Project4 矩阵加速

Name:郭一潼
SID:11911702
CS205 Project4 Report
代码链接如下https://github.com/Godblessmycode1/CS205

介绍

Project要求介绍

1. 只能使用c语言
2. 加速矩阵
3. 矩阵储存浮点数

Project完成情况介绍

1. 探究了矩阵大规模情况下时间非线性增加原因
2. 调研了加速原理
3. 实现了omp,simd以及模拟cache尝试优化

开发环境

- x86_64
  - vscode (version 1.71)
  - WSL (version 2)
  - Ubuntu(22.04)
  - g++(11.2.0)

文件结构介绍

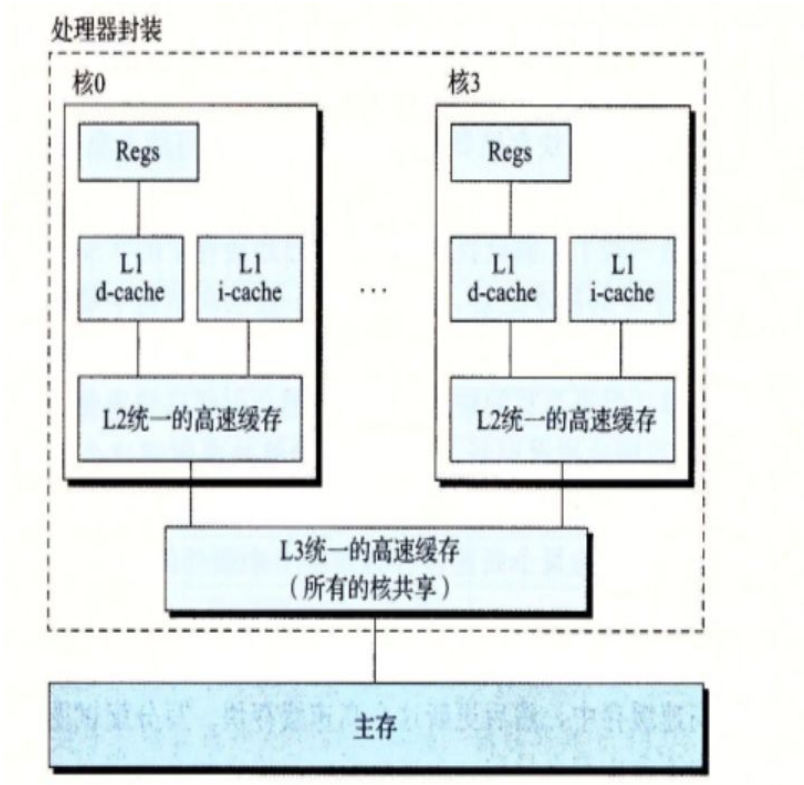
matrix.c 与 lab3方法类似，**新增了通过文件给矩阵赋值的API**
data文件夹是储存矩阵数据的地方,以及生成随机数据的脚本
multiply.c是矩阵加速乘法的实现。
test_correct.c是探究算法的正确性，以16*16矩阵为例，已测试完算法正确性。
test.c则是测量不同长度矩阵运算时间。

思路

参考了二年级学的计算机组成原理的基础知识。考虑了**cahce命中率,main memory,pipeline模型，多线程以及simd等指令架构**，从这些方面尝试提高运算效率。

矩阵运算速度慢原因

这里涉及到计算机的储存模型



矩阵规模小时，数据都存在cache里，距离cpu和alu较近，运算速度快，当矩阵规模大时，数据存在main memory里面，计算过程中会出现cache miss从而需要将数据从main memory传到cpu,时间较长。

