Linux 环境多线程编程

信息

Aa 姓名	# 学号	编辑日期
梁鑫嵘	200110619	@September 18, 2021

多线程初试

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int glob = 6;
void increse_num(int *arg);

int main() {
  int ret;
  pthread_t thrd1, thrd2, thrd3;
  int flag = 1;
  ret = pthread_create(&thrd1, NULL, (void *)increse_num, &flag);
  sleep(1);  //如果无sleep(1)结果会有什么不同？
  flag = 2;
  ret = pthread_create(&thrd2, NULL, (void *)increse_num, &flag);
  pthread_join(thrd1, NULL);
  pthread_join(thrd2, NULL);
  flag = 0;
  increse_num(&flag);  //如果放在pthread_join之前结果会怎样？
  return 0;
}
```

```
void increse_num(int *arg) {
  int flag = *arg;
  if (flag == 0) {
    glob = glob + 1;
    // sleep(10);  //如果都加了sleep，输出的结果有什么不同？
    // 没有什么不同
    printf("parent %d: glob=%d\n", flag, glob);
  } else if (flag == 1) {
    glob = glob + 2;
    // sleep(10);
    printf("child %d: glob=%d\n", flag, glob);
  } else {
    glob = glob + 3;
    printf("child %d: glob=%d\n", flag, glob);
  }
}
```

运行结果为：

```
➜  src git:(master) ✗ ./thread_demo_single_var
child 1: glob=8
child 2: glob=11
parent 0: glob=12
➜  src git:(master) ✗
```

此时，运行顺序为：

1. 创建 thread1
2. thread1 执行完毕，`glob: 6 => 8`
3. 创建 thread2
4. thread2 执行完毕，`glob: 8 => 11`
5. thread1 和 thread2 执行完毕，主线程继续向下执行
6. `glob: 11 => 12`

如果去掉创建 `thrd1` 之后的 `Sleep()` ，结果为：

```
➜  src git:(master) ✗ ./thread_demo_single_var_s1
child 2: glob=9
child 2: glob=12
parent 0: glob=13
➜  src git:(master) ✗
```

可以看到，两个线程都表示自己的 `flag` 是 `2` ，显然是不太“符合逻辑”的。那么，这是为什么呢？

在去掉了 `Sleep(1)` 之后，thread1 没有执行完就执行了 thread2，两个线程几乎同时开始执行；而这时候主线程已经让 `flag = 2` 了。两个线程并没有像我们想象中那样先运行 thread1 然后再执行 thread2，从而在一些变量上产生了不确定性：如果在主线程执行 `flag = 2` 之前两个线程都执行完了，那么两个线程都认为 `flag = 1` ；如果在主线程执行 `flag = 2` 之后两个线程才执行完，那么两个线程都认为 `flag = 2` 。

这样对资源（全局变量）“竞争”的场景并不存在于多进程当中，因为多进程产生的时候会完全复制整个程序的堆栈，使得变量实际上并不共享，多进程之间的通信依赖系统内核方法等方式实现，而多线程中则需要应用程序自己管理需要共同操作的变量。

多线程对资源的竞争还会表现在对 IO 资源 / 内存资源等只能由一个线程正常操作的资源上，而且如果管理不当后果更加严重。比如，同时对一个文件写入，或者同时向屏幕输出内容，很有可能会得到互相穿插的两个数据片段而造成数据丢失或者产生异常。

解决的方法有加锁、抛出/捕获异常等，之后再说吧。

利用多线程改进 HTTP Server

这里在 lesson03 的 `socket_server_demo.c` 基础上开始优化。

主要优化 / 改变的点：

1. 因为多线程相比于多进程占用资源更少、产生速度更快，我们可以在处理这个问题时开更多的线程。
2. 在访问共享的变量的时候使用线程锁，防止资源被重复访问。
3. 线程之间通信可以使用 `pthread` 的队列，防止某进程等待资源而占用 CPU。
4. 在程序退出之前可以用一个函数释放整体资源。
5. 可以取消延时，然后测试极限性能。

```
// socket_server_threads.c
#include <arpa/inet.h>
#include <errno.h>
#include <malloc.h>
#include <pthread.h>
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define PORT 8002  // 服务器监听端口
```

```
#define SERVER_THREAD_NUM_MAX 512  // 最大线程数
// #define SERVER_SLEEP
int thread_max = SERVER_THREAD_NUM_MAX;

