OpenMP 初试

> code at: /lesson05/src_mat/src, data at: /lesson05/src_mat/data

Demo

```
// openmp_demo.cpp
#include <cstdio>
#include <omp.h>

int main(int argc, char **argv) {
  // 在这里使用这种指令开启 OpenMP
  #pragma omp parallel
  printf("Hello, OpenMP!\n");
  return 0;
}
```

编译添加 OpenMP 相关库选项：`g++ -fopenmp openmp_demo.cpp -o openmp_demo && ./openmp_demo`

在 8 核心 CPU 的计算机上输出：

```
➜  src_mat git:(master) ./openmp_demo
Hello, OpenMP!
Hello, OpenMP!
Hello, OpenMP!
Hello, OpenMP!
Hello, OpenMP!
Hello, OpenMP!
Hello, OpenMP!
Hello, OpenMP!
➜  src_mat git:(master)
```

在开启了 OpenMP 的那一行，一行语句被拆分到 8 个核心分别执行，所以输出了 8 次。

使用 OpenMP 优化矩阵乘法

代码基本和 lesson04 一样，添加了 OpenMP 操作对比，优化了数据展示方式。

添加的 OpenMP **优化**如下：

1. 普通计算的时候用 OpenMP 优化

   ```
   // @prog: 用 OpenMP 执行矩阵乘法
   // @args: a * b -> c
   // @rets: c
   Mat* mat_mul_openmp_native(Mat* a, Mat* b, Mat* c) {
     // 检查是否合法
     if ((!a || !b || !c) || (a->w != b->h)) return NULL;
     // 计时，超时的话就直接返回
     int k = a->w;
     #pragma omp parallel
     for (int x = 0; x < c->w; x++) {
       for (int y = 0; y < c->h; y++) {
         double sum = 0;
         for (int i = 0; i < k; i++) sum += a->data[x][i] * b->data[i][y];
         c->data[x][y] = sum;
       }
     }
     return c;
   }
   ```

   这里把 OpenMP 放在了最外面一层循环，因为按照实验放在最外层已经是相对更快的做法了，暂时不知道原因。

   而且如果 OpenMP 放在最内侧的 i 循环，甚至会计算错误。我不知道这是什么原因。

2. 用 OpenMP 优化 lesson04 中的任务调度程序

```
Mat* mat_mul_openmp(Mat* a, Mat* b, Mat* c, int unrolling) {
  // 检查是否合法
  if (a->w != b->h) {
    return NULL;
  }
  // 首先对 b 进行一个置的转
  Mat* t = mat_transpose(b);
  // 初始化任务列表和线程池
  mat_task_tail = a->h * b->w;
  mat_task_list = malloc(sizeof(int*) * mat_task_tail);
  assert(mat_task_list);
  mat_task_data = malloc(sizeof(int) * mat_task_tail * 2);
  assert(mat_task_data);
  // 初始化任务
  for (int x = 0; x < c->h; x++) {
    for (int y = 0; y < c->w; y++) {
      mat_task_list[x * c->w + y] = &mat_task_data[(x * c->w + y) * 2];
      mat_task_list[x * c->w + y][0] = x;
      mat_task_list[x * c->w + y][1] = y;
    }
  }
  // 填满线程参数
  mat_mul_thread_t* thread_data =
      malloc(sizeof(mat_mul_thread_t) * mat_task_tail);
  for (int i = 0; i < mat_task_tail; i++) {
    thread_data[i].a = a;
    thread_data[i].b = t;
    thread_data[i].c = c;
    thread_data[i].id = i;
    thread_data[i].single = 1;
    thread_data[i].single_index = i;
    thread_data[i].unrolling = unrolling;
  }
  #pragma omp parallel
  for (int t = 0; t < mat_task_tail; t++) {
    mat_mul_cell(thread_data + t);
  }
  return c;
}
```

这段代码中首先单个任务的参数生成好，方便调用的时候独立传参。

lesson04 中是首先填满 CPU 核心大小的线程池，然后每个线程独立领取任务，这里是每个线程只执行一次，不需要领取任务，线程执行之后退出再由 OpenMP 调用执行下一个线程。