调研方法

矩阵加速方法一般分为两种，一种为算法加速，一种为缓存加速。本次探索强调于优化缓存加速。

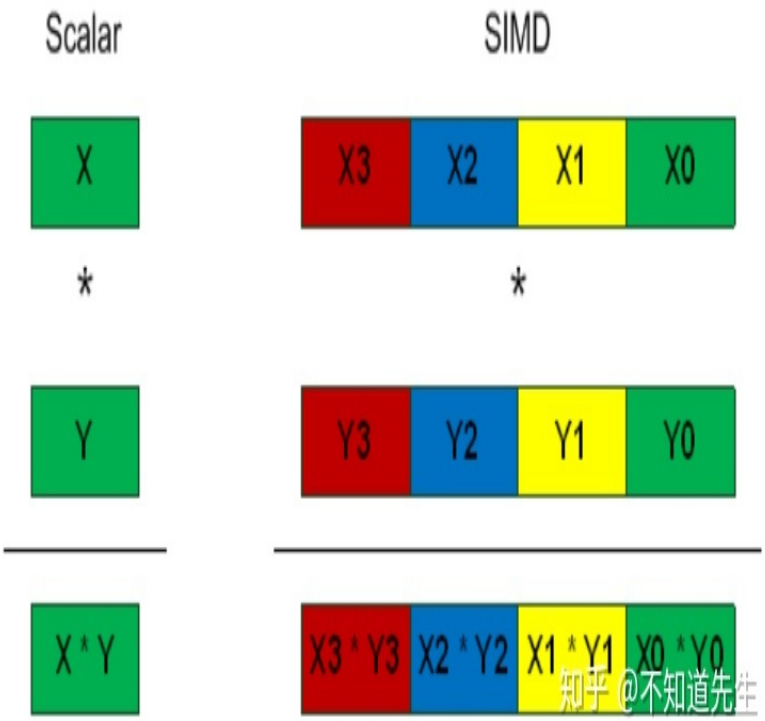
探索的思路

将矩阵转置再相乘

当矩阵相乘时，如果是暴力算法，会一个矩阵的每行和另一个矩阵的每列遍历相乘，**而遍历矩阵的列的时候，若矩阵规模较大会造成cahce miss的情况（元素与它下一列的元素地址间隔太远），从而增加时间。转置后等价于对两个矩阵都是行的遍历是cache友好的。**

通过simd进行继续优化

simd通过cpu的特定指令架构，能用一个指令完成多个对数据操作，从而达到减少计算次数的目的。通过一次处理矩阵一行的多个数据，达到提高计算效率的目的。原理示意图如下：



模拟cache加blocking加simd

真实的心路历程是，上述的simd完成后便不知道如何提升效率，**此时想起了大二学习的计组知识,cache的设计出发点是空间性与时间性(我大概率会访问邻近的数据，以及我大概率会访问刚用过的数据及其周围数据)。将矩阵不断的分块分成4*4的小矩阵，可以通过simd加速运算，同时也符合空间性，同时我在堆上设计了一个4行(矩阵的column长度)的人工cache，从而达到运算过程中直接去人工cache去数据，缓存命中率大幅度提升(相对于之前快了挺多，相对于openblas挺差的)。**

omp加simd加cache加blocking

本来以为omp加入后会提速，但事实是omp加simd并不会提示，**openmp的原理是并行运算，并行运算需要保证数据无相关性否会读写冲突。而我本次lab中使用了simd有相关性内存进行大量的读写。目前我探索的方法里并不能保证openmp的有效性，无速度提升和上面blocking方法时间类似,单独使用omp确实可以提升速度**

矩阵快速翻转

原理如下https://www.cnblogs.com/esing/p/4471543.html**将矩阵分成4*4小块,每次翻转一小块数据，从而提高cache 命中率，也可以通过simd优化，减少矩阵翻转时间，提高矩阵的计算效率。**

代码

矩阵翻转加速

矩阵乘法在上次lab文档中。

```
Matrix* transposedMatrix(Matrix* src,Matrix* dst){
    if (src==NULL)
    {
        return NULL;
    }
    dst=createMatrix(dst,src->column,src->row);//转置行列互换
    for(size_t i=0;i<dst->row;i++){
        for(size_t j=0;j<dst->column;j++){
            dst->matrix_data[i*dst->column+j]=src->matrix_data[j*src->column+i];
        }
    }
    return dst;
}
```

