lab6 - openmp

实验环境

系统版本

```
Linux LAPTOP-BGRVTJ4L 5.15.153.1-microsoft-standard-WSL2 #1 SMP Fri Mar 29 23:14:13 UTC
2024 x86_64 x86_64 x86_64 GNU/Linux
```

```
Ubuntu 22.04.4 LTS
```

编译器版本

```
gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)
```

CPU 物理核数及频率

```
CPU(s):                  16
On-line CPU(s) list:   0-15
Thread(s) per core:       2
Core(s) per socket:       8
```

```
CPU: 3792.655MHz
```

四种矩阵乘实现

Naive

Naive gemm 是**最简单**的矩阵乘实现，*并未进行包括分块和多线程在内的优化*，因此也是四种实现中效率最低的实现。

Naive gemm 的 C 矩阵每个元素的计算公式如下：

$$\mathrm{C}[i,j] = \mathrm{C}[i,j] + \mathrm{A}[i,p] \times \mathrm{B}[p,j]$$

Naive gemm 通常使用最简单的三重循环实现，下面是其核心代码：

```
/* Macros for row-major order */
#define A(i, j) a[(i) * lda + (j)]
#define B(i, j) b[(i) * ldb + (j)]
#define C(i, j) c[(i) * ldc + (j)]

/* Routine for computing C = A * B + C */
void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc)
{
```

```
  int i, j, p;

  for (i = 0; i < m; i++) /* Loop over the rows of C */
  {
    for (j = 0; j < n; j++) /* Loop over the columns of C */
    {
      for (p = 0; p < k; p++)
      { /* Update C( i,j ) with the inner product of the ith row of A and the jth
column of B */
        C(i, j) = C(i, j) + A(i, p) * B(p, j);
      }
    }
  }
}
```

Openblas

Openblas 是 **BLAS（基础线性代数程序集）** 的一种开源实现。

使用 openblas 中的 `cblas_dgemm` 函数实现 gemm，核心代码如下：

```
void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc)
{
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k,
                1.0, a, lda, b, ldb, 1.0, c, ldc);
}
```

Pthread

POSIX线程（英语：POSIX Threads，常被缩写为pthreads）是POSIX的线程标准，定义了创建和操纵线程的一套API。实现POSIX线程标准的库常被称作**pthreads**。

使用 pthread 库进行多线程计算，实现 gemm，核心代码如下：

```
#include "defs.h"
#include <pthread.h>
#include <assert.h>
#include <stdio.h>

#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
#include <math.h>

struct MatrixThreadArgs {
    int m;
    int n;
    int k;
    double *a;
    int lda;
    double *b;
    int ldb;
    double *c;
```

```c
    int ldc;
    int section_x_begin;
    int section_x_end;
    int section_y_begin;
    int section_y_end;
};


void *MatrixThreadCalculate(void *arg) {
    struct MatrixThreadArgs matrixThreadArgs = *((struct MatrixThreadArgs *) arg);
    int k = matrixThreadArgs.k;
    double *a = matrixThreadArgs.a;
    int lda = matrixThreadArgs.lda;
    double *b = matrixThreadArgs.b;
    int ldb = matrixThreadArgs.ldb;
    double *c = matrixThreadArgs.c;
    int ldc = matrixThreadArgs.ldc;
    int section_x_begin = matrixThreadArgs.section_x_begin;
    int section_x_end = matrixThreadArgs.section_x_end;
    int section_y_begin = matrixThreadArgs.section_y_begin;
    int section_y_end = matrixThreadArgs.section_y_end;

    const int block_size = min(64, (max(section_x_end - section_x_begin, section_y_end
- section_y_begin)));
    int block_column_num = ceil((section_x_end - section_x_begin) / block_size);
    int block_row_num = ceil((section_y_end - section_y_begin) / block_size);

    for (int block_x = 0; block_x < block_column_num; block_x++) {
        for (int block_y = 0; block_y < block_row_num; block_y++) {
            int block_base_x = section_x_begin + block_x * block_size;
            int block_base_y = section_y_begin + block_y * block_size;
            int block_end_x = min(section_x_end, block_base_x + block_size);
            int block_end_y = min(section_y_end, block_base_y + block_size);

            for (int i = block_base_x; i < block_end_x; i++) {
                for (int j = block_base_y; j < block_end_y; j++) {
                    for (int p = 0; p < k; p++) {
                        C(i, j) = C(i, j) + A(i, p) * B(p, j);
                    }
                }
            }
        }
    }
    return NULL;
}


void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc) {
    const int x_seperate = 4, y_seperate = 4, thread_num = x_seperate * y_seperate;

    pthread_t threads[thread_num];
    struct MatrixThreadArgs sectionMatrixThreadArgs[thread_num];
    int rc;

    int section_weight = ceil(n / x_seperate);
    int section_height = ceil(m / y_seperate);
```

```c
    int thread_index = 0;
    for (int section_x = 0; section_x < x_seperate; section_x++) {
        for (int section_y = 0; section_y < y_seperate; section_y++) {
            int section_x_begin = section_x * section_weight;
            int section_x_end = min(n, section_x_begin + section_weight);
            int section_y_begin = section_y * section_height;
            int section_y_end = min(m, section_y_begin + section_height);

            sectionMatrixThreadArgs[thread_index] = (struct MatrixThreadArgs){
                .m = m,
                .n = n,
                .k = k,
                .a = a,
                .lda = lda,
                .b = b,
                .ldb = ldb,
                .c = c,
                .ldc = ldc,
                .section_x_begin = section_x_begin,
                .section_x_end = section_x_end,
                .section_y_begin = section_y_begin,
                .section_y_end = section_y_end
            };
            rc = pthread_create(&threads[thread_index], NULL, MatrixThreadCalculate,
&sectionMatrixThreadArgs[thread_index]);
            assert(rc == 0);
            thread_index++;
        }
    }
    for (int i = 0; i < thread_num; i++) {
        rc = pthread_join(threads[i], NULL);
        assert(rc == 0);
    }
}
```

