Asymmetric Squaring Formulae

Jaewook Chung* and M. Anwar Hasan
j4chung@uwaterloo.ca
ahasan@secure.uwaterloo.ca

Electrical and Computer Engineering
and
Centre for Applied Cryptographic Research
University of Waterloo
Ontario, Canada
August 3, 2006

Abstract. We present efficient squaring formulae based on the Toom-Cook multiplication algorithm. The latter always requires at least one non-trivial constant division in the interpolation step. We show such non-trivial divisions are not needed in the case two operands are equal for 3, 4, 5-way squarings. Our analysis shows that our 3-way squaring algorithms have much less overhead than the best known 3-way Toom-Cook algorithm. Our experimental results show that one of our new 3-term squaring methods performs faster than `mpz_mul()` in GNU multiple precision library (GMP) for squaring integers of 2880-6912 bits on Pentium 4 Prescott. For squaring in $\mathbb{Z}[x]$, our algorithms are much superior to other known squaring algorithms for certain range of input size. In addition, we present 4-way and 5-way squaring formulae which do not require any constant divisions by integers other than a power of 2. Under some reasonable assumptions, our 5-way squaring formula is faster than the recently proposed Montgomery's 5-way Karatsuba-like formulae.

1 Introduction

Multiplication is one of the most frequently used arithmetic operation in cryptography and the performance of a cryptosystem often depends mostly on the efficiency of a multiplication operation. Squaring is a special case of multiplication when two operands are identical and it is usually faster than multiplication, but not more than a constant factor.

Over the past four decades, many algorithms have been proposed to perform multiplication operation efficiently. Since Karatsuba discovered the first sub-quadratic multiplication algorithm [6], several innovations have been made on multiplication algorithms [13,3,14,10]. Unfortunately, none of these sub-quadratic multiplication algorithms has been considerably specialized for squaring. In this work, we attempt to fill this gap in the literature. It is not possible to have a squaring algorithm that is asymptotically better than the fastest multiplication algorithm. However, there are possibilities of some optimization by exploiting the fact that two operands are identical. We present three 3-way squaring formulae that are based on the Toom-Cook multiplication algorithm. Detailed methods for obtaining such formulae are presented. Experimental results show that our algorithms have advantages over other 3-way multiplication algorithms for certain range of operand sizes. We also present efficient 4-way and 5-way squaring formulae that are potentially useful in practice.

* This research was funded by Bell Canada Enterprises through the Bell 125th Anniversary Scholarship.

This report is organized as follows. In Section 2, we briefly review known multiplication algorithms. Then we discuss details on the Toom-Cook multiplication algorithm and discuss its issues in Section 3. In Section 4, we show in detail the methods for obtaining potentially better squaring algorithms than the Toom-Cook multiplication algorithm and present our new 3-way squaring formulae. Analysis and implementation results of our squaring algorithms are given in Sections 5 and 6, respectively. We present asymmetric formulae for 4-way and 5-way squaring in Section 7 and conclusions follow in Section 8

2 Review of Multiplication Algorithms

Multiplication is one of the most important basic arithmetic operations in popular public key cryptosystems. In this section, we briefly review some well-known multiplication algorithms. Since cryptographic computations must be exact and efficient, we focus only on the algorithms that compute such results using only integer arithmetic. Let $A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$ be in $\mathbb{Z}[x]$. The product of $A(x)$ and $B(x)$ is computed as follows:

$$C(x) = \sum_{i=0}^{2n-2} c_i x^i = A(x) \cdot B(x), \tag{1}$$

where $c_i = \sum_{j=0}^{i} a_j b_{i-j}$ for $0 \leq i \leq 2n-2$ and $a_j = 0$ and $b_j = 0$ for $j \geq n$ and $j < 0$. Let $L(\cdot)$ denote the set of all integral combinations of the coefficients of a polynomial. We call a computation of form "$a \cdot b$", where $a \in L(A)$ and $b \in L(B)$, a *coefficient multiplication*. The performance of multiplication algorithms are often analyzed in terms of the number of coefficient multiplications required to compute (1). The rest of the computational cost including the cost for computing the linear combinations $a \in L(A)$ and $b \in L(B)$ necessary to compute (1) is referred to as *overhead*. The multiplication $a \cdot b$ can be slower than computing $a_i \cdot b_j$, due to the carries occurring when computing the linear combinations a and b. We count the cost difference of two computations ($a \cdot b$ and $a_i \cdot b_j$) toward the overhead.

In order to compute (1) using paper and pencil, n^2 coefficient multiplications are required. Such a method is called the schoolbook multiplication method. When $A(x) = B(x)$, only $n(n+1)/2$ coefficient multiplications are required, since off-diagonal products (i.e., $a_i b_j$ where $i \neq j$) always occur twice and need to be computed only once. We call this squaring method the schoolbook squaring method.

The first multiplication algorithm that has sub-quadratic complexity was developed by Karatsuba in 1963. The Karatsuba algorithm (KA) performs the multiplication of two 2-term polynomials using only three coefficient multiplications as follows [6]:

$$C(x) = a_1 b_1 x^2 + ((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1)x + a_0 b_0. \tag{2}$$

The time complexity of $O(n^{\log_2 3})$ can be achieved by recursively applying (2). The KA is asymptotically better than the schoolbook method since $\log_2 3 = 1.58 < 2$. However, in real world applications, KA is faster than the schoolbook method only when n is sufficiently large, due to the fact that a larger amount of overhead is required in the KA than in the schoolbook method.

There is a well-known 3-term multiplication method which is shown below [1].

$$\begin{aligned}(a_2x^2 + a_1x + a_0)(b_2x^2 + b_1x + b_0) \\ &= D_2x^4 + (D_5 - D_2 - D_1)x^3 \\ &+ (D_4 - D_0 - D_2 + D_1)x^2 \\ &+ (D_3 - D_0 - D_1)x + D_0,\end{aligned} \tag{3}$$

Algorithm 1 Toom-Cook Multiplication Algorithm

Require: Degree $n-1$ polynomials $A(x)$ and $B(x)$.
Ensure: $C(x) = A(x) \cdot B(x)$.
1: (Evaluation) $u_i = A(x_i)$ and $v_i = B(x_i)$ for $i = 1, \ldots, 2n-1$, where x_i's are all distinct.
2: (Point-Wise Multiplication) $C(x_i) = u_i v_i$ for $i = 1, \ldots 2n-1$.
3: (Interpolation) given $C(x_i)$'s, uniquely determine c_j's for $j = 0, \ldots, 2n-2$, where $C(x) = \sum_{j=0}^{2n-2} c_j x^j$.

where

$$\begin{aligned} D_0 &= a_0 b_0, \quad & D_3 &= (a_0 - a_1)(b_0 - b_1), \\ D_1 &= a_1 b_1, \quad & D_4 &= (a_0 - a_2)(b_0 - b_2), \\ D_2 &= a_2 b_2, \quad & D_5 &= (a_1 - a_2)(b_1 - b_2). \end{aligned}$$

This formula requires 6 coefficient multiplications. In [9], Montgomery shows a family of 3-way multiplication algorithms requiring 6 coefficient multiplications. The method in (3) is a special case. Recursive use of (3) results in $O(n^{\log_3 6})$ time complexity. This method is less efficient than KA in asymptotic sense, but (3) is efficient when the input size is small and the input can be equally separated into three parts. We call (3) as 3-way KA-like formula.

In 1963, Toom developed an elegant idea to perform multiplication of two degree-$(n-1)$ polynomials using only $(2n-1)$ coefficient multiplications [13]. He showed that it is possible to construct a multiplication scheme that has $O(nc^{\sqrt{\log n}})$ operations and $O(c^{\sqrt{\log n}})$ delay. In 1966, Cook improved Toom's idea [3]. The multiplication method they developed is now called the Toom-Cook algorithm. The latter is based on a well-known theory from linear algebra: any degree-n polynomial can be uniquely determined by its evaluation at $(n+1)$ distinct points. Algorithm 1 shows a general idea how the Toom-Cook multiplication algorithm works.

Interestingly, many fast multiplication algorithms having sub-quadratic complexity are related to the Toom-Cook multiplication algorithm. In particular, KA can be considered as a special case of the Toom-Cook multiplication algorithm for the evaluation points $\{0, 1, \infty\}$, where evaluation at ∞ means computing $\lim_{x\to\infty} A(x)/x^{n-1}$ [14]. The Winograd algorithm is very similar to Algorithm 1. The difference is that the Winograd algorithm considers not only integers, but also imaginary numbers for evaluation points. Multiplication methods based on number theoretic transform (NTT) can be viewed also as special cases of the Toom-Cook algorithm. NTT based multiplication algorithms [10] use $x_i = \gamma^i \bmod p$ for $1 \le i \le N$, where γ is a primitive N-th root of unity modulo some prime $p \ge N$, where p is greater than or equal to the largest coefficient of the resulting polynomial, $N \ge 2n-1$ and $N|(p-1)$. In this case, some changes are required in Algorithm 1. Steps 1 and 2 must run through $i = 1$ to N, which may be greater than $2n-1$. Moreover, the computations must be performed in $\mathbb{Z}_p$. NTT based algorithms are asymptotically faster, since steps 1 and 3 can enjoy fast algorithms that requires $O(N \log N)$ operations in $\mathbb{Z}_p$ by choosing N having only small prime factors, or ideally a power of 2.

There are other efficient multiplication algorithms that cannot be derived from Algorithm 1. The 3-way KA-like formula shown in (3) and Montgomery's Karatsuba-like formulae [9] do not appear to be a special case of the Toom-Cook algorithm. Montgomery's formulae use 13, 17 and 22 coefficient multiplications for 5, 6 and 7-way polynomial multiplications, respectively.

For more comprehensive survey on multiplication algorithm, we refer the readers to Daniel Bernstein's paper [2].

