山东大学

SHANDONG UNIVERSITY

计算机系统原理第一次实验报告

小组成员及分工

姓名	班级	学号	分工
鲍泓睿	密码1班	202200460076	ntt 函数多线程优化思路、ntt 函数多线程优化实现以及线程池优化实现以及相应部分的报告撰写
高钲超	密码1班	202200460136	ntt 函数多线程优化思路、ntt 函数 GPU 优化实现以及相应部分的报告撰写
陈万里	网安2班	202200460153	正确性测试，报告排版及撰写
郑傲宇	网安2班	202222460130	基准测试，openmp

目录

1 实验要求

利用**多线程设计**实现NTT的加速。

1. 用多线程设计方法，对NTT的C++简易实现代码进行优化(见附件ntt.cpp)，降低”ntt”函数的运行时间，时间测试方式详见代码。在实验设计中尽量考虑输入线程数、硬件配置（例如处理器核数、cache大小）等各种你认为对运行时间会产生影响的参数。

2. 数论变换（Number Theoretic Transform，NTT）是离散傅里叶变换（Discrete Fourier Transform, DFT）在有限域下的等价物，在计算机科学和密码学中有广泛应用。附件代码中实现了用NTT加速大数乘法的过程。关于NTT的介绍与代码逻辑可以参考：https://zhuanlan.zhihu.com/p/80297169

2 实验环境

若无特殊说明，基准测试均在以下环境内运行。

2.1 硬件环境

CPU	Intel i7-12700H 6P8E P Core @4.70 GHz E Core@3.50 GHz （在 Linux 内核下，CPU 0-11 为 P 核心，12-19 为 E 核心）
RAM	64G DDR5 4800MHz Dual Channel

2.2 软件环境

操作系统内核	Linux kernel 6.9.0
编译器	GCC 13.2.1
编译选项	g++ -o ntt ./ntt.cpp -O3
测试命令	taskset -c 0-11 /usr/bin/perf stat -d -d -d ./ntt

对于以上环境的说明在 “4.1 环境说明” 部分

2.3 正确性测试

运行以下代码，替换 ntt_threads 部分，通过比对最初版本的函数运行结果以及更改后的函数运行结果，测试程序的正确性：

```cpp
#include<iostream>
#include<algorithm>
#include<cstring>
#include<chrono>
#include<thread>
#include<vector>
using namespace std;
const int N = 300100, P = 998244353;      //模数为p，数组长度限制为N
int A[N], B[N], C[N], r[N];
int qpow(int x, int y)//快速模幂算法
{
      int res(1);
      while (y)
     {
            if (y & 1) res = 1ll * res*x%P;
            x = 1ll * x*x%P;
            y >>= 1;
     }
      return res;
}

```

```cpp
void ntt(int *x, int lim, int opt)//正确的 NTT，单线程基准
{
	int i, j, k, m, gn, g, tmp;
	for (i = 0; i < lim; ++i)
		if (r[i] < i)
			swap(x[i], x[r[i]]);
	for (m = 2; m <= lim; m <<= 1)
	{
		k = m >> 1;
		gn = qpow(3, (P - 1) / m);
		for (i = 0; i < lim; i += m)
		{
			g = 1;
			for (j = 0; j < k; j++, g = 1ll * g*gn%P)
			{
				tmp = 1ll * x[i + j + k] * g%P;
				x[i + j + k] = (x[i + j] - tmp + P) % P;
				x[i + j] = (x[i + j] + tmp) % P;
			}
		}
	}
}

void ntt_thread(int *x, int lim, int opt)//本实验中实现的多线程 NTT
{
	// 多线程 NTT 实现
}

int main() {
	srand(time(nullptr));
	int i, lim = 1, n = N / 2;
	for (i = 0; i < n; i++) {
		A[i] = rand() % 10;
		B[i] = A[i]; // 将 A 和 B 初始化为相同的数组
	}
	// 计算适当的 lim 值
	while (lim < n) lim <<= 1;
	// 初始化 r 数组
	for (i = 0; i < lim; ++i) {
		r[i] = (i & 1) * (lim >> 1) + (r[i >> 1] >> 1);
	}
	// 对 A 和 B 进行 NTT
	ntt(A, lim, 1);
	ntt_thread(B, lim, 1);
	for(int i = 0; i < n; i++){
		if(A[i]!=B[i]){
			cout<<"error"<<endl; // 若有错误，即输出 errot
			break;
		}
	}
	// 若无错误，则无输出
```

```
73      return 0;
74  }
```

可以配合如下脚本，连续运行若干次，均未发现错误输出，则基本可以认为程序的正确性不存在问题。同时，这个脚本还能用于临时简单分析，快速计算程序运行平均时间。使用方法：`./test.sh [your cpp file] [test times]`。

```
#!/bin/sh
# 执行一次g++ 参数为$1
arg=$1
g++ -g -O3 $arg

# 创建一个变量来保存总的执行时间
total_time=0

# 循环$2次
for ((i = 0; i < $2; i++)); do
   result=$(./a.out)
   pattern='took ([0-9.]+)e\+([0-9]+)'
   if [[ $result =~ $pattern ]]; then
      value=${BASH_REMATCH[1]}
      exponent=${BASH_REMATCH[2]}
      exponent_value=$(awk "BEGIN {print 10^$exponent}")
      time_decimal=$(awk "BEGIN {print $value * $exponent_value}")
      total_time=$(awk "BEGIN {print $total_time + $time_decimal}")
      echo -e "\033[1;33m[o]\033[0m Original value:\n $result"
   fi
done

# 计算平均时间
average_time=$(awk "BEGIN {print $total_time / $2}")

# 输出平均时间
echo
echo -e "\033[1;32m[+]\033[0m Average time: \033[1;34m$average_time\033[0m
nanoseconds"
```

3 实验过程

3.1 初步优化及问题

3.1.1 学习利用 thread 实现多线程编程

要求利用多线程设计实现 NTT 的加速，我们首先要了解如何利用多线程编程来实现。

通过查阅资料得知，多线程编程是一种同时执行多个线程以完成任务的编程技术。在多核处理器的环境下，充分利用多线程可以提高程序的性能和并发性。其基本方法为：

1. 创建线程：使用编程语言提供的线程库或 API 创建线程，指定线程要执行的函数或代码块。
2. 线程同步：在多线程程序中，多个线程可能同时访问共享资源，需要使用同步机制来确保数据的一致性和正确性。
3. 线程间通信：多个线程之间可能需要进行通信以协调彼此的工作。线程间通信可以通过共享内存、消息队列、信号量等方式来实现。
4. 任务分解：将大任务分解成多个较小的任务单元，每个任务单元由一个线程来执行。这样可以提高程序的并发性和效率。

而在进行多线程优化的时候，其关键思路是并行化，即将程序中的独立任务并行化，利用多线程同时执行，提高整体的处理速度。

在 C++ 中，thread 头文件提供了多线程编程所需的类和函数，用于创建和管理线程。使用 thread 头文件可以在 C++ 程序中实现多线程功能。通过对多线程编程的初步学习，我们利用 thread 进行了初步优化（尽管后来发现是错的）。

为了优化，我们试图对相关函数的不同部分进行拆分，我们观察到函数中有许多 for 循环，而在单线程中，运行 for 循环是依次进行的，所以我们试图将一个大的 for 循环拆分成许多个独立的小的 for 循环进行任务分解，以实现其并行计算。

以下是我们初步尝试对 ntt 进行优化的版本之一。在这个版本中，我们设定线程数 num_threads，之后用lim/num_threads 确定每一个线程处理的长度，然后将这个长度传到后续 for 循环中并行处理，当第一个 for 循环每循环一次，建立一个线程数组，再建立相应线程数的线程依次处理各个部分的操作，我们将 ntt 函数拆分成两个函数，其中 sub_ntt 是进行后续 for 循环操作的函数，具体代码如下：

```
1  void sub_ntt(int* x, int start, int end) {
2      int i, j, k, m, gn, g, tmp;
3      for (m = 2; m <= end; m <<= 1) {
4          k = m >> 1;
5          gn = qpow(3, (P - 1) / m);
6          for (i = start; i < end; i += m) {
7              g = 1;
```

```
                for (j = 0; j < k; j++, g = 1LL * g * gn % P) {
                    tmp = 1LL * x[i + j + k] * g % P;
                    x[i + j + k] = (x[i + j] - tmp + P) % P;
                    x[i + j] = (x[i + j] + tmp) % P;
                }
            }
        }
}

void ntt(int* x, int lim, int opt) {
     for (int i = 0; i < lim; ++i) {
          if (r[i] < i) swap(x[i], x[r[i]]);
     }
     const int num_threads = 7; // 定义线程数量
     vector<thread> threads(num_threads); // 定义线程数组
     int step = lim / num_threads; // 计算每个线程要处理的数据范围
     // 创建并启动线程
     for (int i = 0; i < num_threads; ++i) {
          int start = i * step;
          int end = (i + 1) * step;
          threads[i] = thread(sub_ntt, x, start, end); // 创建线程并加入线程数组
     }
     // 等待所有线程执行完成
     for (auto& thread : threads) {
          thread.join();
     }
}
```

根据阅读代码，我们初步猜测第一个 for 循环是进行排序操作，拆分后可能会影响最终结果，所以保留不变，我们尝试将后边的三个 for 循环进行拆分，通过更改 for 循环的操作范围，来拆分成几个独立的任务。

3.1.2 初步尝试中的问题及猜测

本部分中的测试均为初步测试，较为严谨的基准测试在“4 基准测试”部分

3.1.2.1 性能

```
$ g++ -o ntt ./ntt.cpp # 这里不使用 O3 编译器优化
$ taskset -c 0-11 /usr/bin/perf stat -d -d -d ./ntt
NTT took 9.54296e+06 nanoseconds to execute.

 Performance counter stats for './ntt':

                 42.93 msec task-clock:u                     #      2.825 CPUs utilized
                    0      context-switches:u                #      0.000 /sec
                    0      cpu-migrations:u                  #      0.000 /sec
```

```
                   678        page-faults:u                                 #
15.793 K/sec
       <not counted>        cpu_atom/cycles/u
                                                  (0.00%)
           26,706,256        cpu_core/cycles/u                             #      0.622
GHz                                               (33.99%)
       <not counted>        cpu_atom/instructions/u
                                                  (0.00%)
           52,186,073        cpu_core/instructions/u
                                                  (73.79%)
       <not counted>        cpu_atom/branches/u
                                                  (0.00%)
           13,638,036        cpu_core/branches/u                           #  317.677
M/sec
       <not counted>        cpu_atom/branch-misses/u
                                                  (0.00%)
                61,533        cpu_core/branch-misses/u

                  TopdownL1 (cpu_core)                                 #       22.0 %
tma_backend_bound
                                                                       #        1.5
%  tma_bad_speculation
                                                                       #        4.0
%  tma_frontend_bound
                                                                       #       72.5
%  tma_retiring
       <not counted>        L1-dcache-loads:u
                                                  (0.00%)
         167,239,356        L1-dcache-loads:u                             #      3.896
G/sec
    <not supported>        L1-dcache-load-misses:u

               715,114        L1-dcache-load-misses:u

       <not counted>        LLC-loads:u
                                                  (0.00%)
                49,541        LLC-loads:u                                   #
1.154 M/sec
       <not counted>        LLC-load-misses:u
                                                  (0.00%)
                 3,493        LLC-load-misses:u

       <not counted>        L1-icache-loads:u
                                                  (0.00%)
    <not supported>        L1-icache-loads:u

       <not counted>        L1-icache-load-misses:u
                                                  (0.00%)
       <not counted>        L1-icache-load-misses:u
                                                  (0.00%)
```

```
      <not counted>      dTLB-loads:u
                                          (0.00%)
      <not counted>      dTLB-loads:u
                                          (0.00%)
      <not counted>      dTLB-load-misses:u
                                          (0.00%)
      <not counted>      dTLB-load-misses:u
                                          (0.00%)
   <not supported>      iTLB-loads:u

   <not supported>      iTLB-loads:u

      <not counted>      iTLB-load-misses:u
                                          (0.00%)
      <not counted>      iTLB-load-misses:u
                                          (0.00%)
   <not supported>      L1-dcache-prefetches:u

   <not supported>      L1-dcache-prefetches:u

   <not supported>      L1-dcache-prefetch-misses:u

   <not supported>      L1-dcache-prefetch-misses:u


        0.015198926 seconds time elapsed

        0.028163000 seconds user
        0.000000000 seconds sys
```

3.1.2.2 正确性

使用 2.3 中的代码测试正确性，替换 `ntt_thread` 函数为初步尝试的 `ntt` 函数，发现输出 error，说明该多线程 ntt 函数存在正确性问题：

3.1.2.3 分析与思考

通过运行并记录数据，我们发现这样更改确实能提高运行速度，但是这样更改无法通过正确性测验，ntt 运行的结果是错误的，也就是说将原函数的功能改错了，并且每次并行运算的结果都不一样，我们猜测应该是在更改中忽略了进程之间的相互关系，或者是数据之间还存在其他依赖性，因此我们决定继续研究其原函数的实现思路，以更改实现思路。

3.2 确定优化思路

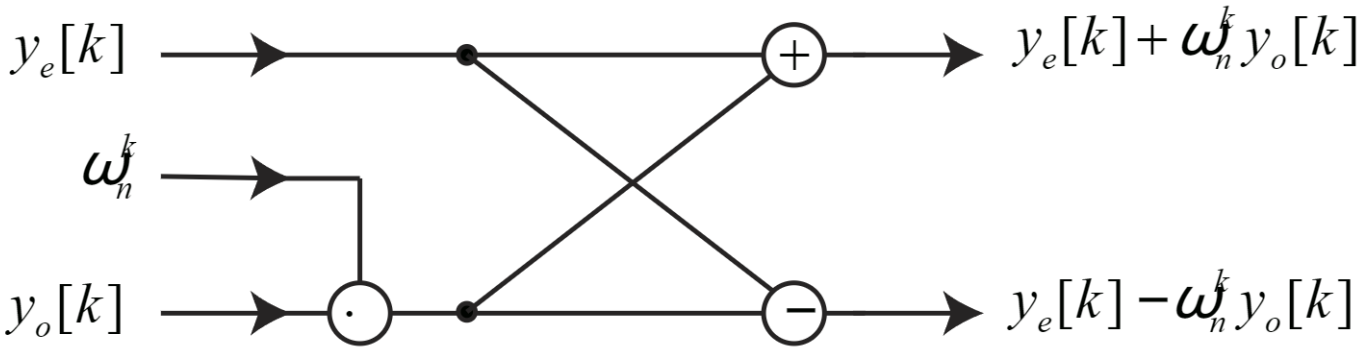
3.2.1 NTT 算法及函数代码分析

实验要求利用多线程设计实现 NTT 的加速，为实现多线程加速，我们要首先了解其基本原理。

由实验要求，我们要对原代码中的 ntt 函数进行优化。根据实验文件所提供的链接（https://zhuanlan.zhihu.com/p/80297169）以及查阅相关资料，我们得知原代码中的 ntt 函数对应 NTT 的迭代计算即蝴蝶操作部分。因为我们是对函数利用多线程设计加速，故重要的是理解其操作过程中的数据关系，研究其并行性。以下将详细介绍蝴蝶操作部分以及对应到代码上的函数中嵌套循环之间的数据计算关系。