Openmp

OpenMP（Open Multi-Processing）是一套支持*跨平台共享内存方式*的**多线程并发的编程 API**。

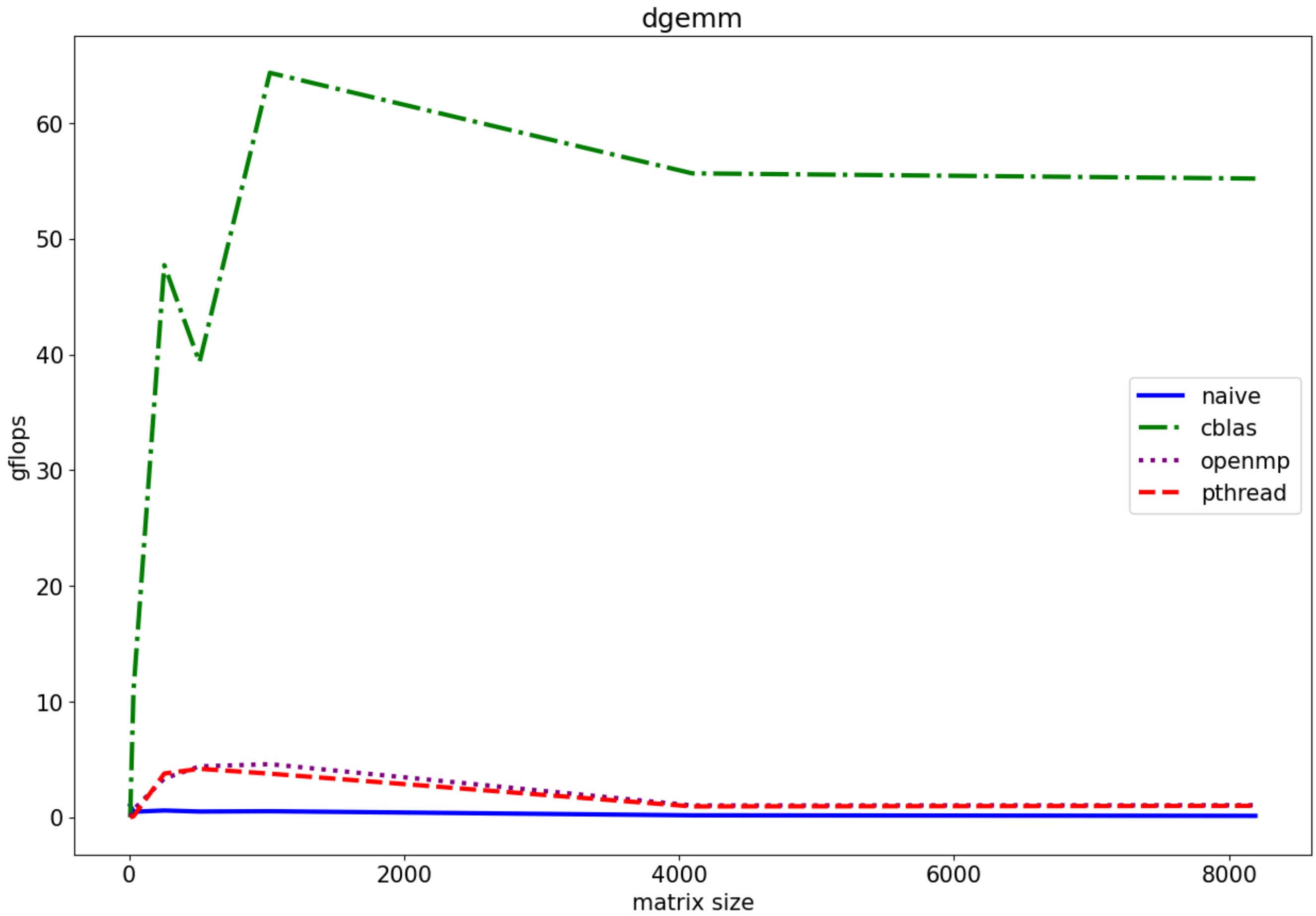
核心代码如下：

```c
void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc)
{
  int i, j, p;
  #pragma omp parallel for private(j, p)
  for (i = 0; i < m; i++) /* Loop over the rows of C */
  {
    for (j = 0; j < n; j++) /* Loop over the columns of C */
    {
      for (p = 0; p < k; p++)
      { /* Update C( i,j ) with the inner product of the ith row of A and the jth
column of B */
```

```
        C(i, j) = C(i, j) + A(i, p) * B(p, j);
        // printf("Thread %d: i=%d, j=%d, p=%d\n", omp_get_thread_num(), i, j, p);
      }
    }
    // printf("Thread %d: i=%d\n", omp_get_thread_num(), i);
  }
}
```

