

Efficient dot product over word-size finite fields

Jean-Guillaume Dumas

Laboratoire de Modélisation et Calcul. 50 av. des Mathématiques, B.P. 53, 38041 Grenoble, France.
Jean-Guillaume.Dumas@imag.fr, www-lmc.imag.fr/lmc-mosaic/Jean-Guillaume.Dumas

Abstract. We want to achieve efficiency for the exact computation of the dot product of two vectors over word size finite fields. We therefore compare the practical behaviors of a wide range of implementation techniques using different representations. The techniques used include floating point representations, discrete logarithms, tabulations, Montgomery reduction, delayed modulus. Our implementations have many symbolic linear algebra applications: matrix multiplication, symbolic triangularization, system solving, exact determinant computation, matrix normal form are such examples.

1 Introduction

Finite fields play a crucial role in computational algebra. Indeed, finite fields are the basic representation used to solve many integer problems. The whole solutions are then gathered via the Chinese remainders or lifted p -adically. Among those problems are integer polynomial factorization [21], integer system solving [3], integer matrix normal forms [8] or integer determinant [13]. Finite fields are also of intrinsic use, especially in cryptology (for instance for large integer factorization [16], discrete logarithm computations [17]) or for error correcting codes.

Moreover, nearly all of these problems involve linear algebra resolutions and dot product is the core of many linear algebra routines. Fast routines for matrix multiplication [6, §3.3.3], triangular system solving and matrix factorizations [7, §3.3] on the one hand, iterative methods on the other hand (see e.g. [20, 14, 9], etc.) make extensive usage of an efficient dot product over finite fields. In this paper, we compare the practical behavior of many implementations of this essential routine. Of course, this behavior is highly dependent on the machine arithmetic and on the finite field representations used.

Our focus is more specifically on prime fields of word size cardinality at most, with any characteristic. Section 2 therefore proposes several possible representations while section 3 presents some algorithms for the dot product itself. Experiments and choice of representation/algorithm are also presented in section 3.

2 Prime field representations

We present here various methods implementing seven of the basic arithmetic operations:

- the addition.
- the subtraction.
- the negation.
- the multiplication.
- the division.
- a multiplication followed by an addition ($r \leftarrow a * x + y$) or *AXPY* (also called “fused-mac” within hardware).
- a multiplication followed by an in-place addition ($r \leftarrow a * x + r$) or *AXPYIN*.

Within linear algebra in general (e.g. Gaussian elimination, or matrix-vector iterations) and for dot product in particular, these last two operations are the most widely used. We now present different ways to implement these operations.

2.1 Classical representation with integer division

The classical representation, with positive values between 0 and $p - 1$, for p the prime, will be denoted by “ Z_pz ”.

- Addition is a signed addition followed by a test. An additional subtraction of the modulus is made when necessary.
- Subtraction is similar.
- Multiplication is machine multiplication and machine remainder.
- Division is performed via the extended gcd algorithm.
- *AXPY* is a machine multiplication and a machine addition followed by only one machine remainder.

For the results to be correct, the intermediate *AXPY* value must not overflow. For a m -bit machine integer, the prime must therefore be below $2^{\frac{m-1}{2}} - 1$ if signed values are used. For 32 and 64 bits this gives primes below 46337 and below 3037000493.

Note that with some care those operations can be extended to work with unsigned machine integers (e.g. an additional test is required for the subtraction). The possible primes then being below 65521 and 4294967291.

2.2 Montgomery representation

To avoid the costly machine remainders, Montgomery designed another reduction [15]:

when $\gcd(p, B) = 1$, $n_{im} \equiv -p^{-1} \pmod{B}$ and T is such that $0 \leq T \leq pB$,
 if $U \equiv Tn_{im} \pmod{B}$,
 then $(T + Up)/B$ is an integer and $(T + Up)/B \equiv TB^{-1} \pmod{p}$.

The idea of Montgomery is to set B to half the word size. Thus, multiplication and divisions by B will just be shifts and remaindering by B is just the application of a bit-mask. Then, one can use the reduction to perform the remainderings by p . Indeed the example implementation of this reduction shown below with one shift, two bit-masks and two machine multiplications is often much less expensive than a machine remaindering:

Montgomery reduction

```
#define MASK 65535UL
#define B 65536UL
#define HALF_BITS 16
/* nim is precomputed to -1/p mod B with the extended gcd */
...
unsigned long c0 (c & MASK);          /* c      mod B      */
c0 = (c0 * nim) & MASK;              /* -c/p  mod B      */
c += c0 * p;                         /* c = 0 mod B      */
c >>= HALF_BITS;                    /* high bits of c    */
return (c > p ? c - p : c);          /* c is between 0 and 2p */
```

The idea is then to change the representation of the elements: every element a is stored as $aB \pmod{p}$. Then additions, subtractions are unchanged and the prime field multiplication is now a machine multiplication, followed by only one Montgomery reduction. Nevertheless, one has to be careful when implementing the *AXPY* operator since axB^2 cannot be added to yB directly. We will see section 3.2 that this is actually not a problem for the dot product. Finally, the primes must verify $(p - 1)^2 + p * (B - 1) < B^2$, which gives $p \leq 40499$ (resp. $p \leq 2654435761$) for $B = 2^{16}$ (resp. $B = 2^{32}$).

2.3 Floating point representation

Yet another way to perform the reduction is to use the floating point routines. According to [2, Table 3.1], for most of the architectures (alpha, AMD, Pentium IV, Itanium, Sun Solaris, etc.) those routines are faster than the integer ones (except for the Pentium III). The idea is then to compute $T \bmod p$ by way of a precomputation of a high precision numerical inverse of p :

$$T \bmod p = T - \lfloor T * \frac{1}{p} \rfloor * p.$$

The idea here is that floating point division and truncation are quite fast when compared to machine remaindering. Now on floating point architectures the round-off can induce a ± 1 error when the flooring is computed. This requires then an adjustment as implemented e.g. in Shoup's NTL [18] :

NTL's floating point reduction

```
double P, invP, T;
...
T -= floor(T*invP)*P;
if (T >= P) T -= P;
else if (T < 0) T += P;
```

2.4 Discrete logarithms

This representation is also known as Zech logarithms, see e.g. [4] and references therein. The idea is to use a generator of the multiplicative group, namely a primitive element. Then, every non zero element is a power of this primitive element and this exponent can be used as an internal representation:

$$\begin{cases} 0 & \text{if } x = 0 \\ q-1 & \text{if } x = 1 \\ i & \text{if } x = g^i \text{ and } 1 \leq i < q-1 \end{cases}$$

Then many tricks can be used to perform the operations that require some extra tables see e.g. [11, 1]. This representation can be used for prime fields as well as for their extensions, we will therefore use the notation GF q . The operations are then:

- Multiplication and division of invertible elements are just an index addition and subtraction modulo $\bar{q} = q - 1$.
- Negation is identity in characteristic 2 and addition of $i_{-1} = \frac{q-1}{2}$ modulo \bar{q} in odd characteristic.
- Addition is now quite complex. If g^i and g^j are invertibles to be added then their sum is $g^i + g^j = g^i(1 + g^{j-i})$. The latter can be implemented using index addition and subtraction and access to a "plus one" table (*t_plus1*[]) of size q . This table gives the exponent h of any number of the form $1 + g^k$, so that $g^h = 1 + g^k$.

