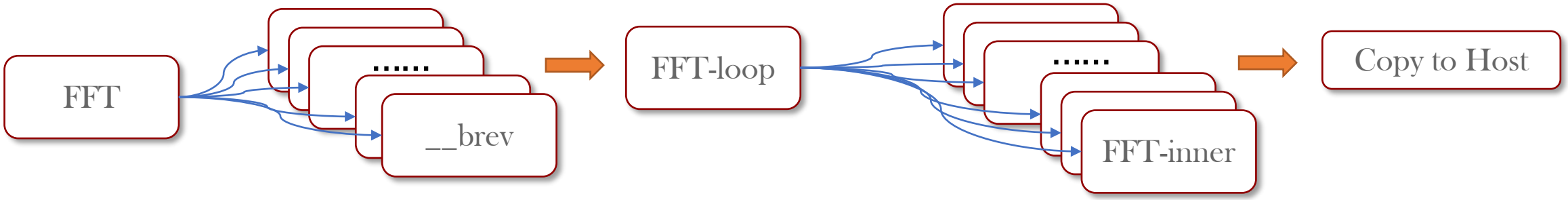
FFT and iFFT under GPU

Implementations and Optimizations on FFT

2021.12.27

Yang Yang



Overview of Polynomial Multiplication

Given two polynomials :

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \qquad B(x) = \sum_{j=0}^{n-1} b_j x^j$$

The multiplication will be like :

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j \qquad c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

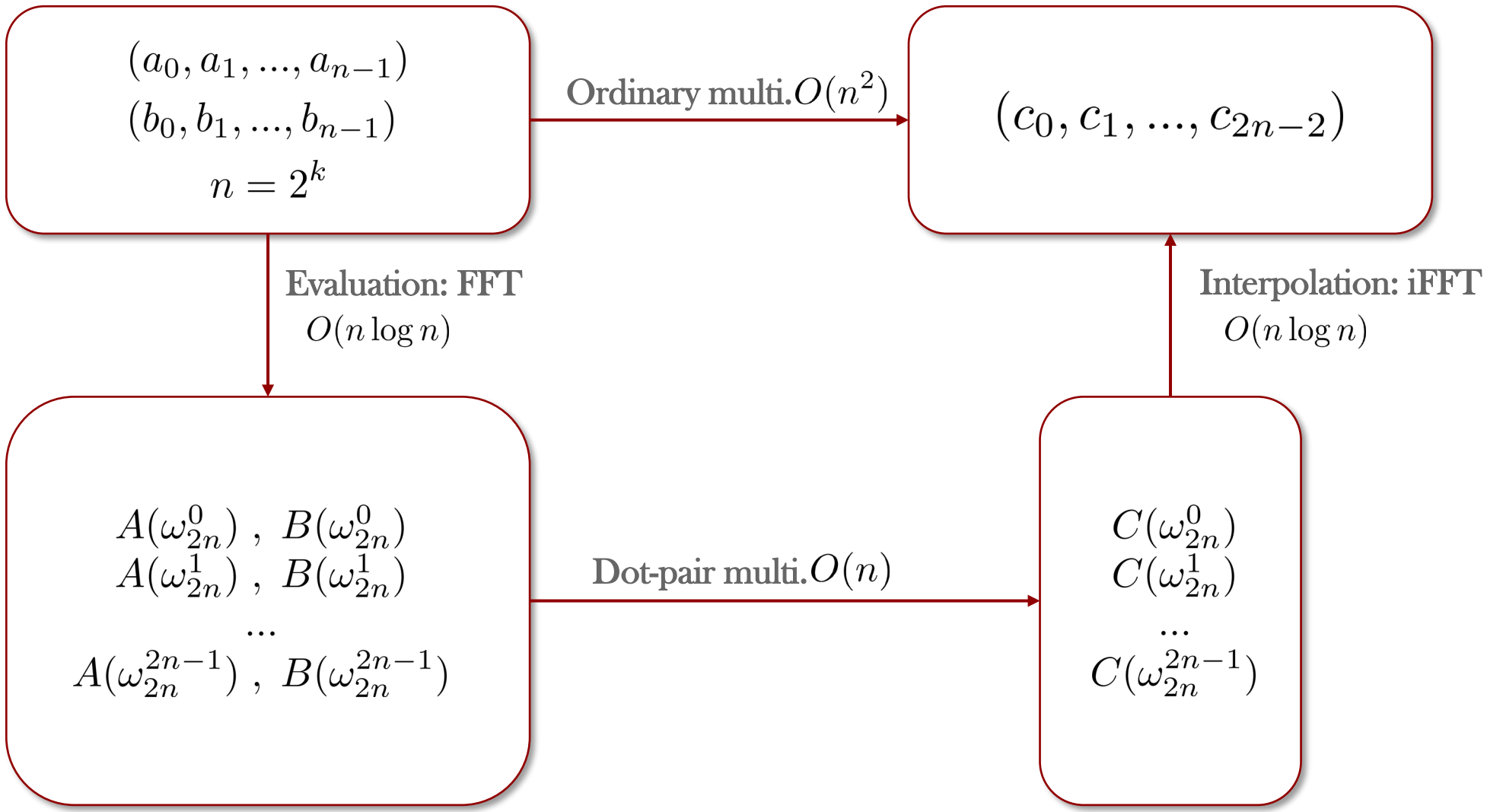
Ordinary coefficients multiplication will be in : $O(n^2)$

