BULGARIAN ACADEMY OF SCIENCES
INSTITUTE OF MATHEMATHICS AND INFORMATICS

MARIA RUMENOVA PASHINSKA-GADZHEVA

OPTIMIZATION AND PARALLELIZATION OF ALGORITHMS, RELATED TO CODING THEORY

Summary of PhD Thesis

Professional Area:
4.6 Informatics and Computer Science
Scientific Discipline:
Informatics

PhD Supervisor:
Prof. Dr. Sci. Iliya Bouyukliev

Veliko Tarnovo
2024

Introduction

In the current work, approaches for optimization and parallelization of algorithms related to coding theory are considered. One of the main issues addressed here is related to the choice of a parallelization method depending on the task. The tasks that are solved are important both in theoretical and practical aspects. In practical terms, coding theory is related to communication and computer technologies related to information protection, compression, archiving, authentication and data storage, hashing, blockchain and others [12, 13, 26].

A large part of the problems that are studied in coding theory, require the use of parallel computing to solve them [4, 5, 18, 35]. For this purpose, a good knowledge of the different technologies and interfaces used for parallel computing is necessary in order to ensure their optimal use. Modern computing technologies are developing in many directions. One of the main ones is the development of multiprocessor and multicore architectures that allow calculations to be performed simultaneously. Another direction for the development of computing technology is the integration of architectures designed for specific tasks into general purpose computing systems. Such are accelerators, which are characterized by a large number of computing units [24, 29]. Systems that, in addition to a central processor with a standard architecture, also use an accelerator are called heterogeneous. As a result of this development, parallel computing systems are becoming increasingly accessible. In recent decades, computer architectures have evolved by adding more cores (computational units) to a single processor and more processors to a given system. Extended vector registers are also added to each computing core of the central processors. Each of the presented characteristics of parallel architectures is the subject of independent and extensive study. In the current work, a specific approach to parallelization that uses these extended registers will be discussed in more detail. Such an approach is called vectorization. The main idea is to perform operations on a set of elements simultaneously. Registers of various lengths, multiples of 128, have been developed, and additional instructions for the central processors have been created to work with them. Vectorization using extended registers can be performed automatically by the compiler or using special functions developed for the C/C++ languages. These functions, together with newly defined data types, are included in libraries and allow easy use of extended registers without explicit knowledge of assembly instructions. This approach also allows the use of other parallel interfaces in combination with vectorization, thus providing additional acceleration.

The achieved speedup with vectorization depends on the following main

factors:

- used instructions - CPU instructions can be considered as light (bitwise operations, comparison, addition, etc.) and heavy (blending, permutations, etc.). Light instructions are executed approximately as fast as a standard instruction (often they are executed in one CPU clock cycle), unlike heavy ones. This categorization depends on the hardware implementation.
- vectorization factor - indicates the maximum number of coordinates of a given vector that can be stored in a given register [2]. This factor depends on the size of the register and the way the elements are represented in memory.
- utilization factor - shows how much of the register is used in the calculations. Its value is between 0 and 1, with 1 using the entire register, 1/2 using half of it, and 0 using no register. This factor depends on the representation of the elements and the length of the code for which the calculations are performed.

These parameters are used in analysing the effectiveness of the implementation of the presented algorithms.

The current dissertation presents vectorization of algorithms used in the study of linear codes. Solving the main problems in coding theory involves many different algorithmic problems, such as finding weight invariants (weight spectrum, codewords with fixed-weight , etc.) of a linear code. The main goal of the dissertation is to develop optimized algorithms for calculating weight invariants of linear codes, which are based on the weight distribution of the code. They are part of many algorithms for solving code generation and classification problems. For this purpose, an optimized library **LinCodeWeightInv** has been developed, using vectorization. The library includes an interface part, a testing and verification module, and a full description of the included functionalities. Some of the algorithms are also included in the **QExtNewEdition** software package. These software packages have been used to study families of codes related to binary linear codes reaching the Grey-Rankin bound.

The current work considers the following main goals, objectives and ways to solve them:

- Vectorized implementation of algorithms for calculation of the weight spectrum of a linear code over fields with q elements, where $q \leq 64$. Two main types of algorithms have been developed for the implementation - optimized high-level algorithms that describe the approach for generating a new codeword and vectorized low-level algorithms that implement the

operations of vector addition and calculating the weight of a vector. The main optimization in the high-level algorithms consists in generating only nonproportional codewords, with each new codeword being obtained solely by vector addition. Low-level algorithms include various implementations of functions for vector addition and calculation of the weight of a vector, using special instructions and extended vector registers. The developed functions differ depending on the number of elements in the field and the selected set of extended instructions for the central processor. The efficiency of the implemented algorithms in x86 architectures has been analysed, using registers with a length of 128 and 256 bits and the corresponding instructions designed to work with them. The implementations are compared with functions for calculation of the weight distribution in the widely used computer algebra packages Magma and GAP. The impact of the compiler when using direct vectorization is also analysed.

- Extension of the basic low-level algorithms for working with prime fields with less than 128 elements is developed. A representation of the field elements in memory using an unsigned data type with a size of 8 bits is considered. This representation allows calculations to be performed over larger prime fields without going out of the range of the values for the given data type. This is possible since we use only vector addition. Its implementation uses a special instructions with saturation. The saturation functions for the extended vector registers allow a valid value to be written when going out of the range of the given data type (the minimum or maximum for the given type depending on the result) instead of truncating part of the value or misinterpreting the bitwise representation of the result. The developed algorithms are implemented using the extended instruction sets SSE4.1 and AVX512 for x86 architectures and the NEON instruction set for ARM architectures. To implement the algorithms, the main characteristics of AVX512 and NEON instructions have been studied. The efficiency of the different implementations has been analysed, using an algorithm for finding the weight spectrum of a linear code.

- The relationships between families of codes with two and three weights, related to self-complementary codes that reach the Grey-Rankin bound, have been analysed. For this purpose, four families of codes with two weights and two families of codes with three weights are defined. The relationships between the defined families are described and constructions for obtaining codes from different families are presented for a given code from any of the other families of codes. Their relationships with self-complementary codes

that reach the Grey-Rankin bound are also described. Constructions for generating codes with two weights from the described families with dimension $k+2$ are presented for a given code from a corresponding family with dimension k. These relationships between the codes are used to construct self-complementary codes with parameters $[120, 9; \{56, 64, 120\}]$.

