# Fast vector arithmetic over $\mathbb{F}_3$

K. Coolsaet

**Abstract**

We show how binary machine instructions can be used to implement fast vector operations over the finite field $\mathbb{F}_3$. Apart from the standard operations of addition, subtraction and dot product, we also consider combined addition and subtraction, weight, Hamming distance, and iteration over all vectors of a given length.

Tests show that our implementation can be as much as 10 times faster than the standard method of using modular arithmetic on arrays of bytes. For computing the Hamming distance even a factor of 33 can sometimes be reached, provided a recent CPU is used.

## 1 Introduction

It is common knowledge that the fastest way to work with vectors over the field $\mathbb{F}_2$ of two elements is to represent such vectors as bit vectors and use binary CPU instructions to process them, in particular 'exlusive or' for addition, and 'and' for multiplication. The inherent bit parallellism of a 64-bit CPU allows vectors of $\mathbb{F}_2^{64}$ to be processed as fast as single integers.

It is less well known that a similar technique can be used to parallellize vector operations over the field $\mathbb{F}_3$ of three elements, representing every element of the field as a pair of bits and emulating the field operations of $\mathbb{F}_3$ as combinations of standard binary machine instructions.

In [3] Kawahara et al. use such a representation to perform fast vector addition and subtraction needing six binary operations for each ternary operation. They also show that fewer operations do not suffice.

Received by the editors June 2012.
Communicated by J. Doyen.
2000 *Mathematics Subject Classification :* 65Y04, 12E30, 12-04.
*Key words and phrases :* Fast vector arithmetic, GF(3), 64-bit operations, Hamming distance, dot product.

In this paper we shall additionally present fast binary methods for computing the dot product of two vectors and determining the Hamming distance between two vectors. We also show that if we need both the sum and the difference of the same pair of vectors, then this can be done considerably faster than adding and subtracting them separately. (This is for instance useful when generating all linear combinations of a given set of vectors.) We also describe a fast method for iterating through all vectors of a given length. All of these techniques should prove very effective for computer applications in ternary codes and finite geometries.

We implemented several benchmark programs to estimate how fast our operations are in comparison to the standard way of representing vectors as arrays of bytes with modular arithmetic to add, subtract or multiply them. We ran these tests on various types of CPU and also compared several variants of binary representations. One slightly unexpected result was that an implementation of addition and subtraction using seven operations ran at exactly the same speed as the implementation of Kawahara et al., which uses only six. The number of steps in which operations can be performed in parallel seems to be a better predictor for the actual execution time than simply the number of operations.

In summary, for vectors of length 64, addition, subtraction and dot product can be done up to 10 times faster using our methods instead of the standard implementations. For computing the Hamming distance we even managed to improve the running time by a factor of 33 (provided the CPU is sufficiently recent).

In Section 2 we present and prove the formulas that form the basis of our implementation (cf. Theorem 1). In Section 3.1 we describe how vector addition, subtraction and a combination of those can be done by representing a vector of $\mathbb{F}_3^n$, $n \leq 64$, as two 64-bit words. Weight and Hamming distance are treated in Section 3.2, the dot product in Section 3.3 and iteration over all elements of $\mathbb{F}_3^n$ in Section 3.4. In Section 3.5 we describe an alternative representation which uses three 64-bit words for each vector, and one which needs only a single 64-bit word, provided $n \leq 32$. The results of our benchmarks are presented in Section 4 and in Section 5 we end with some final remarks.

## 2 Basic operations

Although in this text we will be working at the same time with elements of both $\mathbb{F}_2$ and $\mathbb{F}_3$, it will always be clear from context which is which and therefore we use the same standard mathematical notations for addition, subtraction and multiplication, irrespective of the field in which we are working.

Recall that binary addition is the same as binary 'exclusive or' and multiplication is the same as binary 'and'. We shall use the notation '$|$' for binary 'or' and '$\neg$' for binary 'not'. In terms of field operations we have $x \mid y = xy + x + y$ and $\neg x = x + 1$. The following properties are easily derived :

$$x \mid y = x \mid (x+y) = y \mid (x+y), \qquad x \mid (y+z) = (x \mid y) + (x \mid z) + x, \qquad (1)$$

for all $x, y, z \in \mathbb{F}_2$.

For $d \in \mathbb{F}_3$ we define $d_0, d_1, d_2 \in \mathbb{F}_2$ according to the following table :

| $d$ | $d_0$ | $d_1$ | $d_2$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 2 | 1 | 1 | 0 |

(2)

Note that $d_0 + d_1 + d_2 = 0$ and that in all cases $d_0 \mid d_1 = d_1 \mid d_2 = d_2 \mid d_0 = 1$, or equivalently $d_2 = d_0 + d_1 = d_0 d_1 + 1$, $d_0 = d_1 + d_2 = d_1 d_2 + 1$, $d_1 = d_2 + d_0 = d_2 d_0 + 1$. The permutation $d \mapsto d + 1$ in $\mathbb{F}_3$ translates to $d_0 \mapsto d_1 \mapsto d_2 \mapsto d_0$ in $\mathbb{F}_2$. The involution $d \leftrightarrow -d$ corresponds to the interchange $d_1 \leftrightarrow d_2$.

In Section 3 we shall discuss several representations of (vectors of) ternary digits. In most of the cases we shall represent one digit $d$ by a pair of bits $(d_1, d_2)$, although also a 3-bit representation $(d_0, d_1, d_2)$ shall be considered.

The following theorem serves as the basis for the rest of this paper.

