矩阵乘法的优化

环境

Ubuntu22.04 插电状态

笔记本关闭显示,不进入休眠,注销用户,在其他电脑上利用ssh连接

考虑到的性能影响:

vscode-server的ssh远程编程进程可能会对性能有影响

参考

- https://blog.csdn.net/wwxy1995/article/details/114762108
- https://renzibei.com/2021/06/30/optimize-gemm/
- https://zhenhuaw.me/blog/2019/gemm-optimization.html
- https://developer.aliyun.com/article/9265
- https://zhuanlan.zhihu.com/p/94649418
- https://blog.csdn.net/weixin_42826139/article/details/86358449
- https://zhuanlan.zhihu.com/p/362854057
- https://lzzmm.github.io/2021/09/10/GEMM/
- http://yuenshome.space/timeline/2018-12/optimize-cpu-gemm/
- https://zhuanlan.zhihu.com/p/146250334
- https://blog.csdn.net/oqqENvY12/article/details/88593322
- https://blog.csdn.net/realxie/article/details/7260072

总结

这部分花的时间是最久的,其中经过了代码更改、群里更新统一代码再次学习、学习指令集等过程

网上提供的现成资料并不是很丰富,部分文章难度感觉有点高

虽然时间花的多,但是并没有什么很好的收益..痛苦

反思一下,深陷在指令集,但没有突破

第一次:扩大单次计算量

注意到默认矩阵是 `499*499` ,如果单纯的增加单次计算的话很明显要进行一些处理,但是线代还在预习阶段,这部分处理有困难

考虑到单纯验证方法,我把矩阵改为了 `496*496`

for循环嵌套更改为

```
for(col=1;col<N;col+=4)
  for(row=1;row<N;row+=4)
    for(int k=1;k<N;k++)
```

性能计算结果

```
初始矩阵乘法运行耗时: 503.509539 ms
优化后矩阵乘法运行耗时: 386.209961 ms
加速比为: 1.303720
CHECK:
-RIGHT-
```

综合几次程序测试,**加速比大约稳定在1.30-1.33左右**,偶有1.4以上的加速比,但比例不高

似乎只要同时计算的更多,加速比会越高,但是测试了几组数据,似乎在4*4这样的分块是比例最高的

代码(仅修改部分):

```
  int col = 1;
  int row = 1;

  for(col=1;col<N;col+=4)
  {
    for(row=1;row<N;row+=4)
    {
      for(int k=1;k<N;k++)
      {
        c[col][row] += a[col][k]*b[k][row];
        c[col][row+1] += a[col][k]*b[k][row+1];
        c[col][row+2] += a[col][k]*b[k][row+2];
        c[col][row+3] += a[col][k]*b[k][row+3];

        c[col+1][row] += a[col+1][k]*b[k][row];
        c[col+1][row+1] += a[col+1][k]*b[k][row+1];
        c[col+1][row+2] += a[col+1][k]*b[k][row+2];
        c[col+1][row+3] += a[col+1][k]*b[k][row+3];

        c[col+2][row] += a[col+2][k]*b[k][row];
        c[col+2][row+1] += a[col+2][k]*b[k][row+1];
        c[col+2][row+2] += a[col+2][k]*b[k][row+2];
        c[col+2][row+3] += a[col+2][k]*b[k][row+3];

        c[col+3][row] += a[col+3][k]*b[k][row];
        c[col+3][row+1] += a[col+3][k]*b[k][row+1];
        c[col+3][row+2] += a[col+3][k]*b[k][row+2];
        c[col+3][row+3] += a[col+3][k]*b[k][row+3];
      }
    }
  }
```

第二次：调换循环顺序

我们把顺序由 `ijk` 更改为 `ikj`

这时候，不考虑同时计算，我们直接祭出最基础的代码

```
double s;

for(int i=1;i<N;i+=1)
{
  for(int k=1;k<N;k++)
  {
    s=a[i][k];
    for(int j=1;j<N;j++)
    {
      c[i][j]+=s*b[k][j];
    }
  }
}
```

结果

```
初始矩阵乘法运行耗时: 517.483207 ms
优化后矩阵乘法运行耗时: 324.142196 ms
加速比为: 1.596470
CHECK:
-RIGHT-
```

单纯调换了一下位置，速度直接来到了1.6x，再取十次结果的平均值，计算出平均值约为：

```
(1.596470+1.645169+1.556864+1.649172+1.828069+1.614608+1.604318+1.671952+1.671397+1.61
5165) /10 = 1.6453184
```