FFT along with iFFT(FFT-inverse) will reduce this process by value-based representation to: $O(n \log n)$

$$P(\omega^j) = P_e(\omega^{2j}) + \omega^j P_o(\omega^{2j})$$

$$P(\omega^{j+\frac{n}{2}}) = P_e(\omega^{2j}) - \omega^j P_o(\omega^{2j})$$

Overview of Polynomial Multiplication



Benchmark Environment

Local :
CPU : Intel i7-8550U 8 Core
GPU: NVIDIA MX-250
Language: C++
Compiler: NVIDIA 11.3 g++ 8.3.0
OS :Debian GNU/Linux 10 (buster)
Benchmark: Google Benchmark

Server :
CPU : Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz 12cores 24threads
GPU: Tesla V100-SXM2
Language: C++
Compiler: NVIDIA 10.1 (with O3 optimization)
OS : Ubuntu1 16.04.9
Benchmark: C++ chrono

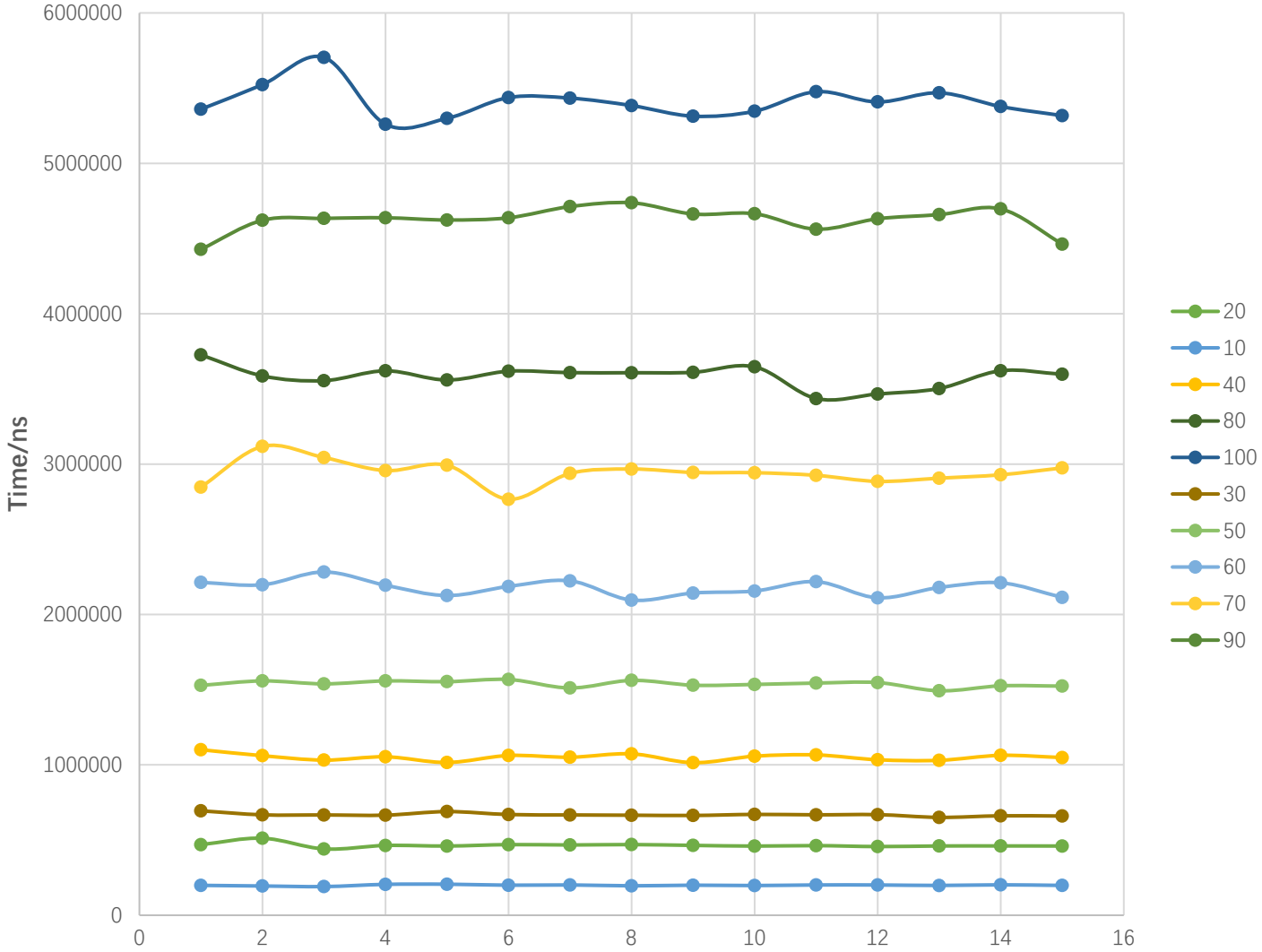
Ordinary Multiplication Method Benchmark

