A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm

Pierrick Gaudry, Alexander Kruppa, Paul Zimmermann

A GMP-BASED IMPLEMENTATION OF SCHÖNHAGE-STRASSEN'S LARGE INTEGER MULTIPLICATION ALGORITHM

PIERRICK GAUDRY, ALEXANDER KRUPPA, AND PAUL ZIMMERMANN

Abstract. Schönhage-Strassen's algorithm is one of the best known algorithms for multiplying large integers. Implementing it efficiently is of utmost importance, since many other algorithms rely on it as a subroutine. We present here an improved implementation, based on the one distributed within the GMP library. The following ideas and techniques were used or tried: faster arithmetic modulo $2^n + 1$, improved cache locality, Mersenne transforms, Chinese Remainder Reconstruction, the $\sqrt{2}$ trick, Harley's and Granlund's tricks, improved tuning. We also discuss some ideas we plan to try in the future.

Introduction

Since Schönhage and Strassen have shown in 1971 how to multiply two N-bit integers in $O(N \log N \log\log N)$ time [21], several authors showed how to reduce other operations — inverse, division, square root, gcd, base conversion, elementary functions — to multiplication, possibly with $\log N$ multiplicative factors [5, 8, 17, 18, 20, 23]. It has now become common practice to express complexities in terms of the cost $M(N)$ to multiply two N-bit numbers, and many researchers tried hard to get the best possible constants in front of $M(N)$ for the above-mentioned operations (see for example [6, 16]).

Strangely, much less effort was made for decreasing the implicit constant in $M(N)$ itself, although any gain on that constant will give a similar gain on all multiplication-based operations. Some authors reported on implementations of large integer arithmetic for specific hardware or as part of a number-theoretic project [2, 10]. In this article we concentrate on the question of an optimized implementation of Schönhage-Strassen's algorithm on a classical workstation.

In the last years, the multiplication of large integers has found several new applications in "real life", and not only in computing billions of digits of π. One such application is the segmentation method (called Kronecker substitution in [25]) to reduce the multiplication of polynomials with integer coefficients to one huge integer multiplication; this is used for example in the GMP-ECM software [27]. Another example is the multiplication or factorization of multivariate polynomials [23, 24].

In this article we detail several ideas or techniques that may be used to implement Schönhage-Strassen's algorithm (SSA) efficiently. As a consequence, we obtain what we believe is the best existing implementation of SSA on current processors; this implementation might be used as a reference to compare with other algorithms based on Fast Fourier Transform, in particular those using complex floating-point numbers.

The paper is organized as follows: §1 revisits the original SSA and defines the notations used in the rest of the paper; §2 describes the different ideas and techniques we tried, explains which ones were useful, and which ones were not; finally §3 provides timing figures and graphs obtained with our new GMP implementation, and compares it to other implementations.

1. Schönhage-Strassen's Algorithm

Throughout the paper we use w for the computer word size in bits — usually 32 or 64 — and denote by N the number of bits of the numbers we want to multiply.

Several descriptions of SSA can be found in the literature, see [11, 21] for example. We recall it here to establish the notations.

Let R_N^+ — or simply R_N — be the ring of integers modulo 2^N+1. SSA reduces integer multiplication to multiplication in R_N, which reduces to polynomial multiplication in $\mathbb{Z}[x] \bmod (x^K+1)$, which in turn reduces to polynomial multiplication in $R_n[x] \bmod (x^K+1)$, which finally reduces to multiplication in R_n:

$$\mathbb{Z} \Longrightarrow R_N \Longrightarrow \mathbb{Z}[x] \bmod (x^K+1) \Longrightarrow R_n[x] \bmod (x^K+1) \Longrightarrow R_n.$$

The first reduction — from $\mathbb{Z}$ to R_N — is simple: to multiply two non-negative integers of u and v bits, it suffices to compute their product mod 2^N+1 for $N \geq u+v$.

The second reduction — from R_N to $\mathbb{Z}[x] \bmod (x^K+1)$ — works as follows. Assume $N = 2^k M$ for integers k and M, and define $K := 2^k$. An integer $a \in [0, 2^N]$ can be uniquely written

$$a = \sum_{i=0}^{K-1} a_i 2^{iM},$$

with $0 \leq a_i < 2^M$ for $i < K-1$, and $0 \leq a_{K-1} \leq 2^M$. Such an integer is the value at $x = 2^M$ of the polynomial

$$A(x) = \sum_{i=0}^{K-1} a_i x^i.$$

Assume one decomposes an integer $b \in R_N$ in the same manner, and let $C(x)$ be the product $A(x)B(x)$ over $\mathbb{Z}[x]$:

$$C(x) = \sum_{i=0}^{2K-2} c_i x^i.$$

One now has $ab = A(2^M)B(2^M) = C(2^M)$, thus $ab = \sum_{i=0}^{2K-2} c_i 2^{iM}$. Now what we really want is $ab \bmod (2^N+1)$, i.e.,

$$(c_0 - c_K) + (c_1 - c_{K+1})2^M + \cdots + (c_{K-2} - c_{2K-2})2^{(K-2)M} + c_{K-1}2^{(K-1)M}. \tag{1}$$

This value can be obtained from $C^+(x) := A(x)B(x) \bmod (x^K+1)$, since $ab \equiv C^+(2^M) \bmod (2^N+1)$.

To determine $C^+(x) = A(x)B(x) \bmod (x^K+1)$, one uses a *negacyclic convolution* over the ring R_n, i.e., modulo 2^n+1, where n is taken large enough so that the coefficients of $C^+(x)$ can be recovered exactly. For $0 \leq i \leq K-1$, one has $0 \leq c_i = \sum_{j=0}^{i} a_j b_{i-j} < (i+1)2^{2M}$. Similarly for $K \leq i \leq 2K-3$, one has $0 \leq c_i < (2K-1-i)2^{2M}$ and finally $0 \leq c_{2K-2} \leq 2^{2M}$. With the convention that $c_{2K-1} = 0$, one thus has for $0 \leq i < K$:

$$((i+1)-K)2^{2M} \leq \bar{c}_i := c_i - c_{i+K} < (i+1)2^{2M}. \tag{2}$$

So each coefficient of $C(x) \bmod (x^K+1)$ is confined to an interval of length $K2^{2M}$, and so it suffices to have $2^n+1 \geq K2^{2M}$, i.e., $n \geq 2M+k$, to be able to recover the coefficients $\bar{c}_i$ exactly[1].

[1] One might require $n \geq 2M+k+1$ to get a lifting algorithm from R_n to $\mathbb{Z}$ which is independent of i.

The negacyclic convolution $A(x)B(x) \bmod (x^K+1)$ can be performed efficiently using the Fast Fourier Transform (FFT). More precisely, SSA uses here a Discrete Weighted Transform (DWT) [10]. Assume $\omega = \theta^2$ is a primitive Kth root of unity in R_n. (All operations in this paragraph are in R_n.) Given $(a_i)_{0\le i<K}$, the weight signal is $(a'_i := \theta^i a_i)_{0\le i<K}$. The forward transform computes $(\hat{a}_i := \sum_{j=0}^{K-1} \omega^{ij} a'_j)_{0\le i<K}$, and similarly for $(\hat{b}_i)$. One then multiplies $\hat{a}_i$ and $\hat{b}_i$ together in R_n (pointwise products): let $\hat{c}_i = \hat{a}_i\hat{b}_i$. The backward transform computes $(c'_i := \sum_{j=0}^{K-1} \omega^{-ij}\hat{c}_j)_{0\le i<K}$:

$$\begin{aligned} c'_i &= \sum_{j=0}^{K-1} \omega^{-ij}\hat{a}_j\hat{b}_j \\ &= \sum_{j=0}^{K-1} \omega^{-ij} \left(\sum_{l=0}^{K-1} \omega^{jl} a'_l\right)\left(\sum_{m=0}^{K-1} \omega^{jm} b'_m\right) \\ &= \sum_{l,m=0}^{K-1} a_l b_m \theta^{l+m} \sum_{j=0}^{K-1} \omega^{j(l+m-i)}. \end{aligned}$$

Since ω is a primitive Kth root of unity, $\sum_{j=0}^{K-1} \omega^{j(l+m-i)}$ is zero unless $l+m-i \equiv 0 \bmod K$, which holds for $l+m=i$ or $l+m=i+K$. Since $\theta^K \equiv -1 \bmod (2^n+1)$, it follows:

$$c'_i = K\theta^i \sum_{l=0}^{K-1} (a_l b_{i-l} - a_l b_{i+K-l}) = K\theta^i(c_i - c_{i+K}),$$

where b_m is assumed zero for m outside the range $[0, K-1]$.

