Big Integer Multiplication Using Parallel FFT

Zhihao Lv

January 30, 2020

Abstract

The Fast Fourier Transform(FFT) reduce the time complexity of big integer multiplication to $\Theta(n\log n)$. However, its parallel implementation is still needed due to the high algorithm constant causing by mass calculation of complex numbers. In this paper we discuss three ways to implement parallel FFT including OpenMP with shared memory, CUDA with cuFFT library and MPI with distributed memory. Details of implementation and runtime comparison is presented in the paper.

1 Introduction

1.1 FFT[2]

Fast Fourier transform is the algorithm that calculates the dicrete Fourier transform of secquence $X =< X[0], X[1], ..., X[n-1] >$ into sequence $Y =< Y[0], Y[1], ..., Y[n-1] >$ in $\Theta(n\log n)$:

$$Y[i] = \sum_{k=0}^{n-1} X[k]\omega^{ki} \qquad i = 0, 1, ..., n-1 \tag{1}$$

Here ω is the primitive n_{th} square root of unity in the complex plane. Assume that n is a power of 2(if not, we can add zero for aligment). Because of the properties of ω, Equation. 1 can be derived into:

$$Y[i] = \sum_{k=0}^{(n/2)-1} X[2k]\widetilde{\omega}^{ki} + \omega^{i} \sum_{k=0}^{(n/2)-1} X[2k+1]\widetilde{\omega}^{ki} \tag{2}$$

Here $\widetilde{\omega}$ is the primitive n_{th} square root of unity in the complex plane. Note that two summation on right side both require a half size calculation compared to the orignal problem, which satisfied the divide and conquer strategy.

1.2 Big Interger Multiplication

When numbers in multiplication is bigger 10^{18}, we cannot store them in *long long int* data type but to design high precision algorithm. Simple simulation with time complexity $\Theta(n\log n)$ is too slow for scientific computing. It is natural to link an integer with a polynomial, for example $234 \mapsto 2x^3 + 3x^2 + 4$. Since the multiplication of polynomials with

the point value representation is $\Theta(n)$, we can interpolate two polynomial with ω^{ki} by FFT and multiply in $\Theta(n)$. When performing the inverse Fourier transform, interpolate with ω^{-ki}:

$$Y_1[i] = \sum_{k=0}^{n-1} X_1[k]\omega^{ki} \qquad Y_2[i] = \sum_{k=0}^{n-1} X_2[k]\omega^{ki} \tag{3}$$

$$Z[i] = Y_1[i] * Y_2[i] \tag{4}$$

$$X[i] = \frac{1}{n}\sum_{k=0}^{n-1} Z[k]\omega^{-ki} \tag{5}$$

The whole process can be done within $\Theta(n\log n)$.

2 Serial Algorithm

2.1 Iterative Algorithm

It is not difficult to write a recursive FFT algorithm, but apparently it is too slow and some principles can be found in the recursive process(Figure. 1).

For n equals to 2^m, when returning to i^{th} level, the calculations are decribe in Equation. 6:

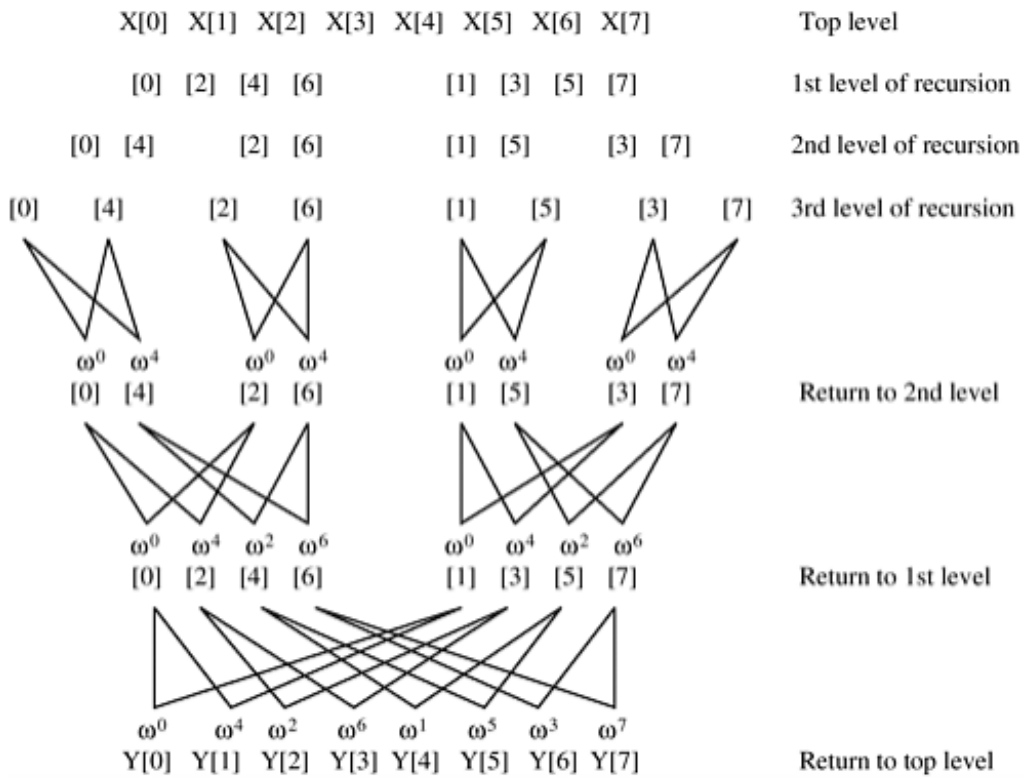

Figure 1: Recursive Process

$$\begin{cases} X_i[k] = X_{i+1}[k] + \omega^k X_{i+1}[k+2^{m-i}] \\ X_i[k+2^{m-i}] = X_{i+1}[k] - \omega^k X_{i+1}[k+2^{m-i}] \end{cases} \tag{6}$$

For every pair of two points, it is named as **Butterfly Operation**(Figure. 2).

Comparing the sequence at the bottom level and the top level, we can find that they

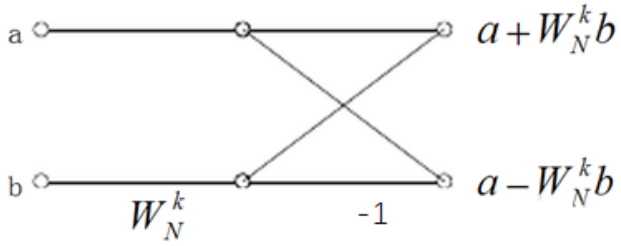

Figure 2: Butterfly Operation

Top(Decimal)	Top(Binary)	Bottom(Decimal)	Bottom(Binary)
0	000	0	000
1	001	4	100
2	010	2	010
3	011	6	110
4	100	1	001
5	101	5	101
6	110	3	011
7	111	7	111

Table 1: Bitwise Reverse

are reverse in bitwise(Table. 1).

Proposition 2.1 *The order of the end level sequence in FFT recursive process is exactly the bitwise reverse of the initial order.*

Therefore, if we swap the position in bitwise, butterfly operation can be performed iteratively, thus allowing the feasibility parallel algorithm.

2.2 Serial Code

Implementation of bitwise reverse is as Figure. 3.

```
1 for(int i=0;i<t;i++) pos[i]=(pos[i>>1]>>1)|((i&1)<<(n0-1));
```

Figure 3: Bitwise Reverse

Implementation of FFT is as Figure. 4

```
void FFT(cpx x[],int f){
    for(int i=0;i<t;i++)
        if(i<pos[i]) swap(x[i],x[pos[i]]);
    for(int i=1;i<t;i<<=1){
        cpx Wn=cpx(cos(PI/i),f*sin(PI/i));
        for(int j=0;j<t;j+=i<<1){
            cpx w=1;
            for(int k=0;k<i;k++,w*=Wn){
                cpx p=x[j+k],q=w*x[j+k+i];
                x[j+k]=p+q,x[j+k+i]=p-q;
            }
        }
    }if(f==-1) for(int i=0;i<t;i++) x[i]/=t;
}
```