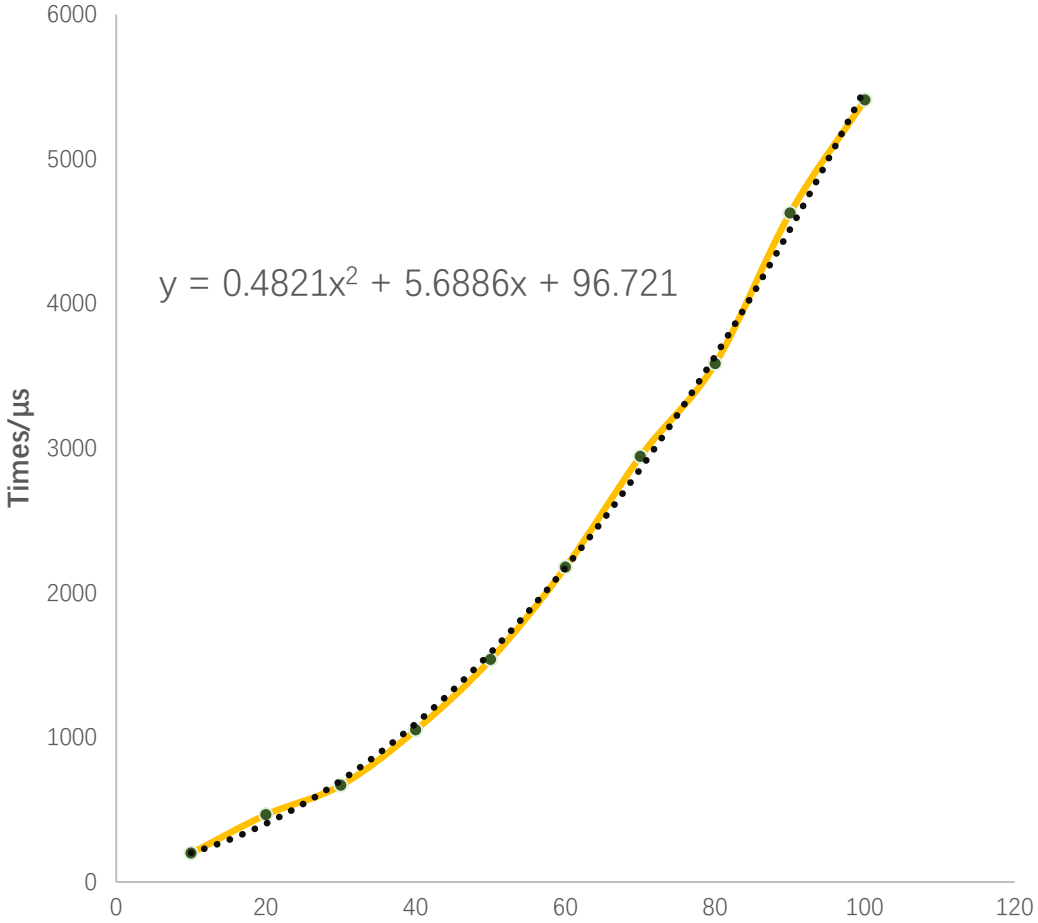
$O(n^2)$

```
friend Polynomial &operator*(const Polynomial &a, const Polynomial &b)
{
    ......
    for (int j = 0; j <= c_order; j++) {
        for (int k = 0; k <= j; k++) {
            cj += (a.coeff[k] * b.coeff[j - k]);
        }
        c_coeff.push_back(cj);
    }
    ......
}
```

Ordinary method:
9501 iterations on average for each multiplication

order	avg_time(μs)
10	4.96532
20	26.6619
30	59.0193
40	139.44
50	283.4867
60	560.6607
70	960.5893
80	1448.13
90	2354.206
100	3257.07


6000000
5000000
4000000
3000000
2000000
1000000
0
Time/ns
0
2
4
6
8
10
12
14
16
20
10
40
80
100
30
50
60
70
90


6000
5000
4000
3000
2000
1000
0
Times/μs
y = 0.4821x2 + 5.6886x + 96.721
0
20
40
60
80
100
120