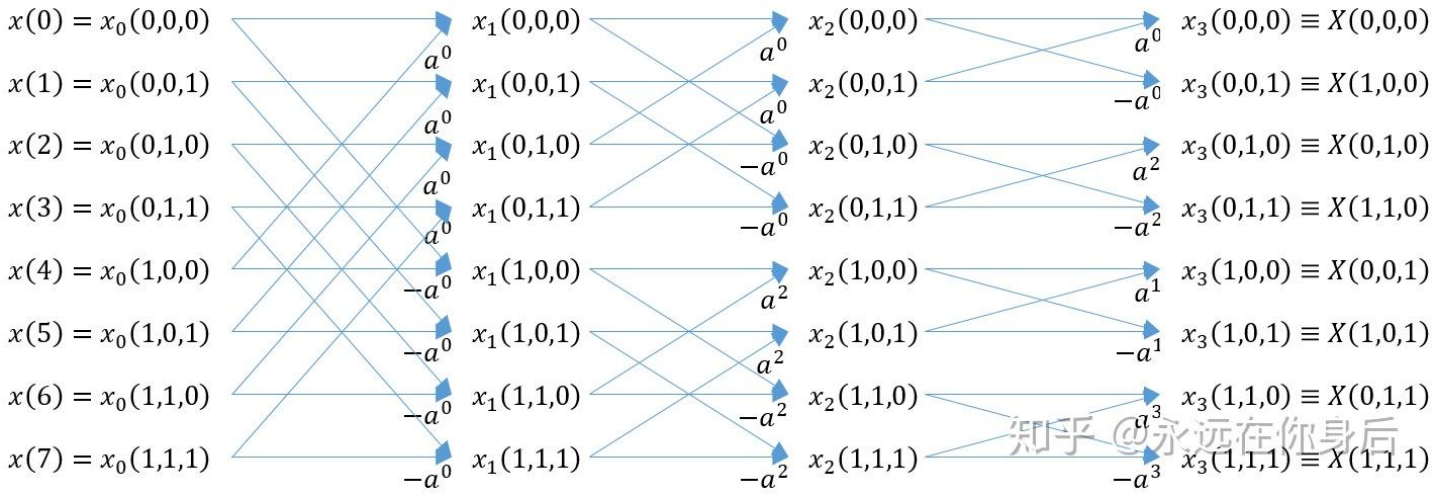
3.2.1.1 蝴蝶操作分析

蝴蝶操作得名于其数据在操作过程图中的形状，以下是 CT 蝶形操作的示意图：



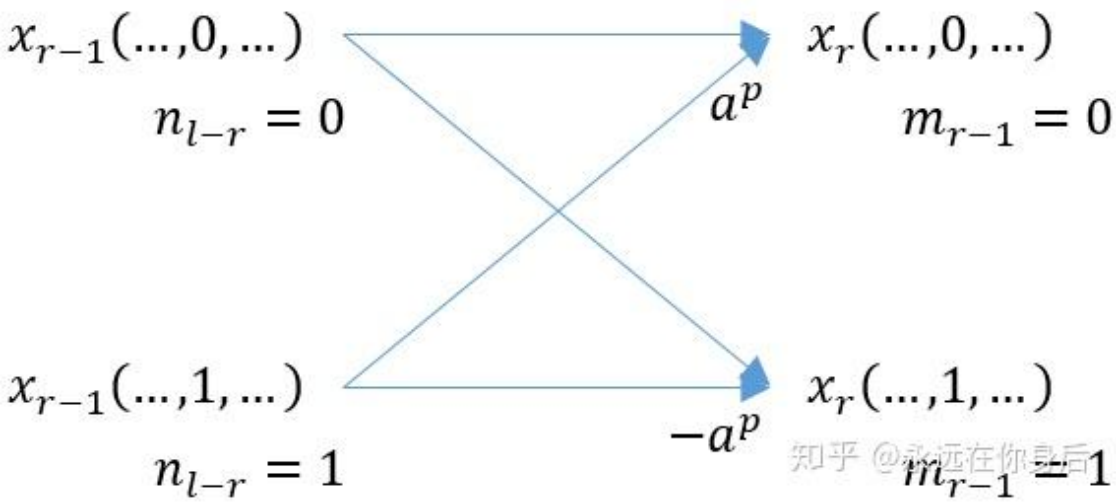
其数学原理不再在此赘述，详细可查看相关链接：https://zhuanlan.zhihu.com/p/80297169

而 FFT 迭代实现就是通过一个个蝴蝶操作实现的，其具体过程如下图所示，图中迭代次数为三次，只有八组数据：



因具体计算操作为数学原理与本实验无关，故在此不做介绍。

计算首先从最右端开始，记为第一轮运算，之后从右往左依次进行，左边每一列的运算都依赖于右边运算的结果。我们在计算每一列的时候，都是从上往下依次计算每组的蝴蝶操作，其中一组蝴蝶操作可以用下图表示：

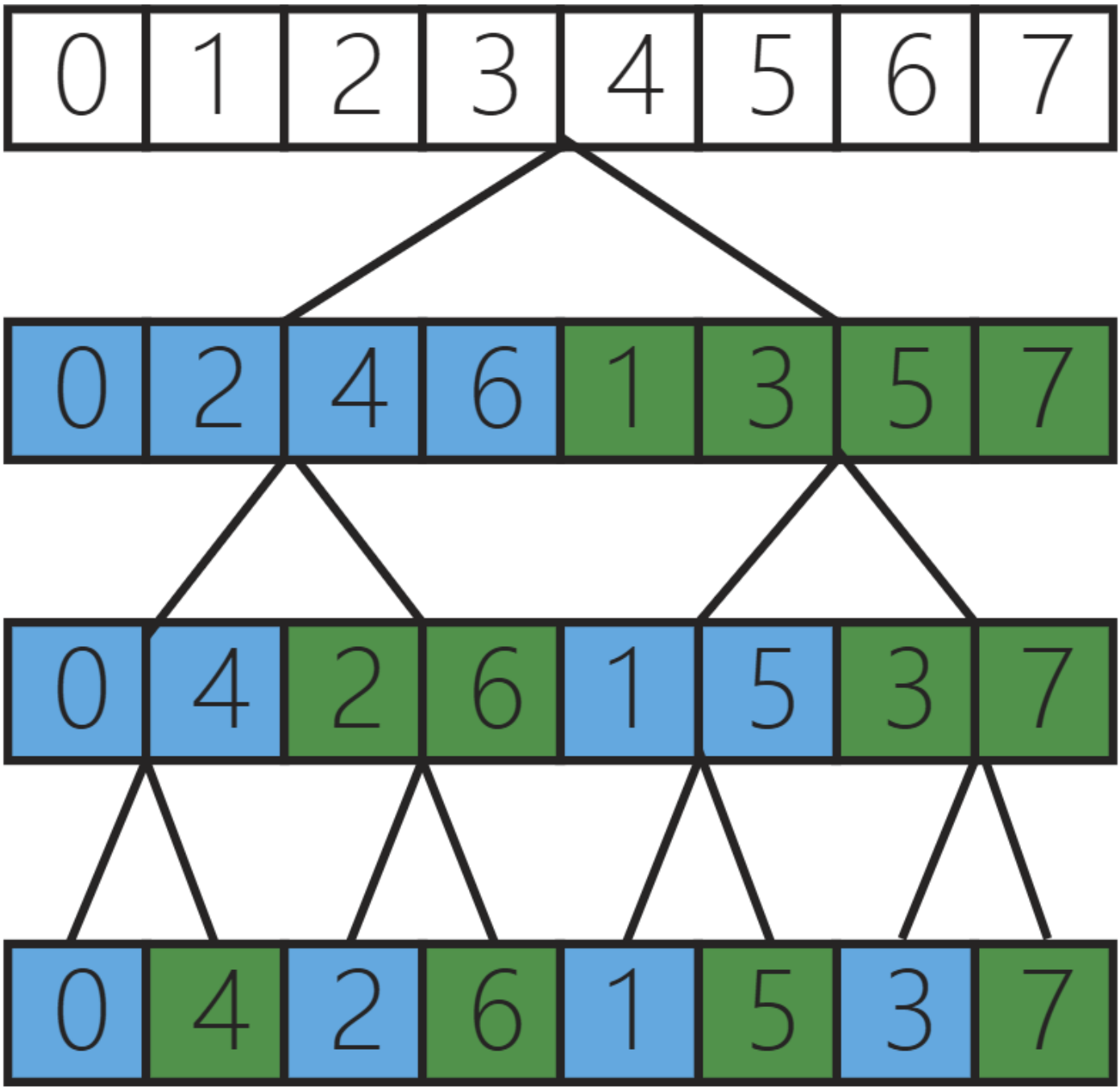


每一列中的蝴蝶操作依次进行，在这之中，数据之间没有依赖性，都是独立的运算。

由此我们便初步有了优化的思路，即将每一列的蝴蝶操作并行计算，这样从理论上可以提高运算速度，因为单线程运算时，是每个蝴蝶操作依次进行，但多线程可以同时计算多个操作，理论上同时计算的操作越多，速度越快（但这种想法也有局限性，后续会提到），但这种思路还需要对应到具体代码实现中去。

3.2.1.2 函数中数据计算关系分析

在已经了解到蝴蝶操作的基本原理之后，分析相关代码，研究其在函数中嵌套循环之间的数据计算关系。通过阅读代码可知，代码中 ntt 函数的实现原理与上述介绍大体相同，具体操作关系可以大体用下图表示（以8组数据进行举例）：



如图所示，函数代码先将数据通过比较排序一分为二分成两组，之后对这两组数据进行蝴蝶操作。由上述蝴蝶操作原理可知，对应到图中去蝴蝶操作是自底向上进行的（图中数字只代表数据名称，并不代表具体数据）。具体操作对应代码可分为三步，第一步是对具体每一组数据进行蝴蝶操作，第二步是每一层的所有组都进行蝴蝶操作，第三步是对自底向上每一层的数据都进行蝴蝶变化。其中每一步都依赖前一步的运算结果，我们通过对函数原代码的做注释以解释其具体过程，以下是具体注释：

```
1  void ntt2(int* x, int lim, int opt) // 定义一个名为ntt2的函数，接受一个整数数组x，一个限制值lim和一个选项opt
2  {
3        int i, j, k, m, gn, g, tmp; // 定义一些用于循环和计算的整数变量
4        for (i = 0; i < lim; ++i) // 遍历0到lim
5              if (r[i] < i) // 如果r数组的第i个元素小于i
6                    swap(x[i], x[r[i]]); // 交换x数组的第i个元素和第r[i]个元素，这是NTT的位逆序置换步骤
7        for (m = 2; m <= lim; m <<= 1) // 从2开始，每次将m左移一位（即乘以2），直到m大于lim，这是NTT的迭代过程，m是当前的DFT（离散傅里叶变换）长度，即对整体的操作
8       {
9              k = m >> 1; // 将m右移一位（即除以2），赋值给k，k是当前DFT长度的一半，也是蝴蝶操作的跨度
10             gn = qpow(3, (P - 1) / m); // 计算3的(P - 1) / m次方，结果赋值给gn，gn是本次DFT的主n次单位根
11             for (i = 0; i < lim; i += m) // 从0开始，每次增加m，直到i大于lim，这是对每个DFT进行操作的过程，对应图中每一层的变换操作
12            {
13                   g = 1; // 将g设置为1，g是当前的n次单位根的幂
14                   for (j = 0; j < k; j++, g = 1ll * g * gn % P) // 从0开始，每次增加1，直到j等于k，同时更新g的值，这是对DFT内部进行蝴蝶操作的过程，即每一组具体的蝴蝶操作
15                  {
16                         tmp = 1ll * x[i + j + k] * g % P; // 计算x[i + j + k] * g % P的结果，赋值给tmp，这是蝴蝶操作的一部分
17                         x[i + j + k] = (x[i + j] - tmp + P) % P; // 更新x[i + j + k]的值，这是蝴蝶操作的一部分
18                         x[i + j] = (x[i + j] + tmp) % P; // 更新x[i + j]的值，这是蝴蝶操作的一部分
19                  }
20            }
21       }
22 }
```

由于第一次排序数据有依赖关系，后边三步（对应三个 for 循环）中也有数据依赖关系，所以不能盲目进行多线程设计。

根据我们的优化思路对应到代码可知，我们要将函数中第三个 for 循环转换成多线程计算。

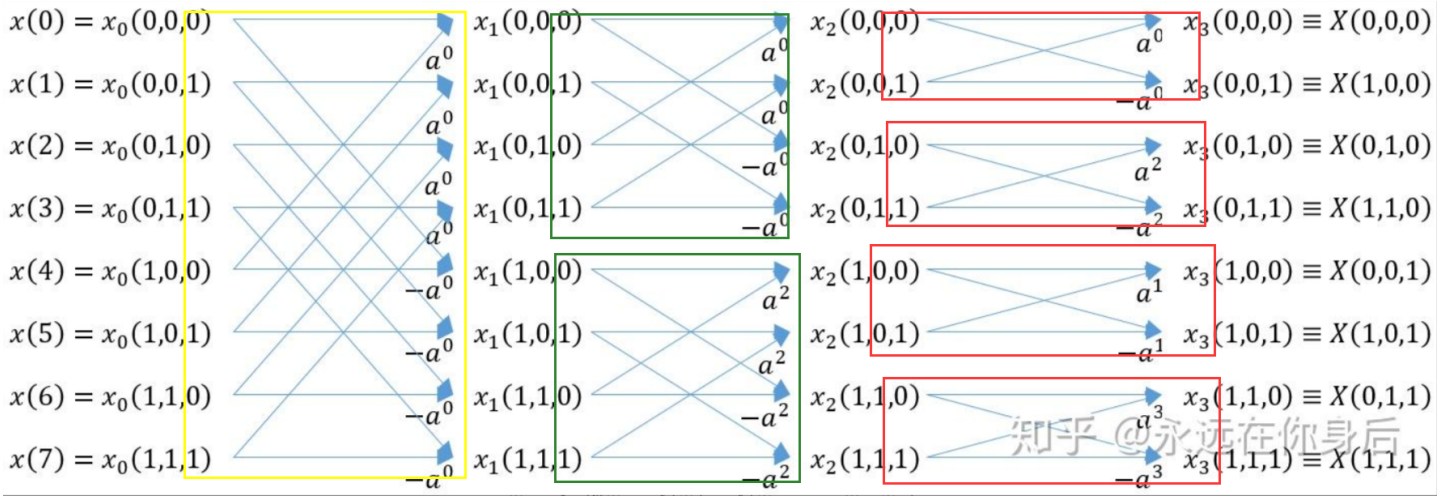
3.3 利用多线程优化

确定优化思路之后，我们决定继续利用 thread 实现多线程优化，在初步优化的基础之上进行改进。

因为最后一次 for 循环是对每一组进行蝴蝶操作，所以其无法再拆分并行计算，故将其放在并行计算之外；对于最开始的 for 循环，它是将数据比较排序一分为二分成两组，若并行计算会影响其结果，故将其放在并行计算之外；对于第二个 for 循环，它是有底向上进行遍历，每次循环都依赖之前的结果故无法进行拆分。所以最终实现方法就是将第三个循环进行合理的拆分以实现并行计算。