- An optimized and portable library for calculation of weighted invariants of linear codes over fields $\mathbb{F}_q$, where $q \leq 64$, has been developed. The library includes six main interface functions, for which three different ways to input data have been developed. Two main modules have been created for using the library - an interface module and a module for testing and verification. The developed library works for x86 and ARM architectures and different sets of registers. For the implementation of the library, various platforms have been studied, which are used for creating software and preparing full documentation.

Chapter 1

This chapter presents some basic definitions and theorems in coding theory, as well as architectures and approaches for parallelization of algorithms. The exposition follows the monographs [25, 30, 32]. Let $\mathbb{F}_q$ be the finite field with q elements, and $\mathbb{F}_q^n$ be the n-dimensional vector space over the field. The *Hamming distance* between two vectors $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ is the number of coordinates in which they differ. It is denoted by $d(x, y) = |\{i | x_i \neq y_i\}|$. The *Hamming weight* of a vector $x = (x_1, \ldots, x_n)$ is the number of nonzero coordinates of the vector. It is denoted by $wt(x) = |\{i | x_i \neq 0\}|$.

A *linear code* of length n, dimension k over a finite field with q elements is called any k-dimensional subspace of the n-dimensional vector space $\mathbb{F}_q^n$. The elements of the code are called *codewords*, and the parameters n and k are called the *length* and *dimensionality* of the code, respectively. A matrix with k rows and n columns, whose rows form a basis of the code C, is called a *generator matrix* for the code. A code is *projective* if it does not contain zero and proportional coordinates.

A fundamental parameter of a linear code is its *minimum distance* $d(C)$, which is the smallest of all distances between two different codewords. The *minimum weight* of a linear code is called the smallest of all nonzero weights in the code. For a linear code C, the minimum weight and the minimum distance coincide. A linear code C of length n, dimension k, and minimum distance d is denoted by $[n, k, d]$.

The residual code of a given code C with respect to a codeword $x \in C$ is called the code $Res(C, x)$, which is obtained by restricting C to the zero coordinates of the codeword x. The sequence $(A_0, A_1, \ldots, A_n)$, where A_i is the number of codewords with weight i, is called the *weight spectrum* of the linear code. For every linear $[n, k, d]$ code, $A_0 = 1$ and $A_1 = A_2 = \cdots = A_{d-1} = 0$. It can also be seen that $A_0 + A_1 + \cdots + A_n = q^k$. Two linear $[n, k, d]$ codes over a field $\mathbb{F}_q$ are called equivalent if all codewords of one code can be obtained from the codewords of the other by a sequence of the following transformations:

- permutation of coordinates;
- multiplication of elements in a given coordinate by a nonzero element of $\mathbb{F}_q$;
- application of an automorphism of the field to elements in all coordinate positions.

The equivalence relation divides the set of all codes with given parameters into equivalence classes. The classification problem for linear codes consists of finding a representative of each equivalence class.

Some of the methods and algorithms for generation and classification of linear codes use the entire or partial weight spectrum of the code. Equivalent codes have equal minimum distances and weight spectra, since the allowed transformations preserve the Hamming weight. This makes the problem of finding the weight spectrum of a linear code fundamental in coding theory. The problem has been shown to be NP-complete. [3].

The problems of construction, classification, finding a weight spectrum for linear codes generally require a large computational resource. Then the optimization and parallelization of algorithms are important for solving the given problem in a reasonable time. More about the optimization of C/C++ program code and some popular parallelization techniques can be found in[20, 21, 31, 33, 34, 36].

Vectorization is among the most suitable approaches for parallelization of algorithms related to the study of linear codes, since the main calculations are related to operations on vectors. Vectorization in modern processors is based on the use of extended registers, the length of which is a multiple of the length of a computer word. The achieved speedup with vectorization depends on the following main factors:

- the used instructions (light or heavy);
- vectorization factor [2];

- utilization factor.

These parameters are used in analysing the effectiveness of the implementation of the presented algorithms.

Additional data types and instructions have been developed to work with extended registers. The instructions for 128-bit registers on x86-based CPUs are known as Streaming SIMD Extensions (SSE). Other instructions exist for different architectures (e.g. NEON instructions for ARM architectures). The instructions for 256-bit registers are called Advanced Vector Extensions (AVX), while the instructions for 512-bit registers are known collectively as AVX512.

One basic function available on most modern CPUs is a function for finding the number of non-zero bits in a computer word, *popcnt*. This function is used in algorithms for finding the weight of a vector.

Chapter 2

This chapter discusses two main types of algorithms used to calculate the weight spectrum of a linear code - high-level algorithms, which present the approach to generating a new codeword, and low-level algorithms, which perform the basic calculations. The high-level algorithms are based on the approach using emulation of nested loop presented in[9].

High-level algorithms

The generation of only non-proportional codewords is performed by skipping linear combinations whose first non-zero coefficient is different from 1. For this purpose, a temporary matrix is used, which in row i contains a codeword obtained as a linear combination of i rows of the generator matrix.

The main optimization of the presented algorithm consists of replacing the vector multiplication operation, which is computationally heavy, with a vector addition operation. When working over prime fields, this is easily achievable, since the elements of a prime field are residues modulo p. The addition and multiplication operations are performed modulo p. Then, multiplication by an element of the field can be replaced by adding the same row of the generator matrix to the current linear combination.

The basic idea of the optimization can be described as follows: if the current row of the generator matrix is added to a linear combination of i rows with a coefficient 1 (it is added for the first time to some linear combination of the $i-1$ row), then it is added to row $(i-1)$ of the temporary matrix. Thus, a linear combination of i rows is obtained. Otherwise (the current row of G is added with a coefficient $\neq$ 1 in the linear combination), the row is added to an existing linear combination of i rows. For the fields $\mathbb{F}_2$ and $\mathbb{F}_3$, the description of the algorithm can be simplified.