Algorithm 2 Zimmerman's 3-Way Interpolation

Require: $(S_1, S_2, S_3, S_4, S_5)$ as in (4).
Ensure: $C(x) = A(x) \cdot B(x)$.

1: $T_1 \leftarrow 2S_4 + S_3$	$(= 18c_4 + 6c_3 + 6c_2 + 3c_0)$
2: $T_1 \leftarrow T_1/3$	$(= 6c_4 + 2c_3 + 2c_2 + c_0)$
3: $T_1 \leftarrow S_1 + T_1$	$(= 6c_4 + 2c_3 + 2c_2 + 2c_0)$
4: $T_1 \leftarrow T_1/2$	$(= 3c_4 + c_3 + c_2 + c_0)$.
5: $T_1 \leftarrow T_1 - 2S_5$	$(= c_4 + c_3 + c_2 + c_0)$
6: $T_2 \leftarrow (S_2 + S_4)/2$	$(= c_4 + c_2 + c_0)$
7: $S_2 \leftarrow S_1 - T_1$	$(= c_1)$
8: $S_3 \leftarrow T_2 - S_1 - S_5$	$(= c_2)$
9: $S_4 \leftarrow T_1 - T_2$	$(= c_3)$
10: **return** $C(x) = S_5x^4 + S_4x^3 + S_32x^2 + S_2x + S_1$.	

2.1 Zimmerman's 3-Term Toom-Cook Multiplication

This method has been developed by Zimmerman and implemented in GMP library as subroutines of `mpz_mul()`. Zimmerman uses $\{0, 1, 1/2, 2, \infty\}$ for the set of evaluation points.

Let $A(x) = a_2x^2+a_1x+a_0$, $B(x) = b_2x^2+b_1x+b_0$ and $C(x) = A(x)B(x) = c_4x^4+c_3x^3+c_2x^2+c_1x+c_0$. Evaluation of $A(x)$ and $B(x)$ at $x_i \in \{0, 1, 1/2, 2, \infty\}$ and point-wise multiplication of $A(x_i)$'s and $B(x_i)$'s result in the following system of equations:

$$\begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \end{bmatrix} = \begin{bmatrix} a_0b_0 \\ (a_2 + a_1 + a_0)(b_2 + b_1 + b_0) \\ (4a_2 + 2a_1 + a_0)(4b_2 + 2b_1 + b_0) \\ (a_2 - a_1 + a_0)(b_2 - b_1 + b_0) \\ a_2b_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix}. \tag{4}$$

Then the above linear system can be solved very efficiently using row operations as shown in Algorithm 2. The latter requires 8 additions/subtractions, 4 shifts, 1 division by 3. To the best of our knowledge this is by far the best method for performing the 3-term Toom-Cook multiplication.

3 Further Details on the Toom-Cook Multiplication Algorithm

In Section 2, we have reviewed various multiplication algorithms including the Toom-Cook multiplication algorithm. In this section, we look into details on the Toom-Cook algorithm, especially on its interpolation step. The interpolation step can be easily performed by using the Lagrange interpolation polynomial.

$$C(x) = \sum_{j=1}^{2n-1} C_j(x),$$

where

$$C_j(x) = C(x_j) \prod_{1 \le k \le 2n-1, k \ne j} \frac{x - x_k}{x_j - x_k}.$$

Alternatively, the Chinese remainder theorem (CRT) can be used. We can view the evaluation of a polynomial at point x_i as computing modulo a linear polynomial $(x - x_i)$, since computing $C(x) = A(x)B(x) \bmod (x - x_i)$ for $i = 1, \ldots, 2n-1$ is equivalent to computing $C(x_i)$'s. The CRT can combine the $(2n-1)$ distinct equivalence relations to compute the unique polynomial $C(x)$.

$$C(x) = \sum_{i=1}^{2n-1} C(x_i) M_i M_i',$$

where,

$$\begin{aligned} M &= \prod_{i=1}^{2n-1} (x - x_i), \\ M_i &= M/(x - x_i), \\ M_i' &= M_i^{-1} \bmod (x - x_i) = \frac{1}{\prod_{1 \le j \le 2n-1, i \ne j} (x_i - x_j)}. \end{aligned} \tag{5}$$

Even though the description of the CRT method looks quite different from the Lagrange interpolation method, the actual computation of the former is exactly the same as that of the latter, since $\prod_{1 \le k \le 2n-1, k \ne j} \frac{x - x_k}{x_j - x_k} = M_i M_i'$. Note that, in (5), M_i's and M_i''s can be pre-computed, for fixed x_i's.

By noticing that the interpolation step in Algorithm 1 solves a system of $(2n-1)$ linear equations with $(2n-1)$ unknown values (the coefficients of $C(x)$), we can construct the following linear system:

$$\begin{bmatrix} 1 & x_1 & \cdots & x_1^{2n-2} \\ 1 & x_2 & \cdots & x_2^{2n-2} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{2n-1} & \cdots & x_{2n-1}^{2n-2} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2n-2} \end{bmatrix} = \begin{bmatrix} C(x_1) \\ C(x_2) \\ \vdots \\ C(x_{2n-1}) \end{bmatrix}. \tag{6}$$

The $(2n-1) \times (2n-1)$ matrix on the left hand side of (6) is called the Vandermonde matrix. We denote it by V. A Vandermonde matrix has a known determinant, $D = \prod_{1 \le j < i \le 2n-1} (x_i - x_j)$. The system (6) is uniquely solvable, since x_i's are all distinct in Algorithm 1. Computing the inverse matrix can be pre-computed for fixed set of x_i's. Therefore, the coefficients c_i's can be easily obtained by multiplying the inverse matrix to the both sides of (6). Zuras uses this approach for 3, 4 and 5-way Toom-Cook multiplication algorithms [15]. However, this interpolation method is hardly useful in practice. It requires at most n^2 constant multiplications and at most n constant divisions for matrix-vector product. Among such constant divisions, at least one divisor must have an odd, non-trivial factor if $n > 2$.

Theorem 1. *There is no set of distinct integers $\{x_1, \ldots, x_s\}$'s such that $D = \det V$ is a power of 2; $D = \prod_{1 \le j < i \le 2n-1} (x_i - x_j) = \pm 2^k$ for some positive integer k, if $s > 3$.*

Proof. We need to show that there exists at least one pair (x_i, x_j), where $i \ne j$, such that $|x_i - x_j|$ is not a power of 2, if $s > 3$. We prove this by contradiction. We suppose we have a set of $s \ge 4$ distinct integers $\{x_1, \ldots, x_s\}$ such that D is a power of 2. Then, without loss of generality, x_2, x_3 and x_4 can be expressed as follows:

$$\begin{aligned} x_2 &= x_1 + (-1)^{u_1} 2^{v_1}, \\ x_3 &= x_1 + (-1)^{u_2} 2^{v_2}, \\ x_4 &= x_1 + (-1)^{u_3} 2^{v_3}, \end{aligned}$$

where u_i's are 0 or 1 and $v_i \in \mathbb{N} \cup \{0\}$. If $u_i = u_j$, then $v_i \neq v_j$. There are two cases:

1. All u_i's are the same: $u_1 = u_2 = u_3$.
   It follows that
   $$|x_2 - x_3| = |2^{v_1} - 2^{v_2}|,$$
   $$|x_3 - x_4| = |2^{v_2} - 2^{v_3}|,$$
   $$|x_4 - x_2| = |2^{v_3} - 2^{v_1}|.$$
   Note that v_i's must be distinct, since otherwise x_i's are not distinct. The value $|2^{v_1} - 2^{v_2}|$ can be a power of 2 if and only if $|v_1 - v_2| \leq 1$. But $v_1 \neq v_2$. Therefore, $v_1 = v_2 \pm 1$. Without loss of generality, we can let $v_1 = v_2 + 1$. Then there is no such v_3 that satisfies both $|v_3 - v_2| = \pm 1$ and $|v_3 - v_2 - 1| = \pm 1$. If $v_3 - v_2 = 1$, then $|v_3 - v_1| = |v_3 - v_2 - 1| = 0$. If $v_3 - v_2 = -1$, then $|v_3 - v_1| = 2$.
2. One of u_i's is different.
   Without loss of generality, we can let $u_1 = u_2 \neq u_3$. It follows that
   $$|x_2 - x_3| = |2^{v_1} - 2^{v_2}|,$$
   $$|x_3 - x_4| = |2^{v_2} + 2^{v_3}|,$$
   $$|x_4 - x_2| = |2^{v_3} + 2^{v_1}|.$$

Note that $v_1 \neq v_2$ and v_3 may be equal to either one of v_1 or v_2. It is easily seen that the value $|x_3 - x_4|$ is a power of 2 if and only if $v_2 = v_3$. Suppose $v_2 = v_3$. However, $|x_4 - x_2|$ can not be a power of 2, since v_3 must be different from v_1.

Theorem 2. *In the Toom-Cook multiplication algorithm, at least one non-trivial constant division must occur for* $n > 2$.

Proof. The $(2n-1)$-th row vector of V^{-1} is $(L_1, L_1, \ldots, L_{2n-1})$, where

$$L_i = \prod_{1 \leq j \leq 2n-1, j \neq i} \frac{1}{x_i - x_j}.$$

Hence, the inverse matrix V^{-1} must have entries whose denominators are factors of D. Moreover, L_i's have all the factors of D, since $D^2 = |\prod_{i=1}^{2n-1} L_i|$. Due to Theorem 1, the odd, non-trivial factor of D must divide at least one of $1/L_i$'s. Therefore, the Toom-Cook multiplication algorithm must have at least one non-trivial constant division in the interpolation step, for $n > 2$.

