БЪЛГАРСКА АКАДЕМИЯ НА НАУКИТЕ
ИНСТИТУТ ПО МАТЕМАТИКА И ИНФОРМАТИКА

МАРИЯ РУМЕНОВА ПАШИНСКА-ГАДЖЕВА

ОПТИМИЗАЦИЯ И ПАРАЛЕЛИЗАЦИЯ НА АЛГОРИТМИ, СВЪРЗАНИ С ТЕОРИЯ НА КОДИРАНЕТО

Д И С Е Р Т А Ц И Я
за придобиване на образователна и научна степен "доктор"

Професионално направление:
4.6 Информатика и компютърни науки
Научна специалност:
Информатика

Научен ръководител:
проф. дмн Илия Буюклиев

Велико Търново
2024

Съдържание

Увод

В текущата работа са разгледани подходи за оптимизация и паралелизация на алгоритми от теория на кодирането. Един от основните въпроси, засегнати тук, е свързан с избора на метод за паралелизация в зависимост от задачата. Задачите, които се решават, са важни както в теоретичен, така и в практически аспект. В практическо отношение теория на кодирането е свързана с комуникационни и компютърни технологии, отнасящи се до защита на информацията, компресиране, архивиране, автентикация и съхранение на данните, хеширане, блокчейн и други [24, 25, 52].

Голяма част от изследваните проблеми изискват използването на паралелни изчисления за тяхното решаване [11, 12, 35, 77]. За целта е необходимо добро познаване на различните технологии и интерфейси, използвани за паралелни изчисления, за да бъде възможно оптималното им прилагане. Съвременните изчислителни технологии се развиват в много направления. Едно от основните е разработването на многопроцесорни и многоядрени архитектури, които позволяват извършване на изчисления едновременно. Друго направление за развитие на изчислителната техника е в интегриране на архитектури, предназначени за специфични задачи в системи за общи изчисления. Такива са ускорителите, които се характеризират с голям брой изчислителни единици [49, 57]. Системите, които в допълнение към централен процесор със стандартна архитектура използват и ускорител, се наричат хетерогенни. В следствие на това развитие, паралелните изчислителни системи стават все по-достъпни. В последните десетилетия компютърните архитектури се развиват, като се добавят повече ядра (изчислителни единици) за един процесор и повече процесори за дадена система. Във всяко изчислително ядро на централните процесори също така се добавят и разширени векторни регистри. Всяка от представените характеристики на паралелните архитектури е обект на самостоятелно и обстойно изучаване. В текущата работа по-обстойно ще бъде коментиран специфичен подход за паралелизация, който използва тези разширени регистри.

Такъв подход се нарича векторизация. Тя се изразява в изпълнение на операции върху множество от елементи едновременно. Разработени са регистри с различни дължини, кратни на 128, като за работа с тях са създадени и допълнителни инструкции за централните процесори. Векторизацията чрез разширени регистри може да бъде извършена автоматично от компилатора или като се използват специални функции, разработени за езиците C/C++. Тези функции заедно с нови дефинирани типове от данни са включени в библиотеки и позволяват лесното използване на разширените регистри без изрично познание на асемблерските инструкции. Този подход също така дава възможност за използването на други паралелни интерфейси в комбинация с векторизацията, като така предоставя допълнително ускорение.

Нека с $\mathbb{F}_q$ се означи крайното поле с q елемента и с $\mathbb{F}_q^n$ - n-мерното векторно пространство над полето. *Разтояние по Хеминг* между два вектора $x = (x_1, \ldots, x_n)$ и $y = (y_1, \ldots, y_n)$ се нарича броят на координатите, в които те се различават. Означава се с $d(x, y) = |\{i | x_i \neq y_i\}|$. *Тегло по Хеминг* на вектор $x = (x_1, \ldots, x_n)$ се нарича броят на ненулевите координати на вектора. Означава се с $wt(x) = |\{i | x_i \neq 0\}|$. *Линеен код* с дължина n, размерност k над крайно поле с q елемента се нарича всяко k-мерно подпространство на n-мерното векторно пространство. Елементите на кода се наричат *кодови думи*, а параметрите n и k се наричат съответно *дължина* и *размерност* на кода. Матрица с k реда и n стълба, чиито редове образуват базис на кода C, се нарича *пораждаща матрица* за кода. Един основен параметър на линеен код е неговото *минимално разстояние* $d(C)$, което е най-малкото от всички разстояния между две кодови думи. *Минималното тегло* на линеен код се нарича най-малкото от всички ненулеви тегла в кода. За линеен код C минималното тегло и минималното разстояние съвпадат. Линеен код C с дължина n, размерност k и минимално разстояние d се означава с $[n, k, d]$. Редицата $(A_0, A_1, \ldots, A_n)$, където A_i е броят на кодовите думи с тегло i, се нарича *тегловен спектър* на линейния код. Два линейни $[n, k, d]$ кода над поле F_q се наричат еквивалентни, ако всички кодови думи на единия код, могат да се получат от кодовите думи на другия чрез последователност от следните трансформации:

- пермутация на координатите;
- умножение на елементите в дадена координата с ненулев елемент на F_q;
- прилагане на автоморфизъм на полето към елементите във всички координатни позиции.

Релацията еквивалентност на кодове разбива множеството от всички ко-

дове с еднакви параметри на класове на еквивалентност.

Когато линейните кодове биват разглеждани като комбинаторни структури, могат да бъдат обособени две основни задачи - конструиране и класификация. Задачата за конструиране може да се разгледа като построяване на пораждаща матрица за зададени параметри n, k, q и d. Под класифициране се разбира намиране на броя на класовете на еквивалентност и представител на всеки клас. Решаването на тези задачи включва много и различни алгоритмични проблеми, като намиране на тегловни инварианти (тегловен спектър, кодови думи с фиксирано тегло и др.) на линеен код при зададена пораждаща матрица. Основната цел на дисертацията е разработване на оптимизирани алгоритми за изчисление на тегловни инварианти на линейни кодове, които са базирани на тегловното разпределение на кода. Те са част от много алгоритми за решаване на задачите за генериране и класификация на кодове. За целта е разработена оптимизирана библиотека **LinCodeWeightInv**, като е използвана векторизация. Библиотеката включва интерфейсна част, модул за тестване и верификация и пълно описание на включените функционалности. За бързодействието на даден софтуер, библиотека и имплементация на алгоритми е важен както избрания подход за паралелизация, така и избора на компилатор.

