A Comparison of Two Methods for Montgomery Multiplication Using C++

Oskar Asplin
asplin@umail.ucsb.edu

June 2017

Abstract

This paper will examine how to implement the classic Montgomery algorithm and the CIOS algorithm, and compare their runtime. Both algorithms will be implemented in the C++ programming language and tested using various sets of input parameters. The CIOS algorithm is stated to be more effective, so I will find out if this is true for my implementation, and to what degree they differ. Their runtime for unique sets of input will be compared and analyzed.

1 Introduction

Montgomery multiplication is a method for performing fast modular multiplication, and it was introduced by Peter L. Montgomery in 1985. It has made a huge difference in the world of binary computation and cryptographic engineering, by improving the efficiency of modular multiplication. The CIOS (Coarsely Integrated Operand Scanning) algorithm is another method of performing Montgomery multiplication, which is even more efficient. CIOS only requires 1.5 integer multiplications instead of 3 as the classic method needs.

In this project I have implemented the two algorithms in C++ in an attempt to further understand how they work, and test how they compare to each other in terms of runtime.

2 Implementation in C++

Before writing code, the two methods was first studied by reviewing professor Koç's course material in CS 293G, as well as Wikipedia and other academic sources online. This is the Pseudo code for the classic Montgomery multiplication:

function MonPro(a, b)
input: a, b, n, r, n'
output: $u = a \cdot b \cdot r^{-1} \bmod n$

1: $t \leftarrow a \cdot b$
2: $m \leftarrow t \cdot n' \pmod{r}$
3: $u \leftarrow (t + m \cdot n)/r$
4: **if** $u \geq n$ **then** $u \leftarrow u - n$
5: **return** u

It was fairly simple to code in C++, however I met several errors as I ran it with test input. By adjusting the code in order to fix the errors, I was able to make a solid code that can take any set of input parameters (a,b,n) of type int16 which returned the correct output u.

The Pseudo code for the CIOS method is as follows:

function MonPro(a, b)
input: a, b, n, r, n'
output: $u = a \cdot b \cdot r^{-1} \bmod n$
1: $u \leftarrow 0$
2: **for** $i = 0$ **to** $k - 1$
3a: $u \leftarrow u + A_i \cdot b$
3b: $u \leftarrow u + U_0 \cdot n$
3c: $u \leftarrow u/2$
4: **if** $u \geq n$ **then** $u \leftarrow u - n$
5: **return** u

The CIOS method was a bit easier to implement, but I could not resolve the problem of going bit by bit through A without doing a new byte multiplication. In hardware implementation of CIOS it is probably more effective than the classical method, but in my case, step 3 lead to more byte multiplications.

Both algorithms used the Extended Euclidean Algorithm which I made using a template I found online. Its Pseudo code is shown below:

$$\begin{aligned} (u, v) &\leftarrow \text{EEA}(a, n) \\ u \cdot a - v \cdot n &= 1 \\ a^{-1} &= u \pmod{n} \end{aligned}$$

The code for the two methods is added in section 5.

3 Results

After successfully implementing both Montgomery methods and testing them with numerous inputs to see they worked properly, I made a function to time them. 100

000 random input sets were created and then fed into the two Montgomery functions. Both algorithms were timed separately, and the test was performed 200 times.

The results of randomly generated int16-inputs are shown in figure 1 and 2. As seen in the figures, the classic method performed much better than CIOS, which was approximately 71% slower. The median time for the classic Montgomery was 314 μs, while the median for the CIOS was 539 μs. This means that a single product calculation takes about 314 ns and 539 ns for the two methods. In theory the CIOS should perform better at a hardware level as it needs fewer multiplications, but due to my implementation in a higher level language (C++), the CIOS algorithm was not able to perform as fast as it could. I believe the multiplications at step 3 in the algorithm were performed as byte multiplications instead of going through A bitwise. I tried different ways of performing the third step, but was not able to improve the runtime.

