1 Introduction

The *mult* application is for multiplying two large decimal numbers with reasonable speed. The program gives accurate results up to several megabytes long numbers in less than a minute on a fast computer. This is achieved as follows: First we look at the numbers as a polynomials (which is evaluated at 10). Then we apply Discrete Fourier Transform to convert the polynomial to point-value representation. In this representation we can multiply two polynomials in linear time. After the multiplication step we convert the result back to coefficient representation, propagate the carry and output the result. The slowest part of the algorithm is the representation conversion which takes $O(n \log n)$ time thus making the whole algorithm's time complexity $O(n \log n)$ where n is the number of digits of the result.

2 Code structure

There will be two source files: `mult.c` and `fft.asm`.

2.1 `mult.c` file's structure

1 ⟨*mult.c* 1⟩≡

```
#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

⟨structs, globals and externs 3a⟩
⟨helper functions 9a⟩
⟨main function 10a⟩
```

2.2 `fft.asm` file's structure

This file will contain only the FFT algorithm and nothing else.

2 ⟨*fft.asm* 2⟩≡
  ⟨*helper macros* 8a⟩
```
section .data
```
  ⟨*data section* 6c⟩
```
section .text
```
  ⟨*symbol definitions* 3c⟩
  ⟨*iterative-fft function's body* 4a⟩

3 Fast Fourier Transform

This algorithm will be not explained in details because it is just a simple implementation taken as is from the book Introduction To Algorithms. The pseudocode described in the book goes as follows:
d - the direction of the transformation (it's 1.0 or -1.0)
n - the size of the arrays, power of two
A - an input array of n complex numbers
Y - an output array of n complex numbers, the DFT of A

Iterative-FFT(A, Y, n, d)
  Bit-Reverse-Copy(A, Y, n)
  for $s \leftarrow 1$ to $\lg n$ do
    $m \leftarrow 2^s$
    $\omega_m \leftarrow e^{d \cdot 2\pi i/m}$
    for $k \leftarrow 0$ to $n-1$ by m do
      $\omega \leftarrow 1$
      for $j \leftarrow 0$ to $m/2-1$ do
        $t \leftarrow \omega Y[k+j+m/2]$
        $u \leftarrow Y[k+j]$
        $Y[k+j] \leftarrow u+t$
        $Y[k+j+m/2] \leftarrow u-t$
        $\omega \leftarrow \omega\omega_m$

As you can see the algorithm uses complex numbers, so we have to define them. We will represent a complex number with two doubles. We add this structure to `mult.c` as follows:

3a ⟨*structs, globals and externs* 3a⟩≡
```
struct complex_num {
        double real;
        double imag;
};
```

We assume that each instance of this structure is aligned to 16 bytes, because we will be using SSE registers to hold these complex numbers. The low part of the register contains the real part and the high 8 bytes contain the imaginary part of the complex number. Now that we have the definition of a complex numbers we can define the prototype of our function in the `mult.c` file:

3b ⟨*structs, globals and externs* 3a⟩+≡
```
void iterative_fft(const struct complex_num *A,
        struct complex_num *Y, int n, double d);
```

We have to make the symbol visible from outside the source of `fft.asm`:

3c ⟨*symbol definitions* 3c⟩≡
```
global iterative_fft
```

We are now ready to implement the algorithm line by line. We first begin with the routine's entry and exit points:

4a ⟨*iterative-fft function's body* 4a⟩≡

```
iterative_fft:

        push ebp
        mov ebp, esp
        sub esp, 36
        push ebx
        push edi
        push esi

        push dword [ebp+16]
        push dword [ebp+12]
        push dword [ebp+8]
        call bit_reverse_copy
        add esp, 12

        ⟨fft's main loop 5⟩

        pop esi
        pop edi
        pop ebx
        mov esp, ebp
        pop ebp
```

`bit_reverse_copy` has almost the same prototype as `iterative_fft` (just without the direction parameter) and is implemented as a helper function in `mult.c`, so we have to add it as an external symbol:

