Serdica J. Computing **2** (2008), 137–144

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

EFFICIENT COMPUTING OF SOME VECTOR OPERATIONS OVER $GF(3)$ AND $GF(4)$

Iliya Bouyukliev, Valentin Bakoev

Abstract. The problem of efficient computing of the affine vector operations (addition of two vectors and multiplication of a vector by a scalar over $GF(q)$), and also the weight of a given vector, is important for many problems in coding theory, cryptography, VLSI technology etc. In this paper we propose a new way of representing vectors over $GF(3)$ and $GF(4)$ and we describe an efficient performance of these affine operations. Computing weights of binary vectors is also discussed.

1. Introduction.

Many algorithms in coding theory, cryptography, combinatorial circuit theory, VLSI technology, communications etc. use the standard affine vector operations (addition of two vectors and multiplication of a vector by a scalar) over a given finite field $GF(q)$, and also computing the weights of vectors. For example, they exploit such operations for:

— generating the codewords of a linear code given by its generator matrix;

— computing the minimum distance or the weight spectrum of a linear code;

— encryption of a given message and/or decryption of a given ciphertext by applying a given secret key in a block-serial manner. Affine vector operations over $GF(2^p)$ are very important and mostly used in cryptography;

ACM Computing Classification System (1998): F.2.1, G.4, I.1.1.

Key words: Finite field, Bit-wise representation, Addition of vectors, Multiplication of a vector by a scalar.

— design of adders, multipliers and decoders (for some communication devices), which use Reed-Solomon codes for detecting and correcting errors, etc.

Efficient implementations of such operations (as subalgorithms in the main algorithms) play a significant role in determining the general running-time of these algorithms. Any such implementation strongly depends on the way the vectors are represented, which implies the ways of performance of the affine vector operations and the corresponding computational costs. These problems have occurred while trying to optimize the generation of codewords and computing the number of codewords of fixed weights in linear codes [4]. The algorithms represented there are implemented in the package for investigation of linear codes Q-Extension [3] written by the first author. During their development we have performed many experiments trying to solve the same problems, for the same codes with Q-Extension and with MAGMA Computational Algebra System [5]. Comparing the running-times, we concluded that MAGMA maintains an efficient representation of vectors and affine operations with them over $GF(q)$. By applying new ways for:

(1) representation of vectors (codewords) over $GF(q)$, for $q = 2, 3, 4$;

(2) realization of the affine vector operations over these fields;

(3) computing the weight of a codeword,

we succeeded to in speeding up the algorithms more than 10 times. These ways are represented and discussed in this paper.

2. Preliminaries. The simplest and most popular way of representing an n-dimensional vector over $GF(q)$ is by using a one-dimensional array with n elements (usually bytes and then we have a *"byte-wise representation"*), which store the vector's coordinates. So the addition of two vectors and the multiplication of a vector by a scalar are implemented in n steps (for each pair of coordinates or coordinate–scalar) by looking-up in tables, representing the addition and the multiplication in $GF(q)$. These tables are computed and filled in preliminarily, and the implementation of arithmetic operations by them is known as *"tabular arithmetic"*. Another possible approach is to perform the operations over integers and to take the result modulo q, which is known as *"modular arithmetic"*. It costs significantly more computational time than the tabular arithmetic. The computing of the weight of a given vector (i.e. the number of its non-zero coordinates) is just counting the non-zero elements of the corresponding array. Hence each of these operations has a running-time of $\Theta(n)$.

The *bit-wise representation* of vectors is quite popular. This is the most natural way, especially for $q = 2$, for binary computers, where each coordinate of the binary vector is represented in one bit. If w is the length of the computer word in bits, then any n-dimensional binary vector is represented in an array of $\lceil n/w \rceil$

computer words. The addition of computer words over $GF(2)$ is implemented as a "bit-wise sum modulo 2" on a hardware level. The multiplication of a binary vector by a scalar $\lambda \in GF(2)$ is trivial.

For a given $q > 2$, $k = \lceil \log_2 q \rceil$ bits are necessary for the bit-wise representation of the elements of GF(q). Usually, one more bit (called "carry-bit") is used for ignoring the carry in addition of integers. If the computer word has w bits, then $m = \lfloor w/(k+1) \rfloor$ elements of $GF(q)$ can be represented in a single computer word. So an n-dimensional vector over $GF(q)$ is represented as one-dimensional array of $\lceil n/m \rceil$ computer words.