翻转矩阵与simd

```
Matrix* multiplyTransposeSimd(Matrix* res,Matrix* matrix1,Matrix* matrix2){
   if(matrix1==NULL||matrix2==NULL){
      return NULL;
   }
   __m128 vector1;
   __m128 vector2;
   __m128 res_temp;
   float* matrix1_vector=(float*)aligned_alloc(16,sizeof(float)*matrix1-
>column);//32字节确保对齐,同时储存即将放入simd中vector元素
   float* matrix2_vector=(float*)aligned_alloc(16,sizeof(float)*matrix1-
>column);//32字节确保对齐
   Matrix* matrix2_transposed=transposedMatrix(matrix2,matrix2_transposed);//先翻
转
   res=createMatrix(res,matrix1->row,matrix2->column);//创建矩阵
   for(size_t i=0;i<res->row;i++){
      memcpy(matrix1_vector,matrix1->matrix_data+i*matrix1-
>column,sizeof(float)*matrix1->column);
      for(size_t j=0;j<res->column;j++){
         memcpy(matrix2_vector,matrix2_transposed->matrix_data+j*matrix1-
>column,sizeof(float)*matrix1->column);
```

```
            for(size_t k=0;k<matrix1->column;k+=4){
                vector1=_mm_load_ps(matrix1_vector+k);
                vector2=_mm_load_ps(matrix2_vector+k);
                res_temp=_mm_dp_ps(vector1,vector2,0xf1);
                res->matrix_data[i*res->column+j]+=res_temp[0];
            }
        }
    }
    deleteMatrix(matrix2_transposed);
    free(matrix1_vector);
    free(matrix2_vector);
    return res;
}
```