4b ⟨*symbol definitions* 3c⟩+≡

```
extern bit_reverse_copy
```

Inside the routine the stack has the following structure:

ebp+20	d
ebp+16	n
ebp+12	Y
ebp+ 8	A
ebp+ 4	return address
ebp	old ebp
ebp-32	32 byte long scratch area

The registers are used as follows:

eax	scratch register
ebx	scratch register
ecx	j
edx	k
esi	m
edi	n
xmm0	w_m
xmm1	w
xmm2	t
xmm3	u
xmm4-7	scratch registers

In the main loop we calculate m's value from its previous value by doubling it. The $s \leq \lg n$ part is replaced with $m \leq n$ (because $m = 2^s$). We don't calculate s variable's value because it is not used.

5 ⟨*fft's main loop* 5⟩≡

```
        mov edi, [ebp+16]
        mov esi, 2
.main_loop_begin:
        cmp esi, edi
        jg .main_loop_end

        ⟨w_m ← e^{d·2πi/m} 6a⟩
        ⟨fft's middle loop 6b⟩

        add esi, esi
        jmp .main_loop_begin
.main_loop_end:
```

We know that $e^{2\pi i/m} = \cos(d \cdot 2\pi/m) + i\sin(2\pi/m)$. To calculate this we use the FPU.

6a $\langle w_m \leftarrow e^{d \cdot 2\pi i/m}$ 6a$\rangle\equiv$

```
        mov [ebp-32], esi
        fldpi
        fldpi
        faddp
        fld qword [ebp+20]
        fmulp
        fild dword [ebp-32]
        fdivp
        fsincos

        lea eax, [ebp-16];
        and eax, 0xfffffff0
        fstp qword [eax]
        fstp qword [eax+8]
        movapd xmm0, [eax]
```

Note that we need to have the complex number aligned to 16 bytes, therefore we *binary and* the address where we temporarily store the result.

6b $\langle$*fft's middle loop* 6b$\rangle\equiv$

```
        mov edx, 0
  .middle_loop_begin:
        cmp edx, edi
        jge .middle_loop_end

        ⟨w ← 1 6d⟩
        ⟨fft's inner loop 7⟩

        add edx, esi
        jmp .middle_loop_begin
  .middle_loop_end:
```

To easily load one to w we store the constant 1 to the data section:

6c $\langle$*data section* 6c$\rangle\equiv$

```
  complex_one dq 1.0, 0.0
```

Now we just load this to w.

6d $\langle w \leftarrow 1$ 6d$\rangle\equiv$

```
        movapd xmm1, [complex_one]
```

In the comparison we need the value of $m/2$ so we divide m's value by two and after the comparison we restore it's original value. Also eax will contain the value of Y+(k+j)*16 and ebx the value of Y+(k+j+m/2)*16.

7 ⟨*fft's inner loop* 7⟩≡

```
        mov ecx, 0
        mov eax, edx
        shl eax, 4
        add eax, [ebp+12]
        lea ebx, [eax + esi*8]
.inner_loop_begin:
        shr esi, 1
        cmp ecx, esi
        jge .inner_loop_end
        shl esi, 1

        ⟨fft's inner loop's body 8b⟩

        add ecx, 1
        add eax, 16
        add ebx, 16
        jmp .inner_loop_begin
.inner_loop_end:
        shl esi, 1
```

Before we can write the inner loop's body, we must make macro for complex number multiplication. The multiplication takes 4 parameters. These parameters must be xmm registers, the first one is the destination, the 2. and 3. are the arguments and 4. is a scratch register. The macro destroys the value of the 4. register and stores the result in 1. register.