The elements of a composite field $\mathbb{F}_q$, where $q = p^m$, can be represented as polynomials with coefficients over the finite field $\mathbb{F}_p$ and degree less than m. The elements of the field can also be represented as vectors in the prime field. Then the addition of vectors over the field is easy to implement. For optimization, the presented algorithm uses preliminary calculations, which are performed once before starting the main calculations. The set of generator matrices $M = \{G, xG, \ldots, x^{m-1}G\}$ is generated in advance. A primitive generator polynomial is used and therefore for the primitive element we have $\alpha = x$. In the presence of a set M, the multiplication of a row of the generator matrix by an element of the field can be replaced by using the corresponding row of a suitable matrix from M. To select a matrix from M, the transition sequence of a p-ary Gray code is used.

Low-level algorithms

This section presents approaches to implementing low-level algorithms for vector addition and calculation of the weight of a vector. The implementations are different depending on the values of q. The basis of these algorithms is in the appropriate representation of the elements of the field in a register. Two main representations of the elements of the field in memory are considered - bitwise and bytewise.

The use of bitwise representation of the elements and bitwise operations can be considered as the most natural approach to parallelization. For fields with 2, 3 and 4 elements, different bitwise representations and approaches to performing operations have been developed [1, 8, 15, 23, 28].

Vectors over a field with two elements have a natural binary representation - each bit can correspond to one coordinate of the vector. An interesting case is the n-dimensional vector space, where $n \leq 64$. In this case, more than half of the bits of the register do not carry information. To generate the codewords of a code C with length $n \leq 64$, one can use the $[n, k-1]$ subcode C' with a generator

matrix G', obtained from G by removing the last row g_k. We also consider the coset $g_k + C' = \{g_k + c | c \in C'\}$. Then the original code C can be represented as the set $C' \cup (g_k + C')$. By appropriately writing the generator matrix and the temporary matrix, codewords of C' and $(g_k + C')$ are generated simultaneously. To perform the operation of calculation of the weight of a vector, a 128-bit register can be considered as an array of two 64-bit computer words. The weights of two generated codewords are calculated using the *popcnt* instruction. For $n > 64$, the same instruction is used to calculate the weight, summing the obtained values.

For finite fields with three elements, there are different approaches to bitwise representation of the elements of the field [1, 8, 15, 23, 28]. One such representation allows for the storage of an element of the field in two bits, with the addition of the elements being performed by six bitwise operations [15]. This representation is used in low-level algorithms for fields with three elements. It is expressed by the following representation $\Pi : \mathbb{F}_3 \to \mathbb{F}_2^2$, where

$$\Pi(0) = (1, 1)$$

$$\Pi(1) = (1, 0)$$

$$\Pi(2) = (0, 1)$$

The mapping Π can be extended for vectors over the finite field to the mapping $\pi : \mathbb{F}_3^n \to \mathbb{F}_2^{2n}$, where

$$\pi(v) = (\alpha_1, \alpha_2, \ldots, \alpha_n, \beta_1, \beta_2, \ldots, \beta_n),$$

$$v = (v_1, v_2, \ldots, v_n) \in \mathbb{F}_3^n, \Pi(v_i) = (\alpha_i, \beta_i).$$

Using these representations, the operation of vector addition over $\mathbb{F}_3$ can be implemented in 5 register operations. The representation π also allows easy calculation of the number of nonzero elements of a given vector. Since the element 0 is mapped to $(1, 1)$, for the calculation of the weight of a vector v using the representation

$$\pi(v) = (\alpha_1, \alpha_2, \ldots, \alpha_n, \beta_1, \beta_2, \ldots, \beta_n),$$

a *XOR* operation is used for the α and β parts of $\pi(v)$. Afterwards, a *popcnt* function is executed.

The elements of the prime field $\mathbb{F}_p$ can be represented as integers in the interval $[0, p-1]$. Therefore, in cases where $p \neq 2, 3$, 8-bit integers can be used for $p < 128$. The presented algorithm for vector addition over a prime field uses three extended register instructions - addition, subtraction and blending. The operation

is performed for fields with up to 64 elements, because a memory representation using character data types is used. For performing the blending instruction, it is important whether the result of the subtraction is a negative number.

Two main approaches have been considered for finding the weight of a vector over a prime field. The first approach finds the number of zero bytes and uses several calls to the heavy function *popcnt*. The number of calls depends on the length of the register. The second approach finds the number of zero coordinates using a mask, bitwise operations, and a single call to the *popcnt* function.

The vectors in $\mathbb{F}_{p^m}^n$ are represented as vectors in $\mathbb{F}_p^{mn}$. Then the operation vectors addition reduces to its implementation over the prime field. Finding the weight of a vector again is reduced to performing the operations over the prime field, using additional bitwise operations. The number and type of additional operations depends on p.

Performance analysis

To analyse the efficiency of the developed algorithms, comparisons were made with the computer algebra systems Magma and GAP. Also, comparisons were made between the running time of the implementations with 128- and 256-bit registers, as well as a comparison between the running time of the implementation with 256-bit registers and the implementation without vectorization of the algorithms. Experimental results show that when using bitwise representation ($\mathbb{F}_2$,$\mathbb{F}_4$, $\mathbb{F}_3$,$\mathbb{F}_9$, $\mathbb{F}_{27}$) the 256-bit version is comparable to the 128-bit version. For composite fields, the execution time with AVX2 is between 10% and 30% less compared to SSE4.1. In the case of bytewise representation ($\mathbb{F}_5$, $\mathbb{F}_{25}$), the 256-bit version is between 1.2 and 1.7 times faster than the 128-bit version.

For the prime fields $\mathbb{F}_2$, $\mathbb{F}_3$ and $\mathbb{F}_5$, between 1.4 and 2.5 times faster calculations are observed compared to the Magma system, with the speedup increasing with increasing code length. For composite fields, the presented vectorized version is between 1.9 ($\mathbb{F}_4, n = 500$) and 43.4 ($\mathbb{F}_{27}, n = 500$) times faster than Magma. When compared to GAP, the experimental results present a speedup between 3.3 times ($\mathbb{F}_2$) and 1480 times ($\mathbb{F}_{27}$). Here again, larger speedups are observed for composite fields.