// #define SERVER_DEBUG

#ifdef SERVER_DEBUG
#define pdebug(...) printf(__VA_ARGS__)
#else
#define pdebug(...)
#endif  // SERVER_DEBUG

// 添加一个互斥锁，防止对全局变量的同时读写操作
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int glob = 0;
// 线程池
pthread_t pool[SERVER_THREAD_NUM_MAX];
int pool_index[SERVER_THREAD_NUM_MAX];
pthread_cond_t pool_cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t pool_mutex = PTHREAD_MUTEX_INITIALIZER;
int pool_size = 0;
int pool_front = 0;
int pool_back = 0;

int exiting = 0;

void enqueue(int index) {
  pool_index[pool_back] = index;
  pool_back = (pool_back + 1) % SERVER_THREAD_NUM_MAX;
  pool_size++;
}

int dequeue() {
  int index = pool_index[pool_front];
  pool_front = (pool_front + 1) % SERVER_THREAD_NUM_MAX;
  pool_size--;
  return index;
}

int queue_full() { return pool_size == SERVER_THREAD_NUM_MAX; }

int queue_empty() { return pool_size == 0; }

int server_socket;

typedef struct {
  int client_socket;
  int index;
} ThreadData;

void sleep_ms(int ms) {
  struct timeval delay;
  delay.tv_sec = 0;
  delay.tv_usec = ms * 1000;  // 20 ms
  select(0, NULL, NULL, NULL, &delay);
}

void send_handle(ThreadData *thread_data) {
```

```
  if (!thread_data) {
    perror("Null on thread_data!\n");
  }
  int client_socket = thread_data->client_socket;
  int index = thread_data->index;
  printf("Start child thread: client = %d, index = %d\n", client_socket, index);
  // 释放 ThreadData 资源
  free(thread_data);
  char status[] = "HTTP/1.0 200 OK\r\n";
  char header[] =
      "Server: DWBServer\r\nContent-Type: text/html;charset=utf-8\r\n\r\n";
  const char body_template[] =
      "<html>"
      "<head>"
      "<title>C语言构建小型Web服务器</title>"
      "</head>"
      "<body>"
      "<h2>欢迎</h2>"
      "<p>Hello，World</p>"
      "<br />"
      "<code>glob = %d</code>"
      "</body>"
      "</html>";
  char body[1024];

#ifdef SERVER_SLEEP
  pdebug("enter sleeping...\n");
  sleep(10);  //让进程进入睡眠状态，单位是秒。
#endif
  // sleep_ms(1);
  // 处理信息
  // 进入锁区
  pthread_mutex_lock(&mutex);
  // 改变全局变量
  glob += 1;
  sprintf(body, body_template, glob);
  // 自动退出
  if (glob == thread_max) {
    kill(getpid(), SIGINT);
    printf("Killing myself...\n");
  }
  // 退出锁区
  pthread_mutex_unlock(&mutex);
  pdebug("finish sleeping...\n");

  write(client_socket, status, sizeof(status) - 1);
  write(client_socket, header, sizeof(header) - 1);
  write(client_socket, body, strlen(body) * sizeof(char));

  close(client_socket);
  pdebug("client %d closed.\n", client_socket);
  // 添加当前空缺位置到队列
  pthread_mutex_lock(&pool_mutex);
  while (queue_full()) {
    pthread_cond_wait(&pool_cond, &pool_mutex);
  }
  enqueue(index);
  pool[index] = 0;
```

```
  pthread_mutex_unlock(&pool_mutex);
  pthread_cond_signal(&pool_cond);
}

void on_exit_handler() {
  static int first_run = 1;
  if (first_run) {
    first_run = 0;
    if (server_socket) {
      // close(server_socket);
      // 强制释放资源
      if (shutdown(server_socket, 2)) {
        printf("shutdown socket error: %s (errno: %d)\n", strerror(errno),
               errno);
      }
      server_socket = 0;
    }
  } else {
    // 释放资源然后退出
    printf("on_exit_handler()...\n");
    for (int i = 0; i < pool_size; i++) {
      if (pool[i] == 0) continue;
      // 等待所有线程退出
      printf("Waiting pid %lu\n", pool[i]);
      pthread_join(pool[i], NULL);
    }
    system("killall wget");
  }
}