Gflops

下图为四种 gemm 实现在不同矩阵规模下的 gflops 曲线图：



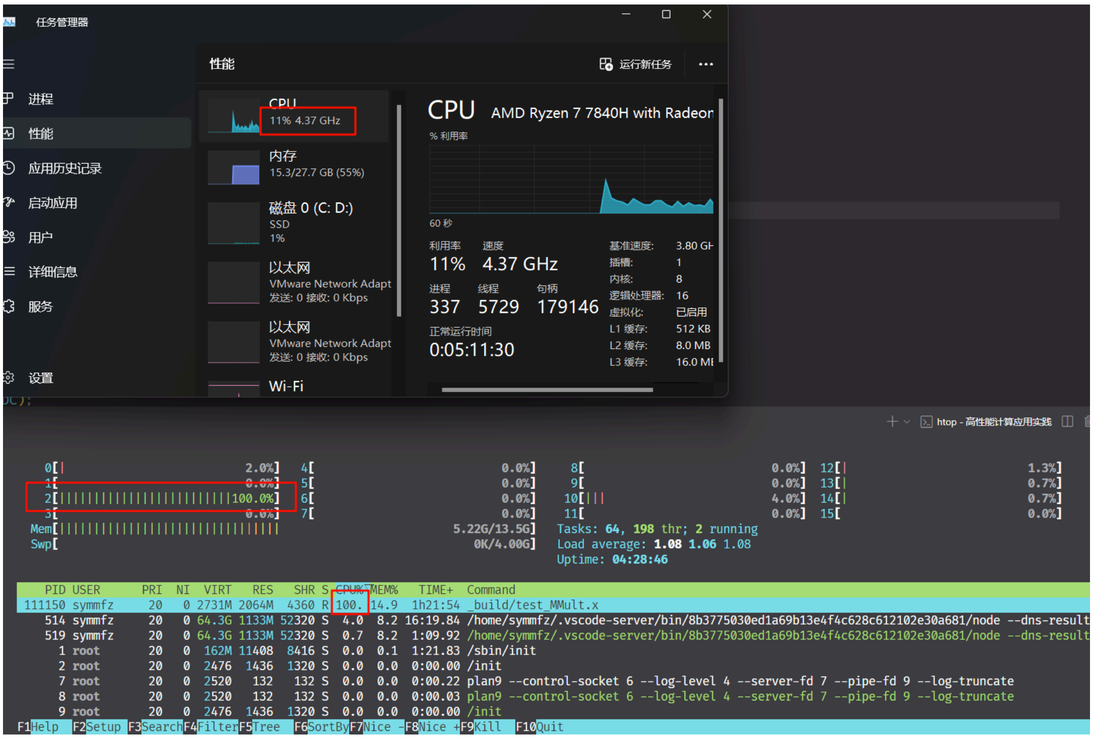
根据上图，不难发现以下结论：

- Cblas 实现的 gemm 在绝大多数矩阵规模下，gflops 都显著高于其他四种实现，最高可达 naive 实现的 500 倍以上
- Cblas 实现在较低矩阵规模时 gflops 较低，随着矩阵规模的上涨 gfops 值先快速上涨然后稳定。Gflops 峰值出现在 `1024*1024` 的矩阵规模附近，峰值大小约为 `64.35`
- Openmp 和 pthread 实现的 gflops 曲面相似，从 gflops 的大小上看，大于 naive 实现并显著小于 cblas 实现；从曲线的变化上看，随着矩阵规模的增大，gflops 先增后减，峰值出现在 `512*512` 或 `1024*1024` 附近，峰值 gflops 约为 `4.4`
- Naive 实现的 gflops 值在矩阵规模大于 `32*32` 时最低。从趋势上看，gflops 值大致随着矩阵规模的增大而减小