There are heuristic approaches for small n to reduce the number of constant divisions and its sizes as much as possible. Such methods perform elementary row operations on both sides of (6) until the system is solved, rather than multiplying the inverse matrix of V. For instance, Paul Zimmerman's implementation in GNU multiple precision library (GMP) v4.2.1 uses only one constant division by 3 for the 3-way Toom-Cook multiplication algorithm as shown in Section 2.1. We believe Zimmerman's method is by far the best 3-way Toom-Cook multiplication algorithm.

Even though there exist methods for fast exact division by a constant [5], [8], divisions by constants are very time consuming operation. Figure 1 shows the timing ratio of GMP's exact division by 3 to the fastest available large integer squaring. In GMP, exact division by 3 is implemented in the function `mpn_divexact_by3()` and `mpz_mul()` calls a squaring subroutine when it is called with

equal operands. When timing `mpz_mul()` and `mpn_divexact_by3()`, we used $3u$-bit and $(2u+6)$-bit operands, respectively. Note that, if the input size is $3u$-bit for the 3-way Toom-Cook multiplication algorithm, Algorithm 2 requires one exact division by 3 of at most $(2u+6)$-bit operand. We can easily observe that the exact division by 3 is very slow compared to the entire squaring operation for small operand sizes.

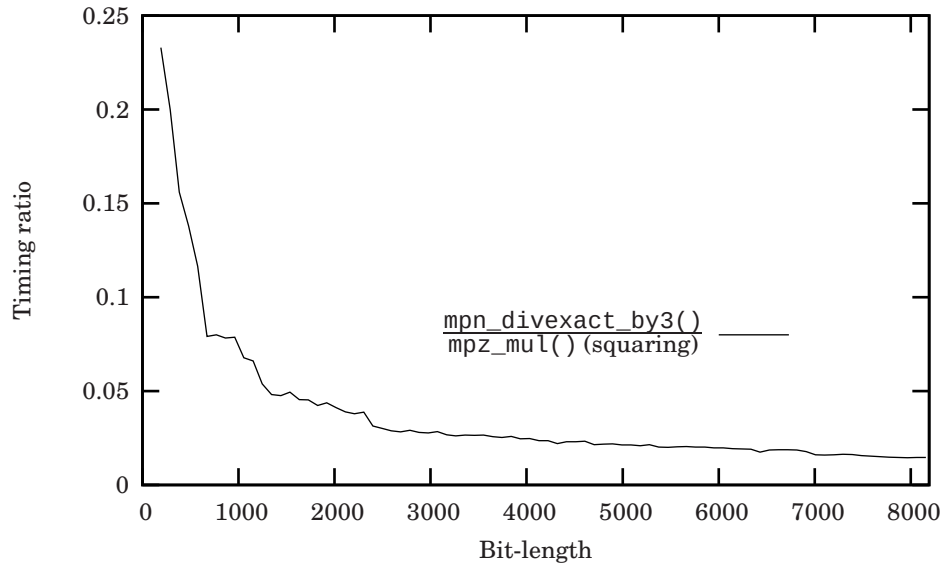

Fig. 1. Timing Ratio of `mpn_divexact_by3()` to `mpz_mul()`

The choice of evaluation points is very important for Algorithm 1, since it significantly affects the performance of the Toom-Cook multiplication algorithm. Toom and Cook proposed $x_i \in \{-n+1, \ldots, -1, 0, 1, \ldots, n-1\}$ and $x_i \in \{0, 1, 2, \ldots, 2n-2\}$, respectively. Knuth proposed the use of powers of 2 and their negatives [7], [2]. Winograd proposed ∞ as one of the evaluation points [14]. Winograd also noted that x_i's can be fractions, e.g., $x_i = p/q$, where evaluation at a rational point p/q means computing $q^{n-1}A(p/q)$. Note that it can be proven that the inclusion of ∞ and rational numbers for evaluation points does not change the fact that the determinant of the matrix in (6) must have factors other than 2 for $n > 2$. In 1994, Zuras proposed the use of reciprocally symmetric set, e.g., $x_i \in \{1, \infty, 0, 2, 1/2, -2, -1/2, \ldots\}$ [15]. Harley used the same evaluation points as Zuras ($\{1, \infty, 0, 2, 1/2\}$) in implementing the Toom-Cook multiplication routine in GNU multiple precision (GMP) arithmetic library version 4.1.4 [4]. He used simple row operations to solve the system (6) instead of multiplying an inverse matrix. He performed interpolation by using only one exact division by 3 for $n = 3$. Paul Zimmerman recently improved Harley's method using a simpler set: $x_i = \{0, 1, -1, 2, \infty\}$. This algorithm has been implemented in GMP 4.2.1 [4].

In [14], Winograd proves that Algorithm 1 uses the least possible number of coefficient multiplications. Unfortunately, the cost involved in evaluation and interpolation steps cannot be ignored even for small n. In fact, the evaluation and interpolation cost overwhelms the entire computation

time for multiplying polynomials having only small coefficients. To reduce the amount of overhead, Winograd proposes the use of remainder arithmetic by modulo cyclotomic polynomials, whose zeros are on unit circle in complex domain. For example, he proposed using $x, (x-1), (x+1)$ and (x^2+1) for 3-way multiplications. The Winograd algorithm can be viewed as the Toom-Cook algorithm for $x_i = \{0, 1, -1, j, -j\}$. However, this method needs one more coefficient multiplication than the Toom-Cook algorithm, since $A(x) \cdot B(x) \bmod (x^2+1)$ requires three coefficient multiplications. However, it has an advantage that there is no constant division during the interpolation step.

4 New Squaring Formulae

To the best of our knowledge, no sub-quadratic multiplication algorithms reviewed in Section 2 have been considerably specialized for squaring. We attempts to fill in this gap in the literature. Of course, there is no squaring algorithm which is asymptotically faster than the fastest multiplication algorithm [15] and it is not a goal of this work to find such squaring algorithms.

In Section 3, we have seen that non-trivial constant divisions in the Toom-Cook algorithm are unavoidable. There are multiplication algorithms not requiring the constant division, but they use more than $(2n-1)$ coefficient multiplications. Winograd shows methods for avoiding such constant divisions and reducing overhead in interpolation, but it is always at the sacrifice of an increased number of coefficient multiplications [14]. NTT based multiplication algorithms do not require non-trivial constant divisions if N is a power of 2, but this means that N must be greater than $2n-1$.

However, squaring can be performed without the non-trivial constant division using exactly $(2n-1)$ multiplications, at least for $n = 3, 4$ and 5. In this section, we present three potentially useful explicit formulae for 3-way squaring that do not require a non-trivial constant division. Our new squaring algorithms are similar to the Toom-Cook multiplication algorithm, but we use different approach for constructing a linear system on c_i's to achieve faster evaluation and interpolation. This new approach allows us to find squaring formulae that do not require any non-trivial constant divisions. Our squaring formulae require only the theoretic minimum number of coefficient multiplications, which is five for 3-way multiplication. We present only one explicit formula for each of 4-way and 5-way squaring in Section 7.

All sub-quadratic multiplication algorithms we have reviewed in Section 2 are *symmetric* algorithms in the sense that all point-wise multiplications are squarings when $A(x) = B(x)$. On the other hand, our new squaring formulae are *asymmetric* algorithms, since they involve at least one point-wise multiplication of two different values.

Compared to the Toom-Cook multiplication algorithm, our algorithms are not advantageous for squaring very large size operands, since squaring operation is usually faster than multiplication operation for all ranges of operand sizes in practice. The schoolbook squaring algorithm performs better than the schoolbook multiplication algorithm. Similarly, the same holds true for symmetric sub-quadratic algorithms, since most implementations of sub-quadratic algorithms use the schoolbook methods for the base case. Figure 2 shows the timing results of multiplication and squaring routines (both are called from `mpz_mul()`) in GMP.

The timing difference between multiplication and squaring is not significant for small operands, but the difference becomes larger as operand size grows. Hence, it easily follows that symmetric squaring algorithms are more advantageous for squaring very large size operands. However, there is a possibility that the proposed asymmetric algorithms are advantageous for squaring relatively small operand sizes, for which the effect of reduced overhead in evaluation and interpolation steps

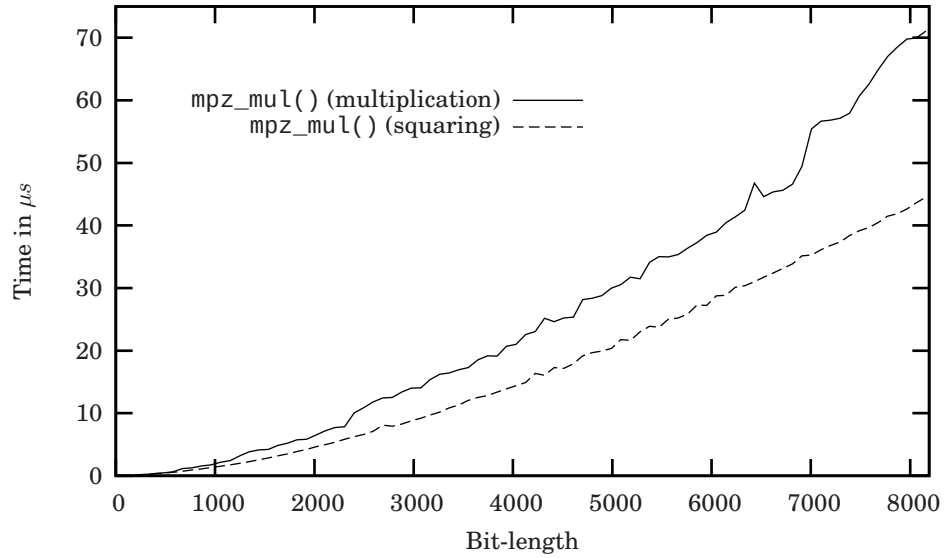

Fig. 2. Timing Results of `mpz_mul()` (multiplication) and `mpz_mul()` (squaring)

is greater than that of the lost efficiency in asymmetric point-wise multiplication step. In fact, our experimental results in Section 6, show that one of our squaring formulae performs better than the long integer squaring function in GMP for certain range of operand sizes.