Table 1 shows the number of elementary operations to implement Zech logarithms for an odd characteristic finite field. Only one table of size q is considered. Those operations are of three types: mean number of exponent additions and subtractions (+/-), number of tests and number of table accesses.

We have counted 1.5 index operations when a correction by \bar{q} actually arises only for half the possible values. The fact that the mean number of index operations is 3.75 for the subtraction is easily proved in view of the possible values taken by $j - i + \frac{q-1}{2}$ for varying i and j . In this case, $j - i + i_{-1}$ is between $-\frac{\bar{q}}{2}$ and $\frac{3\bar{q}}{2}$ and requires a correction by \bar{q} only two eighth of the time as shown in figure 1 for $q = 101$.

| Operation | Elements | Indices | Cost | | |
|----------------|-------------|---|------|-------|----------|
| | | | +/- | Tests | Accesses |
| Multiplication | $g^i * g^j$ | $i + j \ (-\bar{q})$ | 1.5 | 1 | 0 |
| Division | g^i / g^j | $i - j \ (+\bar{q})$ | 1.5 | 1 | 0 |
| Negation | $-g^i$ | $i - i_{-1} \ (+\bar{q})$ | 1.5 | 1 | 0 |
| Addition | $g^i + g^j$ | $k = j - i \ (+\bar{q})$ $i + t_plus1[k] \ (-\bar{q})$ | 3 | 2 | 1 |
| Subtraction | $g^i - g^j$ | $k = j - i + i_{-1} \ (\pm\bar{q})$ $i + t_plus1[k] \ (-\bar{q})$ | 3.75 | 2.875 | 1 |

Table 1. Number of elementary operations to implement Zech logarithms for an odd characteristic

The total number of additions or subtractions is then $2 + 0.25 + 1 + 0.5 = 3.75$ and the number of tests $1 + 0.875 + 1 = 2.875$ follows (one test towards zero, one only in case of a positive value i.e. seven eighth of the time, and a last one after the table lookup). It is possible to actually reduce the number of index exponents, (for instance replacing $i + x$ by $i + x - \frac{q-1}{2}$) but to the price of an extra test. In general, such a test ($a > q$?) is as costly as the $a - q$ operation. We therefore propose an implementation minimizing the total cost with a single table.

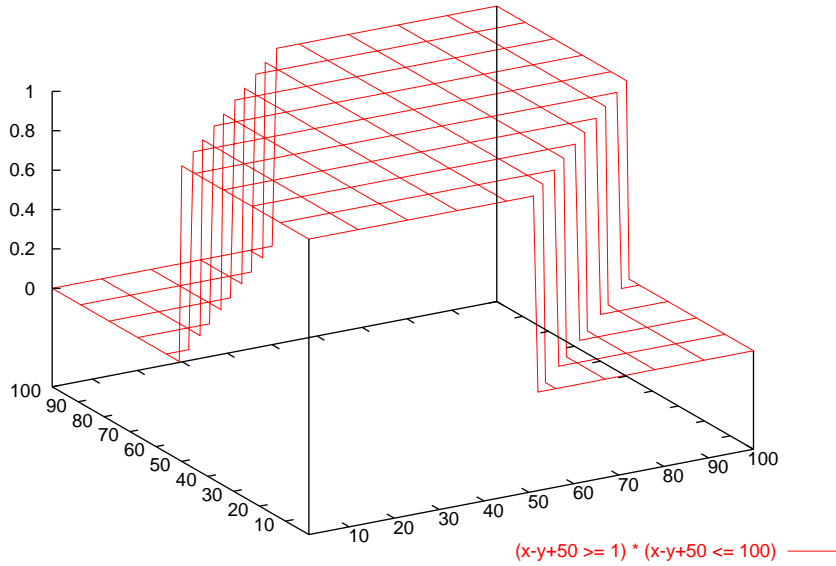


Fig. 1. $j - i + \frac{q-1}{2}$ for i and j between 1 and $q - 1$, for $q = 101$

These operations are valid as long as the index operations do not overflow, since those are just signed additions or subtractions. This gives maximal prime value of e.g. 1073741789 for 32 bits integer. However, the table size is now a limiting factor: indeed with a field of size 2^{26} the table is already 256 Mb. Table reduction is then mandatory. We will not deal with such optimizations in this paper, see e.g. [12, 4] for more details.

Fully tabulated A last possibility is to further tabulate the Zech logarithm representation. The idea is to code 0 by $2\bar{q}$ instead of 0. Then a table can be made for the multiplication:

- $t_mul[k] = k$ for $0 \leq k < \bar{q}$.
- $t_mul[k] = k - \bar{q}$ for $q - 1 \leq k < 2\bar{q}$.
- $t_mul[k] = 2\bar{q}$ for $2\bar{q} \leq k \leq 4\bar{q}$.

The same can be done for the division via a shift of the table and creation of the negative values, thus giving a table of size $5q$. For the addition, the t_plus1 has also to be extended to other values, to a size of $4q$. For subtraction, an extra table of size $4q$ has also to be created. When adding the back and forth conversion tables, this gives a total of $15q$. Even with some table reduction techniques, this becomes quite huge and quite useless nowadays when memory accesses are a lot more expensive than arithmetic operations [5].

2.5 Field extensions

In this paper, we mainly focus on prime fields. Notwithstanding, some of the presented implementations can be used for other fields or rings. The classical representation can e.g. be used with non prime modulus. Extension fields, or Galois fields of size p^d can also be implemented the following way:

1. **Integer division** Extension fields would be implemented with polynomial arithmetic.
2. **Montgomery Reduction** Direct use of Montgomery reduction is not possible, but there exists some efficient polynomial reductions. See e.g. [1] and references therein.
3. **Floating point** One can store the extensions elements as a floating point evaluation of the polynomials to a prime q different from the characteristic of the field. This “q-adic” methods works for some small extensions as demonstrated in [6].
4. **Zech logarithms** A very interesting property is that whenever this implementation is not at all valid for non prime modulus, it remains identical for field extensions. Indeed one can also find generators modulo an irreducible polynomial or even build extensions with primitive polynomials (X is thus a generator) [10, 5]. In this case the classical representation would introduce polynomial arithmetic. This discrete logarithm representation, on the contrary, would remain atomic, thus inducing a speed-up factor of $O(d^2)$, for d the extension degree. See e.g. [6, §4] for more details.