翻转矩阵加simd加分块加人工cache

```
Matrix* multiplyTransposeBlockingSimd(Matrix* res,Matrix* matrix1,Matrix* matrix2)
{
    if (matrix1==NULL||matrix2==NULL)
    {
        return NULL;
    }
    //这里不仅使用了寄存器变量，还使用了blocking技术，因为simd点乘技术一次运行4个浮点数，所以分成4*4小块。
    Matrix* matrix2_transposed=transposedMatrix(matrix2,matrix2_transposed);
    res=createMatrix(res,matrix1->row,matrix2->column);
    float* row_cache1[4];//用于储存8*8分块matrix1矩阵的8行数据,提高缓存命中率.
    float* row_cache2[4];//用于储存matrix2的8列，即matrix2_transposed的8行数据，提高缓存命中率。
    float* temp=(float*)aligned_alloc(16,sizeof(float)*4);
    float* res_store=(float*)aligned_alloc(16,sizeof(float)*4);
    __m128 matrix1_row[4];//matrix1 row的simd vector。
    __m128 matrix2_row[4];//matrix2的simd vector
    __m128 res_row;//储存点乘结果.
    __m128 res_value_row[4];//储存未进行计算前res 第(i,j)个4*4小矩阵每行的值，计算过程中不断更新。
    for(int i=0;i<4;i++){ //开4个float[column] array 用于储存每行，方便cahce hit,直接在这里取值就可以了
            row_cache1[i]=(float*)aligned_alloc(16,sizeof(float)*matrix1->column);
            row_cache2[i]=(float*)aligned_alloc(16,sizeof(float)*matrix1->column);
    }
    float* temp_pointer1;//用于指向matrix1的行地址
    float* temp_pointer2;//用于指向matrix2的列地址，即matrix2_transposed的行地址。
    float* res_pointer;//用于指向res行的首地址。
    // #pragma omp parallel for num_threads(threadcount)
    for(size_t i=0;i<res->row;i+=4){ //因为一次取4*4小矩阵计算,i=i+4,j=j+4。(i,j)是第几个4块小矩阵的index tuple,需要第i行所有小矩阵和第j行所有小矩阵相乘。
        temp_pointer1=matrix1->matrix_data+i*res->column;
        for(int h=0;h<4;h++){
            memcpy(row_cache1[h],temp_pointer1+h*res->column,sizeof(float)*res->column);//缓存i~i+3行数据
```

```
        }
        for(size_t j=0;j<res->column;j+=4){
           //开始进行数据缓存，将j~j+3行数据放进temp2
            temp_pointer2=matrix2_transposed->matrix_data+j*res->column;
            res_pointer=res->matrix_data+i*res->column;
            for(int t=0;t<4;t++){
                 memcpy(row_cache2[t],temp_pointer2+t*res->column,sizeof(float)*res-
>column);
            }
            //缓存结束
            for(int k=0;k<4;k++){//将res_value load一下。
             memcpy(temp,res_pointer+j+k*res->column,sizeof(float)*4);
             res_value_row[k]=_mm_load_ps(temp); //将(i,j)的4*4矩阵的4行都存入vector
            }
            int time_index[4]; //0xf1,0xf2,0xf4,0xf8是simd中 vector相乘的参数，文档中
可以了解。
            time_index[0]=0xf1;
            time_index[1]=0xf2;
            time_index[2]=0xf4;
            time_index[3]=0xf8;
            for(size_t k=0;k<res->column;k=k+4){
               for(int row=0;row<4;row++){
                 matrix1_row[row]=_mm_load_ps(row_cache1[row]+k);
                 matrix2_row[row]=_mm_load_ps(row_cache2[row]+k);
               }
               //开始进行向量乘法,同时根据
               for(int l=0;l<4;l++){
                 for(int m=0;m<4;m++){

res_row=_mm_dp_ps(matrix1_row[l],matrix2_row[m],time_index[m]);
                      res_value_row[l]=_mm_add_ps(res_value_row[l],res_row);//更新新
的值。
                 }
               }
               for(int l=0;l<4;l++){
                 _mm_store_ps(res_store,res_value_row[l]);
                 memcpy(res_pointer+j+l*res->column,res_store,sizeof(float)*4);
               }
            }
        }
     }
   deleteMatrix(matrix2_transposed);
   for(int i=0;i<4;i++){
      free(row_cache1[i]);
      free(row_cache2[i]);
   }
   return res;
}
```