到这里可以到 `1.64` 左右的加速比

第三次：结合前面

直接贴代码：

```
  double s[4];

  for(int i=1;i<N;i+=4)
  {
    for(int k=1;k<N;k++)
    {
      s[0]=a[i][k];
```

```
        s[1]=a[i+1][k];
        s[2]=a[i+2][k];
        s[3]=a[i+3][k];
        for(int j=1;j<N;j+=4)
        {
          c[i][j]+=s[0]*b[k][j];
          c[i][j+1]+=s[0]*b[k][j+1];
          c[i][j+2]+=s[0]*b[k][j+2];
          c[i][j+3]+=s[0]*b[k][j+3];

          c[i+1][j]+=s[1]*b[k][j];
          c[i+1][j+1]+=s[1]*b[k][j+1];
          c[i+1][j+2]+=s[1]*b[k][j+2];
          c[i+1][j+3]+=s[1]*b[k][j+3];

          c[i+2][j]+=s[2]*b[k][j];
          c[i+2][j+1]+=s[2]*b[k][j+1];
          c[i+2][j+2]+=s[2]*b[k][j+2];
          c[i+2][j+3]+=s[2]*b[k][j+3];

          c[i+3][j]+=s[3]*b[k][j];
          c[i+3][j+1]+=s[3]*b[k][j+1];
          c[i+3][j+2]+=s[3]*b[k][j+2];
          c[i+3][j+3]+=s[3]*b[k][j+3];
        }
      }
    }
```

经过十次平均计算,基本可以达到 `1.86` 左右的加速比,最小值出现了 `1.794797` ,最大值则是 `1.919372`

单论最大值的话,基本上是1+0.6+0.3的即视感,也就是说这二者的关系是并列的

但是实际上仍然有小部分亏损

第四次:多线程并行

我们考虑一下最基础的代码

```
for(int i=1;i<N;i++)
    for(int j=1;j<N;j++)
      for(int k=1;k<N;k++)
        c_0[i][j] += a[i][k]*b[k][j];
```

分析第一个for,可以发现第一个for中各个i之间的处理没有联系

也就是说,我们可以创建i个线程,同时处理

头文件添加 `#include <threads.h>`

创建线程函数

```
void multi_thread_solve(int *p)
{
  int core = start_prefix++;

  for(int j=1;j<N;j++)
      for(int k=1;k<N;k++)
        c[core][j] += a[core][k]*b[k][j];
}
```

执行过程中加入线程

```
  thrd_t thid[N+10];
  for(int i=1;i<N;i++)
  {
    thrd_create(&thid[i], (thrd_start_t)multi_thread_solve,NULL);
  }

  for(int i=1;i<N;i++)
  {
    thrd_join(thid[i],NULL);
  }
```

由于使用了多线程并行,因此我们可以得到良好的加速比

在本机实测加速比平均为 `3.501106`

但是!多线程的处理上会出现问题,容易导致计算出错,也就是说可靠性不高

解决方案通常是**加锁,但是加锁会影响速度**,也就是说达不到3.5这么高了

第n+不知道多少次:利用AVX指令集加速

利用AVX指令集加速

这里有一些头文件的前置

```
#include <mmintrin.h>   //mmx, 4个64位寄存器
#include <xmmintrin.h>  //sse, 8个128位寄存器
#include <emmintrin.h>  //sse2, 8个128位寄存器
#include <pmmintrin.h>  //sse3, 8个128位寄存器
#include <smmintrin.h>  //sse4.1, 8个128位寄存器
#include <nmmintrin.h>  //sse4.2, 8个128位寄存器
#include <immintrin.h>  // avx, 16个256位寄存器
```

这部分花了非常长的时间,但是效果并没有那么的好,我感觉原因有如下:

- 平转,刚刚开始学线代,一些计算方法上不熟悉

- 指令集加速是一个相对陌生的地方
- 有点偏底层,理解难度上更大

翻了一晚上,翻到了一篇用指令集实现矩阵乘

理论上这道题应该是可以攻克的,但是把..这时候不知道如何从float改成double了,嗯..果然还是没有学到精髓

由于不懂得如何修改,我在float的条件下进行了测试:

利用avx指令集加速,我可以做到 `三倍以上` 的加速比,最好成绩是 `3.216054`

以时间上来举例,最初算法基本上是徘徊在 `482ms` 左右的,而使用avx指令集可以做到 `157ms` 的平均成绩

但是还没有变成自己的东西,嗯..