实验结果

运行环境标在图片标题上。（CPU 所标频率为默频，运行时频率更高；SIMD 指 AVX256 指令集优化。）

小矩阵情况：$N \in [2^1, \lfloor 2^{7.5} \rfloor]$，重复 50 次取平均值

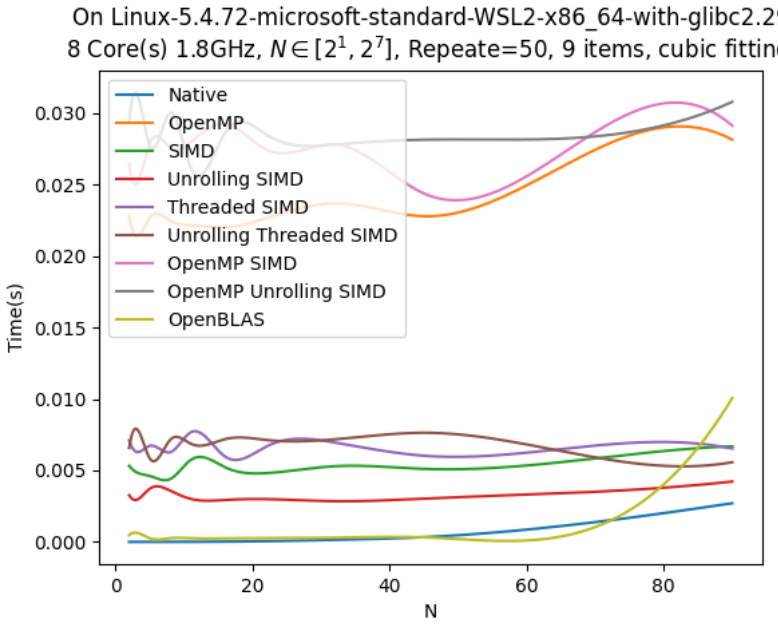

图1，openmp_wsl_s1_m7_r50_cubic.png

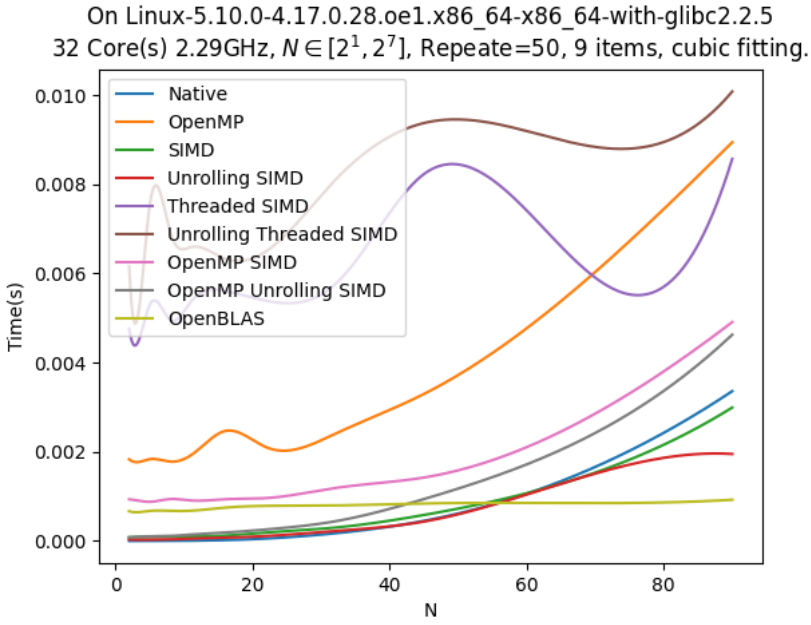

图2，openmp_server_s1_m7_r50_cubic.png

1. 在小矩阵的情况下，因为比较复杂的算法都需要一些内存分配等规划时间和空间，所以这些算法往往不如直接进行矩阵计算快，也不如直接计算要稳定。
2. SIMD（单指令单数据）表现相对稳定，在这种情况下相比直接计算慢的不多。
3. 多线程计算分配内存的花销一直存在，而且在小矩阵的情况下很明显。
4. OpenMP 的优化在小矩阵的情况下并不好，比很多方法差很多，说明对 OpenMP 库来说，计算拆得太零碎对整体计算优化不大。
5. OpenBLAS 在计算小矩阵的时候速度和直接计算差不多，说明内部对小矩阵计算也有所优化。

