适当增大单次循环的计算量，可以使寄存器不重复分配的同时减少循环带来的开销。

```
void matrix_multiplication(double matrix1[N][M], double matrix2[M][P],
                           double result_matrix[N][P]) {
    for (int i = 0; i < N; i+=4) {
        for (int j = 0; j < P; j++) {
            compute4*1(matrix1, matrix2, result_matrix, i, j, P);
        }
    }
}
```

优化-一次计算16组元素

既然寄存器的数目足够，我们也可以将第二层循环的步长也增加到4，这样每次循环可以计算16组元素。

```
void matrix_multiplication(double matrix1[N][M], double matrix2[M][P],
                           double result_matrix[N][P]) {
    for (int i = 0; i < N; i+=4) {
        for (int j = 0; j < P; j+=4) {
            compute4*4(matrix1, matrix2, result_matrix, i, k, P);
        }
    }
}
```



优化-分块

我们之前的优化都是基于缓存可以直接放下一行元素这样的假设，但是回顾我们的缓存结构，这个假设是不成立的。

矩阵大小	Byte（总）	Byte（行/列）
512*512	2MB	4KB
1024*1024	8MB	8KB
2048*2048	32MB	16KB

缓存级数	缓存大小	缓存行大小	缓存组相联数
1	32KB	64B	12
2	1.25MB	64B	20
3	39MB	64B	12

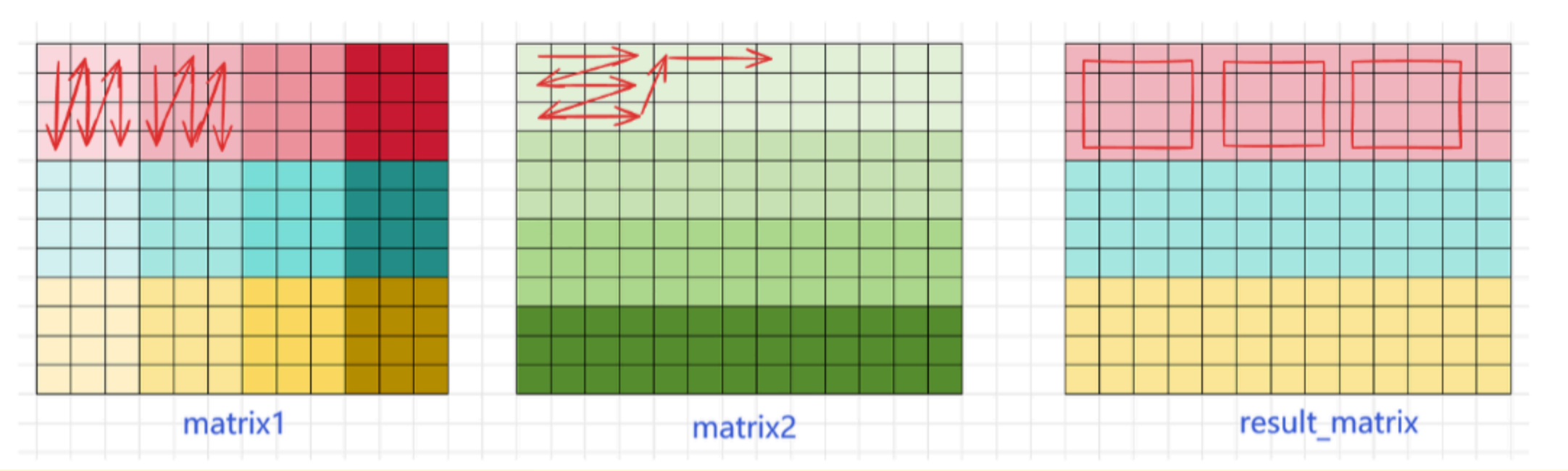
很显然我们的一级缓存不可能放下一行，而二级缓存放不下一个完整的矩阵，分块计算使我们充分利用了缓存的空间局部性，将矩阵分块，每次计算一个小块，这样每次计算的数据都在缓存中。

```
void matrix_multiplication(double matrix1[N][M], double matrix2[M][P],
                           double result_matrix[N][P]) {
    for (int i = 0; i < N; i+=BLOCK_SIZE) {
        for (int j = 0; j < P; j+=BLOCK_SIZE) {
            for (int k = 0; k < M; k+=BLOCK_SIZE) {
                for (int ii = i; ii < i+BLOCK_SIZE; ii++) {
                    for (int jj = j; jj < j+BLOCK_SIZE; jj++) {
                        for (int kk = k; kk < k+BLOCK_SIZE; kk++) {
                            result_matrix[ii][jj] += matrix1[ii][kk] * matrix2[kk][jj];
                        }
                    }
                }
            }
        }
    }
}
```


matrix1
matrix2
result_matrix
matrix1
matrix2
result_matrix

优化-使分块具有局部性

一维化后转置



优化-多线程