





About Methods of Vector Addition over Finite Fields Using Extended Vector Registers

Maria Pashinska-Gadzheva^(✉)  and Iliya Bouyukliev 

Institute of Mathematics and Informatics, Bulgarian Academy of Sciences,
Sofia, Bulgaria
{mariqpashinska,iliyab}@math.bas.bg

Abstract. We present optimized algorithm for vector addition over finite prime fields using the extended vector registers of modern central processing units (CPU) and the corresponding extended Intel instruction sets SSE, AVX and AVX512. The presented algorithm is based on representation of the elements of the fields using unsigned 8-bit packed integer, thus allowing for computations over prime fields with up to 127 elements. The efficiency of the presented method is demonstrated in an algorithm for calculating the weight distribution of a linear code over the finite field which is known to be an NP-complete problem. An optimized approach for computing the weight of a vector is also given. The experimental results show faster execution times compared to the corresponding algorithms in the Magma and GUAVA package for GAP packages for finite fields larger than 3.

Keywords: Vector addition over prime fields · Extended registers · Linear codes

1 Introduction

Finite field arithmetic is a basis of many scientific computations connected to mathematical foundations of informatics and more precisely cryptography and coding theory among others. The vector operations over finite fields are connected to many theoretical and practical tasks. Therefore, effective algorithms for vector arithmetic in addition to hardware optimizations make many problems in this area solvable. In this paper, we present some of our work on vector

The research of the first author is partially supported by the Bulgarian National Science Fund under Contract No KP-06-H62/2/13.12.2022. The work of the second author is partially supported by the Bulgarian Ministry of Education and Science, grant no. D01-325/01.12.2023 for NCHDC. The authors acknowledge also the access to the e-infrastructure provided by the Grant No. D01-325/01.12.2023 “National Centre for High Performance and Distributed Computing” of the Ministry of Education and Science of Bulgaria.

addition over prime fields with at most 127 elements. The optimizations of the addition operation are based on vectorization using extended registers present in modern CPUs and a set of specific instructions developed by Intel [1]. This research is an extension of our work on a library [2] for the calculation of some weight characteristics of linear codes over finite fields with $p^m \leq 64$ elements for a prime p . Our goal is by using the same resources - field representation and size of the registers - to implement vector addition over two times larger fields. We compare the efficiency of different instruction sets on the same hardware. We also evaluate the effectiveness of the proposed methods compared to different mathematical commercial and open source software. We use an algorithm for calculation of the weight distribution of a linear code for this purpose.

The extended registers present in modern CPUs are intended to introduce another level of SIMD parallelism that is different from multi-core computations. The presentation of uniform data (in many cases arrays) as vectors in these specialized registers allow to execute operation over multiple elements at the same time as a single CPU instruction. This process is also known as vecotrization. Intel has developed different sets of instructions for the extended registers generalized as SSE, AVX and AVX512 [1] based on the register length. They implement some operations over different data types including addition of integer and real numbers, bitwise and logical operations, etc. For our research purposes, we need the implementation of modulo operation that gives us the remainder after division presented in C/C++ by the ‘%’ operator. This operation is at the base of the arithmetic over finite fields and does not have direct implementation for the extended registers due to its complexity. Therefore, our algorithm for vector addition over prime fields is based on the replacement this specific operations by a set existing functions for the different extended registers. This implementation is more efficient than the existing software packages such as MAGMA [3] and GUAVA package for GAP [4].

The following Sect. 2 gives some preliminaries on linear codes. Section 3 describes in detail the proposed methods for vector addition and the calculation of the weight of a vector. Section 4 presents some experimental results on the efficiency of the proposed algorithms. We give some conclusionary remarks on the use of extended vector registers in Sect. 5.

2 Preliminaries

Without a loss of generality we can consider the elements of finite prime field F_p as the residuals modulo p [5]. Thus, the elements of F_p can be represented by the set of integer numbers $\{0, 1, \dots, p-1\}$. Let us consider the n dimensional vector space F_p^n . The (*Hamming*) weight of a vector $v \in F_p^n$ is defined as the number of its non-zero coordinates. A linear code is any k -dimensional subspace of the vector space F_p^n . The elements of a linear code are called *codewords* and the parameters k and n are called *dimension* and *length* of the code respectively. A $k \times n$ matrix G whose rows form a basis of the code is called *generator matrix*. Therefore, all codewords can be generated as linear combination of the rows of

a generator matrix. The weight distribution of a linear code C is the sequence (A_0, A_1, \dots, A_n) , where A_i is the number of codewords with weight $i = 0, 1, \dots, n$.