Because of their numerous applications, the affine vector operations over fields of the type $GF(2^p)$ are well-studied (for example, the field $GF(2^8)$ is used by the U.S. Advanced Encryption Standard, AES). When the field elements are regarded as polynomials of degree $p - 1$ with coefficients from $GF(2)$, the usual bit-wise representation of the element is in p bits, storing the coefficients of the corresponding polynomials. The addition of two vectors over $GF(2^p)$ is trivial, it is a bit-wise sum modulo 2 of the computer words representing the vectors. So the attention of the researchers is focused on the multiplication in such fields, which is more difficult. It is reduced to multiplication of the corresponding polynomials, division of the result by a given irreducible polynomial of degree p and taking the remainder. There are many efficient hardware and software implementations of this approach, some of them are patented. When the elements of $GF(2^p)$ are represented as powers of a primitive element, the multiplication of two elements is reduced to addition of the powers and taking the result modulo $2^p - 1$.

The affine vector operations over fields of type $GF(q)$, where q is prime, are not so well-studied. Some techniques for representations of vectors over $GF(3)$ and for addition of vectors over it are considered in [2, 1]. Tabular and tabular-arithmetic realizations are investigated, the results of their applications in generating the codewords of some ternary codes are discussed and compared. Addition of vectors over $GF(q), q > 2$, is considered in [6], where a bit-wise representation of the coordinates with an additional carry-bit is used. An algorithm for addition of two vectors is developed for the theoretical model "virtual two-address Random Access Machine". This algorithm performs bit-wise addition of two vectors and reduces the result modulo p, applying masks, conjunctions and shifts. A C++ implementation of this algorithm is presented in [1] and practical results about some ternary codes are given. They show that this purely arithmetic algorithm has a better running-time than the tabular algorithms mentioned above.

3. Weights of binary vectors. Four classical algorithms for computing the weight of a binary vector B, stored in an n-bit computer word, are given in [7]. The first of them checks consecutively the bits of B and counts the ones among them. So it runs in time $\Theta(n)$. The second algorithm runs as follows: while $B \neq 0$ it assigns $B = B \wedge (B-1)$ and counts these steps. Here "$\wedge$" denotes bit-wise conjunction, it replaces the rightmost one in B with zero on each step. Hence its running-time is proportional to the weight of B, i.e., it is $O(n)$. The third algorithm uses masks, shifts and bit-wise additions and it computes the weight of B in $\log_2 n$ steps. The last algorithm is based on the idea of solving the problem for all possible binary vectors with n coordinates and storing the obtained results in a table. It uses one-dimensional array W with 2^n elements and for each $i, 0 \leq i \leq 2^n - 1$, $W[i]$ contains the weight of the binary representation of the integer i. So, this weight can be taken from $W[i]$ in a constant time.

The usage of the fourth algorithm in computing the weights of binary vectors is the most effective solution for many applications among the mentioned above. We shall discuss an implementation of it.

The most essential task of the fourth algorithm is to fill in the table of weights W. We propose a simple and efficient way to do this by the following function `Fill_Weights`, written in C++. It computes the weights of all n-bits binary vectors and fills them in the array W, i.e. $W[i]$ stores the weight of the vector which is the binary representation of the integer i, for $i = 0, 1, \ldots, 2^n - 1$. It is known that if i is an integer, $0 \leq i \leq 2^k - 1$, represented by the binary vector α_i, whose weight is $wt(\alpha_i)$, and the integer $j = i + 2^k$ is represented by the vector α_j, then its weight $wt(\alpha_j) = wt(\alpha_i) + 1$. Hence we can define the weights in the array W inductively. Obviously $W[0] = 0$, $W[1] = 1$. If the values of $W[i]$ are known, for $0 \leq i \leq 2^k - 1$, then $W[i + 2^k] = W[i] + 1$, for $0 \leq i \leq 2^k - 1$ and for $k = 1, 2, \ldots, 2^{n-1}$. The function `Fill_Weights` is based on this property, which implies its correctness. It computes the weight of each serial vector directly, in a constant time, and so its running-time is $\Theta(2^n)$. Obviously, this high computational cost will be compensated when the weights of at least $2^n/n$ binary vectors have to be computed. So it is necessary to choose an appropriate size of the array W, to estimate the number of computing of weights in advance and to decide whether to use this algorithm. When its usage is not justified, some of the three mentioned above algorithms for computing the weight of a single binary vector should be used.

```
typedef unsigned short  cword;              // the type of the computer word
const unsigned int  n = sizeof(cword) * 8;  // number of bits in a computer word
const unsigned int  dim = 1 << n;           // the size  dim = 2^n;
unsigned char  W [dim];                     // the array containing the weights

void Fill_Weights ( )
{
   cword k= 1;
   W[0]= 0;  W[1]= 1;  // initial values, for k=1
   int m= 2;     // m = 2^k
   while (k < n)
   {
      for (int i= 0; i< m; i++)
         W[i+m]= W[i]+1;
      k++;
      m<<=1;     // instead of m=2*m
   }
}
```