**Theorem 1.** *Let* $v, w \in \mathbb{F}_3$. *Then*

$$n = -v \quad \Leftrightarrow \quad \begin{cases} n_1 &= v_2 \\ n_2 &= v_1 \end{cases} \tag{3}$$

$$s = v + w \quad \Leftrightarrow \quad \begin{cases} s_1 &= (v_0 + w_1) \mid (v_1 + w_0) = (v_0 + w_1) \mid (v_2 + w_2) \\ &= (v_2 + w_2) \mid ((v_1 + w_1) + v_2) \\ &= ((v_1 + w_1) \mid (v_2 + w_2)) + v_2 w_2 \\ s_2 &= (v_0 + w_2) \mid (v_2 + w_0) = (v_0 + w_2) \mid (v_1 + w_1) \\ &= (v_1 + w_1) \mid ((v_2 + w_2) + v_1) \\ &= ((v_1 + w_1) \mid (v_2 + w_2)) + v_1 w_1 \end{cases} \tag{4}$$

$$d = v - w \quad \Leftrightarrow \quad \begin{cases} d_1 &= (v_0 + w_2) \mid (v_1 + w_0) = (v_0 + w_2) \mid (v_2 + w_1) \\ &= (v_2 + w_1) \mid ((v_1 + w_2) + v_2) \\ &= ((v_1 + w_2) \mid (v_2 + w_1)) + v_2 w_1 \\ d_2 &= (v_0 + w_1) \mid (v_2 + w_0) = (v_0 + w_1) \mid (v_1 + w_2) \\ &= (v_1 + w_2) \mid ((v_2 + w_1) + v_1) \\ &= ((v_1 + w_2) \mid (v_2 + w_1)) + v_1 w_2 \end{cases} \tag{5}$$

$$p = vw \quad \Leftrightarrow \quad \begin{cases} p_1 &= v_1 w_2 \mid v_2 w_1 = (v_1 \mid w_1)(v_2 \mid w_2) \\ p_2 &= v_1 w_1 \mid v_2 w_2 = (v_1 \mid w_2)(v_2 \mid w_1) \\ p_0 &= v_0 w_0 \\ \neg p_2 &= v_0 w_0 (v_1 + w_1) = v_0 w_0 (v_2 + w_2) \\ &= (v_1 + w_1)(v_2 + w_2) = (v_1 \mid w_2) + (v_2 \mid w_1) \end{cases} \tag{6}$$

*Proof.* The result for the negation $n$ is a trivial consequence of the definition (2). The results for the difference $d$ follow from those of the sum $s$ by substituting $-w$ for $w$ (i.e., interchanging $w_1 \leftrightarrow w_2$).

Now consider $s_1$. Using the first of the identities (1) twice, we find

$$\begin{aligned} (v_0 + w_1) \mid (v_1 + w_0) &= (v_0 + w_1) \mid (v_1 + w_0 + v_0 + w_1) \\ &= (v_0 + w_1) \mid (v_2 + w_2) \\ &= (v_0 + w_1 + v_2 + w_2) \mid (v_2 + w_2) \\ &= (v_1 + w_1 + w_2) \mid (v_2 + w_2). \end{aligned}$$

Again by (1) we obtain

$$\begin{aligned}((v_1+w_1)+w_2) \mid (v_2+w_2) &= ((v_1+w_1) \mid (v_2+w_2)) + (w_2 \mid (v_2+w_2)) \\ &\qquad +(v_2+w_2) \\ &= ((v_1+w_1) \mid (v_2+w_2)) + (v_2 \mid w_2) + (v_2+w_2) \\ &= ((v_1+w_1) \mid (v_2+w_2)) + v_2w_2.\end{aligned}$$

This proves that the three formulas for $s_1$ in (4) are indeed equivalent. The same holds for $s_2$, which can be obtained from $s_1$ by interchanging $v_1 \leftrightarrow v_2$ and $w_1 \leftrightarrow w_2$.

Note that the formulas for $s_1$ and $s_2$ are invariant under the transformations $v \mapsto v+1$, $w \mapsto w-1$, i.e., $v_0 \mapsto v_1 \mapsto v_2 \mapsto v_0$, $w_0 \mapsto w_2 \mapsto w_1 \mapsto w_0$. It is therefore sufficient to prove the validity of these formulas in the special case $v = w$. And indeed, in that case the first formula in (4) reduces to $s_1 = v_0 + v_1 = v_2$ and similarly $s_2 = v_0 + v_2 = v_1$. Therefore $s = -v = 2v$, as expected.

This leaves (6). We have $vw = 0$ if and only if $v = 0$ or $w = 0$ if and only if $v_0 = 0$ and $w_0 = 0$ if and only if $v_0w_0 = 0$. This proves the value for $p_0$. Similarly, $vw = 1$ if and only if $v = w = 1$ or $v = w = 2$, i.e., if and only if $v_1 = w_1 = 0$ or $v_2 = w_2 = 0$. And this is equivalent to $(\neg v_1)(\neg w_1) \mid (\neg v_2)(\neg w_2) = 1$, or by De Morgan's laws, $(v_1 \mid w_1)(v_2 \mid w_1) = 0$, yielding the second formula for $p_1$. By distributivity of 'or' over 'and' we also find

$$v_1w_2 \mid v_2w_1 = (v_1 \mid v_2)(v_1 \mid w_1)(w_2 \mid v_2)(w_2 \mid w_1) = (v_1 \mid w_1)(v_2 \mid w_2),$$

which proves the equivalence of the first and second expression for $p_1$. The expressions for $p_2$ can be proved in a similar way.

Again by De Morgan's laws, we find

$$\neg p_2 = \neg((v_1w_1) \mid (v_2w_2)) = (v_1w_1+1)(v_2w_2+1) = (v_1+w_1)(v_2+w_2).$$

Also

$$\begin{aligned}(v_1 \mid w_2) + (v_2 \mid w_1) &= v_1w_2 + v_1 + w_2 + v_2w_1 + v_2 + w_1 \\ &= v_1w_2 + v_2w_1 + (v_1+v_2) + (w_1+w_2) \\ &= v_1w_2 + v_2w_1 + v_1v_2 + w_1w_2 = (v_1+w_1)(v_2+w_2).\end{aligned}$$

Finally

$$\begin{aligned}v_0w_0(v_2+w_2) &= (v_0v_2)w_0 + (w_0w_2)v_0 = (v_1+1)w_0 + (w_1+1)v_0 \\ &= (v_1+1)(w_1+w_2) + (w_1+1)(v_1+v_2) \\ &= v_1w_2 + w_1 + w_2 + w_1v_2 + v_1 + v_2,\end{aligned}$$

which is $(v_1+w_1)(v_2+w_2)$ as before. ∎

(An alternative proof of this theorem consists of trying all 9 possible combinations of $v, w$ and checking each result. This can be automated and provides a good test for any implementation of these operations.)