When compared to a non-vectorized version of the algorithm, speedups between 7.8 ($\mathbb{F}_{25}$) and 58.9 ($\mathbb{F}_2$) times are observed. This is due to the high vectorization factor. In bitwise representation, the vectorization factor for a 128-bit register is between 21 and 128 depending on the field. Then, when compared to a

non-vectorized version using addition and multiplication tables for operations with elements over the field, a large speedup is naturally obtained.

Vectorization and compiler efficiency

It is known that the choice of compiler can also affect the speed of algorithms [2, 6]. For this, a comparison was made between three widely used compilers for the C/C++ languages - gcc(mingw for Windows operating system), clang and msvc. Experimental results were obtained for the Windows and Ubuntu operating systems, with only the first two compilers considered for Ubuntu. The calculations were performed for binary fields and 128- and 256-bit registers. The experimental results show that the execution times of a function for calculation of the weight spectrum of linear code when compiled with the three selected compilers are close. It is observed that the functions compiled with gcc are more suitable for use with 256-bit registers. The functions compiled with clang and msvc give better execution times for 128-bit registers.

Chapter 3

Chapter 3 describes the main features of vectorizing algorithms with the AVX512 and Neon extended instructions. The AVX512 instruction family is divided into various subcategories. Some of the features of the AVX512 instructions, which are available on some modern x86-type architectures, are as follows:

- masking registers (opmask) - used for efficient merging and conditional writing to the result registers.
- *AVX512Foundation* - basic operations for working with registers (arithmetic operations, logical operations, bitwise operations, functions for writing data to memory, etc.).
- *AVX512VPOPCNTDQ* - functions that allow finding the weight of integer elements written to a register. The functions of this subcategory are fundamental for the efficiency of the developed algorithms.
- *AVX512BW* (*Byte and Word Instructions*) - extends the basic functionalities defined in *AVX512Foundation* by adding functions for 8 and 16-bit integers. Such functions are used to implement algorithms for vector addition over prime fields with bytewise representation.

Some of the main features of NEON instructions for ARM architectures are as follows:

- Vectorization by writing elements of the same type in 64- or 128-bit registers.
- A different syntax that describes the type of data that will be written to the register, its size, and the number of elements that will be written to the register. For example, int8x16_t contains 16 integer elements with a size of 8 bits.
- Availability of all comparison operations ($<, >, =, \leq, \geq$)
- The functions for finding the number of non-zero bits are developed by finding their number for each byte written to the register. To find the weight of an entire vector register, it is necessary to add the resulting values, which is possible with a single function.

Unsigned data types and saturation functions

The extended vector instructions for the x86 and ARM architectures have saturation functions. When a function is executed using saturation, the result of the operation is equal to the result of the standard execution of the operation, provided that the range of the given data type is not exceeded. If the result of the operation is outside the range of the data type stored in the register for a given element, then a variable containing only zero bits is written to the output register for the corresponding element when the value is less than the lower limit for valid value of the given data type. When the result is greater than the upper limit for valid values, a variable of the appropriate length containing only non-zero bits is written to the register at the corresponding position.

The functions implemented by saturation allow vector addition algorithms to be implemented for vectors over prime finite fields with $q < 128$ elements and a large vectorization factor. This allows computations to be performed on larger fields using the same representation and resources used to implement the low-level algorithms. To do this, the elements of the field are written in unsigned data types.

When implementing the vector addition operation using unsigned data types and saturation functions, 7 lightweight operations (arithmetic, comparison operations, and bitwise operations) are used for the SSE, AVX, and NEON instructions. When implemented with AVX512, the operation is implemented using 4 instructions, thanks to the masking registers. When implemented with AVX512,

the function for calculation of the weight of a is implemented using 2 functions - a comparison and a *popcnt* instruction for the resulting 64-bit register.

Experimental results

The efficiency of AVX512 and NEON instructions is analysed, using the algorithm for calculation of the spectrum of a linear code implemented using unsigned data types and saturation instructions. The execution time using AVX512 is between 1.5 and 4.5 times faster than when using SSE instructions. The execution time using NEON is between 1.3 and 2.9 times faster than when using SSE instructions. When implementing the low-level algorithms using bitwise representation and $n \leq 500$, the implementations using AVX512 and NEON instructions have similar execution times. Experimental results also show that at a utilization factor less than 1/2, it is appropriate to use registers with a length of 128 bits.

Chapter 4

A binary linear $[n, k]$ code C is called self-complementary if for every codeword $x \in C$, its complement $\overline{x}$, where $x + \overline{x} = \mathbf{1}$, is also a codeword ($\overline{x} \in C$). For any length n and minimal distance d, the Grey–Rankin bound is an upper bound for the cardinality of a binary self-complementary code C. It states that:

$$|C| \leq \frac{8d(n-d)}{n-(n-2d)^2}. \tag{1}$$

provided that the right-hand side is positive. The bound also holds for nonlinear codes, but we consider only linear codes here. The parameters of linear codes that reach the Grey-Rankin bound are:

$$\begin{gathered}[2^{2m-1}-2^{m-1}, 2m+1, 2^{2m-2}-2^{m-1}], \\ [2^{2m-1}+2^{m-1}, 2m+1, 2^{2m-2}].\end{gathered} \tag{2}$$

Self-complementary codes that reach the Grey-Rankin bound have been widely studied. Of particular interest are their subcodes, due to their connections to strongly regular graphs [10, 11, 16], quasi-symmetric SDP designs [22, 27], bent [14, 17] and vectorial bent funtions [19] and others. It is proven that inequivalent codes yield nonisomorphic SDP designs and the number of inequivalent codes

and its associated SDP design coincide. Jungnickel and Tonchev showed that the numbers of nonisomorphic quasisymmetric SDP designs and inequivalent self-complementary codes with parameters shown in (2) grow exponentially with m from an argument by Kantor [27].