4.1 Our Approach

In Section 3, we have shown that the Toom-Cook multiplication algorithm requires $(2n-1)$ distinct evaluation points for constructing a system of $(2n-1)$ linear equations having $(2n-1)$ unknown values (the coefficients of $C(x) = A(x) \cdot B(x)$). As shown in Theorems 1 and 2, such a construction method always introduces at least one non-trivial constant division in the interpolation step. For $n = 3$, even the best known 3-way Toom-Cook multiplication method, shown in Section 2.1, requires one constant division by 3 during interpolation step.

To completely eliminate the constant divisions, we take a different approach for constructing a linear system. Below we give detailed methods for obtaining linear equations on c_i's that cannot be derived by directly evaluating $C(x) = A(x)^2$. Our approach allows us to find linear equations of c_i's such that the corresponding linear system does not involve a Vandermonde matrix.

1. Taking modulo $(x^2 + ux + v^2)$, where u and v are some integers: By taking modulo $(x^2 + ux + v^2)$ on both sides of $C(x) = A(x)^2$, we obtain

$$c'_1 x + c'_0 \equiv (a'_1 x + a'_0)^2 \pmod{(x^2 + ux + v^2)}, \tag{7}$$

where $a'_1 x + a'_0 = A(x) \bmod (x^2 + ux + v^2)$ and $c'_1 x + c'_0 = C(x) \bmod (x^2 + ux + v^2)$. Then it follows that

$$c'_1 x + c'_0 \equiv a'_1(2'a'_0 - ua'_1)x + (a'_0 - va'_1)(a'_0 + va'_1) \pmod{(x^2 + ux + v^2)}. \tag{8}$$

It is interesting to see that computing both c'_0 and c'_1 requires only two coefficient multiplications. Hence, we obtain two useful linear equations for c'_is as follows:

$$\begin{aligned} c'_1 &= a'_1(2'a'_0 - ua'_1), \\ c'_0 &= (a'_0 - va'_1)(a'_0 + va'_1). \end{aligned} \tag{9}$$

Therefore, by choosing some small integers u and v, we can obtain useful linear equations on c_i's. Such equations cannot be obtained by simply evaluating $C(x) = A(x)^2$.
For example, suppose that $A(x)$ is a degree-2 polynomial, $A(x) = a_2x^2 + a_1x + a_0$. Let $u = 0$ and $v = 1$. We take modulo $(x^2 + 1)$ on both sides of $C(x) = A(x)^2$ and obtain

$$\begin{aligned} c_4 - c_2 + c_0 = (a_0 - a_2)^2 - a_1^2 = (a_0 - a_2 + a_1)(a_0 - a_2 - a_1), \\ c_1 - c_3 = 2a_1(a_0 - a_2). \end{aligned} \tag{10}$$

This method is different from the Winograd algorithm. In the Winograd algorithm, after taking modulo a second degree polynomial, c'_0 and c'_1 are simultaneously computed using KA which requires three coefficient multiplications [14]. However, the computation of c'_0 is independent from that of c'_1 in (10). Hence, we can select only one of the two linear equations in (10) without sacrificing the efficiency.
We remark that a special case of this idea is known for efficient implementation of finite field squaring in $\mathbb{Z}_{p^2}[x]/f(x)$ where $f(x) = x^2 + x + 1$ [11].

2. Hermite interpolation: Interpolation using the evaluations of derivatives is known as the Hermite interpolation. Interestingly, for squaring, each evaluation of the first derivative of $C(x)$ requires only one coefficient multiplication, since $C'(x) = 2A(x) \cdot A'(x)$.
Evaluating the first derivative of $C(x)$ gives linear relations, some of which may not be obtained by evaluating $C(x) = A(x)^2$. For example, when $A(x)$ is a degree-2 polynomial, the first derivative of $C(x) = A(x)^2$ results in the following:

$$4c_4x^3 + 3c_3x^2 + 2c_2x + c_1 = 2(a_2x^2 + a_1x + a_0)(2a_2x + a_1). \tag{11}$$

Some interesting evaluations of (11) are given below.
(a) $x = 0$: $c_1 = 2a_0a_1$.
(b) $x = \infty$: $c_4 = a_2^2$. (The same result can be obtained also by evaluating $C(x) = A(x)^2$ at $x = \infty$.)
(c) $x = 1$: $4c_4 + 3c_3 + 2c_2 + c_1 = 2(a_2 + a_1 + a_0)(2a_2 + a_1)$.
(d) $x = -1$: $-4c_4 + 3c_3 - 2c_2 + c_1 = 2(a_2 - a_1 + a_0)(-2a_2 + a_1)$.
All of the above linear equations are reasonably simple and requires only one coefficient multiplication for each evaluation point.

3. $A(x_i)^2 - A(x_j)^2 = (A(x_i) + A(x_j)) \cdot (A(x_i) - A(x_j))$ for $x_i \neq x_j$.
Using this method, we can combine two distinct evaluations of $A(x)$ into one.

4. Duality: any function computing c_i can be used to compute c_{2n-1-i} with no changes. In other words, if $c_i = f(a_0, \ldots, a_{n-2}, a_{n-1})$, then $c_{2n-1-i} = f(a_{n-1}, \ldots, a_1, a_0)$ [12].
Hence, we can safely substitute c_i to c_{2n-1-i} and a_j to a_{n-1-j} for all $0 \leq i \leq 2n-1$ and $0 \leq j \leq n-1$ in any linear equations on c_i's. For example, $c_3 = 2a_2a_1$ is a dual of $c_1 = 2a_0a_1$. This is a well-known fact and a similar argument holds for multiplications.

4.2 New 3-way Squaring

Let $\boldsymbol{C} = (c_4, c_3, c_2, c_1, c_0)$. To construct a 3-way squaring algorithm computing $C(x) = A(x)^2$ that requires only five coefficient multiplications, we need to find a five-tuple $(i_0, i_1, i_2, i_3, i_4)$, where i_j's are all distinct, such that

- There exists a u_{i_j}, which is a product of two elements (not necessarily distinct) from $L(A)$, for each vector $\boldsymbol{L}_{i_j}$ such that $\boldsymbol{L}_{i_j} \circ \boldsymbol{C} = u_{i_j}$, where $\circ$ is a dot product.
- The set of vectors $\{\boldsymbol{L}_{i_0}, \ldots, \boldsymbol{L}_{i_3}, \boldsymbol{L}_{i_4}\}$ forms a basis in $\mathbb{Z}^5$.

Let $M = (\boldsymbol{L}_{i_4}, \ldots, \boldsymbol{L}_{i_1}, \boldsymbol{L}_{i_0})^T$. If we can find a five-tuple $(i_0, \ldots, i_3, i_4)$ which makes $\det M$ a power of 2, we get a squaring algorithm that require only 5 coefficient multiplications and no non-trivial constant divisions.

We have identified 20 potentially useful $\boldsymbol{L}_i$'s and u_i's by directly evaluating $C(x) = A(x)^2$ and by using our new construction methods given above, and show them in Table 1. Note that $\boldsymbol{L}_9$–$\boldsymbol{L}_{29}$ have been obtained using the methods given above and they cannot be obtained by simply evaluating $C(x) = A(x)^2$.

Table 1. List of Candidate Vectors

i	$\boldsymbol{L}_i$	$u_i = \boldsymbol{L}_i \circ \boldsymbol{C}$	Comment
1	$(0,0,0,0,1)$	a_0^2	$C(0)$
2	$(1,0,0,0,0)$	a_2^2	$C(\infty)$
3	$(1,1,1,1,1)$	$(a_2+a_1+a_0)^2$	$C(1)$
4	$(1,-1,1,-1,1)$	$(a_2-a_1+a_0)^2$	$C(-1)$
5	$(16,8,4,2,1)$	$(4a_2+2a_1+a_0)^2$	$C(2)$
6	$(16,-8,4,-2,1)$	$(4a_2-2a_1+a_0)^2$	$C(-2)$
7	$(1,2,4,8,16)$	$(a_2+2a_1+4a_0)^2$	$2^4 \cdot C(1/2)$
8	$(1,-2,4,-8,16)$	$(a_2-2a_1+4a_0)^2$	$2^4 \cdot C(-1/2)$
9	$(0,0,0,1,0)$	$2a_0a_1$	$C'(0)$
10	$(0,1,0,0,0)$	$2a_1a_2$	Dual of 9
11	$(4,3,2,1,0)$	$2(a_2+a_1+a_0)(2a_2+a_1)$	$C'(1)$
12	$(-4,3,-2,1,0)$	$2(a_2-a_1+a_0)(-2a_2+a_1)$	$C'(-1)$
13	$(0,1,2,3,4)$	$2(a_2+a_1+a_0)(2a_0+a_1)$	Dual of 11
14	$(0,1,-2,3,-4)$	$2(a_2-a_1+a_0)(a_1-2a_0)$	Dual of 12
15	$(1,0,-1,0,1)$	$(a_0-a_2+a_1)(a_0-a_2-a_1)$	Constant term of $C(x) \bmod (t^2+1)$
16	$(0,-1,0,1,0)$	$2a_1(a_0-a_2)$	t's coefficient of $C(x) \bmod (t^2+1)$
17	$(-1,0,1,1,0)$	$(a_1-a_2+2a_0)(a_1+a_2)$	t's coefficient of $C(x) \bmod (t^2-t+1)$
18	$(0,-1,-1,0,1)$	$(a_0-a_1-2a_2)(a_0+a_1)$	Constant term of $C(x) \bmod (t^2-t+1)$
19	$(1,0,-1,1,0)$	$(a_2+a_1-2a_2)(a_2-a_1)$	t's coefficient of $C(x) \bmod (t^2+t+1)$
20	$(0,1,-1,0,1)$	$(a_0+a_1-2a_2)(a_0-a_1)$	Constant term of $C(x) \bmod (t^2+t+1)$
21	$(1,0,0,0,-1)$	$(a_0+a_2)(a_0-a_2)$	$A(0)^2 - A(\infty)^2$
22	$(0,1,0,1,0)$	$2a_1(a_2+a_0)$	$(A(1)^2 - A(-1)^2)/2$
23	$(0,4,0,1,0)$	$2a_1(4a_2+a_0)$	$(A(2)^2 - A(-2)^2)/4$
24	$(0,1,0,4,0)$	$2a_1(4a_0+a_2)$	$4(A(1/2)^2 - A(-1/2)^2)$

There are a total of 42504 possible combinations of $(i_0, i_1, i_2, i_3, i_4)$ and 34268 of them make $\{\boldsymbol{L}_{i_0}, \ldots, \boldsymbol{L}_{i_3}, \boldsymbol{L}_{i_4}\}$ a linearly independent set. We divide these 34268 combinations into the following three sets:

Set I: there are three or more i_j's such that $i_j \geq 9$.
Set II: there are only two i_j's such that $i_j \geq 9$.
Set III: there is only one i_j such that $i_j \geq 9$.