4. Computing the affine vector operations over $GF(3)$ and $GF(4)$. We assume that the vectors have n coordinates and n is equal to the size of the computer word (when the number of the coordinates is smaller, then the vector can be filled at the left by zeroes to the size of the computer word). Further we use the signs "$\neg$", "$\wedge$", "$\vee$", and "$\oplus$" to denote the bit-wise boolean functions: negation (binary complement), conjunction, disjunction and exclusive or (sum modulo 2) of computer words, correspondingly. We propose a representation of vectors, which is different from the one given above.

Firstly we consider the field $GF(3)$. Let $\alpha = (a_{n-1}, \dots, a_1, a_0)$ be a vector over it. We represent α in three computer words α_0, α_1 and α_2 as follows: $\alpha_0 = (a'_{n-1}, \dots, a'_1, a'_0)$, where the bit $a'_i = 1$ when $a_i = 0$ and $a'_i = 0$ otherwise, for $i = 0, 1, \dots, n-1$ (i.e. α_0 is a characteristic vector of these coordinates of α, equal to 0). Analogously, $\alpha_1 = (a''_{n-1}, \dots, a''_1, a''_0)$, $a''_i = 1$ if $a_i = 1$ and $a''_i = 0$ otherwise, for $i = 0, 1, \dots, n-1$ (α_1 is a characteristic vector of these coordinates in α, equal to 1), and $\alpha_2 = (a'''_{n-1}, \dots, a'''_1, a'''_0)$, $a'''_i = 1$ if $a_i = 2$ and $a'''_i = 0$ otherwise, for $i = 0, 1, \dots, n-1$ (α_2 is a characteristic vector of these coordinates in α, equal to 2). So, for the bits in position i in α_0, α_1 and α_2, only one of them is 1, and the rest two bits are zeroes, for $0 \le i \le n-1$. For example, if $\alpha = (1, 2, 1, 0, 2, 2, 1, 0)$, then $\alpha_0 = (0, 0, 0, 1, 0, 0, 0, 1)$, $\alpha_1 = (1, 0, 1, 0, 0, 0, 1, 0)$ and $\alpha_2 = (0, 1, 0, 0, 1, 1, 0, 0)$. We denote this way for representation of a given vector α by $\alpha = (\alpha_0, \alpha_1, \alpha_2)$.

The addition and the multiplication in $GF(3)$ are represented by the following well-known tables:

$+$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Addition in $GF(3)$

$*$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Multiplication in $GF(3)$

Let $\alpha = (a_{n-1}, \dots, a_1, a_0)$ and $\beta = (b_{n-1}, \dots, b_1, b_0)$ be vectors over $GF(3)$, $\alpha = (\alpha_0, \alpha_1, \alpha_2)$, and $\beta = (\beta_0, \beta_1, \beta_2)$. If their sum is the vector $\gamma =$

$(\gamma_0, \gamma_1, \gamma_2)$, then (following the table for addition) we obtain:

$$\begin{array}{rcl}\gamma_1 & = & (\alpha_1 \wedge \beta_0) \vee (\alpha_0 \wedge \beta_1) \vee (\alpha_2 \wedge \beta_2);\\ \gamma_2 & = & (\alpha_0 \wedge \beta_2) \vee (\alpha_1 \wedge \beta_1) \vee (\alpha_2 \wedge \beta_0);\\ \gamma_0 & = & \neg(\gamma_1 \vee \gamma_2).\end{array}$$

Obviously, this way of addition needs exactly 12 boolean operations and 3 assignments.

