

Project 1: Divide and Conquer Analysis Report

October 26, 2017

Sabbir Ahmed & Zafar Mamarakhimov

1 Description

A recursive, divide-and-conquer algorithm was developed and analyzed to multiply together lists of complex numbers. Two different multiplication methods were used to compute the same products to analyze the crossover point.

2 Crossover Point

The point at which the asymptotically better algorithm becomes faster is known as the *crossover point*.

2.1 Theoretical Results

Assume multiplication of two b -bit integers is $O(b^2)$ and addition of two b -bit integers is $O(b)$.

Consider, if $n = 2$, then computing the product of the list simply requires a single complex multiplication. For `cmu13()`, this means three multiplies of b -bit numbers, and five additions. There are some constants $c > 0$ and $d > 0$ such that the time to compute a b -bit multiply is bounded by cb^2 and the time for a b -bit addition is bounded by db . Therefore, the time to multiply the two complex numbers, considering only multiplications and additions, is bounded by $3cb^2 + 5db$.

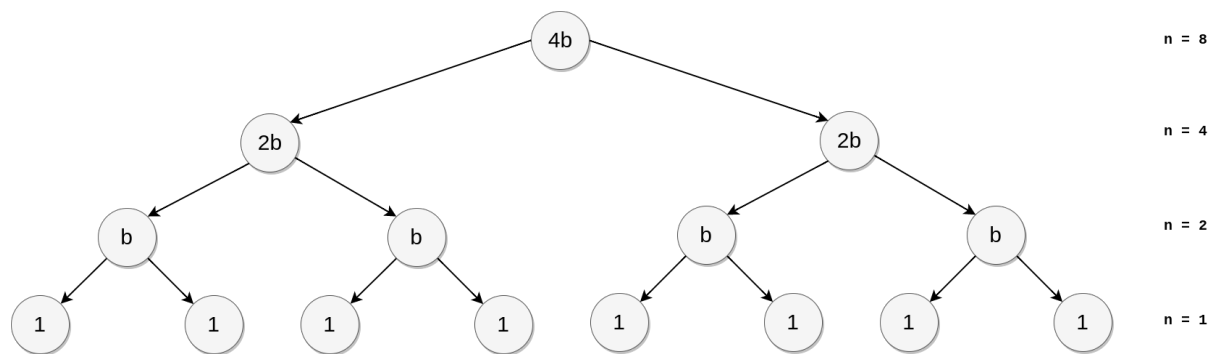


Figure 1: Tree Visualization of the Divide and Conquer Method of Multiplication. n is the Number of Integers Multiplied

According to the binary tree visualized, the number of bits grow by $\frac{n}{2}$, where n is the total number of integers being multiplied. The recurrence relation may be derived from this method:

$$\begin{aligned}
T(n) &= 3c \left(\frac{n}{2}b \right)^2 + 5d \left(\frac{n}{2}b \right) \\
&= \frac{3}{4}cb^2n^2 + \frac{5}{2}dbn
\end{aligned}$$

Asymptotically, the runtime may be computed using the recurrence relation as:

$$\begin{aligned}
&= \frac{3}{4}cb^2n^2 + \frac{5}{2}dbn \\
&\leq \frac{3}{4}cb^2n^2 + cb^2n^2 \\
&= \frac{5}{4}cb^2n^2
\end{aligned}$$

Continuing with $n = 2$, my `cmul4()` would make four b -bit multiplies and two b -bit additions, so its running time is bounded by $4cb^2 + 2db$.

3 Implementation

The project was written in C++11 and built with GCC v5.4.0. The GMP library, along with its C++ wrapper, GMPXX, were used to handle the multiprecision arithmetic. The recursive divide-and-conquer functions for both the three- and four-multiplication methods were implemented identically. The algorithm of the implementation is described in the pseudocode snippet provided in Algorithm 3.1.