void signal_handler() {
  // 捕获 Ctrl+C
  exiting = 1;
  printf("Captured Ctrl+C SIGINT, exiting...\n");
  on_exit_handler();
  // exit(0);
}

int main(int argc, char **argv) {
  int port = PORT;
  if (argc >= 2) {
    port = atoi(argv[1]);
  }
  if (argc >= 3) {
    thread_max = atoi(argv[2]);
  }
  printf("Listen on port %d, thread_max = %d\n", port, thread_max);
  server_socket = socket(AF_INET, SOCK_STREAM, 0);  //初始化套接字
  struct sockaddr_in server_addr;

  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family = AF_INET;  // 服务地址和端口配置
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  server_addr.sin_port = htons(port);

  bind(server_socket, (struct sockaddr *)&server_addr,
       sizeof(server_addr));  //将本地地址绑定到所创建的套接字上
```

```
    listen(
        server_socket,
        SERVER_THREAD_NUM_MAX);  //开始监听是否有客户端连接，第二个参数是最大监听数

    char buf[1024];
    int client_socket;

    // 注册监听 SIGINT
    signal(SIGINT, signal_handler);
    // 注册退出时函数
    atexit(on_exit_handler);

    int ret;
    pthread_t th;

    // 自己启动测试脚本
    // sprintf(buf, "sh run.sh %d &", SERVER_THREAD_NUM_MAX * 3);
    sprintf(buf, "sh run.sh %d %d &", port, thread_max);
    printf("Starting run.sh: %s\n", buf);
    int r = system(buf);
    printf("r = %d\n", r);

    // 装满可用 index
    for (int i = 0; i < SERVER_THREAD_NUM_MAX; i++) enqueue(i);

    while (1) {
      if (exiting || !server_socket) break;
      printf("====== waiting for client's request =====\n");
      if ((client_socket = accept(server_socket, (struct sockaddr *)NULL,
                                  NULL)) == -1) {  //等待客户端（浏览器）连接
        if (exiting) {
          printf("exiting accept..\n");
          return 0;
        } else {
          printf("accept socket error: %s (errno: %d)\n", strerror(errno), errno);
        }
        continue;
      }
      printf("====== waiting for read request data =====\n");
      read(client_socket, buf, 1024);  //读取客户端内容，这里是HTTP的请求数据
      // printf("%s",buf);     // 打印读取的内容

      // Chiro: 在这里另开一个线程，用于处理用户请求
      printf("client_socket = %d\n", client_socket);
      // 取出线程池可用列表头部
      while (queue_empty()) {
        // 等待队列数据
        pthread_cond_wait(&pool_cond, &pool_mutex);
      }
      int index = dequeue();
      ThreadData *thread_data = NULL;
      thread_data = malloc(sizeof(ThreadData));
      thread_data->client_socket = client_socket;
      thread_data->index = index;
      ret =
          pthread_create(&th, NULL, (void *(*)(void *))send_handle, thread_data);
      if (ret != 0) {
        perror("Cannot start new thread!!\n");
```

```
      // return 1;
      break;
    }
  }
  return 0;
}
```

```
#!/bin/bash

# run.sh
# cd ../data/

pids=( )
start=$(date +%s.%N)
th_num=$2
port=$1

if [ -z "$th_num" ]; then
  th_num=1024
fi
if [ -z "$port" ]; then
  th_num=8001
fi

# for J in `seq 1000`; do

  echo Starting $th_num threads.
  for I in `seq $th_num`; do
    echo ----request $I: `wget --tries=1 -q -O - 127.0.0.1:$port` &
    pid=$!
    pids[$I]=$pid
  done
  echo Started $th_num threads.
  for I in `seq $th_num`; do
    wait ${pids[$I]}
  done
  end=$(date +%s.%N)
  delta=`echo | awk -v start=$start -v end=$end '{printf("%.3f", end-start)}'`