2.6 Atomic comparisons

We first present a comparison between the preceding implementation possibilities. The idea is to compare just the atomic operations. “%” denotes an implementation using machine remaindering (machine division) for every operation. This is just to give a comparing scale. “NTL” denotes NTL’s floating point flooring for multiplication ; “Z/pZ” denotes our implementation of the classical representation when tests ensure that machine remaindering is used only when really needed. Last “GFq” denotes the discrete logarithm implementation of section 2.4. In order to be able to compare those single operations, the experiment is an application of the arithmetic operator on vectors of a given size (e.g. 256 for figures 2, 3 and 4).

We compare the number of millions of field arithmetic operations per second, *Mop/s*.

Figure 2 shows the results on a UltraSparc II 250 Mhz, with compiler “gcc version solaris2.9/3.3.2”. First one can see that the need of Euclid’s algorithm for the field division is a huge drawback of all the implementations save one. Indeed division over “GFq” is just an index subtraction. Next, we see that floating point operations are quite faster than integer operations: NTL’s multiplication is better than the integer one. Now, on this machine, memory accesses are not yet much slower than arithmetic operations. This is the reason why discrete logarithm addition is only 2 to 3 times slower than one arithmetic call. This, enables the “GFq” AXPY (base operation of most of the linear algebra operators) to be the fastest.

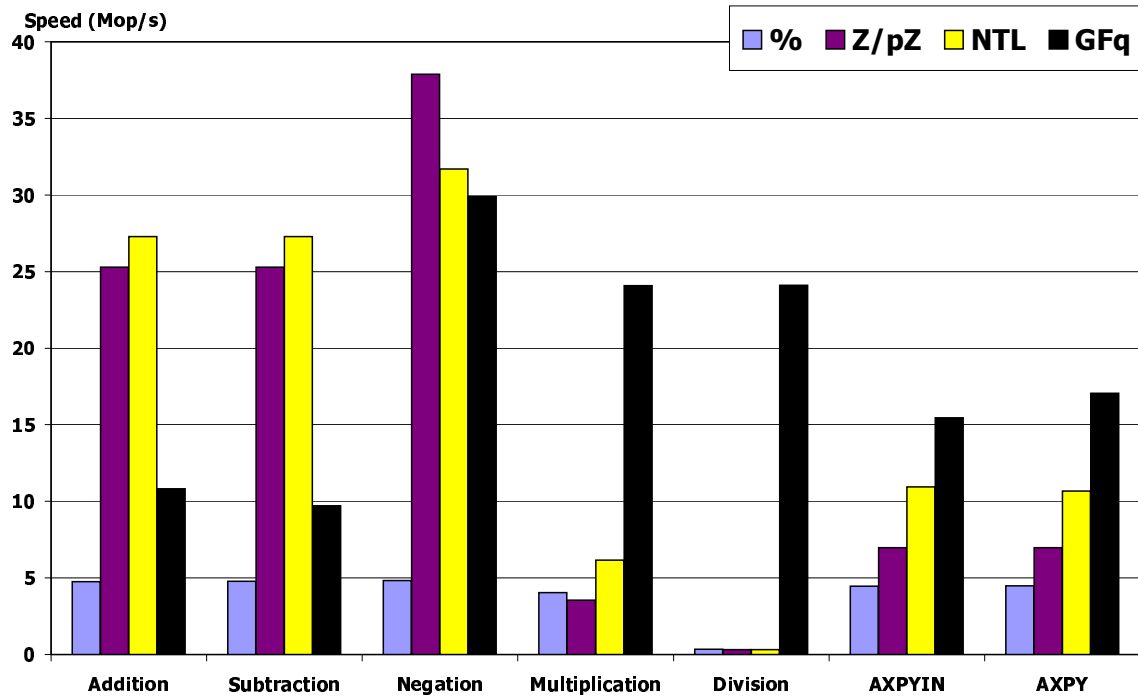


Fig. 2. Single arithmetic operation modulo 32749 on a sparc ultra II, 250 MHz

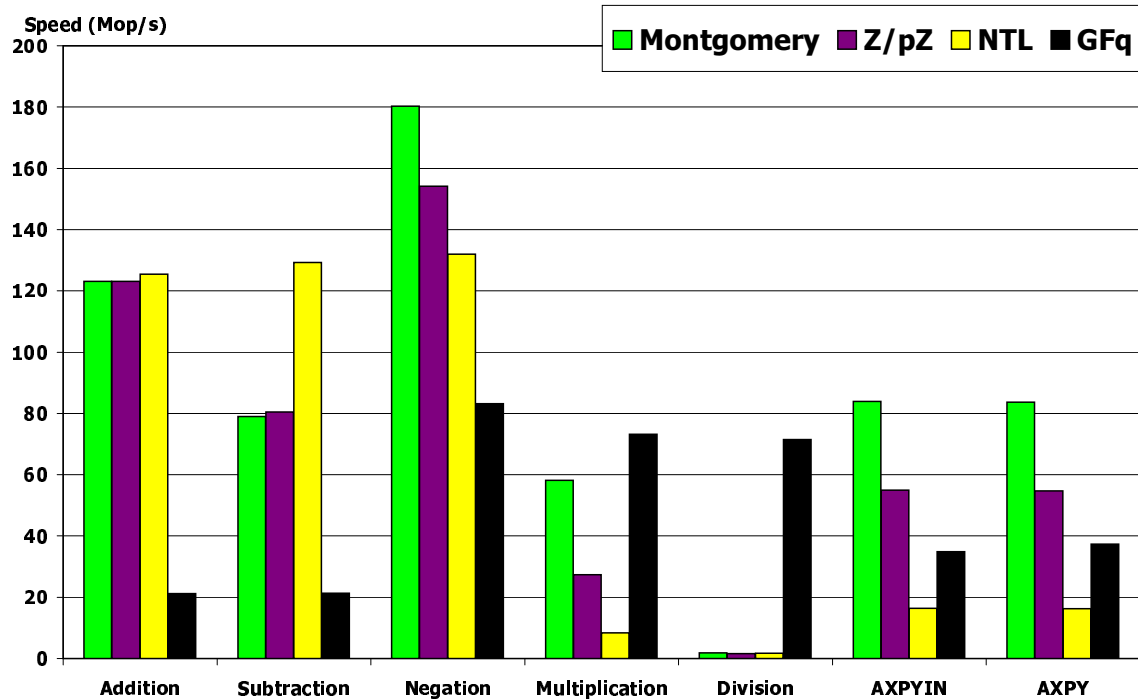


Fig. 3. Single arithmetic operation modulo 32749 on a Pentium III, 1 GHz, cache 256 Kb