大矩阵情况：$N \in [2^7, \lfloor 2^{10.5} \rfloor]$，重复 2 次取平均值

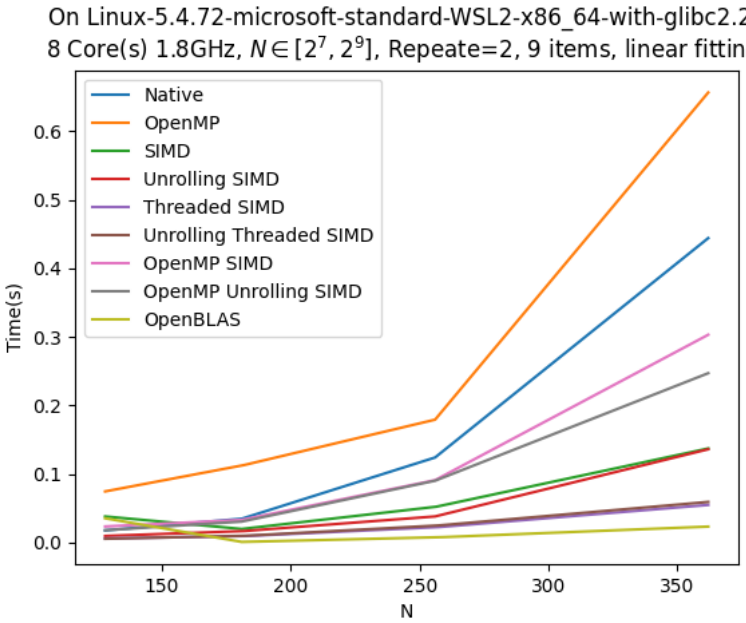

图3，openmp_wsl_s7_m9_r2_linear.png

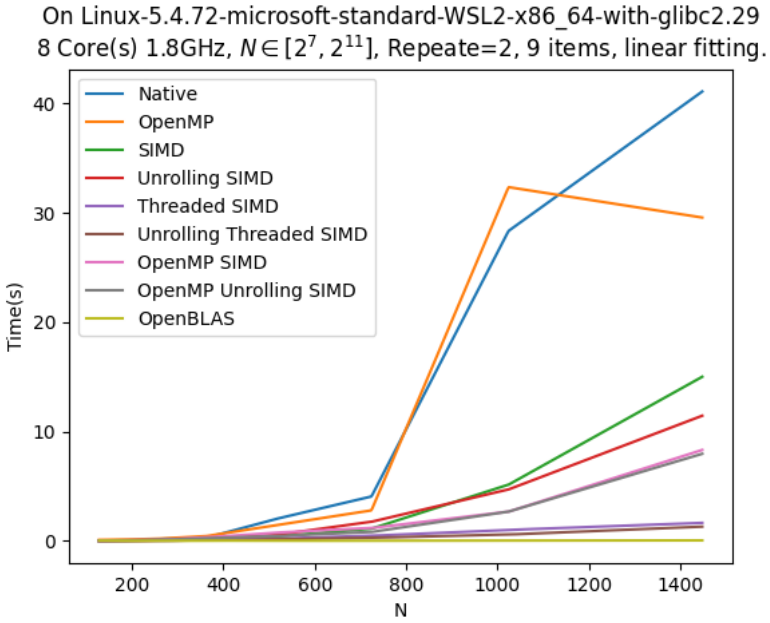

图4，openmp_wsl_s7_m11_r2_linear.png

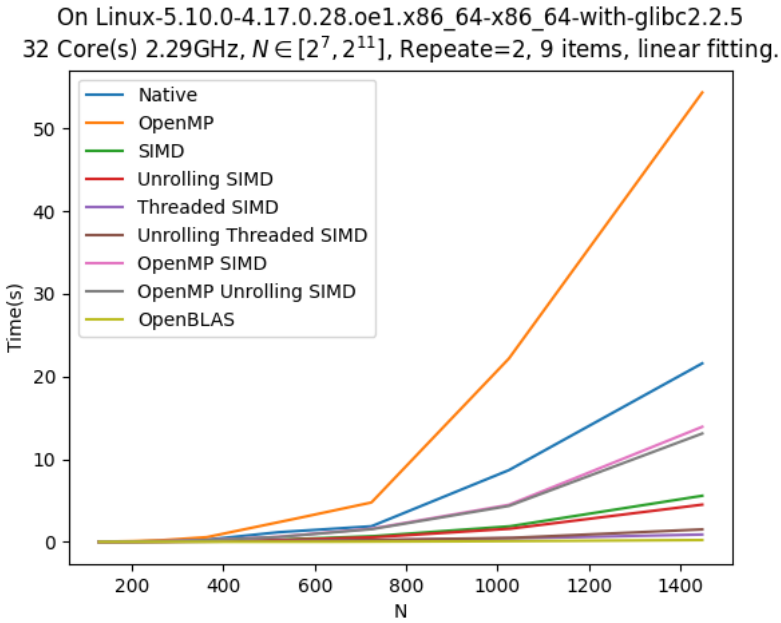

图5，openmp_server_s7_m11_r2_linear.png

1. OpenMP 在此实验中表现并不算好。

   首先是橙色线的 "OpenMP"，即在最外层加了自动展开的 Native 算法，在$E \in [2^7, 2^9)$的时候表现仍然不如 Native。明明并行块已经足够大，而且并行块之间并没有数据冲突，但是速度就是很慢。在 N = 1448 且运行环境为 8 核的时候， OpenMP 优化终于快过 Native，但是并没快过太多，而且和手动管理多线程的优化方法还有很大差别。