  echo Used $delta s

# done
```

编译：`gcc socket_server_threads.c -o socket_server_threads -lpthread -g`

执行：`./socket_server_threads <端口> <客户端线程数量>`

结果： `./socket_server_threads 8004 1024` 然后 `nload lo` ，网速峰值约 8MB/s，如果增大单次请求大小，吞吐量还能继续上升。

利用多线程改进矩阵乘法

实验内容

使用多线程方法改进简单的单线程矩阵计算方法，提高矩阵计算效率。

设计方案

矩阵乘法原理：设两个矩阵$A_{m\times k}$、$B_{k\times n}$相乘得$C_{m\times n}$，则所得矩阵$C_{i,j} = \Sigma_{p=0}^{k} A_{i,p} \times B_{p,j}$。

观察矩阵乘法得知，对于C中每单个元素，都是A的行和B的列的对应元素相乘的和，其读取A、B元素的过程是互不干扰的，写入结果矩阵C的过程也是不互相干扰的，故求$C_{i,j}$的过程是一个个独立的求解过程。

设计方案

相比于原来的计算方案，增加多线程优化，每次最多分配P（处理器个数）个线程，保证每个 CPU 核心都能够分配到计算任务。

我们可以将矩阵C设计成每个子线程都能独立访问到的形式，然后设计一个线程池和任务队列，开始的时候启动P个任务，然后每个任务完成后对应线程各自领取下一个任务，直到任务队列为空。

具体的话还是有许多需要考虑的问题：

1. 启动线程也是需要占用运行时间的，如何平衡这个消耗？

   我们可以启动处理器数量个线程，分别处理相对独立的事，可以更加充分地利用多核心处理器的性能。

   而如果线程处理任务结束，不关闭线程而只是重新分配线程的任务，可以避免线程的多次开启关闭，也能增加一定性能（大概）。

2. 处理单个任务（计算$C_{i,j}$）的时候，首先计算两个 `double` 小数的积，然后计算这些积的和，有没有更加快速的方案？

   注意到，一个 `double` 占用 64 字节的空间，但是处理器大都有一次处理更多位数的指令，称为 SIMD（Single Instruction Multiple Data， 单指令多数据流），对应 SSE、SSE2、AVX等指令集指令。我们可以使用这样的指令，在一个指令周期内执行多次运算。这里我打算使用 AVX 系列指令，细节如下：

   1. 编译时在编译选项中加入 `-mavx` 表示打开对 AVX 指令的支持。
   2. `#include <immintrin.h>` 引入相关操作。注意，这是 Linux 下 GCC 编译器的做法，其他编译器不一定相同。
   3. AVX 指令一次能计算 256 位宽的数据，所以一次能计算 4 对数据的运算，即向量长度为 4。我们将输入数据对齐成按照 256 位对齐，然后送到 AVX 对应寄存器中，称为“加载（load）”。

4. 调用指令集相关命令计算，结果存入一个结果寄存器中。
5. 计算完成后从结果寄存器中拿出数据，SIMD 加速计算完成。
6. 计算剩余部分。

3. 如何使用循环展开？

   在使用 AVX 指令集的基础上实现手动的四路循环展开。在代码中复制 4 份一样的算法，写在同一个循环内，让编译器自动展开成 4 路；然后每两路相加合并，合并成的两个结果再相加，得到 16 对 `double` 的积的和。

主要代码

同 lesson02，本次也是使用 Python 以不同参数调用程序以得到实验结果并绘制图像。

多线程矩阵相乘的几种不同实现

```
// @prog: 执行矩阵乘法，返回一个新的矩阵作为运算结果，或者直接在目的矩阵操作
// @args: a * b -> c
// @rets: c
Mat* mat_mul(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;
  struct timespec start, end;
  clock_gettime(CLOCK_REALTIME, &start);
  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;
    }
    if ((c->h >= 256 || c->w >= 256) && mat_native_time_limit > eps) {
      clock_gettime(CLOCK_REALTIME, &end);
      double delta = time_delta(start, end);
      if (delta > mat_native_time_limit) {
        mat_native_timeout = 1;
        return c;
      }
    }
  }
  return c;
}

// @prog:
// 多线程执行矩阵乘法，返回一个新的矩阵作为运算结果，或者直接在目的矩阵操作
// @args: a * b -> c
// @rets: c
int** mat_task_list = NULL;
int* mat_task_data = NULL;
```

