

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 $3cb^2 + 5db$.

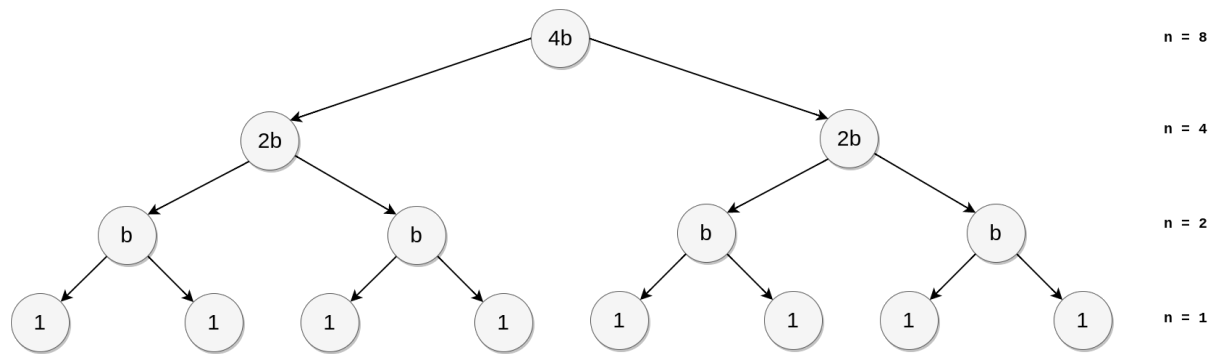


Figure 1: Tree Visualization of the Divide and Conquer Method of Multiplication. n is the Number of Integers Multiplied

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:

$$\begin{aligned}
f_1(n) &= 3c \left(\frac{n}{2}b \right)^2 + 5d \left(\frac{n}{2}b \right) \\
&= \frac{3}{4}cb^2n^2 + \frac{5}{2}dbn
\end{aligned}$$

Similarly, for `cmul4()` would make four b -bit multiplies and two b -bit additions, so its running time is bounded by $4cb^2 + 2db$. The recurrence relation may be derived from this method:

$$\begin{aligned}
f_2(n) &= 4c \left(\frac{n}{2}b \right)^2 + 2d \left(\frac{n}{2}b \right) \\
&= cb^2n^2 + 2dbn
\end{aligned}$$

Asymptotically, the runtime for either of the methods may be computed using the recurrence relation. Since the multiplication of n integers require $\frac{n}{2}f(b)$ computations:

$$= xb^2T\left(\frac{n}{2}\right), x > 0$$

Each levels of the tree comprises of $2^i b^2 n$, for $i = 0, 1, \dots, \lceil \lg n \rceil$. Therefore:

$$\begin{aligned}
\sum_{i=0}^{\lceil \lg n \rceil} 2^i x b^2 n &= x b^2 n \sum_{i=0}^{\lceil \lg n \rceil} 2^i \\
&= x b^2 n \left(\frac{2^{\lceil \lg n \rceil + 1} - 1}{2 - 1} \right) \\
&= x b^2 n \left(2^{\lceil \lg n \rceil + 1} - 1 \right) \\
&\leq x b^2 n \left(2^{\lceil \lg n + 1 \rceil + 1} - 1 \right) \\
&\approx x b^2 n (2^{\lg n} - 1) \\
&= x b^2 n (n - 1) \\
&= x b^2 n^2 - x b^2 n \\
&= \mathcal{O}(b^2 n^2)
\end{aligned}$$

Referring back to the multiplication functions, f_1 and f_2 can also be used to estimate the crossover points.

$$\begin{aligned}
 f_1(n) &= f_2(n) \\
 \frac{3}{4}cb^2n^2 + \frac{5}{2}dbn &= cb^2n^2 + 2dbn \\
 b &= \frac{6c}{d} \frac{1}{n} \\
 \Rightarrow b &\propto \frac{1}{n}
 \end{aligned}$$

Therefore, the bit size is inversely proportional to the number of multiplications when computing the crossover point. The larger the number of multiplications, the earlier the two methods will crossover.