## 3 Vector arithmetic

Consider an $n$-tuple $V = (v^{(1)}, \ldots, v^{(n)}) \in \mathbb{F}_3^n$, i.e., a vector of length $n$ with elements in $\mathbb{F}_3$. We shall write $V_i$ for the bit vector $V_i \stackrel{\text{def}}{=} (v_i^{(1)}, \ldots, v_i^{(n)}) \in \mathbb{F}_2^n$. If $V, W$ are vectors of the same length, then we write $V + W$, $V - W$, $VW$ for the elementwise sum, difference and product of $V$ and $W$ (both in $\mathbb{F}_2^n$ and $\mathbb{F}_3^n$). We also write $V_i \mid W_j$ for the elementwise binary 'or' of two vectors $V_i, W_j \in \mathbb{F}_2^n$. The elementwise multiplication $VW$ should not be confused with the dot product $V \cdot W \stackrel{\text{def}}{=} V^{(1)}W^{(1)} + \cdots + V^{(n)}W^{(n)} \in \mathbb{F}_3$, equal to the sum of the elements of the vector $VW$. We denote the weight of $V$ by $\|V\|$, i.e., the number of elements of $V$ that are different from zero. The Hamming distance $\|V - W\|$ counts the number of positions $i$ for which $V^{(i)}$ and $W^{(i)}$ differ.

We have used Theorem 1 in three different implementations of vector arithmetic over $\mathbb{F}_3$ which we discuss below (including some additional variants). We present fast methods for computing $V + W$, $V - W$, $V \cdot W$, $\|V\|$ and $\|V - W\|$ largely in terms of the binary vector operations $V_iW_j$, $V_i + W_j$ and $V_i \mid W_j$ which, when $n \leq 64$, correspond to single CPU instructions on standard 64-bit microprocessors. For the dot product, weight and the Hamming distance we also use 64-bit remainder, multiplication and shifts and a special 'population count' instruction, if available.

In what follows we will always assume that $n \leq 64$. Vectors of larger dimension can still be handled by partitioning them into blocks of size $\leq 64$.

### 3.1 Two words for each vector

The most direct way to apply Theorem 1 is to store a vector $V$ in two separate machine words that correspond directly to $V_1$ and $V_2$. Negation, addition, subtraction and elementwise multiplication can then be implemented as straight translations of the formulas of the theorem.

For addition (and similarly, subtraction) we may use the second and fifth line of (4) to obtain an implementation in as few as six operations :

$$\begin{array}{lcllcl} T_1 & \leftarrow & V_1 + W_1 & T_2 & \leftarrow & V_2 + W_2 \\ U_1 & \leftarrow & T_1 + V_2 & U_2 & \leftarrow & T_2 + V_1 \\ S_1 & \leftarrow & T_2 \mid U_1 & S_2 & \leftarrow & T_1 \mid U_2 \end{array} \tag{7}$$

($S$ contains the result, $T$ and $U$ are auxiliary variables.)

This is the same implementation discussed in [3], where it is also proved that at least six operations are needed. However, on modern CPUs the number of operations is not always the best measure of speed. We have also tested the following implementation, based on the third and sixth line of (4), which needs seven operations :

$$\begin{array}{lcllcllcllcl} T_1 & \leftarrow & V_1 + W_1 & T_2 & \leftarrow & V_2 + W_2 & U_1 & \leftarrow & V_1W_1 & U_2 & \leftarrow & V_2V_2 \\ U_* & \leftarrow & T_1 \mid T_2 & & & & & & & & & \\ S_1 & \leftarrow & U_* + U_2 & S_2 & \leftarrow & U_* + U_1 & & & & & & \end{array} \tag{8}$$

It turns out that our benchmark tests (cf. Section 4) show no significant difference in speed on modern CPUs between the 6- and 7-op versions of addition and subtraction. (For the somewhat less recent AMD Opteron 2212, the 7-op version is slower by a factor of $\approx 1.05$.) The reason for this is probably that the CPU manages to execute several operations in parallel, and needs only three parallel steps in both cases.

Where both $V + W$ and $V - W$ are needed at the same time, it is possible to shave off another 2 operations. There are various ways to accomplish this. For example, based on the first and fourth lines of (4–5), we may write

$$\begin{array}{lcllcllcllcl} V_0 & \leftarrow & V_1 + V_2 & W_0 & \leftarrow & W_1 + W_2 & & & & & & \\ T_1 & \leftarrow & V_0 + W_1 & T_2 & \leftarrow & V_0 + W_2 & U_1 & \leftarrow & W_0 + V_1 & U_2 & \leftarrow & W_0 + V_2 \\ S_1 & \leftarrow & T_1 \mid U_1 & S_2 & \leftarrow & T_2 \mid U_2 & D_1 & \leftarrow & T_2 \mid U_1 & D_2 & \leftarrow & T_1 \mid U_2 \end{array} \tag{9}$$

yielding 10 (instead of 12) operations in 3 parallel steps. Because we use the same number of parallel steps, one might even expect that (9) and (7) take the same execution time, and indeed on recent CPUs this is almost the case, cf. Section 4.

### 3.2 Weight and Hamming distance

**Theorem 2.** *Let $v, w \in \mathbb{F}_3$. Then*

$$\begin{aligned} v \neq w &\iff (v_1 + w_1) \mid (v_2 + w_2) = 1, \\ v \neq 0 &\iff v_1 + v_2 = 1. \end{aligned}$$

*Proof.* We have $v_1 + w_1 = 1$ if and only if $v_1 \neq w_1$, and $v_2 + w_2 = 1$ if and only if $v_2 \neq w_2$. Because $v \neq w$ if and only if $v_1 \neq w_1$ or $v_2 \neq w_2$, the first part of the theorem follows. The second part follows from the fact that $v$ is zero precisely when $v_0$ is zero. ∎

This theorem can be used to compute both the Hamming distance $\|V - W\|$ between two vectors, or the weight $\|V\|$ of a vector. These problems are now reduced to determining the weight of the binary vectors $(V_1 + W_1) \mid (V_2 + W_2)$ and $V_1 + V_2$.