SSA thus consists of five consecutive steps, where all computations in steps (2) to (4) are done modulo 2^n+1:

(1) the "decompose" step, which extracts from a the M-bit parts a_i, and multiplies them by the weight signal θ^i, obtaining a'_i (similarly for b_i);
(2) the "forward transform", which computes $(\hat{a}_0, \ldots, \hat{a}_{K-1})$ from $(a'_0, \ldots, a'_{K-1})$ (similarly for $\hat{b}_i$);
(3) the "pointwise product" step, which computes $\hat{c}_i = \hat{a}_i\hat{b}_i$, for $0 \le i < K$;
(4) the "backward transform", which computes $(c'_0, \ldots, c'_{K-1})$ from $(\hat{c}_0, \ldots, \hat{c}_{K-1})$;
(5) the "recompose" step, which divides c'_i by $2^k\theta^i$, and construct the final result as $\bar{c}_0 + \bar{c}_1 2^M + \cdots + \bar{c}_{K-1}2^{(K-1)M}$. Some $\bar{c}_i$, defined in Eq. (2), may be negative, but the sum is necessarily non-negative.

For a given input bit-size N, several choices of the FFT length K may be possible. SSA is thus a whole family of algorithms: we call FFT-K — or FFT-2^k — the algorithm splitting the inputs into $K = 2^k$ parts. For a given input size N, one of the main practical problems is how to choose the best value of the FFT length K, and thus of the bit-size n of the smaller multiplies (see §2.6).

1.1. **Choice of n and Efficiency.** SSA takes for n a multiple of K, so that $\omega = 2^{2n/K}$ is a primitive Kth root of unity, and $\theta = 2^{n/K}$ is used for the weight signal. This ensures that all FFT butterflies only involve additions/subtractions and shifts on a radix 2 computer (see §2.1).

In practice one may additionally require n to be a multiple of the word size w, to make the arithmetic in 2^n+1 simpler. Indeed, a number from R_n is then represented by n/w machine words, plus one additional bit of weight 2^n. We call this a *semi-normalized* representation, since values up to $2^{2n}-1$ can be represented.

For a given bit size N divisible by $K=2^k$, we define the *efficiency* of the FFT-K scheme:

$$\frac{2N/K+k}{n},$$

where n is the smallest multiple of K larger than or equal to $2N/K+k$. For example for $N=1,000,448$ and $K=2^{10}$, we have $2N/K+k=1964$, and the next multiple of K is $n=2048$, therefore the efficiency is $\frac{1964}{2048}\approx 96\%$. For $N=1,044,480$ with the same value of K, we have $2N/K+k=2050$, and the next multiple of K is $n=3072$, with an efficiency of about 67%. The FFT scheme is close to optimal when its efficiency is near 100%.

Note that a scheme with efficiency below 50% does not need to be considered. Indeed, this means that $2N/K+k\le\frac{1}{2}n$, which necessarily implies that $n=K$ (remember n has to be divisible by K). Then the FFT scheme of length $K/2$ can be performed with the same value of n, since $2(N/(K/2))+(k-1)<4N/K+2k\le n$, and n is a multiple of $K/2$.

From this last remark, we can assume $2N/K\ge\frac{1}{2}n$ — neglecting the small k term —, which together with $n\ge K$ gives:

$$K\le 2\sqrt{N}. \tag{3}$$

1.2. **Complexity Analysis.** The multiplication of two numbers in R_N, whose cost will be denoted by $M^+(N)$, reduces to K multiplications in R_n plus $O(K\log_2 K)$ butterflies in R_n, whose cost is $O(n)$ each. Assuming an efficiency close to 100%, we have $n\approx 2N/K$, thus:

$$M^+(N)\le KM^+(2N/K)+O(K\log_2 K)O(2N/K).$$

(The other operations have a cost $O(N)$, which may be contained in the $O(K\log_2 K)O(2N/K)$ term.) By choosing K such that $2N/K\le\lambda\sqrt{N}$ for some constant $\lambda>1$, and assuming $M^+(N)=c(N)N\log N$ with $c=o(\log N)$, this yields:

$$\begin{aligned} M^+(N) &\le 2c(\lambda\sqrt{N})N\log(\lambda\sqrt{N})+O(N\log N)\\ &= c(\lambda\sqrt{N})N\log N+O(N\log N).\end{aligned}$$

Dividing by $N\log N$, it follows:

$$c(N)\le c(\lambda\sqrt{N})+O(1),$$

which admits as solution $c(N)=O(\log\log N)$.

In practice one might want to choose for K the largest power of two satisfying Eq. (3), which guarantees an efficiency of at least 50%. But then the FFT of length $K/2$ might be better.

One big difference with FFT algorithms using complex numbers where the pointwise products involve small precision numbers (usually double-precision of 53 bits, i.e., of size $O(1)$) is that here, the pointwise products involve numbers of size $n=\Theta(\sqrt{N})$.

1.3. **GMP 4.1.4 and 4.2.1 Implementations.** Versions 4.1.4 and 4.2.1 of GMP have the same implementation for the multiplication in R_N. They only differ in the way this modular multiplication is used for the plain integer multiplication in $\mathbb{Z}$. Assume one wants to compute a non-negative integer product of at most ℓ bits.

GMP 4.1.4 uses a straightforward scheme: simply compute that product in R_N for $N \geq \ell$.

GMP 4.2.1 uses the following scheme: compute the product simultaneously in R_{2N} and R_{3N}, with $5N \geq \ell$, and reconstruct it using the Chinese Remainder Theorem (CRT). Indeed, $2^{2N}+1$ and $2^{3N}+1$ are relatively prime, and the CRT reconstruction can be performed efficiently, i.e., with additions/subtractions and shifts only.

Although the theoretical complexity of both schemes is the same, the GMP 4.2.1 scheme yields a smaller practical cost (see Fig. 3), and a smaller memory usage.

2. Our Improvements

We describe in this section the ideas and techniques we tried to improve the GMP implementation of SSA. We started from the GMP 4.2.1 implementation, and used the graph of the multiplication time up to $1,000,000$ words on an Opteron as benchmark. After encoding each idea, if the new graph was better than the old one, the new idea was validated, otherwise it was discarded. Each technique did save only 5% up to 20%, but all techniques together did save a factor of about 2 with respect to GMP 4.2.1

2.1. **Arithmetic Modulo 2^n+1.** Arithmetic operations modulo 2^n+1 have to be performed during the forward and backward transforms, when applying the weight signal, and when unapplying it. Thanks to the fact that the primitive roots of unity are powers of two, the only needed operations are additions, subtractions, and multiplications by a power of two. Indeed, divisions by 2^k can be reduced to multiplications by 2^{2n-k}.

We recall that we require n to be a multiple of the number w of bits per word. Since n must also be a multiple of $K = 2^k$, this is not a real constraint, unless $k < 5$ on a 32-bit computer, or $k < 6$ on a 64-bit computer. Let $m = n/w$ be the number of computer words corresponding to an n-bit number. A residue mod 2^n+1 has a semi-normalized representation with m full words and one carry of weight 2^n:

$$a = (a_m, a_{m-1}, \ldots, a_0),$$

with $0 \leq a_i < 2^w$ for $0 \leq i < m$, and $0 \leq a_m \leq 1$.

The addition of two such representations is done as follows (we give here the GMP code):

```
c = a[m] + b[m] + mpn_add_n (r, a, b, m);
r[m] = (r[0] < c);
MPN_DECR_U (r, m + 1, c - r[m]);
```

The first line adds $(a_{m-1}, \ldots, a_0)$ with $(b_{m-1}, \ldots, b_0)$, puts the low m words of the result in $(r_{m-1}, \ldots, r_0)$, and adds the out carry to $a_m + b_m$; we thus have $0 \leq c \leq 3$. The second line yields $r_m = 0$ if $r_0 \geq c$, in which case we simply subtract c from r_0 at the third line. Otherwise $r_m = 1$, and we subtract $c-1$ from r_0: a borrow may propagate, but at most to r_m. In all cases $r = a + b \bmod (2^n+1)$, and r is semi-normalized.

The subtraction is done in a similar manner:

```
c = a[m] - b[m] - mpn_sub_n (r, a, b, m);
r[m] = (c == 1);
```

```
MPN_INCR_U (r, m + 1, r[m] - c);
```

After the first line, we have $-2 \leq c \leq 1$. If $c = 1$, then $r_m = 1$ at the second line, and the third line does nothing. Otherwise, $r_m = 0$ at the second line, and we add $-c$ to r_0, where the carry may propagate up to r_m. In all cases $r = a - b \bmod (2^n + 1)$, and r is semi-normalized.

The multiplication by 2^e is more tricky to implement. However this operation mainly appears in the butterflies — see below — $[a, t] \leftarrow [a + b, (a - b)2^e]$ of the forward and backward transforms, which may be performed as follows:

```
Bfy(a, b, t, e)
   1. Write e = d*w + s with 0 <= s < w # w is the number of bits per word
   2. Decompose a = (ah, al) where ah contains the upper d words, idem for b
   3. t <- (al - bl, bh - ah)
   4. a <- a + b
   5. t <- t * 2^s
```

Step 3 means that the most significant words from t are formed with `al - bl`, and the least significant words with `bh - ah`, where we assume that borrows are propagated, so that t is semi-normalized. Thus the only real multiplication by a power of two is that of step 5, which may be efficiently performed with GMP's `mpn_lshift` routine.