```
int mat_task_tail = 0;
pthread_mutex_t mat_task_mutex = PTHREAD_MUTEX_INITIALIZER;

void mat_mul_cell(mat_mul_thread_t* thread_data) {
  Mat* a = thread_data->a;
  Mat* b = thread_data->b;
  Mat* c = thread_data->c;
  int id = thread_data->id;
  int unrolling = thread_data->unrolling;
  free(thread_data);
  int k = a->w;
  int index = 0;

  const size_t block_size = 4;
  size_t block_count = k / block_size;
  size_t remain_count = k % block_size;
  __m256d sum_sp, sum_bl;
  __m256d load_a, load_b;
  __m256d load_a_0, load_b_0, load_a_1, load_b_1, load_a_2, load_b_2, load_a_3,
      load_b_3;
  __m256d sum_sp_0, sum_sp_1, sum_sp_2, sum_sp_3, sum_sp_0_0, sum_sp_0_1,
      sum_sp_0_0_0;
  while (1) {
    pthread_mutex_lock(&mat_task_mutex);
    if (mat_task_tail == 0) {
      pthread_mutex_unlock(&mat_task_mutex);
      break;
    }
    index = --mat_task_tail;
    pthread_mutex_unlock(&mat_task_mutex);
    double sum = 0;
    sum_bl = _mm256_setzero_pd();

    int x = mat_task_list[index][0], y = mat_task_list[index][1];
    double *p_a = a->content + (x * ((a->w / 4 + 1) * 4)),
           *p_b = b->content + (y * ((b->w / 4 + 1) * 4));
    // 使用 AVX 指令集
    if (!unrolling) {
      for (int i = 0; i < block_count; i++) {
        load_a = _mm256_load_pd(p_a);
        load_b = _mm256_load_pd(p_b);
        p_a += block_size;
        p_b += block_size;
        sum_sp = _mm256_mul_pd(load_a, load_b);
        sum_bl = _mm256_add_pd(sum_sp, sum_bl);
      }
    } else {
      int block_remain = block_count % 4;
      for (int i = 0; i < block_count / 4; i++) {
        load_a_0 = _mm256_load_pd(p_a + 0);
        load_b_0 = _mm256_load_pd(p_b + 0);
        load_a_1 = _mm256_load_pd(p_a + 4);
        load_b_1 = _mm256_load_pd(p_b + 4);
        load_a_2 = _mm256_load_pd(p_a + 8);
        load_b_2 = _mm256_load_pd(p_b + 8);
        load_a_3 = _mm256_load_pd(p_a + 12);
        load_b_3 = _mm256_load_pd(p_b + 12);
        p_a += block_size * 4;
```

```
        p_b += block_size * 4;
        sum_sp_0 = _mm256_mul_pd(load_a_0, load_b_0);
        sum_sp_1 = _mm256_mul_pd(load_a_1, load_b_1);
        sum_sp_2 = _mm256_mul_pd(load_a_2, load_b_2);
        sum_sp_3 = _mm256_mul_pd(load_a_3, load_b_3);
        sum_sp_0_0 = _mm256_add_pd(sum_sp_0, sum_sp_1);
        sum_sp_0_1 = _mm256_add_pd(sum_sp_2, sum_sp_3);
        sum_sp_0_0_0 = _mm256_add_pd(sum_sp_0_0, sum_sp_0_1);
        sum_bl = _mm256_add_pd(sum_sp_0_0_0, sum_bl);
      }
      for (int i = 0; i < block_remain; i++) {
        load_a = _mm256_load_pd(p_a);
        load_b = _mm256_load_pd(p_b);
        p_a += block_size;
        p_b += block_size;
        sum_sp = _mm256_mul_pd(load_a, load_b);
        sum_bl = _mm256_add_pd(sum_sp, sum_bl);
      }
    }

    if (block_count > 0) {
      double* p = (double*)(&sum_bl);
      sum = p[0] + p[1] + p[2] + p[3];
    } else {
      sum = 0;
    }
    for (int i = 0; i < remain_count; i++) {
      sum += a->data[x][i + block_size * block_count] *
             b->data[y][i + block_size * block_count];
    }
    c->data[x][y] = sum;
  }
}

double mat_native_time_limit = 0;
int mat_native_timeout = 0;