3.3.1 改进后的初步尝试

我们在确定优化思路之后再次做了初步尝试。在这次尝试的版本中，我们试图将每一层的每一组蝴蝶操作都并行计算，这样我们可以最大限度地将数据进行并行运算以提高性能，也就是说，每一层的线程数是由每一层有多少次蝴蝶操作决定的，即 num_threads = lim / m，其中 m 是每一层的每一组的长度（离散傅里叶变换长度），每到新的一层就计算一次所需要的线程数，然后建立相应大小的线程数组。具体实现示例如下图所示，其中一个方框为一个线程要处理的一组蝴蝶操作，相同颜色的方框个数即为一层的线程数：



在这个实现中，每个线程处理的长度就是每一组蝴蝶变换的长度，所以不会发生访问资源冲突以及数据有依赖性而被错误更改的问题，从理论上避免了因为并行计算导致结果不正确。具体实现代码如下：

```
void sub_ntt(int* x, int k, int g, int gn, int i) {
      int j, tmp;
      for (j = 0; j < k; j++, g = 1ll * g * gn % P) {
            tmp = 1ll * x[i + j + k] * g % P;
            x[i + j + k] = (x[i + j] - tmp + P) % P;
            x[i + j] = (x[i + j] + tmp) % P;
     }
}
// NTT变换

void ntt(int* x, int lim, int opt) {
```

```
    int i, j, k, m, gn, g, tmp;
    for (i = 0; i < lim; ++i) {
        if (r[i] < i) {
            swap(x[i], x[r[i]]);
        }
    }
    for (m = 2; m <= lim; m <<= 1) {
        const int num_threads = lim / m; // 定义线程数量
        k = m >> 1;
        gn = qpow(3, (P - 1) / m);
        vector<thread> threads(num_threads); // 定义线程数组
        int i = 0;
        for (int j = 0; j < num_threads; ++j) {
            g = 1;
            threads[j] = thread(sub_ntt, x, k, g, gn, i); // 创建线程并加入线程数组
            i += m;
        }
        for (int i = 0; i < num_threads; ++i) {
            threads[i].join();
        }
    }
}
```

3.3.2 改进后初步尝试中的问题与分析

3.3.2.1 性能

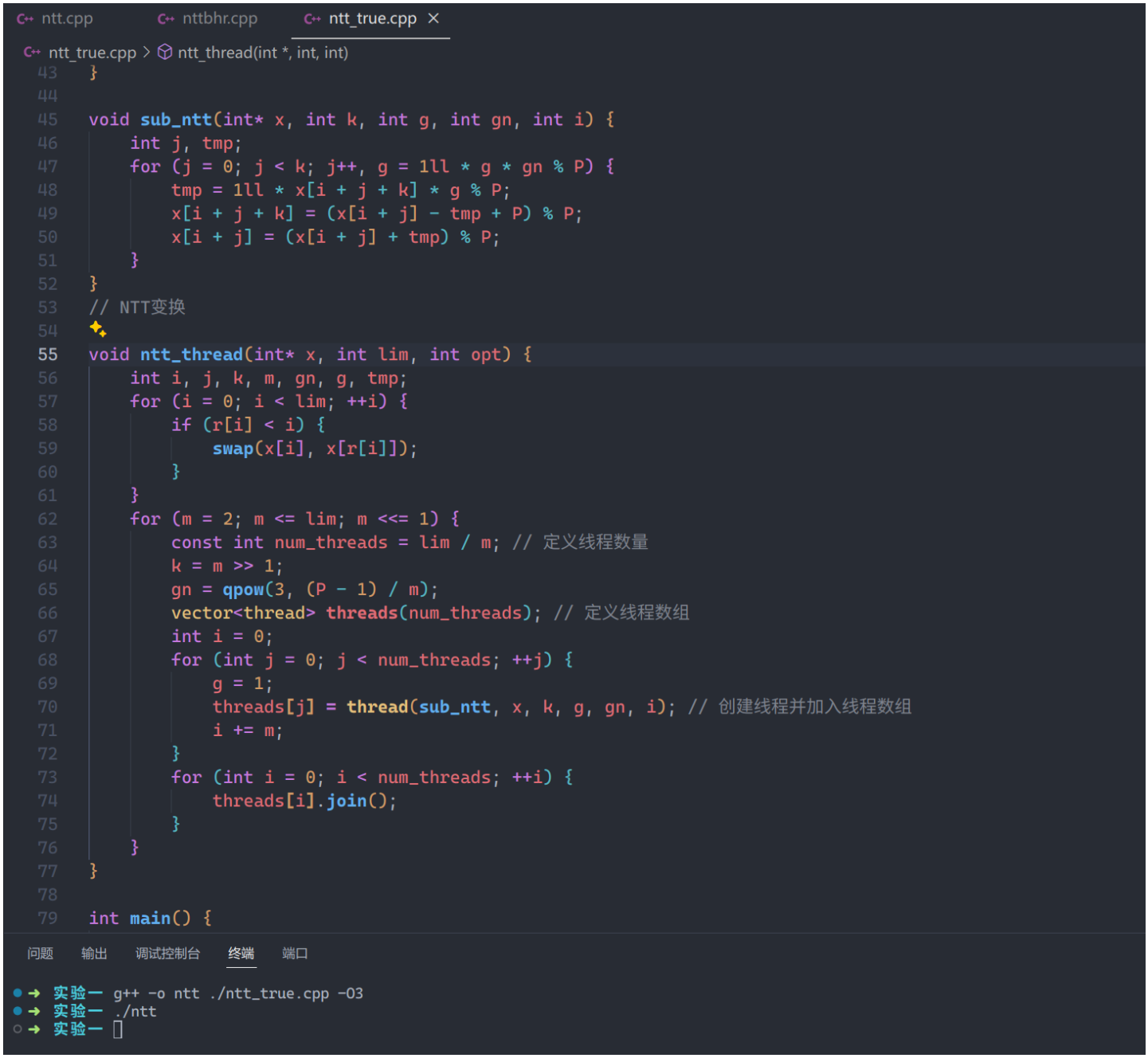
```
$ g++ -o ntt ./ntt.cpp -O3
$ taskset -c 0-11 /usr/bin/perf stat -d -d -d ./ntt
NTT took 1.75777e+10 nanoseconds to execute.

 Performance counter stats for './ntt':

            22,747.18 msec task-clock:u                              #      1.294 CPUs utilized
                     0      context-switches:u                       #      0.000 /sec
                     0      cpu-migrations:u                         #      0.000 /sec
             1,068,461      page-faults:u                            #     46.971 K/sec
       <not counted>      cpu_atom/cycles/u                                       (0.00%)
       3,288,133,442      cpu_core/cycles/u                        #      0.145 GHz                         (45.57%)
       <not counted>      cpu_atom/instructions/u                                 (0.00%)
       2,001,395,836      cpu_core/instructions/u                                 (52.16%)
```

```
       <not counted>      cpu_atom/branches/u                                         (0.00%)
         417,918,590      cpu_core/branches/u                   #   18.372 M/sec    (59.22%)
       <not counted>      cpu_atom/branch-misses/u                                    (0.00%)
             5,799,272    cpu_core/branch-misses/u                                    (66.23%)
                  TopdownL1 (cpu_core)                 #      42.5 %  tma_backend_bound
                                                       #       3.6 %  tma_bad_speculation
                                                       #      39.9 %  tma_frontend_bound
                                                       #      13.9 %  tma_retiring             (72.02%)
       <not counted>      L1-dcache-loads:u                                           (0.00%)
         504,888,573      L1-dcache-loads:u                   #   22.196 M/sec    (76.64%)
    <not supported>      L1-dcache-load-misses:u

            69,235,826    L1-dcache-load-misses:u                                     (75.51%)
       <not counted>      LLC-loads:u                                                 (0.00%)
            10,546,989    LLC-loads:u                         #  463.661 K/sec    (75.38%)
       <not counted>      LLC-load-misses:u                                           (0.00%)
             1,852,643    LLC-load-misses:u                                           (74.58%)
       <not counted>      L1-icache-loads:u                                           (0.00%)
    <not supported>      L1-icache-loads:u

       <not counted>      L1-icache-load-misses:u                                     (0.00%)
         225,819,118      L1-icache-load-misses:u                                     (33.85%)
       <not counted>      dTLB-loads:u                                                (0.00%)
         438,315,867      dTLB-loads:u                        #   19.269 M/sec    (35.99%)
       <not counted>      dTLB-load-misses:u                                          (0.00%)
             8,800,454    dTLB-load-misses:u                                          (36.64%)
    <not supported>      iTLB-loads:u
```

```
    <not supported>      iTLB-loads:u

       <not counted>      iTLB-load-misses:u
                                    (0.00%)
           9,371,312      iTLB-load-misses:u
                                      (37.95%)
    <not supported>      L1-dcache-prefetches:u

    <not supported>      L1-dcache-prefetches:u

    <not supported>      L1-dcache-prefetch-misses:u

    <not supported>      L1-dcache-prefetch-misses:u


        17.583227238 seconds time elapsed

         0.683600000 seconds user
        16.767055000 seconds sys
```

可以看到该 NTT 函数执行时间较长（17s），并且绝大多数时间开销都属于 sys ，这里即为创建线程与上下文切换的开销。

3.3.2.2 正确性



可以看到该 NTT 不存在正确性问题，没有输出 error。

3.3.2.3 分析与思考

在运行后我们发现，这样修改之后运行时间会变得非常长，但是结果正确性没有问题，也就是说理论上这样确实不会改变 ntt 优化后的结果，只是在时间开销反而会更大了。

于是我们尝试寻找原因，当我们把数组长度减小后，我们发现运行时间会显著减小，也就是说运行时间增大和程序的计算量有关，但当数组长度增大时，运行时间会呈指数级增长，这意味着数组长度越长对时间的影响也就越大。

查阅相关知识得知，每个线程的建立都会有时间开销，而且线程数与处理器核心数相关。这意味着我们即使开出 1000 个线程，但实际最多也就只有处理器相应核心数在运行，其余线程也需要等待处理器运行完任务才能运行，而且在每一层的蝴蝶操作中建立线程的数目太多会造成巨大的开销，严重影响性能。根据调试我们发现，lim=2^14，也就是说在第一层循环中，我们要进行 num_threads = lim / m=2^13 次线程的建立，下一层又会有 2^12 次线程的建立，一直到最后边几层才会减少，这造成了巨大的开销，也是为什么数组长度增大时运行时间会呈指数级增长。因此我们需要根据实际情况降低线程数以避免此种情况的发生，并且还要做到不影响数据之间的并行计算。

3.3.3 成功的多线程优化

在上述改进后初步尝试的分析中，我们已经了解到了问题在于线程数巨大所带来的开销，但是这样运行后结果是正确的。所以在此基础之上，我们再次对思路进行改进。在之前的尝试中，我们是核心数由每一层蝴蝶变化的组数确定，这样会造成底层蝴蝶变换的线程数太大。但是我们发现，每一层的蝴蝶操作组数都是 2^n 个，因为每进行下一层操作，组长变长一倍，所以我们得出新的设计思路。

因为目前的处理器，基本上都可以处理 8 线程及以上，所以当一层的蝴蝶操作组数大于等于 8 时，我们固定线程数为 8，每一个线程处理的数据长度为 lim / num_threads（num_threads=8），然后每个线程进行相应范围内的（lim / num_threads）*m 组蝴蝶操作；当一层的蝴蝶操作组数小于 8 时，线程数由该层的蝴蝶操作组数决定，即同之前所述。

这样设计之后，因为每一个线程处理的长度大于等于每一组蝴蝶操作的长度，且可以均分一层数据，所以每一个线程都正好处理完一组蝴蝶操作后结束，不会造成资源冲突以及数据有依赖性而被错误更改的问题，也就是可以保证结果的正确性，同时在当一层的蝴蝶操作组数大于等于 8 时，我们固定了线程数目的大小，大大降低了线程数建立所带来的开销，理论上可以解决之前出现的问题。具体实现代码如下：

```
void sub_ntt(int* x, int k, int gn, int start, int end, int m) {
      int j, tmp, i, g;
      for (i = start; i < end; i += m)
     {
            g = 1;
            //lll的目的是防止乘法的数据溢出
            for (j = 0; j < k; j++, g = lll * g * gn % P)
           {
                  //mtx.lock();
                  tmp = lll * x[i + j + k] * g % P;
                  x[i + j + k] = (x[i + j] - tmp + P) % P;
                  x[i + j] = (x[i + j] + tmp) % P;
                  //mtx.unlock();
           }
     }
}
void ntt(int* x, int lim, int opt) {
      int i, j, k, m, gn, g, tmp;
      for (i = 0; i < lim; ++i) {
            if (r[i] < i) {
```

```
                swap(x[i], x[r[i]]);
            }
        }
        //lim=2^14
        vector<thread> threads(8); // 定义线程数组
        for (m = 2; m <= lim; m <<= 1) {
            int num_threads = lim / m; // 计算一层的蝴蝶操作组数
            if (num_threads >= 8) {
                num_threads = 8; // 当一层的蝴蝶操作组数大于等于 8 时，线程数为 8
            }
            else {
                num_threads = num_threads; // 当一层的蝴蝶操作组数小于 8 时，线程数等于该层的蝴蝶操作组数
            }
            k = m >> 1;
            gn = qpow(3, (P - 1) / m);
            int step = lim / num_threads; // 计算线程处理的数据长度
            for (int i = 0; i < num_threads; i++) {
                int start = i * step;
                int end = (i == num_threads - 1) ? lim : (i + 1) * step;
                threads[i] = thread(sub_ntt, x,k,gn,start, end, m); // 创建线程并加入线程数组
                    }
            for (int i = 0; i < num_threads; ++i) {
                threads[i].join();
            }
        }
    }
```