Sets I, II and III have 13584, 5946 and 1012 combinations, respectively. In Set I, it is easily seen that combinations $(1, 2, 9, 10, 15)$, $(1, 2, 9, 10, 17)$, $(1, 2, 9, 10, 18)$, $(1, 2, 9, 10, 19)$ and $(1, 2, 9, 10, 20)$ lead to the simplest interpolation step. Note that $\boldsymbol{L}_1$, $\boldsymbol{L}_2$, $\boldsymbol{L}_9$, $\boldsymbol{L}_{10}$ immediately give the coefficients c_0, c_1, c_3 and c_4 of $C(x)$. The remaining coefficient c_2 can be obtained by at most two additions/subtractions. Among the five contenders, $(1, 2, 9, 10, 15)$ is the best choice, since computing u_{15} is easier than computing u_{17}, u_{18}, u_{19} and u_{20}.

In Set II, there are 124 combinations of $(i_0, \ldots, i_3, i_4)$ such that $|\det M| = 1$. To narrow down our search, we have considered only the combinations that lead to M such that the entries of M^{-1} are relatively small. Combinations $(1, 2, 3, 9, 10)$, $(1, 2, 4, 9, 10)$, $(1, 2, 4, 9, 22)$ and $(1, 2, 4, 10, 22)$ are the best, and they lead to the simplest form of M^{-1}. Combination $(1, 2, 4, 9, 10)$ is more advantageous than $(1, 2, 3, 9, 10)$, since computing u_4 is more efficient than computing u_3. Note that $(a_2 + a_1 + a_0)$ could be at most 1 bit longer than $(a_2 - a_1 + a_0)$. Moreover, $(1, 2, 4, 9, 10)$ is better than $(1, 2, 4, 9, 22)$ and $(1, 2, 4, 10, 22)$ since computing u_9 and u_{10} is faster than computing u_9 and u_{22} or computing u_{10} and u_{22}. We have also considered combinations that results in $|\det M| = 2, 4, 8$ and 16, but could not find a better combination than $(1, 2, 3, 4, 9)$ and $(1, 2, 3, 4, 10)$.

In Set III, there is no combination that makes $|\det M| = 1$, but there are 26 combinations that makes $|\det M| = 2$. Among these 26 combinations, $(1, 2, 3, 4, 9)$ and $(1, 2, 3, 4, 10)$ lead to the most efficient squaring algorithm. We have also considered combinations that result in $|\det M| = 4, 8$ and 16, but could not find a better combination than $(1, 2, 3, 4, 9)$ and $(1, 2, 3, 4, 10)$.

We have derived three new squaring methods from Set I, II and III.

1. Squaring Method 1 (SQR_1)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} a_2^2 \\ 2a_1a_2 \\ (a_0 - a_2 + a_1)(a_0 - a_2 - a_1) \\ 2a_1a_0 \\ a_0^2 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} \tag{12}$$

   The computation of S_i's requires 3 coefficient multiplications and 2 coefficient squarings. The determinant of the 5×5 matrix in (12) is -1, meaning the interpolation can be performed without bit-shift or constant division. In fact, the coefficients c_0, c_1, c_3 and c_4 are already given. The coefficients c_2 can be computed with one addition and one subtraction: $c_2 = S_0 + S_4 - S_2$.
2. Squaring Method 2 (SQR_2)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} a_2^2 \\ 2a_1a_2 \\ (a_2 - a_1 + a_0)^2 \\ 2a_1a_0 \\ a_0^2 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} \tag{13}$$

   This algorithm requires 2 coefficient multiplications and 3 coefficient squarings. The coefficients c_0, c_1, c_3 and c_4 are already given. The remaining coefficient c_2 can be obtained using only 4 additions/subtractions: $c_2 = S_2 + S_1 + S_3 - S_0 - S_4$.

3. Squaring Method 3 (SQR_3)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} a_2^2 \\ 2a_1a_2 \\ (a_2 - a_1 + a_0)^2 \\ (a_2 + a_1 + a_0)^2 \\ a_0^2 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} \tag{14}$$

This algorithm requires 1 coefficient multiplication and 4 coefficient squarings. The coefficients c_0, c_3 and c_4 are already given. The coefficients c_1 and c_2 are computed using only 5 additions/subtractions and 1 bit-shift.

$$\begin{aligned} T_1 &= (S_1 + S_2)/2, \\ c_1 &= S_1 - T_1 - S_3, \\ c_2 &= T_1 - S_4 - S_0. \end{aligned} \tag{15}$$

5 Analysis

In this section, we analyze the squaring algorithms SQR_1, SQR_2 and SQR_3 presented in Section 4. The analysis may differ depending on how the various squaring algorithms are used in specific applications (e.g., long integer squaring, squaring in extension field $GF(p^m)$, polynomial squaring in $\mathbb{Z}[x]$,...). In this section, we assume that our squaring formulae are applied to the arithmetic in $\mathbb{Z}[x]$. We also compare our algorithms with other known squaring algorithms: schoolbook squaring algorithm, 3-way KA-like formula and Zimmerman's 3-way Toom-Cook algorithms.

We denote the digit size of the representation by b. Addition or subtraction of two u-digit integers requires $\mathcal{A}(u)$ time. Multiplication and squaring of two u-digit integers require $\mathcal{M}(u)$ and $\mathcal{S}(u)$ times, respectively. Bit-shift of u digit integers require $\mathcal{B}(u)$ time. During evaluation and interpolation step, there are cases when the operands to addition/subtraction and shift are a few bits larger than u or $2u$ digits, where the coefficients of $A(x)$ are at most u digits long. For simplicity, we ignore this overhead caused by carries and borrows. However, we do not ignore the overhead involved in multiplying two integers that are slightly longer than u digits. For example, an integer s and t are only 1-bit longer than u digits. Then we can write $s = s_h b + s_l$ and $t = t_h b + t_l$, where $|s_h|, |t_h| \leq 1$ and $0 \leq s_l, t_l < b^u$. The time required to compute $s \cdot t$ is at most $\mathcal{M}(u) + 2\mathcal{A}(u)$. For simplicity, we ignore the cost for multiplying carries, i.e., s_h and t_h.[1] The time required to compute s^2 is $\mathcal{S}(u)+\mathcal{B}(u)+\mathcal{A}(u)$ in the worst case. When computing a product $2a_ia_j$, we always compute a_ia_j first and then perform the bit-shift later. It is reasonable to assume that $\mathcal{A}(\cdot)$ and $\mathcal{B}(\cdot)$ are linear functions; $\mathcal{A}(fu + gv) = f\mathcal{A}(u) + g\mathcal{A}(v)$ and $\mathcal{B}(fu + gv) = f\mathcal{B}(u) + g\mathcal{B}(v)$. The exact division by 3 of an u-digit integer used in the 3-way Toom-Cook algorithm shown in Section 2.1 is denoted by $\mathcal{D}_3(u)$.

We assume that $A(x) = a_2x^2 + a_1x + a_0$ is the input, where a_i's are u digits long. Table 2 shows our analysis results. Table 3 shows the conditions for which our squaring algorithms are superior to the other algorithms.

[1] Note, however, that the 3-way Toom-Cook multiplication algorithm in GMP v4.2.1 uses extra digit for carries and borrows instead of handling them separately. This method has a trade-off. Using extra digit reduces the number of additions and subtractions, but coefficient multiplications and squarings becomes slower. We do not know which method is better, but we believe the difference is very small.