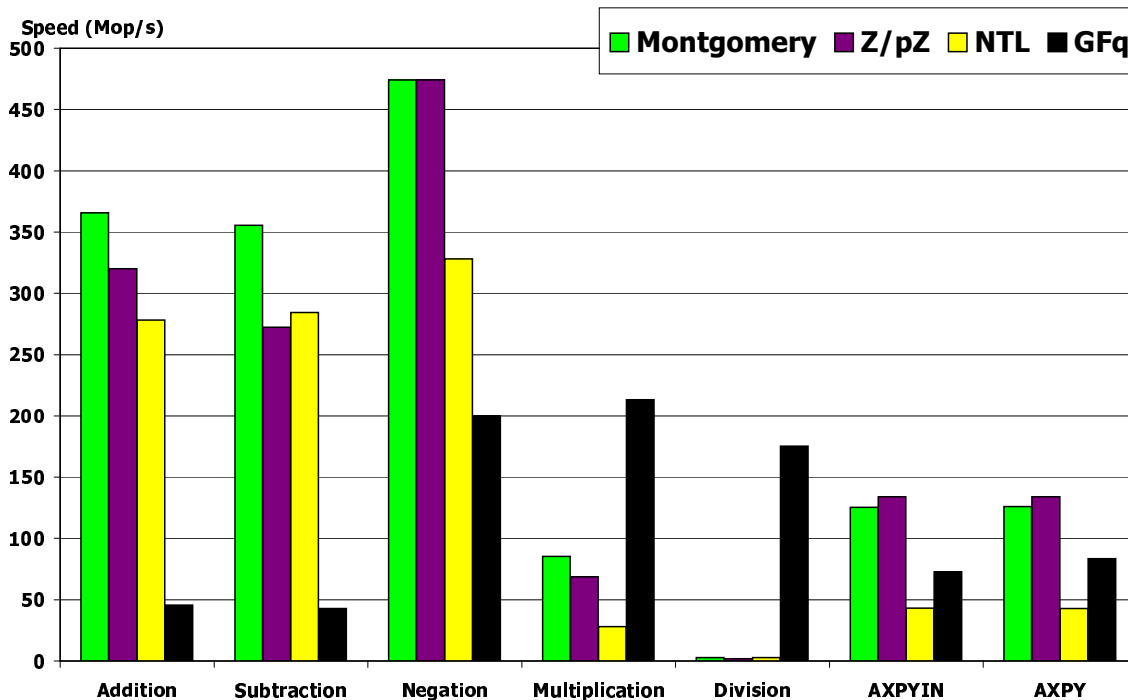


Fig. 4. Single arithmetic operation modulo 32749 on a Pentium IV, 2.4 GHz, cache 512 Kb

On newer PC, namely Intel Pentium III and IV of figures 3 and 4, the compiler used for the C/C++ programs was “gcc version 3.2.3 20030309 (Debian prerelease)”. One can see that now memory accesses have become much too slow. Then any tabulated implementation is penalized, except for extremely small modulus. NTL’s implementation is also penalized, both because of better integer operations and because of a pretty bad flooring (casting to integer) of the floating point representation. Now, for Montgomery reduction, this trick is very efficient for the multiplication. However; it seems that it becomes less useful as the machine division improves as shown by the AXPY results of figure 4. As shown section 2.2 the Montgomery AXPY is less impressive because one has to compute first the multiplication, one reduction and then only the addition and tests. This is due to our choice of representation aB . “Zpz”, not suffering from this distinction between multiplied and added values can perform the multiplication and addition before the reduction so that one test and sometimes a correction by p are saved. Nevertheless, we will see in next section that our choice of representation is not anymore a disadvantage for the dot product.

3 Dot products

In this section, we extend the results of [6, §3.1]. Two main techniques are used: regrouping operations before performing the remaindering, and performing this remaindering only on demand. Several new variants of the representations of section 2 are tested and compared. For “GFq” and “Montgomery” representations the dot products are of course performed with their representations. In particular the timings presented do not include conversions. The argument is that the dot product is computed to be used within another higher level computation. In this paradigm, conversions will only be useful for reading the values in the beginning and for writing the results at the end of the whole program.

3.1 53 and 64 bits

The first idea is to use a representation where no overflow can occur during the course of the dot product. The division is then delayed to the end of the product: if the available mantissa is of m

bits and the modulo is p , the division happens at worst every λ multiplications where λ verifies the following condition:

$$\lambda(p-1)^2 < 2^m \quad (1)$$

For instance when using 64 bits integers with numbers modulo 40009, a dot product of vectors of size lower than $1.15 \cdot 10^{10}$ will never overflow. Hence one has just to perform the AXPY without any division. A single machine remaindering is needed at the end for the whole computation. This produces very good speed ups for 53 (double representation) and 64 bits storage as shown on curves (5) and (6) in figure 5.

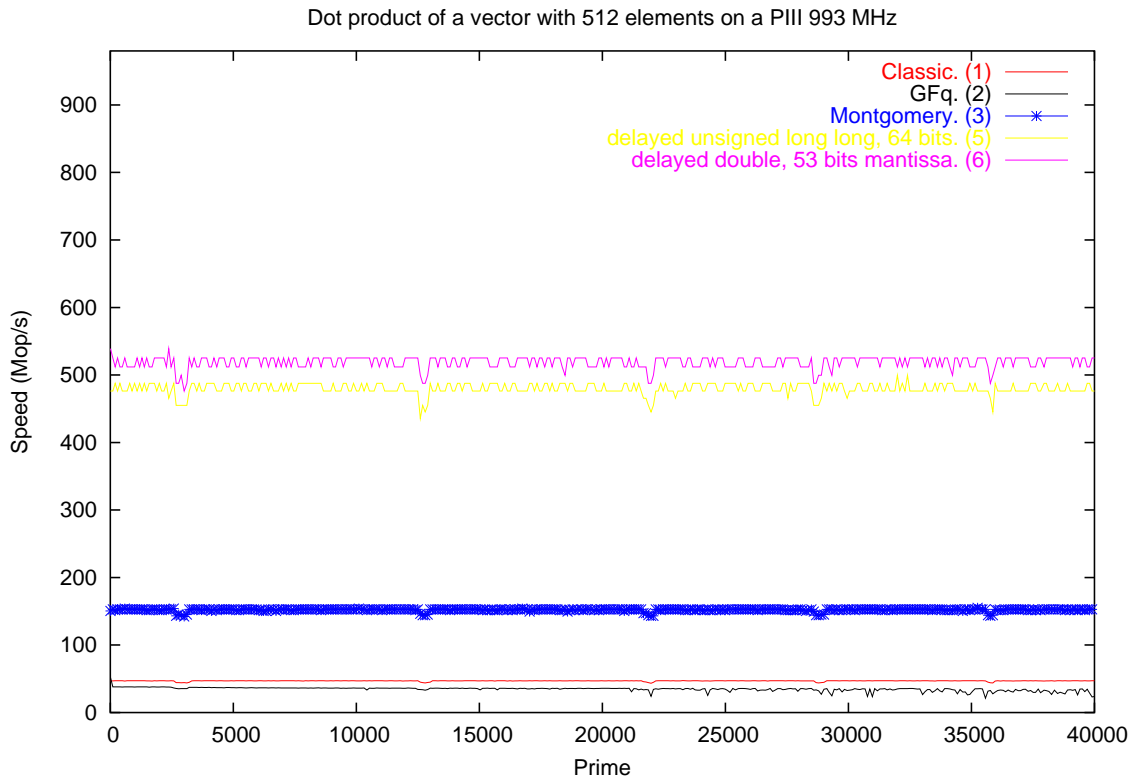


Fig. 5. Dot product by delayed division, on a PIII

There, the floating point representation performs the division “à la NTL” using a floating point precomputation of the inverse and is slightly better than the 64-bit integer representation. Note also the very good behavior of an implementation of P. Zimmermann [22] of Montgomery reduction over 32-bit integers.