If one has a combined `addsub` routine which computes simultaneously $x + y$ and $x - y$ faster than two separate calls, then step 4 can be written `a <- (bh + ah, al + bl)`, which shows that t and a may be computed with two `addsub` calls.

2.2. **Cache locality during the transforms.** When multiplying large integers with SSA, the time spent in accessing data for performing the Fourier transforms is non-negligible. The literature is rich of many papers dealing with the organization of the computations in order to improve the locality. However most of these papers are concerned with contexts which are different from ours: usually the coefficients are small and most often they are complex numbers represented as a pair of `double`'s. Also there is a variety of target platforms, from embedded hardware implementations to super-scalar computers.

We have tried to apply several of these approaches in our context which is the following:

- The target platform is a standard PC workstation;
- The size of the coefficients is at least a few cache lines;
- The coefficients are modular integers.

In this work, we concentrate on multiplying large, but not huge integers. By this we mean that we consider only 3 levels of memory for our data: L1 cache, L2 cache, and standard RAM. In the future we might consider also the case where we have to use the hard disk as a 4th level.

Here are the orders of magnitude for these memories, to fix ideas: on a typical Opteron, a cache line is 64 bytes; the L1 data cache is 64 kB; the L2 cache is 1 MB; the RAM is 8 GB. The smallest coefficient size (i.e., n-bit residues) we consider is about 50 machine words, that is 400 bytes. For very large integers, a single coefficient hardly fits in the L1 cache. For instance, in our implementation, when multiplying two integers of $105,000,000$ words modulo $2^N + 1$, a transform of length 2^{15} is used, with coefficients of size 52 kB.

In an FFT computation, the main operation is the butterfly operation that we detailed in the previous section. This is an operation in a ring R_n that computes $a + b$ and $(a - b)\omega$,

where a and b are coefficients in R_n and ω is some root of unity. In SSA this root of unity is always a power of 2. In the pseudo-code below we adopt the notation `Bfy(a, b, omega)` for a butterfly operation, instead of the complete one used in §2.1. Therefore, it must be understood that in fact a temporary buffer has to be used, and the `omega` that is passed is actually just a shift value, so that raising ω to some power is a multiplication of this shift value.

The very first FFT algorithm is the iterative one. In our context this is a really bad idea. The main advantage of it is that the data is accessed in a sequential way. In the case where the coefficients are small enough so that several of them fit in a cache line, this saves many cache misses. But in our case, contiguity is irrelevant due to the size of the coefficients compared to cache lines.

The next very classical FFT algorithm is the recursive one.

```
FFT(A, index, k, omega)
    if k == 0
        return;
    K = 2^{k-1}
    for i := 0 to K-1
        Bfy(A[i], A[i+K], omega^i)
    FFT(A, index,   k-1, omega^2)
    FFT(A, index+K, k-1, omega^2)
```

In this algorithm, at a certain level of recursion, we work on a small set of coefficients, so that they must fit in the cache. This version (or a variant of it) was implemented in GMP up to version 4.2.1. This behaves well for moderate sizes, but when multiplying large numbers, everything fits in the cache only at the tail of the recursion, so that most of the transform is already done when we are at last in the cache. The problem is that before getting to the appropriate recursion level, the accesses are very cache unfriendly.

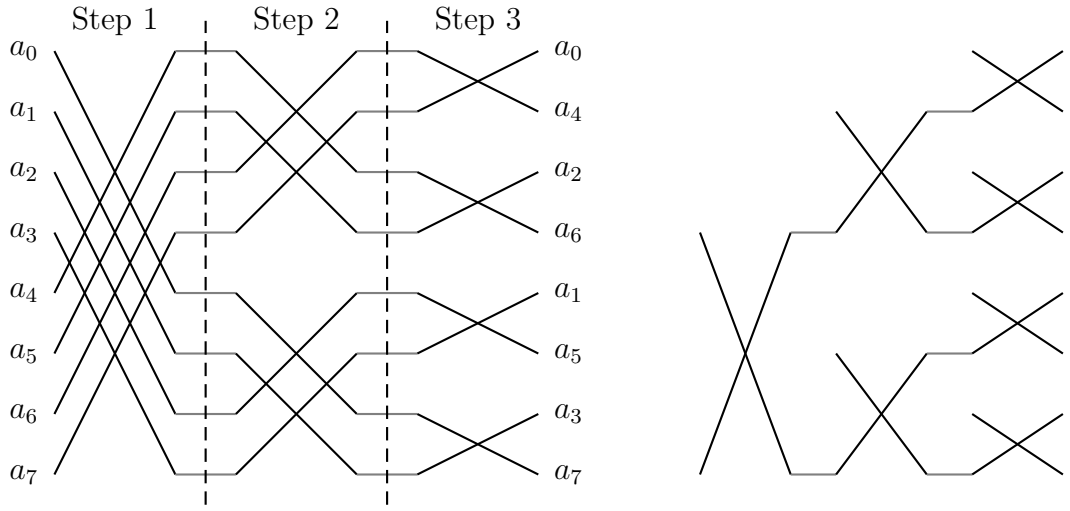
In order to improve the locality for large transforms, we have tried three strategies found in the literature: the Belgian approach, the radix-2^k transform, and Bailey's 4-step algorithm.

2.2.1. *The Belgian transform.* In [9], Brockmeyer et *al.* propose a way of organizing the transform that reduces cache misses. In order to explain it, let us first define a tree of butterflies as follows (we don't mention the root of unity for simplicity):

```
TreeBfy(A, index, depth, stride)
    Bfy(A[index], A[index+stride])
    if depth > 1
        TreeBfy(A, index-stride/2, depth-1, stride/2)
        TreeBfy(A, index+stride/2, depth-1, stride/2)
```

An example of a tree of depth 3 is given on the right of Figure 1. Now, the depth of a butterfly tree is bounded by a value that is not the same for every tree. For instance, on Figure 1, the butterfly tree that starts with the butterfly between a_0 and a_4 has depth 1: one can not continue the tree on step 2. Similarly, the tree starting with the butterfly between a_1 and a_5 has depth 1, the tree starting between a_2 and a_6 has depth 2 and the tree starting between a_3 and a_7 has depth 3. More generally, the depth can be computed by a simple formula.

FIGURE 1. The FFT circuit of length 8 and a butterfly tree of depth 3.



One can check that by considering all the trees of butterflies starting with an operation at step 1, we cover the complete FFT circuit. It remains to find the right ordering for computing those trees of butterflies. For instance, on the example of Figure 1, it is important to do the tree that starts between a_3 and a_7 in the end, since it requires data from all the other trees.

One solution is to perform the trees of butterflies following the `BitReverse` order. For an integer i whose binary representation fits in at most k bits, the value `BitReverse(i,k)` is the integer one obtains by reading the k bits (maybe padded with zeros), in the opposite order. One obtains the following algorithm, where `ord_2` stands for the number of trailing zeros in the binary representation of an integer (together with the 4-line `TreeBfy` routine, this is a recursive description of the 36-line routine from [9, Code 6.1]):

```
BelgianFFT(A, k)
    K = 2^{k-1}
    for i := 0 to K-1
        TreeBfy(A, BitReverse(i, k-1), 1+ord_2(i+1), K)
```

Inside a tree of butterflies, we see that most of the time, the butterfly operation will involve a coefficient that has been used just before, so that it should still be in the cache. Therefore a 50% cache-hit is guaranteed by construction, and we can hope for more if the data is not too large compared to the cache size.

We have implemented this in GMP, and this saved a few percent for large sizes, thus confirming the fact that this approach is better than the classical recursive transform.

2.2.2. *Higher radix transforms.* The principle of higher radix transforms is to use an atomic operation which groups several butterflies. In the book [1] the reader will find a description of several variants in this spirit. The classical FFT can be viewed as a radix-2 transform. The next step is a radix-4 transform, where the atomic operation has 4 inputs and 4 outputs (without counting roots of unity) and groups 4 butterflies of 2 consecutive steps of the FFT.

We can then build a recursive algorithm upon this atomic operation. Of course, since we perform 2 steps at a time, the number of steps in the recursion is reduced by a factor of 2, and we have to handle separately the last step when the FFT level k is odd. The

resulting code is the following, where the main operation is the group of 4 butterflies inside the for-loop.

```
Radix4FFT(A, index, k, omega)
    if k == 0
        return;
    if k == 1
        Bfy(A[index], A[index+1], 1);
        return;

    K1 = 2^{k-1}
    K2 = 2^{k-2}

    for j = 0 to K2-1 do
        Bfy(A, index+j,    index+j+K1,    omega^j);
        Bfy(A, index+j+K2, index+j+K1+K2, omega^(j+K2));
        Bfy(A, index+j,    index+j+K2,    omega^(2*j));
        Bfy(A, index+j+K1, index+j+K1+K2, omega^(2*j));
    end for;

    Radix4FFTrec(A, index,       k-2, omega^4);
    Radix4FFTrec(A, index+K2,    k-2, omega^4);
    Radix4FFTrec(A, index+K1,    k-2, omega^4);
    Radix4FFTrec(A, index+K1+K2, k-2, omega^4);
```

In the literature, the main interest for higher radix transforms comes from the fact that the number of operations is reduced for a transform of complex numbers (this is done by exhibiting a free multiplication by i). In our case, the number of operations remains the same. However, in the atomic block each input is used in two butterflies, so that the number of cache misses is less than 50%, just like in the Belgian approach. Furthermore, with the recursive structure, just like in the classical recursive FFT, at some point we deal with a number of inputs which is small enough so that everything fits in the cache.