При векторизация, бързодействието, което се постига, зависи от следните основни фактори:

- използваните инструкции - инструкциите за централните процесори условно могат да бъдат разглеждани като леки (побитови операции, сравнение, събиране и др.) и тежки (съчетаване, пермутации и др.). Тежките инструкции използват по-голям изчислителен ресурс от леките, като при работа с регистри могат да съдържат в своята имплементация изпълнението на няколко хардуерни инструкции. Тази категоризация зависи от хардуерната имплементация.

- коефициент на векторизация - показва максималния брой на координатите на даден вектор, които могат да бъдат записани в даден регистър [6]. Този коефициент зависи от големината на регистъра и начина на представяне на елементите в паметта.

- коефициент на използване - показва каква част от регистъра е използвана при изчисленията. Стойността му е между 0 и 1, като при 1 се използва целият регистър, при 1/2 - половината, а при 0 не се използва регистър. Този коефициент зависи от представянето на елементите и дължината на кода, за който се извършват изчисленията.

Тези параметри се използват при анализирането на ефективността на реализацията на представените алгоритми.

В текущата работа са разгледани следните основни цели, задачи и начините за решаването им:

- Векторизирана реализация на алгоритми за намиране на тегловен спектър на линеен код над полета с до 64 елемента, включително. За имплементацията са разработени два основни типа от алгоритми - оптимизирани алгоритми от високо ниво, които описват подхода за генериране на нова кодова дума и векторизирани алгоритми от ниско ниво, които реализират операциите събиране на вектори и намиране на теглото на вектор. Основната оптимизация в алгоритмите от високо ниво се състои в генерирането само на непропорционални кодови думи, като всяка нова кодова дума се получава единствено чрез събиране на вектори. Алгоритмите от ниско ниво включват в себе си различни реализации на функции за събиране на вектори и намирането на теглото на вектор, като се използват специални инструкции и разширени векторни регистри. Разработените функции се различават в зависимост от броя на елементите в полето и избрания набор от разширени инструкции за централния процесор. Анализирана е ефективността на имплементираните алгоритми в x86 архитектури, като се използват регистри с дължина 128 и 256 бита и съответните инструкции, предназначени за работа с тях. Имплементациите са сравнени с функции за намирането на тегловното разпределение в широко използваните пакети за компютърна алгебра Magma и GAP. Също така е анализирано влиянието на компилатора при използване на директна векторизация.

- Разширяване на основните алгоритми от ниско ниво за работа над прости полета с по-малко от 128 елемента. Разгледано е представяне на елементите на полето в паметта, като се използва беззнаков тип от данни с големина 8 бита. Това представяне позволява изчисленията да бъдат извършени над по-големи прости полета, без да се излиза от обхвата на стойности за дадения тип от данни. Това е възможно, тъй като се използва единствено операцията събиране на вектори и наличието на функции със сатурация. Функциите със сатурация за разширените векторни регистри позволяват при излизане от обхавата на дадения тип от данни да бъде записана валидна стойност (минималната или максималната за дадения тип в зависимост от резултата) вместо изрязване на част от стойността или грешно интерпретиране на побитовото представяне на резултата. Разработените

алгоритми са имплементирани чрез разширените набори от инструкции SSE4.1 и AVX512 за x86 архитектури и NEON набор от регистри за ARM архитектури. За реализиране на алгоритмите са изучени основните характеристики на инструкциите от вида AVX512 и NEON. Анализирана е ефективността на различните реализации, като е използван алгоритъм за намиране на тегловен спектър на линеен код.

- Анализиране на връзките между фамилии от кодове с две и три тегла, свързани със самодопълнителни кодове, които достигат границата на Грей-Ранкин. За целта са дефинирани четири фамилии от кодове с две тегла и две фамилии от кодове с три тегла. Описани са връзките между дефинираните фамилии, като са представени конструкции за получаване на кодове от различните фамилии при даден код от коя да е от останалите фамилии от кодове. Описани са и връзките им със самодопълнителни кодове, достигащи границата на Грей-Ранкин. Представени са конструкции за построяване на кодове с две тегла от описаните фамилии с размерност $k+2$ при зададен код от съответна фамилия с размерност k. Тези връзки между кодовете са използвани за конструиране на самодопълнителни кодове с параметри $[120, 9, \{56; 64; 120\}]$.

- Разработване на оптимизирана и преносима библиотека за намиране на тегловни инварианти на линейни кодове над полета $\mathbb{F}_q$, където $q \leq 64$. Библиотеката включва шест основни интерфейсни функции, за които са разработени три различни начина за въвеждане на данните. Създадени са два основни модула за използване на библиотеката - интерфейсен модул и модул за тестване и верификация. Разработената библиотека работи за x86 и ARM архитектури и различни набори от регистри. За имплементирането на библиотеката са изучени различни платформи, които се използват за създаване на софтуер и изготвяне на пълна документация.

В Глава 1 са представени базови понятия свързани с линейни кодове и паралелизацията на алгоритми. В Раздел 1.1 са дадени основни дефиниции и теореми, свързани с линейни кодове, тегловните им характеристики и самодопълнителни кодове, достигащи границата на Грей-Ранкин. Раздел 1.2 са разгледани основни подходи за паралелизация на алгоритми. Особено внимание е отделено на паралелизацията чрез векторизация, тъй като е основния подход, използван за паралелизиране на представените алгоритми.

