# 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。

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

```
int main(int argc, char **argv) {
  #pragma omp parallel
  {
    #pragma omp for
    for (int i = 0; i < 64; i++) printf("i = %d\n", i);
  }
  return 0;
}
```

使用 `#pragma omp for` 对 `for` 循环开启 OpenMP。 `g++ -o openmp_demo_for openmp_demo_for.cpp -fopenmp && ./openmp_demo_for` 输出为：

```
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 16
i = 17
i = 18
i = 19
i = 20
i = 21
i = 22
i = 23
i = 8
i = 9
i = 10
i = 11
i = 12
i = 13
i = 14
i = 15
i = 48
......
```

运行环境为 8 核心 CPU。所以可知，OpenMP 现在按照 8 个线程分组执行。

# 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

```
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
  {
#pragma omp for
    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;
}
```

如在最内层循环开启 OpenMP，需要使用 `#pragma omp parallel for reduction(+:sum)` 。

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
  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**

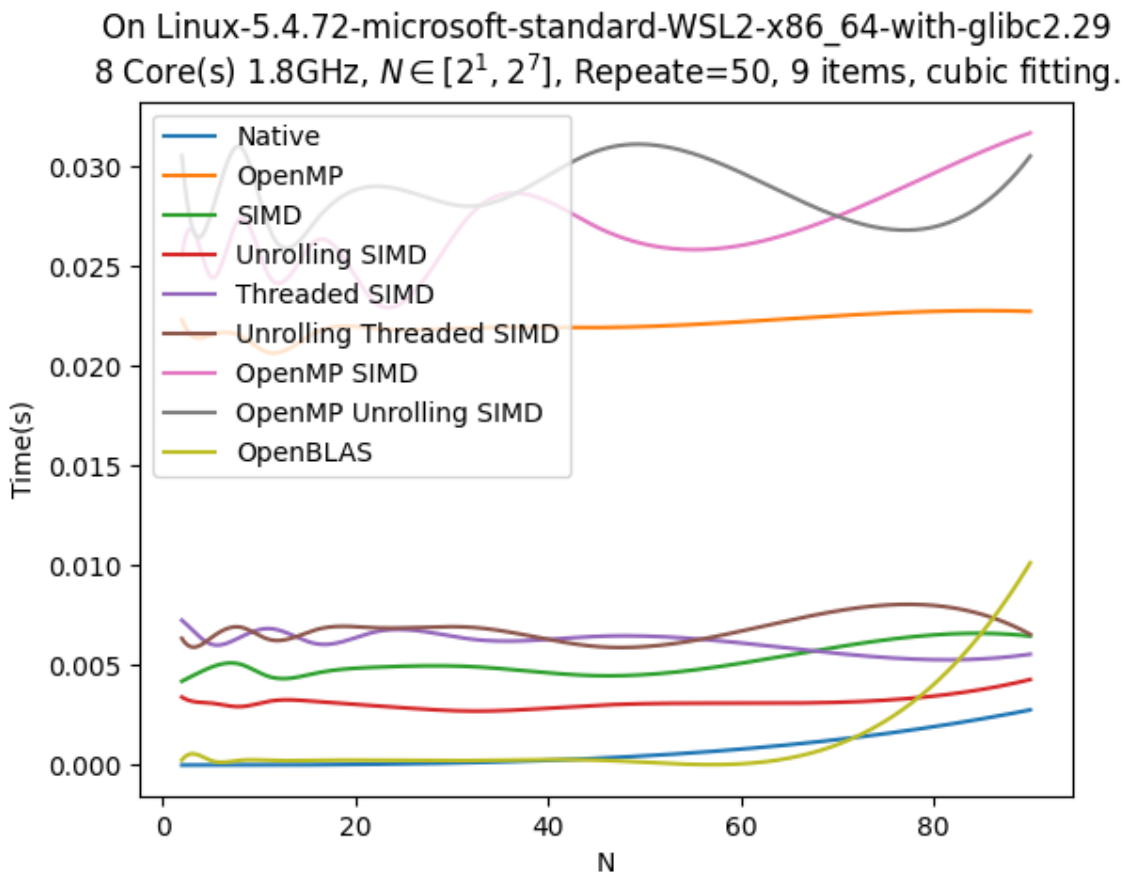


图1，openmp_update_s1_m7_r50_cubic.png