FFT Based Method Benchmark

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

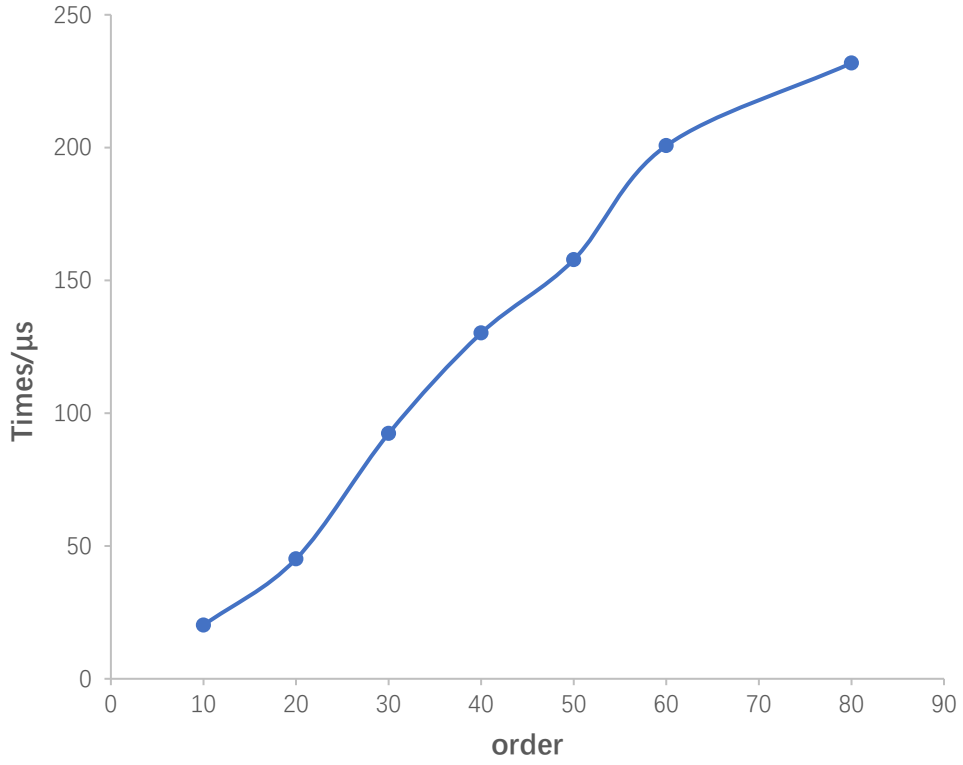
$O(n \log n)$

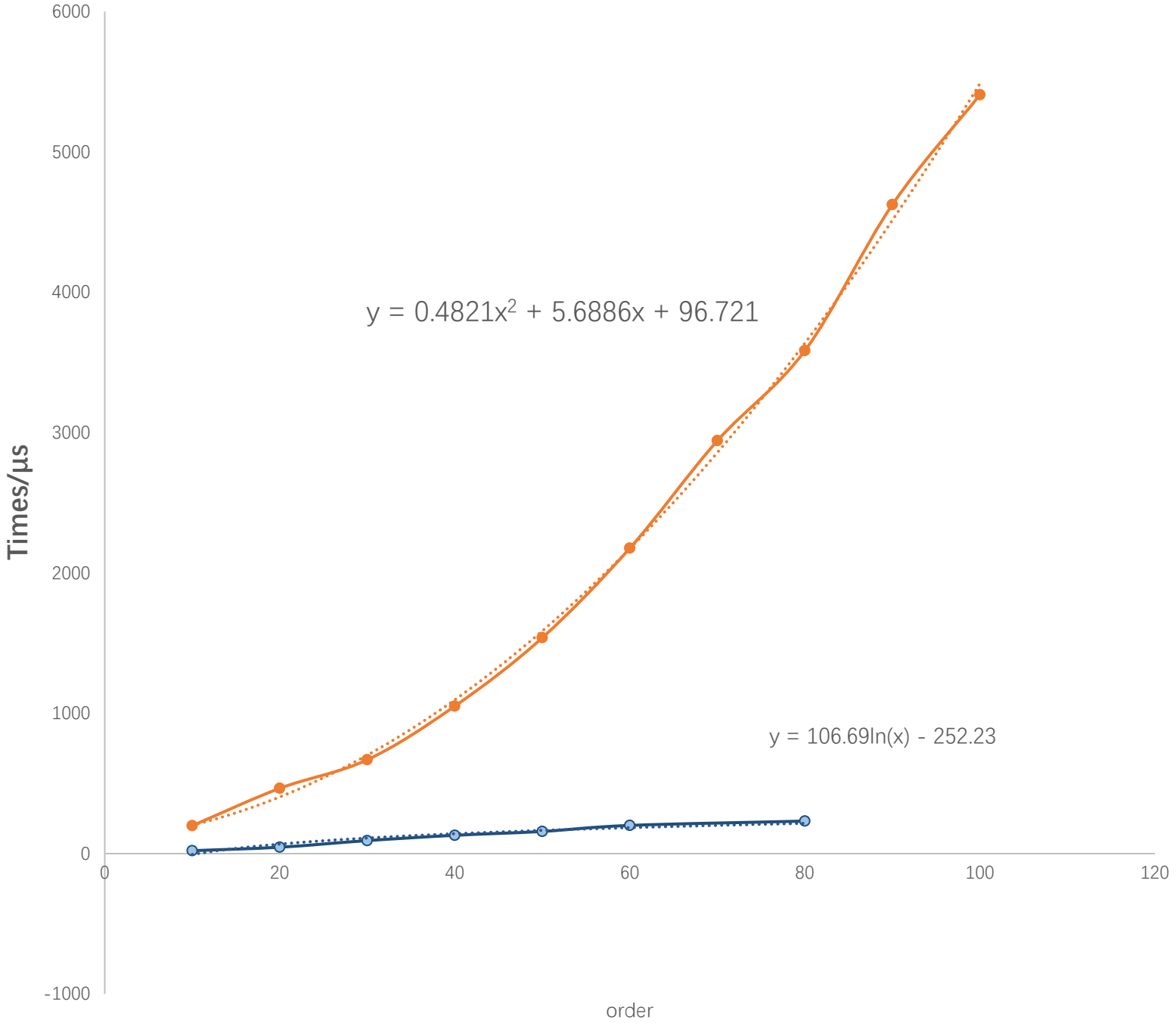
```
void rFFT(complex<double> a[], unsigned N, int flg){
    if (N == 1){
        return;
    }
    int half = N >> 1;
    ......
    rFFT(a0,half,flg);
    rFFT(a1,half,flg);
    complex<double> wn = complex<double>(cos(2 * pi / N), flg*sin(2*pi/N));
    complex<double> w = complex<double>(1,0);
    for(int k=0;k<=half-1;k++){
        complex<double> u = w*a1[k];
        a[k]=a0[k]+u;
        a[k+half]=a0[k]-u;
        w *= wn;
    }
}
```

$$T(N) = 2 * T(\frac{N}{2}) + \Theta(N) = \Theta(N \log N)$$

rFFT method:
12254 iterations on average for each multiplication

order	time(μs)
10	20.142
20	45.104
30	92.304
40	130.133
50	157.751
60	200.697
80	231.768