В Глава 2 са разгледани основните алгоритми за намиране на тегловен спектър на линеен код. Известно е, че задачата за изчислението му е NP-пълна [10]. Има фамилии кодове, чиито тегловен спектър е известен, например фа-

милиите кодове на Хеминг, Голей и Рийд-Соломон. За някои класове от кодове могат да бъдат използвани специализирани алгоритми и техники [9, 43, 82]. В общият случай се генерират всички непропорционални кодови думи на кода [17, 46]. В тази глава са разгледани два основни вида алгоритми, използвани за намирането на тегловен спектър на линеен код - алгоритми от високо ниво, които представят подхода за генериране на нова кодова дума и алгоритми от ниско ниво, които извършват основните изчисления. Алгоритмите от високо ниво са базирани на подход за емулация на вложени цикли, представен в [17]. Представените алгоритми от ниско ниво са разработени като е използвана директна векторизация на подходите за събиране на вектори и намирането на теглото на вектор. Оптималното имплементиране на векторизация се базира на подходящо представяне не елементите на полето в паметта. Един от широко изучаваните и популярни подходи за оптимално записване на елементи на крайно поле в паметта е побитовото представяне [1, 16, 30, 48, 56]. Алгоритмите от ниско ниво използват побитово представяне за полетата $\mathbb{F}_2, \mathbb{F}_4, \mathbb{F}_{3^m}$, където $m = 1, 2, 3$. Побитовото представяне на елементите на полетата $\mathbb{F}_{3^m}$ е базирано на [30]. Имплементациите на алгоритмите са представени чрез разширени векторни регистри за x86 архитектури. Разгледана е ефективността на алгоритмите, като разработените функции са сравнени с популярната система за компютърна алгебра Magma [13] и пакета Guava за системата GAP [31], в които съществуват функции за намиране на тегловен спектър на линеен код. Анализирана е ефективността на векторизираните алгоритми като е направено сравнение със скаларна версия на базовия алгоритъм от високо ниво, реализиран чрез таблици за събиране и умножение. Известно е, че изборът на компилатор също може да повлияе на бързодействието на алгоритмите [6, 14]. За това е извършен анализ на ефективността на различни компилатори при използването на векторизация в представените алгоритми.

Глава 3 представя подход за оптимизиране на алгоритми за събиране на вектори чрез беззнакови типове от данни. Тези типове от данни позволяват операцията събиране на вектори над крайни полета да бъде имплементирана за прости полета с до 128 елемента без да бъдат използвани допълнителни ресурси за представяне на елементите на полето. За целта се използват и специализирани инструкции чрез сатурация, налични за някои операции с регистри [70]. Подробно са разгледани наборите от регистри и инструкции AVX512 за x86 архитектури и NEON за ARM архитектури [7, 51, 64]. Представени са някои от техните особености, които са от значение за имплементациите на описаните алгоритми. Тези инструкции могат да бъдат използвани за подобряване на времето за работа на различни алгоритми и софтуерни продукти [32, 47, 58, 64, 74].

Въпросът за ефективността на разширените инструкции AVX512 в сравнение с останалите разширени регистри при x86 архитектурите и при използването им в програми в комбинация с многонишкови и многопроцесорни изчисления също е обект на изследване [27, 83]. Тук е анализирана ефективността на AVX512 в сравнение с реализации на описаните алгоритми с инструкции SSE4.1, AVX2 за x86 архитектури. Също така е анализирана ефективността на NEON инструкциите. За целта са извършени експериментални изчисления за кодове над избрани фиксирани полета, дължини между 100 и 3000 и различни размерности.

В Глава 4 са разгледани двоични линейни самодопълнителни кодове, достигащи границата на Грей-Ранкин. Известно е, че броят на тези кодове нараства експоненциално [53]. Параметрите, за които съществуват, са дадени в [62], като те са класифицирани за $k \leq 7$. Интересът към изучаването на тези кодове е породен от връзките им с други комбинаторни структури като квазисиметрични SDP дизайни [45, 53], силно регулярни графи [20, 22, 33], бент [26, 34] и векторни бент функции [36] и други. В тази глава са дефинирани 6 фамилии от двоични линейни кодове с две и три тегла, свързани с двоични линейни самодопълнителни кодове, достигащи границата на Грей-Ранкин. Целта на изучаването на тези фамилии от кодове е използването на описаните връзки и конструкции за класифициране на самодопълнителни кодове, достигащи границата на Грей-Ранкин с размерност 9. За целта са изучени връзките между отделните фамилии от кодове. Представени са някои основни конструкции за построяване на кодове от различните фамилии при зададен код от някоя от тях. Разгледана е конструкция за получаване на кодове с по-висока размерност $k + 2$ чрез дадени кодове с размерност k за дефинираните фамилии от кодове с две тегла. Изчислителните резултати представят частична класификация за фамилиите от кодове с размерност 8 и самодопълнителни кодове, достигащи границата на Грей-Ранкин с размерност 9.

В Глава 5 е представена библиотеката за намиране на тегловни инварианти **LinCodeWeightInv**. Съществуват няколко оптимизирани библиотеки за изчисления с вектори, сред които NTL (A Library for doing Number Theory) [75] и VCL (Vector Class Library)[42]. Библиотеката NTL е разработена основно за извършване на бързи изчисления с полиноми. Библиотеката VCL дефинира класове за работа с вектори, като операциите се извършват чрез векторизация и разширени векторни регистри. Двете библиотеки имат своите предимства, но не предоставят функционалност за извършване на изчисления с вектори над крайни полета чрез векторизация. От друга страна, системите за компютърна алгебра като Magma и GAP не са разработени с цел изучаване на линей-

ни кодове. Това ги прави трудни за използване в задачите за конструиране и класификация. За решаването на тези задачи могат да бъдат използвани модулите *Generation* и *LCequivalence* на програмата *QextNewEdition* [15]. Целта на представената библиотека е тя да бъде използвана като допълнение при разработването на алгоритми и програми за решаването на основните задачи в теория на кодирането, свързани с линейни кодове. В главата са описани основните функционалности и начините за използване на библиотеката. Подробно са коментирани интерфейсните функции и модула за тестване и верификация. Представени са основни проблеми при разработване на преносим софтуер и използваните софтуерни продукти за тяхното преодоляване. Представена е структурата на библиотеката и разработените начини за компилиране, инсталиране и използване във външни проекти.