矩阵快速翻转代码

```
void tranBlock4(const __m128 s1,const __m128 s2, const __m128 s3, const __m128
s4,float* d1,float* d2,float* d3,float*d4){//s1,s2,s3,s4是src里面存的数
据,d1,d2,d3,d4是存放数据位置。
   //  printf("a0 %f,%f,%f,%f\n",s1[0],s1[1],s1[2],s1[3]);
   //  printf("b0 %f,%f,%f,%f\n",s2[0],s2[1],s2[2],s2[3]);
   //  printf("c0 %f,%f,%f,%f\n",s3[0],s3[1],s3[2],s3[3]);
   //  printf("d0 %f,%f,%f,%f\n",s4[0],s4[1],s4[2],s4[3]);
     __m128 t1,t2,t3,t4,t5,t6,t7,t8;
     __m128 test1,test2,test3,test4;
    t1=_mm_permute_ps(s1,0b11011000);//将第一行数据进行交换得到a0,a2,a1,a3
    t2=_mm_permute_ps(s2,0b11011000);//将第二行数据进行交换得到b0,b2,b1,b3
    t3=_mm_permute_ps(s3,0b01110010);//将第三行数据进行交换得到c2,c0,c3,c1
    t4=_mm_permute_ps(s4,0b01110010);//将第四行数据进行交换得到d2,d0,d3,d1
    t5=_mm_blend_ps(t1,t3,0b1010);//合并原序列第一行和重排序列第三行得到a0,c0,a1,c1
    t6=_mm_blend_ps(t2,t4,0b1010);//b0,d0,b1,d1
    t7=_mm_blend_ps(t3,t1,0b1010);//c2,a2,c3,a3
    t8=_mm_blend_ps(t4,s2,0b1100);//d2,b2,d3,b3
    t7=_mm_permute_ps(t7,0b01001110); //a2,c2,a3,c3
    t8=_mm_permute_ps(t8,0b01001110); //b2,d2,b3,d3
    _mm_store_ps(d1,_mm_unpacklo_ps(t5,t6));//生成转置子块，并写入对应位置
a0,b0,c0,d0
    _mm_store_ps(d2,_mm_unpackhi_ps(t5,t6));//a1,b1,c1,d1
    _mm_store_ps(d3,_mm_unpacklo_ps(t7,t8));//a2,b2,c2,d2
    _mm_store_ps(d4,_mm_unpackhi_ps(t7,t8));//a3,b3,c3,d3
}
void tranBlock8(float** src_array,float** dst_array){
    __m128 t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,t14,t15,t16;
    t1=_mm_load_ps(src_array[0]);
    t2=_mm_load_ps(src_array[1]);
    t3=_mm_load_ps(src_array[2]);
    t5=_mm_load_ps(src_array[0]+4);
    t6=_mm_load_ps(src_array[1]+4);
    t7=_mm_load_ps(src_array[2]+4);
    t4=_mm_load_ps(src_array[3]);
    t8=_mm_load_ps(src_array[3]+4);
    t9=_mm_load_ps(src_array[4]);
    t13=_mm_load_ps(src_array[4]+4);
    t10=_mm_load_ps(src_array[5]);
    t14=_mm_load_ps(src_array[5]+4);
    t11=_mm_load_ps(src_array[6]);
    t15=_mm_load_ps(src_array[6]+4);
    t12=_mm_load_ps(src_array[7]);
    t16=_mm_load_ps(src_array[7]+4);
    tranBlock4(t1,t2,t3,t4,dst_array[0],dst_array[1],dst_array[2],dst_array[3]);

tranBlock4(t9,t10,t11,t12,dst_array[0]+4,dst_array[1]+4,dst_array[2]+4,dst_array[3
]+4);
    tranBlock4(t5,t6,t7,t8,dst_array[4],dst_array[5],dst_array[6],dst_array[7]);

tranBlock4(t13,t14,t15,t16,dst_array[4]+4,dst_array[5]+4,dst_array[6]+4,dst_array[
7]+4);
}
void fastTrans(float* src,float* dst,size_t row,size_t col){//这里是将矩阵分成8*8个
```

```
小块转置
   float* src_array[8];
   float* dst_array[8];
   row=16;
   col=16;
   for(size_t i=0;i<row;i=i+8){ //第src_array第几行,应当更行dst_array列数
      for(size_t j=0;j<col;j=j+8){ //src_array第几个小块,应当更新dst_array行数。
         for(size_t l=0;l<8;l++){
            dst_array[l]=dst+j*row+i+l*row;
            src_array[l]=src+i*col+l*col+j;
         }
         tranBlock8(src_array,dst_array);
      }
   }
}
Matrix* fastTransposedMatrix(Matrix* src,Matrix* dst){
   if(src==NULL){
      return NULL;
   }
   dst=createMatrix(dst,src->column,src->row);
   fastTrans(src->matrix_data,dst->matrix_data,src->row,src->column);
   return dst;
}
```

结果对比图

本机资源不支持运算64k*64k矩阵计算，电脑会卡顿以及崩溃。

size	my time	openblas
16*16	0.000003s	0.000002s
128*128	0.000752s	0.000601s
1k*1k	0.415978s	0.040902s
8k*8k	205.821533s	13.786215s

结果截图

思考与总结

通过本次project,我学到了simd的用法(最主要是读文档的能力),同时也明白了**内存对齐的重要性，血泪教训**，也更理解了之前大二专业课学到的计算机体系架构知识。感觉自己想到的人工cache缓存数据提高命中率很开心hh,看到速度提升的那一刻感觉一切都是值得的，也更了解了openmp并行运算的底层原理，以及日后如何优化代码。