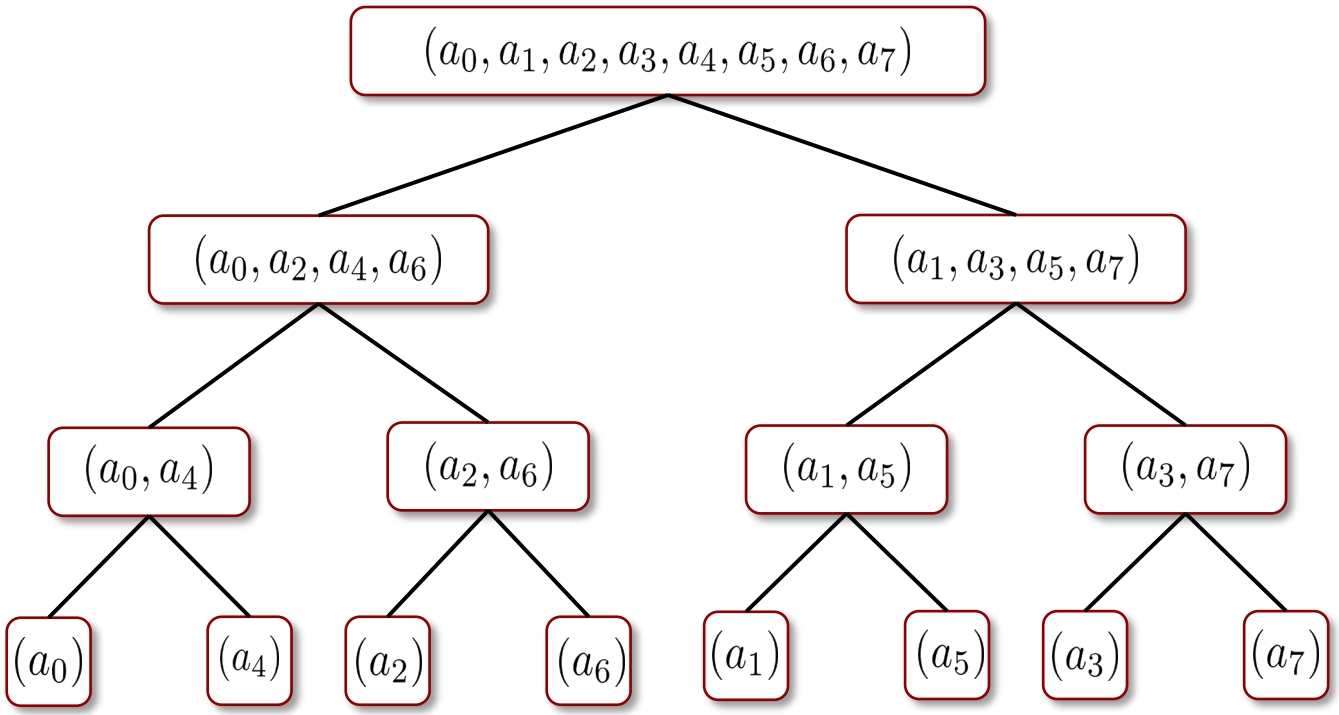


method	order	time(μs)
OM	80	3584
rFFT	80	231

Be faster ?

Butterfly Operation and Parallelization

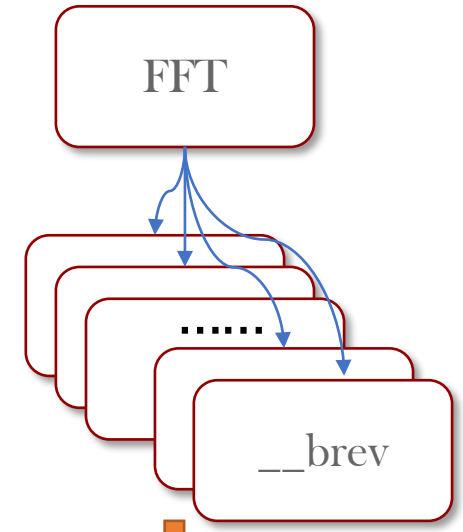
The process of rFFT when N=8



Change recursion to iteration by sort the array.
Put all these iterations on GPU

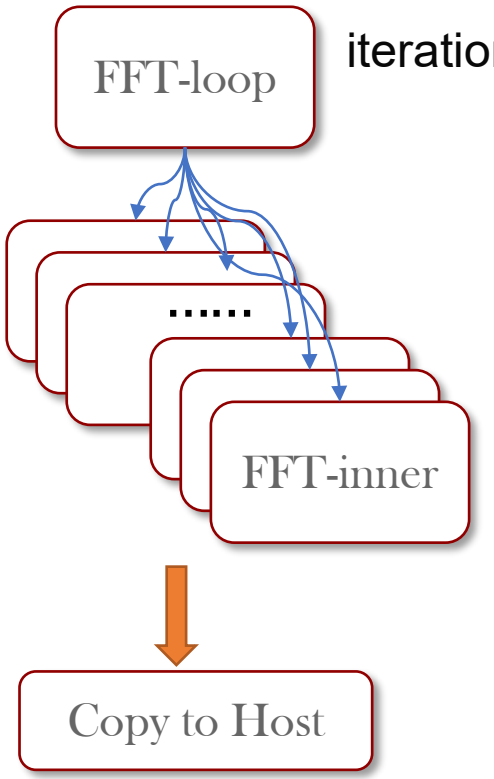
```
for(half =1;half<N;half<<1){
        w_m;
        for(j=0;j<N;j+=2*half){
                for(k=0;k<half;k++){
                        t
                        a[j+k]
                        a[j+k+half]
                }
        }
}
```

```
__device__ void iteration_FFT_core_loop(Complex* __restrict__ reordered, int j, int k, int m, int N,int flag) {
            ……
            Complex t,u;
            t.x = __cosf((2.0*PI*k) / (1.0*m));
            t.y = - __sinf(flag*(2.0*PI*k) / (1.0*m));
            ……
            reordered [k + j] = u*t;
            reordered [k + j + m / 2] = u-t;
      }
}
__global__ void iteration_FFT_core_loop_paralel(Complex* __restrict__ reordered, int m, int N, int threadN,int flag) {
      int j = (blockIdx.x *threadN + threadIdx.x)*m;
      for (int k = 0; k < m / 2; k++) {
            iteration_FFT_core_loop(reordered, j, k, m, N,flag);
      }
}
void FFT(Complex *__restrict__ data, size_t N, size_t threads,int flag) {
      ……
      int s = log2(N);
      bits_rev << < ceil(N / threads), threads >> > (reoredred, device,s,threads);
      for (int i = 1; i <= s; i++) {
            int m = 1 << i;
            iteration_FFT_core_loop_paralel <<< ceil((float)N / m / threads), threads >> > (reoredred, m, N, threads,flag);
      }
      ……
}
```