Let us consider the multiplication of the vector $\alpha = (a_{n-1}, \dots, a_1, a_0) = (\alpha_0, \alpha_1, \alpha_2)$ by the scalar $\lambda \in GF(3)$. If $\lambda.\alpha = \delta = (\delta_0, \delta_1, \delta_2)$, we have three cases, depending on λ:

— if $\lambda = 0$, then: $\delta_0 = 2^n - 1$, $\delta_1 = \tilde{0}$ and $\delta_2 = \tilde{0}$, where the constant $2^n - 1$ is computed preliminarily, and $\tilde{0}$ denotes the zero-vector of n coordinates;

— if $\lambda = 1$, then: $\delta_0 = \alpha_0$, $\delta_1 = \alpha_1$ and $\delta_2 = \alpha_2$;

— if $\lambda = 2$, then: $\delta_0 = \alpha_0$, $\delta_1 = \alpha_2$ and $\delta_2 = \alpha_1$.

So, this realization needs at most two checks (to choose the case) and no other operations, except the assignments.

To compute the weight of a given vector $\alpha = (\alpha_0, \alpha_1, \alpha_2)$, it is sufficient to compute the weight of the binary vector $(\alpha_1 \vee \alpha_2)$.

Now we consider the field $GF(4) = \{0, 1, x, x+1\}$. Any vector $\alpha = (a_{n-1}, \dots, a_1, a_0)$ over $GF(4)$ can be represented in three computer words again: α_0, α_1 and α_x. Analogously to the previous case $\alpha_0 = (a'_{n-1}, \dots, a'_1, a'_0)$ is a characteristic vector of these coordinates in α, equal to 0. The vector $\alpha_1 = (a''_{n-1}, \dots, a''_1, a''_0)$, $a''_i = 1$ when $a_i = 1$ or $a_i = x+1$, and otherwise $a''_i = 0$, for $i = 0, 1, \dots, n-1$ (i.e., α_1 is a characteristic vector of these coordinates in α, where either 1, or $x+1$ take a part). Similarly, $\alpha_x = (a'''_{n-1}, \dots, a'''_1, a'''_0)$, $a'''_i = 1$ when $a_i = x$ or $a_i = x+1$, and otherwise $a'''_i = 0$, for $i = 0, 1, \dots, n-1$ (so α_2 is a characteristic vector of these coordinates in α, where either x, or $x+1$ take a part). For example, if $\alpha = (1, x+1, 1, x, 0, 0, x, x+1)$, then $\alpha_0 = (0, 0, 0, 0, 1, 1, 0, 0)$, $\alpha_1 = (1, 1, 1, 0, 0, 0, 0, 1)$ and $\alpha_x = (0, 1, 0, 1, 0, 0, 1, 1)$. We denote this way for representation of a given vector α over $GF(4)$ by $\alpha = (\alpha_0, \alpha_1, \alpha_x)$.

The following tables represent addition and multiplication in $GF(4)$.

$+$	0	1	x	$x+1$
0	0	1	x	$x+1$
1	1	0	$x+1$	x
x	x	$x+1$	0	1
$x+1$	$x+1$	x	1	0

Addition in $GF(4)$

$*$	0	1	x	$x+1$
0	0	0	0	0
1	0	1	x	$x+1$
x	0	x	$x+1$	1
$x+1$	0	$x+1$	1	x

Multiplication in $GF(4)$

Let us consider the addition of two vectors $\alpha = (a_{n-1}, \dots, a_1, a_0)$ and $\beta = (b_{n-1}, \dots, b_1, b_0)$ over $GF(4)$, $\alpha = (\alpha_0, \alpha_1, \alpha_x)$ and $\beta = (\beta_0, \beta_1, \beta_x)$. Let $\alpha + \beta = \gamma = (\gamma_0, \gamma_1, \gamma_x)$. Following the table of addition we obtain:

$$\begin{aligned} \gamma_1 &= \alpha_1 \oplus \beta_1; \\ \gamma_x &= \alpha_x \oplus \beta_x; \\ \gamma_0 &= \neg(\gamma_1 \vee \gamma_x). \end{aligned}$$

So, exactly four operations are sufficient to obtain the vector γ.