Научно-приложни приноси

- Разработване на алгоритъм за генериране на непропорционални кодови думи, който работи за съставни полета само чрез събиране на два вектора.
- Разработване на алгоритъм за събиране на вектори чрез SSE, AVX и AVX512 инструкции при използване на побитово представяне на елементите на полета $\mathbb{F}_2$, $\mathbb{F}_4$ и полетата с характеристика 3.
- Разработване на алгоритъм за събиране на вектори чрез SSE, AVX и AVX512 инструкции при байтово представяне за прости и съставни полета с до 64 елемента, включително, и намиране на теглото на вектор чрез едно извикване на *popcnt* инструкция.
- Анализ на ефективността на работата на различни компилатори с SSE и AVX инструкции при векторизация.
- Изучаване на характеристиките на наборите от инструкции AVX512 и NEON и анализиране на тяхната ефективност.
- Анализиране на ефективността на различните видове инструкции в архитектури от вида x86.
- Разработване на алгоритъм за събиране на вектори чрез SSE, AVX и AVX512 инструкции при представяне на елементите на полета с до 128 елемента чрез беззнакови цели числа.
- Дефиниране на фамилии от кодове с две и три тегла, свързани с кодове, достигащи границата на Грей-Ранкин, и описание на връзките между отделните фамилии.
- Разработване на конструкция за построяване на кодове от дадена фамилия с размерност k чрез кодове от съответна фамилия с размерност $k+2$.
- Разработване на математически софтуер (библиотека LinCodeWeightInv)

за намиране на теглови инварианти на линейни кодове над полета с до 64 елемента, включително, като се използват SSE4.1, AVX2 и AVX512 инструкции за x86 архитектури и NEON инструкции за ARM архитектури.

- Представяне на особености и основните акценти при създаването на математически софтуер с отворен код.

Апробация на резултатите

Резултатите, включени в дисертацията, са получени самостоятелно [P2, P3] и в съавторство с:

- Буюклиев [P1, P5, P6]
- Буюклиев и Буюклиева [P4]

Включените статиите са публикувани или изпратени за рецензия в:

- *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]

Резултати по дисертацията са докладвани на:

- Конференция с международно участие *"Иновативно STEM образование"* 2021-2022 г. [D1, D3]
- Национален семинар по теория на кодирането *Проф. Стефан Додунеков*, 2021-2022 г. [D2, D5]
- International Conference Automatics and Informatics, Varna, Bulgaria, 2022 г.[D4]
- 4-th Interdisciplinary PhD Forum with International Participation, Sandanski, Bulgaria, 2023 г. [D6]
- 14th International Conference on Large-Scale Scientific Computations, Sozopol, Bulgaria, 2023 г. [D7]

- Семинар *"HPC for Mathematics and Applications"*, София, България, 2023 г. [D8]
- International Conference *"Cryptography and Coding Theory"*, Perugia, Italy, 2023 г. [D9]

Благодарности

Благодаря на всички съавтори за насоките и чудесния работен процес. Благодаря на колегите от секция "Математически основи на информатиката"към Института по математика и информатика (ИМИ) на Българската академия на науките (БАН) за създадената работна атмосфера, съветите и конструктивната критика. Получената обратна връзка през годините несъмнено допринесе за цялостната стойност на изследванията. Също така бих искала да изразя своята признателност към ръководството и служителите на ИМИ-БАН за предоставените възможности, подкрепа и достъп до изчислителните ресурси. Благодаря и на колегите от Факултета по математика и информатика към Великотърновския университет за възможността да се запозная и с друг начин за усвояване на знания - чрез преподаване.

Специални благодарности отправям към моя научен ръководител проф. д.м.н. Илия Буюклиев за търпението, подкрепата, конструктивните съвети и интересните теми, с които ме запозна при работата върху дисертационния труд. Съветите и придобитите умения ще ми помагат и за напред в развитието ми както в академично отношение, така и като личност.

Накрая бих искала да изразя признателността си към моя съпруг за търпението и подкрепата, които оказа през академичния ми път и във всяко едно отношение. Също така му благодаря, че е мой неотлъчен спътник във всички начинания.

Глава 1

Основни понятия в теория на кодирането. Оптимизация и паралелизация на алгоритми

В тази глава са представени някои основни дефиниции и твърдения в теория на кодирането, както и архитектури и подходи за паралелизация на алгоритми. В раздел 1.1 са дадени основни понятия, свързани с линейни кодове. В изложението се следват монографиите [50, 60, 65]. Представени са и основните понятия, твърдения и граници, свързани с двоични самодопълнителни линейни кодове, достигащи граница на Грей-Ранкин. В раздел 1.2 са разгледани някои основни паралелни архитектури.

1.1 Основни понятия

1.1.1 Линейни кодове

Нека разгледаме крайно поле $\mathbb{F}_q$ с q елемента. С $\mathbb{F}_q^n$ бележим n-мерното векторно пространство над полето $\mathbb{F}_q$.

Дефиниция 1.1. *Разстояние (по Хеминг) между два вектора* $x = (x_1, x_2, \ldots, x_n)$ *и* $y = (y_1, y_2, \ldots, y_n)$, $x, y, \in \mathbb{F}_q^n$ *се нарича броя на координатите, в които те се различават. Означава се с* $d(x, y) = |\{i | x_i \neq y_i\}|$.

Разстоянието по Хеминг задава метрика в $\mathbb{F}_q^n$. За $x, y, z \in \mathbb{F}_q^n$ имаме:

$$d(x,y) = 0 \iff x = y$$

$$d(x,y) = d(y,x)$$

$$d(x,y) \leq d(x,z) + d(z,y)$$

Дефиниция 1.2. *Тегло (по Хеминг) на вектор* $x = (x_1, x_2, \ldots, x_n) \in \mathbb{F}_q^n$ *се нарича броя на ненулевите му координати. Означава се с* $wt(x) = |\{i | x_i \neq 0\}|$.

В сила са следните връзки между тегло и разстояние по Хеминг:

$$wt(x) = d(x, \mathbf{0}),$$

$$d(x,y) = wt(x - y),$$

където $\mathbf{0} = (0, 0, \ldots, 0)$ и $\mathbf{0}$, $x, y \in \mathbb{F}_q^n$. Всяко непразно подмножество C на $\mathbb{F}_q^n$ се нарича q-ичен код с дължина n. Елементите на C се наричат кодови думи. Кодове над $\mathbb{F}_2$ се наричат двоични.