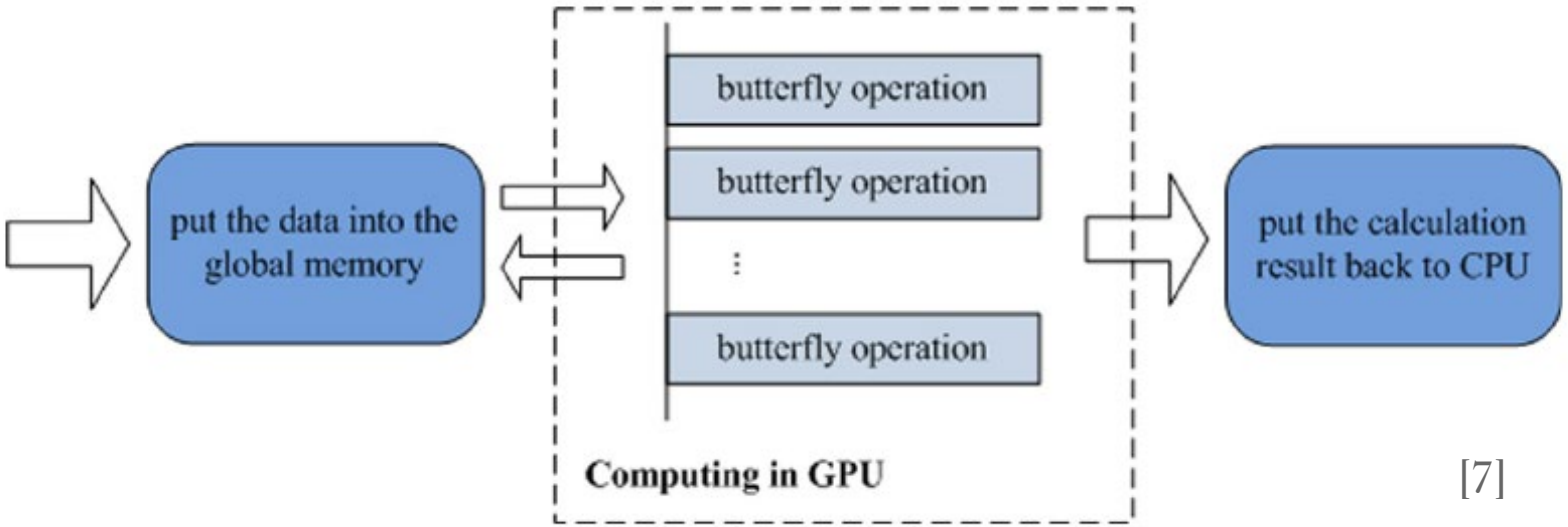


bits_rev <<< ceil(N / threads), threads >>> ;

cudaDeviceSynchronize()