Mat* mat_mul_threaded(Mat* a, Mat* b, Mat* c, int processor_number,
                      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);
  // puts("B^T:");
  // mat_print(t);
  // 初始化任务列表和线程池
  mat_task_list = malloc(sizeof(int*) * (a->h * b->w));
  assert(mat_task_list);
  mat_task_data = malloc(sizeof(int) * (a->h * b->w) * 2);
  assert(mat_task_data);
  mat_task_tail = a->h * b->w;
  // printf("processor_number = %d\n", processor_number);
  pthread_t* pool = malloc(sizeof(pthread_t) * processor_number);
  assert(pool);
  // 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;
    }
  }
  // 填满线程池
  int ret = 0;
  for (int i = 0; i < processor_number; i++) {
    mat_mul_thread_t* thread_data = malloc(sizeof(mat_mul_thread_t));
    thread_data->a = a;
    thread_data->b = t;
    thread_data->c = c;
    thread_data->id = i;
    thread_data->unrolling = unrolling;
    ret = pthread_create(&pool[i], NULL, (void* (*)(void*))mat_mul_cell,
                         thread_data);
    if (ret) {
      printf("Cannot create thread!\n");
    }
  }
  // 等待所有线程执行完毕
  for (int i = 0; i < processor_number; i++) {
    pthread_join(pool[i], NULL);
  }
  // puts("C:");
  // mat_print(c);
  return c;
}
```

OpenBLAS 的调用代码

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, A->w, B->h, A->h, 1,
            A->content, A->w, B->content, B->h, 0, (*C)->content, (*C)->h);
```

编译命令行 (Makefile)，由 Python 调用

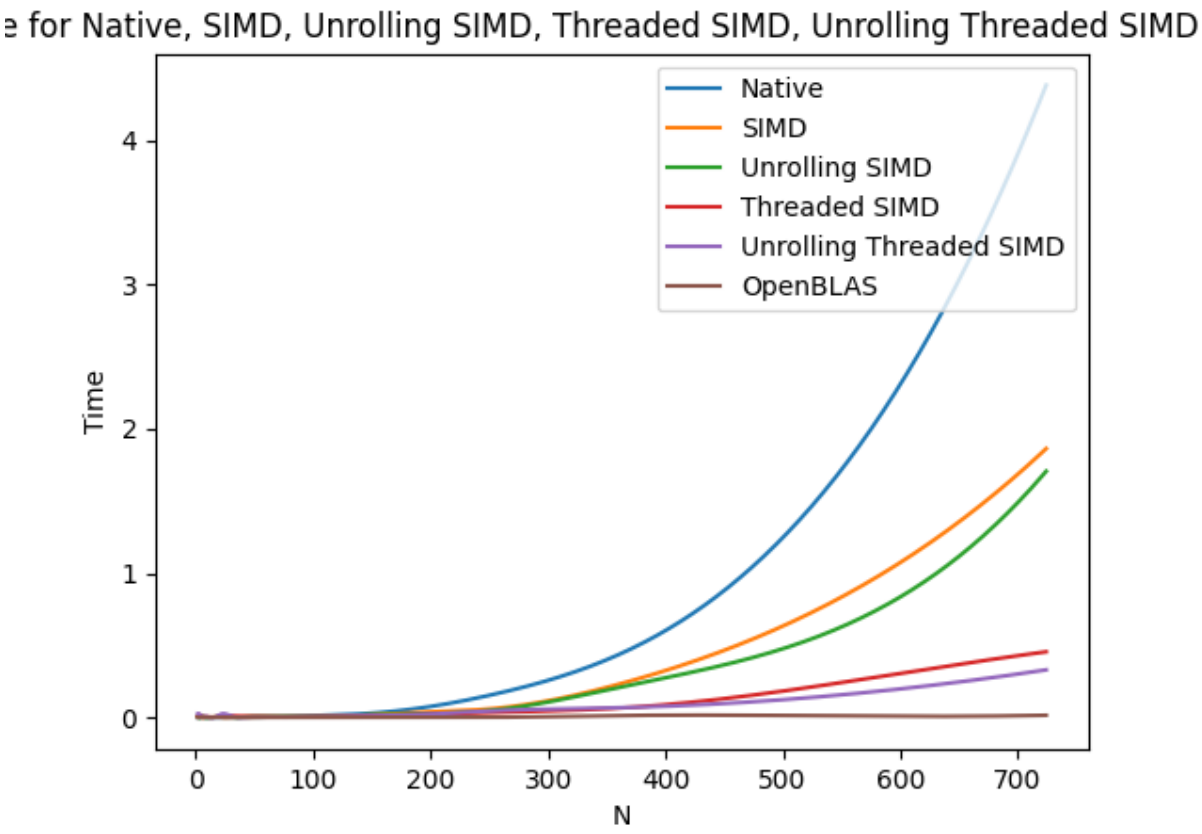
```
gcc src/mat.c src/mat_mul_test.c src/utils.c \
    -o ${build_path}/mat_mul_test \
    -lm \
    -L${OpenBLAS_LIBRARIES} \
    -I${OpenBLAS_INCLUDE_DIRS} \
    -lopenblas \
    -static \
    -lpthread \
    -mavx
```