Дефиниция 1.3. *Линеен код* C *над* $\mathbb{F}_q$ *се нарича всяко* k*-мерно подпространство на векторното пространство* $\mathbb{F}_q^n$*, където* k *се нарича размерност на кода, а* n *- дължина. Казваме, че* C *е* q*-ичен* $[n,k]$ *код или* $[n,k]_q$ *код. Подкод на* C *се нарича всяко линейно подпространство на кода. Обем на кода се нарича броя на кодовите думи и се бележи с* $|C|$.

Дефиниция 1.4. *Минимално разстояние на кода* C *се нарича най-малкото от разстоянията между две различни кодови думи. Означава се с* $d(C) = min\{d(x,y) | x \neq y, x, y \in C\}$.

Дефиниция 1.5. *Минимално тегло на кода* C *се нарича най-малкото от всички ненулеви тегла на кодови думи в кода.*

Линеен код C с минимално разстояние $d(C) = d$ се нарича q-ичен $[n,k,d]$ код или $[n,k,d]_q$ код. За линейни кодове минималното разстояние и минималното тегло съвпадат:

$$d(C) = min\{d(x,y) | x \neq y, x, y \in C\} = min\{wt(x) | x \in C, x \neq 0\}$$

Един линеен $[n,k,d]_q$ код може да се използва за откриване на $(d-1)$ грешки и коригиране на $\lfloor (d-1)/2 \rfloor$ грешки. Един код се нарича δ-делим, ако всички негови тегла се делят на δ. Ако $\delta = 2$, то кода се нарича четнотегловен, а ако $\delta = 4$, то кода се нарича двойночетен.

Дефиниция 1.6. *Пораждаща матрица на линеен $[n,k]_q$ код се нарича всяка $k \times n$ матрица G с елементи в $\mathbb{F}_q$, чиито редове образуват базис на кода. Казваме, че G е в систематичен вид, ако $G = (I_k|A)$, където I_k е единичната матрица от ред k.*

Нека с $(x,y) = \sum_{i=1}^{n} x_i y_i$ бележим евклидово скаларно произведение на векторите $x, y \in F_q^n$.

Дефиниция 1.7. *Два вектора наричаме ортогонални, ако тяхното евклидово скаларно произведение е 0.*

Дефиниция 1.8. *Ортогонален (дуален) код на C се нарича неговото ортогонално допълнение $C^{\perp} = \{u \in F_q^n | (u,v) = 0, \forall v \in F_q^n\}$. Минималното разстояние на кода $C^{\perp}$ се нарича дуално разстояние на кода C. Означава се с $d^{\perp}$.*

Дефиниция 1.9. *Един линеен код се нарича самоотогонален, ако $C \subseteq C^{\perp}$ и самодуален, ако $C = C^{\perp}$.*

Дуалния код на даден $[n,k,d]$ код C е с параметри $[n, n-k, d^{\perp}]$. Пораждащата матрица на дуалния код се нарича проверочна матрица за C и се означава с H. Ако пораждащата матрица G на кода е в систематичен вид ($G = (I_k|A)$), то проверочната матрица е от вида $H = (-A^T|I_{n-k})$. Дуалното разстояние на линеен код C дава информация за координатите на кода. Ако $d^{\perp} = 1$, то кода има поне една нулева координата. След премахване на нулевите координати на кода C получаваме код със същата размерност, кодови думи със същите тегла и по-малка дължина, наречена *ефективна дължина* на кода. Ако $d^{\perp} > 1$, то ефективната дължина на кода е n. Ако $d^{\perp} = 2$, то кода има пропорционални координати. Ако $d^{\perp} > 2$, кода се нарича *проективен*.

Дефиниция 1.10. *Остатъчен код на даден код C по отношение на кодова дума $x \in C$ се нарича код $Res(C,x)$, който се получава от рестрикцията на C върху нулевите координати на кодовата дума x.*

Теорема 1.1. *[50] Нека C е двоичен $[n,k,d]$ код с дуално разстояние $d^{\perp}$ и $c \in C$ е с тегло w, където $w < 2d$. Тогава остатъчния код $Res(C,c)$ е $[n-w, k-1, d']$ код, където $d' \geq d - w + \lceil w/2 \rceil$.*

Лема 1.1. *Нека C е двоичен линеен код, който е 2^m-делим, където m е естествено число и $m > 2$. Тогава остатъчният код $Res(C,c)$ относно кодова дума c е 2^{m-1}-делим.*

Доказателство. Нека c е във вида $c = (\underbrace{11\ldots1}_{w}\underbrace{00\ldots0}_{n-w})$, където $w = \mathrm{wt}(c)$ и n е дължината на кода. Тогава за всяка кодова дума $v_2 \in Res(C, c)$ съществува вектор $v_1 \in \mathbb{F}_2^w$, такъв че $v = (v_1, v_2) \in C$. Тогава $c + v = (\mathbf{1} + v_1, v_2) \in C$ е кодова дума с тегло $w - \mathrm{wt}(v_1) + \mathrm{wt}(v_2)$. Тъй като 2^m дели w, $\mathrm{wt}(v_1) + \mathrm{wt}(v_2)$ и $w - \mathrm{wt}(v_1) + \mathrm{wt}(v_2)$, тогава 2^{m-1} дели $\mathrm{wt}(v_2)$. Следователно $Res(C, c)$ се дели на 2^{m-1}. □

Дефиниция 1.11. *Два линейни $[n, k, d]_q$ кода наричаме еквивалентни, ако всички кодови думи на единия код могат да се получат от кодовите думи на другия чрез последователност от следните трансформации:*

- *пермутация на координатите;*
- *умножение на елементите в дадена координата с ненулев елемент на $\mathbb{F}_q$;*
- *прилагане на автоморфизъм на полето към елементите във всички координатни позиции.*

Автоморфизъм на линеен код се нарича всяка последователност от трансформации, дадени в Дефиниция 1.11, която изобразява всяка кодова дума на C в кодова дума на същия код. Множеството от всички автоморфизми на C образува група, която се нарича група от автоморфизми на кода и се бележи с $Aut(C)$. Релацията на еквивалентност разделя множеството на всички кодове с дадени параметри на класове на еквивалентност. Задачата за класифициране на линейни кодове се състои в намирането на представител на всеки клас на еквивалентност. В някои алгоритми за конструиране при решаването на задачата за класификация на линейни кодове се използват остатъчни такива за генерирането на нови кодове с желани параметри [4, 55].