For older computers there are lots of well-known tricks to compute the weight of a binary vector $\|\beta\|$. The following method (taylored to 64 bits) seems to be the fastest at this time of writing [1]. We give a version in the programming language C :

```
t = beta - ((beta >> 1) & 0x5555555555555555L);
t = (t & 0x3333333333333333L) + ((t >> 2) & 0x3333333333333333L);
t = ((t + (t >> 4)) & 0x0f0f0f0f0f0f0f0fL;
weight = (t * 0x0101010101010101L) >> 56;
```

(10)

Fortunately, recent CPUs (e.g., the Nehalem-based Intel Xeon processors) have a 'population count' machine instruction that computes $\|\beta\|$ directly and is much faster than (10).

### 3.3 The dot product

An implementation of the elementwise multiplication $VW$ in 6 operations can be obtained directly from (6). This operation does not occur very often in practice and the only reason it is treated here is because we can use it to compute the dot product $V \cdot W$, which is the sum of all entries of $VW$, computed modulo 3.

With $P = VW$, the dot product satisfies

$$V \cdot W = \|P_2\| - \|P_1\| \pmod 3, \tag{11}$$

for indeed, every ternary digit 0 in $P$ will now contribute $1 - 1$ to the result, every 1 contributes $1 - 0$ and every 2 contributes $0 - 1$.

If your CPU supports the 'population count' instruction, formula (11) is almost the fastest implementation of the dot product, although it is even better to implement it as $\|P_2\| + 2\|P_1\| \pmod 3$ because then you avoid taking remainders of negative numbers. We can improve this further by using the following identity instead :

$$V \cdot W = \|P_0\| + \|\neg P_2\| \pmod 3. \tag{12}$$

Note that $P_0$ and $\neg P_2$ can be computed together in only five operations.

For older processors we again have to resort to tricks. We first establish a fast method of computing $\|\beta\|$ modulo 3, faster than first computing $\|\beta\|$ using (10) and then reducing the result modulo 3. ($\beta \in \mathbb{F}_2^n$.)

Write $\beta_0, \beta_1, \ldots$ for the subsequent bits of $\beta$ considered as elements of $\mathbb{Z}$ (lowest significant bit first). We want to compute $\|\beta\|$ mod 3. Consider the positive integer $[[\beta]]$ that has $\beta$ as its binary representation, i.e.,

$$[[\beta]] \stackrel{\text{def}}{=} \beta_0 + \beta_1 2 + \beta_2 2^2 + \beta_3 2^3 + \beta_4 2^4 + \beta_5 2^5 + \cdots = \sum_{i=0}^{n-1} \beta_i 2^i$$

(with additions and multiplications in $\mathbb{Z}$). Transform $[[\beta]]$ to a new number $[[\gamma]]$ by shifting the bits one position towards the lower significant end and zeroing the bits on odd positions, i.e.,

$$\gamma = (\beta_1, 0, \beta_3, 0, \beta_5, 0, \ldots), \quad [[\gamma]] = \beta_1 + \beta_3 2^2 + \beta_5 2^4 + \cdots .$$

Subtracting $[[\gamma]]$ from $[[\beta]]$ then yields

$$[[\beta]] - [[\gamma]] = (\beta_0 + \beta_1) + (\beta_2 + \beta_3)2^2 + (\beta_4 + \beta_5)2^4 + \cdots$$

Finally, because $2^{2k} = 4^k = 1 \pmod 3$, we have

$$[[\beta]] - [[\gamma]] = \sum_{i=1}^{b} \beta_i = \|\beta\| \pmod 3,$$

the required result.

In notation of the programming language C this translates to

```
( beta - ((beta >> 1) & 0x5555555555555555L) ) % 3
```
(13)

which needs only 4 operations. Note the similarity to the first line of (10). By means of this 'weight modulo 3' operation, formula (12) translates to

$$V \cdot W = (\|P_1\| \bmod 3 + \|\neg P_2\| \bmod 3) \bmod 3. \tag{14}$$

The additional remainder operation is needed because we want results to be either 0, 1 or 2. This is unfortunate because now we need three modulo operations in total, and these are slow in comparison to the other machine operations.

We can avoid this problem by postponing taking the remainder until the very last moment. This leads to the following C expression to compute the dot product

```
(  (p0 - ((p0 >> 1) & 0x5555555555555555L)) +
   (notp2 - ((notp2 >> 1) & 0x5555555555555555L)) ) % 3
```
(15)

Sadly, because the addition in this expression could lead to an overflow, this method only works when the length of the vectors is strictly smaller than 64.

### 3.4 Iteration

Sometimes it is necessary to iterate through all possible vectors of a given length $n$. In the case of $\mathbb{F}_2^n$ this is straightforward : we simply iterate through all integers $0, 1, \ldots, 2^n - 1$ in their bit representation. Something similar but slightly more complicated, works for $\mathbb{F}_3^n$.

For $\beta \in \mathbb{F}_2^n$, $\beta \neq (0, \ldots, 0)$, define $\operatorname{pred}\beta$ to be the unique element of $\mathbb{F}_2^n$ that satisfies

$$[[\operatorname{pred}\beta]] = [[\beta]] - 1 \qquad \text{(subtraction in } \mathbb{Z}\text{)}.$$

Computing $\operatorname{pred}\beta$ translates to a simple 64-bit integer subtraction by 1.

For $V \in \mathbb{F}_3^n$, $V \neq (2, \ldots, 2)$, define $\operatorname{succ} V$ to be the unique element of $\mathbb{F}_3^n$ that satisfies

$$W = \operatorname{succ} V \iff \begin{cases} W_1 &= (\operatorname{pred} V_2) \mid \neg V_1 \\ W_2 &= V_1 \end{cases} \tag{16}$$

It takes only three machine operations to compute $\operatorname{succ} V$ from $V$. Note that this operation is well defined : $W_1^{(i)}$ and $W_2^{(i)}$ cannot be zero at the same time for any $i$, and hence $(W_1, W_2)$ is always a valid binary representation of some ternary vector.