Projective Complementary Codes and Self-Complementary Equivalence

Definition 1. *Let C be a projective $[n,k]$ code with generator matrix G. Lets reorder the columns in the generator matrix S_k of the simplex code such that the obtained matrix is $S'_k = (G|\overline{G})$. The matrix S'_k generate a simplex code (more precisely, the matrices S_k and S'_k generate equivalent codes, and we call them both simplex codes). Then the matrix $\overline{G}$ generates a code of length $2^k - 1 - n$ called the projective complementary code of C and denoted by $\overline{C}$.*

Lets consider the following construction for a given even integer n:

- If C is an $[n, k-1]$ linear code that is not self-complementary, $\widehat{C} = C \cup (\mathbf{1} + C)$. This means that if G is a generator matrix of C, the matrix $\widehat{G} = \begin{pmatrix} 1 & \cdots & 1 \\ & G & \end{pmatrix}$ is a generator matrix of $\widehat{C}$.
- If C' is an $[n-1, k-1]$ linear code that is not self-complementary, $\widehat{C'} = (0|C') \cup (1|\mathbf{1} + C')$. This means that if G' is a generator matrix of C', the matrix $\widehat{G'} = \begin{pmatrix} 1 & \cdots & 1 \\ & G' & 0 \end{pmatrix}$ is a generator matrix of $\widehat{C'}$.

We can now introduce an equivalence relation as follows:

Definition 2 (Self-Complementary Equivalence)**.** *Lets consider binary liner codes C_1 and C_2 that foes not contain the vector **1**.*

- *Let C_1 and C_2 be $[n, k-1]$ codes. They are called* self-complementary equivalent *if the codes $\widehat{C_1}$ and $\widehat{C_2}$ are equivalent self-complementary codes with length n.*
- *Let C_1 and C_2 codes with parameters $[n-1, k-1]$ and $[n, k-1]$, respectively. They are called* self-complementary equivalent *if the codes $\widehat{C'_1}$ and $\widehat{C_2}$ are equivalent self-complementary codes with length n. Analogously, if C_1 and C_2 are codes with parameters $[n, k-1]$ and $[n-1, k-1]$, respectively, they are* self-complementary equivalent *codes, if their respective self-complementary codes with length n are equivalent.*

- *Let C_1 and C_2 be $[n-1, k-1]$ codes. They are called* self-complementary equivalent *if the codes $\widehat{C_1'}$ and $\widehat{C_2'}$ are equivalent self-complementary codes with length n.*

With this equivalence relation in mind we have that for the self-complementary code C all subcodes, not containing the all-ones vector, define a self-complementary equivalence class (SCE). Thus, if we have a representative from the class we know all codes of the class up to equivalence.

Families of codes with two and three weights

Let C be a self-complementary binary projective code with dimension $k+1$ that meets the Grey-Rankin bound. Using the following notations, where $k = 2m$:

$$t_k = 2^{k-2}, t_{k^\pm} = t_k \pm 2^{m-1},$$

$$T_{k^\pm} = 2^{k-1} \pm 2^{m-1}$$

we define:

- four families of two-weight linear codes with the following parameters:
  - Φ_{k^-}: $[T_{k^-}, k, \{t_{k^-}, t_k\}]$
  - Φ_{k^+}: $[T_{k^+}, k, \{t_k; t_{k^+}\}]$
  - Φ'_{k^-}: $[T_{k^-} - 1, k, \{t_{k^-}, t_k\}]$
  - Φ'_{k^+}: $[T_{k^+} - 1, k, \{t_k; t_{k^+}\}]$
- two families of three-weight codes with the following parameters:
  - Ψ_k: $[2^k, k+1, \{T_{k^-}; 2^{k-1}; T_{k^+}\}]$
  - Ψ'_k: $[2^k - 1, k+1, \{T_{k^-}; 2^{k-1}; T_{k^+}\}]$

The main relations between these six families are the following:

- Codes from the following pairs of families are projective complementary codes: Φ_{k^+} and Φ'_{k^-}; Φ_{k^-} and Φ'_{k^+}; Ψ_k and Ψ'_k.
- From a code in one of the families Φ_{k^-}, Φ_{k^+}, Φ'_{k^-}, Φ'_{k^+} one can construct codes from the other three families.
- From a code from the family Ψ_k one can construct a code in Ψ'_k and vice versa.

- If C is a code in from the family $\Phi_{k^{\pm}}$ (or $\Phi'_{k^{\pm}}$) then its residual code with respect to a codeword of weight $t_{k^{\pm}}$ belongs to the family Ψ_k (resp. Ψ'_k).
- Let Ψ_{S_k} and Ψ'_{S_k} be codes from the families Ψ_k and Ψ'_k, respectively, such that they contain the simplex code as a subcode. If $C \in \Psi_{S_k}$ (resp. $C \in \Psi'_{S_k}$) and $v \in C$ has weight $T_{k^{\pm}}$ then $Res(C, v) \in \Phi_{k^{\mp}}$ (resp. $\Phi'_{k^{\mp}}$).

Construction of codes from Φ_{k+2} using codes with dimension k

For code in Φ_k we consider the following constructions for generation of codes with dimension $k+2$. These constructions are named *self-complementary lifting (SCL)*.

Theorem 1. *Let $A \in \Phi_{k^{\pm}}$ be a code with a generator matrix G_A and B be the code generated by the matrix*

$$G_B = \left(\begin{array}{c|c|c|c} G_A & G_A & G_A & \overline{G_A} \\ 1\ldots1 & 1\ldots1 & 0\ldots0 & 0\ldots0 \\ 0\ldots0 & 1\ldots1 & 1\ldots1 & 0\ldots0 \end{array}\right).$$

Then $B \in \Phi'_{(k+2)^{\pm}}$.

Theorem 2. *Let $A \in \Phi'_{k^{\pm}}$ has generator matrix G_A and B has generator matrix*

$$\left(\begin{array}{c|c|c|c} G_A\ 0 & G_A\ 0 & G_A\ 0 & \overline{G_A} \\ 1\ldots1 & 1\ldots1 & 0\ldots0 & 0\ldots0 \\ 0\ldots0 & 1\ldots1 & 1\ldots1 & 0\ldots0 \end{array}\right).$$

Then $B \in \Phi_{(k+2)^{\pm}}$.