Figure 4: Serial Code

3 OpenMP

3.1 Modification of Serial Code

Based on multithreading, the main principle of OpenMP is adding preprocessor directive *#pragma omp* to fork multiple thread and perform the following clause in parallel. The key part for parallelization is the FFT function. Of course, some other part like pre-pocessing will also be parallel if they do not have data dependencies.
In the serial code(Figure. 4), we can find that variable j and k in two for-loops from line 8 to line 10 actually have no dependency. If we make a copy of array x we can also overcome the data dependencies.

```
1  void FFT(cpx x[],bool f,cpx om[]){ //f==1 IDFT, f==0 DFT
2      int i,tmp,j,k;cpx t1;
3      #pragma omp parallel for shared(x,pos)
4      for(i=0;i<t;i++)
5          if(i<pos[i]) swap(x[i],x[pos[i]]);
6      for(i=1;i<t;i<<=1){
7          tmp=i<<1;
8          #pragma omp parallel \
9          shared(x,buf,om,i,t,tmp) private(j,k,t1)
10         {
11                 #pragma omp for
12                 for(j=0;j<t;j+=tmp){
13                 for(k=0;k<i;k++){
14                         t1=om[t/tmp*k]*x[j+k+i];
15                     buf[j+k]=x[j+k]+t1;
16                     buf[j+k+i]=x[j+k]-t1;
17                 }
18                 }
19                 #pragma omp for
20                 for(j=0;j<t;j++)         x[j] = buf[j];
21                 }
22     }
23     if(f)
24         #pragma omp parallel for shared(x)
25         for(i=0;i<t;i++) x[i]/=t;
26 }
```

Figure 5: OpenMP Code

3.2 Results

As is shown in the Figure. 6, the acceleration of OpenMP is not as good as I expected. The best result here is that about 2.3 times acceleration when $n = 10^6$ and $Thread = 32$. Only parallelizing the inner loop of FFT, clearly is not enough.