1.1.2 Тегловни характеристики

Дефиниция 1.12. *Тегловен спектър на линеен $[n, k, d]_q$ код се нарича наредената $(n+1)$-орка $(A_0, A_1, \ldots, A_n)$, където A_i е броят на кодовите думи с тегло i, $i = 0, 1, \ldots, n$. Полиномът $W_C(x) = \sum_{i=0}^{n} A_i x^i$ се нарича тегловна функция на кода.*

За всеки линеен $[n, k, d]$ код е в сила $A_0 = 1$ и $A_1 = A_2 = \cdots = A_{d-1} = 0$. Също така се вижда, че $A_0 + A_1 + \cdots + A_n = q^k$. Ако два линейни кода са еквивалентни, то техните спектри са равни, като обратното не е в сила. Теорема

1.2 дава връзка между тегловния спектър на линеен $[n, k, d]_q$ код C и неговият дуален код $C^\perp$.

Теорема 1.2 (Тъждества на MacWilliams). *[60]Нека C е линеен $[n, k, d]_q$ код, $C^\perp$ е неговият ортогонален, а $(A_0, A_1, \ldots, A_n)$ и $(B_0, B_1, \ldots, B_n)$ са съответно спектрите на C и $C^\perp$. Тогава:*

$$B_j = \frac{1}{|C|} \sum_{i=0}^{n} A_i K_j(i, n),$$

където

$$K_j(x, n) = \sum_{i=0}^{j} (-1)^i (q-1)^{j-i} \binom{x}{i} \binom{n-x}{j-i}$$

са полиномите на Кравчук.

Сред най-важните тегловни характеристики на един линеен код са минималното му разстояние и тегловният му спектър. Броя на грешките, които кода може да открие и поправи, зависи от минималното разстояние на кода, докато тегловният спектър дава информация за вероятността за откриване и поправяне на грешки [38, 59]. Някои от методите и алгоритмите за генериране и класификация на линейни кодове използват целия или частичен спектър на кода. Еквивалентни кодове имат равни минимални разстояния и тегловни спектри, тъй като трансформациите, описани в Дефиниция 1.11, запазват теглото по Хеминг. Това прави задачата за намиране на тегловен спектър на линеен код основна в теория на кодирането. Доказано е, че задачата е NP-пълна [10]. За някои фамилии от кодове (кодове на Хеминг, Голей, Рид-Соломон), тегловният спектър е известен.

Различни подходи могат да се използват при изчислението на тегловния спектър на някои специални типове кодове като БЧХ [43], циклични кодове [9], поляризирани кодове [82] и други. Специфичен подход за намирането на тегловното разпределение на код на Рид-Малер е представен в [5]. В общия случай е необходимо да се намери теглото на всички непропорционални кодови думи. За решаването на проблема в общия случай са разработени различни подходи [17, 19, 46]. Основният подход за решаването на задачата, представен в [46], се базира на използването на q-ичен код на Грей за генериране на всички кодови думи. За генериране само на непропорционални кодови думи може да се използва модификация на q-ичен код на Грей, представена в [54]. Алгоритъмът, представен в [19], използва характеристичен вектор на кода и дискретна трансформация за намиране на тегловното разпределение. При този метод не се

генерират кодови думи. Той е подходящ за линейни кодове с голяма дължина и малка размерност. Разработена е паралелна реализация за двоични линейни кодове, като се използват ускорители, която е представена в [68]. В [17] е представен метод за намиране на броя на кодовите думи с тегло не по-голямо от дадено естествено число m. Задачата за намиране на тегловен спектър може да се разглежда като частен случай, където $m = k$ за линеен $[n, k]_q$ код (търсят се кодови думи с тегло не по-голямо от k).

1.1.3 Самодопълнителни кодове

Дефиниция 1.13. *Двоичен линеен $[n, k]$ код C се нарича самодопълнителен, ако за всяка кодова дума $x \in C$, нейното допълнение $\overline{x}$, където $x + \overline{x} = \mathbf{1}$, също е кодова дума ($\overline{x} \in C$).*

За обема на двоичен линеен $[n, k, d]$ самодопълнителен код е в сила границата на Грей-Ранкин, дадена чрез следното неравенство, ако дясната страна е положително число:

$$|C| \leq \frac{8d(n-d)}{n-(n-2d)^2}. \tag{1.1}$$

Тази граница е в сила за линейни и нелинейни двоични самодопълнителни кодове. Самодопълнителните кодове, които са разгледани в тази работа са двоични линейни кодове. Параметрите на линейни кодове, които достигат равенство при границата на Грей-Ранкин, са

$$\begin{gathered}[2^{2m-1} - 2^{m-1}, 2m+1, 2^{2m-2} - 2^{m-1}], \\ [2^{2m-1} + 2^{m-1}, 2m+1, 2^{2m-2}].\end{gathered} \tag{1.2}$$

Самодопълнителни кодове, които достигат границата на Грей-Ранкин, са широко изучавани. От особен интерес са техните подкодове, поради връзките им със силно регулярни графи [20, 22, 33], квазисиметрични SDP дизайни [45, 53], бент [26, 34] и векторни бент функции [36], четни двоични кодове и други. Връзката между четнотегловни двоични линейни кодове е представена в Лема 1.2. Известно е, че всички самоортогонални кодове са четнотегловни.

Лема 1.2. *Нека C е четнотегловен двоичен линеен код с пораждаща матрица от вида $G = (I_k|A)$. Дуалният код на C е самодопълнителен.*

Доказателство. Тъй като C е четнотегловен, всеки ред на матрицата A има нечетно тегло. Тогава матрицата $H = (A^T|I_{n-k})$, която е пораждаща матрица

на дуалния код на C, има колони, чиито тегла са нечетни. Сумата от всички редове на H ще даде като резултат единичния вектор $\mathbf{1} = (1, \ldots, 1)$. □

Множеството от всички кодови думи с минимално тегло в двоичен линеен самодопълнителен код с четна дължина, достигащ равенство в (1.1), съставя множеството от блокове на квазисиметричен SDP дизайн [45, 53]. Тогава от нееквивалентните кодове могат да се получат неизоморфни SDP дизайни. Следователно броят на нееквивалентните кодове е равен на броя на неизоморфните SDP дизайни. Доказано е, че техният брой нараства експоненциално с нарастване на размерността [53].