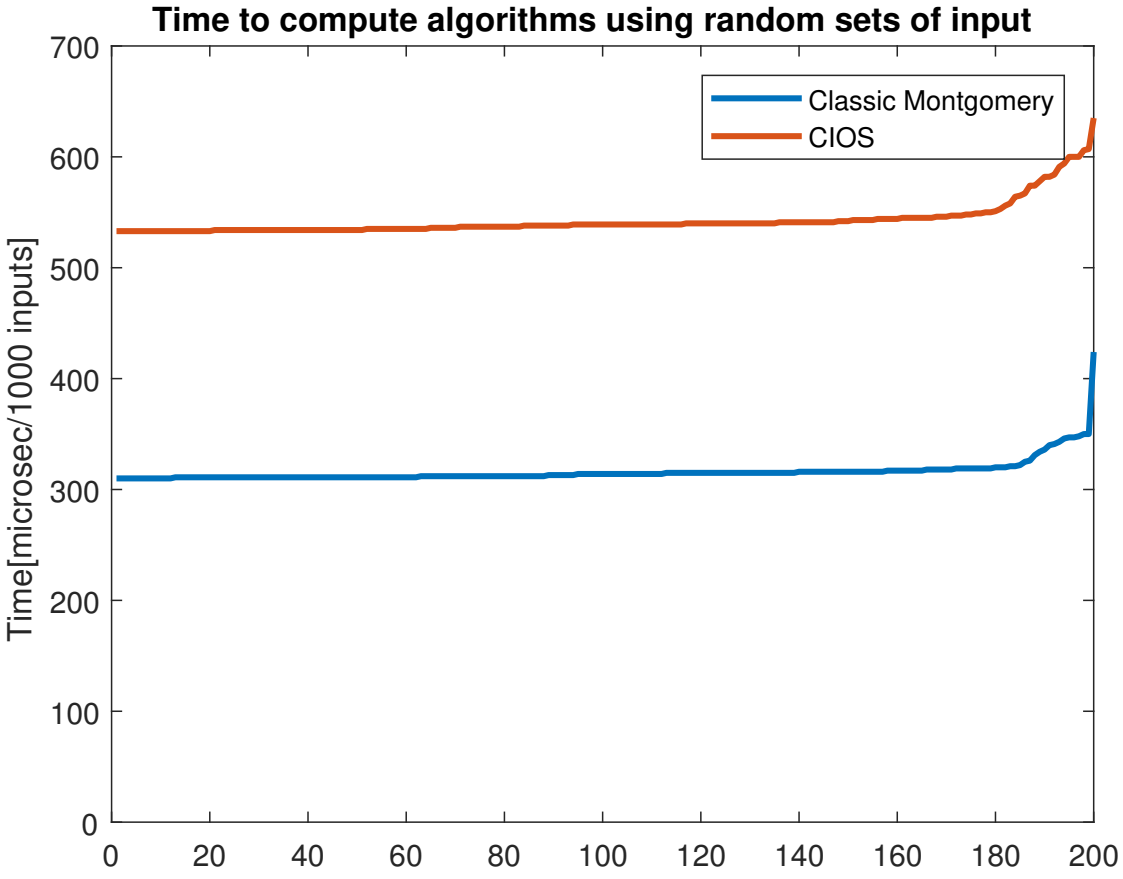

Figure 1: Plot showing result of 200 tests with 100 000 random inputs

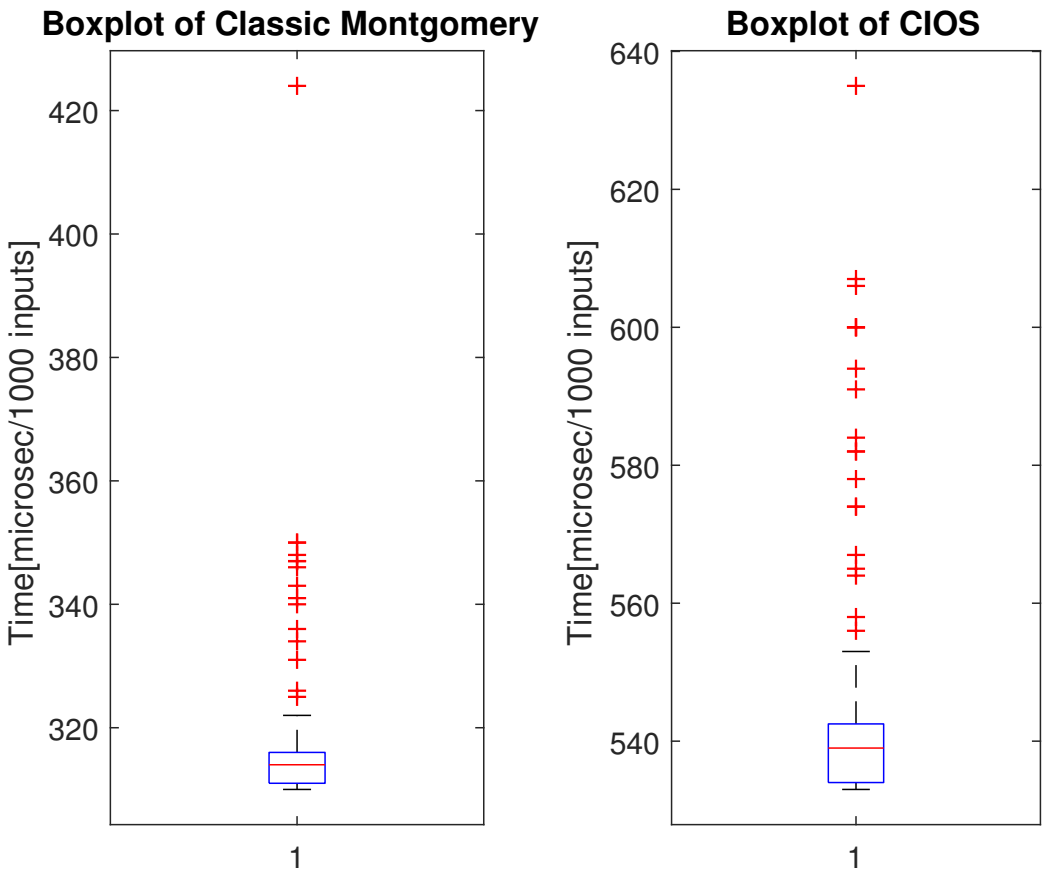

Figure 2: Boxplot showing result of 200 tests with 100 000 random inputs

When bounding the input to numbers up to 256 (int8), both methods performed better, and the CIOS method improved significantly. The results can be seen in figure 3 and 4. The classic method now had a median of 265 μs and CIOS had a median of 333 μs. Though CIOS is still 26% slower, its performance improved drastically. Again I believe this is due to step three in the algorithm, which now has to perform fewer multiplications. The number of operations for the classic method is constant in both cases.