截图

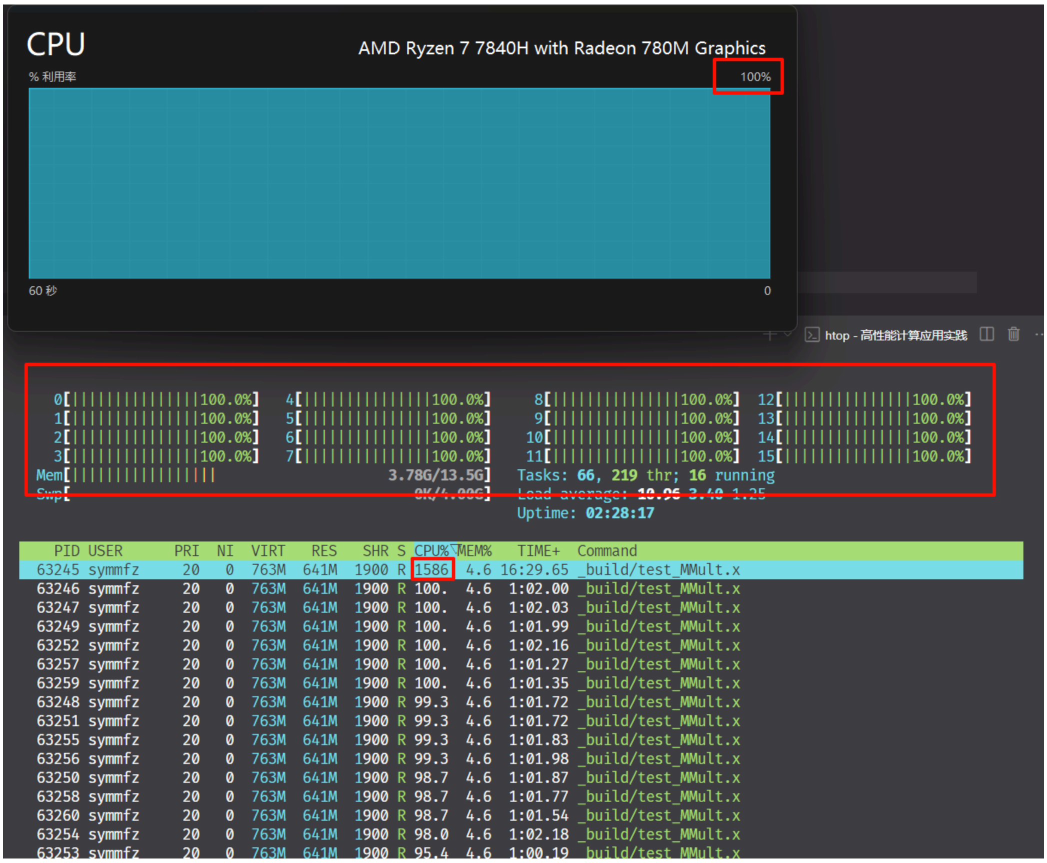
Naive

Naive 为单线程运行，无法充分利用 CPU 的性能。



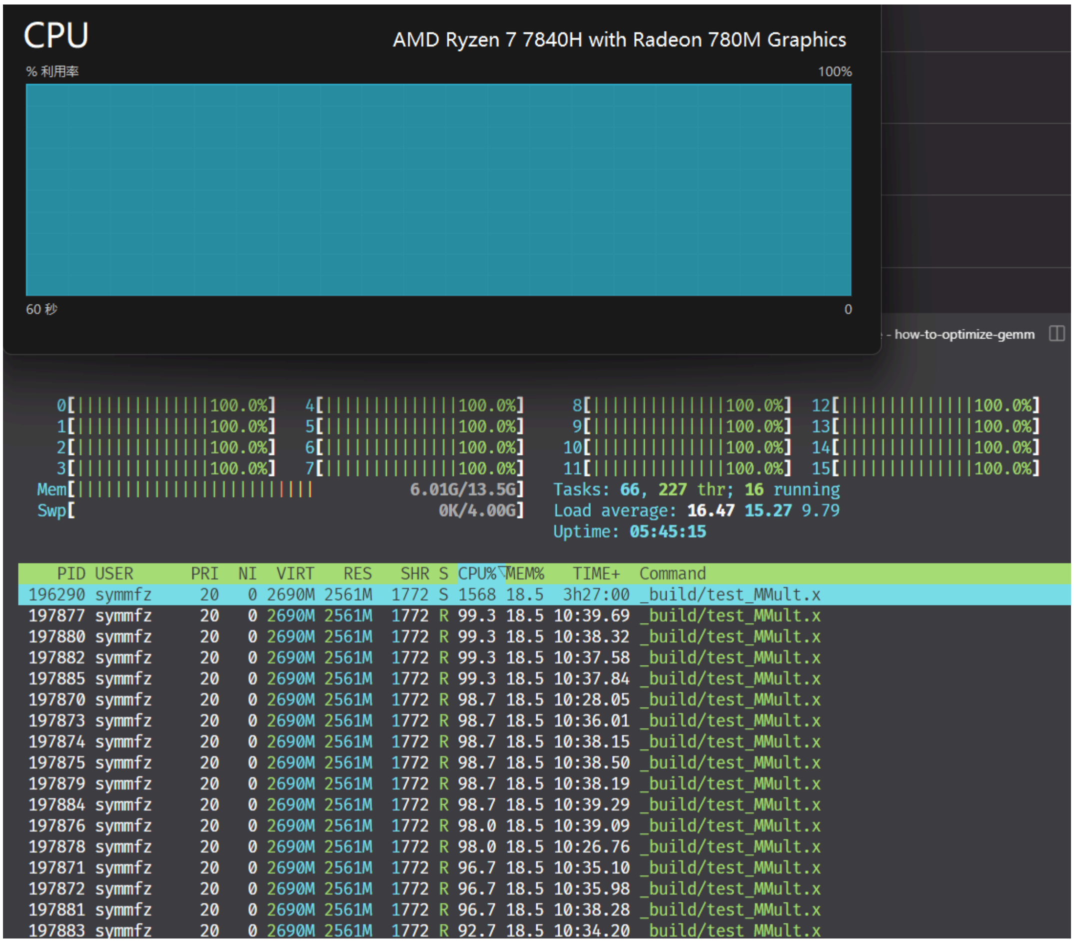
openblas

Openblas 可以充分利用 CPU 性能。



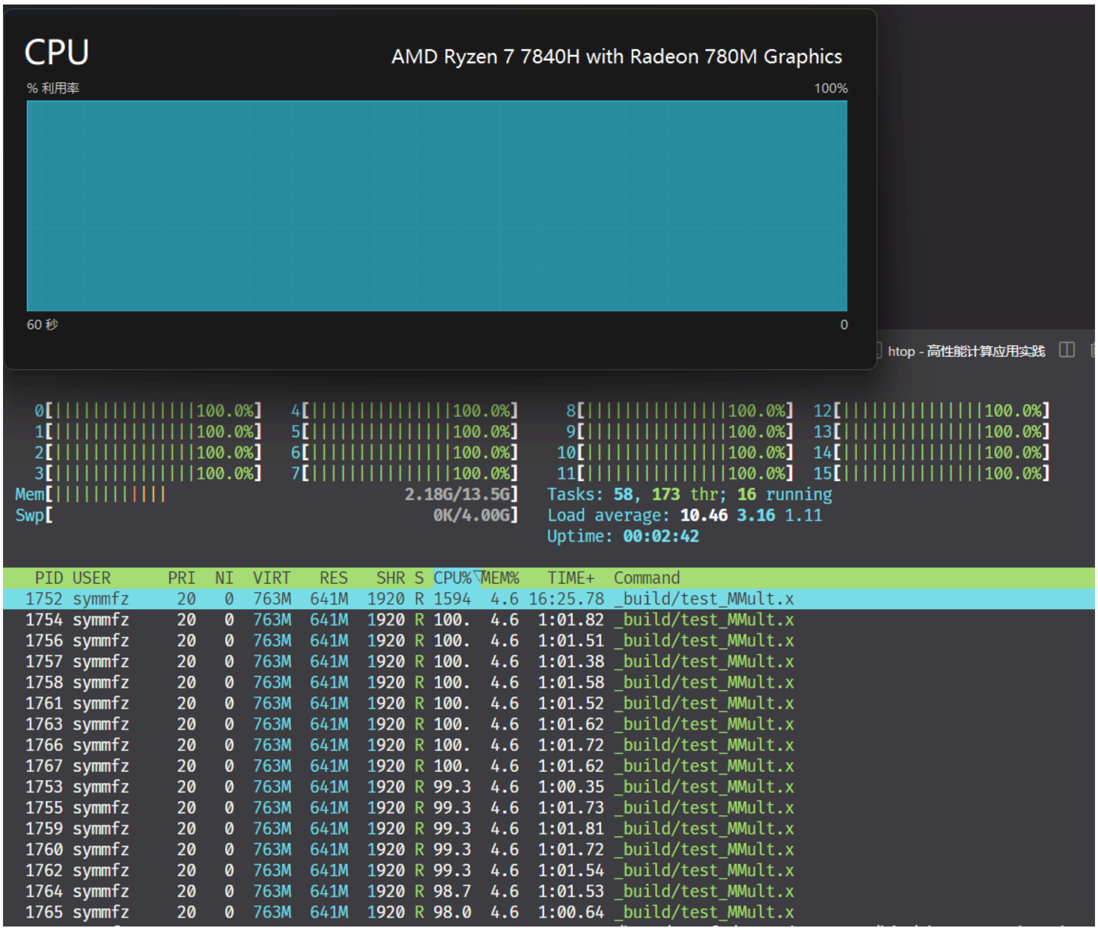
Pthread

Pthread 16 线程运行



Openmp

Openmp 使用 16 线程运行，可以充分利用 CPU 性能。



```
top - 23:24:49 up  2:31,  1 user,  load average: 2.12, 0.79, 0.42
Tasks:  61 total,   3 running,  58 sleeping,   0 stopped,   0 zombie
%Cpu(s): 98.4 us,  1.1 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.4 si,  0.0 st
MiB Mem :  13824.3 total,  11115.6 free,   2294.9 used,    413.9 buff/cache
MiB Swap:   4096.0 total,   4096.0 free,      0.0 used.  11256.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  88539 symmfz    20   0  289908 166092   1920 R  1582   1.2   3:30.61 test_MMult.x
```

```
─fish──sh──sh──sh──node─┬─node─┬─fish──test_MMult.x──15*[{test_MMult.x}]
                        │      ├─fish─┬─pstree
                        │      │      └─2*[{fish}]
                        │      └─12*[{node}]
                        ├─node──12*[{node}]
                        ├─node─┬─cpptools──25*[{cpptools}]
                        │      ├─node──10*[{node}]
                        │      ├─2*[node──6*[{node}]]
                        │      └─16*[{node}]
                        └─10*[{node}]
```