One universal approach to calculate the weight distribution of a linear code is to generate all $(q^k - 1)/(q - 1)$ non proportional codewords as linear combinations of the rows of a generator matrix. For prime fields it can be achieved by using addition of only two vectors [6]. This problem is known to be NP-complete [7] and the optimizations are expected to integrate parallelization. Another approach to optimize the execution time of such algorithm is to accelerate the other two main parts of the computations - vector addition and the calculation of the weight of a vector.

The primary purpose of this work is to present implementation of the vector addition function and calculation of the weight of the vector by using the different instructions form SSE, AVX and AVX512 instruction sets. The arithmetic for finite fields F_2 , F_3 and F_4 can be implemented using bitwise representation [8,9]. Therefore, here we consider only fields F_p , where $p > 3$. The elements of the field can be represented by packed 8-bit integers. The implementation of vector addition is different for the signed and unsigned data types. When using signed integer numbers there is further limitation on p , primarily $p < 64$. However, here we use a smaller number of instructions. The unsigned data type allows for the computations to be executed for fields with larger number of elements ($p < 128$). The number of instructions that are used is larger but it does not affect the performance significantly. Therefore, we will present in more detail an implementation using unsigned data type.

The second main function needed for the computations calculates the weight of a vector in the field F_p . The main idea is to find the number of zero elements w_0 in a vector. Therefore, the weight of the vector can be easily obtained using a formula that depends on the length of the used register. The value of w_0 is calculated using instructions for the extended register that mark the zero elements and a the CPU instruction *popcnt*. It returns the number of non-zero bits in a 32-bit or 64-bit computer word depending on the architecture. Its latency is greater than some of the instructions for the extended registers, hence we minimize its use. There is also an alternative for calculation of the non-zero bits in a computer word that uses masks [8].

3 Implementation

In this section we present different implementations of the vector addition over prime finite field F_p with $p > 3$ elements. Let $a, b, c \in F_p^n$ and $wt(c)$ is the weight of vector c . We want to calculate $c = a + b \mod p$ and the value of $wt(c)$. The addition of elements over F_p can be computed by subtracting p when the result is greater than p . Algorithm 1 shows the implementation using this method and the calculation of the weight of the resulting vector. The parallelized versions with extended registers is based on Algorithm 1. The following subsections describe in detail the method for these calculations with different instruction sets.

Algorithm 1. Vector addition and weight computation using subtraction

```

1: function ADD_WEIGHT(a,b,c)
2:   int wt  $\leftarrow$  0;
3:   for(int i = 1; i=n; i++) do{
4:     c[i] = (a[i] + b[i]);
5:     if (c[i]  $\geq$  p) then c[i] = c[i] - p;
6:     if(c[i] $\neq$ 0) then wt++;}
7: return wt;
8: end function

```

3.1 Vector Addition Using Extended Vector Registers

In this subsection we present a method for computing $C = A + B \bmod p$ using SSE, AVX and AVX512 instruction sets, where A , B and C are extended registers of appropriate length. The elements of F_p are represented by 8-bit integers. We can store 16, 32 or 64 elements into registers with 128, 256 and 512 bits respectively. For simplicity the following descriptions are for vectors with length 16, 32, and 64 for the three types of registers. The different implementations of vector addition over F_p also use at least one of following constant registers $ZERO = (0, 0, \dots, 0)$ and $P = (p, p, \dots, p)$ with the appropriate length for the different register lengths.

There are two main approaches for the implementation based on whether the elements are presented by signed or unsigned integers. The implementation integrating signed data types has three main steps - addition of the two vectors, subtraction of vector P and an appropriate blending of the results of the previous steps. Here we use an instruction that allows to blend two vectors (e.g. `_mm_blendv_epi8` for 128-bit register). The resulting register gets one component of two registers based on the value of the corresponding component in a third (mask) register. This algorithm works correctly for $p < 64$ since the addition of two elements greater than 64 can result in a negative integer (data overflow).