The multiplication of a vector $\alpha = (\alpha_0, \alpha_1, \alpha_x)$ by the scalar $\lambda \in GF(4)$ is almost as simple. Let $\lambda.\alpha{=}\delta{=}(\delta_0, \delta_1, \delta_x)$. We have four cases, depending on λ:

— if $\lambda = 0$, then: $\delta_0 = 2^n - 1$, $\delta_1 = \tilde{0}$ and $\delta_x = \tilde{0}$;

— if $\lambda = 1$, then: $\delta_0 = \alpha_0$, $\delta_1 = \alpha_1$ and $\delta_x = \alpha_x$;

— if $\lambda = x$, then: $\delta_0 = \alpha_0$, $\delta_1 = \alpha_x$ and $\delta_x = \alpha_1 \oplus \alpha_x$;

— if $\lambda = x + 1$, then $(x+1).\alpha = x.\alpha + \alpha$. Using the given realizations of these two operations we obtain: $\delta_1 = \alpha_1 \oplus \alpha_x$, $\delta_x = \alpha_1$ and $\delta_0 = \neg(\delta_1 \vee \delta_x)$.

So, for the multiplication we have at most three checks (for choosing the case for λ) and no more than three operations (apart from the assignments).

The weight of a given vector $\alpha = (\alpha_0, \alpha_1,\ \alpha_x)$ is the same as the weight of the binary vector $(\alpha_1 \vee \alpha_x)$.

We note that the vector α_0 is unnecessary in the representation of α over $GF(4)$. It can be omitted and so $\alpha = (\alpha_1,\ \alpha_x)$. Obviously, this representation does not change the way of computing the affine vector operations over $GF(4)$ given above, except that γ_0 and δ_0 should not be computed. Hence the computing of these operations becomes much faster.

5. Conclusions. Here we proposed a new way for a bit-wise representation of vectors over $GF(3)$ and $GF(4)$. We considered implementations of the affine vector operations over each of these fields. Computing the weight of a given vector, represented in this way, was also discussed. We determined the computational costs of these operations, which allows us to call them efficient. Another reason to do this are the results of our experiments, mentioned in the beginning. We applied these representations and operations in the package Q-Extension. Using them in the procedures for computing the weight spectrum, minimum distance and the number of codewords of fixed weights in linear codes makes the corresponding algorithms run more than 10 times faster (in comparison with the usage of byte-wise representation of the codewords and the tabular arithmetic operations). The experimental results show, that for $GF(2), GF(3)$ and $GF(4)$, Q-Extension and MAGMA 2.11-13 (student version) have comparable running-times for solving the same problems, for the same codes, on the same computer. Moreover in computing the weight spectrum Q-Extension runs faster.

REFERENCES

[1] Baicheva Ts. Covering radii of certain classes of linear codes, Ph.D. Thesis, 1998.

[2] Baicheva Ts., K. Manev. Finding a linear closure of a set of vectors over a finite field of a non-binary characteristic. *Mathematics and Education in Mathematics* **23** (1994), 313–318. (in Bulgarian)

[3] Bouyukliev I. What is Q-Extension? *Serdica J. Computing* **1** (2007), 115–130. `http://www.moi.math.bas.bg/~iliya/Q_ext.htm`

[4] Bouyukliev I., V. Bakoev. A Method for Efficient Computing the Number of Codewords of Fixed Weights in Linear Codes. Discrete Applied Mathematics (to appear).

[5] Computational Algebra Group at the University of Sydney, The Magma Computational Algebra System, http://magma.maths.usyd.edu.au/magma/

[6] Manev K., R. Stefanov. Yet Another Algorithm for Addition of Vectors in Non Binary Finite Field, ACCT Fifth Intern. Workshop, June 1–7, 1996, Sozopol, Bulgaria, pp. 190–194.

[7] Reingold E., J. Nievergelt, N. Deo. Combinatorial algorithms. Theory and practice, Prentice-Hall, 1977.

Iliya Bouyukliev
Institute of Mathematics and Informatics
Bulgarian Academy of Sciences
P.O. Box 323
5000 Veliko Tarnovo, Bulgaria
e-mail: `iliya@moi.math.bas.bg`

Valentin Bakoev
Department of Mathematics and Informatics
Veliko Tarnovo University
2, Theodosi Tarnovski Str.
5000 Veliko Tarnovo, Bulgaria
e-mail: `v_bakoev@yahoo.com`

Received March 6, 2008
Final Accepted May 29, 2008