3.2 AXPY blocks

The extension of this idea is to specialize dot product in order to make several multiplications and additions before performing the division (which is then delayed), even with a small storage. Indeed, one needs to perform a division only when the intermediate result is able to overflow.

Blocked dot product

```

res = 0;
unsigned long i=0; if (K<DIM) while ( i < (DIM/K)*K ) {
    for(unsigned long j = 0; j < K; ++j, ++i) res += a[i]*b[i];
    res %= P;
}
for(; i< DIM; ++i) res += a[i]*b[i];
res %= P;

```

This method will be referred as “block-XXX”. Figure 6 shows that it is optimal for small primes since it performs nearly one arithmetic operation per processor cycle. Then, the step shape of the curve reflects the thresholds when an additional division is made.

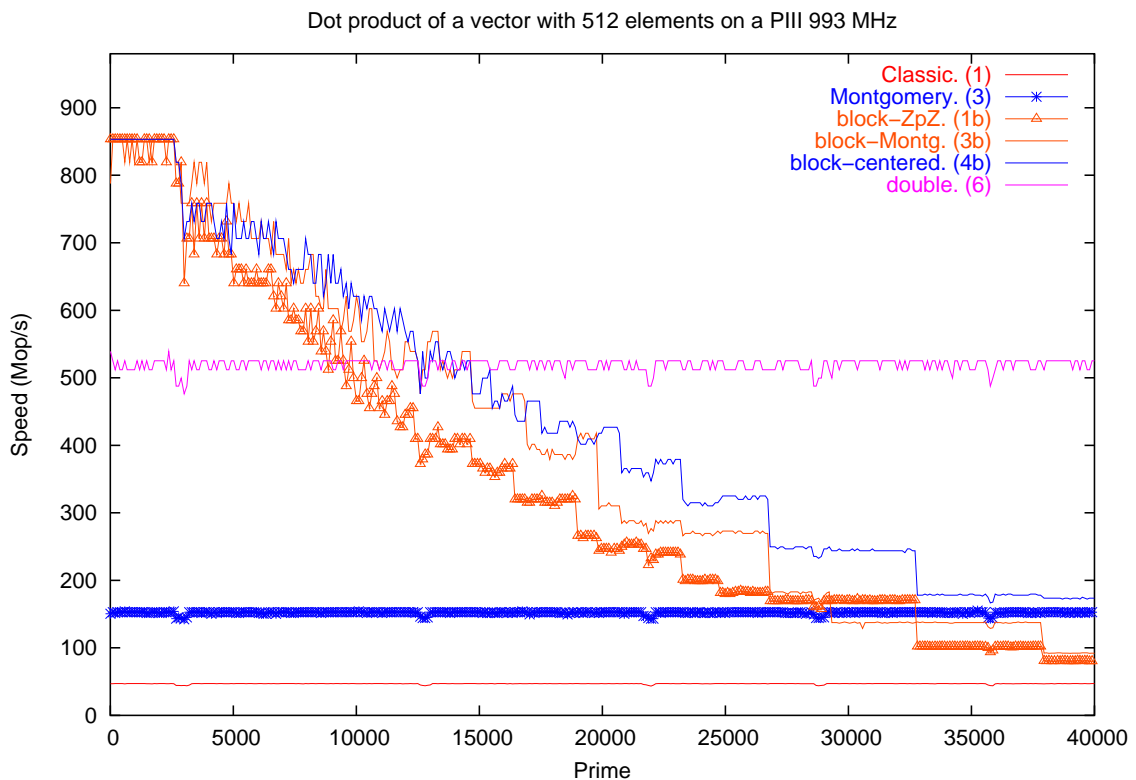


Fig. 6. Dot product by blocked and delayed division, on a PIII

Choice of Montgomery representation We see here, that our choice of representation (aB) for Montgomery is interesting. Indeed, the basic dot product operation is a cumulative AXPY. A classical AXPY would then be $axB^2 + yB$, henceforth needing an additional reduction before the addition of yB . Now, within a dot product, each one of the added values is in fact the result of a multiplication. Therefore, the additions are between elements of the form $sB^2 + x_i y_i B^2$. This proves that the reduction can indeed be delayed.

Centered representation Another idea is to use a centered representation for “ $\mathbb{Z}/p\mathbb{Z}$ ”: indeed if elements are stored between $-\frac{p-1}{2}$ and $\frac{p-1}{2}$, one can double the sizes of the blocks (equation 1 now becomes $\lambda_{centered}(\frac{p-1}{2})^2 < 2^{m-1}$).

Still and all, for small primes, any one of the blocked representation is better than a floating point representation. The slight differences between the three being the different thresholds for a single additional reduction.

3.3 Division on demand

The second idea is to let the overflow occur ! Then one should detect this overflow and correct the result if needed. Indeed, suppose that we have added a product ab to the accumulated result t and that an overflow has occurred. The variable t now contains actually $t - 2^m$.

Well, the idea is just to precompute a correction $CORR = 2^m \bmod p$ and add this correction whenever an overflow has occurred.