Algorithm 2 shows a detailed implementation using unsigned numbers. There are a few differences when working with unsigned data type. Firstly, Algorithm 2 allows for computations to be executed using 8-bit integers for finite fields with up to 127 elements without encountering data overflow. The instructions for unsigned integers integrate saturation. When an instruction uses saturation the result is either 0 or FF whenever it gets out of range for the given data type. This implementation uses more instructions. As can be seen, the supplementary operations generate a mask register, marking which elements of the addition result are to be chosen for vector C . An advantage here is the integration of bitwise operations, executed over the full registers.

Let us now consider the implementation using 128, 256 and 512 bit registers. Most operations in the algorithm have a corresponding functions for the 128 and 256-bit registers. One important part is the comparison in the case of unsigned packed data types. The SSE and AVX instructions do not include such comparison functions. Therefore, we compare the data as signed integer only for ‘equal’

Algorithm 2. Vector addition for $p < 128$

```

1: function ADD(A,B,C)
2: const P  $\leftarrow (p, p, \dots, p)$ , ZERO  $\leftarrow (0, 0, \dots, 0)$ 
3: r1  $\leftarrow A + B$  // component sum of vectors using saturation
4: r2  $\leftarrow r1 - P$ ; // component subtraction using saturation
5: m1: if  $r1[i] == P$  then  $m1[i] \leftarrow FF$  else  $m1[i] \leftarrow 0$  //component comparison
6: m2: if  $r2[i] == 0$  then  $m2[i] \leftarrow FF$  else  $m2[i] \leftarrow 0$  //component comparison
7: r3  $\leftarrow m1$  XOR  $m2$  // bitwise XOR
8: r4  $\leftarrow r3$  AND  $r1$  // bitwise AND
9: c  $\leftarrow r4$  OR  $r2$  //bitwise OR
10: end function

```

operation guaranteeing exact match of the value regardless of the its representation. Example 1 presents a C/C++ implementation of Algorithm 2 using SSE instructions, where the comments show a partial example. The version that uses 256-bit registers is analogous and therefore is not presented here.

Example 1. //a=(1,0,0,100,70,70,...), b=(0,1,0,1,70,20,...), p=101

```

void add_128_u(__m128i a, __m128i b, __m128i c){
__m128i r1, r2, r3, r4, m1, m2, m3;
r1 = _mm_adds_epu8(a,b);          //( 1, 1, 0,101,140, 90,...)
r2 = _mm_subs_epu8(r1,P);          //( 0, 0, 0, 0, 39, 0,...)
m1 = _mm_cmpeq_epi8(r1,P);         //( 0, 0, 0, FF, 0, 0,...)
m2 = _mm_cmpeq_epi8(r2, ZERO);     //(FF,FF,FF, FF, 0, FF,...)
r3 = _mm_xor_si128(m1, m2);         //(FF,FF,FF, 0, 0, FF,...)
r4 = _mm_and_si128(r1,r3);          //( 1, 1, 0, 0, 0, 90,...)
c = _mm_or_si128(r4, r2);}         //( 1, 1, 0, 0, 39, 90,...)

```

The instruction set AVX512 has a function for ‘less than’ comparison of unsigned data types. Thus, when working with 512-bit registers we can compare the addition result for ‘less than’ P storing the result in a single *mask*. Here we will also have additional steps since the comparison functions return as result a mask register. Its length depends on the number of elements in the register of the corresponding data type (in our case 64 bits). It has non-zero bit i if the comparison is true for the i -th element. We execute supplementary command to transform this mask into a larger register needed for the rest of the computations using an all one bit register f and a blend function. Example 2 shows implementation using 512-bit registers and AVX512 instruction set.

Example 2. void add_512_u(__m512i a, __m512i b, __m512i c){

```

__mmask64 mask; __m512i r1, r2, r3, r4, f = _mm512_set1_epi8(-1);
r1 = _mm512_adds_epu8(a, b);
r2 = _mm512_subs_epu8(r1, P);
mask = _mm512_cmplt_epu8_mask(r1,P);
r3 = _mm512_mask_blend_epi8(mask, ZERO,f);
r4 = _mm512_and_si512(r3, r1);
c = _mm512_or_si512(r4,r2);}