实验命令行

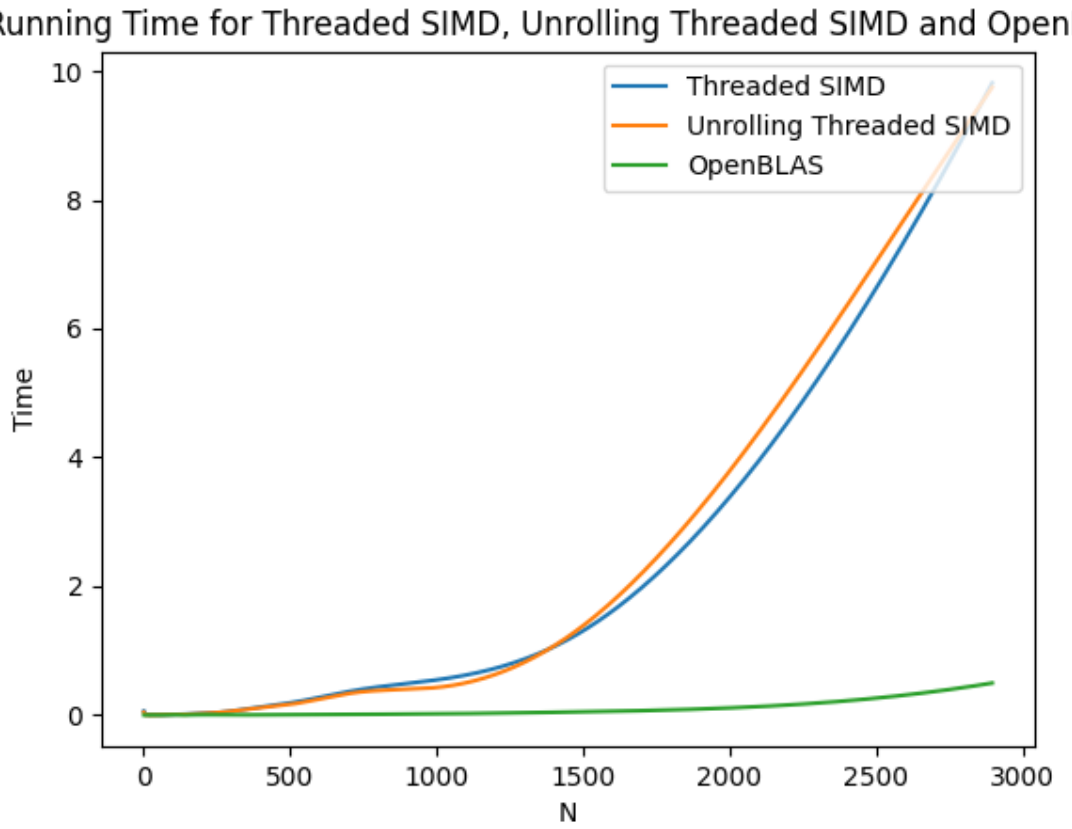
```
# 测试所有 6 种方案
python3 mul_test_all.py -b make -m 10 -s 1 -o plot_all.png
# 修改 c 文件后测试多线程的 3 种
python3 mul_test_all.py -b make -m 12 -s 1 -o plot_threaded.png
```