8a ⟨*helper macros* 8a⟩≡

```
%macro complex_multiply 4
        movapd %1, %2    ; 1 = [2r | 2i]
        movapd %4, %3    ; 4 = [3r | 3i]
        shufpd %1, %2, 3 ; 1 = [2i | 2i]
        shufpd %4, %3, 1 ; 4 = [3i | 3r]
        mulpd %4, %1     ; 4 = [3i*2i | 3r*2i]
        movapd %1, %4    ; 1 = [3i*2i | 3r*2i]
        subsd %1, %4     ; 1 = [0 | 3r*2i]
        subsd %1, %4     ; 1 = [-3i*2i | 3r*2i]
        movapd %4, %2    ; 4 = [2r | 2i]
        shufpd %4, %2, 0 ; 4 = [2r | 2r]
        mulpd %4, %3     ; 4 = [2r*3r | 2r*3i]
        addpd %1, %4     ; 1 = [2r*3r - 3i*2i | 2r*3i + 3r*2i]
%endmacro
```

It's not the most effective way to multiply, but it is enough for our purposes.

8b ⟨*fft's inner loop's body* 8b⟩≡

```
        movapd xmm4, [ebx]
        complex_multiply xmm2, xmm1, xmm4, xmm5
        movapd xmm3, [eax]
        movapd xmm6, xmm3
        movapd xmm7, xmm3
        addpd xmm6, xmm2
        subpd xmm7, xmm2
        movapd [eax], xmm6       ; A[k+j]     = u + t
        movapd [ebx], xmm7       ; A[k+j+m/2] = u - t
        complex_multiply xmm4, xmm0, xmm1, xmm5
        movapd xmm1, xmm4        ; w = w*wn
```

The last remaining part for our FFT routine is the `bit_reverse_copy` routine. We will define it in the `main.c`. But for the bit reverse copy we will need a helper function called `rev`. This function takes two parameters: k and n. The function will reverse the bits of k as if it were a $\lg n$ bit long number.

9a ⟨*helper functions* 9a⟩≡

```
int rev(int k, int n)
{
        int r = 0;
        n /= 2;
        while (n > 0) {
                r = r*2 + k%2;
                k /= 2;
                n /= 2;
        }
        return r;
}
```

Now we can define `bit_reverse_copy` as in the book:

9b ⟨*helper functions* 9a⟩+≡

```
void bit_reverse_copy(const struct complex_num *A,
        struct complex_num *Y, int n)
{
        int k;
        for (k = 0; k < n; ++k)
                Y[rev(k, n)] = A[k];
}
```

4 Multiplication

Now that we have the FFT routine, we can write our multiplication part. Let's start with the main function: we will have 7 variables. n will be the length of the working arrays and it will be a power of two. The result must be less than or equal to n digits. Then we have A, B, C arrays of complex numbers which represent numbers. A and B are the input numbers and the result will be in C. These numbers have their imaginary part 0 and the real part is an integer between 0 and 9. Then we have AA, BB, CC these will be the corresponding numbers DFT. So the main function has the following appearance:

10a ⟨*main function* 10a⟩≡

```
int main(int argc, char **argv)
{
        int i; /* general purpose local variable */
        ⟨main's local variables 10b⟩
        ⟨input 12⟩
        ⟨calculation 10c⟩
        ⟨output 15c⟩
        return 0;
}
```

We add to the local variables the variables described above. The arrays will be dynamically allocated by the IO when it determines n's value.

10b ⟨*main's local variables* 10b⟩≡

```
int n;
struct complex_num *A, *B, *C;
struct complex_num *AA, *BB, *CC;
```

Once we have the input we just convert the polynomials to their point-value representation, multiply them, convert the result back and carry propagate the result:

10c ⟨*calculation* 10c⟩≡

```
⟨convert A and B to point-value representation 10d⟩
⟨point-value multiply AA and BB 11b⟩
⟨convert CC back to coefficient representation 11a⟩
⟨carry propagate C 11d⟩
```

The conversion to the point-value representation is just taking the DFT of the numbers:

10d ⟨*convert A and B to point-value representation* 10d⟩≡

```
iterative_fft(A, AA, n, 1.0);
iterative_fft(B, BB, n, 1.0);
```

The conversion back is very similar, we just have to normalize after the DFT:

11a ⟨*convert CC back to coefficient representation* 11a⟩≡
```
iterative_fft(CC, C, n, -1.0);
for (i = 0; i < n; ++i) {
        C[i].real /= n;
        C[i].imag /= n;
}
```

We know that $CC(x) = AA(x)BB(x)$. This is true for every point, so:

11b ⟨*point-value multiply AA and BB* 11b⟩≡
```
for (i = 0; i < n; ++i) {
        /* we have to do complex multiplication */
        double a = AA[i].real, b = AA[i].imag;
        double c = BB[i].real, d = BB[i].imag;
        CC[i].real = a*c - b*d;
        CC[i].imag = a*d + b*c;
}
```

We don't just carry propagate C but also round its coefficients. Note that we are carry propagating only the real part because the imaginary part is not important and its value should be zero if calculated with infinite precision.

11c ⟨*main's local variables* 10b⟩+≡
```
double carry;
```

11d ⟨*carry propagate C* 11d⟩≡
```
carry = 0.0;
for (i = 0; i < n; ++i) {
        C[i].real = floor(C[i].real + carry + 0.5); /* round */
        carry = floor(C[i].real / 10.0);
        C[i].real = fmod(C[i].real, 10.0);
}
```

5 Notes

Note that we are using base 10 for our calculations whereas we could have use base 100, 1000 or even more. We do this because of precision issues. For example if we have to multiply two 10^7 long numbers in the form of 9...9 * 9...9 then in one position 10^7 of carry is accumulated which is fine because doubles have accurate precision around 15 digits and 10^7 is only 7 digits. But if we have used 100 as base the number of digits in the accumulated carry in the same case would be $2 \cdot \log_{10}(10^7/2) = 13.4$. If we add numerous precision issues in the FFT we would get incorrect results so we stay at base 10 even though it is slower. (Using base 100 instead of 10 would give four times faster program, using 1000 would give 9 times faster program).
The other thing to note that this application is very memory hungry. We represent each digit with 16 bytes and allocate 6 such arrays. The result is usually twice as big as the input so we need $16 \cdot 6 \cdot 2 \cdot n \approx 200n$ bytes of memory. For example to multiply two 10^7 long numbers we need 3 GB of memory! Fortunately the machine on which this has been tested has 4 GB of memory.

6 Input/Output

To extract a number from an input file we just read it's digits as long as we can. When we find a non-digit character we stop reading the given file. If couldn't read a single character then we assume that the file has 0 as a number.
To use the program the user needs to supply two filenames which from the numbers will be read. We then check how long are the numbers and set n to a suitable value. After that we allocate the arrays and read the input in.

12 ⟨*input* 12⟩≡

```
if (argc < 3) {
        fprintf(stderr, "Usage: %s filename1 filename2\n", argv[0]);
        exit(1);
}
⟨determine lengths 13c⟩
⟨determine n's value 14a⟩
⟨allocate the arrays 14c⟩
⟨read the files 15b⟩
```

For determining the length of a number in a file we will use a helper function:

13a ⟨*helper functions* 9a⟩+≡
```
int get_size(const char *fname)
{
        FILE *f;
        int sz;

        f = fopen(fname, "r");
        if (f == 0) {
                fprintf(stderr, "Couldn't open %s!\n", fname);
                exit(1);
        }
        for (sz = 0; sz >= 0 && isdigit(getc(f)); ++sz)
                ;
        if (sz < 0) {
                fprintf(stderr, "File %s is too long!\n", fname);
                exit(1);
        }
        fclose(f);
        return sz;
}
```

Now we can fill this into the main function:

13b ⟨*main's local variables* 10b⟩+≡
```
int len1, len2;
```

13c ⟨*determine lengths* 13c⟩≡
```
len1 = get_size(argv[1]);
len2 = get_size(argv[2]);
```

To determine n's value we define a helper function called `next_two_power` which takes t as a parameter and returns the closest strictly greater power of two.