在这个代码中，我们首先进行线程数与每一层蝴蝶操作的组数比较，以确定线程数，之后根据线程数计算处理的数据长度，然后创建线程进行相应的并行计算。

3.3.4 优化结果与分析

3.3.4.1 性能

```
$ g++ -o ntt ./ntt.cpp -O3
$ taskset -c 0-7 /usr/bin/perf stat -d -d -d ./ntt
NTT took 1.41858e+07 nanoseconds to execute.

 Performance counter stats for './ntt':

                38.81 msec task-clock:u                     #    1.945 CPUs utilized
                   0      context-switches:u               #    0.000 /sec
                   0      cpu-migrations:u                 #    0.000 /sec
               1,174      page-faults:u                    #   30.249 K/sec
```

```
       <not counted>      cpu_atom/cycles/u                                             (0.00%)
            40,656,236      cpu_core/cycles/u                       #      1.048 GHz                         (8.85%)
       <not counted>      cpu_atom/instructions/u                                       (0.00%)
            76,544,814      cpu_core/instructions/u                                       (28.00%)
       <not counted>      cpu_atom/branches/u                                           (0.00%)
            11,062,708      cpu_core/branches/u                     #  285.041 M/sec                       (84.54%)
       <not counted>      cpu_atom/branch-misses/u                                      (0.00%)
                44,290      cpu_core/branch-misses/u                                      (99.70%)
                 TopdownL1 (cpu_core)                 #       54.5 %  tma_backend_bound
                                                      #        0.7 %  tma_bad_speculation
                                                      #        2.7 %  tma_frontend_bound
                                                      #       42.1 %  tma_retiring
       <not counted>      L1-dcache-loads:u                                             (0.00%)
            21,454,653      L1-dcache-loads:u                       #  552.799 M/sec
   <not supported>      L1-dcache-load-misses:u

               471,023      L1-dcache-load-misses:u

       <not counted>      LLC-loads:u                                                   (0.00%)
                25,922      LLC-loads:u                             #  667.904 K/sec
       <not counted>      LLC-load-misses:u                                             (0.00%)
                 4,897      LLC-load-misses:u

       <not counted>      L1-icache-loads:u                                             (0.00%)
   <not supported>      L1-icache-loads:u

       <not counted>      L1-icache-load-misses:u                                       (0.00%)
       <not counted>      L1-icache-load-misses:u                                       (0.00%)
       <not counted>      dTLB-loads:u                                                  (0.00%)
```

```
36          <not counted>      dTLB-loads:u
                                                  (0.00%)
37          <not counted>      dTLB-load-misses:u
                                            (0.00%)
38          <not counted>      dTLB-load-misses:u
                                            (0.00%)
39      <not supported>      iTLB-loads:u

40      <not supported>      iTLB-loads:u

41          <not counted>      iTLB-load-misses:u
                                            (0.00%)
42          <not counted>      iTLB-load-misses:u
                                            (0.00%)
43      <not supported>      L1-dcache-prefetches:u

44      <not supported>      L1-dcache-prefetches:u

45      <not supported>      L1-dcache-prefetch-misses:u

46      <not supported>      L1-dcache-prefetch-misses:u


47
48            0.019952322 seconds time elapsed
49
50            0.013357000 seconds user
51            0.005725000 seconds sys
```

可见该函数实现了 1.945 CPUs 的利用率，运行时间为 14ms 。

3.3.4.2 正确性

ntt.cpp ntt_pool.cpp nttbhr.cpp ntt_true.cpp ×

ntt_true.cpp > main()

```cpp
void ntt_thread(int* x, int lim, int opt) {
        if (num_threads >= 8) {
            num_threads = 8; // 当一层的蝴蝶操作组数大于等于 8 时，线程数为 8
        }
        else {
            num_threads = num_threads; // 当一层的蝴蝶操作组数小于 8 时，线程数等于该层的蝴蝶操作组数
        }
        k = m >> 1;
        gn = qpow(3, (P - 1) / m);
        int step = lim / num_threads; // 计算线程处理的数据长度
        for (int i = 0; i < num_threads; i++) {
            int start = i * step;
            int end = (i == num_threads - 1) ? lim : (i + 1) * step;
            threads[i] = thread(sub_ntt, x,k,gn,start, end, m); // 创建线程并加入线程数组
                }
        for (int i = 0; i < num_threads; ++i) {
            threads[i].join();
        }
    }
}

int main() {
    srand(time(nullptr));
    int i, lim = 1, n = N / 2;
    for (i = 0; i < n; i++) {
        A[i] = rand() % 10;
        B[i] = A[i]; // 将 A 和 B 初始化为相同的数组
    }
    // 计算适当的 lim 值
    while (lim < n) lim <<= 1;
    // 初始化 r 数组
    for (i = 0; i < lim; ++i) {
        r[i] = (i & 1) * (lim >> 1) + (r[i >> 1] >> 1);
    }
    // 对 A 和 B 进行 NTT
    ntt(A, lim, 1);
    ntt_thread(B, lim, 1);
```

问题 输出 调试控制台 终端 端口

```
➜ 实验一 g++ -o ntt ./ntt_true.cpp
➜ 实验一 ./ntt
➜ 实验一
```

可见该 NTT 函数是正确的。

3.3.4.3 分析与思考

运行之后我们发现，这样优化性能会有所提升，且正确性不会改变，实现成功的多线程优化。

虽然我们已经成功实现，但是在每一层的蝴蝶操作中，我们还是会不能避免得进行一些线程的建立，这仍会造成一定的开销。在本次实验中，会有 10 层的蝴蝶操作中每层会有 8 次线程的建立操作，且要对两个数组进行操作。我们有没有一种能够不用每次都建立线程，即建立一次便可循环使用该线程的方法呢？

经过查阅相关资料得知，利用线程池进行多线程优化可以完成这一目的。于是我们尝试利用线程池进行进一步优化。

3.4 利用线程池优化

3.4.1 线程池优化的初步尝试

通过学习相关知识我们了解到，线程池可以解决线程创建与销毁的开销以及线程竞争造成的性能瓶颈。通过预先创建一组线程并复用它们，线程池有效地降低了线程创建和销毁的时间和资源消耗。同时，通过管理线程并发数量，线程池有助于减少线程之间的竞争，增加资源利用率，并提高程序运行的性能。

线程池通过预先创建和调度复用线程来实现资源优化。这个过程主要包括：创建线程、任务队列与调度、以及线程执行及回收。

在了解到相关知识后，我们初步建立了线程池进行优化，线程池相关代码如下：

```
class ThreadPool {
public:
      ThreadPool(size_t numThreads) {
            for (size_t i = 0; i < numThreads; ++i) {
                  workers.emplace_back([this] {
                        while (true) {
                              function<void()> task;
                              {
                                    unique_lock<mutex> lock(this->m);
                                    this->cv.wait(lock, [this] { return this->stop || !this->tasks.empty(); });
                                    if (this->stop && this->tasks.empty()) return;
                                    task = move(this->tasks.front());
                                    this->tasks.pop();
                              }
                              task();
                        }
                        });
            }
      }

      ~ThreadPool() {
            {
                  unique_lock<mutex> lock(m);
                  stop = true;
            }
            cv.notify_all();
            for (thread& worker : workers) {
                  worker.join();
            }
      }

      template<class F>
      void enqueue(F&& f) {
            {
```

```
                unique_lock<mutex> lock(m);
                tasks.emplace(forward<F>(f));
            }
            cv.notify_one();
        }

private:
        vector<thread> workers;
        queue<function<void()>> tasks;
        mutex m;
        condition_variable cv;
        atomic<bool> stop = false;
};
```

这个线程池实现了创建线程、任务队列与调度、以及线程执行及回收等线程池最基本的功能。之后我们利用该线程池对 ntt 函数进行优化，优化思路与之前相同，只不过在实现方式上，我们先建立一个线程池，再调度线程池里的线程进行并行计算。具体实现代码如下：

```
void sub_ntt(int* x, int k, int gn, int start, int end, int m) {
        int j, tmp, i, g;
        for (i = start; i < end; i += m)
        {
            g = 1;
            //lll的目的是防止乘法的数据溢出
            for (j = 0; j < k; j++, g = lll * g * gn % P)
            {
                //mtx.lock();
                tmp = lll * x[i + j + k] * g % P;
                x[i + j + k] = (x[i + j] - tmp + P) % P;
                x[i + j] = (x[i + j] + tmp) % P;
                //mtx.unlock();
            }
        }
}
ThreadPool pool(8); // 创建线程池
void ntt6(int* x, int lim, int opt) {
        int i, k, m, gn, num_threads;
        for (i = 0; i < lim; ++i) {
            if (r[i] < i) {
                swap(x[i], x[r[i]]);
            }
        }
        //lim=2^14
        for (m = 2; m <= lim; m <<= 1) {
            num_threads = lim / m; // 定义线程数量
            k = m >> 1;
            gn = qpow(3, (P - 1) / m);
            int step = lim / num_threads;
            for (int i = 0; i < num_threads; i++) {
```

```
                int start = i * step;
                int end = (i == num_threads - 1) ? lim : (i + 1) * step;
                pool.enqueue(sub_ntt, x, k, gn, start, end, m); // 将任务添加到线程池
            }
        }
    }
```

在这个代码中，我们创建了一个 8 线程的线程池，之后利用这些线程依次进行每一组的蝴蝶操作，思路同之前的优化思路。

3.4.2 线程池初步尝试中的问题与分析

3.4.2.1 性能

```
$ g++ -o ntt ./ntt_pool.cpp -O3
$ taskset -c 0-11 /usr/bin/perf stat -d -d -d ./ntt


NTT took 4.77641e+08 nanoseconds to execute.

 Performance counter stats for './ntt':

              2,476.91 msec task-clock:u                     #      5.125 CPUs utilized
                     0        context-switches:u             #      0.000 /sec
                     0        cpu-migrations:u               #      0.000 /sec
                   756        page-faults:u                  #    305.219 /sec
       <not counted>        cpu_atom/cycles/u                                             (0.00%)
       1,538,088,947        cpu_core/cycles/u              #      0.621 GHz                (34.71%)
       <not counted>        cpu_atom/instructions/u                                       (0.00%)
         791,960,791        cpu_core/instructions/u                                       (42.12%)
       <not counted>        cpu_atom/branches/u                                           (0.00%)
         149,209,395        cpu_core/branches/u            #     60.240 M/sec            (49.17%)
       <not counted>        cpu_atom/branch-misses/u                                      (0.00%)
           2,318,898        cpu_core/branch-misses/u                                      (56.32%)
                   TopdownL1 (cpu_core)                   #       49.8 % tma_backend_bound
```

```
                                                                  #       10.1
%  tma_bad_speculation
                                                                  #       21.1
%  tma_frontend_bound
                                                                  #       19.0
%  tma_retiring                  (63.54%)
       <not counted>      L1-dcache-loads:u
                                                 (0.00%)
         187,869,015      L1-dcache-loads:u                       #   75.848
M/sec                            (69.46%)
    <not supported>       L1-dcache-load-misses:u

          12,415,826      L1-dcache-load-misses:u
                                   (71.33%)
       <not counted>      LLC-loads:u
                                    (0.00%)
           7,104,503      LLC-loads:u                             #    2.868
M/sec                            (72.63%)
       <not counted>      LLC-load-misses:u
                                    (0.00%)
                7,396       LLC-load-misses:u
                                    (73.44%)
       <not counted>      L1-icache-loads:u
                                    (0.00%)
    <not supported>       L1-icache-loads:u

       <not counted>      L1-icache-load-misses:u
                                    (0.00%)
          26,130,333      L1-icache-load-misses:u
                                   (30.75%)
       <not counted>      dTLB-loads:u
                                     (0.00%)
         157,380,314      dTLB-loads:u                            #   63.539
M/sec                            (32.05%)
       <not counted>      dTLB-load-misses:u
                                     (0.00%)
                8,466       dTLB-load-misses:u
                                     (30.52%)
    <not supported>       iTLB-loads:u

    <not supported>       iTLB-loads:u

       <not counted>      iTLB-load-misses:u
                                     (0.00%)
              847,917      iTLB-load-misses:u
                                     (29.15%)
    <not supported>       L1-dcache-prefetches:u

    <not supported>       L1-dcache-prefetches:u
```

```
   <not supported>        L1-dcache-prefetch-misses:u

   <not supported>        L1-dcache-prefetch-misses:u


         0.483319638 seconds time elapsed

         0.556107000 seconds user
         1.951112000 seconds sys
```

可见该函数实现了 5.125 CPUs 的利用率，运行时间为 477ms 。

3.4.2.2 正确性

ntt.cpp　ntt_pool.cpp　nttbhr.cpp　ntt_true.cpp ×

ntt_true.cpp > ntt6(int *, int, int)

```cpp
void ntt6(int* x, int lim, int opt) {
    ThreadPool pool(8); // 创建线程池
    int i, k, m, gn, num_threads;
    for (i = 0; i < lim; ++i) {
        if (r[i] < i) {
            swap(x[i], x[r[i]]);
        }
    }
    //lim=2^14
    for (m = 2; m <= lim; m <<= 1) {
        num_threads = lim / m; // 定义线程数量
        k = m >> 1;
        gn = qpow(3, (P - 1) / m);
        int step = lim / num_threads;
        for (int i = 0; i < num_threads; i++) {
            int start = i * step;
            int end = (i == num_threads - 1) ? lim : (i + 1) * step;
            pool.enqueue(sub_ntt, x, k, gn, start, end, m); // 将任务添加到线程池
        }
    }
}

int main() {
    srand(time(nullptr));
    int i, lim = 1, n = N / 2;
    for (i = 0; i < n; i++) {
        A[i] = rand() % 10;
        B[i] = A[i]; // 将 A 和 B 初始化为相同的数组
    }
    // 计算适当的 lim 值
    while (lim < n) lim <<= 1;
    // 初始化 r 数组
    for (i = 0; i < lim; ++i) {
        r[i] = (i & 1) * (lim >> 1) + (r[i >> 1] >> 1);
    }
    // 对 A 和 B 进行 NTT
    ntt(A, lim, 1);
    ntt6(B, lim, 1);
```

问题　输出　调试控制台　终端　端口

```
➜ 实验一 g++ -o ntt ./ntt_true.cpp -O3
➜ 实验一 ./ntt
error
➜ 实验一
```

输出 error ，可见该 NTT 是错误的

3.4.2.3 分析与思考

通过运行我们发现，这样修改之后运行速度有了提升，但是结果却是错的，这说明利用线程池优化确实能够提高程序运行的速度，里边进行了并行计算，但是可能更改了运算顺序或者由于数据之间的依赖性而导致结果错误。

经过分析我们猜测，由于线程执行任务需要时间，在没有线程空闲时，后边一层的蝴蝶操作此时处于任务队列中；而一旦有线程执行完任务释放出来，则会立即从任务队列中读取任务再次执行，此时当前这一层的其他蝴蝶操作还没有完成，但是已经开始执行下一层蝴蝶操作的运算了，由于在蝴蝶操作中后一层的运算依赖于前一层运算的结果，所以此时会发生冲突，导致结果错误。所以我们需要修改线程池读取任务的机制，使其在一层运算操作完成后，再进行下一列的操作。

3.4.3 成功的线程池优化

我们于之前的初步尝试中发现了问题，并尝试在其基础上加以改进。我们需要解决线程池读取任务队列的问题，使其能够实现当一层的运算全部完成之后，再进行下一层的操作。

通过思考，我们决定对每个线程设置一个状态量，这个状态量表示此线程是否处于执行任务的状态，当任务完成时此状态设置为 false，运行时设置为 true。之后再定义一个检测线程池中线程状态的函数，该函数检测所有线程的运行状态，当所有线程都处于空闲状态时变返回 false，表示可以进行下一层的操作。同时，每个线程运行完自己的任务之后，设置为暂停状态，防止从任务队列中继续读取任务。