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <x86intrin.h>

const int M = 1000;
//串行矩阵乘
void matrix(float **a, float **b, float **c){
  int i, j, k;
  for(i = 0; i < M; i++){
    for(k = 0; k < M; k++){
      for(j = 0; j < M; j = j+1){
        c[i][j] += a[i][k] * b[k][j];
        //c[i][j+1] += a[i][k] * b[k][j+1];
        //c[i][j+2] += a[i][k] * b[k][j+2];
        //c[i][j+3] += a[i][k] * b[k][j+3];
      }
    }
  }
}
//两层循环展开
void matrix_loop_two(float **a, float **b, float **c){
  int i, j, k;
  for(i = 0; i < M; i++){
    for(k = 0; k < M; k++){
      for(j = 0; j < M; j = j+2){
        c[i][j] += a[i][k] * b[k][j];
        c[i][j+1] += a[i][k] * b[k][j+1];
        //c[i][j+2] += a[i][k] * b[k][j+2];
        //c[i][j+3] += a[i][k] * b[k][j+3];
      }
    }
  }
```

```
}
//数组赋值
void value(float **a){
  int i, j, t = 1.0;
  for(i = 0; i < M; i++){
    for(j = 0; j < M; j++){
      a[i][j] = t;
    }
    t++;
  }
}
//打印数组
void print(float **a){
  int i, j;
  for(i = 0; i < M; i++){
    for(j = 0; j < M; j++){
      printf("%.2f  ", a[i][j]);
    }
    printf("\n");
  }
  printf("************************************************************\n");
}
//重置数组元素
void reset(float **a){
  int i, j;
  for(i = 0; i < M; i++){
    for(j = 0; j < M; j++){
      a[i][j] = 0.0;
    }
  }
}
//数组转置存储
void rotate(float **b, float **copy_b){
  int i, j;
  for(i = 0; i < M; i++){
    for(j = 0; j < M; j++){
      copy_b[j][i] = b[i][j];
    }
  }
}

//avx指令向量乘
void avx_matrix(float **a, float **b, float **c){
  int i, j, k;
  float sum = 0.0;
  float assist = 0.0;
  //加载a的数组的寄存器，行row加载，连续存储
  __m256 r0, r1, r2, r3, r4, r5, r6, r7;
  //加载b数组的寄存器，col加载，转置后列变行，连续存储
  __m256 c0, c1, c2, c3, c4, c5, c6, c7;
```

```
__m256 avx_mul0, avx_mul1, avx_mul2, avx_mul3,
  avx_mul4, avx_mul5, avx_mul6, avx_mul7;
__m256 avx_sum0 = _mm256_setzero_ps();
__m256 avx_sum1 = _mm256_setzero_ps();
__m256 avx_sum2 = _mm256_setzero_ps();
__m256 avx_sum3 = _mm256_setzero_ps();
__m256 avx_sum4 = _mm256_setzero_ps();
__m256 avx_sum5 = _mm256_setzero_ps();
__m256 avx_sum6 = _mm256_setzero_ps();
__m256 avx_sum7 = _mm256_setzero_ps();
__m256 avx_zero = _mm256_setzero_ps();
//方阵中每行或每列取64个数据，放到8个寄存器中
int copy_M = M - M % 64;
//剩余不足64个的数据
int reserve = M % 64;
for(i = 0; i < M; i++){
  for(j = 0; j < M; j++){
    for(k = 0; k < copy_M; k = k + 64){
      r0 = _mm256_loadu_ps(&a[i][k]);
      r1 = _mm256_loadu_ps(&a[i][k+8]);
      r2 = _mm256_loadu_ps(&a[i][k+16]);
      r3 = _mm256_loadu_ps(&a[i][k+24]);
      r4 = _mm256_loadu_ps(&a[i][k+32]);
      r5 = _mm256_loadu_ps(&a[i][k+40]);
      r6 = _mm256_loadu_ps(&a[i][k+48]);
      r7 = _mm256_loadu_ps(&a[i][k+56]);

      c0 = _mm256_loadu_ps(&b[i][k]);
      c1 = _mm256_loadu_ps(&b[i][k+8]);
      c2 = _mm256_loadu_ps(&b[i][k+16]);
      c3 = _mm256_loadu_ps(&b[i][k+24]);
      c4 = _mm256_loadu_ps(&b[i][k+32]);
      c5 = _mm256_loadu_ps(&b[i][k+40]);
      c6 = _mm256_loadu_ps(&b[i][k+48]);
      c7 = _mm256_loadu_ps(&b[i][k+56]);

      avx_mul0 = _mm256_mul_ps(r0, c0);
      avx_mul1 = _mm256_mul_ps(r1, c1);
      avx_mul2 = _mm256_mul_ps(r2, c2);
      avx_mul3 = _mm256_mul_ps(r3, c3);
      avx_mul4 = _mm256_mul_ps(r4, c4);
      avx_mul5 = _mm256_mul_ps(r5, c5);
      avx_mul6 = _mm256_mul_ps(r6, c6);
      avx_mul7 = _mm256_mul_ps(r7, c7);

      avx_sum0 = _mm256_add_ps(avx_sum0, avx_mul0);
      avx_sum1 = _mm256_add_ps(avx_sum1, avx_mul1);
      avx_sum2 = _mm256_add_ps(avx_sum2, avx_mul2);
      avx_sum3 = _mm256_add_ps(avx_sum3, avx_mul3);
      avx_sum4 = _mm256_add_ps(avx_sum4, avx_mul4);
```