13d ⟨*helper functions* 9a⟩+≡
```
int next_two_power(int t)
{
        int k = 1;
        while (t > 0) {
                k *= 2;
                t /= 2;
        }
        return k;
}
```

Armed with this function we can determine n value as follows: If we know that the number of digits in the result will be less than or equal as len1+len2 then we can just find a power of two which is greater than or equal as that value:

14a ⟨*determine n's value* 14a⟩≡

```
n = next_two_power(len1 + len2 - 1);
if (n < 0) {
        fprintf(stderr, "The resulting number is too big!\n");
        exit(1);
}
```

Note that the next power of two might not fit into an integer.
Instead of calling the memory allocator six times we allocate everything with one allocation and set the pointers into this big allocation. We also make sure that the arrays are aligned to 16 bytes and that they are zero initialized.

14b ⟨*main's local variables* 10b⟩+≡

```
char *data;
```

14c ⟨*allocate the arrays* 14c⟩≡

```
data = calloc(16, 6*n + 1);
A = (void*) ((long)(data + 15) & ~15);
B = A + n;
C = B + n;
AA = C + n;
BB = AA + n;
CC = BB + n;
```

To read the files we introduce a helper function (this function assumes that array passed to it is zero initialized):

15a ⟨*helper functions* 9a⟩+≡

```
void read_file(const char *fname, int sz, struct complex_num *D)
{
        int rd;
        FILE *f;

        f = fopen(fname, "r");
        if (f == 0) {
                fprintf(stderr, "Couldn't open %s!\n", fname);
                exit(1);
        }

        rd = 0;
        while (rd < sz) {
                int c;
                c = getc(f);
                if (c == EOF || !isdigit(c)) {
                        fprintf(stderr,
                                "The file %s has changed!\n",
                                fname);
                }

                rd += 1;
                D[sz-rd].real = c - '0';
        }
        fclose(f);
}
```

Note that the number is written backwards into the array because at the first index in the array is the least significant digit's value. We just use this function to read both files:

15b ⟨*read the files* 15b⟩≡

```
read_file(argv[1], len1, A);
read_file(argv[2], len2, B);
```

For output we just print C's value and deallocate the allocated arrays.

15c ⟨*output* 15c⟩≡

⟨*find the most significant nonzero digit's position* 16a⟩
⟨*print C's value from the most significant digit's position* 16b⟩
`free(data);`

16a ⟨*find the most significant nonzero digit's position* 16a⟩≡
```
for (i = n-1; i > 0 && (int) C[i].real == 0; --i)
        ;
```

16b ⟨*print C's value from the most significant digit's position* 16b⟩≡
```
for (; i >= 0; --i)
        printf("%0.0f", C[i].real);
puts("");
```

7 Test cases

You can find in the project's directory several test cases in the form n`a`, n`b`, n`c`. n`c` is the result of n`a`$\cdot n$`b`. You can test this in `bash` by executing for example `cmp 1c <(./mult 1a 1b)` command. If there is an error cmp will tell you, otherwise if everything is fine then nothing happens.

8 Makefile

17 ⟨*Makefile* 17⟩≡

```
all: mult.pdf mult

mult: mult.o fft.o
	gcc -m32 -o mult mult.o fft.o -lm

mult.o: mult.c
	gcc -c -o mult.o -m32 -g -Wall -W -ansi mult.c

mult.c: mult.nw
	notangle -L -Rmult.c mult.nw > mult.c

fft.o: fft.asm
	nasm -f elf -o fft.o fft.asm

fft.asm: mult.nw
	notangle -L'%%line %-1L %F%N' -Rfft.asm mult.nw > fft.asm

# output errors in the OR case by rerunning latex in the case of failure
# so when no error happened the output is clean
# pdflatex must be run twice, in order to generate and use the index it builds
mult.pdf: mult.tex
	pdflatex -halt-on-error -file-line-error mult.tex > /dev/null || \
	pdflatex -halt-on-error -file-line-error mult.tex | tail
	pdflatex -halt-on-error -file-line-error mult.tex > /dev/null
	rm -f mult.log mult.aux mult.out mult.toc

mult.tex: mult.nw
	noweave -delay -x mult.nw > mult.tex
```