Lab1: FFT

一、运行结果

```
user20302010040@ubuntu01:~$ ./fft 25
size: 33554432
minimal time spent: 3879.3119 ms
threads: 24, baseline time spent: 256.9555 ms
result: correct (err = 2.983364e-11)
user20302010040@ubuntu01:~$
```

二、算法思路

主要目标是实现cache-oblivious的FFT算法，本部分描述串行化实现，下一部分进行优化。

首先需要实现cache-oblivious的矩阵转置函数，论文中的相关描述如下图所示：

> Optimal work and cache complexities can be obtained with a divide-and-conquer strategy, however. If $n \geq m$, the REC-TRANSPOSE algorithm partitions
>
> $$A = (A_1\ A_2)\,, \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$
>
> and recursively executes REC-TRANSPOSE(A_1, B_1) and REC-TRANSPOSE(A_2, B_2). Otherwise, it divides matrix A horizontally and matrix B vertically and likewise performs two transpositions recursively. The next two lemmas provide upper and lower bounds on the performance of this algorithm.

该函数中，对于矩阵$A_{m\times n}$和需要求得的转置矩阵$B_{n\times m}$，按照n和m的大小关系进行横切或纵切，递归地对小矩阵进行转置操作，从而减少cache miss。由于FFT中使用该函数时，被转置的矩阵其实是一维数组，所以在实现函数`transpose`时，按照一维数组的形参，如下：

```
B[j * M + i] = A[i * N + j];
```

函数`student_fft`分为六步，首先要将输入输出数组视为二维矩阵（$X_{n1\times n2}, Y_{n2\times n1}$），求出$n_1$和$n_2$，满足$n_1 == n_2$或$n_1 == 2n_2$，之后$X$上的$(j1, j2)$为$X[j_1 n_2 + j_2]$，$Y$上的$(i_2, i_1)$为$Y[i_2 n_1 + i_1]$，可推导公式得：

$$Y[i_2 n_1 + i_1] = \Sigma_{j_2=0}^{n_2-1}((\Sigma_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] W_{n_1}^{-i_1 j_1}) W_n^{-i_1 j_2}) W_{n_2}^{-i_2 j_2}. i_1, j_1 \in [0, n_1 - 1], i_2, j_2 \in [0, n_2 - 1]$$

六步分别为：转置，FFT，乘twiddle factor，转置，FFT，转置。

- 转置。$x[j_1 n_2 + j_2] \longrightarrow a[j_2 n_1 + j_1]$

```
transpose(n1, n2, in, arr1, n1, n2, 0, 0);
```

- FFT。$a[j_2n_1+j_1] \longrightarrow b[j_2n_1+i_1]$

```
  for (int i = 0; i < n2; i++) {
    naive_fft(n1, arr1 + n1 * i, arr2 + n1 * i);
  }
```

- 乘twiddle factor。$b[j_2n_1+i_1] \longrightarrow c[j_2n_1+i_1]$

```
  for (int i = 0; i < n2; i++) {
    double theta_i = theta0 * i;
    for (int j = 0; j < n1; j++) {
      arr2[i * n1 + j] *= (cos(theta_i * j) - sin(theta_i * j) * I);
    }
  }
```

- 转置。$c[j_2n_1+i_1] \longrightarrow d[i_1n_2+j_2]$

```
transpose(n2, n1, arr2, arr1, n2, n1, 0, 0);
```

- FFT。$d[i_1n_2+j_2] \longrightarrow e[i_1n_2+i_2]$

```
  for (int i = 0; i < n1; i++) {
    naive_fft(n2, arr1 + n2 * i, arr2 + n2 * i);
  }
```

- 转置。$e[i_1n_2+i_2] \longrightarrow y[i_2n_1+i_1]$

```
transpose(n1, n2, arr2, out, n1, n2, 0, 0);
```

三、优化方法

- 使用OpenMP进行并行设计。在student_fft的step2和step5均涉及到复数个fft的计算，通过并行可以进行程序的优化；同时，step3的乘法同样可以利用并行来加快速度。
- 使用cache-oblivious的矩阵转置。该转置方法通过减少cache miss的次数来优化程序。
- 避免重复计算。在student_fft和naive_fft中均提前计算`2 * PI / n`，该值在循环中多次使用，在循环迭代中保持不变，单次计算来减小计算开销。