CS205 C/ C++ Programming - Project 4: Matrix multiplication race

Name: Lv Yue
SID: 11710420

This project is hosted at https://github.com/JustLittleFive/Matrix_Multiplication_Race

Part1: Analysis

Project goal: Implement matrix multiplication in C and try HARD to improve its speed, then compare with OpenBlas or else.

Part2: Core code

The `header.h` include all required *C standard library header files*, declare everything need by benchmark.

The `source2.c` provides the time and correctness benchmark of `OpenBlas` and the different implemented functions in `source1.c`.

The `source1.c` implement 5 different method for matrix multiply:

1. The `matmul_native` function is the most native way to do matrix mutiply...?

```
void matmul_native(int m, int n, int p, const fdata* x, int ldx, const fdata* y,
                   int ldy, fdata* z, int ldz) {
  int i, j, k;
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
      for (k = 0; k < p; k++) {
        z[i * ldz + j] += x[i * ldx + k] * y[k * ldy + j];
      }
    }
  }
}
```

No! It is already optimized by allocate fdata with alignment.

```
typedef float fdata;

// allocate fdata with alignment to optimize memory and cache
#define ALLOC_ARR(n) \
  (fdata*)calloc_align(sizeof(fdata) * 4, (n) * sizeof(fdata))

#define ALLOC_ARRD(n) \
  (double*)calloc_align(sizeof(double) * 4, (n) * sizeof(double))
```

However, it is still using looping-looping-loop in calculate order, as the baseline of this project.

---

2. The `matmul_native_with_omp` function is one way to improve `matmul_native`,
   by using OpenMP to parallzlize the `for` loop. However we won't use it again due to its unstable output, which can magnify the loss of precision.

---

3. The `matmul_cache_opt` function is optimizing cache hit rate by reversing loop order.

```
// naive method with cache hit optimization
void matmul_cache_opt(int m, int n, int p, const fdata* x, int ldx,
                      const fdata* y, int ldy, fdata* z, int ldz) {
  // i-k-j order matrix mul, optimize the cache hits.
  int i, j, k;
// #pragma omp parallel for
  for (i = 0; i < m; i++) {
    for (k = 0; k < p; k++) {
      register fdata aik = x[i * ldx + k];
      for (j = 0; j <= n - 4; j += 4) {
        // loop unrolling
        register const fdata* yptr = y + (k * ldy + j);
        register fdata* zptr = z + (i * ldz + j);

        // move pointer instead of computing address,
        // which involves multiplication
        *zptr += aik * (*yptr);
        zptr++;
        yptr++;
        *zptr += aik * (*yptr);
        zptr++;
        yptr++;
        *zptr += aik * (*yptr);
        zptr++;
        yptr++;
        *zptr += aik * (*yptr);
        zptr++;
        yptr++;
      }
      switch (n - j) {
```

The memory access discontinuity is one of the main reasons of high time cost of matrix multiplication. Assume the `n` dimentions matrix store in one dimention array and store by row, then calculate one element in normal way will jump `1` time in one matrix and `n` times in another matrix in memory. Which means if calculate in `i-j-k` order will jump `n^3+n^2-n` times, but if calculate in `i-k-j` order will only cost `n^2`

jumps.
This is how to optimize cache hit rate by reversing loop order.

---

4. The `matmul_dc_opt` function is based on `matmul_cache_opt` function adding divide-and-conquer optimization. The goal of divide is matrix size reach `64^2` of smaller.

---

5. The `matmul_strassen` function is strassen algorithm combine with `matmul_cache_opt`.

```
void matmul_strassen(int m, int n, int p, const fdata* a, int lda,
                     const fdata* b, int ldb, fdata* c, int ldc) {
  // when reach goal size just use matmul_cache_opt
  if (m <= STRAN_THR && n <= STRAN_THR && p <= STRAN_THR) {
    matmul_cache_opt(m, n, p, a, lda, b, ldb, c, ldc);
    return;
  }

  // get HALF of the input size.
  int hm = m / 2;
  int hn = n / 2;
  int hp = p / 2;

  // padding flag on each dimension
  int padding_m = (m % 2);
  int padding_n = (n % 2);
  int padding_p = (p % 2);

  // the size of matrix after padding,
  // (m % 2 == 1 means m is odd and padding is needed.)
  // which also is the matrix size involves matrix mul.
  int phm = hm + padding_m;
  int phn = hn + padding_n;
  int php = hp + padding_p;

  // intermidiate matrices
  fdata* lhs = ALLOC_ARR(phm * php);  // left-hand-side
  fdata* rhs = ALLOC_ARR(php * phn);  // right-hand-side
  fdata* l;
  fdata* r;
```