```
        avx_sum5 = _mm256_add_ps(avx_sum5, avx_mul5);
        avx_sum6 = _mm256_add_ps(avx_sum6, avx_mul6);
        avx_sum7 = _mm256_add_ps(avx_sum7, avx_mul7);
      }
      //每次向量乘并求和
      avx_sum0 = _mm256_add_ps(avx_sum0, avx_sum1);
      avx_sum2 = _mm256_add_ps(avx_sum2, avx_sum3);
      avx_sum4 = _mm256_add_ps(avx_sum4, avx_sum5);
      avx_sum6 = _mm256_add_ps(avx_sum6, avx_sum7);
      avx_sum0 = _mm256_add_ps(avx_sum0, avx_sum2);
      avx_sum2 = _mm256_add_ps(avx_sum4, avx_sum6);
      avx_sum0 = _mm256_add_ps(avx_sum0, avx_sum2);
      //每次求出的c[i][j]
      avx_sum0 = _mm256_hadd_ps(avx_sum0, avx_zero);
      avx_sum0 = _mm256_hadd_ps(avx_sum0, avx_zero);

      assist =  avx_sum0[0] + avx_sum0[4];
      c[i][j] += assist;
      //寄存器归0
      avx_sum0 = _mm256_setzero_ps();
      avx_sum1 = _mm256_setzero_ps();
      avx_sum2 = _mm256_setzero_ps();
      avx_sum3 = _mm256_setzero_ps();
      avx_sum4 = _mm256_setzero_ps();
      avx_sum5 = _mm256_setzero_ps();
      avx_sum6 = _mm256_setzero_ps();
      avx_sum7 = _mm256_setzero_ps();
    }

  }
  //处理第二个矩阵的列向量reserve
  assist = 0.0;
  for(i = 0; i < M; i++){
    for(j = 0; j < M; j = j+1){
      for(k = 0; k < reserve; k++){
        assist += a[i][copy_M+k] * b[j][copy_M+k];
      }
      c[i][j] += assist;
      assist = 0.0;
    }
  }
}
int main(){

  clock_t start, end;
  int i;
  //copy_b  b的转置存储
  float **a, **b, **c, **copy_b;
  a = (float**)malloc(sizeof(float*) * M);
  b = (float**)malloc(sizeof(float*) * M);
```

```
c = (float**)malloc(sizeof(float*) * M);
copy_b = (float**)malloc(sizeof(float*) * M);

a[0] = (float*)malloc(sizeof(float) * M * M);
b[0] = (float*)malloc(sizeof(float) * M * M);
c[0] = (float*)malloc(sizeof(float) * M * M);
copy_b[0] = (float*)malloc(sizeof(float) * M * M);
//保证申请的空间连续
for(i = 1; i < M; i++){
  a[i] = a[i-1] + M;
  b[i] = b[i-1] + M;
  c[i] = c[i-1] + M;
  copy_b[i] = copy_b[i-1] + M;
}

value(a);
value(b);
reset(c);

//normal_mul
start = clock();
matrix(a, b, c);
end = clock();
printf("normal :  c[20][9] = %f", c[20][9]);
double time = (double)(end - start) / CLOCKS_PER_SEC;
printf("   waste time = %f\n", time);

//normal_loop_four mul
reset(c);
start = clock();
matrix_loop_two(a, b, c);
end = clock();
printf("matrix_loop_two :  c[20][9] = %f", c[20][9]);
time = (double)(end - start) / CLOCKS_PER_SEC;
printf("   waste time = %f\n", time);

//avx_loop_four mul
reset(c);
start = clock();
rotate(b, copy_b);
avx_matrix(a, copy_b, c);
end = clock();
printf("avx_loop_eight :  c[20][9] = %f", c[20][9]);
time = (double)(end - start) / CLOCKS_PER_SEC;
printf("  waste time = %f\n", time);

free(a[0]);
free(b[0]);
free(c[0]);
free(a);
```

```
    free(b);
    free(c);
}
```

总结

这部分花的时间较多,从最初的1.0优化到多线程平均的3.5,但是可靠性不高

如果使用上avx指令集加速,可以达到3.2这样子,但是可靠性高

不过avx仍然需要学啊()