Now for the overflow detection, we use the following trick: since $0 < ab < 2^m$, an overflow has occurred if and only if $t + ab < t$. The “Z/pZ” code now should look like the following:

Unsigned Overflow detection trick

```
sum = 0;
for(unsigned long i = 0; i<DIM; ++i) {
    product = a[i]*b[i];
    sum += product; if (sum < product) sum += CORR;
}
```

Of course one can also apply this trick to Montgomery reduction. Indeed, as shown on curve (3c) in figure 7, the trick of using a representation storing $aB \bmod p$ for any element a enables to perform only one reduction at the end of each block. We see also, that as soon as one reduction is needed,

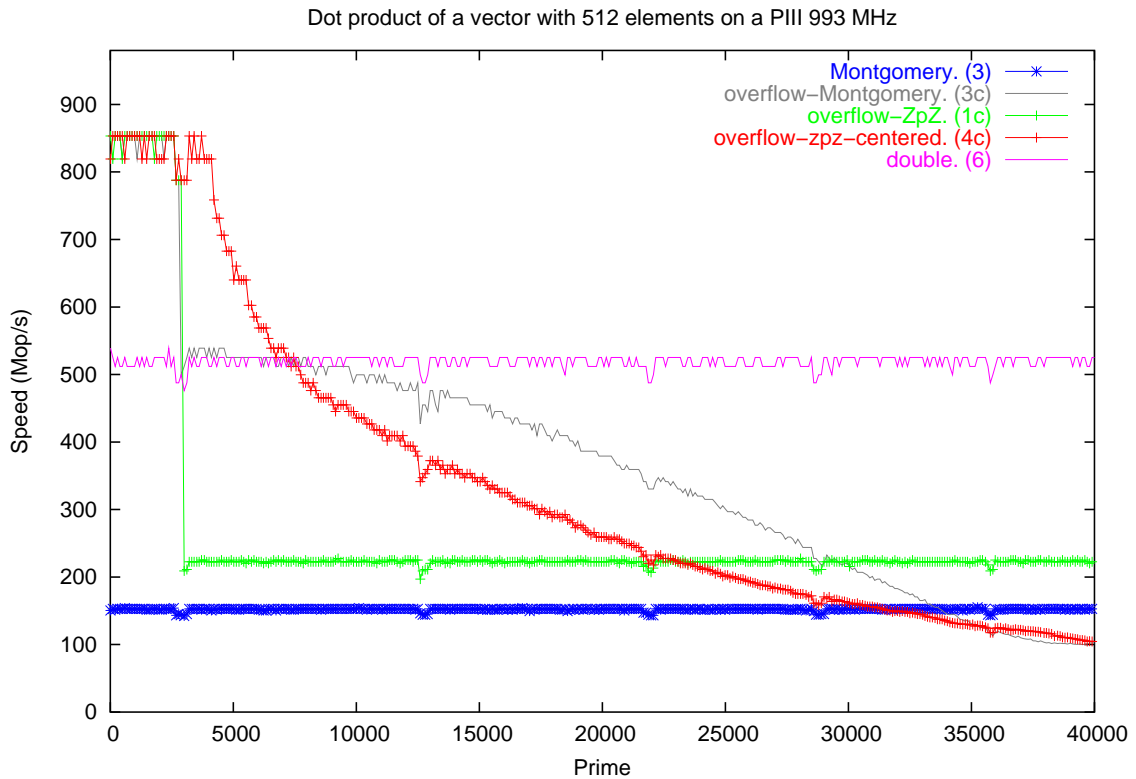


Fig. 7. Overflow detection, on a PIII

the drop of performances is tremendous. The pipeline is completely broken and “overflow-ZpZ” as well as “overflow-Montgomery” are rapidly outperformed by the floating point representation. The centered representation can be used also in this case. This gives the better performances for small primes and enables a slower drop of performances. This is due to its higher threshold. However, for this representation, the unsigned trick does not apply directly anymore. The overflow and underflow need to be detected each by two tests:

Signed overflow detection trick

```

if      ((sum < product) && (product - sum < 0)) sum += CORR;
else if ((product < sum) && (sum - product < 0)) sum -= CORR;

```

Thus, the total of four tests is costlier and for bigger primes this overhead is too expensive as shown on curve (4c) of figure 7.

3.4 Hybrid

Of course, one can mix both 3.2 and 3.3 approaches and delay even the overflow test when p is small. One has just to slightly change the bound on λ so that, when adding the correction, no new overflow occur:

$$\lambda(p-1)^2 + (p-1) = \lambda p(p-1) < 2^m. \quad (2)$$

This method will be referred as “block-overflow-XXX”.

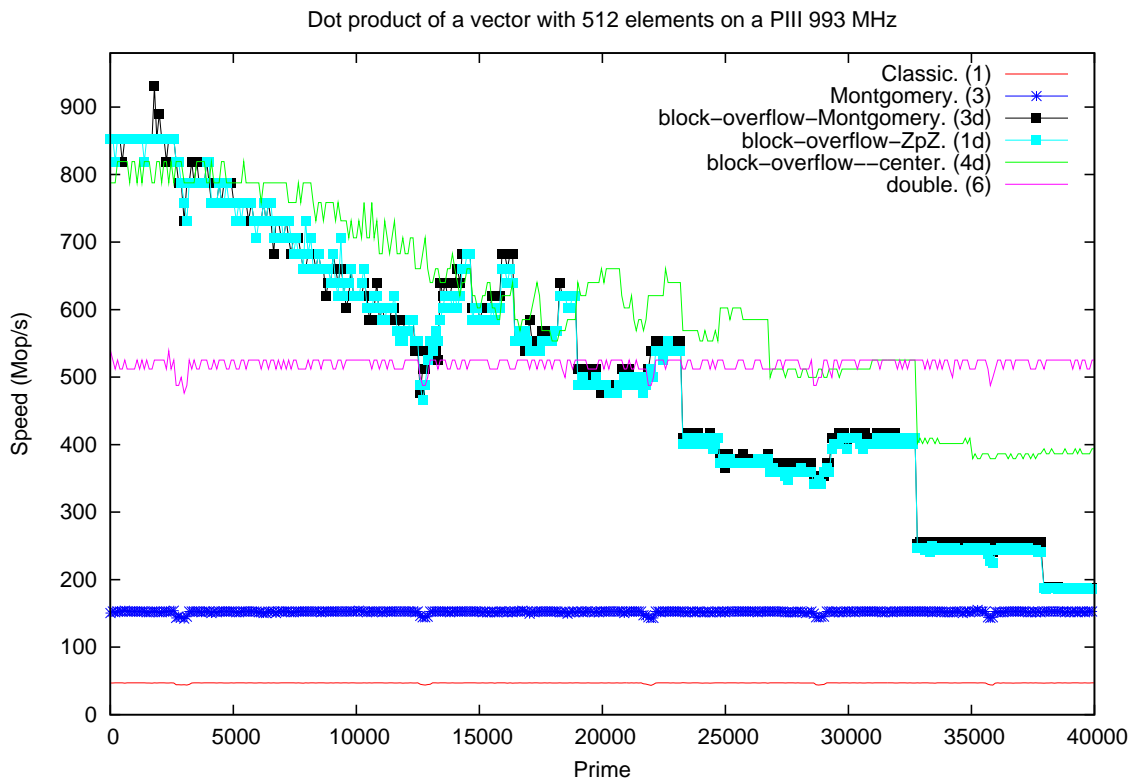


Fig. 8. Hybrid (block AND overflow detection), on a PIII

We compared those ideas on a vector of size 512 with 32 bits (unsigned long on PIII). First, we see that as long as $512p(p-1) < 2^m$, we obtain *quasi optimal performances*, since only one division

is performed at the end. Then, when the prime exceeds the bound (i.e. for $p(p-1) > 2^{32-9}$, which is $p > 2897$) an extra division has to be made. On the one hand, this is dramatically shown by the drops of figure 7.