该线程池的实现代码如下：

```
class ThreadPool {
public:
      ThreadPool(size_t numThreads) : stop(false), pause(false) {
            // 创建指定数量的线程
            for (size_t i = 0; i < numThreads; ++i) {
                  threads.emplace_back([this] {
                        // 每个线程都会执行这个无限循环
                        while (true) {
                              std::function<void()> task;
                              {
                                    // 获取互斥锁，等待条件变量
                                    std::unique_lock<std::mutex> lock(queueMutex);
                                    condition.wait(lock, [this] { return stop ||
(!tasks.empty() && !pause); });
                                    // 如果线程池停止且任务队列为空，则退出循环
                                    if (stop && tasks.empty()) {
                                          return;
                                    }
                                    // 从任务队列中取出一个任务
                                    task = std::move(tasks.front());
                                    tasks.pop();
                              }
                              // 更新线程状态为正在运行
```

```
                        {
                                std::unique_lock<std::mutex> lock(statusMutex);
                                threadStatus[std::this_thread::get_id()] = true;
                        }
                        // 执行任务
                        task();
                        // 更新线程状态为非运行
                        {
                                std::unique_lock<std::mutex> lock(statusMutex);
                                threadStatus[std::this_thread::get_id()] = false;
                        }
                        // 如果任务队列为空且线程池处于暂停状态，则等待
                        std::unique_lock<std::mutex> lock(queueMutex);
                        if (tasks.empty() && pause) {
                                condition.wait(lock, [this] { return stop || !pause; });
                        }
                }
                });
        }
    }
    ~ThreadPool() {
        // 停止线程池
        {
            std::unique_lock<std::mutex> lock(queueMutex);
            stop = true;
        }
        // 唤醒所有线程
        condition.notify_all();
        // 等待所有线程结束
        for (auto& thread : threads) {
            thread.join();
        }
    }
    // 添加任务到线程池
    template <class F, class... Args>
    void enqueue(F&& f, Args&&... args) {
        auto task = std::bind(std::forward<F>(f), std::forward<Args>(args)...);
        {
            std::unique_lock<std::mutex> lock(queueMutex);
            tasks.emplace(std::move(task));
        }
        // 唤醒一个等待的线程
        condition.notify_one();
    }
    // 设置线程池的暂停状态
    void setPause(bool value) {
        std::unique_lock<std::mutex> lock(queueMutex);
        pause = value;
        if (!pause) {
            // 如果暂停标志被设置为false，唤醒所有等待的线程
```

```
                condition.notify_all();
            }
        }
        // 检查是否有线程正在运行
        bool isAnyThreadRunning() {
            std::unique_lock<std::mutex> lock(statusMutex);
            for (auto& status : threadStatus) {
                if (status.second) {
                    return true; // 如果有线程正在运行，返回true
                }
            }
            return false; // 所有线程都不在运行，返回false
        }

private:
        std::vector<std::thread> threads; // 线程池中的线程
        std::queue<std::function<void()>> tasks; // 任务队列
        std::mutex queueMutex; // 保护任务队列的互斥锁
        std::condition_variable condition; // 条件变量
        bool stop; // 线程池停止标志
        bool pause; // 线程池暂停标志
        std::unordered_map<std::thread::id, bool> threadStatus; // 线程状态
        std::mutex statusMutex; // 保护线程状态的互斥锁
};
```

通过修改线程池，理论上实现了我们对于线程池读取任务队列操作的要求。

相应的，我们对于 ntt 函数的多线程实现也需要修改。我们创立线程池之后，与之前同样的操作，将任务分割后分配给线程池中的线程执行，不同的是，在执行每一层的蝴蝶操作之前，我们都需要对线程池中的线程进行检测，如果它们都处于空闲状态，那么便可以继续该层的操作，如果不是，那么便须等待线程池中的线程都执行完任务，才可进行下一步操作。同时，为了避免因为每一层的蝴蝶操作组数不同导致的线程数的动态变化问题，在当线程数小于 8 时，我们依然通过建立线程数组的方式实现多线程运行。具体实现代码如下：

```
void sub_ntt(int* x, int k, int gn, int start, int end, int m) {
        int j, tmp, i, g;
        for (i = start; i < end; i += m)
        {
            g = 1;
            //lll的目的是防止乘法的数据溢出
            for (j = 0; j < k; j++, g = lll * g * gn % P)
            {
                //mtx.lock();
                tmp = lll * x[i + j + k] * g % P;
                x[i + j + k] = (x[i + j] - tmp + P) % P;
                x[i + j] = (x[i + j] + tmp) % P;
                //mtx.unlock();
            }
        }
```

```
}
ThreadPool pool(8); // 创建线程池
void ntt6(int* x, int lim, int opt) {
      int i, k, m, gn,num_threads;
      for (i = 0; i < lim; ++i) {
            if (r[i] < i) {
                  swap(x[i], x[r[i]]);
            }
      }
      //lim=2^14
      for (m = 2; m <= lim; m <<= 1) {
            num_threads = lim / m; // 定义线程数量
            if (num_threads >= 8) {
                  num_threads = 8;
                  k = m >> 1;
                  gn = qpow(3, (P - 1) / m);
                  int step = lim / num_threads;
                  while (pool.isAnyThreadRunning())
                  {
                        this_thread::sleep_for(chrono::nanoseconds(1)); // 暂停1纳秒
                  }
                  for (int i = 0; i < num_threads; i++) {
                        int start = i * step;
                        int end = (i == num_threads - 1) ? lim : (i + 1) * step;
                        pool.enqueue(sub_ntt, x, k, gn, start, end, m); // 将任务添加到线程池
                  }
            }
            else {
                  num_threads = num_threads;
                  k = m >> 1;
                  gn = qpow(3, (P - 1) / m);
                  vector<thread> threads(num_threads); // 定义线程数组
                  int step = lim / num_threads;
                  for (int i = 0; i < num_threads; i++) {
                        int start = i * step;
                        int end = (i == num_threads - 1) ? lim : (i + 1) * step;
                        threads[i] = thread(sub_ntt, x, k, gn, start, end, m); // 创建线程并加入线程数组
                  }
                  for (int i = 0; i < num_threads; ++i) {
                        threads[i].join();
                  }
            }
      }
}
```

理论上，这种修改会解决之前出现的问题，在保证性能优化的同时不改变结果正确性。

3.4.4 优化结果与分析

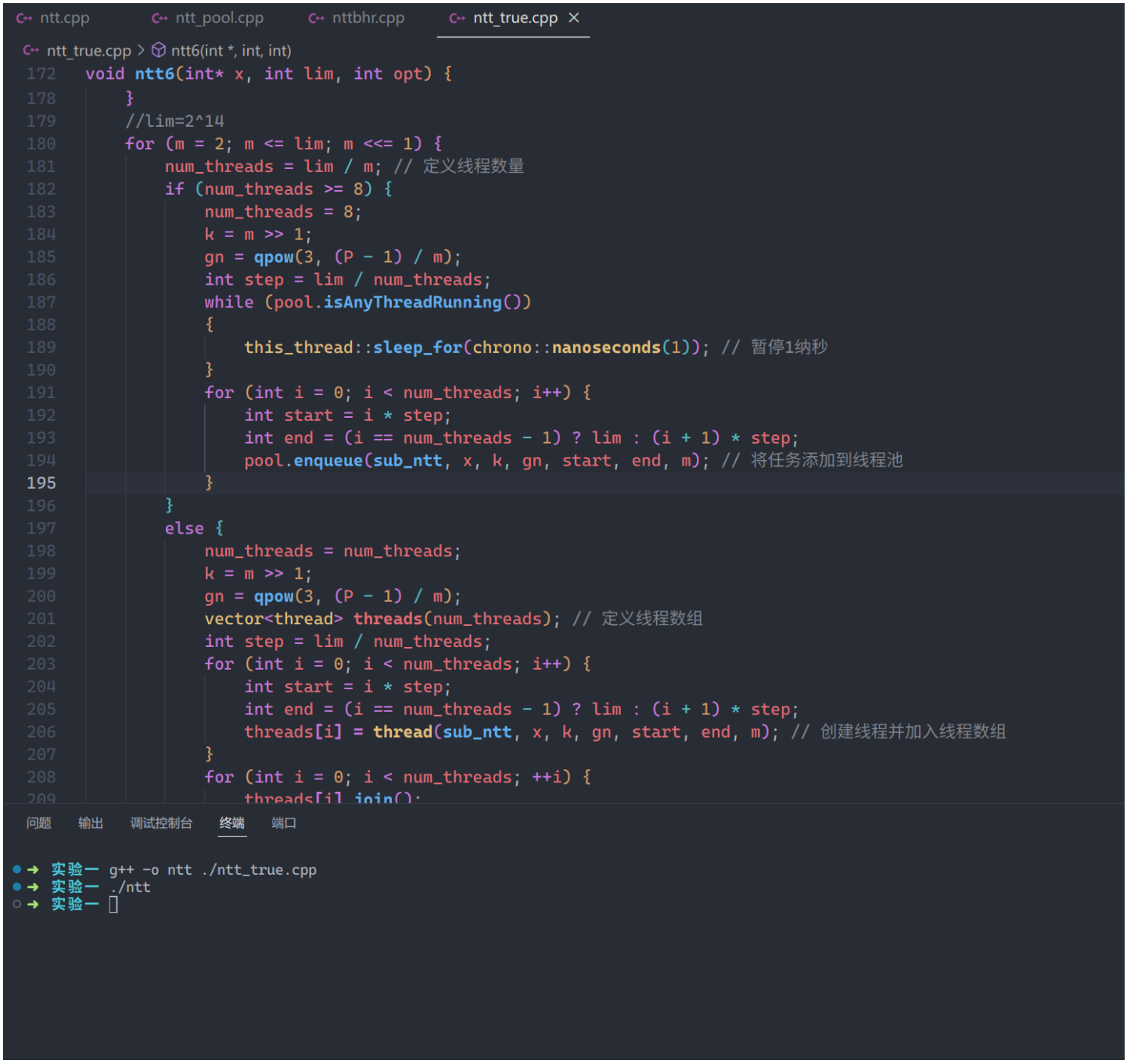
3.4.4.1 性能

```
$ g++ -o ntt ./ntt.cpp -O3
$ taskset -c 0-7 /usr/bin/perf stat -d -d -d ./ntt
NTT took 8.4249e+06 nanoseconds to execute.

 Performance counter stats for './ntt':

                 35.73 msec task-clock:u                                     #      2.418 CPUs utilized
                    0        context-switches:u                             #      0.000 /sec
                    0        cpu-migrations:u                               #      0.000 /sec
                  982        page-faults:u                                  #     27.485 K/sec
      <not counted>        cpu_atom/cycles/u                                             (0.00%)
          86,743,308        cpu_core/cycles/u                              #      2.428 GHz                         (79.51%)
      <not counted>        cpu_atom/instructions/u                                       (0.00%)
         212,688,998        cpu_core/instructions/u                                       (94.28%)
      <not counted>        cpu_atom/branches/u                                           (0.00%)
          11,770,048        cpu_core/branches/u                            #    329.433 M/sec
      <not counted>        cpu_atom/branch-misses/u                                      (0.00%)
              48,073        cpu_core/branch-misses/u

                TopdownL1 (cpu_core)                 #     39.4 %  tma_backend_bound
                                                     #      2.2 %  tma_bad_speculation
                                                     #      9.9 %  tma_frontend_bound
                                                     #     48.5 %  tma_retiring
      <not counted>        L1-dcache-loads:u                                             (0.00%)
          21,332,065        L1-dcache-loads:u                              #    597.064 M/sec
   <not supported>        L1-dcache-load-misses:u

             440,761        L1-dcache-load-misses:u
```

```
      <not counted>      LLC-loads:u                                                 (0.00%)
                27,607      LLC-loads:u                                   #  772.694 K/sec
      <not counted>      LLC-load-misses:u                                           (0.00%)
                 3,997      LLC-load-misses:u

      <not counted>      L1-icache-loads:u                                           (0.00%)
   <not supported>      L1-icache-loads:u

      <not counted>      L1-icache-load-misses:u                                     (0.00%)
      <not counted>      L1-icache-load-misses:u                                     (0.00%)
      <not counted>      dTLB-loads:u                                                (0.00%)
      <not counted>      dTLB-loads:u                                                (0.00%)
      <not counted>      dTLB-load-misses:u                                          (0.00%)
      <not counted>      dTLB-load-misses:u                                          (0.00%)
   <not supported>      iTLB-loads:u

   <not supported>      iTLB-loads:u

      <not counted>      iTLB-load-misses:u                                          (0.00%)
      <not counted>      iTLB-load-misses:u                                          (0.00%)
   <not supported>      L1-dcache-prefetches:u

   <not supported>      L1-dcache-prefetches:u

   <not supported>      L1-dcache-prefetch-misses:u

   <not supported>      L1-dcache-prefetch-misses:u


        0.014777762 seconds time elapsed

        0.027406000 seconds user
        0.006034000 seconds sys
```

可见该函数实现了 2.418 CPUs 的利用率，运行时间为 8ms 。

3.4.4.2 正确性



可见该 NTT 函数是正确的。

3.4.4.3 分析与思考

通过运行我们发现，这样优化性能会进一步提升，且正确性不会改变，实现成功的线程池优化。

虽然我们已经实现了利用线程池优化，但程序可能还有其他方式进行优化，我们也希望能够进行尝试，详见该报告中之后的 5 一些额外的尝试 部分。

4 基准测试

为评估优化效果，在多线程实验前，先进行单线程的基准测试。

4.1 环境说明

通过查询资料得知，实验环境中的 CPU 采用了 Intel 的“性能混合架构”，其 14 个核心中，有 6 个属于高性能（主频高，核心规模大，完整的超标量设计，支持超线程）的 P 核心，8个属于性能较低（主频低，核心规模小，超标量设计少，不支持超线程），能耗较低的 E 核心。

在实验中，我们发现：

- 程序并非自始至终运行在同一个核心上，操作系统内核会对其进行核心间的调度
- 同类核心间的调度对程序运行时间影响不大
- 不同核心间的调度对程序的运行时间影响较大，会产生较大的性能开销
    - 同时，我们在 AMD EPYC 7302 16C32T 这颗 CPU 上也进行了测试
        - 该 CPU 没有大小核，但是有 chiplet 设计
        - 在不同 chiplet 间调度也会产生较大的性能开销

4.2 测试方式

- 为保证单线程效率最优，使用`-O3`编译器优化，最大程度降低因代码写法和函数调用造成的性能开销
- 使用`taskset -c 0-11` 实现核心绑定，绑定 NTT 可执行文件到该 CPU 的 P 核心，从而最大程度减少因程序在 P 核心与 E 核心之间调度而造成的性能开销
- 使用 `perf` 工具实现简单的 Profiling ，在指令层面测试性能，参数为 `stat -d -d -d ./ntt`
- 最终运行程序的完整命令为：`taskset -c 0-11 /usr/bin/perf stat -d -d -d ./ntt`
- 为保证 CPU 不出现较为严重的过热降频，两次测试之间间隔 5s，共测试 10 次

4.2.1 单线程 NTT 基准测试

- t1：NTT 程序自身采集的运行时间，单位为 ns
- t2：perf 工具采集的 CPU 时间，单位为 ms
- F： perf 工具采集的 CPU 频率，单位为 GHz
- I： perf 工具采集的 CPU 指令数