Table 2. Analysis Results of Various Squaring Algorithms

Algorithm	$\mathcal{S}\&\mathcal{M}$	Overhead
3-way Toom-Cook	$5\mathcal{S}(u)$	$14\mathcal{B}(u) + 25\mathcal{A}(u) + \mathcal{D}_3(2u)$
Schoolbook sqr.	$3\mathcal{S}(u) + 3\mathcal{M}(u)$	$6\mathcal{B}(u) + 2\mathcal{A}(u)$
3-way KA-like	$6\mathcal{S}(u)$	$3\mathcal{B}(u) + 20\mathcal{A}(u)$
SQR_1	$2\mathcal{S}(u) + 3\mathcal{M}(u)$	$5\mathcal{B}(u) + 9\mathcal{A}(u)$
SQR_2	$3\mathcal{S}(u) + 2\mathcal{M}(u)$	$5\mathcal{B}(u) + 11\mathcal{A}(u)$
SQR_3	$4\mathcal{S}(u) + 1\mathcal{M}(u)$	$6\mathcal{B}(u) + 15\mathcal{A}(u)$

Table 3. Conditions for Which SQR_i's Are Faster Than Other 3-way Algorithms

i	SQR_i vs. 3-way Toom-Cook
1	$3\mathcal{M}(u) < 3\mathcal{S}(u) + 9\mathcal{B}(u) + 16\mathcal{A}(u) + \mathcal{D}_3(2u)$
2	$2\mathcal{M}(u) < 2\mathcal{S}(u) + 9\mathcal{B}(u) + 14\mathcal{A}(u) + \mathcal{D}_3(2u)$
3	$\mathcal{M}(u) < \mathcal{S}(u) + 8\mathcal{B}(u) + 10\mathcal{A}(u) + \mathcal{D}_3(2u)$

i	SQR_i vs. Schoolbook sqr.
1	$7\mathcal{A}(u) < S(u) + \mathcal{B}(u)$
2	$9\mathcal{A}(u) < \mathcal{M}(u) + \mathcal{B}(u)$
3	$\mathcal{S}(u) + 13\mathcal{A}(u) < 2\mathcal{M}(u)$

i	SQR_i vs. 3-way KA-like
1	$3\mathcal{M}(u) + 2\mathcal{B}(u) < 4\mathcal{S}(u) + 11\mathcal{A}(u)$
2	$2\mathcal{M}(u) + 2\mathcal{B}(u) < 3\mathcal{S}(u) + 9\mathcal{A}(u)$
3	$\mathcal{M}(u) + 3\mathcal{B}(u) < 2\mathcal{S}(u) + 5\mathcal{A}(u)$

Table 3 shows that there is apparently no single algorithm that is absolutely superior to the others. Without considering the actual values $\mathcal{S}(u)$, $\mathcal{M}(u)$, $\mathcal{B}(u)$, $\mathcal{A}(u)$ and $\mathcal{D}_3(2u)$, which are very application specific, it is not easy to decide which algorithm is faster than the rest. However, one thing that is clear from Table 3 is that the 3-way Toom-Cook algorithm becomes the best for squaring polynomials as u increases. The timings $\mathcal{B}(u)$, $\mathcal{A}(u)$ and $\mathcal{D}_3(2u)$ grow linearly with u, but timings of multiplication ($\mathcal{M}(u)$) and squaring ($\mathcal{S}(u)$) grow quadratically or sub-quadratically depending on the methods used for point-wise multiplications. It is obvious that, for large u, the effect of reduced overhead in our algorithms will be offset by the timing difference in multiplication and squaring.

However, SQR_i's have very little amount of overhead compared to the 3-way Toom-Cook multiplication algorithm. Hence, it is possible that, for some small u, the timing difference of multiplication and squaring is small enough that some of the conditions in Table 3 will be satisfied. In fact, our implementation results given in Section 6 confirm that there is a range of u where some conditions of Table 3 are satisfied.

6 Implementation Results

To verify the practical usefulness of our algorithms, we have implemented in software the functions for large integer squaring, and the functions for degree-2 polynomial squaring in $\mathbb{Z}[x]$ using SQR_1, SQR_2 and SQR_3 presented in Section 4. Our experiments have been performed on Linux running on Intel Pentium 4 Prescott 3.2GHz. We have used GCC 4.0.2 to compile all programs. We have compiled GMP library v4.2.1 in two passes. Between the first and the second passes of compilations,

we have performed GMP's tuneup program so that GMP uses the optimal threshold points between multiplication algorithms. We have ensured that our program do not link with the shared library of GMP, since functions from shared libraries do not perform as efficient as those from static libraries.

6.1 Application to Large Integer Squaring

We have compared our implementation of SQR_i's with GMP's squaring function. For fair comparison, our algorithms have been written so that it can replace `mpn_toom3_sqr_n()` function in GMP. Note that `mpn_toom3_sqr_n()` is a low level implementation of the algorithm shown in Section 2.1 for squaring case.[2] Our squaring algorithms are written in a similar way as the function `mpn_toom3_sqr_n()` using the functions prefixed with `mpn_`. We have not used any optimization tricks or special functions other than those used in `mpn_toom3_sqr_n()`. We have ensured that our implementations produce correct results for varying bit-sizes of input.

When timing our squaring algorithms we have replaced `mpn_toom3_sqr_n()` with our functions and called from the top level function `mpz_mul()`. To prevent `mpz_mul()` from using the schoolbook squaring algorithm and KA, we have modified `mpn_sqr_n()`, which chooses the best one among various squaring algorithms depending on the input size, so that only our algorithms are called for all ranges of input sizes. For timing the 3-way Toom-Cook multiplication algorithm, we have forced `mpn_sqr_n()` to choose only the original `mpn_toom3_sqr_n()`.

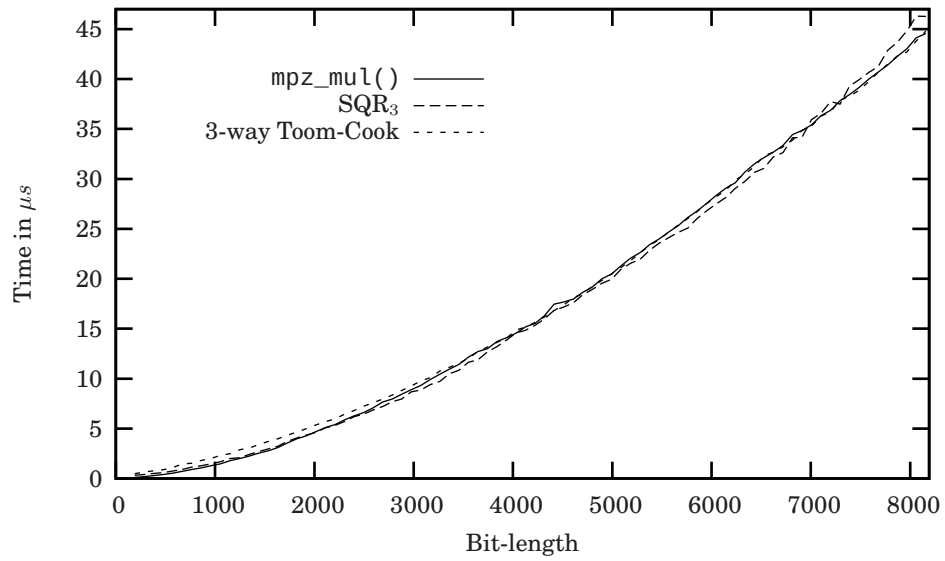

Fig. 3. Timing Results of Squaring Algorithms

[2] Even though the 3-way Toom-Cook squaring is separately implemented, it uses the same 3-way Toom-Cook multiplication algorithm given in Section 2.1.

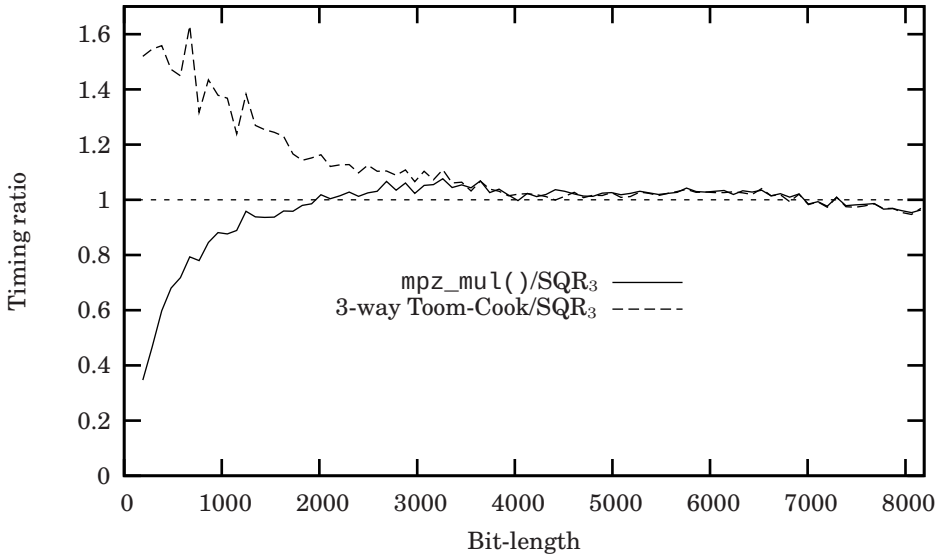

Fig. 4. Timing Ratio of SQR_3 vs. Other Algorithms

Figure 3 shows the timing results of SQR_3, `mpz_mul()` and the 3-way Toom-Cook multiplication algorithm. Figure 4 shows the timing ratio of `mpz_mul()` and the 3-way Toom-Cook multiplication algorithm to SQR_3. Note that `mpz_mul()` uses the schoolbook squaring algorithm for small operands, KA for input longer than 2112 bits, the 3-way Toom-Cook algorithm for input longer than 3712 bits. In our experiments, we have found that SQR_1 and SQR_2 are slower than `mpz_mul()` for all sizes of input. Thus, we have not included their timing results. Our experiments show that SQR_3 outperforms `mul_mul()` for operands that are about 2016–6912 bits long. An interesting consequence of this result is that GMP does not need the KA for squaring on Pentium 4 Prescott 3.2GHz.

We have also performed the same experiment on Pentium II 350MHz and Pentium III Mobile 1.13GHz. We have plotted the results in Figures 5 and 6. On both processors, SQR_3 performed better than `mpz_mul()` for about 2000–4000-bit operands by upto 5-7%. For 1500–2000-bit size operands, there is no clear winner. The GMP tuneup program has determined that the crossover between the classical multiplication and the KA is 48 words (1536 bits) and the crossover between the KA and 3-way Toom-Cook multiplication algorithm is 89 words (2848 bits). Hence, GMP's KA squaring is not necessary on both Pentium II 350MHz and Pentium III Mobile 1.13GHz.