We have tested this approach, and this was faster than the Belgian transform by a few percent.

The next step after radix 4 is radix 8 which works in the same spirit, but grouping 3 levels at a time. We have also implemented it, but this saved nothing, and was even sometimes slower than the radix 4 approach. Our explanation is that for small numbers, radix 4 is close to optimal with respect to cache locality, and for large numbers, the number of coefficients that fit in the cache is rather small and we have misses inside the atomic block of 12 butterflies. Further investigation is needed to validate this explanation.

More generally, radix 2^t groups t levels together, with a total of $t2^{t-1}$ butterflies, over 2^t residues. If all those residues fit in the cache, the cache miss rate is less than $1/t$. Thus the optimal strategy seems to choose for t the largest integer such that $2^t n$ bits fit in the cache (either L1 or L2, in fact the smallest cache where a single radix 2 butterfly fits).

2.2.3. *Bailey's 4-step algorithm.* The algorithm we describe in this section can be found in a paper by Bailey [3]. In there, the reader will find earlier references tracing back the original idea. For simplicity we stick to the "Bailey's algorithm" denomination.

A way of seeing Bailey's 4-step transform algorithm is as a radix-$\sqrt{K}$ transform, where $K = 2^k$ is the length of the input sequence. In other words, instead of grouping 2 steps as in radix-4, or 3 steps as in radix-8, we group $k/2$ steps. To be more general, let us write $k = k_1 + k_2$, where k_1 and k_2 are to be thought as close to $k/2$, but this is not really necessary. Then Bailey's 4-step algorithm consists in the following phases:

(1) Perform 2^{k_2} transforms of length 2^{k_1};
(2) Multiply the data by weights;
(3) Perform 2^{k_1} transforms of length 2^{k_2}.

There are only three phases in this description. The fourth phase is usually some matrix transposition[2], but this is irrelevant in our case: the coefficients are large so that we keep a table of pointers to them, and this transposition is just pointer exchanges which are basically for free, and fit very well in the cache.

The second step involving weights is due to the fact that in the usual description of Bailey's 4-step algorithm, the transforms of length 2^{k_1} are exactly Fourier transforms, whereas the needed operation is a twisted Fourier transform where the roots of unity involved in the butterflies are different (since they involve a 2^k-th root of unity, whereas the classical transform of length 2^{k_1} involves a 2^{k_1}-th root of unity). In the classical FFT setting this is very interesting, since we can then reuse some small-dimension implementation that has been very well optimized.

In our case, we have found it better to write a separate code for this twisted FFT, so that we merge the first and second phases. We obtain the following pseudo-code, in which before and after each call to the FFT or its twisted variant, we copy the appropriate data (i.e., pointers only) into an auxiliary table B. The transposition that occurs in the classical 4-step algorithm is still visible by the fact that the intricate loops of Phase 1 and Phase 3 are exchanged, thus looping on columns first in Phase 1 and on rows first in Phase 3.

```
Bailey(A, k, k1, k2, omega)
    K1 = 2^k1;
    K2 = 2^k2;

    // Phase 1:
    for i = 0 to K2-1 do
        for j := 0 to K1-1 do
            B[j] = A[i+K2*j]
        twistedFFT(B, i, k1, k, omega);
        for j = 0 to K1-1 do
            A[i+K2*j] = B[j];

    // No Phase 2!

    // Phase 3:
    for j := 0 to K1-1 do
        for i = 0 to K2-1 do
```

[2]Indeed, Bailey's algorithm might be viewed as a two-dimensional transform of a matrix with 2^{k_1} rows and 2^{k_2} columns, where Phase 1 performs 2^{k_2} one-dimensional transforms on the columns, and Phase 3 performs 2^{k_1} one-dimensional transforms on the rows.

```
            B[i] = A[i+K2*j]
        FFT(B, k2, omega^K1);
        for i = 0 to K2-1 do
            A[i+K2*j] = B[i];
```

The pseudo-code for the twisted FFT is as follows:

```
twistedFFTrec(A, index, k, omega, expo_curr, expo_mult)
    if k == 0 then
        return;
    K = 2^{k-1}
    for j := 0 to K-1 do
        root := omega^(expo_curr*j + expo_mult);
        Bfy(A[index+j], A[index+j+K], root);
    twistedFFTrec(A, index,   k-1, omega, 2*expo_curr, 2*expo_mult);
    twistedFFTrec(A, index+K, k-1, omega, 2*expo_curr, 2*expo_mult);

twistedFFT(A, rank, k1, k, omega)
    expo_curr = 2^{k-k1};
    twistedFFTrec(A, 1, k1, omega, 2^{k-k1}, rank);
```

The interest of this way of organizing the computation is again not due to a reduction of the number of operations, since they are exactly the same as with the other FFT approaches mentioned above. The goal is to help locality. Indeed, assume that $\sqrt{K}$ coefficients fit in the cache, then the number of cache misses is at most $2K$, since each call to the internal FFT or twisted FFT operates on $\sqrt{K}$ coefficients.

Of course we are interested in numbers for which $\sqrt{K}$ coefficients do not fit in the L1 cache, but for all numbers we might want to multiply, they do fit in the L2 cache. Therefore the structure of the code follows the memory hierarchy: at the top level of Bailey's algorithm, we deal with the RAM vs L2 cache locality question, then in each internal FFT or twisted FFT, we can take care of the L2 vs L1 cache locality question. This is done by using the radix-4 variant inside our Bailey-algorithm implementation.

We have implemented this approach (with a threshold for activating Bailey's algorithm only for large sizes), and combined with radix-4, this gave us our current best timings. We have also tried a higher dimensional transform, in particular 3 steps of size $\sqrt[3]{K}$. This did not help for the sizes we are currently considering.

This combination of Bailey's and radix-4 algorithm is the conclusion of our attempt to get the best static approach. A better idea might be to have a dynamic approach. Assuming that the sizes of the caches are known at run time, one could compute how many coefficients can fit in the L2 cache and decide of the number of levels to group accordingly. Then, inside the transform that corresponds to the L2 cache size, one can group again steps according to the number of coefficients that fit in the L1 cache.

Another variant that we have not fully tried is to keep Phase 2 and use a classical FFT instead of the twisted FFT for Phase 1. The additional cost of the multiplication by roots of unity could be balanced by the fact that inside the inner FFT, the butterflies will involve roots of unity of smaller order than in the twisted FFT. Therefore there is a larger proportion of the corresponding shifts that are going to be a multiple of the word length, which is an easy

operation compared to a general shift. For getting profit of that, one needs an implementation of the butterfly operation that takes full advantage of this property.

2.2.4. *Mixing several phases.* Another way to improve locality is to merge different phases of the algorithm in order to do as much work as possible on the data while they are in the cache. An easy improvement in this spirit is to mix the pointwise multiplication and the backward transform, in particular when Bailey's algorithm is used. Indeed, after the two forward transforms have been computed, one can load the data corresponding to the first column, do the pointwise multiplication of its elements, and readily perform the small transform of this column. Then the data corresponding to the second column is loaded, multiplied and transformed, and so on. In this way, one saves one full pass on the data. Taking the idea one step further, assuming that the forward transform for the first input number has been done already (or that we are squaring one number), after performing the column-wise forward transform on the second number we can immediately do the point-wise multiply and the backward transform on the column, so saving another pass over memory.

Following this idea, we can also merge the "decompose" and "recompose" steps (see p. 3) with the transform, again to save a pass on the data. In the case of the "decompose" step, there is more to it since one can also save unnecessary copies by merging it with the first step of the transform.

The "decompose" step consists of cutting parts of M bits from the input numbers, then multiplying each part a_i by θ^i modulo 2^n+1, giving a'_i. If one closely looks at the first FFT level (see for example the recursive FFT in §2.2), it will perform a butterfly between a'_i and $a'_{i+K/2}$ with θ^{2i} as multiplier. This will compute $a'_i + a'_{i+K/2}$ and $a'_i - a'_{i+K/2}$, and multiply the latter by θ^{2i}. It can be seen that the M non-zero bits from a'_i and $a'_{i+K/2}$ do not overlap, thus no real addition or subtraction is required: the results $a'_i + a'_{i+K/2}$ and $a'_i - a'_{i+K/2}$ can be obtained with just copies and one-complements. As a consequence, it should be possible to completely avoid the "decompose" step and the first FFT level, by directly starting from the second FFT level, which for instance will add $a'_i + a'_{i+K/2}$ to $(a'_j - a'_{j+K/2})\theta^{2j}$; here the four operands $a'_i, a'_{i+K/2}, a'_j, a'_{j+K/2}$ will be directly taken from the input integer a, and the implicit multiplier θ^{2j} will be used to know where to add or subtract a'_j and $a'_{j+K/2}$. This example illustrates the kind of savings obtained by avoiding trivial operations like copies and one-complements, and furthermore improving the locality. This idea is not yet implemented.