- C：perf 工具采集的 CPU 使用数，体现并行度

次数	t1	t2	F	I	C
1	2.4909E+07	30.61	3.662GHz	358,731,853	0.982 CPUs
2	2.5359E+07	30.97	3.752GHz	361,920,168	0.982 CPUs
3	2.4950E+07	30.95	3.419GHz	342,186,722	0.985 CPUs
4	2.5348E+07	31.22	4.054GHz	411,861,595	0.984 CPUs
5	2.5210E+07	31.19	4.356GHz	457,522,587	0.981 CPUs
6	2.6025E+07	31.97	3.631GHz	341,112,062	0.984 CPUs
7	2.5392E+07	31.58	4.246GHz	457,054,613	0.981 CPUs
8	2.5046E+07	30.61	3.513GHz	236,848,997	0.985 CPUs
9	2.5482E+07	31.51	3.752GHz	369,213,254	0.982 CPUs
10	2.5014E+07	30.82	4.037GHz	413,310,418	0.981 CPUs
avg	2.5273E+07	31.14	3.842GHz	374,976,227	0.983 CPUs

4.2.2 成功的多线程优化 NTT 函数基准测试

- t1：NTT 程序自身采集的运行时间，单位为 ns
- t2：perf 工具采集的 CPU 时间，单位为 ms，即所有线程运行的总时间
- F： perf 工具采集的 CPU 频率，单位为 GHz，这里多线程下主频统计有错误
- I： perf 工具采集的 CPU 指令数，这里多线程下主频指令数只统计了主线程
- C：perf 工具采集的 CPU 使用数，体现并行度

次数	t1	t2	F	I	C
1	1.3934E+07	38.4	0.797GHz	70,884,54	1.937 CPUs
2	1.4684E+07	39.84	0.646GHz	70,167,944	1.922 CPUs
3	1.5189E+07	39.94	0.712GHz	76,005,659	1.901 CPUs
4	1.4920E+07	40.58	0.492GHz	33,337,221	1.933 CPUs
5	1.4292E+07	39.51	0.757GHz	70,489,192	1.932 CPUs
6	1.3902E+07	38.48	1.316GHz	76,963,956	1.953 CPUs
7	1.4724E+07	40.51	0.720GHz	71,582,965	1.939 CPUs
8	1.4258E+07	39.11	0.624GHz	72,247,242	1.964 CPUs
9	1.4270E+07	38.8	1.254GHz	83,444,284	1.966 CPUs
10	1.4229E+07	38.47	1.039GHz	76,673,301	1.943 CPUs
avg	1.4440E+07	39.36	0.836GHz	70,101,307	1.939 CPUs

4.2.3 使用 Intel VTune 进行更加“严谨”的 Profiling

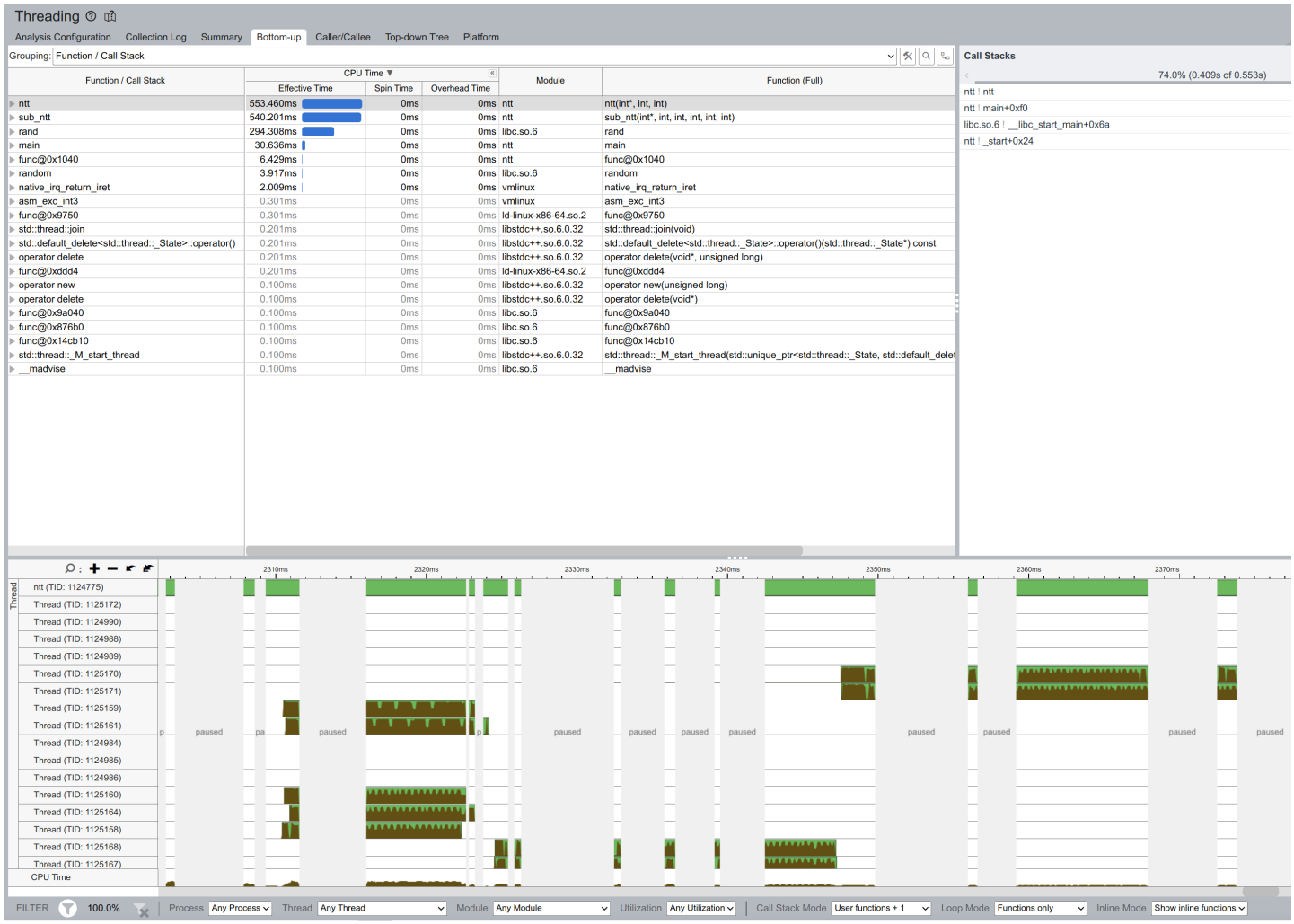
perf 工具能够提供基础的性能信息，要真正观察多线程程序的运行细节，我们使用了 Intel VTune 工具进行了更严谨的 Profiling。

4.2.3.1 对 NTT 程序作出一些修改

原始的 NTT 程序的运行时间只有数（十）毫秒，VTune 工具无法采集到足够多的性能数据，这里把 NTT 程序中的 `N` 从原始的 `300100` 修改为 `30000100`，手动延长运行时间。

4.2.3.2 Threading 分析

本次实验是多线程 NTT 的实现，因此这里主要进行线程相关的分析。使用环境说明中的编译指令重新编译后，做 Threading 分析：



将时间尺度缩放到计算部分，可见在 NTT 计算部分实现了多线程，各个线程之间根据依赖关系进行调度，运行和暂停，峰值可以实现五个计算线程同时运行。

4.3 局限性

虽然在基准测试中使用了核心绑定，提高优先级等措施，但在超线程、内核线程调度器策略等因素的影响下，基准测试数据仍有一定的局限性。要想得出更加准确有效的数据，需要换用线程调度更不积极的内核，保证一个线程绑定一个 CPU 核心，关闭 CPU 的超线程功能，尽量关闭系统内其他程序，最大程度提高 NTT 程序的优先级。

同时因为缺少部分内核模块与驱动，目前难以进行指令层面的 Profiling ，难以分析程序真正的瓶颈部分。

5 一些额外的尝试

5.1 OpenMP

通过查找资料得知，OpenMP 是一套 C/C++ 并行编程框架，仅需要对程序进行极小的改动，即可实现并行化。

对原单线程 NTT 函数作出如下修改：

```
void ntt(int *x, int lim, int opt) // 请使用多线程编程思想，对此函数进行优化
{
      int i, j, k, m, gn, g, tmp;
      for (i = 0; i < lim; ++i)
            if (r[i] < i)
                  swap(x[i], x[r[i]]);
      for (m = 2; m <= lim; m <<= 1)
     {
            k = m >> 1;
            gn = qpow(3, (P - 1) / m);
            #pragma omp parallel for // 加入 openmp 编译预处理选项
            for (i = 0; i < lim; i += m)
           {
                  g = 1;
                  // 1ll的目的是防止乘法的数据溢出
                  for (j = 0; j < k; j++)
                 {
                        tmp = 1ll * x[i + j + k] * g % P;
                        x[i + j + k] = (x[i + j] - tmp + P) % P;
                        x[i + j] = (x[i + j] + tmp) % P;
                        g = 1ll * g * gn % P; // 将 g 的迭代从 for 中移出至循环体内
                 }
           }
     }
}
```

编译选项为：`g++ -o nttomp -fopenmp -O3 ./ntt.cpp`

5.1.1 正确性测试

理论上正确使用 OpenMP 不会对程序的正确性造成影响，使用了同样的代码验证了正确性。

5.1.2 性能测试

```
$ taskset -c 0-7 /usr/bin/perf stat -d -d -d ./nttomp
NTT took 7.74293e+06 nanoseconds to execute.

 Performance counter stats for './nttomp':

```

```
                 58.52 msec task-clock                       #    4.188
CPUs utilized
                     3        context-switches                 #   51.265
/sec
                     1        cpu-migrations                   #
17.088 /sec
                   969        page-faults                      #
16.558 K/sec
      <not counted>        cpu_atom/cycles/
                                                  (0.00%)
           58,485,371        cpu_core/cycles/                 #    0.999
GHz                                               (7.55%)
      <not counted>        cpu_atom/instructions/
                                                  (0.00%)
           66,032,223        cpu_core/instructions/
                                                  (29.63%)
      <not counted>        cpu_atom/branches/
                                                  (0.00%)
           12,795,292        cpu_core/branches/               #  218.649
M/sec                                             (55.63%)
      <not counted>        cpu_atom/branch-misses/
                                                  (0.00%)
               61,941        cpu_core/branch-misses/
                                                  (90.12%)
                TopdownL1 (cpu_core)                 #       51.1 %
tma_backend_bound
                                                     #        2.2
%  tma_bad_speculation
                                                     #       11.0
%  tma_frontend_bound
                                                     #       35.7
%  tma_retiring                   (99.07%)
      <not counted>        L1-dcache-loads
                                                  (0.00%)
           24,756,106        L1-dcache-loads                  #  423.037
M/sec
   <not supported>        L1-dcache-load-misses

               446,565        L1-dcache-load-misses

      <not counted>        LLC-loads
                                                  (0.00%)
               46,876        LLC-loads                        #
801.027 K/sec
      <not counted>        LLC-load-misses
                                                  (0.00%)
               11,610        LLC-load-misses

      <not counted>        L1-icache-loads
                                                  (0.00%)
```

```
   <not supported>        L1-icache-loads

      <not counted>        L1-icache-load-misses
                                    (0.00%)
      <not counted>        L1-icache-load-misses
                                    (0.00%)
      <not counted>        dTLB-loads
                                       (0.00%)
      <not counted>        dTLB-loads
                                       (0.00%)
      <not counted>        dTLB-load-misses
                                    (0.00%)
      <not counted>        dTLB-load-misses
                                    (0.00%)
   <not supported>        iTLB-loads

   <not supported>        iTLB-loads

      <not counted>        iTLB-load-misses
                                    (0.00%)
      <not counted>        iTLB-load-misses
                                    (0.00%)
   <not supported>        L1-dcache-prefetches

   <not supported>        L1-dcache-prefetches

   <not supported>        L1-dcache-prefetch-misses

   <not supported>        L1-dcache-prefetch-misses


         0.013973460 seconds time elapsed

         0.047814000 seconds user
         0.000000000 seconds sys
```

5.1.3 基准测试

使用同样的软硬件环境进行基准测试：

- t1：NTT 程序自身采集的运行时间，单位为 ns
- t2：perf 工具采集的 CPU 时间，单位为 ms，即所有线程运行的总时间
- F： perf 工具采集的 CPU 频率，单位为 GHz，这里多线程下主频统计有错误
- I： perf 工具采集的 CPU 指令数，这里多线程下主频指令数只统计了主线程
- C：perf 工具采集的 CPU 使用数，体现并行度（在 OpenMP 环境下不完全体现并行度）

次数	t1	t2	F	I	C
1	6.9651E+06	53.48	1.038GHz	70,357,204	4.243 CPUs
2	7.1158E+06	54.62	1.002GHz	66,647,231	4.142 CPUs
3	6.7368E+06	51.93	1.057GHz	63,654,963	4.137 CPUs
4	6.9241E+06	53.14	1.027GHz	67,446,529	4.136 CPUs
5	7.1291E+06	54.71	0.988GHz	65,716,493	4.261 CPUs
6	7.3208E+06	56.52	1.009GHz	66,789,853	4.176 CPUs
7	6.8365E+06	52.64	1.045GHz	66,082,542	4.285 CPUs
8	6.7354E+06	52.54	1.005GHz	60,247,793	4.099 CPUs
9	7.1136E+06	54.67	0.980GHz	63,823,194	4.256 CPUs
10	7.0637E+06	55.51	0.967GHz	60,608,932	4.019 CPUs
avg	6.9941E+06	53.98	1.012GHz	65,137,473	4.175 CPUs

5.1.4 结论

OpenMP 相比于手写的多线程 NTT ，并行度更高，虽然有更大的开销（体现为 CPU 时间会更长），但是更高的并行度会加快它的运行速度。

5.2 ntt 函数的GPU优化

5.2.1 GPU 优化思路

该算法实现了基于CUDA的NTT算法。NTT是一种类似于快速傅里叶变换（FFT）的算法，用于在模素数P的有限域上进行多项式乘法。这个代码的目标是在GPU上并行计算NTT，以提高性能。

下面是对这个GPU优化NTT函数的思路分析：

设备端CUDA核函数 ntt_kernel：

1.ntt_kernel 是在GPU上执行的主要函数，用于执行NTT算法。每个线程处理输入数组的一个元素。

使用CUDA中的线程索引计算当前线程的位置 tid。首先，执行一系列置换操作，将输入数据按照 r 数组的预先计算好的置换顺序重新排列。然后，执行NTT算法的主要步骤：外层循环控制蝴蝶操作的步长 m，每次翻倍，直到超过数组长度。在每个步长下，内层循环按照当前步长执行蝴蝶操作。这些操作是NTT算法的核心，使用预计算的根 gn 进行乘法和加法。将计算得到的结果写回全局内存。

2.主机端函数 ntt_gpu：

ntt_gpu 函数负责在主机上管理GPU内存，并调用设备端核函数执行NTT。首先，分配所需的设备内存，并将输入数据从主机内存复制到设备内存。然后，确定启动核函数的网格和块的维度，以便将工作负载分配到GPU上的多个线程。调用 ntt_kernel 核函数执行NTT计算。最后，将计算结果从设备内存复制回主机内存，并释放设备内存。