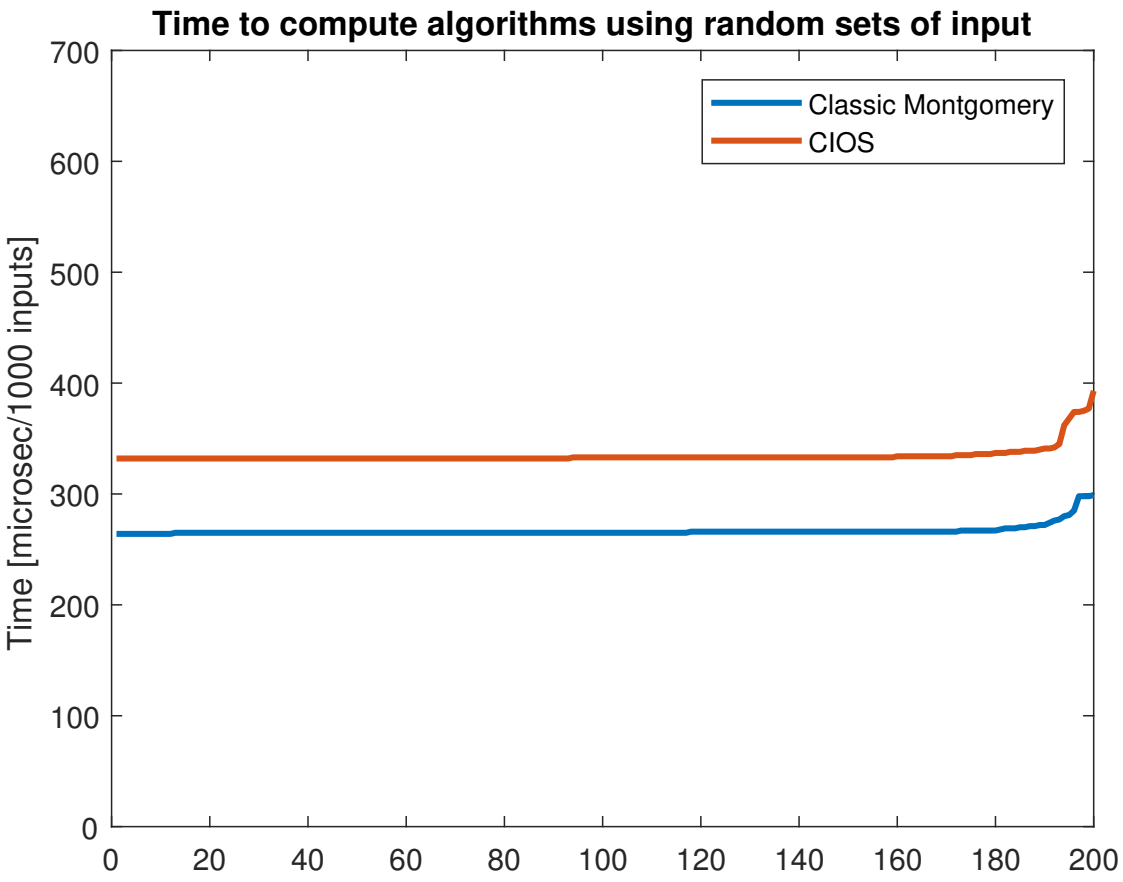

Figure 3: Plot showing result of 200 tests with 100 000 random int8-inputs

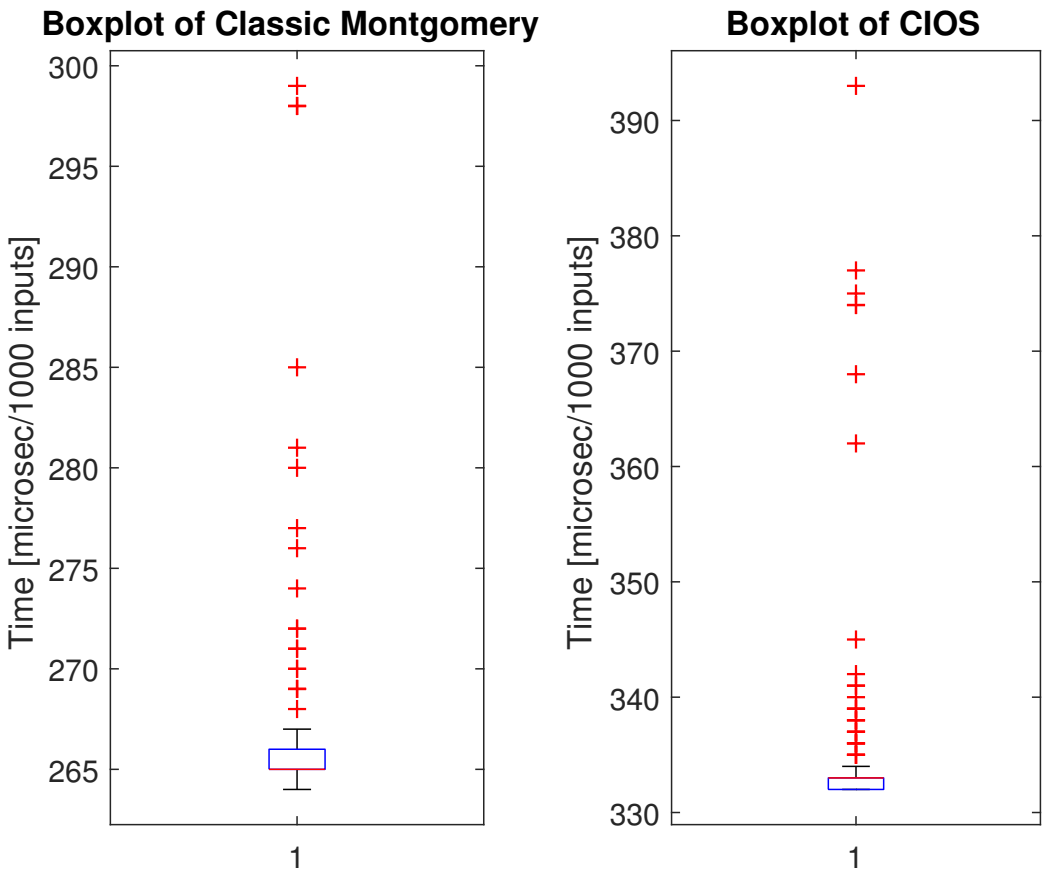

Figure 4: Boxplot showing result of 200 tests with 100 000 random int8-inputs

All testing was done on a computer with an Intel Core i7-4500U processor with 1.8-3GHz performance.

4 Conclusion

This project has taught me how two different methods of performing Montgomery multiplication work in theory, and how they can be implemented using C++. As stated in the results, the CIOS is more effective if properly implemented in a lower programming language or at hardware level, however I was not able to achieve this ideal use of the algorithm. Therefore we can see significantly better results for the classic method in my C++ coding. Since I have little knowledge of how the compiler breaks down my written code, I do not know the exact number of multiplications or the exact execution in the compiled program. If one were to perfectly compare the difference between the two algorithms, one would have to code in an assembly language or know how the compiler operates.

5 C++ code

5.1 monPro.cpp

```cpp
#include <iostream>
using namespace std;
#include <bitset>
#include <stdio.h>
#include <math.h>       /* pow */
#include <stdlib.h>     /* srand, rand */
#include <sys/time.h>   /* gettimeofday, timeval */
#include <algorithm>    /* sort, begin, end */

#define NUM_INPUTS 100000
#define NUM_SAMPLES 200
#define INT16_SIZE 65536

// ax + by = gcd(a, b)
int gcdExtended(int a, int b, int *x, int *y)
{
    // Base Case
    if (a == 0)
    {
        *x = 0;
        *y = 1;
        return b;
    }

    int x1, y1; // To store results of recursive call
    int gcd = gcdExtended(b%a, a, &x1, &y1);

    // Update x and y using results of recursive call
    *x = y1 - (b/a) * x1;
    *y = x1;

    return gcd;
}

// Find position of the MSB in Int16
int msbPosInt(int a)
{
    int bitCompare = 65536; // bit value: 1000000000000000
    int pos = 16;
    int b = a & bitCompare;
    while (b == 0)
    {
        bitCompare = bitCompare >> 1;
        pos -= 1;
        b = a & bitCompare;
    }
    return pos;
}