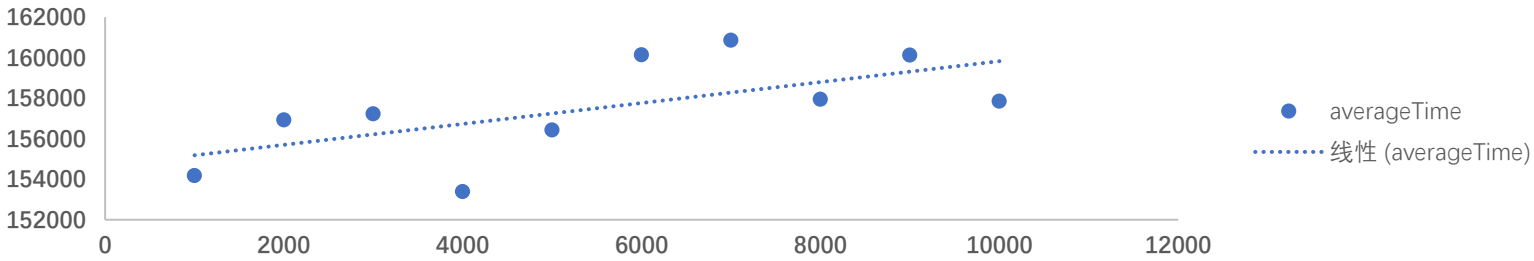


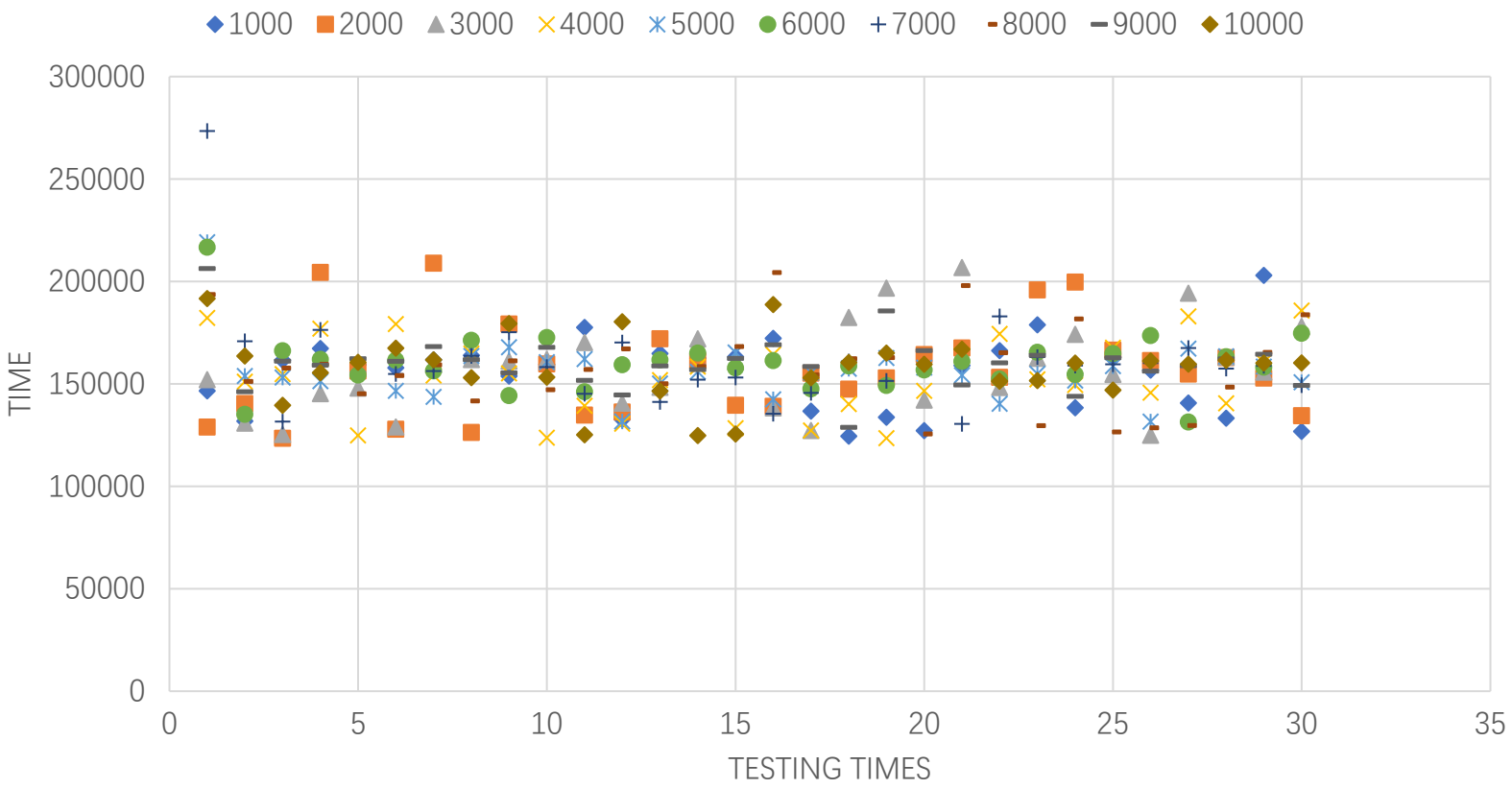
iteration_FFT_core_loop_paralel <<< ceil(N / m / threads), threads >>>

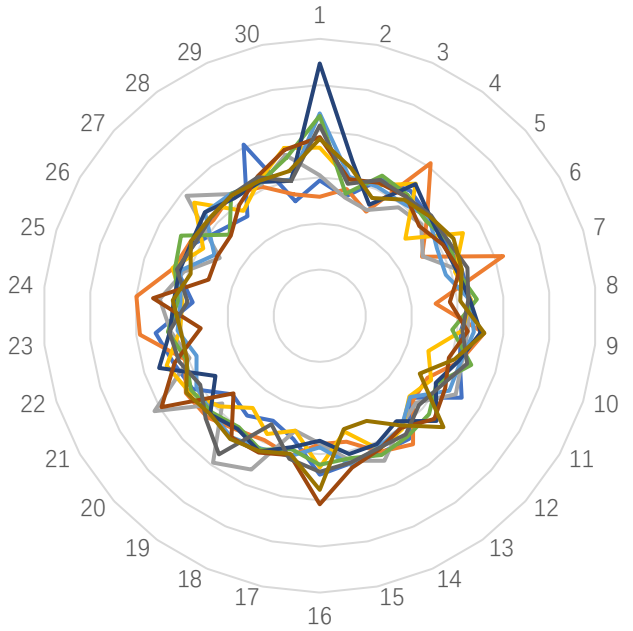


[7]