3.主函数 main：

初始化输入数组 A 和 B，并计算 lim 的值。

初始化置换数组 r。

调用 ntt_gpu 函数执行NTT算法，并计时执行时间。对结果进行后续处理

总的来说，这个GPU优化的NTT函数的思路是通过CUDA并行计算框架，在GPU上同时处理多个元素，利用线程级并行性和数据并行性加速NTT算法的执行。

当进一步详细地分析GPU优化NTT函数时，我们可以着重考虑以下几个方面：

1. CUDA并行模型：

   (a) CUDA编程模型基于主机（CPU）和设备（GPU）之间的并行性。

   (b) 主机端负责管理设备内存、启动核函数以及处理核函数执行后的结果。

   (c) 设备端使用线程网格和线程块来组织并行计算任务。每个线程块中的线程可以协作共享数据和同步执行。

2. 设备端CUDA核函数 `ntt_kernel`：

   (a) `ntt_kernel` 函数中的每个线程负责处理输入数组的一个元素。

   (b) 使用线程索引 `tid` 计算当前线程在输入数组中的位置。

   (c) 在执行NTT算法之前，进行了一系列的置换操作，以确保输入数组满足NTT算法的要求。

   (d) NTT算法的核心是蝴蝶操作，这里通过循环执行，每次处理不同步长的蝴蝶操作，直到涵盖整个数组。

   (e) 在蝴蝶操作中，对当前元素和对应位置的元素进行乘法和加法操作，并使用预先计算好的根进行计算。

3. 主机端函数 `ntt_gpu`：

   (a) `ntt_gpu` 函数负责在主机上管理GPU内存，并调用设备端核函数执行NTT。

   (b) 首先，分配设备内存，并将输入数据从主机内存复制到设备内存。

   (c) 然后，确定启动核函数的网格和块的维度，以便将工作负载分配到多个线程上。

   (d) 调用核函数执行NTT计算。

   (e) 最后，将计算结果从设备内存复制回主机内存，并释放设备内存。

4. 主函数 `main`：

   (a) 初始化输入数组和相关参数。

   (b) 调用 `ntt_gpu` 函数执行NTT计算，并计时执行时间。

   (c) 对结果进行后续处理（这里并没有进行乘法操作，只是简单输出部分结果）。

通过CUDA并行计算框架，这个GPU优化的NTT函数利用了设备上大量的线程并行执行NTT算法，从而加速了整个计算过程。同时，通过GPU的并行性，可以有效地处理大规模的数据集，提高了算法的可扩展性和性能。

5.2.2 代码分析

1. 头文件包含

include <cuda_runtime.h>

include

include

include

include

include

- `<cuda_runtime.h>`：CUDA运行时API的头文件，提供了用于CUDA编程的函数和类型定义。
- `<iostream>`：C++标准输入输出流库。
- `<algorithm>`：C++标准算法库。
- `<cstring>`：C标准字符串操作库。
- `<chrono>`：C++标准时间库，用于计时。

1. 常量定义：

```
cppCopy code
const int N = 300100, P = 998244353;
```

1. 设备端CUDA函数：

   (a) `device_swap`：设备端函数，用于交换两个整数的值。

   (b) `ntt_kernel`：设备端CUDA核函数，实现NTT算法。

2. 主机端函数：

(a) `ntt_gpu`：主机端函数，调用设备端核函数执行NTT算法。

3. 逆NTT函数：

```
cppCopy code
void inverse_ntt(int *x, int lim) {// Your implementation of inverse NTT
}
```

1. 主函数 `main()`：

   (a) 定义数组 `A[]`、`B[]`、`C[]` 和 `r[]`。

   (b) 初始化数组 `A[]` 和 `B[]`。

   (c) 计算适当的 `lim` 值，确保是2的幂。

   (d) 初始化 `r[]` 数组。

   (e) 调用 `ntt_gpu()` 函数执行NTT算法，并计时执行时间。

   (f) 执行数组 `C[]` 的乘法操作。

   (g) 调用逆NTT函数（但实际上是空的）。

   (h) 输出部分结果。

2. GPU内存管理：

   (a) 使用 `cudaMalloc()` 在GPU上分配内存。

   (b) 使用 `cudaMemcpy()` 将数据从主机复制到设备，以及将结果从设备复制回主机。

   (c) 使用 `cudaFree()` 释放在GPU上分配的内存。

3. 核函数启动：

   (a) 使用 `<<<blocksPerGrid, threadsPerBlock>>>` 启动核函数，确定块和网格的维度。

这段代码的主要目的是通过CUDA并行计算执行NTT算法，并且可以用于加速多项式乘法。

ntt_kernel函数：

- cppCopy code global void ntt_kernel(int *x, int lim, int *r, int P, int *d_result) {int tid = blockIdx.x * blockDim.x + threadIdx.x; if (tid < lim) {// NTT算法int i, j, k, m, gn, g, tmp;for (i = 0; i < lim; ++i)if (r[i] < i)device_swap(x[i], x[r[i]]); for (m = 2; m <= lim; m <<= 1) { k = m >> 1; gn = 3; // 因为我们在设备端无法使用qpow函数，直接取3的幂次方for (i = 0; i < lim; i +=

```
m) {                g = 1;for (j = 0; j < k; j++, g = 1llg*gn%P) {
tmp = 1llx[i + j + k] * g%P;                x[i + j + k] = (x[i + j] - tmp + P) %
P;                x[i + j] = (x[i + j] + tmp) % P;                }        }     }
// 将结果写回全局内存        d_result[tid] = x[tid];    } }
```

解释：

- `global`：这个关键词表示这是一个CUDA核函数，将从主机代码调用并在GPU上执行。
- ntt_kernel函数接受几个参数：
  - `int *x`：指向输入数组的指针。
  - `int lim`：数组的大小。
  - `int *r`：NTT算法中使用的另一个输入数组。
  - `int P`：模数。
  - `int *d_result`：指向设备上结果数组的指针。
- `tid`：每个线程的唯一标识符，根据块和线程索引计算得出。

该核函数首先根据排列数组r[]重新排列输入数组x[]。这是NTT算法中的必要步骤。

然后，它在输入数组x[]上执行NTT算法。

最后，它将结果写回设备上的全局内存。

ntt_gpu函数：

```
cppCopy code
void ntt_gpu(int *x, int lim, int *r, int P, int *result) {int *d_x, *d_r,
*d_result;
// 在GPU上分配内存cudaMalloc((void**)&d_x, lim *
sizeof(int));cudaMalloc((void**)&d_r, lim *
sizeof(int));cudaMalloc((void**)&d_result, lim * sizeof(int));
// 将数据从主机复制到设备cudaMemcpy(d_x, x, lim * sizeof(int),
cudaMemcpyHostToDevice);cudaMemcpy(d_r, r, lim * sizeof(int),
cudaMemcpyHostToDevice);
// 确定块和网格的维度int threadsPerBlock = 256;int blocksPerGrid = (lim +
threadsPerBlock - 1) / threadsPerBlock;
// 启动核函数
      ntt_kernel<<<blocksPerGrid, threadsPerBlock>>>(d_x, lim, d_r, P, d_result);
// 将结果从设备复制回主机cudaMemcpy(result, d_result, lim * sizeof(int),
cudaMemcpyDeviceToHost);
// 在GPU上释放分配的内存cudaFree(d_x);cudaFree(d_r);cudaFree(d_result);
}
```

ntt_gpu是一个主机函数，负责在GPU上执行NTT算法。它首先为输入数组x[]、排列数组r[]和结果数组result[]在GPU上分配内存。然后，它将输入数据（x[]和r[]）从主机内存复制到设备内存。根据数组大小计算每个块的线程数量和块的数量。它使用计算得出的块和线程启动ntt_kernel CUDA核函数。在核函数执行后，它将结果从设备内存复制回主机内存。最后，它释放在GPU上分配的内存。

总体来说，ntt_gpu函数管理内存分配，在主机和设备之间进行数据传输，启动CUDA核函数，并在GPU上处理内存释放。实际的计算由ntt_kernel CUDA核函数执行。

5.2.3 结果分析和可行性分析

```
NTT(GPU) took 3.40322e+07 nanoseconds to execute.
A:
675225 773821243
B:
675225 150618887
C:
3 9
```

我们进行分析可知，我们固话输入的AB函数进行ntt运算，我们输出ABC三个数组的前两个值，具有实现可行性。在优化程度上，我们不难发现并没有明显的时间优化，我们推测是因为数据量小造成GPU分配内存时间抵消了数据运算优化时间

5.2.4 代码实现

```cpp
#include <cuda_runtime.h>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <chrono>

using namespace std;

const int N = 300100, P = 998244353;      // 模数为P，数组长度限制为N
// const int MAX_THREADS = 4; // 最大线程数量
// 在CUDA设备代码中手动交换数组元素
__device__ void device_swap(int& a, int& b) {
      int temp = a;
      a = b;
      b = temp;
}

// 设备端CUDA核函数，执行NTT算法
__global__ void ntt_kernel(int *x, int lim, int *r, int P, int *d_result) {
      int tid = blockIdx.x * blockDim.x + threadIdx.x;

      if (tid < lim) {
```

```
            // NTT算法
            int i, j, k, m, gn, g, tmp;
            for (i = 0; i < lim; ++i)
                  if (r[i] < i)
                  device_swap(x[i], x[r[i]]);

            for (m = 2; m <= lim; m <<= 1) {
                  k = m >> 1;
                  gn = 3; // 因为我们在设备端无法使用qpow函数，直接取3的幂次方
                  for (i = 0; i < lim; i += m) {
                        g = 1;
                        for (j = 0; j < k; j++, g = 1ll * g*gn%P) {
                              tmp = 1ll * x[i + j + k] * g%P;
                              x[i + j + k] = (x[i + j] - tmp + P) % P;
                              x[i + j] = (x[i + j] + tmp) % P;
                        }
                  }
            }

            // 将结果写回全局内存
            d_result[tid] = x[tid];
      }
}

// 主机端函数，调用设备端核函数执行NTT
void ntt_gpu(int *x, int lim, int *r, int P, int *result) {
      int *d_x, *d_r, *d_result;

      // 在GPU上分配内存
      cudaMalloc((void**)&d_x, lim * sizeof(int));
      cudaMalloc((void**)&d_r, lim * sizeof(int));
      cudaMalloc((void**)&d_result, lim * sizeof(int));

      // 将数据从主机复制到设备
      cudaMemcpy(d_x, x, lim * sizeof(int), cudaMemcpyHostToDevice);
      cudaMemcpy(d_r, r, lim * sizeof(int), cudaMemcpyHostToDevice);

      // 确定块和网格的维度
      int threadsPerBlock = 256;
      int blocksPerGrid = (lim + threadsPerBlock - 1) / threadsPerBlock;

      // 启动核函数
      ntt_kernel<<<blocksPerGrid, threadsPerBlock>>>(d_x, lim, d_r, P, d_result);

      // 将结果从设备复制回主机
      cudaMemcpy(result, d_result, lim * sizeof(int), cudaMemcpyDeviceToHost);

      // 在GPU上释放分配的内存
      cudaFree(d_x);
      cudaFree(d_r);
      cudaFree(d_result);
```

```
}

// 逆NTT函数，与实验无关，保持原样
void inverse_ntt(int *x, int lim) {
	// Your implementation of inverse NTT
}

int main() {
	const int n = N / 2; // 设 n 是 N 的一半，确保进行乘法后数据位数不会溢出
	int A[N], B[N], C[N], r[N];

	// 初始化数组A和B
	for (int i = 0; i < n; i++) {
		A[i] = i % 10; // 使用 i%10 使得数组元素为 0 到 9 之间的循环值
		B[i] = (i + 1) % 10; // 使用 (i+1)%10 使得数组元素为 1 到 10 之间的循环值
	}

	// 计算适当的 lim 值
	int lim = 1;
	while (lim < n) lim <<= 1;

	// 初始化 r 数组
	for (int i = 0; i < lim; ++i) {
		r[i] = (i & 1) * (lim >> 1) + (r[i >> 1] >> 1);
	}

	// 运行GPU上的NTT算法
	auto start = chrono::high_resolution_clock::now();
	ntt_gpu(A, lim, r, P, A);
	ntt_gpu(B, lim, r, P, B);
	auto end = chrono::high_resolution_clock::now();
	chrono::duration<double, nano> elapsed = end - start;

	cout << "NTT took " << elapsed.count() << " nanoseconds to execute." <<
endl;

	// 进行 C = A * B 的过程，与实验无关
	for (int i = 0; i < lim; ++i) {
		C[i] = 1ll * A[i] * B[i] % P;
	}

	// 对 C 进行逆 NTT，与实验无关
	inverse_ntt(C, lim);

	// 输出前两位元素
	cout << "A:" << endl;
	for (int i = 0; i < 2; ++i) {
		cout << A[i] << " ";
	}
	cout << endl;

```

```
124         cout << "B位:" << endl;
125         for (int i = 0; i < 2; ++i) {
126               cout << B[i] << " ";
127       }
128         cout << endl;
129
130         cout << "C:" << endl;
131         for (int i = 0; i < 2; ++i) {
132               cout << C[i] << " ";
133       }
134         cout << endl;
135
136         return 0;
137   }
138
```

5.3 CPU 和 GPU 运行 ntt 函数的对比

5.3.1 思路来源

我们随后十分好奇如果多线程的cpu和GPU运行ntt函数会有什么情况，两者差别会很大吗，但由于时间原因，我们尝试多次多线程ntt的cuda版本，但均没有成功，于是我们参考了一部分网上的代码，如下。

5.3.2 代码实现

```
1   #include <iostream>
2   #include <chrono>
3   #include <ctime>
4   #include <cstdlib>
5
6   #define r 17492915097719143606//模数的原根
7   #define p 0xFFFFFFFF00000001//通常情况下的模数
8
9   void rand_vector(uint64_t *vec,size_t n,uint64_t maxn){
10        srand(time(0));
11        for(size_t i=0;i<n;i++)
12              vec[i]=rand()%maxn;
13  }
14
15  /* base % mod */
16  inline __host__ __device__ uint64_t modulo(uint64_t base){
17        uint64_t result=base%p;
18        return result;
19  }
20
21  inline __host__ __device__ uint64_t mod_exp(uint64_t x,uint64_t y){
22        uint64_t res=1;
23        while(y){
```