1.2 Оптимизация и паралелизация на алгоритми

Задачите за конструиране, класификация, намиране на тегловен спектър в общия случай изискват голям изчислителен ресурс. Тогава оптимизацията и паралелизацията на алгоритмите са важни за решаването на дадените задачи в обозримо време. Под оптимизация може да се разбира както намаляване на изчислителната сложност на използвания алгоритъм, така и намаляване на използваните изчислителни ресурси и / или времето за изпълнение на програмната имплементация на алгоритъма. Често намаляването на използвания ресурс е за сметка на бързодействието и обратно. Повече за оптимизацията на програмен код на C/C++ може да се намери в [41]. Под паралелизация се разбира извършването на множество от изчисления едновременно. В този раздел са разгледани някои от основните техники за паралелизация на алгоритми и основни C/C++ интерфейси за работа с тях. В Раздел 1.2.2 подробно е описан подхода за паралелизация чрез векторизация и разширени векторни регистри, който е използван за реализация на алгоритмите в Глави 2 и 3.

1.2.1 Базови паралелни подходи

Паралелизация на даден алгоритъм може да се постигне по два основни начина - разделяне на задачата на подзадачи и разделяне на данните, така че изчисленията да се извършват едновременно за различни данни [40, 66, 81]. Основния проблем при първият подход е разделянето на задачата по подходящ начин, така че различните подзадачи да могат да се решават едновременно. Резултатът от решаването на избраните подзадачи трябва да дава крайния резултат на първоначалната задача, понякога след допълнителна обработка или обединение на получените резултати. Възможността за извършване на няколко

изчисления едновременно се предоставя от хардуерния ресурс. Съвременната изчислителна техника разполага с повече от една изчислителни единици, като тяхната архитектура може да бъде еднотипна (хомогенни системи) или с различни по вид изчислителни единици (хетерогенни системи). В литературата се разглеждат голям брой стратегии за паралелизация в зависимост от организация и разпределение на данните, начините за комуникация и други, като те могат да бъдат и архитектурно зависими. Но базисните подходи за оптимизация и паралелизация могат да се разгледат и като технологии, получени като подходящи разширения на езика C/C++ с необходимия допълнителен софтуер. Някои от базовите паралелни подходи и интерфейси за езика C/C++ са следните:

- Векторизация - основната идея е изчисления да се извършват за множество от елементи или променливи едновременно, за разлика от последователен алгоритъм, където операциите се извършават за един елемент/променлива. Векторизацията също така е известна като паралелизация от типа една инструкция, множество данни (Single Instruction, Multiple Data - SIMD) в таксономията на Флин [39]. Този тип паралелни изчисления могат да се извършат както на ниво инструкции (на централния процесор) така и на ниво изчислителна единица. Векторизацията е подходяща за задачи, в които тежката изчислителна част се състои в обработката на голямо количество от данни с еднотипни операции върху тях. Поради своята същност, при този тип паралелизация често се използват и последователни изчисления, изпълняващи допълнителна обработка на данните.

- Паралелни изчисления в системи със споделена памет - основната идея при този тип архитектури е извършването на паралелни изчисления само за тежката изчислителна част от алгоритъма. За разлика от векторизацията, при този модел е възможно отделните изчисления да се извършват върху еднотипни данни, така и да се изпълняват различни типове изчисления от отделните изчислителни единици. Друга разлика при този подход, е възможността изчисления, които се изпълняват върху еднотипни данни, да бъдат с по-голяма сложност. Тук пресмятанията се извършват от изчислителни единици, които имат достъп до една и съща физическа памет. Преди започване и след извършване на паралелните изчисления, алгоритъмът се изпълнява последователно от единствена изчислителна единица. За езиците C/C++ и Fortran е разработен паралелният интерфейс OpenMP. Повече информация за OpenMP и изчисленията в системи със споделена памет може да се намери в [61, 73].

- Паралелни изчисления в системи с разпределена памет - изчисленията при този модел са независими, като се изпълняват върху различни изчислителни единици, всяка от които разполага със свое адресно пространство. От хардуерна гледна точка, тези системи мога да бъдат както хомогенни, така и хетерогенни. Този подход е удачен при наличието на подходящо разделяне на задачата на подзадачи, които са независими една от друга. Основен проблем, който може да намали ефективността при използването на този подход, е извършването на честа комуникация между отделните изчислителни единици. За езиците C/C++ и Fortran е разработен интерфейс за предаване на съобщения (Message Passing Interface; MPI), който дефинира начина на работа и предаване на съобщения в системи с разпределена памет. Повече информация за MPI и изчисленията в системи с разпределена памет може да се намери в [73].

- Хетерогенни системи с ускорители - В тези системи основната идея за паралелизация е тежките изчисления да се извършат от ускорител, който разполага с голям броя изчислителни единици с различна архитектура. При този подход има последователна част от програмата или алгоритъма, която може да извършва подготвителни изчисления (като подготовка на данните), управление на ускорителя (контрол на паметта, инициализация и стартиране на изчисленията върху ускорителя и др.), последващи изчисления, които могат да бъдат необходими за получаване на крайния резултат. Изчислителните единици в ускорителите имат различна архитектура от стандартните централни процесори, като се характеризират с големия си брой. Често се използва разпаралелване спрямо данните от типа SIMD. Сред популярните интерфейси за работа в хетерогенни системи с ускорители на производителя NVidia е програмното разширение на езика C - Compute Unified Device Architecture (CUDA), поради лесната интеграция със C/C++ програмен код и широкото използване на графичните платки за изчисления с общо предназначение (general-purpose computing). Графични платки са използвани за различни задачи и проблеми в теория на кодирането [3, 11, 35, 69].