2.3. **Fermat and Mersenne Transforms.** The reason why SSA uses negacyclic convolutions is because the algorithm can be used recursively: the "pointwise products" modulo 2^n+1 can in turn be performed using the same algorithm, each one giving rise to K' smaller pointwise products modulo $2^{n'}+1$. (In that case, n must satisfy an additional divisibility condition related to K'.) A drawback of this approach is that it requires a weighted transform, i.e., additional operations before the forward transforms and after the backward transform. However, if one looks carefully, power-of-two roots of unity are needed only at the "lower level", i.e., in R_n^+. Therefore one can replace R_N by R_N^- — i.e., the ring of integers modulo 2^N-1 — in the original algorithm, and replace the weighted transform by a classical cyclic convolution, to compute a product mod 2^N-1. This works only at the top level of the algorithm, and not recursively. We call this a "Mersenne transform", whereas the original

SSA performs a "Fermat transform"[3]. This idea of using a Mersenne transform is already present in [4] where it is called "cyclic Schönhage-Strassen trick".

Despite the fact that it can be used at the top level only, the Mersenne transform is nevertheless very interesting for the following reasons:

- a Mersenne transform modulo $2^N - 1$, combined with a Fermat transform modulo $2^N + 1$ and CRT reconstruction, can be used to compute a product of two N-bit integers, in a similar way as the GMP 4.2.1 approach (see §1.3);
- a Mersenne transform can use a larger FFT length $K = 2^k$ than the corresponding Fermat transform. Indeed, while K must divide N for the Fermat transform, so that $\theta = 2^{N/K}$ is a power of two, it only needs to divide $2N$ for the Mersenne transform, so that $\omega = 2^{2N/K}$ is a power of two. In practice, this improves the efficiency for K near $\sqrt{N}$, and enables one to use a value of K close to optimal.

The constraint on the FFT length can still be decreased by using the "$\sqrt{2}$ trick" (see §2.4).

The above idea can be generalized to a Fermat transform mod $2^{aN} + 1$ and a Mersenne transform mod $2^{bN} - 1$ for small integers a, b.

Lemma 1. *Let a, b be two positive integers. Then at least one of* $\gcd(2^a + 1, 2^b - 1)$ *and* $\gcd(2^a - 1, 2^b + 1)$ *is* 1.

Proof. First define the subtractive-Euclidean sequence associated to a, b as follows: $a_0 = a$, $a_1 = b$, and for $i \geq 1$, as long as $a_i > 0$, $a_{i+1} = |a_i - a_{i-1}|$. When $a_i > 0$, we have $a_{i+1} := |a_i - a_{i-1}| < a_{i-1}$, thus the sum of two consecutive terms is strictly decreasing. We define the length of this sequence to be the index of the last non-zero term. For example the sequence corresponding to $a = 17, b = 10$ has length 9: $a_0 = 17, a_1 = 10, a_2 = 7, a_3 = 3, a_4 = 4, a_5 = 1, a_6 = 3, a_7 = 2, a_8 = 1, a_9 = 1, a_{10} = 0$.

Assume now that we want to compute the gcd, say g, of $2^a + \epsilon_a$ and $2^b + \epsilon_b$. We write $\epsilon_0 = \epsilon_a$, $\epsilon_1 = \epsilon_b$, and we use the subtractive-Euclidean sequence associated to a, b. Assume at some point we have $g = \gcd(2^{a_{i-1}} + \epsilon_{i-1}, 2^{a_i} + \epsilon_i)$. In the case $a_{i-1} \geq a_i$, we have $a_{i-1} = a_i + a_{i+1}$: we multiply $2^{a_i} + \epsilon_i$ by $2^{a_{i+1}}$, and subtract $2^{a_{i-1}} + \epsilon_{i-1}$ to obtain $\epsilon_i 2^{a_{i+1}} - \epsilon_{i-1}$. Multiplying by ϵ_i we get:

$$g = \gcd(2^{a_i} + \epsilon_i, 2^{a_{i+1}} - \epsilon_i \epsilon_{i-1}). \tag{4}$$

In the other case, $a_{i-1} < a_i$, we obtain the same formula.

Equation (4) gives $\epsilon_{i+1} = -\epsilon_i \epsilon_{i-1}$ for the sequence of signs. If follows: if $\epsilon_0 = \epsilon_1 = -1$, then $\epsilon_i = -1$ for all i; otherwise, the signs ϵ_i are periodic of period $[+, +, -]$. Indeed, if $\epsilon_0 = \epsilon_1 = -1$, all ϵ_i are -1. Otherwise, $(\epsilon_{i-1}, \epsilon_i) = (+, +)$ gives $\epsilon_{i+1} = -$, $(+, -)$ gives $+$, and $(-, +)$ gives $+$.

With $g = \gcd(a, b)$, and l the length of the subtractive-Euclidean sequence of (a, b), we deduce that:

- $\gcd(2^a - 1, 2^b - 1) = 2^g - 1$;
- $\gcd(2^a + 1, 2^b - 1) = 2^g + 1$ if $l \equiv 0 \bmod 3$, and 1 otherwise;
- $\gcd(2^a - 1, 2^b + 1) = 2^g + 1$ if $l \equiv 2 \bmod 3$, and 1 otherwise;
- $\gcd(2^a + 1, 2^b + 1) = 2^g + 1$ if $l \equiv 1 \bmod 3$, and 1 otherwise.

[3] In the whole paper, a Fermat transform, product, or scheme is meant modulo $2^N + 1$, without N being necessarily a power of two as in Fermat numbers.

With the above example, we have $l = 9$, thus $\gcd(2^{17}+1, 2^{10}-1) = 3$, and all other combinations give a trivial gcd. □

It follows from Lemma 1 that for two positive integers a and b, either $2^{aN}+1$ and $2^{bN}-1$ are coprime, or $2^{aN}-1$ and $2^{bN}+1$ are coprime, thus we can use one Fermat transform of size aN (respectively bN) and one Mersenne transform of size bN (resp. aN). However this does not imply that the reconstruction is easy: in practice we used $b = 1$ and make only a vary (see §2.6.2).

2.4. **The $\sqrt{2}$ trick.** Since all prime factors p of 2^n+1 are $p \equiv 1 \pmod 8$ if $4 \mid n$, 2 is a quadratic residue (mod n), and it turns out that $\sqrt{2}$ is of a simple enough form to make it useful as a root of unity with power-of-two order. Specifically,

$$\left(2^{3n/4} - 2^{n/4}\right)^2 \equiv 2 \pmod{2^n+1},$$

which is easily checked by expanding the square. Hence we can use $\sqrt{2} = 2^{3n/4} - 2^{n/4}$ as a root of unity of order 2^{k+2} in the transform to double the possible transform length for a given n. In the case of the negacyclic transform, this allows a length 2^{k+1} transform, and $\sqrt{2}$ is used only in the weight signal. For a cyclic transform, $\sqrt{2}$ is used normally as a root of unity during the transform, allowing a transform length of 2^{k+2}. This idea is mentioned in [4, §9] where it is credited without reference to Schönhage, but we have been unable to track down the original source. In our implementation, this $\sqrt{2}$ trick saved roughly 10% on the total time of integer multiplication.

It is natural to ask if even higher roots of unity can be used to effect even greater transform lengths. Unfortunately the answer is no, as prime divisors of 2^n+1 are not necessarily congruent to 1 (mod 2^{k+3}), so an order 2^{k+3} root of unity may not exist. It might be possible to perform a transform modulo a large divisor d of 2^n+1 where each prime divisor of d satisfies the required form, but this would require knowing the prime factorization of the 2^n+1 to be used and, on average, rule out half of the prime factors so that the useable size of 2^n+1 will likewise be reduced by half, thus negating the gain obtained from doubling the transform length.

For some 2^n+1, however, a root of unity of order 2^{k+3} is available. In example, for the eighth Fermat number, $F_8 = 2^{256}+1$, the complete factorization is known and the two prime factors p_1, p_2 happen to be $p_1 \equiv p_2 \equiv 1 \pmod{2^{11}}$. We therefore have an order 2048 root of unity ω, with $\omega^4 \equiv 2 \pmod{F_8}$. Unfortunately, there appears to be no simple expression for this ω in the form of a small sum of powers of two, as there is for $\sqrt{2}$. A multiplication by ω during the transform therefore requires a multiple precision multiplication of residues modulo F_8. Doubling the transform length by use of this ω means having to do four times as many multiple precision products which makes the use of fourth or higher roots of 2 inefficient.