FFT-GPU Based Method Benchmark

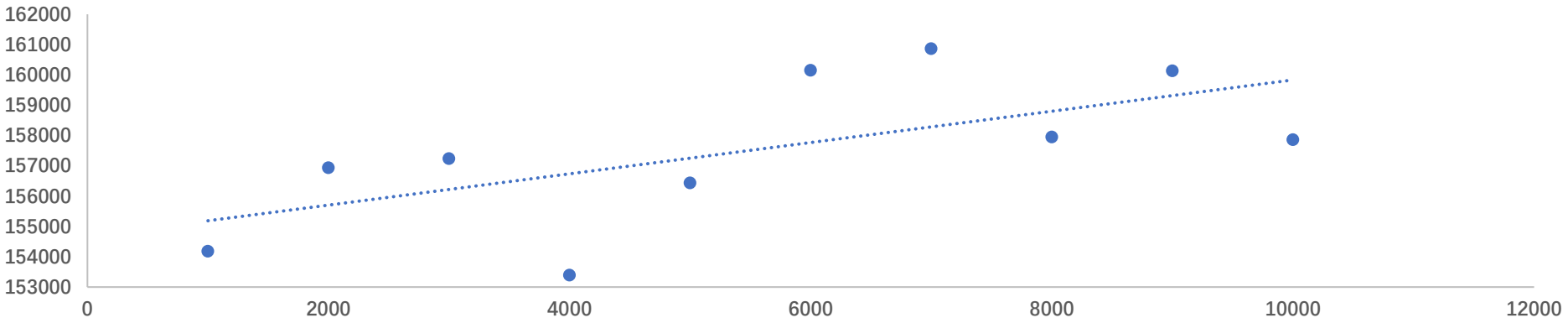


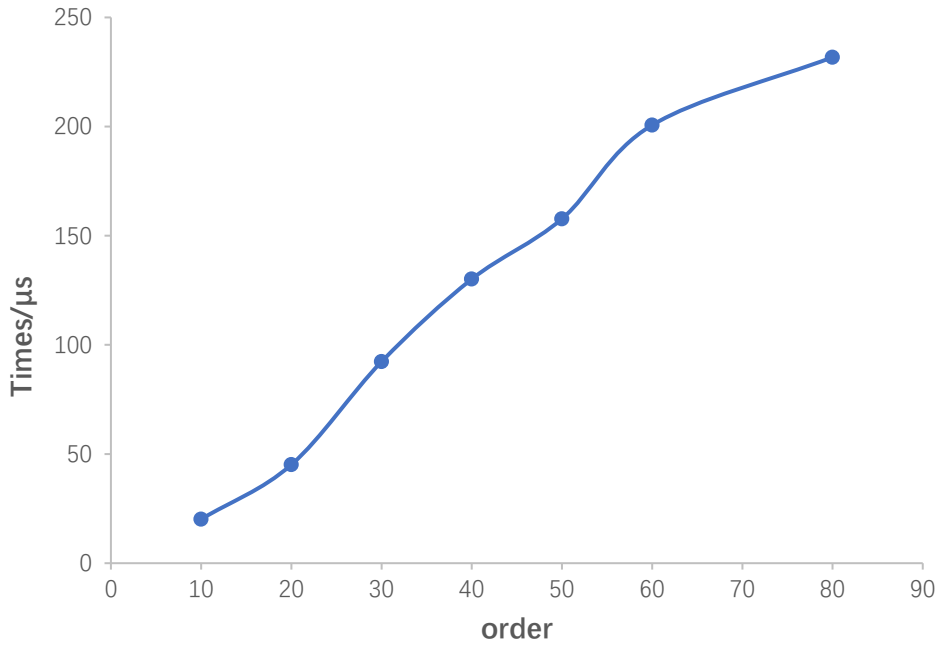


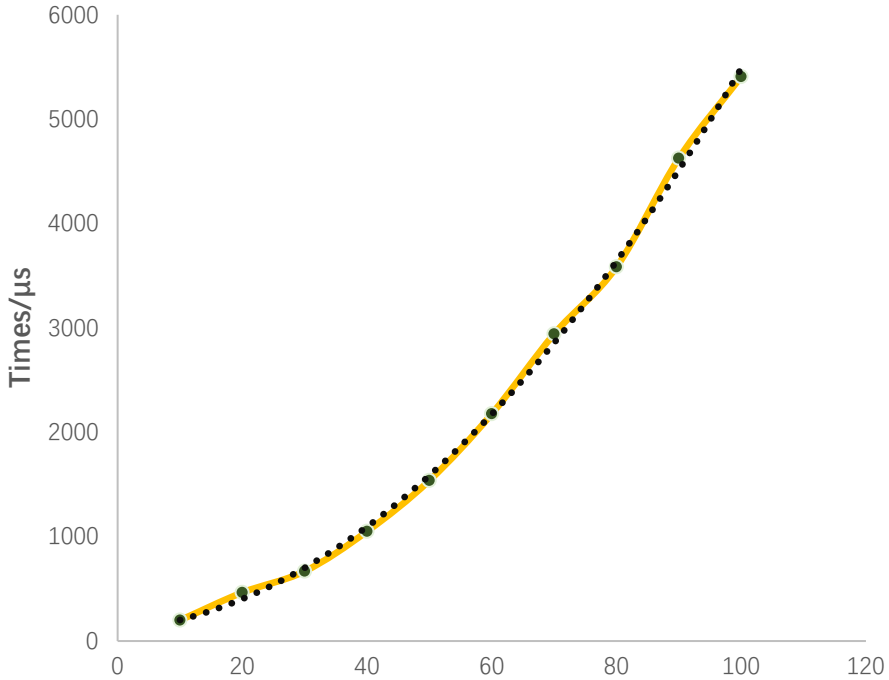


order	times(μs)
1000	154181
2000	156935
3000	157233
4000	153390
5000	156434
6000	160146
7000	160861
8000	157947
9000	160130
10000	157857

Comparison


162000
161000
160000
159000
158000
157000
156000
155000
154000
153000
0
2000
4000
6000
8000
10000
12000


250
200
150
100
50
0
Times/μs
0
10
20
30
40
50
60
70
80
90
order


6000
5000
4000
3000
2000
1000
0
Times/μs
0
20
40
60
80
100
120

Reference

[0] https://docs.nvidia.com/cuda/cufft/index.html
[2] https://developer.nvidia.com/cuda-example
[3] https://docs.nvidia.com/cuda/archive/11.3.0/pdf/CUDA_Math_API.pdf
[4] https://zhuanlan.zhihu.com/p/110897470
[5] Kenneth Moreland and Edward Angel. "The FFT on a GPU"
SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware (2003).
[6] Yu-hsuan Shih, Garrett Wright, Joakim Andén, Johannes Blaschke and Alex H. Barnett.
"cuFINUFFT: a load-balanced GPU library for general-purpose nonuniform FFTs"
International Parallel and Distributed Processing Symposium (2021).
[7] Fan Zhang, Chen Hu, Qiang Yin and Wei Hu.
"A GPU Based Memory Optimized Parallel Method For FFT Implementation"
arXiv: Distributed, Parallel, and Cluster Computing (2017): n. pag.
[8] Amir Gholami, Judith Hill, Dhairya Malhotra and George Biros.
"AccFFT: A library for distributed-memory FFT on CPU and GPU architectures"
arXiv: Distributed, Parallel, and Cluster Computing (2015): n. pag.
[9] C.L.R.S. "Introduction to Algorithms"