Lab 3 & Lab 5

Lab 3 - optimize-gemm

问题

> Question 1
>
> 多个 c 代码中有相同的 MY_MMult 函数，怎么判断可执行文件调用的是哪个版本的 MY_MMult 函数？是 makefile 中的哪行代码决定的？

> Answer
>
> MY_MMult 函数的版本由 makefile 文件决定，具体来说是由下面这行代码决定的：
>
> ```
> NEW  := openblas_MMult
> ```
>
> 这个决定是在以下这行代码中实现的：
>
> ```
> OBJS  := $(BUILD_DIR)/util.o $(BUILD_DIR)/REF_MMult.o $(BUILD_DIR)/test_MMult.o
> $(BUILD_DIR)/$(NEW).o
> ```
>
> 改变 NEW 的值即可改变调用的 MY_MMult，例如，上面 NEW 的值为 `openblas_MMult` ，表示 MY_MMult 将调用 openblas 实现的版本

> Question 2
>
> 性能数据 `_data/output_MMult0.M` 是怎么生成的？C 代码中只是将数据输出到终端并没有写入文件。

> Answer
>
> 性能数据 `_data/output_MMult0.m` 是通过将运行 `$(BUILD_DIR)/test_MMult.x` 的输出重定向到文件来生成的。这在 `run` 目标中的以下行完成：
>
> ```
> $(BUILD_DIR)/test_MMult.X >> $(DATA_DIR)/output_$(NEW).M
> ```
>
> 上面这行代码表示将程序的输出追加到性能数据文件中。

Lab 5 - thread

截图

```
top - 00:44:48 up  1:09,  1 user,  load average: 2.24, 1.59, 1.11
Tasks:   1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s): 52.3 us,  0.6 sy,  0.0 ni, 45.7 id,  0.0 wa,  0.0 hi,  1.4 si,  0.0 st
MiB Mem :  13824.3 total,  11132.9 free,   2135.1 used,    556.3 buff/cache
MiB Swap:   4096.0 total,   4096.0 free,      0.0 used.  11426.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  37461 symmfz    20   0  232364 165904   1776 S 800.0   1.2   3:11.61 test_MMult.x
```

```
top - 00:40:05 up  1:04,  1 user,  load average: 2.72, 1.56, 0.97
Threads:   9 total,   8 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s): 49.6 us,  0.3 sy,  0.0 ni, 48.8 id,  0.0 wa,  0.0 hi,  1.2 si,  0.0 st
MiB Mem :  13824.3 total,  11130.1 free,   2138.7 used,    555.5 buff/cache
MiB Swap:   4096.0 total,   4096.0 free,      0.0 used.  11423.2 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  35215 symmfz    20   0  232364 165884   1760 R  99.9   1.2   0:13.17 test_MMult.x
  35216 symmfz    20   0  232364 165884   1760 R  99.9   1.2   0:13.17 test_MMult.x
  35217 symmfz    20   0  232364 165884   1760 R  99.9   1.2   0:13.17 test_MMult.x
  35218 symmfz    20   0  232364 165884   1760 R  99.9   1.2   0:13.17 test_MMult.x
  35219 symmfz    20   0  232364 165884   1760 R  99.9   1.2   0:13.17 test_MMult.x
  35220 symmfz    20   0  232364 165884   1760 R  99.9   1.2   0:13.17 test_MMult.x
  35221 symmfz    20   0  232364 165884   1760 R  99.9   1.2   0:13.17 test_MMult.x
  35222 symmfz    20   0  232364 165884   1760 R  99.9   1.2   0:13.17 test_MMult.x
  34099 symmfz    20   0  232364 165884   1760 S   0.0   1.2   1:39.55 test_MMult.x
```