**Theorem 3.** *Consider the sequence $\mathcal{S}$ of vectors of $\mathbb{F}_3^n$ defined by $\mathcal{S}_1 \stackrel{\text{def}}{=} (0, \ldots, 0)$, $\mathcal{S}_{i+1} \stackrel{\text{def}}{=} \operatorname{succ}\mathcal{S}_i$, for all $i$, $i = 1, \ldots, 3^n - 1$. Then*

- *$\mathcal{S}$ contains every element of $\mathbb{F}_3^n$ exactly once,*
- *$\mathcal{S}_{3^n} = (2, \ldots, 2)$ and hence $\mathcal{S}$ is well defined.*

*Proof.* Let us first rephrase the 'succ' operation directly in terms of $\mathbb{F}_3$. Because $V \neq (2, \ldots, 2)$ we may always find $d \leq n$ such that $V$ can be written as

$$V = (2, \ldots, 2, V^{(d)}, V^{(d+1)}, \ldots, V^{(n)}) \text{ with } V^{(d)} \neq 2. \tag{17}$$

We claim that

$$\operatorname{succ} V = (0, \ldots, 0, V^{(d)} + 1, -V^{(d+1)}, \ldots, -V^{(n)}). \tag{18}$$

Indeed, if $V$ is of the form (17), then $V_2$ is of the form

$$V_2 = (0, \ldots, 0, 1, V_2^{(d+1)}, \ldots, V_2^{(n)}),$$

and then

$$\operatorname{pred} V_2 = (1, \ldots, 1, 0, V_2^{(d+1)}, \ldots, V_2^{(n)})$$

and

$$(\operatorname{pred} V_2) \mid \neg V_1 = (1, \ldots, 1, \neg V_1^{(d)}, V_2^{(d+1)} \mid \neg V_1^{(d+1)}, \ldots, V_2^{(n)} \mid \neg V_1^{(n)}).$$

Now, in general for $v \in \mathbb{F}_3$, we have $v_2 \mid \neg v_1 = v_2 \mid (v_1 + 1) = v_2(v_1 + 1) + v_2 + v_1 + 1 = v_2$, hence

$$(\operatorname{pred} V_2) \mid \neg V_1 = (1, \ldots, 1, \neg V_1^{(d)}, V_2^{(d+1)}, \ldots, V_2^{(n)}).$$

Also, if $V^{(d+1)} = 0$ then $(\neg V_1^{(d)}, V_1^{(d)}) = (0, 1)$ which represents $1 \in \mathbb{F}_3$, and if $V^{(d+1)} = 1$ then $(\neg V_1^{(d)}, V_1^{(d)}) = (1, 0)$ which represents 2. This proves (18).

Now, consider $V, W \in \mathbb{F}_3^n$ such that $V^{(1)} = W^{(1)}$, $V^{(2)} = W^{(2)}, \ldots, V^{(k)} = W^{(k)}$ for some $k \leq n$, i.e., such that $V$ and $W$ have the same $k$-prefix. We claim that also $\operatorname{succ} V$ and $\operatorname{succ} W$ must have the same $k$-prefix. Indeed, if $(V^{(1)}, \ldots, V^{(k)}) \neq (2, \ldots, 2)$ then (18) applies to both $V$ and $W$ for the same value of $d < k$, and then the required property is immediate. If on the other hand $V^{(1)} = \cdots = V^{(k)} = W^{(1)} = \cdots = W^{(k)} = 2$, the $k$-prefix of both $\operatorname{succ} V$ and $\operatorname{succ} W$ will be equal to $(0, \ldots, 0)$. (In that case we apply (18) for possibly different values of $d$, but both for $V$ and $W$ we have $d \geq k$.)

It follows from this that the sequence of $k$-suffixes of $\mathcal{S}$ must be periodical. We will prove by induction on $k$ that the period length of this repetition is $3^k$ and that the $k$-prefix of $\mathcal{S}_{3^k}$ has all entries equal to 2.

The first few terms of $\mathcal{S}$ are easily computed to be the following :

$$(0, 0, \ldots, 0),\ (1, 0, \ldots, 0),\ (2, 0, \ldots, 0),\ (0, 1, 0, \ldots, 0),$$

so the 1-prefixes have a period length of 3 and $\mathcal{S}_3$ has the required form. This proves the base case $k = 1$ of our induction.

Now assume the properties hold for a given prefix length $k$. We claim that during one fixed period of $k$-prefixes, the $k+1$-th entries of subsequent vectors repeatedly change sign (i.e, $(\operatorname{succ} V)^{(k+1)} = -V^{(k+1)}$). Indeed, for every vector except the last one of a period, we may apply (18) with a value of $d$ that is at most $k$.

As a consequence, the first $3^k$ elements $V$ of $\mathcal{S}$ will have $V^{(k+1)} = 0$. Element $\mathcal{S}_{3^k}$ has $k+1$-prefix equal to $(2, \ldots, 2, 0)$ and hence, by (18), element $\mathcal{S}_{3^k+1}$ has $k+1$-prefix equal to $(0, \ldots, 0, 1)$. In the second period, $k+1$-th entries alternate between 1 and $-1 = 2$, and because the length of a period is odd, we find that $\mathcal{S}_{2 \cdot 3^k}$ must have $k+1$-prefix equal to $(2, \ldots, 2, 1)$. Similarly, it follows that the

$k+1$-prefix of $\mathcal{S}_{2\cdot 3^k+1}$ is $(0,\ldots,0,2)$ and that of $\mathcal{S}_{3^{k+1}}$ is $(2,\ldots,2,2)$. This proves that the first 3 periods of $k$-prefixes all yield different $k+1$-prefixes, and hence the period length for $k+1$-prefixes must be three times that of the $k$-prefixes (for it cannot be larger).

Finally, it follows that the period of $n$-prefixes is $3^n$, and hence that the elements of $\mathcal{S}$ are all different. ∎

The sequence $\mathcal{S}$ has a somewhat unnatural ordering. For example, the 9 elements of the sequence for $n = 2$ are

$$(0,0),\ (1,0),\ (2,0),\ (0,1),\ (1,2),\ (2,1),\ (0,2),\ (1,1),\ (2,2).$$

### 3.5 Other representations

A second way of representing a vector $V \in \mathbb{F}_3^n$ in computer memory is to store three machine words $V_0, V_1, V_2$ instead of just two. The advantage of this representation is that we now need only 5 operations to compute $VW$ :

$$\begin{array}{lcllcllcl} P_0 & \leftarrow & V_0W_0 & \quad T_1 & \leftarrow & V_1 \mid W_1 & \quad T_2 & \leftarrow & V_2 \mid W_2 \\ P_1 & \leftarrow & T_1T_2 & & & & & & \\ P_2 & \leftarrow & P_0 + P_1 & & & & & & \end{array} \tag{19}$$

and only 3 to compute $P_0$ and $\neg P_2$ in preparation for the dot product (cf. Section 3.3).