On the other hand, however, those drops are levelled by our hybrid approach as shown in figure 8. There the step form of the curve shows that every time a supplementary division is added, performances drop accordingly. Now, as the prime size augments, the block management overhead becomes too important.

One can remark than no significant difference exist between the performances of “block-overflow-Zpz” and “block-overflow-Montgomery”. Indeed, the code is now exactly the same except for a single reduction for the whole dot product. This makes the Montgomery version better but only extremely slightly.

Lastly, here also, an hybrid *centered* version is useful. When compared to Montgomery or Zpz, one can remark that the signed overflow detection is two times as costly as the unsigned overflow detection. However, the block size is twice as big (one loses a bit for the sign but gains 2 bits since the multiplied numbers are then both of absolute value less than $\frac{p-1}{2}$).

3.5 Vector size influence

In this last section, we discuss the vector size influence. Until now we have used vectors of size 512. We see on figure 9 that the best vector size is indeed around this size. Those three figures shows at least two things:

- On the one hand, the floating point representation is the most stable one for half word size primes. Then, vector size do not really infer with its performances. On the other hand, those performances are interesting only for big vector sizes.
- Except when the prime is too big the hybrid centered representation is nearly always the best one.

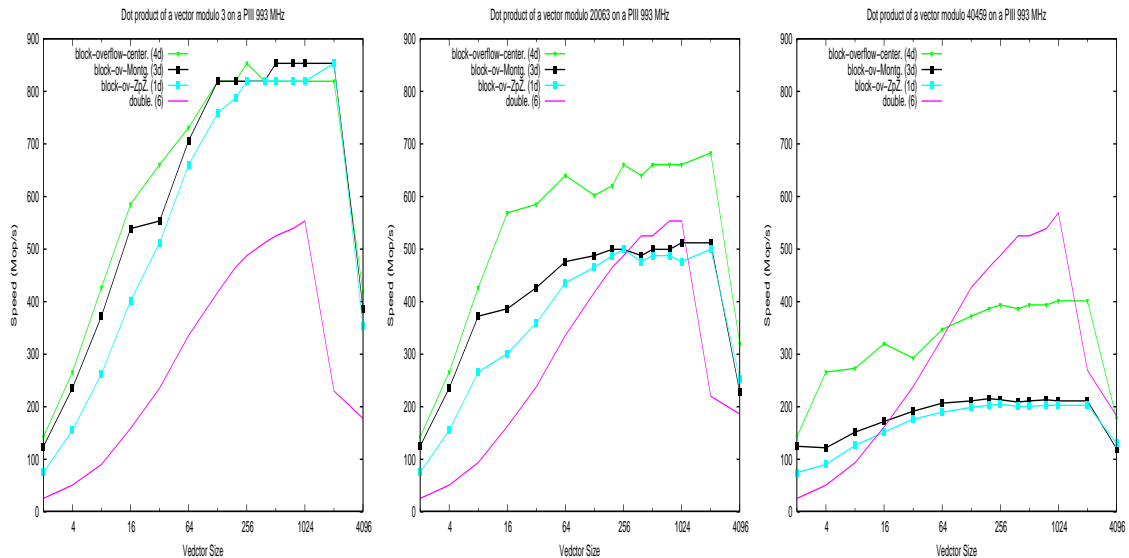


Fig. 9. Hybrid (function of the vector size) modulo 3, 20063 and 40459 on a PIII 993 MHz

4 Conclusion

We have seen different ways to implement a dot product over word size finite fields. The conclusion is that most of the times, a floating point representation is the best implementation. This is even more the case for a Pentium IV 2.4 GHz, as shown figures 10 and 11.

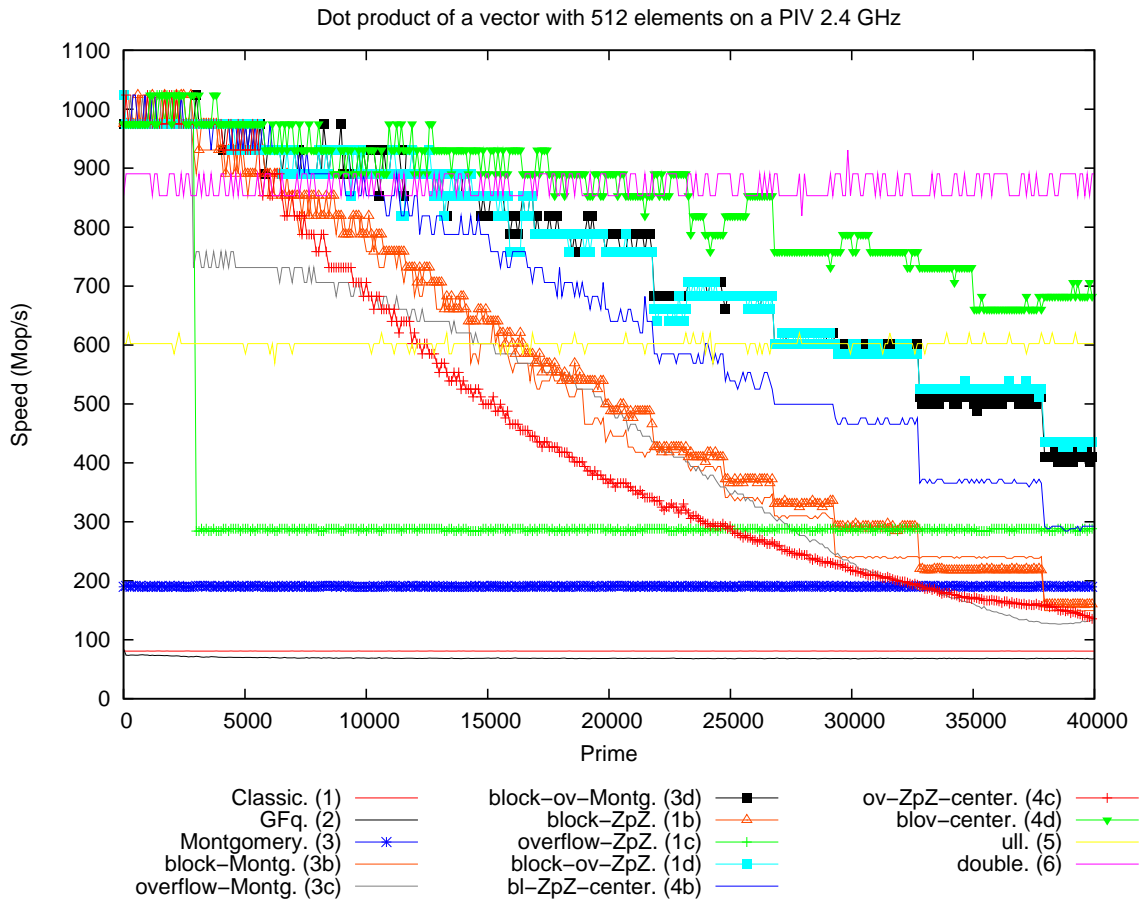


Fig. 10. Hybrid, on a PIV 2.4 GHz

However, with some care, it is possible to improve this speed for small primes by a hybrid method using overflow detection and delayed division. Now, the floating point representation approximately doubles its performances on the PIV 2.4 GHz, when compared to the 1 GHz Pentium III. But surprisingly, the hybrid versions only slightly improve. Still and all, an optimal version should switch from block methods to a floating point representation according to the vector and prime size and to the architecture.