Theorem 3. *From any code $C \in \Phi_k$ codes in Φ_{k+2} for all integers $l \geq 1$ can be constructed.*

Computational results

The following computational results are obtained using the *Generation* module of the package *QextNewEdition*[7]. The following is a summary of the obtained computational results:

- 322039 inequivalent $[63, 7; \{28, 32, 36\}]$ codes have been constructed from 7 inequivalent $[35, 6; \{16, 20\}]$ codes.

- 91337 inequivalent $[119, 8; \{56, 64\}]$ codes have been constructed from 4 inequivalent $[63, 7; \{28, 32, 36\}]$ codes.
- 2946 inequivalent self-complementary $[120, 9; \{56, 64, 120\}]$ code have been constructed from $[119, 8; \{56, 64\}]$ codes. We obtain the following inequivalent subcodes:
  - 175213 subcodes with length 120 and dimension 8;
  - 156763 subcodes with length 119 and dimension 8.
- Using the self-complementary lifting construction on 80 inequivalent $[119, 8; \{56, 64\}]$ code we obtain:
  - 80 inequivalent $[496, 10; \{240, 256\}]$;
  - 80 inequivalent $[2015, 12; \{992, 1024\}$.

Chapter 5

The developed algorithms are included in a library for calculation of some weight invariants of linear codes **LinCodeWeightInv** over a finite field $\mathbb{F}_q$, where $q \leq 64$. These algorithms are implemented using different instruction sets and can be executed on central processors with ARM and x86 architectures. The library includes the following six main functions:

- Calculation of the weight spectrum of a linear code.
- Calculation of the minimal distance of a linear code.
- Calculation of the number of codewords with weight equal for given value w.
- Calculation of the number of codewords with weight less than a given valu w.
- Determining whether codewords with a given weight w exist.
- Determining whether codewords with weight less than a given value w exist.

The main data required for the calculations are the parameters n, k and q and a generator matrix of the code. The following data entry approaches have been created for each of the interface functions:

- Reading input data from a file. The results are written to an output file with an appropriate name for the interface function.

- Generating pseudo-random code based on given values of k, n, q. The function accepts values for k, n, q as parameters and a generator matrix is constructed for the given values.
- Calculation for a generator matrix stored in memory as a two-dimensional array. The function receives as parameters a dynamic two-dimensional array of type *int*, the code parameters n, k, q and a boolean variable indicating whether the elements of the field in the generator matrix are stored in additive or multiplicative form (the variable has the value *true* if multiplicative form is used).

The main functionalities of the developed library are based on the calculation of the weight spectrum of the code. For the operation of the main functions, over 36 implementations of basic functions necessary for performing the calculations have been developed. Two additional modules have been developed for working with the library - an interface program, which aims to facilitate the use of the library, and a module for testing and verification. This module allows the testing of all functionalities for correctness of the calculations and the execution time. For this purpose, an input file is prepared beforehand. It contains linear codes for all finite fields $\mathbb{F}_q$, $q \leq 64$ and the corresponding weight spectra in the form $\{A_i\}$, where $i = 0, \ldots, n$, generated with different software. The average time for performing the calculations for each finite field when obtaining a correct result is recorded in the output file *Results*.

The CMake and Doxygen systems were used to create portable software and a detailed user guide. CMake is a product that allows you to create a C/C++ project. This software controls the basic parameters for compiling the library. A detailed guide for using the library has been created, which includes a description of its main components and how to compile, install and test it. For this purpose, the Doxygen system was used, which creates a detailed description (in our case, over 50 pages) using standardized comments in the program code. In this chapter the main features of these systems that were used in creating the library are described. The overall structure of the library, how to compile it and how to install it using CMake are also described.

Scientific and applied contributions

- Development of an algorithm for generating nonproportionate codewords that works for composite fields by using only vector addition.

- Development of an algorithm for vector addition using SSE, AVX and AVX512 instructions using bitwise representation of the elements of the fields $\mathbb{F}_2$, $\mathbb{F}_4$ and the fields with characteristic 3.
- Development of an algorithm for vector addition using SSE, AVX and AVX512 instructions and bitewise representation for prime and composite fields with q elements, where $q \leq 64$, and calculation of the the weight of a vector using a single call of the *popcnt* instruction.
- Analysis of the performance of different compilers with SSE and AVX instructions with vectorization.
- Studying the characteristics of the AVX512 and NEON instruction sets and analysing their efficiency.
- Analysing the efficiency of different instruction sets in x86 architectures.
- Development of an algorithm for vector addition using SSE, AVX and AVX512 when representing the elements of fields with up to 128 elements using unsigned integers.
- Definition of families of two- and three-weight codes related to codes reaching the Grey-Rankin bound and description of the relationships between the individual families.
- Development of a construction for generation of codes from a given family of dimension $k + 2$ using codes from a corresponding family of dimension k.
- Development of mathematical software (LinCodeWeightInv library) for calculation of some weight invariants of linear codes over fields with q elements, where $q \leq 64$, using SSE4.1, AVX2 and AVX512 instructions for x86 architectures and NEON instructions for ARM architectures.
- Presentation of features and main accents in the creation of open source mathematical software.

Approbation of the results

The results included in the dissertation were obtained independently [P2, P3] and in co-authorship with:

- Bouyukliev [P1, P5, P6]
- Bouyukliev and Bouyuklieva [P4]

The included articles have been published or submitted for review in:

- *Science Series-Innovative STEM Education* [P1, P2]
- *2022 International Conference Automatics and Informatics (ICAI)* [P3]
- *Mathematics* [P4]
- *International Conference on Large-Scale Scientific Computing* [P5]
- *ACM Transactions on Mathematical Software* [P6]

The results of the dissertation have been reported to:

- Conference with International Participation *"Innovative STEM Education"* 2021-2022 [D1, D3]
- National Seminar in Coding Theory *"Prof. Stefan Dodunekov"*, 2021-2022 [D2, D5]
- International Conference Automatics and Informatics, Varna, Bulgaria, 2022 [D4]
- 4-th Interdisciplinary PhD Forum with International Participation, Sandanski, Bulgaria, 2023 [D6]
- 14th International Conference on Large-Scale Scientific Computations, Sozopol, Bulgaria, 2023 [D7]
- Workshop *"HPC for Mathematics and Applications"*, Sofia, Bulgaria, 2023 [D8]
- International Conference *"Cryptography and Coding Theory"*, Perugia, Italy, 2023 [D9]

Acknowledgements

I would like to thank all the co-authors for their guidance and the wonderful work process. I would like to thank my colleagues from the Mathematical Foundations of Informatics section at the Institute of Mathematics and Informatics (IMI) of the Bulgarian Academy of Sciences (BAS) for the created working atmosphere, advice and constructive criticism. The feedback received over the years has undoubtedly contributed to the overall value of the research. I would also like to express my gratitude to the management and staff of IMI-BAS for the

opportunities, support and provided access to computational resources. I would also like to thank my colleagues from the Faculty of Mathematics and Informatics at the University of Veliko Tarnovo for the opportunity to learn about another way of acquiring knowledge - through teaching.