Addition (and subtraction) need 7 operations : compute $S_1, S_2$ as before and then finish with $S_0 \leftarrow S_1 + S_2$. Combined addition and subtraction can be done in 10 operations : drop the first two statements of (9) and add $S_0 \leftarrow S_1 + S_2$, $D_0 \leftarrow D_1 + D_2$ at the end.

If $n \leq 32$ you can represent a vector $V \in \mathbb{F}_3^n$ in a single 64-bit word : store $V_1$ in one half word and $V_2$ in the other (we shall denote the resulting word by $V_1 : V_2$). In some cases it is now possible to combine two 32-bit operations into a single 64-bit operation. For example, the product $P = VW$ can now be computed as follows :

$$\begin{array}{lcllcllcl} T_1 : T_2 & \leftarrow & (V_1 : V_2) \mid (W_1 : W_2) & \quad U_1 : U_2 & \leftarrow & (V_1 : V_2)(W_1 : W_2) \\ P_1 : P_2 & \leftarrow & (T_1 : U_1) \mid (T_2 : U_2) & & & \end{array} \tag{20}$$

This looks like a 3-op implementation, but note that we need to split and recombine two 64-bit words on the way, something which admittedly can be done fast.

Addition (and similarly, subtraction) also needs few operations. The following implementation is derived directly from (7) :

$$\begin{array}{rcl} T_1 : T_2 & \leftarrow & (V_1 : V_2) + (W_1 : W_2) \\ U_1 : U_2 & \leftarrow & (T_1 : T_2) + (V_2 : V_1) \\ S_1 : S_2 & \leftarrow & (T_2 : T_1) \mid (U_1 : U_2), \end{array} \tag{21}$$

good for 3 standard operations and two half word 'swaps' ($X_1 : X_2 \rightarrow X_2 : X_1$) each of which can be encoded as a single machine instruction.

## 4 Benchmarks

We have implemented and measured the speed of the operations discussed in the previous sections in various settings. We used the following test programs :

1. To test vector addition and subtraction we computed the echelon forms of 200000 square $n \times n$ matrices.
2. To test combined addition and subtraction, we computed all 6561 elements of the vector space generated by 8 vectors of length $n$ (and did this 5000 times).
3. We determined the Hamming distance between every pair of vectors in a set of 10000 vectors of length $n$.
4. Likewise, we computed the dot product of every such pair.

We did not measure the speed of iteration (Section 3.4) because it is mostly irrelevant : it is not the iteration itself that will determine the final running time of a program, but the action that is performed at every iteration.

We ran the tests above for consecutive values of $n$. The running time of each test was compared to that of a reference implementation in which vectors are represented as arrays of bytes equal to either 0, 1 or 2, and modulo 3 arithmetic was used for all operations. We did our best to use reasonably efficient code also for the reference implementations. For example, the dot product was first computed over $\mathbb{Z}$ and the remainder was only carried out at the end. Not only does this reduce the operation count, but it also allows the processor to make better use of its SIMD (i.e., vector processing) capabilities.

The test programs were written in C and compiled with an optimizing compiler (Gnu GCC). The source code of our test programs is available from `http://caagt.ugent.be/fast/`.

Spot checks on the generated assembly code convinced us that the compiler managed to use single machine instructions also in those cases where they did not have a direct C equivalent. For example, `'(v<<32)|(v>>32)'` was indeed translated to a single 'rotate right by 32' instruction. It turned out to be important to use a recent version of the compiler : with the newer versions the standard implementations made better use of the SIMD instructions of the CPU, making the overall speed gain of our new methods a little less pronounced.

We compiled and ran the tests on 6 different types of 64-bit CPU :

| Type | Release date |
|---|---|
| AMD Opteron 2212 | 15/08/2006 |
| Intel Xeon X5355 | 14/11/2006 |
| Intel Core2 E8500 | 20/01/2008 |
| Intel Core2 Q9550 | 25/03/2008 |
| Intel Xeon X5570 | 30/03/2009 |
| Intel Xeon X6560 | 16/03/2010 |

In general, the older the processor, the larger the speed gain of our new methods. This sounds a bit counterintuitive, but the reason for this is that our reference

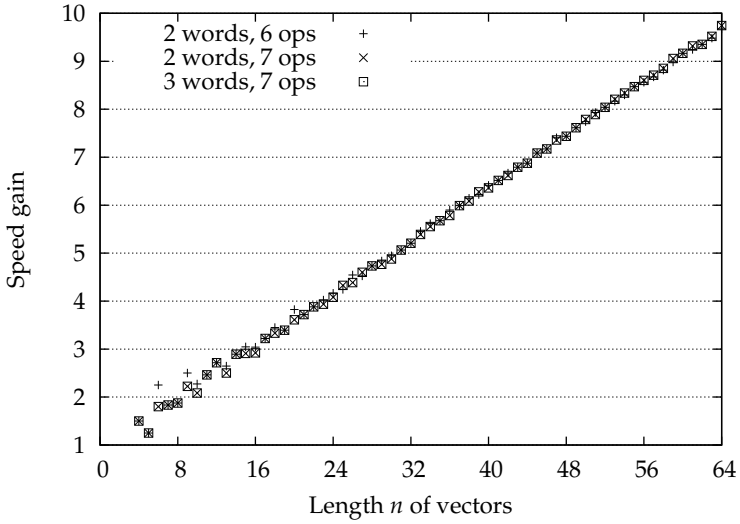


Figure 1: Addition and subtraction of vectors

implementation runs slower on older machines, because the SIMD support is not so good. For the results listed in the following pages we used the timings of the most recent processor (the X6560).