实验数据

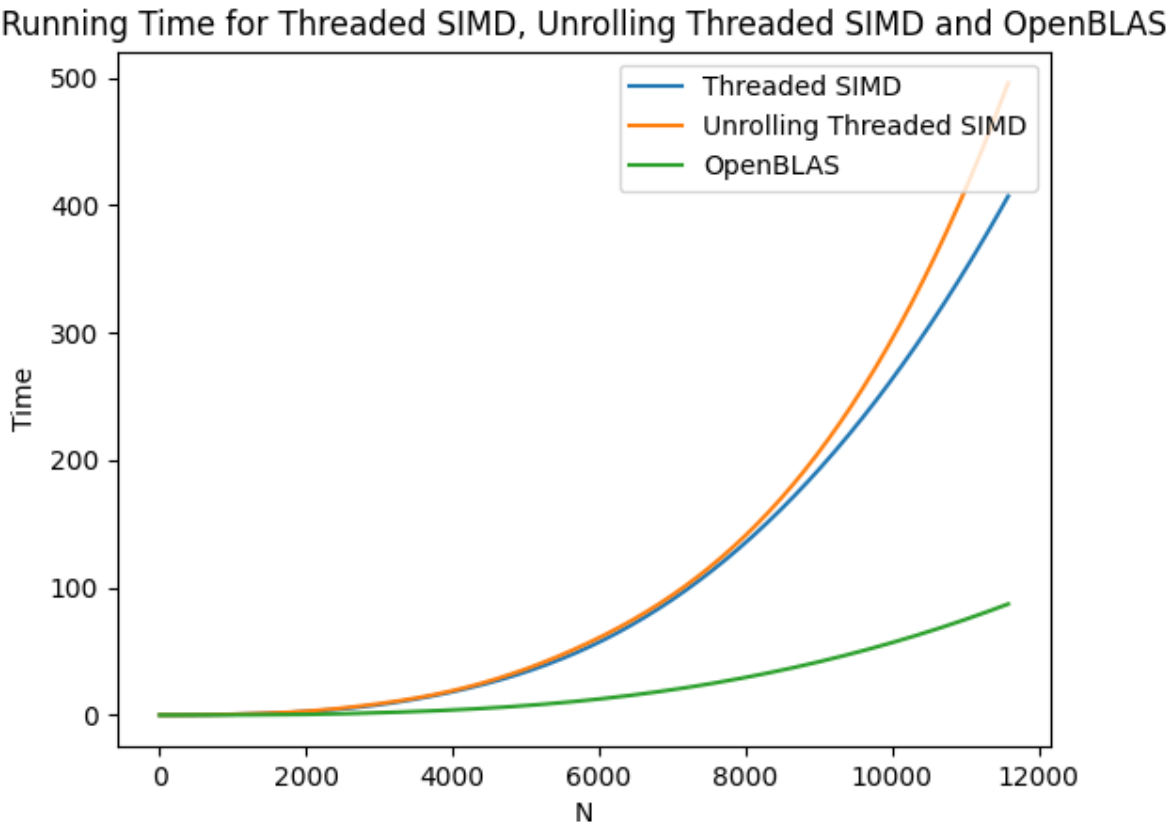
前两张图运行于 WSL2 Ubuntu 20.04，CPU 8 核 4800U；第三张图运行于一台服务器上的虚拟机 OpenEuler，CPU 32 核 Xeon Gold 5218。



图一，plot_all.png



图二，plot_threaded.png



图三，plot_32cores.png

实验结论

1. 多线程并行计算和单线程串行计算的比较

   由图一（plot_all.png）中 SIMD 和 Threaded SIMD 两条曲线可知，随着 N 的增大，多线程计算所用时间要远小于单线程的串行计算；而 Native 又远慢于 SIMD，SIMD 也算是一种并行计算，所以也可得知并行计算要优于单线程的串行计算。

2. 循环展开的性能分析

   由图一，在 8 核心时，手动 4 分支循环展开相比单线程要好上一些；但是在图二、图三，多线程场景下手动的循环展开却性能不如不展开。

   其原因可能是：

   1. 在核心数量更多的时候，4 分支循环展开并不能很好发挥性能
   2. 如果在编译的时候加上更好的优化，性能能更高，手动的 4 分支循环展开落后于不展开的代码更甚，说明编译器本来对于这样的计算就是有优化的，手动优化可能并不比编译器自动优化好，反而影响自动优化

同时，观察到 OpenBLAS 的性能远远比我自己实现的优化算法好得多，在小矩阵更加平稳，在大矩阵更快；而且在 OpenBLAS 运行时只占用一个 CPU 资源，但是速度是我的算法的 5 倍。

经过试验，我对多线程、并行计算等对计算效率的影响有了更深的理解，对于提升代码性能有了更加大的追求。