I would like to express my special gratitude to my supervisor, Prof. Dr. Sci. Iliya Buyukliev, for his patience, support, constructive advice and the interesting topics he introduced me to while working on my dissertation. The advice and acquired skills will help me in my future development, both academically and as a person.

Finally, I would like to express my gratitude to my husband for his patience and support throughout my academic journey and in every way. I also thank him for being my constant companion in all my endeavors.

Bibliography

[1] Байчева, Ц. и Манев, Кр. Намиране на линейната обвивка на множество от вектори над крайно поле с характеристика различна от 2. *Доклади на XXIII Пролетна конференция на СМБ* (1994), 313–318.

[2] Amiri, H., and Shahbahrami, A. Simd programming using intel vector extensions. *Journal of Parallel and Distributed Computing 135* (2020), 83–100.

[3] Berlekamp, E., McEliece, R., and Van Tilborg, H. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory 24*, 3 (1978), 384–386.

[4] Bikov, D., and Bouyukliev, I. Parallel fast walsh transform algorithm and its implementation with cuda on gpus. *CYBERNETICS AND INFORMATION TECHNOLOGIES 18*, 5 (2018), 21–43.

[5] Bikov, D., Bouyukliev, I., and Dzhumalieva-Stoeva, M. Boolsplg: A library with parallel algorithms for boolean functions and s-boxes for gpu. *Mathematics 11*, 8 (2023), 1864.

[6] Botor, T., and Habiballa, H. Compiler optimization for scientific computation in c/c++. In *International Conference of Computational Methods in Sciences and Engineering 2018 (ICCMSE 2018)* (2018), vol. 2040, p. 030004.

[7] Bouyukliev, I. The program generation in the software package qextnewedition. In *International Congress on Mathematical Software* (2020), Springer, pp. 181–189.

[8] Bouyukliev, I., and Bakoev, V. Efficient computing of some vector operations over gf(3) and gf(4). *Serdica Journal of Computing 2*, 2 (2008), 137–144.

[9] Bouyukliev, I., and Bakoev, V. A method for efficiently computing the number of codewords of fixed weights in linear codes. *Discrete applied mathematics 156*, 15 (2008), 2986–3004.

[10] Bouyukliev, I., Fack, V., Willems, W., and Winne, J. Projective two-weight codes with small parameters and their corresponding graphs. *Designes, Codes and Cryptography 41* (2006), 59–78.

[11] Calderbank, R., and Kantor, W. The geometry of two-weight codes. *Bulletin of the London Mathematica Society 18*, 2 (1986), 97–122.

[12] Carlet, C., Charpin, P., and Zinoviev, V. Codes, bent functions and permutations suitable for des-like cryptosystems. *Designs, Codes and Cryptography 15* (1998), 125–156.

[13] Carlet, C., Crama, Y., and Hammer, P. L. Boolean functions for cryptography and error-correcting codes., 2010.

[14] Carlet, C., and Mesnager, S. Four decades of research on bent functions. *Designs, codes and cryptography 78*, 1 (2016), 5–50.

[15] Coolsaet, K. Fast vector arithmetic over f_3. *Bulletin of the Belgian Mathematical Society-Simon Stevin 20*, 2 (2013), 329–344.

[16] Delsarte, P. Weights of linear codes and strongly regular normed spaces. *Discrete Mathematics 3*, 1-3 (1972), 47–64.

[17] Dillon, J., and Schatz, J. Block designs with the symmetric difference property. In *Proceedings of the NSA Mathematical Sciences Meetings* (1987), Fort Meade, USA, The United States Government, pp. 159–164.

[18] Dimitrov, M., and Esslinger, B. Cuda tutorial–cryptanalysis of classical ciphers using modern gpus and cuda. *arXiv preprint arXiv:2103.13937* (2021).

[19] Ding, C., Munemasa, A., and Tonchev, V. D. Bent vectorial functions, codes and designs. *IEEE Transactions on Information Theory 65*, 11 (2019), 7533–7541.

[20] Flynn, M. J., and Rudd, K. W. Parallel architectures. *ACM computing surveys (CSUR) 28*, 1 (1996), 67–70.

[21] Fog, A. Optimizing software in c++: An optimization guide for windows, linux, and mac platforms, 2004.

[22] Gulliver, T., and Harada, M. Codes of lengths 120 and 136 meeting the grey-rankin bound and quasi-symmetric designs. *IEEE Transactions on Information Theory 45*, 2 (1999), 703–706.

[23] Harrison, K., Page, D., and Smart, N. P. Software implementation of finite fields of characteristic three, for use in pairing-based cryptosystems. *LMS Journal of Computation and Mathematics 5* (2002), 181–193.

[24] Hu, L., Che, X., and Zheng, S.-Q. A closer look at gpgpu. *ACM Computing Surveys (CSUR) 48*, 4 (2016), 1–20.

[25] Huffman, W. C., and Pless, V. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, Cambridge, UK, 2003.

[26] Joux, A. *Algorithmic cryptanalysis*. Chapman and Hall/CRC, 2009.

[27] Jungnickel, D., and Tonchev, V. D. Exponential number of quasi-symmetric sdp designs and codes meeting the grey-rankin bound. *Designs, Codes and Cryptography 1*, 3 (1991), 247–253.