6.2 Application to Polynomial Squaring in $\mathbb{Z}[x]$

We have applied our squaring algorithm for performing polynomial multiplication in $\mathbb{Z}[x]$. We have implemented functions for squaring degree-2 polynomials in $\mathbb{Z}[x]$. The implementation uses the functions from GMP library. The results are shown in Table 4. The first column in Table 4 shows the sizes of coefficients in bits. In the table, the best timing for each bit-length is indicated in bold. SQR_1 is the most efficient squaring algorithm for squaring polynomials having small coefficients of up to 576

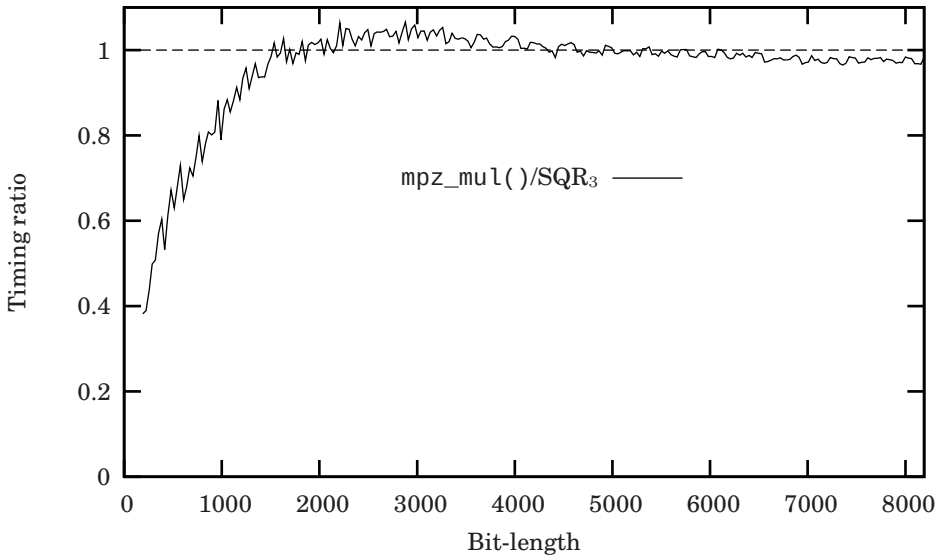

Fig. 5. Timing Ratio of SQR_3 vs. `mpz_mul()` (Pentium II)

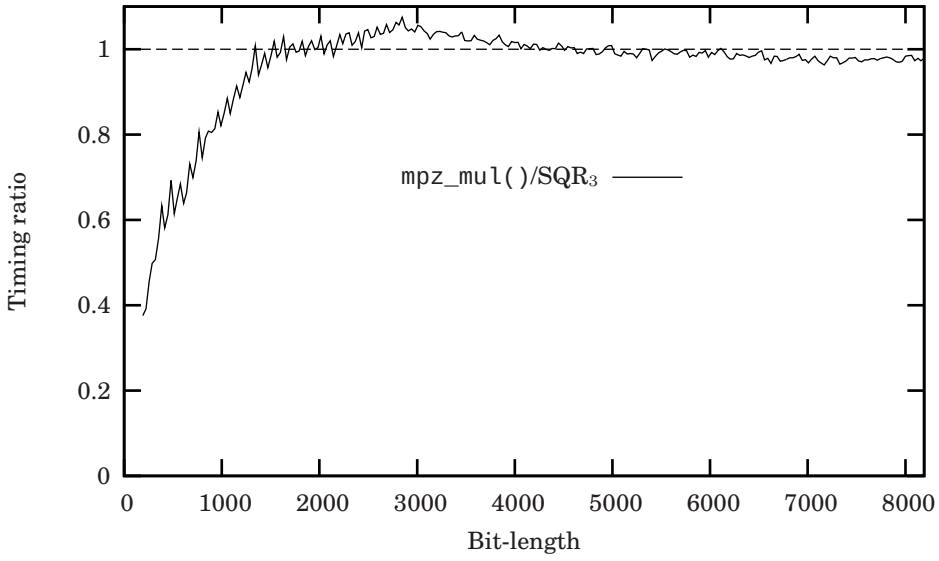

Fig. 6. Timing Ratio of SQR_3 vs. `mpz_mul()` (Pentium III M)

bits. For polynomials with coefficients up to 1216 bits, SQR_3 is the most efficient. However, the 3-way Toom-Cook algorithm becomes the fastest algorithm for squaring degree-2 polynomials whose coefficients are at least 1216 bits long.

Table 4. Timing Results of Polynomial Squaring (unit= μs.)

Bit-length	SQR_1	SQR_2	SQR_3	3-way Toom-Cook	3-way KA-like	Schoolbook sqr.
32	**0.23**	0.32	0.37	0.58	0.43	0.25
256	**1.12**	1.42	1.33	1.68	1.80	1.26
576	**3.43**	3.96	3.59	4.09	4.79	4.15
608	3.89	4.25	**3.85**	4.42	5.25	4.42
768	6.09	6.41	**5.44**	5.87	7.29	6.99
1024	8.37	9.65	**8.17**	8.47	11.12	10.72
1216	11.28	12.67	**10.73**	10.83	14.17	14.38
1248	12.84	14.11	11.51	**11.17**	15.21	16.29
1502	17.15	18.94	15.42	**14.90**	19.85	22.02
1536	19.37	19.52	15.55	**15.21**	20.40	22.31

7 4-way and 5-way Squaring Formulae

We have constructed 4-way and 5-way squaring formulae that do not require any non-trivial constant divisions. We have used the same technique that we applied to construct 3-way formulae. We have chosen the algorithms that have the simplest interpolation among the many candidates we have considered so far. The results shown in this section are to illustrate that the technique we have developed in Section 4 can also be applied to construct n-way squaring formulae for $n > 3$. Note that 5-way is not the limit where non-trivial divisions in interpolation step can be eliminated. Future research will show further results on 4, 5, 6, 7-way squaring formulae.

7.1 New 4-Way Squaring

Let $A(x) = a_3x^3 + a_2x^2 + a_1x + a_0$. To compute $C(x) = \sum_{i=0}^{6} c_i x^i = A(x)^2$, we first compute S_i's as shown below:

$$\begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \\ S_6 \\ S_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 1 & 0 & -1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_6 \\ c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} a_0^2 \\ 2a_0a_1 \\ (a_0 + a_1 - a_2 - a_3)(a_0 - a_1 - a_2 + a_3) \\ (a_0 + a_1 + a_2 + a_3)^2 \\ 2(a_0 - a_2)(a_1 - a_3) \\ 2a_3a_2 \\ a_3^2 \end{bmatrix}. \quad (16)$$

Algorithm 3 New 4-Way Toom-Cook Interpolation for Squaring

Require: $(S_1, S_2, S_3, S_4, S_5, S_6, S_7)$ as in (16).
Ensure: $C(x) = A(x) \cdot B(x)$.

1: $T_1 \leftarrow S_3 + S_4$.	$(= c_5 + 2c_4 + c_3 + c_1 + 2c_0)$
2: $T_2 \leftarrow (T_1 + S_5)/2$.	$(= c_5 + c_4 + c_1 + c_0)$
3: $T_3 \leftarrow S_2 + S_6$.	$(= c_5 + c_1)$
4: $T_4 \leftarrow T_2 - T_3$.	$(= c_4 + c_0)$
5: $T_5 \leftarrow T_3 - S_5$.	$(= c_3)$
6: $T_6 \leftarrow T_4 - S_3$.	$(= c_6 + c_2)$
7: $T_7 \leftarrow T_4 - S_1$.	$(= c_4)$
8: $T_8 \leftarrow T_6 - S_7$.	$(= c_2)$
9: **return** $C(x) = S_7x^6 + S_6x^5 + T_7x^4 + T_5x^3 + T_8x^2 + S_2x + S_1$.	

The linear combinations of a_i's in (16) can be computed using the following:

$$\begin{aligned} T_1 &= a_0 - a_2 \\ T_2 &= a_1 - a_3 \\ T_3 &= T_1 + T_2 \\ T_4 &= T_1 - T_2 \\ T_5 &= a_0 + a_1 + a_2 + a_3. \end{aligned} \tag{17}$$

The determinant of the 7×7 matrix in (16) is 2. This method uses 3 coefficient squarings and 4 coefficient multiplications. Note that KA requires 9 coefficient squarings for squaring a polynomial using 4-way split. Interpolation method is given in Algorithm 3. Algorithm 3 requires only 8 additions/subtractions and 1 bit shift.

Using the same analysis methods in 5, we obtain that our 4-way squaring algorithm requires $3\mathcal{S}(u) + 4\mathcal{M}(u) + 28\mathcal{A}(u) + 13\mathcal{B}(u)$.