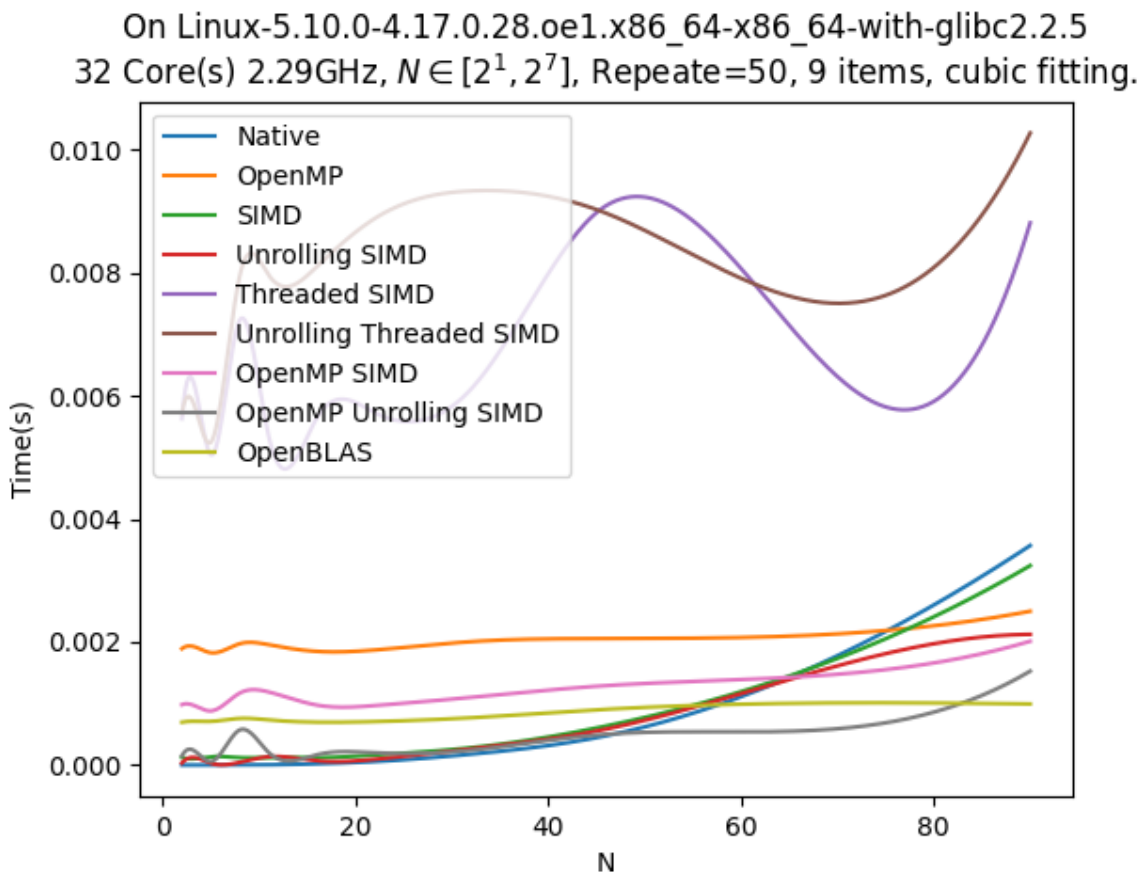


图2，openmp_update_server_s1_m7_r50_cubic.png

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。

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

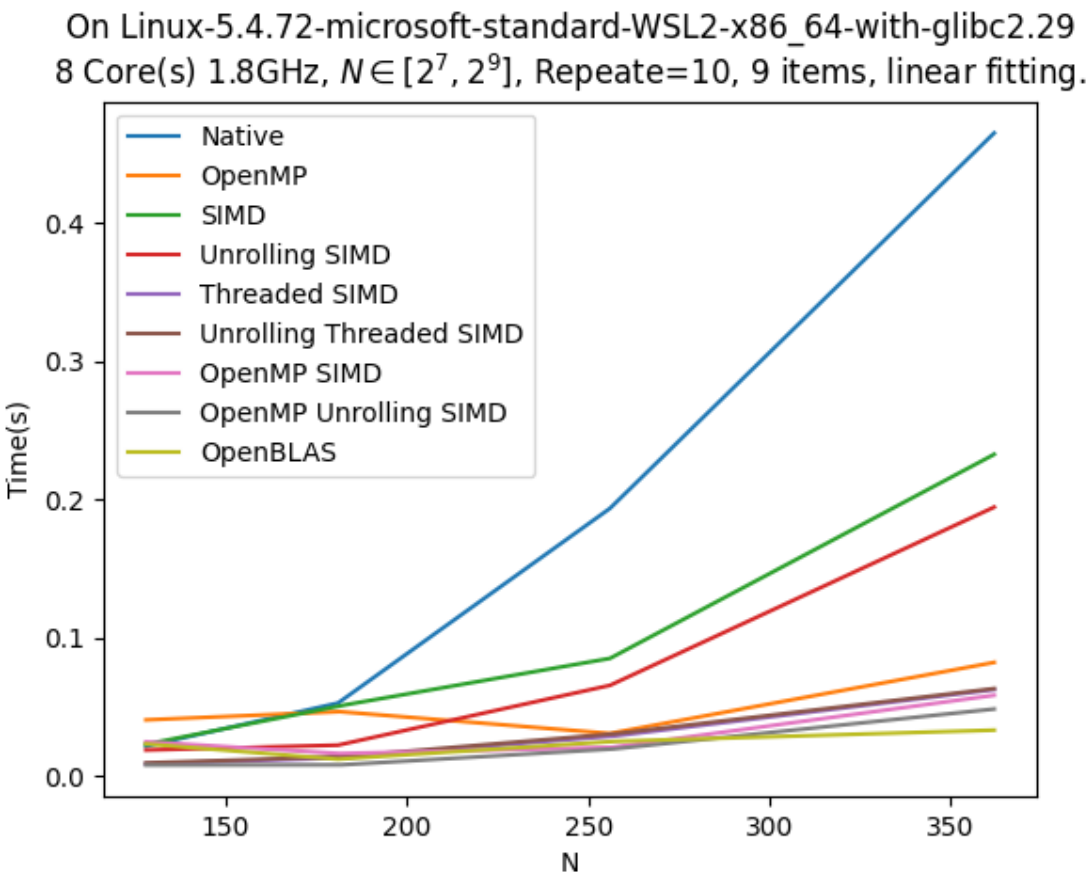


图3，openmp_update_s7_m9_r10_linear.png

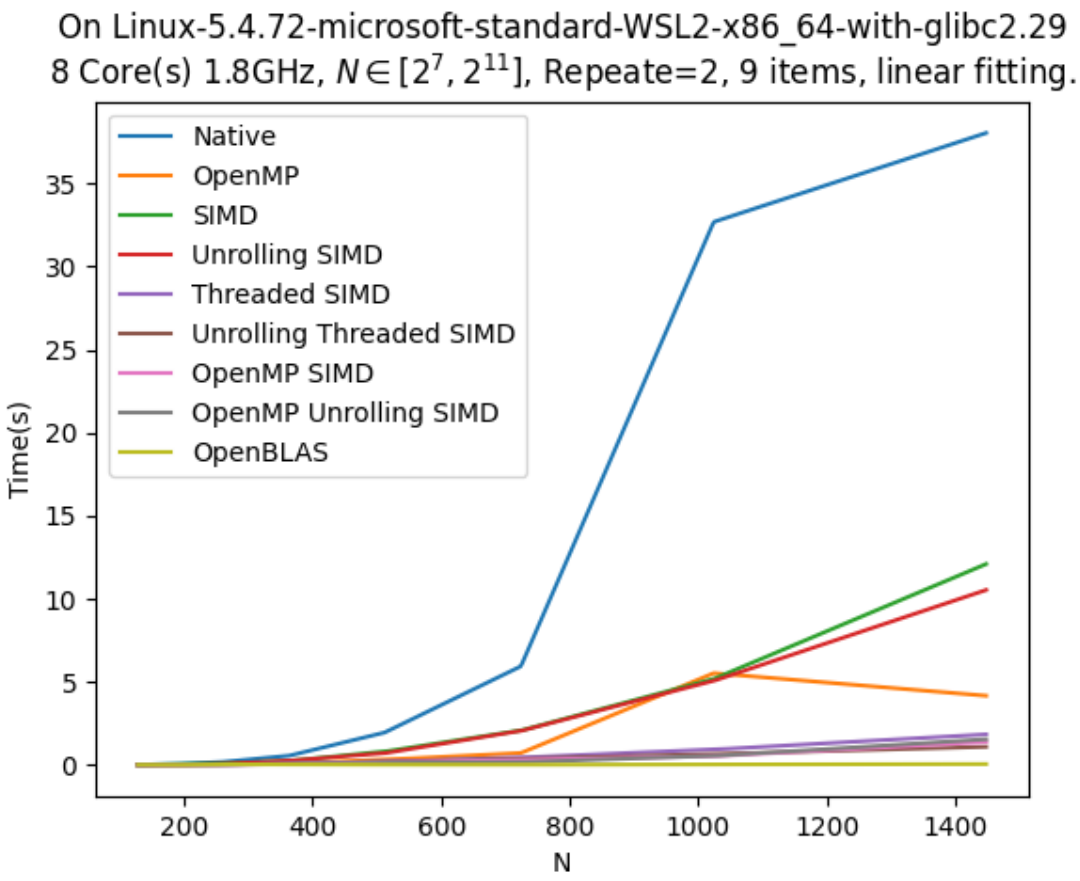


图4，openmp_update_s7_m11_r2_linear.png

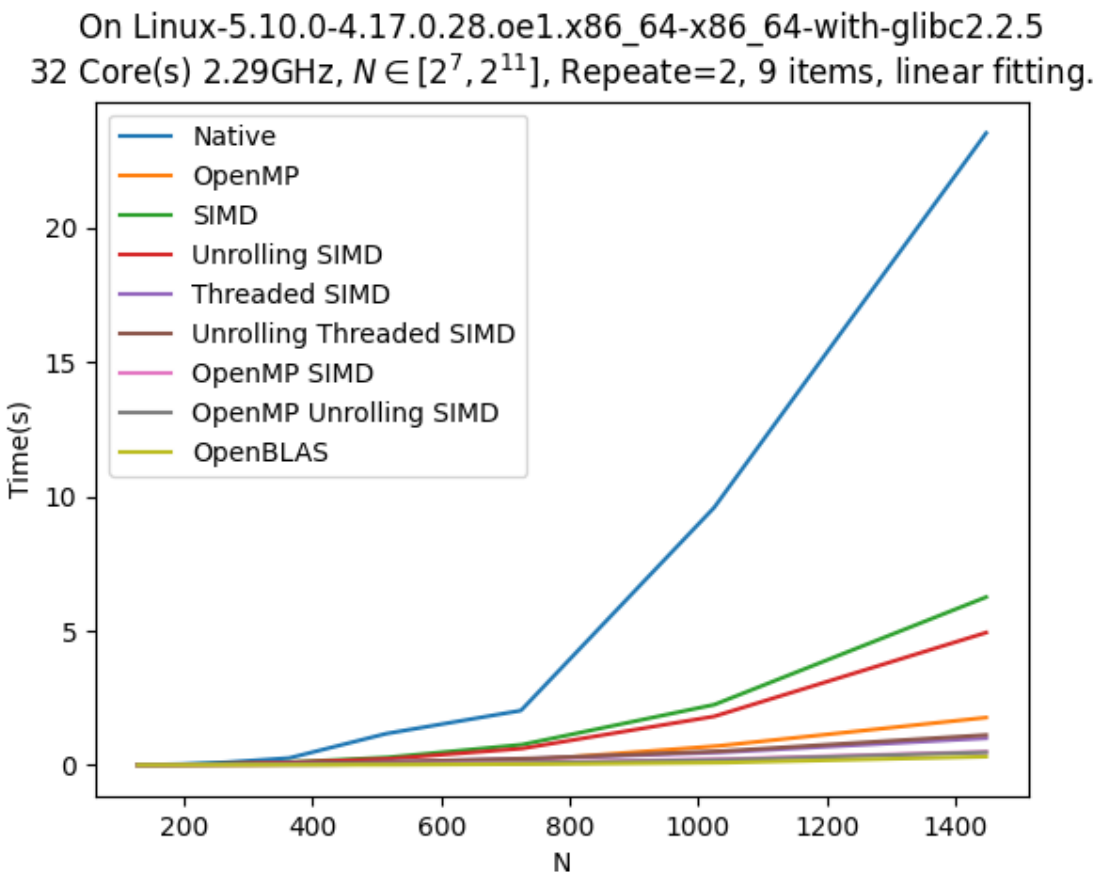


图5，openmp_server_s7_m11_r2_linear.png

1. OpenMP 在此实验中 N 相对较大的时候表现很不错，优化效果很明显。

   首先是橙色线的 "OpenMP"，即在最外层加了 OpenMP 的 Native 算法，效率约为 Native 的 8 倍。

   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。经过 OpenMP 优化后速度也增长到了原来 SIMD 的 8 倍左右。

   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，当 N 比较大，OpenMP 可以自动调度上所有的 CPU 核心进行计算，成倍提高计算速度。

2. 而 OpenMP 和手动的多线程相比呢？上述数据中，OpenMP 优化在 N 较大时效率始终高于手动分配调度的多线程算法，在 N 很小的时候（N < 核心数）可能没有完全应用到所有核心的性能，比手动的多线程要慢一些。

3. OpenBLAS 在大矩阵的计算实验中十分稳定，但是比我实现的几种组合方法还快不少。
4. 自己实现的几种方法中，“OpenMP Unrolling SIMD”效率和 OpenBLAS 接近，但是稳定性和速度还是比不上 OpenBLAS。

## 实验结论

1. 在实际环境测试中，OpenMP 性能优化明显。
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。