Nevertheless, bases for the construction of an optimal dot product over word size finite fields have been presented: the idea is to use an Automated Empirical Optimization of Software [19] in order to produce a library which would determine and choose the best switching thresholds at install time.

Acknowledgements

Grateful thanks to Paul Zimmermann for invaluable advice and comments and several implementations.

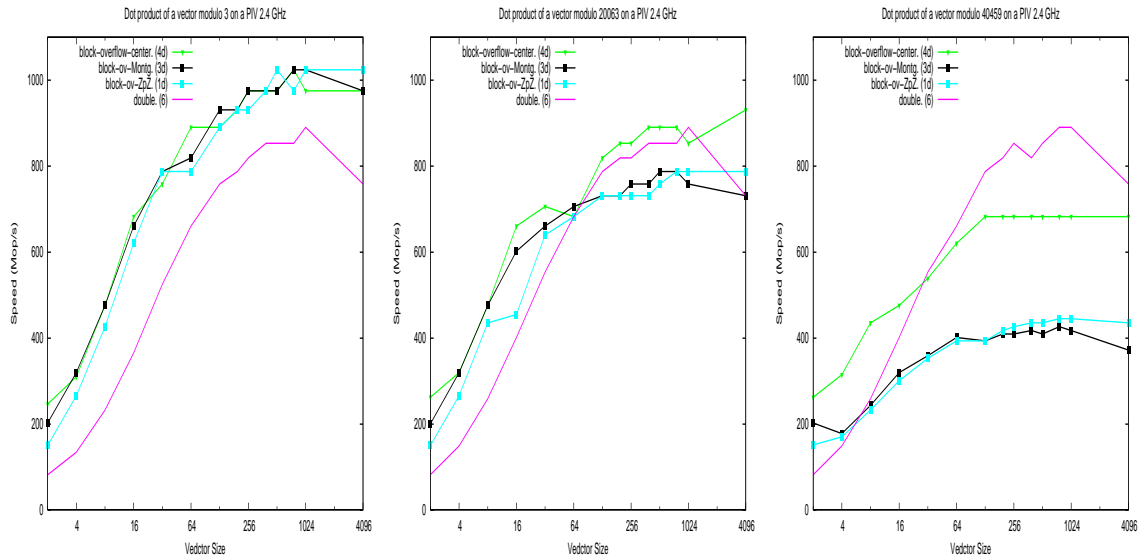


Fig. 11. Hybrid (function of the vector size) modulo 3, 20063 and 40459 on a PIV 2.4 GHz

References

1. D. V. Bailey and C. Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.
2. D. Defour. *Fonctions élémentaires : algorithmes et implémentations efficaces pour l'arrondi correct en double précision*. PhD thesis, École Normale Supérieure de Lyon, Sept. 2003.
3. J. D. Dixon. Exact solution of linear equations using p-adic expansions. *Numerische Mathematik*, 40:137–141, 1982.
4. P. Douillet. Zech logarithms and finite fields. Technical report, Faculté des Sciences, Paris, Feb. 2001.
5. J.-G. Dumas. *Algorithmes parallèles efficaces pour le calcul formel : algèbre linéaire creuse et extensions algébriques*. PhD thesis, Institut National Polytechnique de Grenoble, France, Dec. 2000. <ftp://ftp.imag.fr/pub/Mediatheque.IMAG/theses/2000/Dumas.Jean-Guillaume>.
6. J.-G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In T. Mora, editor, *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, Lille, France*, pages 63–74. ACM Press, New York, July 2002.
7. J.-G. Dumas, P. Giorgi, and C. Pernet. FFPACK: Finite field linear algebra package. In J. Gutierrez, editor, *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, Santander, Spain*. ACM Press, New York, July 2004.
8. J.-G. Dumas, B. D. Saunders, and G. Villard. On efficient sparse integer matrix Smith normal form computations. *Journal of Symbolic Computations*, 32(1/2):71–99, July–Aug. 2001.
9. J.-G. Dumas and G. Villard. Computing the rank of sparse matrices over finite fields. In V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, editors, *Proceedings of the fifth International Workshop on Computer Algebra in Scientific Computing, Yalta, Ukraine*, pages 47–62. Technische Universität München, Germany, Sept. 2002.
10. T. Hansen and G. L. Mullen. Primitive polynomials over finite fields. *Mathematics of Computation*, 59(200):639–643, S47–S50, Oct. 1992.
11. K. Huber. Some comments on Zech's logarithm. *IEEE Transactions on Information Theory*, IT-36:946–950, July 1990.
12. K. Huber. Solving equations in finite fields and some results concerning the structure of $\text{GF}(p^m)$. *IEEE Transactions on Information Theory*, IT-38:1154–1162, May 1992.
13. E. Kaltofen and G. Villard. On the complexity of computing determinants. In *Proceedings of the Fifth Asian Symposium on Computer Mathematics ASCM 2001*, volume 9 of *Lecture Notes Series on Computing*, pages 13–27, Singapore, Sept. 2001.
14. B. A. LaMacchia and A. M. Odlyzko. Solving large sparse linear systems over finite fields. *Lecture Notes in Computer Science*, 537:109–133, 1991. <http://www.research.att.com/~amo/doc/arch/sparse.linear.eq.s.ps>.

15. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
16. P. L. Montgomery. A block Lanczos algorithm for finding dependencies over $\text{GF}[2]$. In L. C. Guillou and J.-J. Quisquater, editors, *Proceedings of the 1995 International Conference on the Theory and Application of Cryptographic Techniques, Saint-Malo, France*, volume 921 of *Lecture Notes in Computer Science*, pages 106–120, May 1995.
17. A. M. Odlyzko. Discrete logarithms: The past and the future. *Designs, Codes, and Cryptography*, 19:129–145, 2000.
18. V. Shoup. NTL 5.3: A library for doing number theory, 2002. www.shoup.net/ntl.
19. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, Jan. 2001. www.elsevier.nl/gej-ng/10/35/21/-47/25/23/article.pdf.
20. D. H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, Jan. 1986.
21. H. Zassenhaus. A remark on the Hensel factorization method. *Mathematics of Computation*, 32(141):287–292, Jan. 1978.
22. P. Zimmermann, 2002. Personal communication.