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  function cmulx_list(complex_array, first, last):
2
3      // if length of the array is 1
4      if (first == last):
5          return complex_array[first]
6
7      mid = (first + last) / 2
8      left_half = cmulx_list(complex_array, first, mid)
9      right_half = cmulx_list(complex_array, mid + 1, last)
10
11     return cmulx(left_half, right_half)

```

The two multiplication methods were implemented using the pseudocode described in Algorithm 2 and Algorithm 3.

Algorithm 3.2: Four Multiplication

```

1  function cmul4(z1, z2):
2
3      pair(0, 1);
4      pair(0) = z1(0) * z2(0) - z1(1) * z2(1); // x1 * x2 - y1 * y2
5      pair(1) = z1(0) * z2(1) + z1(1) * z2(0); // x1 * y2 + y1 * x2
6
7      return pair;

```

Algorithm 3.3: Three Multiplication

```

1  function cmul3(z1, z2):
2
3      r, s = 0 // temporary variables
4      pair(0, 1);
5
6      r = z1(0) * z2(0); // x1 * x2
7      s = z1(1) * z2(1); // y1 * y2
8
9      pair(0) = r - s;
10     pair(1) = (z1(0) + z1(1)) * (z2(0) + z2(1)) - r - s;
11
12     return pair;

```

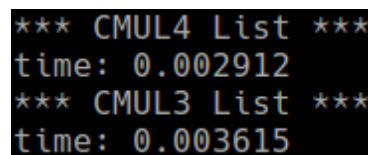
4 Testing and Timing

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_008bit.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

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			