```

3.2 Calculation of the Weight of a Vector

We can calculate the weight of a given vector $c \in F_p^n$ using the formula $n - w_0$, where w_0 is the number of the zero coordinates in c . Let us consider the implementation using 128-bit registers. For simplicity we assume $n = 16$ since the register can contain up to 16 elements ($n = 32$ for 256-bit registers and $n = 64$ for 512-bit registers). We want to construct a register $R = (R_1, R_2)$ that has exactly w_0 non-zero bits in its lower 64 bits (R_2). Therefore, the weight of vector c will be $16 - \text{popcnt}(R_2)$. For the construction of R we first execute component comparison of c with zero. The result is a register with value FF at the positions of the zero elements. We need each of those positions to be represented by a single bit and to accumulate the 128-bit data into a 64-bit computer word. The mask register $h = (2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1)$ and a bitwise AND operation are used for this purpose. The right shift of 64-bits and bitwise OR operation accumulate the result into the described form of R as shown in Example 3. The implementation with 256-bit registers needs to execute two more operations - a right shift of 128 bits and another OR operation. However, a shift of full 128-bit lane is not currently possible. Therefore, we consider the 256-bit register as 2 registers of length 128 bits. The OR operation is executed for those two 128-bit registers. This can be implemented with two auxiliary instructions - a compiler functions that does not translate into a CPU instruction and an extraction function for storing the lower 128-bits.

Example 3. $h = (2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1)$
 $a + b = c \Rightarrow (1, 1, 0, 0, 39, 90, 0, 16, 9, 100, 0, 53, 25, 0, 2, 0)$
 $m: m[i] = c[i] > 0 \Rightarrow (0, 0, \text{FF}, \text{FF}, 0, 0, \text{FF}, 0, 0, 0, \text{FF}, 0, 0, \text{FF}, 0, \text{FF})$
 $r1 = m \text{ AND } h \Rightarrow (0, 0, 2, 2, 0, 0, 2, 0, 0, 0, 1, 0, 0, 1, 0, 1)$
 $r2 = r1 \gg 64 \Rightarrow (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 2, 0)$
 $R = r1 \text{ OR } r2 \Rightarrow (0, 0, 2, 2, 0, 0, 2, 0, 0, 0, 3, 2, 0, 1, 2, 1)$
 $R_2 = (0, 0, 3, 2, 0, 1, 2, 1) \Rightarrow w_0 = \text{popcnt}(R_2)$

The implementation using AVX512 is conceptually different than the 128-bit version. We calculate w_0 only by comparing c to zero. As previously mentioned the compare functions for the 512-bit registers return as a result a mask register with the appropriate length depending on the data type. Therefore, the calculation of $wt(c)$ can be executed using a single popcnt instruction over the resulting mask of the comparison ($64 - \text{popcnt}(_mm512_cmpeq_epi8_mask(c, \text{ZERO}))$).

4 Experimental Results

In this section we present some experimental results for the efficiency of the described methods for vector addition and calculation of the weight of a vector. All presented computations were executed on a single core. We compare the execution times of an algorithm that calculates the weight distribution of a linear code over finite field that contains the two main functions presented in this paper. Table 1 shows the computation times in seconds for the versions

Table 1. Comparison for different vector length

Parameters	128-bit register	256-bit register	512-bit register
$n = 40 \text{ } k = 8 \text{ } p = 17$	5.57	5.21	3.86
$n = 120 \text{ } k = 8 \text{ } p = 17$	9.83	7.25	4.62
$n = 240 \text{ } k = 8 \text{ } p = 17$	15.36	10.58	6.10
$n = 40 \text{ } k = 5 \text{ } p = 101$	1.25	1.14	0.82
$n = 120 \text{ } k = 5 \text{ } p = 101$	2.23	1.69	1.01
$n = 240 \text{ } k = 5 \text{ } p = 101$	3.74	2.48	1.39
$n = 40 \text{ } k = 5 \text{ } p = 127$	3.13	2.85	1.99
$n = 120 \text{ } k = 5 \text{ } p = 127$	5.80	4.22	2.47
$n = 240 \text{ } k = 5 \text{ } p = 127$	9.10	6.19	3.44

integrating the 128, 256 and 512-bit registers. Computations are executed on a Intel Xeon Gold CPU with 24 cores, 48 threads, 2.3 GHz clock frequency and Linux OS. The first column shows the parameters of the codes. The next 3 columns give the execution times in seconds using different register lengths. As can be seen form Table 1 the 128-bit version is approximately 5% slower than the 256-bit version for $n = 40$ and approximately 50% slower for $n = 240$. It is also about 2 times slower than the 512-bit implementation. When we compare 256 and 512-bit version we have an average decrease of 59%. As expected the resulting speedup with the larger registers is greater for bigger lengths.