7.2 New 5-term Squaring Method

Let $A(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$. To compute $C(x) = \sum_{i=0}^{8} c_ix^i = A(x)^2$, we first compute S_i's as shown below:

$$\begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \\ S_6 \\ S_7 \\ S_8 \\ S_9 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 \\ 1 & 1 & 0 & -1 & -1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_8 \\ c_7 \\ c_6 \\ c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix}$$

$$= \begin{bmatrix} a_0^2 \\ a_4^2 \\ (a_0 + a_1 + a_2 + a_3 + a_4)^2 \\ (a_0 - a_1 + a_2 - a_3 + a_4)^2 \\ 2(a_0 - a_2 + a_4)(a_1 - a_3) \\ (a_0 + a_1 - a_2 - a_3 + a_4)(a_0 - a_1 - a_2 + a_3 + a_4) \\ (a_1 + a_2 - a_4)(a_1 - a_2 - a_4 + 2(a_0 - a_3)) \\ 2a_0a_1 \\ 2a_3a_4 \end{bmatrix}. \tag{18}$$

The above system needs 4 squarings and 5 multiplications. The linear combinations of a_i's can be computed as follows using 14 additions or subtractions and 1 shift:

$$\begin{aligned}
T_1 &= a_0 + a_4, & T_8 &= T_5 - T_2 = \mathbf{a_0} - \mathbf{a_1} + \mathbf{a_2} - \mathbf{a_3} + \mathbf{a_4}, \\
T_2 &= a_1 + a_3, & T_9 &= T_6 + T_4 = \mathbf{a_0} + \mathbf{a_1} - \mathbf{a_2} - \mathbf{a_3} + \mathbf{a_4}, \\
T_3 &= a_1 - a_4, & T_{10} &= T_6 - T_4 = \mathbf{a_0} - \mathbf{a_1} - \mathbf{a_2} + \mathbf{a_3} + \mathbf{a_4}, \\
T_4 &= \mathbf{a_1} - \mathbf{a_3}, & T_{11} &= T_3 + a_2 = \mathbf{a_1} + \mathbf{a_2} - \mathbf{a_4}, \\
T_5 &= T_1 + a_2 = a_0 + a_2 + a_4, & T_{12} &= T_3 - a_2 = a_1 - a_2 - a_4, \\
T_6 &= T_1 - a_2 = \mathbf{a_0} - \mathbf{a_2} + \mathbf{a_4}, & T_{13} &= T_{12} - 2(a_0 - a_3) = \mathbf{a_1} - \mathbf{a_2} - \mathbf{a_4} - \mathbf{2}(\mathbf{a_0} - \mathbf{a_3}) \\
T_7 &= T_5 + T_2 = \mathbf{a_0} + \mathbf{a_1} + \mathbf{a_2} + \mathbf{a_3} + \mathbf{a_4}.
\end{aligned}$$

Interpolation can be performed by Algorithm 4. The algorithm requires 18 additions and subtractions, 7 shifts and no divisions by constants.

Note that Montgomery's 5-way formulae requires 13 squarings. Hence, if the ratio of squaring to multiplication is greater than 5.6, there is a very good possibility that our algorithm is superior to Montgomery's 5-way formula.

Using the same analysis technique and assuming that each coefficient a_i is u-digit integer, we obtain that our 5-way squaring algorithm requires at most $4\mathcal{S}(u) + 5\mathcal{M}(u) + 60\mathcal{A}(u) + 26\mathcal{B}(u)$. Montgomery's 5-way algorithm requires at most $13\mathcal{S}(u)+65\mathcal{A}(u)+10\mathcal{B}(u)$ when two operands are identical. Therefore, if $5\mathcal{M}(u) + 16\mathcal{B}(u) < 9\mathcal{S}(u) + 5\mathcal{A}(u)$, then our algorithm is superior. Ignoring the overhead terms ($\mathcal{A}$ and $\mathcal{B}$), our algorithm is superior if squaring/multiplication ratio is more than $5/9 \approx 0.56$. This condition appears to be easily satisfied in practice. Figure 7 shows the timing ratio of squaring and multiplication routines in GMP library. In the figure, the GMP's squaring/multiplication timing ratio is between 0.6-0.8 for operand sizes larger than 500-bits on Pentium 4 Prescott 3.2GHz.

Algorithm 4 New 5-Way Toom-Cook Interpolation for Squaring

Require: $(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)$ as in (18).
Ensure: $C(x) = A(x) \cdot B(x)$.

1: $T_1 \leftarrow S_1 + 2 \cdot S_2 - S_7 + 2 \cdot S_8 + S_9$	$(= c_8 + c_5 + c_4 - c_2 + c_1 + c_0)$
2: $T_2 \leftarrow S_3 - S_4$	$(= 2c_7 + 2c_5 + 2c_3 + 2c_1)$
3: $T_3 \leftarrow 2 \cdot S_5$	$(= -2c_7 + 2c_5 - 2c_3 + 2c_1)$
4: $T_4 \leftarrow T_2 + T_3$	$(= 4c_5 + 4c_1)$
5: $T_5 \leftarrow T_2 - T_3$	$(= 4c_7 + 4c_3)$
6: $T_6 \leftarrow T_4/4$	$(= c_5 + c_1)$
7: $T_7 \leftarrow T_5/4 - S_9$	$(= c_3)$
8: $T_8 \leftarrow T_1 - T_6 - S_6$	$(= c_6)$
9: $T_9 \leftarrow T_6 - S_8$	$(= c_5)$
10: $T_{10} \leftarrow S_3 + S_6$	$(= 2c_8 + c_7 + c_5 + 2c_4 + c_3 + c_1 + 2c_0)$
11: $T_{11} \leftarrow (T_{10} + S_4 + S_6)/4$	$(= c_8 + c_4 + c_0)$
12: $T_{12} \leftarrow T_{11} - T_1 - T_2$	$(= c_4)$
13: $T_{13} \leftarrow (T_{10} + S_5)/2$	$(= c_8 + c_5 + c_4 + c_1 + c_0)$
14: $T_{14} \leftarrow T_{13} - T_1$	$(= c_2)$

15: **return** $C(x) = S_2 t^8 + S_9 t_7 + T_8 t^6 + T_9 t^5 + T_{11} t^4 + T_7 t_3 + T_{13} t^2 + S_8 t + S_1$.

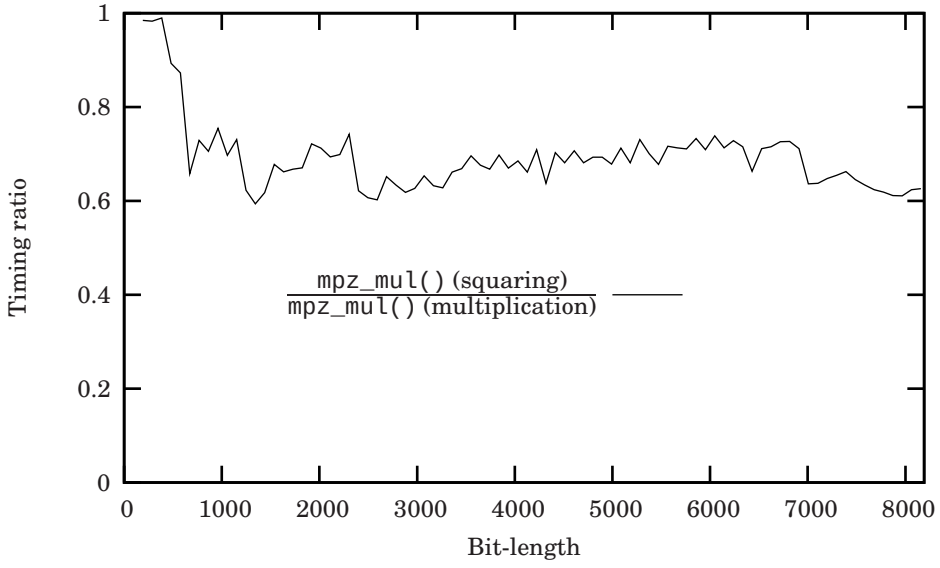

Fig. 7. Timing Ratio of `mpz_mul()` (multiplication) and `mpz_mul()` (squaring)

Usually, additions and subtractions are slower than bit-shifts by a small factor. If the bit-shift is more than 3.2 times faster than additions/subtractions, then our 5-way squaring algorithm is clearly faster than Montgomery's 5-way algorithm for all ranges of input sizes.

8 Conclusions

In this report, we have presented new 3, 4 and 5-way polynomial squaring formulae. Our new formulae are based on the Toom-Cook multiplication algorithm and they require the same number of coefficient multiplications used in the Toom-Cook multiplication algorithm. However, our approach eliminates the need for non-trivial constrant divisions always required in the n-way Toom-Cook multiplication algorithms for $n \geq 3$. Our experimental results confirm that one of our 3-way formulae is slightly faster than GMP's squaring routine for squaring integers of size about 2000-7000 bits. Moreover, according to our implementation results, our squaring formulae are the best for squaring degree-2 polynomials whose coefficients are shorter than about 1200 bits. However, symmetric squaring algorithms are advantageous for squaring very large size operands, since our asymmetric squaring algorithms use at least one point-wise multiplication that cannot be computed by squaring.

References

1. Daniel V. Bailey and Christof Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.
2. Daniel Bernstein. Multidigit multiplication for mathematicians, 1991. Available at `http://cr.yp.to/papers/m3.pdf`.
3. S. A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, May 1966.
4. GNU. GNU multiple precision arithmetic library, 2005. Available at `http://www.swox.com/gmp`.
5. Tudor Jebelean. An algorithm for exact division. *Journal of Symbolic Computation*, 15:169–180, 1993. Research report version available at `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1992/92-35.ps.gz`.
6. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady (English translation)*, 7(7):595–596, 1963.
7. D.E. Knuth. *The Art of Comuter Programming, Vol. 2, Seminumerical Algorithms*. Addison-Wesley, 2nd edition edition, 1981.
8. Werner Krandick and Tudor Jebelean. Bidirectional exact integer division. *Journal of Symbolic Computation*, 21:441–455, 1996. Early technical report version available at `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1994/94-50.ps.gz`.
9. Peter L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transaction on Computers*, 54(3):362–369, 2005.
10. Arnold Schönhage and Volker Strassen. Schnelle mutiplikation grosser zahlen. *Computing*, 7:281–292, 1971.
11. Martijn Stam and Arjen K. Lenstra. Speeding up XTR. In *Advances in Cryptology - ASIACRYPT 2001*, LNCS 2248, pages 125–143. Springer-Verlag, 2001.
12. Berk Sunar. A generalized method for constructing subquadratic complexity $GF(2^k)$ multipliers. *IEEE Transactions on Computers*, 53:1097–1105, 2004.
13. A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Math*, 3:714–716, 1963.
14. Shmuel Winograd. *Arithmetic Complexity of Computations*. CBMS-NSF Regional Conference Series in Applied Mathematics 33. Society for Industrial and Applied Mathematics, 1980.
15. Dan Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, August 1994.