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 box plots. 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()`.

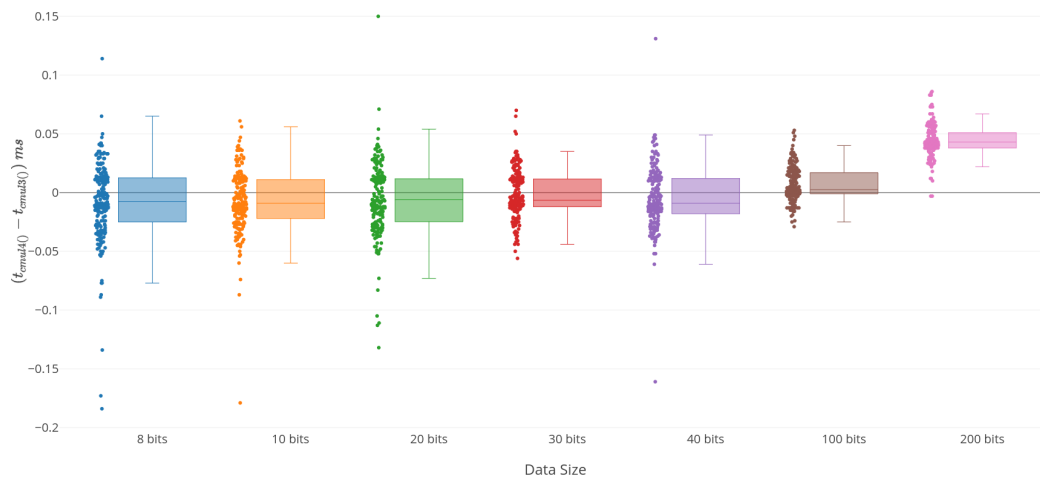


Figure 3: Distribution of Differences in The Multiplication Methods for $n=32$

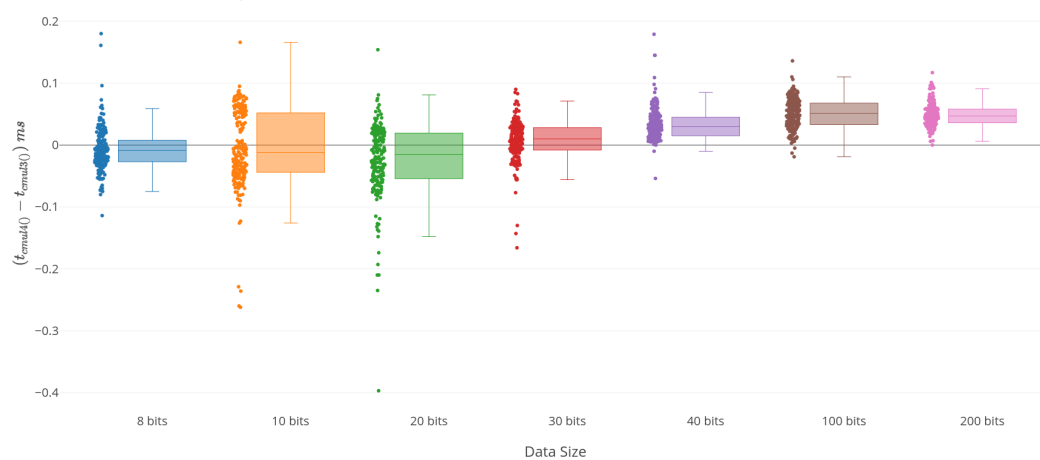


Figure 4: Distribution of Differences in The Multiplication Methods for $n=64$

The means of the times generated during testing were also plotted.

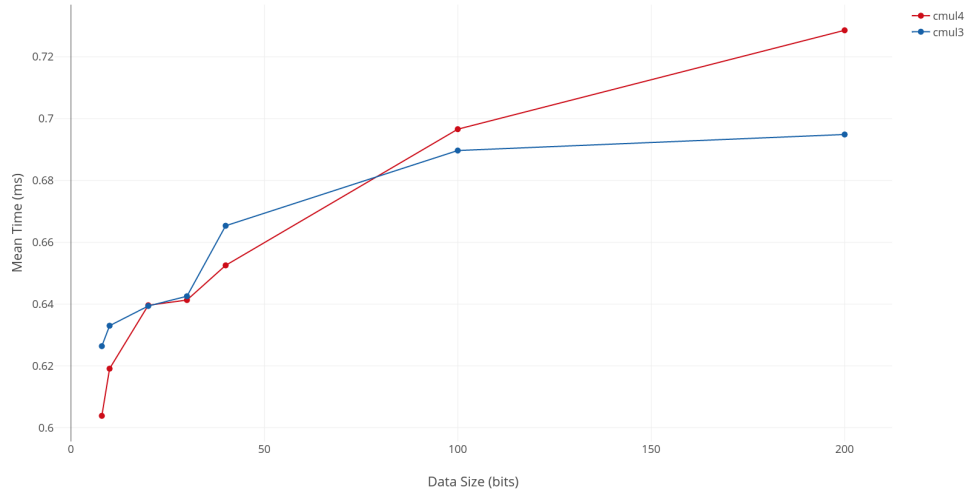


Figure 5: Means of The Multiplication Methods, with the Crossover Point ϵ [75, 100] bits for $n=32$

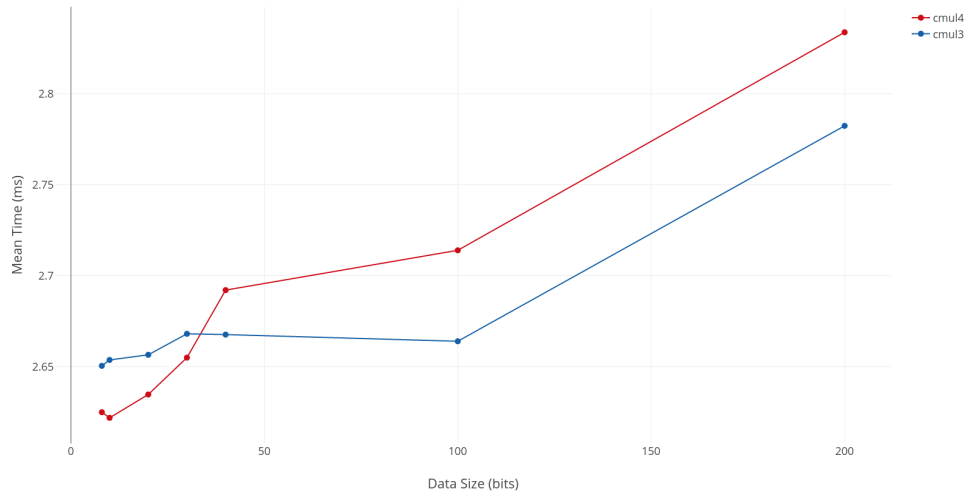


Figure 6: Means of The Multiplication Methods, with the Crossover Point ϵ [40, 60] bits for $n=64$

4.4 Troubleshooting

In the initial design of the source code, the execution times generated were being skewed by the implementation of the overloaded operators in the C++11 wrapper. The wrapper utilized an additional functionality that enabled the

compiler to control the number of cores being used. Since the data were generated using a machine with 4 CPUs, this functionality appeared to generate faster execution times for larger input data. This functionality was removed from the system libraries, and the data were regenerated.

These modifications did not require any changes to the source code since the initial design, but they were updated with minor modifications that do not affect the outputs.