[28] Kawahara, Y., Aoki, K., and Takagi, T. Faster implementation of ηtpairing over gf(3m) using minimum number of logical instructions for gf(3)-addition. In *Pairing-Based Cryptography – Pairing 2008* (Berlin, Heidelberg, 2008), S. D. Galbraith and K. G. Paterson, Eds., Springer Berlin Heidelberg, pp. 282–296.

[29] Kirk, D. B., and Wen-Mei, W. H. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[30] MacWilliams, F., and Sloane, N. *The Theory of Error-Correcting Codes*. North-holland Publishing Company, 1977.

[31] Mattson, T., He, Y., and Koniges, A. *The OpenMP Common Core*. The MIT Press, 2019.

[32] Mullen, G. L., and Panario, D. *Handbook of Finite Fields*. Chapman and Hall, 2013.

[33] Padua, D. *Encyclopedia of parallel computing*. Springer Science & Business Media, 2011.

[34] Quinn, M. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Inc., 2004.

[35] Topalova, S., and Zhelezova, S. Parallelisms of pg (3, 4) with a great number of regular spreads. In *International Conference on Large-Scale Scientific Computing* (2023), Springer, pp. 444–452.

[36] Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., Haddock, S. H., Huff, K. D., Mitchell, I. M., Plumbley, M. D., et al. Best practices for scientific computing. *PLoS biology 12*, 1 (2014), e1001745.

Presented reports

[D1] Pashinska, M., Bouyukliev, I.; Using AVX instructions for optimization of linear binary code weighting algorithms; 3rd National Scientific Conference with International Participation "Innovative STEM Education"; Veliko Tarnovo, Bulgaria, 25.04.2021 - 27.04.2021

[D2] Pashinska-Gadzheva, M., Bouyukliev, I.; SSE 4.1 optimizations for algorithm for calculating the Weight Spectrum of linear codes, National Coding Theory workshop "Professor Stefan Dodunekov"; Zlatograd, Bulgaria; 04.11.2021 - 07.11.2021

[D3] Pashinska-Gadzheva, M.; Build systems for generating independent software; Fourth International Conference "Innovative STEM Education"; Veliko Tarnovo, Bulgaria; 16.03.2022 - 19.03.2022

[D4] Pashinska-Gadzheva, M.; Comparison of compiler efficiency with SSE and AVX instructions; International Conference Automatics and Informatics (ICAI) 2022; Varna, Bulgaria; 06.10.2022 - 08.10.2022

[D5] Pashinska-Gadzheva, M., Bouyukliev, I., Bouyuklieva, S.; Two-weight codes and Grey-Rankin bound, National Coding Theory workshop "Professor Stefan Dodunekov"; Arbanasi, Bulgaria; 09.11.2022 - 13.11.2022

[D6] Pashinska-Gadzheva, M.; Optimization and Parallelization of Algorithms Connected to Coding Theory, 4-th Interdisciplinary PhD Forum with International Participation; Sandanski, Bulgaria; 16.05.2023 - 19.05.2023

[D7] Pashinska-Gadzheva, M., Bouyukliev, I.; About methods of vector addition over finite fields using extended vector register; 14th International Conference

on Large-Scale Scientific Computations; Sozopol, Bulgaria; 05.06.2023 - 09.06.2023

[D8] Pashinska-Gadzheva, M., Bouyukliev, I.; Library for Computing Weight Invariants of Linear Codes Using Vectorization; HPC for Mathematics and Applications, Sofia, Bulgaria; 28.06.2023 - 28.06.2023

[D9] Pashinska-Gadzheva, M., Bouyukliev, I.; About Weight Invariants of Linear Codes and Vectorization with SSE and AVX Instruction Sets, Cryptography and Coding Theory; Perugia, Italy; 21.09.2023 - 22.09.2023

Publications

[P1] Pashinska M., Bouyukliev I., Utilizing AVX Instruction Set for Optimizing Algorithms for Weight Characteristics of Binary Linear Code. Science Series "Innovative STEM Education"IMI-BAS, 2021, ISSN:2683-1333, 151-156

[P2] Pashinska-Gadzheva, M., Build Systems for Generating Independent Software. Science Series "Innovative STEM Education 4, IMI-BAS, 2022, ISSN:2683-1333, 62-68

[P3] Pashinska-Gadzheva, M. Comparison of compiler efficiency with SSE and AVX instructions. International Conference Automatics and Informatics (ICAI), Varna, Bulgaria, 2022, pp. 56-59, indexed in Scopus doi: 10.1109/ICAI55857.2022.9960080

[P4] Bouyukliev, I.; Bouyuklieva, S.; Pashinska-Gadzheva, M. On Some Families of Codes Related to the Even Linear Codes Meeting the Grey–Rankin Bound. Mathematics 2022, 10, 4588, **IF:2.2 SJR: 0.592 Q1**, https://doi.org/10.3390/math10234588

[P5] Pashinska-Gadzheva, M., Bouyukliev, I. About Methods of Vector Addition over Finite Fields Using Extended Vector Registers. In: Lirkov, I., Margenov, S. (eds) Large-Scale Scientific Computations. LSSC 2023. Lecture Notes in Computer Science, Springer, Cham. 2024, vol 13952. pp 427–434, **SJR: 0.606 (2023)**, https://doi.org/10.1007/978-3-031-56208-2_44

[P6] Pashinska-Gadzheva, M., Bouyukliev, I., LinCodeWeightInv: Library for Computing theWeight Distribution of Linear Codes Over Finite Fields,

submited for review after second revision to ACM Transactions on Mathematical Software

List of citations

[P3] Pashinska-Gadzheva, M. Comparison of compiler efficiency with SSE and AVX instructions. International Conference Automatics and Informatics (ICAI), Varna, Bulgaria, 2022, pp. 56-59, doi: 10.1109/ICAI55857.2022.9960080

1. Błażejowski, M. Which C compiler and BLAS/LAPACK library should I use: gretl's numerical efficiency in different configurations. *Computational Statistics*, (2024), 1-26.

2. Izrailov, K. GREMC: Genetic Reverse-Engineering of Machine Code to Search Vulnerabilities in Software for Industry 4.0. Predicting the Size of the Decompiling Source Code. *2024 International Russian Smart Industry Conference*, IEEE, 2024, 622-628.