   然后是“OpenMP SIMD”和“SIMD”的对比（粉色和绿色线），即用 OpenMP 优化过调度的SIMD优化以及单线程的 SIMD 优化。在图 3 中，OpenMP SIMD 仍然慢于 SIMD，在图 4 和图 5，OpenMP SIMD 才快于 单线程的 SIMD。

   再然后是“OpenMP Unrolling SIMD”和“Unrolling SIMD”（灰色和红色线），即用 OpenMP 优化过调度的手动循环展开的 SIMD，以及单线程循环展开的 SIMD。这两条线情况和粉色、绿色线结果相同。

   由上面三组比较可以看出，在 N 特别大的时候，OpenMP 才比优化前有优势。

   那么是不是 OpenMP 没有运行起来呢？我们可以使用 `pstree` 命令查看进程的线程树。

```
➜  build_mkfile git:(master) ✗ ./mat_mul_test 1024 &
[3] 8453
使用: K = 1024
Running with 8 cores.
# OpenMP 启动前, "Native"
➜  build_mkfile git:(master) ✗ pstree -p 8453
mat_mul_test(8453)─┬─{mat_mul_test}(8455)
                   ├─{mat_mul_test}(8456)
                   ├─{mat_mul_test}(8457)
                   ├─{mat_mul_test}(8461)
                   ├─{mat_mul_test}(8462)
                   ├─{mat_mul_test}(8463)
                   └─{mat_mul_test}(8464)
Native: 计算用时: 32.866s
# OpenMP 启动后, "OpenMP"
[1]     OpenMP 计算: 开始计时
pstree -p 8453
mat_mul_test(8453)─┬─{mat_mul_test}(8455)
                   ├─{mat_mul_test}(8456)
                   ├─{mat_mul_test}(8457)
                   ├─{mat_mul_test}(8461)
                   ├─{mat_mul_test}(8462)
                   ├─{mat_mul_test}(8463)
                   ├─{mat_mul_test}(8464)
                   ├─{mat_mul_test}(8825)
                   ├─{mat_mul_test}(8826)
                   ├─{mat_mul_test}(8827)
                   ├─{mat_mul_test}(8828)
                   ├─{mat_mul_test}(8829)
                   ├─{mat_mul_test}(8830)
                   └─{mat_mul_test}(8831)
OpenMP: 计算用时: 28.793s
```