We have implemented and compared 5 different representations of vectors of $\mathbb{F}_3^n$ :

1. The representation of Section 3.1 where we use two 64-bit words for each vector.

2. The same representation but using two 32-bit words when $n \leq 32$. On recent CPUs this makes no significant difference in speed, although using only half the memory might be of advantage in some applications.

3. A representation that uses three 64-bit words for each vector (cf. Section 3.5).

4. A representation that packs two vectors $V_1, V_2 \in \mathbb{F}_2^n$ (with $n \leq 32$) into a single 64-bit word $V_1 : V_2$ (cf. Section 3.5).

5. A variant of this, which stores both $V_1 : V_2$ and $V_2 : V_1$ to avoid 'half word swap' operations. It turns out that this variant always performs worse than the previous one, and we shall not discuss it further.

Let us now turn to the results. In the graphs we plot the 'speed gain' of different methods against the lengths $n$ of the vectors considered. Speed gain is defined as the ratio between the running time of the reference implementation and that of the new implementation.

In Figure 1 we display the results for addition and subtraction, measured by computing the echelon form of a square matrix. We show the results for the six and seven operations version in the two word representation, cf. (7) and (8), and

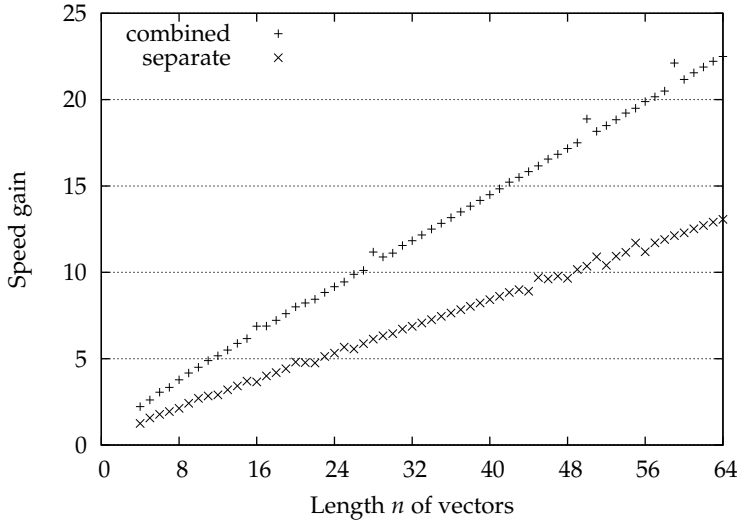


Figure 2: Combined vs. separate addition and subtraction of vectors

also the seven operation version in the three word representation of Section 3.5. It turns out that there is hardly any speed difference between the three implementations.

Figures 2 and 3 display the results of the subspace generation benchmark. The first figure shows that combining addition and subtraction by means of (9) is more than 1.7 faster than executing both operations separately. (This is at least so for the most recent CPUs. For older types the factor is closer to 1.2, reflecting the 12/10 ratio in number of operations.)

This benchmark is the only one in which there is a significant difference between the 1-, 2- and 3-word implementations (cf. Figure 3). However, because the same phenomenon appears when the addition and subtraction are not combined, this is probably a side effect of the generation algorithm itself rather than of the specific implementation of the ternary operations. It does however clearly illustrate that using more memory may significantly degrade performance, as more data needs to be moved around and the chance of cache misses becomes higher.

The most spectacular of our results is the Hamming distance benchmark where speed gains of up to 33 are reached, at least on modern CPUs that have a 'population count' instruction (cf. Section 3.2). But even on older computers a factor of 10 can still be obtained. (See Figure 4.)

Our last benchmark is used to test the dot product. In Figure 5 we compare three versions. The first uses the 'population count' instruction, the second is based on formula (15) and the third and slowest one uses formula (14) which requires three remainders to be taken. As mentioned before, the second method can only be used for $n < 64$, hence on an old computer, with $n = 64$ only the last version is available. The strange nonlinearity of the graphs is not a peculiarity of

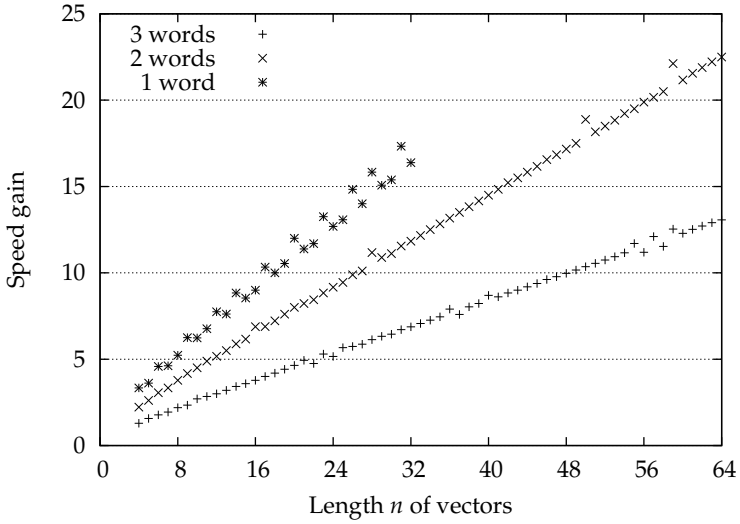


Figure 3: Combined addition and subtraction of vectors

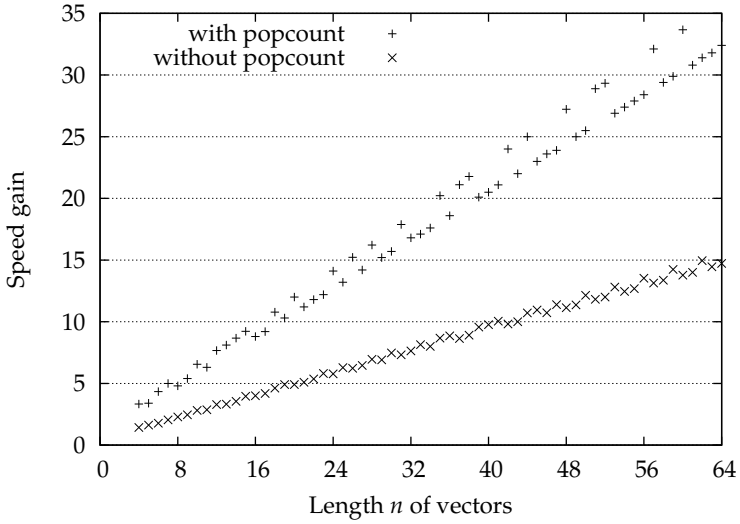


Figure 4: Computing the Hamming distance with and without 'population count' instruction