Algorithm 3.1: Divide and Conquer Multiplication

```

1  /*
2  Uses a divide and conquer method to recursively multiply all the elements in
3  the complex array using either of the multiplication methods
4
5  Inputs:
6    - complex_array: vector of GMP integer pairs
7    - first , last: first and last indices of the subarray
8
9  Outputs:
10   - cmulx() outputs: final complex product
11  */
12  function cmulx_list(complex_array, first, last):
13
14      // if length of the array is 1
15      if (first == last):
16          return complex_array[first]
```

```
17
18     mid = (first + last) / 2
19     left_half = cmulx_list(complex_array, first, mid)
20     right_half = cmulx_list(complex_array, mid + 1, last)
21
22     return cmulx(left_half, right_half)
```

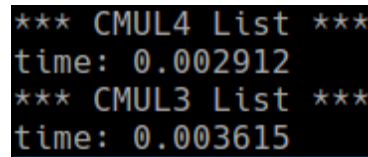
4 Testing and Timing

4.1 Platform Specifications

Before generating the statistics used in the document, information on the CPU and memory usage of the hosting machine were obtained. The following tables details the specifications captured before initializing the testing procedure.

4.2 Testing Methodology

The divide and conquer functions of individual multiplication methods were timed. An example of the output time is provided in Figure 2.



```
*** CMUL4 List ***
time: 0.002912
*** CMUL3 List ***
time: 0.003615
```

Figure 2: Sample Output of The Program Generated by cmplx_numbers_32_8bit.txt

The program, however, generates its output after a single iteration of the function call. The output times may vary on each execution due to the hosting machine fluctuating on its core and memory usage from other running processes. Therefore, further steps were taken to compute additional statistics on the timings.

The sample input data consisted of integers of 8, 10, 20, 30, 40, 100, and 200 bits divided into 32 and 64 repetitions. An additional Makefile has been provided to generate timings on multiple iterations. 200 iterations were used to compute the current statistics. Furthermore, each iteration recompiles before running the program. A clean compilation prevents any external influences and overhead

from the system on the individual timings. All the outputs for a single input are dumped to a buffer file, which is later parsed by an external helper script to generate the following statistics:

- Mean
- Element-to-element differences

4.3 Results

The distribution of the element-to-element differences were plotted against the size of the input data. Outliers were excluded when generating the boxplots. Data points residing above the x-axis indicate instances when `cmul4()` yielded slower runtimes than `cmul3()` (i.e. $\text{cmul4}() - \text{cmul3}() < 0$). The data points on the negative region represent instances when $\text{cmul4}() - \text{cmul3}() > 0$, indicating faster runtimes by `cmul4()` than `cmul3()`.