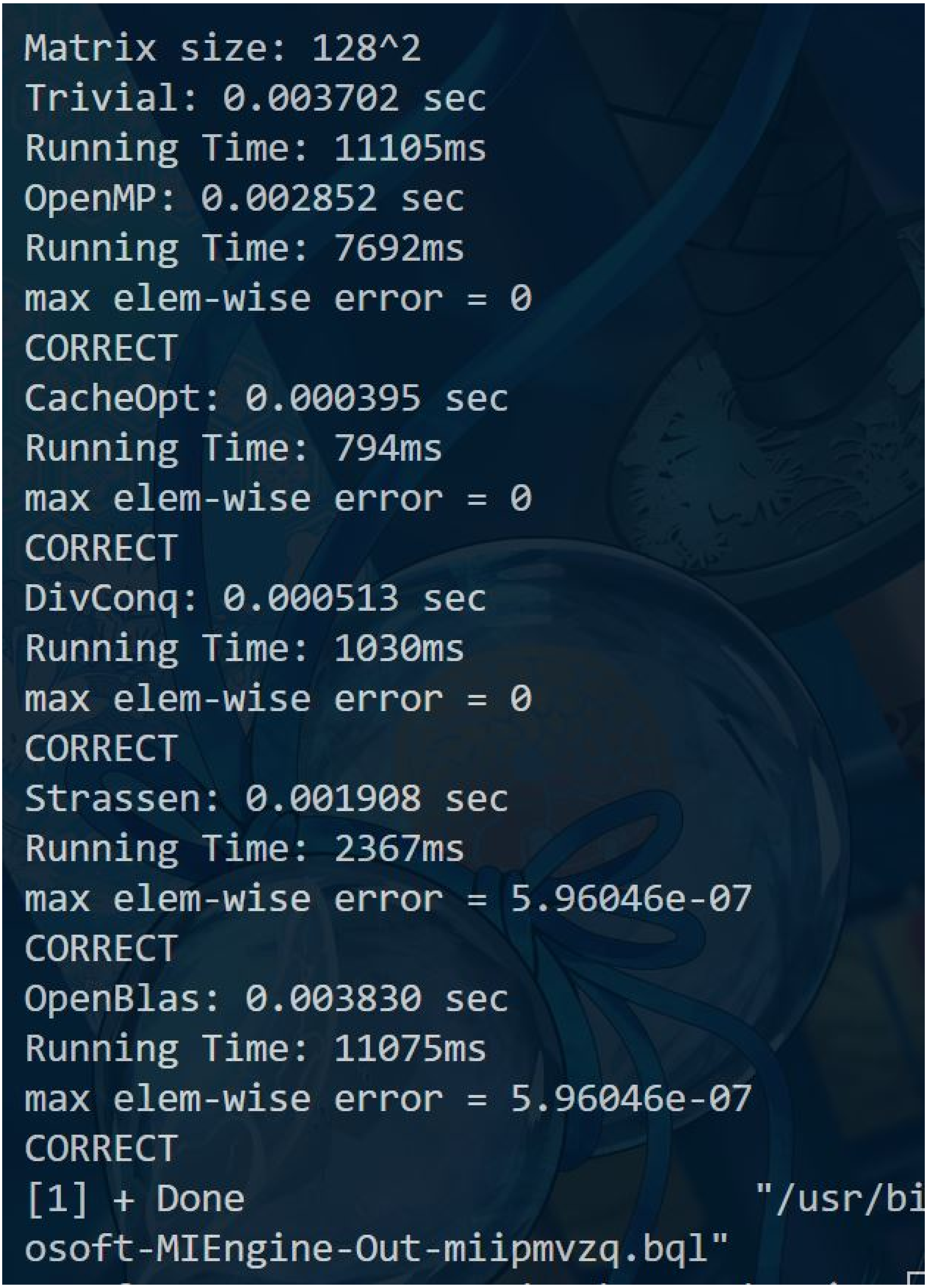
The core idea of strassen algorithm is `2^2 matrix multiplication needed multiplication times can be reduced from 8 to 7`. Use this idea on the primary matrix ***recursively*** is the basic idea of this algorithm.

Part 3: Result & Verification

Test 16*16 matrix multiplication:

```
Matrix size: 16^2
Trivial: 0.000008 sec
Running Time: 30ms
OpenMP: 0.000007 sec
Running Time: 16ms
max elem-wise error = 0
CORRECT
CacheOpt: 0.000003 sec
Running Time: 7ms
max elem-wise error = 0
CORRECT
DivConq: 0.000003 sec
Running Time: 7ms
max elem-wise error = 0
CORRECT
Strassen: 0.000003 sec
Running Time: 6ms
max elem-wise error = 0
CORRECT
OpenBlas: 0.006527 sec
Running Time: 16070ms
max elem-wise error = 0
CORRECT
[1] + Done                    "/
osoft-MIEngine-Out-x0ajnqj1.tdf"
root@DESKTOP-VH54EV2:~/workSpace/pr
```

Test 128x128 matrix multiplication:

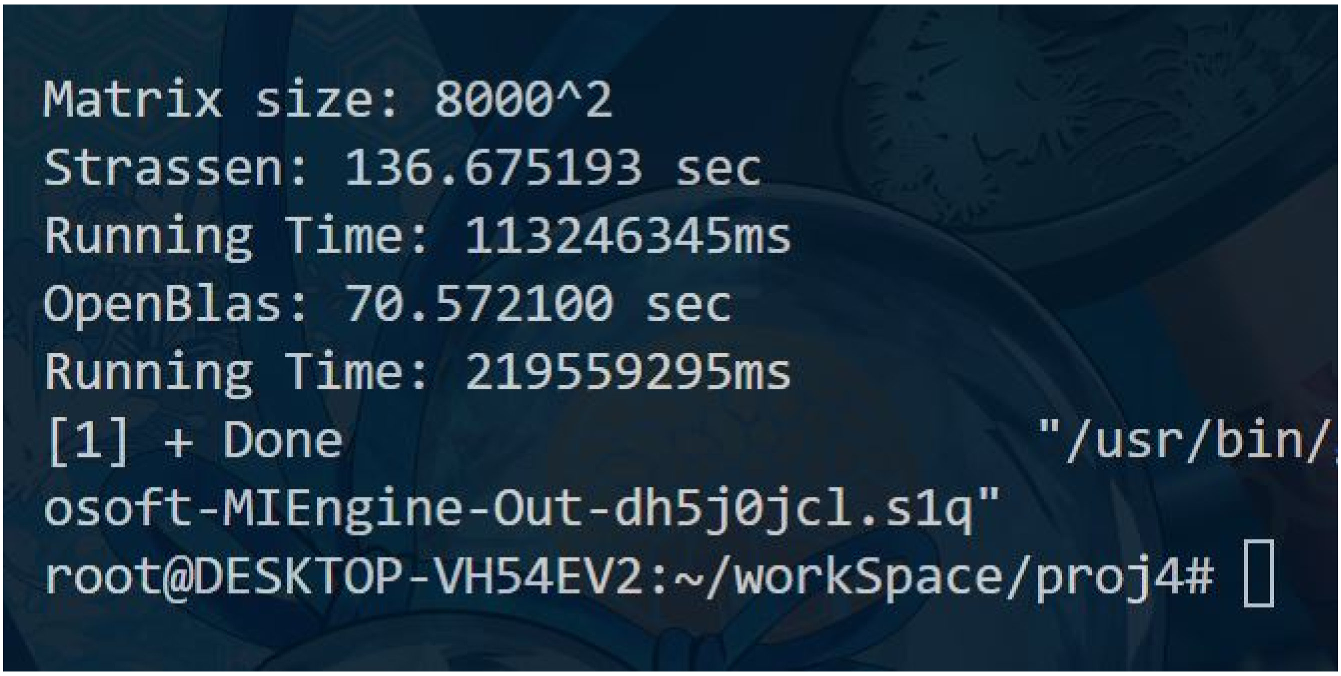


```
Matrix size: 128^2
Trivial: 0.003702 sec
Running Time: 11105ms
OpenMP: 0.002852 sec
Running Time: 7692ms
max elem-wise error = 0
CORRECT
CacheOpt: 0.000395 sec
Running Time: 794ms
max elem-wise error = 0
CORRECT
DivConq: 0.000513 sec
Running Time: 1030ms
max elem-wise error = 0
CORRECT
Strassen: 0.001908 sec
Running Time: 2367ms
max elem-wise error = 5.96046e-07
CORRECT
OpenBlas: 0.003830 sec
Running Time: 11075ms
max elem-wise error = 5.96046e-07
CORRECT
[1] + Done                    "/usr/bi
osoft-MIEngine-Out-miipmvzq.bql"
```

Test 1k*1k matrix multiplication:

```
Matrix size: 1024^2
Trivial: 9.417308 sec
Running Time: 9416137ms
OpenMP: 9.358938 sec
Running Time: 9357884ms
max elem-wise error = 0
CORRECT
CacheOpt: 0.338468 sec
Running Time: 338477ms
max elem-wise error = 0
CORRECT
DivConq: 0.240115 sec
Running Time: 240124ms
max elem-wise error = 0
CORRECT
Strassen: 0.191072 sec
Running Time: 191077ms
max elem-wise error = 1.50502e-05
ERROR
OpenBlas: 0.065765 sec
Running Time: 251816ms
max elem-wise error = 1.50502e-05
ERROR
[1] + Done                       "/usr/bin
osoft-MIEngine-Out-dke3xrdq.qlw"
root@DESKTOP-VH54EV2:~/workSpace/proj4#
```

Test 8k*8k matrix multiplication (the full result is still running, could check on github later):



```
Matrix size: 8000^2
Strassen: 136.675193 sec
Running Time: 113246345ms
OpenBlas: 70.572100 sec
Running Time: 219559295ms
[1] + Done                    "/usr/bin/
osoft-MIEngine-Out-dh5j0jcl.s1q"
root@DESKTOP-VH54EV2:~/workSpace/proj4#
```

64k*64k test cause core dump, because running out of memory.

Part 4 - Difficulties & Solutions

1. `gettimeofday()` and `clock()` return different timing result.
   My choice is to keep both: after the optimization method is the result of `gettimeofday()` timing, and after running time is the result of the above optimization method timing by `clock()`.
2. Out of memory.
   *It cannot be solved, the computer memory and hard disk are neither not enough to run the matrix multiplication of 64k*64k size.*