We conclude that using roots of unity of higher order than 2^{k+2} is not feasible as it requires determining the factorizaton of 2^n+1, will usually work only modulo some of the prime factors of 2^n+1 and even then will lead to forbiddingly expensive arithmetic for lack of a simple form of the roots of unity so obtained.

2.5. **Harley's and Granlund's tricks.** Rob Harley [14] suggested the following trick[4] to improve the efficiency of a given FFT scheme. Assume $2M + k$ is just above an integer multiple of K, say λK. Then we have to use $n = (\lambda + 1)K$, which gives an efficiency of only about $\frac{\lambda}{\lambda+1}$. Harley's idea is to use $n = \lambda K$ instead, and recover the missing information from a CRT-reconstruction with an additional computation modulo the machine word 2^w.

Consider the following example: Take $N = 1,044,480$ with $K = 2^{10}$. We have seen p. 4 that $2N/K+k = 2050$, which gives $n = 3072$, and an efficiency of 67% only with the classical scheme.

Now if we use $n = 2048$, we will recover the coefficients of $C^+(x) \bmod 2^n + 1$ only, whereas each coefficient can take up to 2^{2050} different consecutive values. To recover the correct values of $c_i - c_{K+i}$ in Eq. (1), it suffices to compute the two missing bits, for example by computing $c_i - c_{K+i} \bmod 2^w$ where w is the machine word size in bits. Each $c_i - c_{K+i} \bmod 2^w$ can be computed with K word-by-word products $a_i[0]b_j[0]$, since `u*v` precisely computes $uv \bmod 2^w$ at the machine word level. All $c_i - c_{K+i} \bmod 2^w$ thus require K^2 word products, which is $O(N)$ effort since $K = O(\sqrt{N})$. The CRT reconstruction between $2^n + 1$ and 2^w is also easy to perform. With the above example, we thus trade one 3072-bit product for one 2048-bit product and 1024 word products, without counting the savings in the Fourier transforms.

A drawback of Harley's trick is that when only a few bits are missing, the K^2 word products may become relatively expensive. In the above example, they represent 2^{20} word products. Since only two bits are missing, we can multiply $A(x) \bmod 2^2$ and $B(x) \bmod 2^2$ over $\mathbb{Z}[x]$ using the segmentation method. Since the product coefficients have at most 14 bits, this corresponds to a product of two $(14 \cdot 2^{10})$-bit integers, or two 224-word integers on a 64-bit computer (instead of 2^{20} word products). In summary, if $h \leq w$ bits are missing, one trades K^2 word products for the product of two large integers of $(2h + k)K/w$ words, which can in turn use fast multiplication[5].

Torbjörn Granlund [13] found that this idea — combining computations mod $2^n + 1$ with computations mod 2^h — can also be used at the top-level for the plain integer multiplication, and not only at the lower-level as in Harley's trick. Assume one wants to multiply two integers u and v whose product has m bits, where m is just above an "optimal" Fermat scheme $(2^N + 1, K)$, say $m = N + h$. Then first compute $uv \bmod (2^N + 1)$, and secondly compute $uv \bmod 2^h$, by simply computing the plain integer product $(u \bmod 2^h)(v \bmod 2^h)$, again possibly in turn with fast multiplication. The exact value of uv can be efficiently reconstructed by CRT from both values. We call this idea "Granlund's trick".

With the notations from §1.2, Granlund's trick can be written $M(N+h) = M^+(N)+M(h)$, or $M(N + h) = M^+(N) + M^+(2h)$ if one reduces the plain product modulo 2^h to a modular product modulo $2^{2h} + 1$.

We use neither Harley's nor Granlund's trick in our current implementation. We believe Granlund's trick is less efficient than the generalized Fermat-Mersenne scheme we propose (§2.3), which yields $M(a + b) = M^+(a) + M^-(b)$, if a good efficiency is possible for $M^+(a)$ and $M^-(b)$. As for Harley's trick, we tried it only at the word level, i.e., for $\lambda K < 2M + k \leq \lambda K + w$, which happens in rare cases only, and it made little difference. However, when multiplying two numbers modulo a Fermat number $2^{2^n} + 1$, Harley's trick becomes very attractive, since $2M$ is a power of two in that case.

[4] Bernstein attributes a similar idea to Karp in [4, §9].

[5] Harley's trick extends to $h > w$: together with the segmentation method, the exact same reasoning holds.

2.6. **Improved Tuning.** Apart from ideas to avoid useless copies, and to improve cache locality, we found out that great speedups could be obtained with better tuning schemes, which we describe here. All examples given in that section are related to an Opteron, which is a 64-bit processor.

2.6.1. *Tuning the Fermat and Mersenne Transforms.* Until version 4.2.1, GMP used a naive tuning scheme for the FFT multiplication. For the Fermat transforms modulo $2^N + 1$, an FFT of length 2^k was used for $t_k \leq N < t_{k+1}$, where t_k is the smallest bit-size for which FFT-2^k is faster than FFT-2^{k-1}. For example on an Opteron, the default `gmp-mparam.h` file uses $k = 4$ for a size less than 528 machine words, then $k = 5$ for less than 1184 words, and so on:

```
#define MUL_FFT_TABLE  { 528, 1184, 2880, 5376, 11264, 36864, 114688, 327680,
                         1310720, 3145728, 12582912, 0 }
```

A special rule is used for the last entry: here $k = 14$ is used for less than $m = 12582912$ words, $k = 15$ is used for less than $4m = 50331648$ words, and then $k = 16$ is used. An additional single threshold determines up from which size — still in words — a Fermat transform mod $2^n + 1$ is faster than a full product of two n-bit integers:

```
#define MUL_FFT_MODF_THRESHOLD              544
```

For a product mod 2^n+1 of at least 544 words, GMP 4.2.1 therefore uses a Fermat transform, with $k = 5$ until 1183 words according to the above `MUL_FFT_TABLE`. Below that threshold, the algorithm used is the 3-way Toom-Cook algorithm, followed by a reduction mod $2^n + 1$.

This scheme is clearly not optimal, since the FFT-2^k curves cross each other several times, as shown by Figure 2.

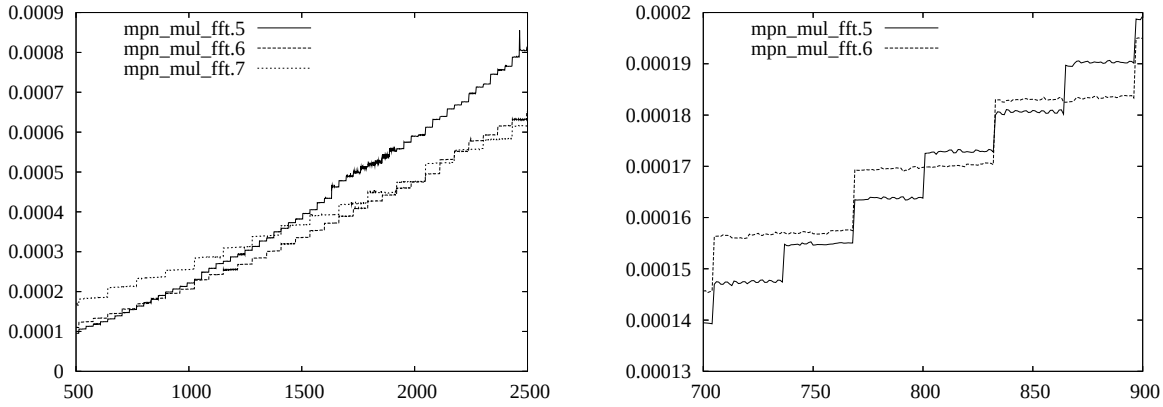


FIGURE 2. Time in seconds needed to multiply two numbers modulo $2^n + 1$ with an FFT of length 2^k for $k = 5, 6, 7$. On the right, the zoom (with only $k = 5, 6$) illustrates that two curves can cross each over several times.

To take into account those multiple crossings, the new tuning scheme determines word-intervals $[m_1, m_2]$ where the FFT of length 2^k is preferred for Fermat transforms:

```
#define MUL_FFT_TABLE2 {{1, 4 /*66*/}, {401, 5 /*96*/}, {417, 4 /*98*/},
     {433, 5 /*96*/}, {865, 6 /*96*/}, {897, 5 /*98*/}, {929, 6 /*96*/},
     {2113, 7 /*97*/}, {2177, 6 /*98*/}, {2241, 7 /*97*/}, {2305, 6 /*98*/},
     {2369, 7 /*97*/}, {3713, 8 /*93*/}, ...
```

The entry `{433, 5 /*96*/}` means that up from 433 words — and up to the next size of 865 words — FFT-2^5 is preferred, with an efficiency of 96%. A similar table is used for Mersenne transforms.

2.6.2. *Tuning the Plain Integer Multiplication.* Up to GMP 4.2.1, a single threshold controls the plain integer multiplication:

```
#define MUL_FFT_THRESHOLD                7680
```

This means that SSA is used for a product of two integers of at least 7680 words, which corresponds to about 148,000 decimal digits, and the Toom-Cook 3-way algorithm is used below that threshold.