```

```
50 int cios(int a, int b, int n)
51 {
52     int k = msbPosInt(n)+1;
53     int r = pow(2, k);
54     bitset<sizeof(int)*8> a_bits(a);
55     long u = 0;
56 
57     for (int i = 0; i < k; i++)
58     {
59         // Step 3a
60         u = u + a_bits[i]*b;
61         // Step 3b
62         u = u + (u & 1) * n;
63         // Step 3c
64         u = u >> 1;
65     }
66     // Step 4
67     if (u >= n)
68     {
69         u = u - n;
70     }
71     return u;
72 }
73 
74 int classicMonPro(int a, int b, int n)
75 {
76     int k = msbPosInt(n)+1;
77     int r = pow(2, k);
78     int n_dot, x;
79     unsigned int t;
80     long u, m;
81 
82     gcdExtended(r, n, &x, &n_dot);
83     if (x < 0) {
84         x = n + x;
85         n_dot = n_dot - r;
86     }
87     n_dot = -n_dot;
88 
89     // Step 1
90     t = a*b;
91 
92     // Step 2
93     m = t*n_dot;
94     m = m & (r-1);
95 
96     // Step 3
97     u = (t + m*n);
98     int msbR = msbPosInt(r); // divide by r
99     u = u >> msbR;
100 
101     // Step 4
102     if (u >= n){
103         u = u - n;
```

```
104     }
105
106     // Step 5
107     return u;
108 }
109
110 int main()
111 {
112     int u, v;
113     struct timeval tvalBefore, tvalAfter;
114     long classicTime[NUM_SAMPLES], ciosTime[NUM_SAMPLES];
115
116     /* initialize random seed: */
117     srand (time(NULL));
118
119     int* n = new int[NUM_INPUTS];
120     int* a = new int[NUM_INPUTS];
121     int* b = new int[NUM_INPUTS];
122
123     // Begin testing
124     /*****************************************************************/
125     for (int test_x = 0; test_x < NUM_SAMPLES; test_x++)
126     {
127         for (int i = 0; i < NUM_INPUTS; i++)
128         {
129             n[i] = (rand() % (INT16_SIZE-1)/2)*2 + 1; // Need an odd
    number for gcd(r, n) = 1
130             a[i] = rand() % n[i];
131             b[i] = rand() % n[i];
132         }
133
134         // Record time for NUM_INPUTS of the classic montgomery
    algorithm
135         gettimeofday (&tvalBefore, NULL);
136
137         for (int i = 0; i < NUM_INPUTS; i++)
138         {
139             u = classicMonPro(a[i], b[i], n[i]);
140         }
141
142         gettimeofday (&tvalAfter, NULL);
143
144
145         classicTime[test_x] = ((tvalAfter.tv_sec - tvalBefore.tv_sec)
    *1000000L
146                 +tvalAfter.tv_usec) - tvalBefore.tv_usec;
147         // Divide by a scalar to get average time per 1000:
148         classicTime[test_x] /= 100L;
149
150         // Record time for NUM_INPUTS of the CIOS algorithm
151         gettimeofday (&tvalBefore, NULL);
152
153         for (int i = 0; i < NUM_INPUTS; i++)
154         {
```

```
155             v = cios(a[i], b[i], n[i]);
156         }
157 
158         gettimeofday (&tvalAfter, NULL);
159 
160         ciosTime[test_x] = ((tvalAfter.tv_sec - tvalBefore.tv_sec)
    *1000000L
161                 +tvalAfter.tv_usec) - tvalBefore.tv_usec;
162         // Divide by a ascalar to get average time per 1000:
163         ciosTime[test_x] /= 100L;
164     }
165     /*****************************************************************/
166     //End of tetsing
167 
168     /* Sort arrays */
169     sort(classicTime, classicTime + NUM_SAMPLES);
170     sort(ciosTime, ciosTime + NUM_SAMPLES);
171 
172     /* Print results */
173     cout << "Time in microseconds for the two algorithms per 1000 inputs
    :" << endl;
174     printf("classicMonPro min: %ld microseconds\nclassicMonPro max: %ld
    microseconds\n",
175             classicTime[0], classicTime[199]);
176     printf("CIOS min: %ld microseconds\nCIOS max: %ld microseconds\n",
    ciosTime[0], ciosTime[199]);
177 
178     /*
179     // Print all times
180     cout << "Classic Montgomery times:" << endl;
181     for (int i = 0; i < NUM_SAMPLES; i++)
182     {
183         cout << classicTime[i] << endl;
184     }
185 
186     cout << "CIOS times:" << endl;
187     for (int i = 0; i < NUM_SAMPLES; i++)
188     {
189         cout << ciosTime[i] << endl;
190     }
191     */
192 
193     /* Delete dynamic memory usage */
194     delete[] n;
195     delete[] a;
196     delete[] b;
197 
198     return 0;
199 }
```

Code/monPro.cpp