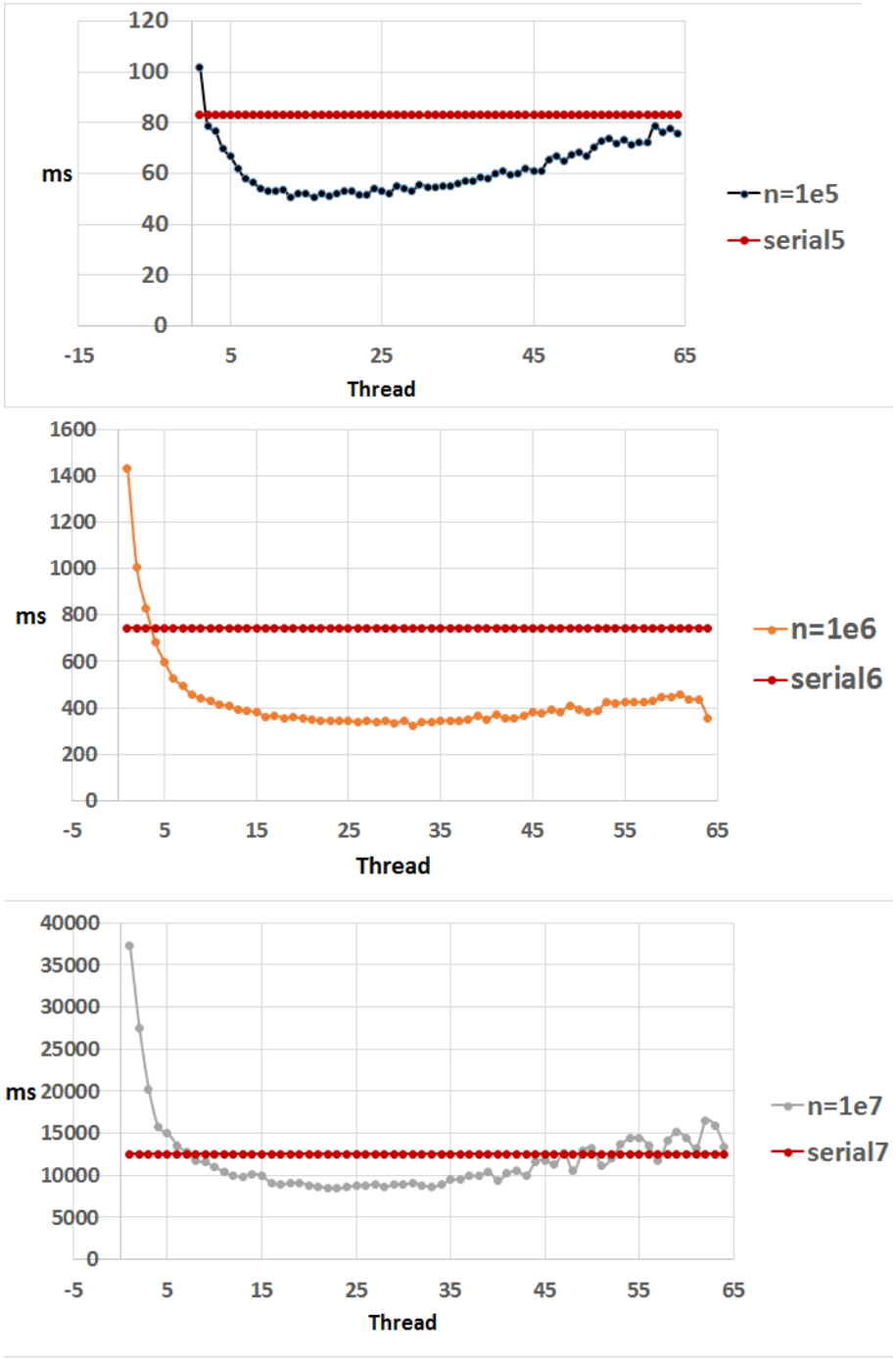

Figure 6: Time consumption

4 CUDA

4.1 cuFFT

The most time-consumption part of FFT is complex number calculation, while GPU computing with SIMD pattern can perfectly solve this problem. However, setting the thread and block number to achieve the best performance could be troublesome so I use CUDA additional library, cuFFT[1], in which the FFT function has been encapsuled and can be used as below(Figure. 7).

```
cudaMalloc((void **)&a, size);
cudaMemcpy(a, h_a, size, cudaMemcpyHostToDevice);
...
cufftHandle plan;
cufftPlan1d(&plan, t, CUFFT_Z2Z, 1);
cufftExecZ2Z(plan, a, a, CUFFT_FORWARD);
...
cufftExecZ2Z(plan, a, a, CUFFT_INVERSE);
cufftDestroy(plan);
```

Figure 7: cuFFT

Other computations such as multiplication and convertion of answers from complex to integer are clearly SIMD pattern and can be written in following form(Figure. 8).

```
__global__ void vector_mul(cufftDoubleComplex *a,
 cufftDoubleComplex *b){
        const int numThreads = blockDim.x * gridDim.x;
        const int threadID=blockIdx.x*blockDim.x+threadIdx.x;
        for (int i = threadID; i < T[0]; i += numThreads) {
                cuDoubleComplex c = cuCmul(a[i], b[i]);
                a[i] = make_cuDoubleComplex(cuCreal(c) / T[0],
                        cuCimag(c) / T[0]);
        }
}
...
cudaGetDeviceProperties(&prop, 0);
vector_mul<<<t / prop.maxThreadsPerBlock + 1,
 prop.maxThreadsPerBlock>>>(a, b);
```