We now use the generalized Fermat-Mersenne scheme described in §2.3 with $b = 1$ (in our implementation we found $1 \leq a \leq 7$ was enough). Again, for each size, the best value of a is determined by our tuning program:

```
#define MUL_FFT_FULL_TABLE2 {{16, 1}, {4224, 2}, {4416, 6}, {4480, 2},
    {4608, 4}, {4640, 2}, {4800, 1}, {5120, 2}, {5184, 1}, {5632, 2}, ...
```

For example, the entry `{4608, 4}` means that to multiply two numbers of 4608 words — or whose product has 2×4608 words — the new algorithm uses one Mersenne transform modulo $2^N - 1$ and one Fermat transform modulo $2^{4N} + 1$. Reconstruction is easy since $2^{aN} + 1 = 2 \bmod (2^N - 1)$.

3. Experimental Results and Conclusion

On July 1st, 2005, Allan Steel wrote a web page [22] entitled "*Magma V2.12-1 is up to 2.3 times faster than GMP 4.1.4 for large integer multiplication*". This was actually our first motivation for working on improving GMP's implementation.

Magma V2.13-6 takes 2.22s to multiply two numbers of 784141 words:

```
$ magma
Magma V2.13-6     Mon Dec 18 2006 11:13:14 on paris     [Seed = 2387452469]
Type ? for help.  Type <Ctrl>-D to quit.

> time Itest(784141,1);
Time: 2.220
```

whereas our GMP development code takes only 0.96s:

```
$ ./speed -s 784141 mpn_mul_n
             mpn_mul_n
784141     0.963854000
```

Thus our GMP-based code is clearly faster than Magma by a factor of 2.3. Note that this does not mean that we have gained a factor $2.3^2 = 5.29$ over GMP 4.1.4. In both cases, 2.3 is the maximal ratio between Magma V2.12-1 and GMP 4.1.4, and between our code and Magma V2.13-6 respectively, following the well known "benchmarketing" strategy[6] (both versions of Magma give very similar timings).

We have tested other freely available packages providing an implementation for large integer arithmetic. Among them, some do not go beyond Karatsuba algorithm (`OpenSSL/BN`, `LiDiA/libI`), some do have some kind of FFT, but are not really made for really large

[6]The word "benchmarketing" has been suggested to us by Torbjörn Granlund.

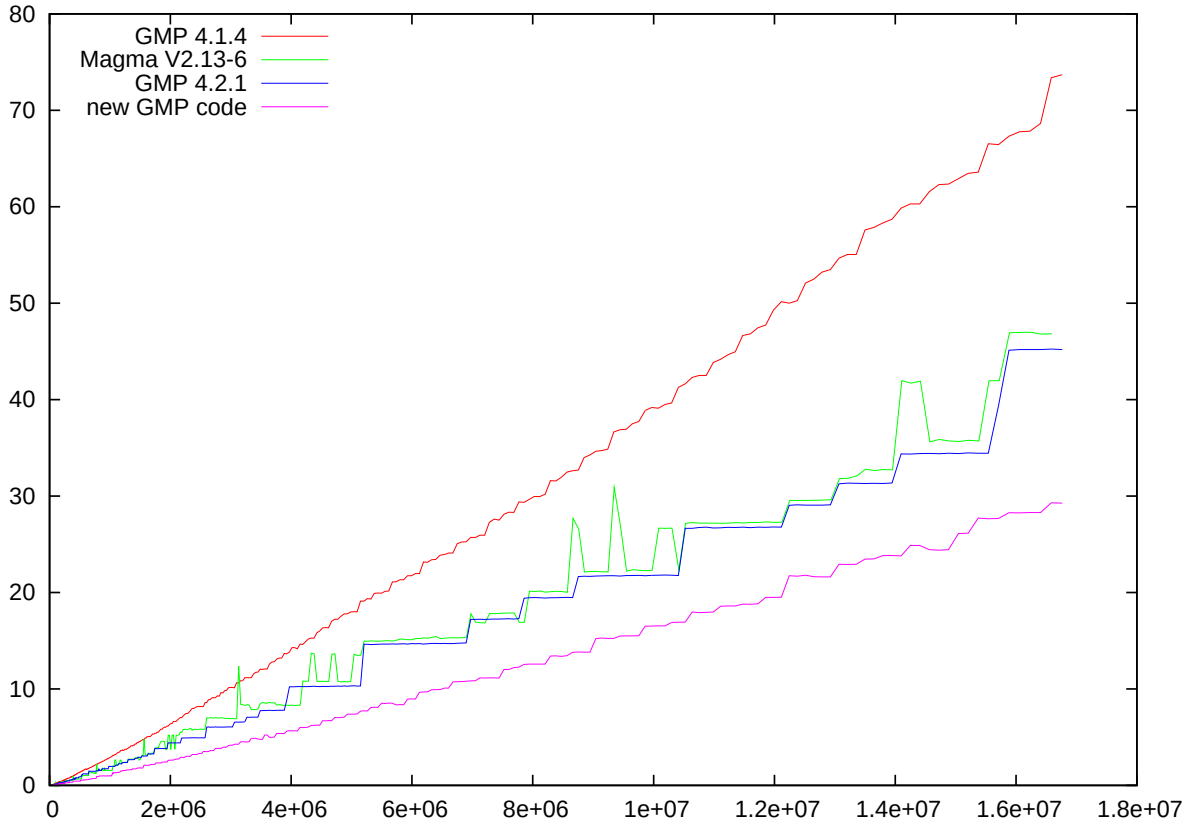


FIGURE 3. Comparison of GMP 4.1.4, GMP 4.2.1, Magma V2.13-6 and our new code for the plain integer multiplication on a 2.4Ghz Opteron (horizontal axis in 64-bit words, vertical axis in seconds).

integers (`arprec`, `Miracl`). Two useful implementations we have tested are `apfloat` and `CLN`. They take about 4 to 5 seconds on our test machine to multiply one million-word integers, whereas we need about 1 second. Bernstein mentions some partial implementation `Zmult` of Schönhage-Strassen's algorithm, with good timings, but right now, only very few sizes are handled, so that the comparison with our software is not really possible.

A program that implements a complex floating-point FFT for integer multiplication is George Woltman's `Prime95`. It is written mainly for the purpose of performing the Lucas-Lehmer algorithm for testing large Mersenne numbers $2^p - 1$ for primality in the Great Internet Mersenne Prime Search [26], and since its inception has found 10 new such primes, each one a new record at the time of its discovery. It uses a DWT for multiplication mod $a2^n \pm c$, with a and c not too large, see [19]. We compared multiplication modulo $2^{2wn} - 1$ in `Prime95` version 24.14.2 with multiplication of n-word integers using our SSA implementation on a Pentium 4 at 3.2 GHz, and on an Opteron 250 at 2.4 GHz, see Figure 4. It is plain that on a Pentium 4, `Prime95` beats our implementation by a wide margin, in fact usually by more than a factor of 10. On the Opteron, the difference is a bit less pronounced, where it is by a factor between 2.5 and 3. The reasons for this architecture dependency of the relative performance is that `Prime95` uses an SSE2 implementation of floating point FFT,

which performs slightly better on the Pentium 4 than on the Opteron at a given clock rate, but more importantly that all-integer arithmetic as in SSA performs poorly on the Pentium 4, but excellent on the Opteron, due to both native 64 bit arithmetic and a very efficient integer ALU. Some other differences between `Prime95` and our implementation need to be pointed out in this context: due to the floating point nature of `Prime95`'s FFT, rounding errors can build up for particular input data to the point where the result will be incorrectly rounded to integers. While occurring with only low probability, this trait may be undesirable in scientific computation. The floating point FFT can be made provably correct, see again [19], but at the cost of using larger FFT lengths, thus giving up some performance. Figure 5 shows the maximum number of bits that can be stored per FFT element of type double so that provably correct rounding is possible. `Prime95`'s default choice uses between 1.3 and 2 times as many, so for multiplication of large integers, demanding provably correct rounding would about double the run time. Also, the DWT in `Prime95` needs to be initialized for a given modulus, and this initialization incurs overhead which becomes very costly if numbers of constantly varying sizes are to be multiplied. Finally, the implementation of the FFT in `Prime95` is done entirely in hand-optimized assembly for the x86 family of processors, and will not run on other architectures.

Another implementation of complex floating point FFT is Guillermo Ballester Valor's `Glucas`. The algorithm it uses is similar to that in `Prime95`, but it is written portably in C. This makes it slower than `Prime95`, but still faster than our code on both the Pentium 4 and the Opteron, as shown in Figure 4.