可见 OpenMP 确实有作用，开始计算后多出了 6 个线程，N = 1024 时最终计算也比 Native 快些。

再看看 N = 256 的时候。

```
#!/bin/bash
./build_mkfile/mat_mul_test 256 &
for i in $(seq 1 10)
do
sleep 0.1 && pstree -p `ps -ef | grep "mat_mul_test" | grep -v grep | awk '{print $2}'`
done
```

```
➜  src_mat git:(master) ✗ sh test_openmp.sh
使用: K = 256
Running with 8 cores.
[0]     Native 计算: 开始计时
mat_mul_test(14287)─┬─{mat_mul_test}(14289)
                    ├─{mat_mul_test}(14290)
                    ├─{mat_mul_test}(14291)
                    ├─{mat_mul_test}(14292)
                    ├─{mat_mul_test}(14293)
                    ├─{mat_mul_test}(14295)
                    └─{mat_mul_test}(14296)
        Native: 计算用时: 0.217s
[1]     OpenMP 计算: 开始计时
mat_mul_test(14287)─┬─{mat_mul_test}(14289)
                    ├─{mat_mul_test}(14290)
                    ├─{mat_mul_test}(14291)
                    ├─{mat_mul_test}(14292)
                    ├─{mat_mul_test}(14293)
                    ├─{mat_mul_test}(14295)
                    ├─{mat_mul_test}(14296)
                    ├─{mat_mul_test}(14304)
                    ├─{mat_mul_test}(14305)
                    ├─{mat_mul_test}(14306)
                    ├─{mat_mul_test}(14307)
                    ├─{mat_mul_test}(14308)
                    ├─{mat_mul_test}(14309)
                    └─{mat_mul_test}(14310)
        OpenMP: 计算用时: 0.281s
[2]     SIMD 计算: 开始计时
mat_mul_test(14287)─┬─{mat_mul_test}(14289)
                    ├─{mat_mul_test}(14290)
                    ├─{mat_mul_test}(14291)
                    ├─{mat_mul_test}(14292)
                    ├─{mat_mul_test}(14293)
                    ├─{mat_mul_test}(14295)
                    ├─{mat_mul_test}(14296)
                    ├─{mat_mul_test}(14304)
                    ├─{mat_mul_test}(14305)
                    ├─{mat_mul_test}(14306)
                    ├─{mat_mul_test}(14307)
                    ├─{mat_mul_test}(14308)
                    ├─{mat_mul_test}(14309)
                    └─{mat_mul_test}(14310)
        SIMD: 计算用时: 0.087s
[3]     Unrolling SIMD 计算: 开始计时
mat_mul_test(14287)─┬─{mat_mul_test}(14289)
                    ├─{mat_mul_test}(14290)
                    ├─{mat_mul_test}(14291)
                    ├─{mat_mul_test}(14292)
                    ├─{mat_mul_test}(14293)
                    ├─{mat_mul_test}(14295)
                    ├─{mat_mul_test}(14296)
                    ├─{mat_mul_test}(14304)
                    ├─{mat_mul_test}(14305)
                    ├─{mat_mul_test}(14306)
                    ├─{mat_mul_test}(14307)
                    ├─{mat_mul_test}(14308)
                    ├─{mat_mul_test}(14309)
                    ├─{mat_mul_test}(14310)
                    └─{mat_mul_test}(14326)
        Unrolling SIMD: 计算用时: 0.073s
[4]     Threaded SIMD 计算: 开始计时
        Threaded SIMD: 计算用时: 0.033s
[5]     Unrolling Threaded SIMD 计算: 开始计时
        Unrolling Threaded SIMD: 计算用时: 0.031s
[6]     OpenMP SIMD 计算: 开始计时
mat_mul_test(14287)─┬─{mat_mul_test}(14289)
                    ├─{mat_mul_test}(14290)
                    ├─{mat_mul_test}(14291)
                    ├─{mat_mul_test}(14292)
                    ├─{mat_mul_test}(14293)
                    ├─{mat_mul_test}(14295)
```

```
                    ├─{mat_mul_test}(14296)
                    ├─{mat_mul_test}(14304)
                    ├─{mat_mul_test}(14305)
                    ├─{mat_mul_test}(14306)
                    ├─{mat_mul_test}(14307)
                    ├─{mat_mul_test}(14308)
                    ├─{mat_mul_test}(14309)
                    └─{mat_mul_test}(14310)
        OpenMP SIMD: 计算用时: 0.118s
[7]     OpenMP Unrolling SIMD 计算: 开始计时
mat_mul_test(14287)─┬─{mat_mul_test}(14289)
                    ├─{mat_mul_test}(14290)
                    ├─{mat_mul_test}(14291)
                    ├─{mat_mul_test}(14292)
                    ├─{mat_mul_test}(14293)
                    ├─{mat_mul_test}(14295)
                    ├─{mat_mul_test}(14296)
                    ├─{mat_mul_test}(14304)
                    ├─{mat_mul_test}(14305)
                    ├─{mat_mul_test}(14306)
                    ├─{mat_mul_test}(14307)
                    ├─{mat_mul_test}(14308)
                    ├─{mat_mul_test}(14309)
                    └─{mat_mul_test}(14310)
        OpenMP Unrolling SIMD: 计算用时: 0.134s
[8]     OpenBLAS 计算: 开始计时
        OpenBLAS: 计算用时: 0.031s
校验...
DONE.
```

OpenMP 确实启动了……就是在这时候 OpenMP “优化”后确实比 Native 更慢了……

2. 而 OpenMP 和手动的多线程相比呢？可以很清楚地看出，“Threaded SIMD”和“Unrolling Threaded SIMD”消耗时间和最稳定最快的 OpenBLAS 最接近，领先 OpenMP 优化的几条线很多。其中，手动循环展开在 N 很大的时候和未手动循环展开也有一定的领先。
3. OpenBLAS 在大矩阵的计算实验中十分稳定，计算的时候却只使用了一个核心，但是比 Unrolling Threaded SIMD 还快不少。

实验结论

1. 在实际环境测试中，OpenMP 性能并没有太大优化，甚至有出现计算错误的情况，很有可能 OpenMP 的使用方法错误了。
2. OpenMP 中方便快捷的 Directives 的使用可以极大的加快开发速度，具有方便易用、兼容性高、高性能的特点。
3. 和 pthread 对比， OpenMP 开发比自己手动管理线程数量和线程池要方便得多，而且兼容性更高，Windows 同样可以简单使用。