Figure 8: Other kernel functions

4.2 Result

As is shown in Table. 2, the acceleration increase with n and is nonlinear compare to $log_2 n$. The acceleration can reach 11 when $n = 2^{25}$, using only a single *Tesla P40*. Nevertheless, due to the memory limit of GPU, it's not possible to compute when $n > 2^{25}$

n	serial(ms)	cufft(ms)	acceleration
32768	68.2045	329.971	0.206698233
65536	123.2825	414.499	0.297425024
131072	249.612	395.863	0.630550934
262144	551.473	446.309	1.235631089
524288	943.8755	465.259	2.028711267
1048576	2098.7495	549.251	3.821108606
2097152	4793.458	791.055	6.059579001
4194304	9232.303	1136.544	8.123137729
8388608	21197.706	2093.241	10.12673951
16777216	41702.4025	3619.876	11.52039539
33554432	85226.7815	7373.329	11.55879255
100000	82.952	314.737	0.263560079
1000000	742.522	428.001	1.734862321
10000000	12499.5892	2840.646	4.400262525

Table 2: cuFFT results

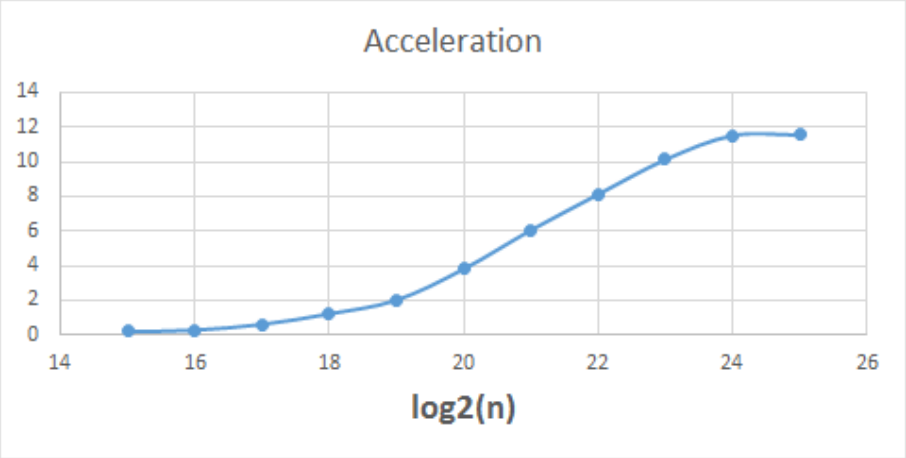

Figure 9: Accleration vs $log_2 n$

5 MPI

5.1 Iterative Process in Parallel

Review the iterative process after bitwise reverse. For example in Figure. 8, if there are 2 processors, in the first two rounds, they will calculate seperately and swap their data regularly in the third round.