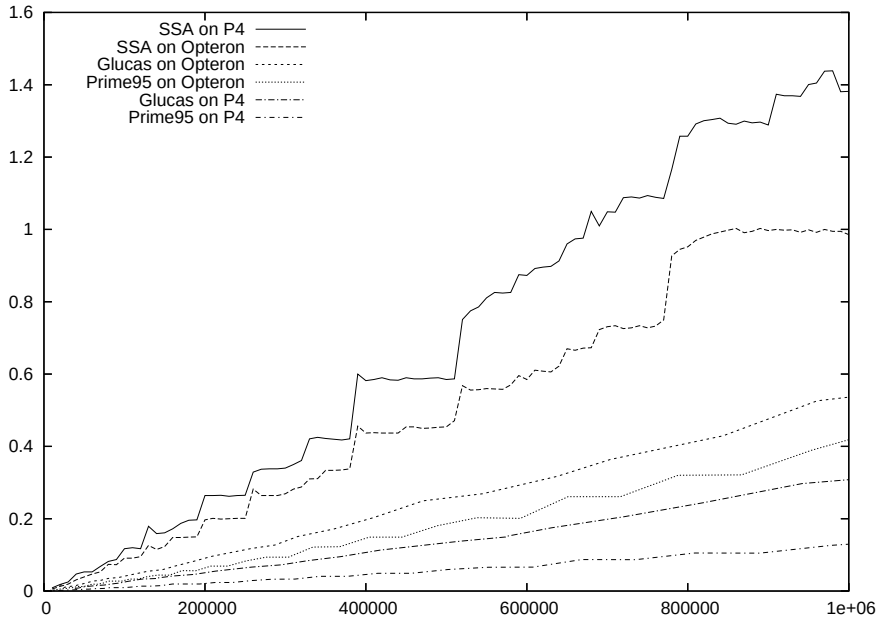


FIGURE 4. Time in seconds for multiplying integers of different word lengths with our SSA implementation, Prime95 and Glucas on a 3.2 GHz Pentium 4 and a 2.4 GHz Opteron. (Prime95 and Glucas use a floating-point FFT, which may give incorrect results in rare cases.)

We conclude this report with a list of ideas we did not try yet.

$K = 2^k$	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{21}	2^{23}	2^{25}
bits/dbl	16	15	14	13	12	11	10	9
N	32768	122880	458752	1703936	6291456	11534336	41943040	150994944
`Prime95`	21.37	21.08	20.78	20.49	20.22	19.94	18.29	17.76

FIGURE 5. Maximal number of bits which can be stored in a double-precision number (53 bits of precision) for different FFT lengths K, and corresponding maximal bit-size N for plain integer multiplication with provably correct rounding according to [19, Th. 5.1], and the number of bits per FFT element used in `Prime95`.

- William Hart and David Harvey suggest [15] to use a two-complement representation for the residues modulo $2^n + 1$, instead of a sign-magnitude representation. This means that a "negative" number x is represented by $2^{n+w} - x$, where w is the word size in bits, and a positive number x is represented by x. As long as the absolute values x do not exceed 2^{n+w-1}, addition and subtraction can be performed by directly calling the low-level GMP routines `mpn_add_n` and `mpn_sub_n`.
- In Schönhage-Strassen's algorithm, a significant proportion of the time is spent in the point-wise multiplications (which can again be done with the same algorithm recursively). Therefore it makes sense to try to optimize this step. We plan to implement specific functions for the FFT of lengths 2^4 and 2^5, where all the loops and functions calls are unrolled and inlined. Another plan is to use Bodrato and Zanoni's optimal Toom-Cook inversion sequences [7], but using a different interpolation phase to take the wraparound effect into account. Consider for example Toom-Cook 4-way: the classical interpolation sequence identifies the seven coefficients of $c_0 + c_1 x + \cdots + c_6 x^6$, whereas we only need to compute $c_0 - c_4$, $c_1 - c_5$, $c_2 - c_6$, and c_3.
- The truncated Fourier transform [24] might be of some use in our context in order to smooth the stairs. Right now, the generalized Mersenne/Fermat strategy already helps in this respect, but one could give it a try.
- As mentioned in §2.2.4, we still have ideas on how to save some cache misses by merging several phases.
- Mix SSA and a floating point FFT. At the high recursion level, SSA's all integer arithmetic can be used to allow for arbitrarily large input numbers without danger of loss of accuracy, and the floating-point approach for the pointwise multiplication when the required transform lengths are short enough to allow provably correct rounding. For example, on a Pentium 4, FFTW-3.1.2 [12] performs a cyclic convolution of length 2^{12} with 16 bits per double about 27% times faster than GMP 4.1.2.
- Another possibility is to use a number theoretic transform, using a few prime numbers that are specifically well-suited for FFT. There might be some platforms where this is faster than Schönhage-Strassen's algorithm.

Acknowledgments. This work was done in collaboration with Torbjörn Granlund, during his visits as invited professor at INRIA Lorraine in March-April and November-December 2006; we also thank him for proof-reading this paper. This work would probably not have been achieved without the initial stimulation from Allan Steel; the authors are very grateful to him. Thanks to Markus Hegland for pointing to the Belgian paper.

References

[1] Arndt, J. *Algorithms for programmers (working title)*. Draft version of 2007-January-05, http://www.jjj.de/fxt/.

[2] Bailey, D. The computation of π to 29,360,000 decimal digits using Borwein's quartically convergent algorithm. *Mathematics of Computation 50* (1988), 283–296.

[3] Bailey, D. FFTs in external or hierarchical memory. *J. Supercomputing 4* (1990), 23–35.

[4] Bernstein, D. J. Multidigit multiplication for mathematicians. http://cr.yp.to/papers.html#m3, 2001.

[5] Bernstein, D. J. Fast multiplication and its applications. http://cr.yp.to/papers.html#multapps, 2004. 47 pages.

[6] Bernstein, D. J. Removing redundancy in high-precision Newton iteration. http://cr.yp.to/fastnewton.html, 2004. 13 pages.

[7] Bodrato, M., and Zanoni, A. What about Toom-Cook matrices optimality? http://bodrato.it/papers/#CIVV2006, Oct. 2006. 14 pages.

[8] Brent, R. P. Multiple-precision zero-finding methods and the complexity of elementary function evaluation. In *Analytic Computational Complexity* (New York, 1975), J. F. Traub, Ed., Academic Press, pp. 151–176.

[9] Brockmeyer, E., Ghez, C., D'Eer, J., Catthoor, F., and Man, H. D. Parametrizable behavioral IP module for a data-localized low-power FFT. In *Proc. IEEE Workshop on Signal Processing Systems (SIPS)* (Taipeh, Taiwan, 1999), IEEE Press, pp. 635–644.

[10] Crandall, R., and Fagin, B. Discrete weighted transforms and large-integer arithmetic. *Mathematics of Computation 62*, 205 (1994), 305–324.

[11] Crandall, R., and Pomerance, C. *Prime Numbers: A Computational Perspective*. Springer-Verlag, 2000.

[12] Frigo, M., and Johnson, S. G. *FFTW (Fastest Fourier Transform in the West)*, 3.1.2 ed. http://fftw.org/.

[13] Granlund, T. Personal communication, Dec. 2006.

[14] Harley, R. Personal communication, Jan. 2000.

[15] Hart, W., and Harvey, D. Personal communication, Dec. 2006.

[16] Karp, A. H., and Markstein, P. High-precision division and square root. *ACM Trans. Math. Softw. 23*, 4 (1997), 561–589.

[17] Knuth, D. The analysis of algorithms. In *Actes du Congrès International des Mathématiciens de 1970* (Paris, 1971), vol. 3, Gauthiers-Villars, pp. 269–274.

[18] Moenck, R., and Borodin, A. Fast modular transforms via division. In *Proceedings of the 13th Annual IEEE Symposium on Switching and Automata Theory* (Oct. 1972), pp. 90–96.

[19] Percival, C. Rapid multiplication modulo the sum and difference of highly composite numbers. *Mathematics of Computation 72*, 241 (2003), 387–395.

[20] Schönhage, A. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica 1* (1971), 139–144.

[21] Schönhage, A., and Strassen, V. Schnelle Multiplikation großer Zahlen. *Computing 7* (1971), 281–292.

[22] Steel, A. Magma V2.12-1 is up to 2.3 times faster than GMP 4.1.4 for large integer multiplication. http://magma.maths.usyd.edu.au/users/allan/intmult.html, July 2005.

[23] Steel, A. Reduce everything to multiplication. Computing by the Numbers: Algorithms, Precision, and Complexity, Workshop for Richard Brent's 60th birthday, Berlin, July 2006. http://www.mathematik.hu-berlin.de/~gaggle/EVENTS/2006/BRENT60/.

[24] van der Hoeven, J. The truncated fourier transform and applications. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation (ISSAC)* (Santander, Spain, 2004), J. Gutierrez, Ed., pp. 290–296.

[25] von zur Gathen, J., and Gerhard, J. *Modern Computer Algebra*. Cambridge University Press, 1999.

[26] Woltman, G., and Kurowski, S. *The Great Internet Mersenne Prime Search*. `http://www.gimps.org/`.

[27] Zimmermann, P., and Dodson, B. 20 years of ECM. In *Proceedings of the 7th Algorithmic Number Theory Symposium (ANTS VII)* (Berlin Heidelberg, 2006), F. Hess, S. Pauli, and M. Pohst, Eds., vol. 4076 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 525–542.