Table 2 shows a comparison of the presented 128 and 512-bit versions and two major software packages for and mathematical computations - Magma and GAP. The first column again presents the code parameters. Columns 2 and 3 show the execution times in seconds for the calculation of the weight distribution of the code using Magma and GAP packages respectively. Columns 4 and 5 show the execution times for 128 and 512-bit version. The Magma calculations were executed online Magma Calculator running in a virtual machine on an Intel Xeon Processor E3-1220, 3.10 GHz. As can be seen form the experimental results both 128 and 512 bit version are a few times faster than Magma calculator and many

Table 2. Comparison for with different software packages

Parameters	Magma	GAP	128-bit register	512-bit register
$n = 40 \text{ } k = 8 \text{ } p = 17$	24.64	840.56	5.57	3.86
$n = 80 \text{ } k = 8 \text{ } p = 17$	51.91	1667.72	7.34	4.64
$n = 40 \text{ } k = 6 \text{ } p = 31$	1.75	101.11	0.37	0.24
$n = 80 \text{ } k = 6 \text{ } p = 31$	3.63	198.84	0.53	0.29
$n = 40 \text{ } k = 5 \text{ } p = 101$	6.45	1188.84	1.25	0.82
$n = 80 \text{ } k = 5 \text{ } p = 101$	12.33	2022.77	1.67	1.00

times faster than the open source Guava package for GAP. Depending on the length of the code the presented 512-bit version gives speedup between 7 and 12 times compared to the Magma calculator. When comparing to the GAP package the presented methods can be between 150 and 2000 times faster.

5 Conclusionary Remarks

Vector addition over prime finite fields and the computation of the weight of a vector can be used in different algorithms connected to many problems in coding theory and other research areas. The presented methods and the experimental results show that using extended vector registers can result in an improvement of the execution time compared to other packages. The comparison of the implementations using different instruction sets show two main points - the presented method allows for the computations not to be affected by the value of p and that the speedup with larger registers is not necessary proportional to the length increase. Furthermore, integration of AVX512 with other parallel application programming interfaces (APIs) may result in unexpected problem and reduced CPU work frequency [1, 10]. Thus, the choice of instruction set for any given application should be carefully considered.

References

1. Intel64 and IA-32 Architectures Optimization Reference Manual. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>. Accessed 21 Feb 2023
2. Pashinska-Gadzheva, M., Bouyukliev, I.: LinCodeWeightInv: Library for Computing the Weight Distribution of Linear Codes Over Finite Fields, in preparation
3. Bosma, W., Cannon, J., Playoust, C.: The Magma algebra system I: the user language. *J. Symbolic Comput.* **24**, 235–265 (1997)
4. GAP package GUAVA. <https://www.gap-system.org/Packages/guava.html>. Accessed 21 Feb 2023
5. Lidl, R., Niederreiter, H.: *Finite Fields*, 2nd edn. Cambridge University Press, Cambridge (1997)
6. Bouyukliev, I., Bakoev, V.: A method for efficiently computing the number of codewords of fixed weights in linear codes. *Discret. Appl. Math.* **156**, 2986–3004 (2008)
7. Berlekamp, E.R., McEliece, R.J., van Tilborg, H.C.: On the inherent intractability of certain coding problems. *IEEE Trans. Inform. Theory* **24**, 384–386 (1978)
8. Coolsaet, K.: Fast vector arithmetic over F_3 . *Bull. Belg. Math. Soc. Simon Stevin* **20**, 329–344 (2013)
9. Bouyukliev, I., Bakoev, V.: Efficient computing of some vector operations over $GF(3)$ and $GF(4)$. *Serdica J. Comput.* **2**(2), 101–108 (2008)
10. Gottschlag, M., Brantsch, P., Belloso F.: Automatic core specialization for AVX-512 applications. In: *Proceedings of the 13th ACM International Systems and Storage Conference (SYSTOR 2020)*, pp. 25–35. Association for Computing Machinery (2020)