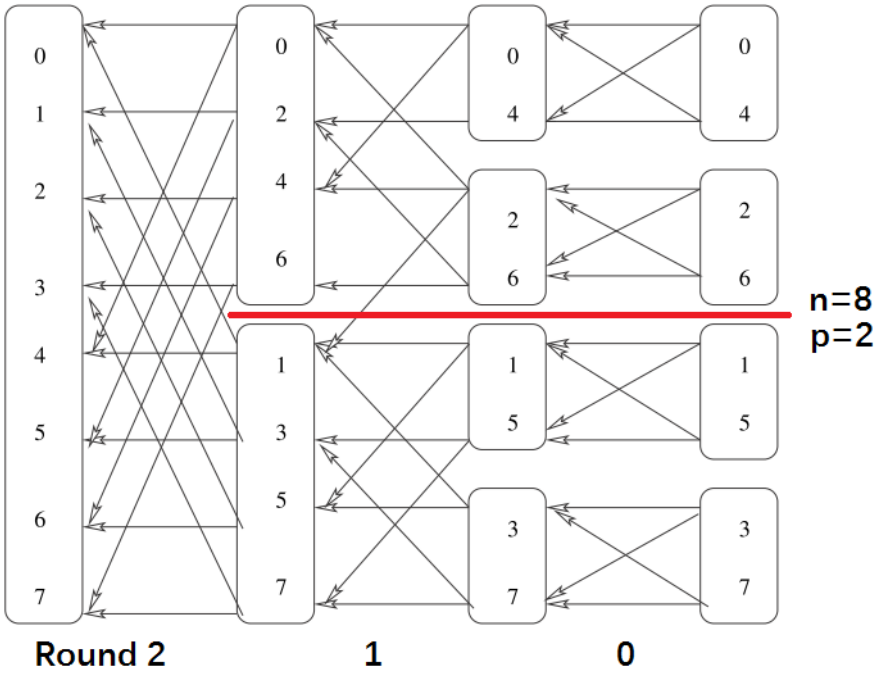

Figure 10: Iterative process[3]

Since we will make up n to a power of 2, it is natural to make the following assumption for alignment:

Assumption 5.1 *Number of processors p must be a power of 2 and less than $\sqrt{n}$*

Using MPI with distributed memory, we can divide the whole sequence in to p sections and distribut them to different processors. After the first $\log 2p$ rounds of iteration is conducted in each processors individually, in the following rounds, the whole section of a single processor will make butterfly operation with another processor. Because of the ditribution of memory of calculation, MPI can perform FFT of very large sequence like $n = 10^9$ within an edurable time.
For one butterfly operation, two processors need to send their sequence to each other and use the both sequence to calculate. If we simply use block send and receive function, a deadlock will occur. One way to solve this is to use buffer send and recieve, which requires more time and memory. Another way is to use sendrecv which is special designed for this situation.

```
inline void butterfly(cpx* x,cpx* tmp,int pa,int pb,
long long i,long long j,int f) {
        int counterpart=id==pa?pb:pa;cpx w;
        MPI_Sendrecv(x,len,MPI_C_DOUBLE_COMPLEX,
        counterpart,id,tmp,len,MPI_C_DOUBLE_COMPLEX,
        counterpart,counterpart,MPI_COMM_WORLD,
        MPI_STATUSES_IGNORE);
        for(int k=0;k<len;k++){
                w=omega(j+k,i,f);
                if(id==pa) x[k]=x[k]+w*c[k];
                else if(id==pb) x[k]=c[k]-w*x[k];
        }
}
```

Figure 11: One butterfly operation with MPI

5.2 Bitwise Reverse

In MPI with ditributed memory, data swaps depend on communication among processors. If we use (Figure. 3) for the reverse of the whole sequence, communication cost will be a disaster for the program.
In *Zhang's book*[3], there is a lemma decribing the princple.

Lemma 5.1 *Suppose* $len = n/p$, $B = \{b_0, b_1 ... b_{n-1}\}$ *is the bitwise reverse of the whole sequence,* $B_i = \{b_{i*len}, ..., b_{(i+1)*len-1}\}$, $v_{ij} = |\{b : b \in B_i, bs = j\}|$, $i, j = 0, 1 ... p-1$ *and* n, p *satisfy Assumption 5.1, we can get:*
(1) $v_{ij} = s/p$
(2) If $b_k/s = j$, *then* $b_{k+p}/s = j$
(3) First p elements in B_i/s *is bitwise reverse of* $0, 1, ..., p-1$
(4) In B_i, *sequence of* $b_{ij}/s = k$ *can be acquired by zoom and translation of bitwise reverse of* $0, 1, ..., s/p-1$

When it comes to the implement of the bitwise reverse, each processor need to send its specific part to all processors, which suits the all-to-all collective communication.