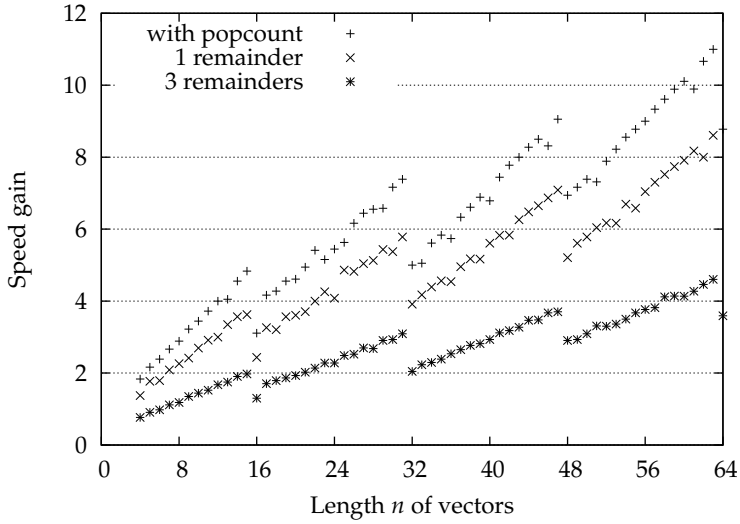


Figure 5: The dot product in three versions

our implementation but is the effect of a good optimizing compiler and a recent CPU on the reference program. Apparently on our test CPU SIMD-instructions allow the standard dot product to be computed 16 elements at a time.

As was mentioned at the start of Section 3.5, one of the advantages of the three word representation is a faster multiplication. Figure 6 shows that the dot product can indeed be computed slightly faster, by a factor of $\approx 1.09$.

## 5 Final remarks

Our methods yield the best results for vectors of length exactly 64, but in practice $n$ will often be smaller. One way to handle a vector $V \in \mathbb{F}_3^n$ of shorter length is to extend it silently with zeroes. This means that the corresponding bit vectors $V_1, V_2$ (and $V_0$) should be extended to 64-bit words by adding ones, which is a bit counterintuitive and easily leads to mistakes. Note however that none of the formulas in Theorem 1 use a binary 'not'. As a consequence, applying these formulas to the (invalid) pairs $(v_1, v_2) = (0, 0)$ and $(w_1, w_2) = (0, 0)$ will always yield the result $(0, 0)$. In other words, if you extend the bit vectors with zeroes instead of ones you will not really run into trouble. Also weight and Hamming distance remain correct. The only exception is the iteration of Section 3.4.

The question naturally arises whether techniques similar to those of this paper would also be useful for vector arithmetic over other small algebraic structures, like $\mathbb{F}_4, \mathbb{F}_5$ or $\mathbb{Z}/4\mathbb{Z}$. Because any Boolean function can be implemented in terms of binary operations, in principle there is no reason why this would be impossible. Only experiments will tell whether the resulting implementations will be sufficiently fast.

In the case of $\mathbb{F}_4$ Bouyukliev and Bakoev have already done some preliminary work by implementing addition and subtraction and multiplication with a

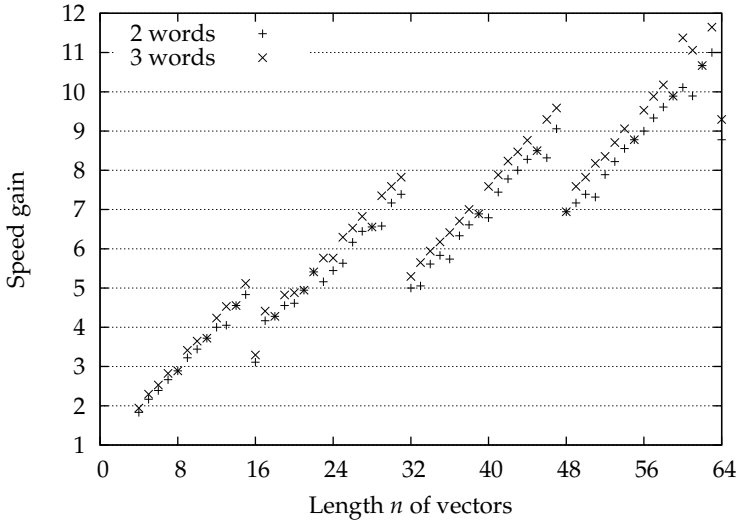


Figure 6: The dot product in 2 and 3 word representations

scalar [2]. It would be useful to extend this work with dot products, weights and Hamming distances as we did for $\mathbb{F}_3$, and with fast implementations of $V + \alpha W$ and $V + \alpha^2 W$, where $\alpha, \alpha^2$ are the elements of $\mathbb{F}_4$ different from 0 and 1. These last two operations could then be used for computing the echelon form of a matrix. (Bouyukliev and Bakoev also considered $\mathbb{F}_3$, but their implementation is slower than ours.)

## References

[1] S. E. Anderson, *Counting bits set, in parallel*, Bit Twiddling Hacks, URL `http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel` (2011)

[2] I. Bouyukliev, V. Bakoev, *Efficient computing of some vector operations over GF(3) and GF(4)*, Serdica J. Computing **2(2)** (2008), 101–108

[3] Y. Kawahara, K. Aoki, T. Takagi, *Faster Implementation of $\eta_T$ Pairing over GF($3^m$) Using Minimum Number of Logical Instructions for GF(3)-Addition*, Pairing 2008, Lecture Notes in Computer Science **5209** (2008), 282–296, Springer-Verlag Berlin Heidelberg

Department of Applied Mathematics and Computer Science,
Ghent University,
Krijgslaan 281–S9, B–9000 Gent, Belgium
`Kris.Coolsaet@UGent.be`