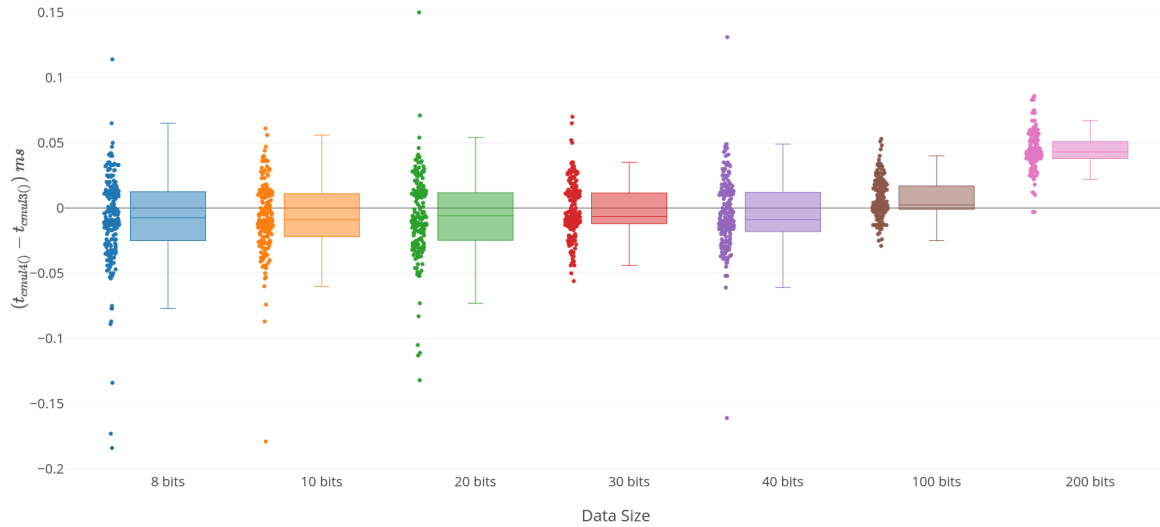


Figure 3: Distribution of Differences in The Multiplication Methods for $n=32$

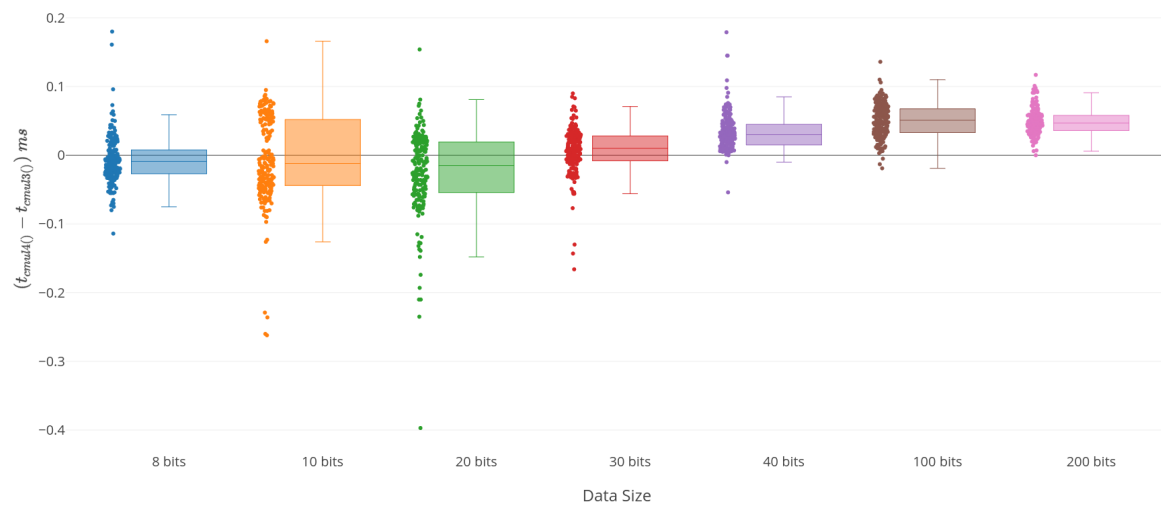


Figure 4: Distribution of Differences in The Multiplication Methods for n=64

Table 1: Information about the CPU Architecture, Generated by `$ lscpu`

| Component | Specification |
|----------------------|--|
| Architecture: | x86_64 |
| CPU op-mode(s): | 32-bit, 64-bit |
| Byte Order: | Little Endian |
| CPU(s): | 4 |
| On-line CPU(s) list: | 0-3 |
| Thread(s) per core: | 2 |
| Core(s) per socket: | 2 |
| Socket(s): | 1 |
| NUMA node(s): | 1 |
| Vendor ID: | GenuineIntel |
| CPU family: | 6 |
| Model: | 142 |
| Model name: | Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz |
| Stepping: | 9 |
| CPU MHz: | 1824.398 |
| CPU max MHz: | 3100.0000 |
| CPU min MHz: | 400.0000 |
| BogoMIPS: | 5423.89 |
| Virtualization: | VT-x |
| L1d cache: | 32K |
| L1i cache: | 32K |
| L2 cache: | 256K |
| L3 cache: | 3072K |
| NUMA node0 CPU(s): | 0-3 |

Table 2: Information about the Memory, Generated by `$ free -gh`

| Component | total | used | free | shared | buff/cache | available |
|-----------|-------|------|------|--------|------------|-----------|
| Mem: | 3.7G | 2.0G | 344M | 291M | 1.4G | 555M |
| Swap: | 7.8G | 1.1G | 6.7G | | | |