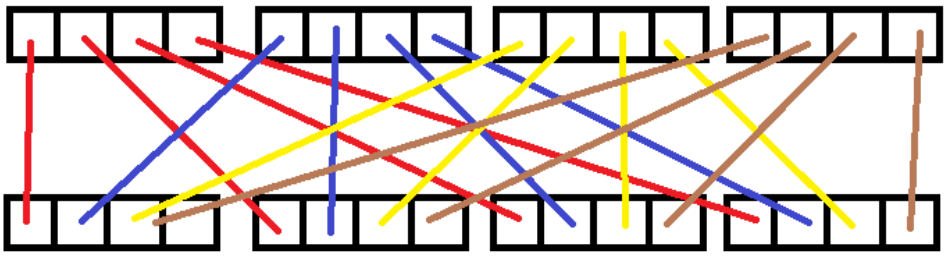

Figure 12: All to all

```
void Bitwise(cpx *aa, cpx *tmp) {
        int zoom = len / sz;
        for (int i = 0; i < len; i++)
                tmp[zoom*p_pos[i%sz] + i/sz] = aa[i];
        MPI_Alltoall(tmp, zoom, MPI_C_DOUBLE_COMPLEX, aa, zoom,
         MPI_C_DOUBLE_COMPLEX, MPI_COMM_WORLD);
}
```

Figure 13: MPI_Alltoall

5.3 Results

Though extra communication costs always exist, the MPI program can keep an acceleration over 5 and approach $p/2$ when $n = 2^{27}$, but it does not mean linear acceleration cannot be achieved. When n is bigger than $n = 2^{27}$, the runtime of serial program is too long to be measured while the MPI program can finish within 10s with 64 processors. Data bigger than 2^{31} is not tested here because the memory server is not enough - though the memory complexity is $\Theta(n)$, n itself is too large under that circumstance.

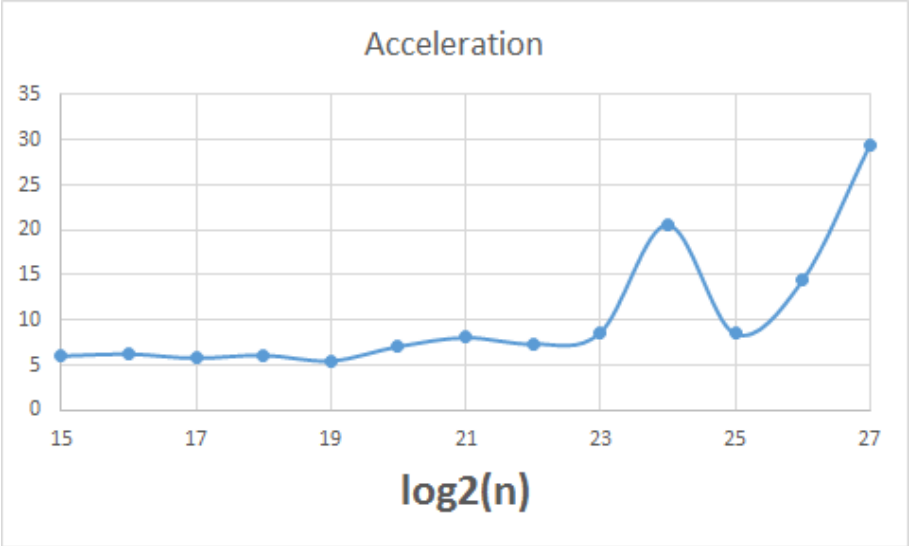

Figure 14: Accleration vs $log_2 n$

n	p=4(s)	p=8(s)	p=16(s)	p=32(s)	p=64(s)	p=128(s)
268435456	146	119	173	132	9.98	12.4
536870912	303	235	361	412	9.98	12.4
1073741824	631	503	650	830	9.98	12.4

Table 3: MPI results

References

[1] *cuFFT :: CUDA Toolkit Documentation*.

[2] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Prallel Computing, Second Edition*. Addison Wesley, January 16, 2003.

[3] L. Zhang, X. Chi, Z. Mo, and R. Li. *Introduction to Prallel Computing*. Tsinghua University, Beijing, 2006.