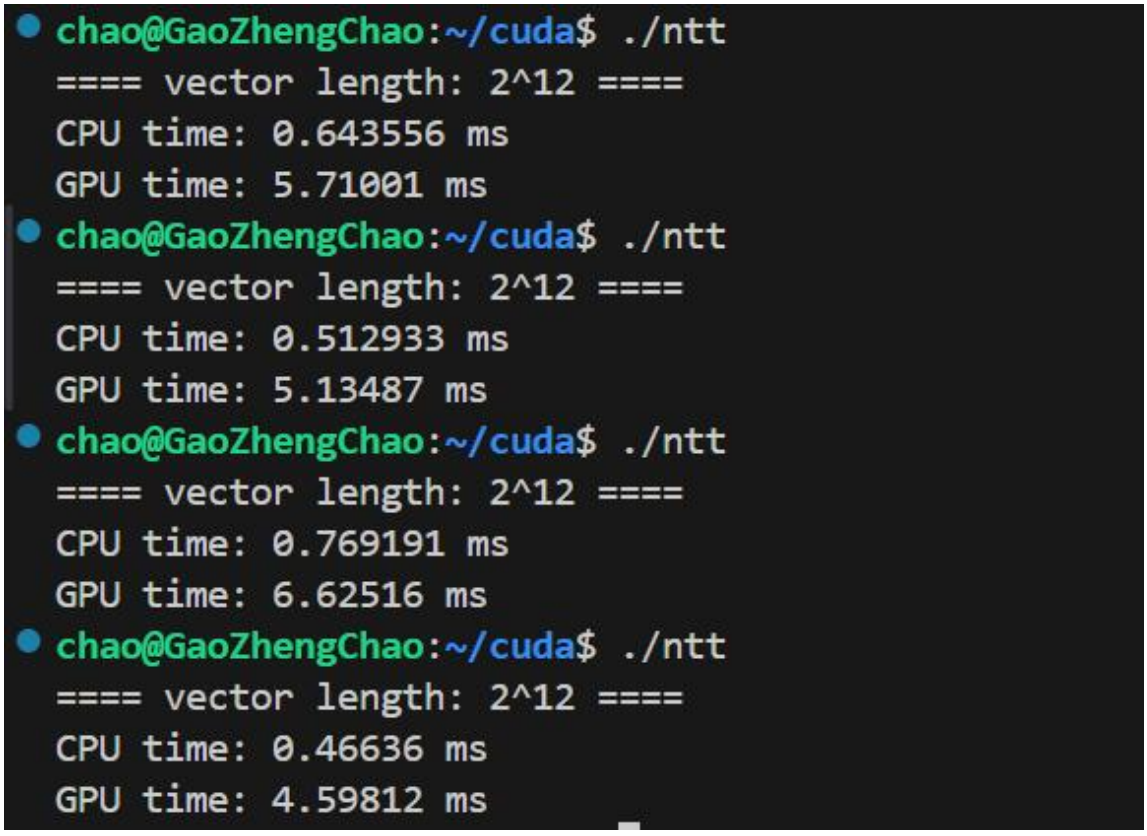
```
            if(y&1) res=modulo(res*x);
            x=modulo(x*x);
            y>>=1;
      }
      return res;
}

inline __device__ uint64_t exp(uint64_t x,uint64_t y){
      uint64_t res=1;
      while(y){
            if(y&1) res=res*x;
            x=x*x;
            y>>=1;
      }
      return res;
}

void ntt_cpu(uint64_t *vec,size_t *rev,size_t n,int bits){
      rev[0]=0;
      for(size_t i=0;i<n;i++)
            rev[i]=(rev[i>>1]>>1)|((i&1)<<(bits-1));
      for(size_t i=0;i<n;i++)
            if(i<rev[i]) std::swap(vec[i],vec[rev[i]]);
      for(size_t i=1;i<n;i<<=1){
            uint64_t wn=mod_exp(r,(p-1)/(2*i));
            for(size_t j=0,d=(i<<1);j<n;j+=d){
                  uint64_t w=1;
                  for(size_t k=0;k<i;k++){
                        uint64_t factor1=vec[j+k];
                        uint64_t factor2=modulo(w*vec[j+k+i]);
                        vec[j+k]=modulo(factor1+factor2);
                        vec[j+k+i]=modulo(factor1-factor2);
                        w=modulo(w*wn);
                  }
            }
      }
}

__global__ void bit_reverse_gpu(uint64_t *d_vec,size_t n,int bits){
      size_t tid=threadIdx.x+blockIdx.x*blockDim.x;
      if(tid>=n) return;
      uint64_t val=d_vec[tid];
      size_t old_id=tid,new_id=0;
      for(int i=0;i<bits;i++){
            int b=old_id&1;
            new_id=(new_id<<1)|b;
            old_id>>=1;
      }
      if(tid<new_id){
            uint64_t temp=d_vec[new_id];
            d_vec[tid]=temp;
```

```
75              d_vec[new_id]=val;
76      }
77  }
78
79  __global__ void twiddle_factor_gpu(uint64_t *d_twiddles,size_t n,int bits){
80      size_t tid=threadIdx.x+blockIdx.x*blockDim.x;
81      size_t total=(n>>1)*bits;
82      if(tid>=total) return;
83      size_t size=(n>>1);
84      size_t num=tid/size;// num-th iteration
85      size_t res=tid%size;// num-th iteration, res-th postion
86      size_t len=exp(2,num);// chunck size
87      d_twiddles[tid]=mod_exp(r,(p-1)/(2*len)*(res%len));
88  }
89
90  /* n/2 threads per iteration */
91  __global__ void ntt_kernel(uint64_t *d_vec,uint64_t *d_twiddles,size_t n,size_t
    iter,size_t chunck_size){
92      size_t tid=blockIdx.x*blockDim.x+threadIdx.x;
93      size_t half=n>>1;
94      if(tid>=half) return;
95      size_t vec_idx=(tid/chunck_size)*(2*chunck_size)+(tid%chunck_size); // index
    of vector
96      size_t twi_idx=iter*half+tid; // index of twiddles
97      uint64_t factor1=d_vec[vec_idx];
98      uint64_t factor2=modulo(d_twiddles[twi_idx]*d_vec[vec_idx+chunck_size]);
99      d_vec[vec_idx]=modulo(factor1+factor2);
100     d_vec[vec_idx+chunck_size]=modulo(factor1-factor2);
101 }
102
103 void ntt_gpu(uint64_t *d_vec,uint64_t *d_twiddles,size_t n,int bits){
104     size_t block_size=128;
105     size_t grid_size=(n+block_size-1)/block_size;
106     bit_reverse_gpu<<<grid_size,block_size>>>(d_vec,n,bits);// bit_reverse: one
    thread for one number
107     size_t total=(n>>1)*bits;
108     grid_size=(total+block_size-1)/block_size;
109     twiddle_factor_gpu<<<grid_size,block_size>>>(d_twiddles,n,bits);//
    preprocessing twiddle factor, bits * n/2
110     size_t iter=0;
111     for(size_t i=1;i<n;i<<=1){
112         grid_size=((n>>1)+block_size-1)/block_size;
113         ntt_kernel<<<grid_size,block_size>>>(d_vec,d_twiddles,n,iter,i);// ntt
114         iter++;
115     }
116     cudaDeviceSynchronize();
117 }
118
119 bool check(uint64_t *cpu_result,uint64_t *gpu_result,size_t n){
120     for(size_t i=0;i<n;i++){
121         if(cpu_result[i]!=gpu_result[i])
```

```
122                 return false;
123     }
124      return true;
125 }
126 
127 int main(int argc,char *argv[]){
128      int bits=12;
129      if(argc==2) bits=atoi(argv[1]);
130      else if(argc>2){
131           std::cerr<<"arguments error"<<std::endl;
132           exit(-1);
133     }
134 
135      double cpu_time=0.0;
136      double gpu_time=0.0;
137      size_t n=(size_t)1<<bits;
138 
139      /* allocate cpu memory */
140      uint64_t *host_vector;
141      size_t *host_rev;
142      host_vector=(uint64_t*)malloc(sizeof(uint64_t)*n);
143      host_rev=(size_t*)malloc(sizeof(size_t)*n);
144 
145      /* allocate gpu memory */
146      uint64_t *device_vector;
147      uint64_t *device_twiddles;
148      // size_t *device_rev;
149      cudaMalloc(&device_vector,sizeof(uint64_t)*n);
150      cudaMalloc(&device_twiddles,sizeof(uint64_t)*(n>>1)*bits);// bits
iterations, n/2 twiddles per iteration
151 
152      /* init vector */
153      rand_vector(host_vector,n,n);
154      /* host to device */
155 
  cudaMemcpy(device_vector,host_vector,sizeof(uint64_t)*n,cudaMemcpyHostToDevice);
156 
157      /* ntt on cpu */
158      auto start = std::chrono::high_resolution_clock::now();
159      ntt_cpu(host_vector,host_rev,n,bits);//======== cpu ntt
160      auto end = std::chrono::high_resolution_clock::now();
161      std::chrono::duration<double, std::milli> elapsed=end-start;
162      cpu_time=elapsed.count();
163 
164      /* ntt on gpu */
165      start = std::chrono::high_resolution_clock::now();
166      ntt_gpu(device_vector,device_twiddles,n,bits);//======== gpu ntt
167      end = std::chrono::high_resolution_clock::now();
168      elapsed=end-start;
169      gpu_time=elapsed.count();
170      /* copy result from gpu to cpu */
```

```
171         uint64_t *gpu_result;
172         gpu_result=(uint64_t*)malloc(sizeof(uint64_t)*n);
173 
    cudaMemcpy(gpu_result,device_vector,sizeof(uint64_t)*n,cudaMemcpyDeviceToHost);
174 
175         std::cout<<"==== vector length: 2^"<<bits<<" ===="<<std::endl;
176         std::cout<<"CPU time: "<<cpu_time<<" ms"<<std::endl;
177         std::cout<<"GPU time: "<<gpu_time<<" ms"<<std::endl;
178         // if(check(host_vector,gpu_result,n))  printf("all correct\n");
179         // else printf("error\n"); // 检查CPU和GPU的输出结果是否相同
180 
181         free(host_vector);
182         free(host_rev);
183         free(gpu_result);
184         cudaFree(device_vector);
185         cudaFree(device_twiddles);
186 
187         return 0;
188 }
```

5.3.3 运行结果



5.3.4 结果分析

我们不难发现GPU的时间比CPU平均大了将近三四倍，我们推测是由于尽管GPU比起CPU处理数据较快，但开多线程较慢的原因，由于数据量过小，所以造成GPU耗时更多。

6 实验结论

6.1 多线程加速分析

根据 4 基准测试中的数据，10组数据取平均值，计算多线程的加速比

$$\frac{2.5273 \times 10^7}{1.4440 \times 10^7} \approx 2.21$$

加速比约为 2.21 。

相比与 OpenMP 的结果，该加速比并不高，猜测主要原因在于线程之间需要互相等待，线程之间的执行顺序和依赖关系仍然有很大的优化空间。

后续可能可以继续在数据的并行分组方面进行优化，以实现更好的优化性能，例如在底层操作变化时提高线程数，根据不同层数研究最优的线程分组，或者是在高层的操作变化中，对数据分组进行优化以避免每层的线程数不同带来的相应开销，继续研究数据之间的关系以提高并行能力。

6.2 多线程性能分析

以成功的多线程为例，观察 perf 工具提供的数据

```
NTT took 1.41858e+07 nanoseconds to execute.
 Performance counter stats for './ntt':
                 38.81 msec task-clock:u                     #      1.945
CPUs utilized
                     0        context-switches:u             #      0.000
/sec
                     0        cpu-migrations:u               #      0.000
/sec
                 1,174        page-faults:u                  #     30.249
K/sec
       <not counted>        cpu_atom/cycles/u
                                                  (0.00%)
            40,656,236        cpu_core/cycles/u              #      1.048
GHz                                 (8.85%)
       <not counted>        cpu_atom/instructions/u
                                    (0.00%)
            76,544,814        cpu_core/instructions/u
                                    (28.00%)
       <not counted>        cpu_atom/branches/u
                                    (0.00%)
            11,062,708        cpu_core/branches/u            #    285.041
M/sec                               (84.54%)
       <not counted>        cpu_atom/branch-misses/u
                                  (0.00%)
                44,290        cpu_core/branch-misses/u
                                  (99.70%)
```

```
                  TopdownL1 (cpu_core)                 #      54.5 %
tma_backend_bound
                                                       #       0.7
%  tma_bad_speculation
                                                       #       2.7
%  tma_frontend_bound
                                                       #      42.1
%  tma_retiring
       <not counted>      L1-dcache-loads:u
                                                  (0.00%)
           21,454,653      L1-dcache-loads:u            #  552.799
M/sec
    <not supported>      L1-dcache-load-misses:u

               471,023      L1-dcache-load-misses:u

       <not counted>      LLC-loads:u
                                                  (0.00%)
                25,922      LLC-loads:u                  #  667.904
K/sec
       <not counted>      LLC-load-misses:u
                                                  (0.00%)
                 4,897      LLC-load-misses:u

       <not counted>      L1-icache-loads:u
                                                  (0.00%)
    <not supported>      L1-icache-loads:u

       <not counted>      L1-icache-load-misses:u
                                                  (0.00%)
       <not counted>      L1-icache-load-misses:u
                                                  (0.00%)
       <not counted>      dTLB-loads:u
                                                  (0.00%)
       <not counted>      dTLB-loads:u
                                                  (0.00%)
       <not counted>      dTLB-load-misses:u
                                                  (0.00%)
       <not counted>      dTLB-load-misses:u
                                                  (0.00%)
    <not supported>      iTLB-loads:u

    <not supported>      iTLB-loads:u

       <not counted>      iTLB-load-misses:u
                                                  (0.00%)
       <not counted>      iTLB-load-misses:u
                                                  (0.00%)
    <not supported>      L1-dcache-prefetches:u
```

```
    <not supported>       L1-dcache-prefetches:u
    <not supported>       L1-dcache-prefetch-misses:u
    <not supported>       L1-dcache-prefetch-misses:u

          0.019952322 seconds time elapsed

          0.013357000 seconds user
          0.005725000 seconds sys
```

可以发现，该程序的 IPC 约为 1.8，并没有充分利用 CPU 性能。

该程序可以利用 1.945 个 CPU 核心，实现了多线程，但并行度较低，同时 sys 占用时间较少，线程性能开销较小。

Branch-miss 率较低，仅为约 0.4%，对 CPU 分支预测的利用较好，分支预测失败率极低。

L1-cache miss 率较低，仅为约 18.8%，对 cache 的利用效率较高。

6.3 多线程优化的相关思考

在进行多线程优化时，首先要理清楚任务执行的基本逻辑，尤其是数据之间的依赖关系以及资源之间的相互作用，以避免资源冲突或者是错误的顺序带来的正确性不同，或者是在必要的操作中加入锁变量以避免资源冲突，其次是对于对立的任务之间进行合理的分组，以实现并行运算。同时，我们也要考虑线程建立与销毁对性能造成的影响，可以利用线程池或者是其他操作来进一步提高性能，综合考虑各方面的利弊确定最终的优化方案。

7 参考资料

1. https://blog.csdn.net/weixin_45825274/article/details/130944885
2. https://zhuanlan.zhihu.com/p/80297169
3. https://zhuanlan.zhihu.com/p/636156144
4. https://catslab-sdu.github.io/ntt%E5%BF%AB%E9%80%9F%E6%95%B0%E8%AE%BA%E5%8F%98%E6%8D%A2.html#13-%E5%BF%AB%E9%80%9F%E5%82%85%E9%87%8C%E5%8F%B6%E5%8F%98%E5%8C%96fft

8 附录

以下是实验相关代码目录，请在压缩包中查看：

1. ntt_thread.cpp 成功的多线程优化相关代码
2. ntt_threadpool.cpp 成功的线程池优化相关代码
3. ntt_gpu_my.cu 基于 GPU 的单线程 ntt 函数相关代码
4. ntt_gpu.cu 并行执行的 ntt 函数在 GPU 和 CPU 下运行的对比相关代码
5. nttomp.cpp 使用 OpenMP 优化 ntt 函数的相关代码