1.2.2 Векторизация

Векторизацията е сред най-подходящите подходи за паралелизация на алгоритми, свързани с изучаване на линейни кодове, тъй като основните изчисления са свързани с операции над вектори. Съвременното разбиране за векторизацията се различава от векторни процесори в хардуерната имплемен-

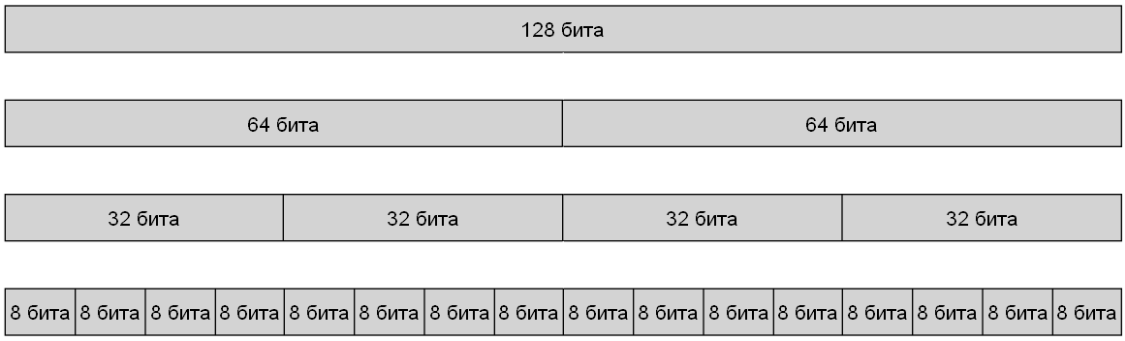
тация. Основният принцип на работа при векторните процесори е извършване на операции върху масиви (вектори) от данни, чиято дължина не е фиксирана. В съвременните процесори операциите се извършват върху скаларни величини (единични елементи). Векторизацията в такива процесори се базира на използване на разширени регистри, чиято дължина е кратна на дължината на една компютърна дума. Векторизация също така може да се постигне и в ускорителите, които разполагат с много по-голям брой изчислителни единици, което позволява множество от операции да се извършва върху по-голям брой елементи едновременно. Един основен недостатък при работата с ускорителите е необходимостта от копиране на памет между различни архитектури (от централен процесор към ускорител и обратно). Поради тази причина в текущата разработка се разглежда векторизация, имплементира чрез разширени регистри и съответни инструкции на централните процесори.

Съвременните централни процесори разполагат с набор от регистри с по-голяма дължина от стандартна компютърна дума. Към момента за x86 архитектури са разработени регистри с дължини от 128, 256 или 512 бита. Векторизацията чрез такива регистри идва от възможността няколко еднотипни елемента да бъдат записани в един разширен регистър с определена дължина. Операциите се изпълняват за всички елементи, записани в регистъра едновременно, поради наличието на множество аритметично-логически устройства (ALU), които работят паралелно. Това позволява изпълнението на операциите върху вектори да бъде толкова бързо, колкото стандартните инструкции върху регистри. За работа с разширените регистри са разработени допълнителни типове от данни и инструкции. За различните архитектури на централни процесори са разработени различни набори от инструкции и типове от данни. Сред най-широко разпространените набори от инструкции, които позволяват лесно използване на разширени регистри за научни цели (в частност за изучаване на линейни кодове), са архитектурите от вида x86, поради широкото им използване в системи за високопроизводителни изчисления. Разработени са следните типове инструкции: SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, AVX512. Инструкциите за 128-битови регистри на централните процесори с x86 архитектури са известни като Streaming SIMD Extensions (SSE), като по-нататък ще ги наричаме с общото наименование SSE инструкции. Съществуват и други инструкции за различни архитектури (напр. NEON инструкции за ARM архитектурите). Инструкциите за 256-битови регистри се наричат Advanced Vector Extensions (AVX), докато инструкциите за 512-битови регистри са известни с общото наименование AVX512. В този раздел са представени някои типове от данни и основни функции от набора от инструкции SSE. Основ-

ните принципи за векторизация са в сила и за останалите типове инструкции, като AVX512 и NEON инструкциите за ARM архитектурите са разгледани в Глава 3 поради тяхната специфика.

Както беше споменато по-горе в един разширен регистър могат да бъдат записани няколко еднотипни елемента от стандартен тип данни. Фигура 1.1 представя записването на елементи от основни типове данни (char, float, unsigned long long int) в 128-битов регистър. Типовете от данни и инструкции за x86 архитектури за езиците C/C++ са с точно определена структура на наименованията. Следните са типовете от данни, разработени за 128-битовите регистри, съдържащи съответно цели числа (char, int, long long int, и др.), реални числа (float) и реални числа с двойна точност (double): __m128i, __m128, __m128d. Аналогично са дефинирани типовете от данни за 256 и 512-битови регистри. Функциите, дефинирани в разширенията от тип SSE, имат следната структура: _mm_<име_на_функцита>_<тип_данни>, където <тип_данни> дава информация за това как да бъде интерпретирана информацията записана в регистрите, за които ще се изпълнява функцията. Възможности за стойността на <тип_данни> са следните:

Фигура 1.1: Представяне на регистър като масив от еднотипни елементи



- si128 – 128-битово цяло число със знак. Аналогични стойности се използват за останалите дължини на регистрите.
- epi8, epi32, epi64 — регистър, съдържащ 8-битови цели числа със знак, 32-битови цели числа със знак или 64-битови цели числа със знак. Броят на елементите от даден тип, които могат да бъдат записани в регистър зависи от дължината на регистъра: 128-битов регистър може да съдържа шестнадесет 8-битови числа, четири 32-битови числа и две 64-битови числа (със или без знак).

- epu8, epu32, epu64 — регистър, съдържащ 8-битови цели числа без знак, 32-битови числа без знак или 64-битови числа без знак. Беззнакови типове от данни и тяхното използване ще бъде разгледано по-подробно в Глава 3.
- ps — регистър, съдържащ пакетирани реални числа
- pd — регистър, съдържащ пакетирани реални числа с двойна точност
- ss — регистър, съдържащ едно реално число с плаваща запетая (използват се само 32 бита от регистъра)
- sd — регистър, съдържащ едно реално число с двойна точност (използват се само 64 бита от регистъра)

Разглежданите алгоритми в текущата работа извършват изчисления върху целочислени типове от данни, които позволяват подходящо представяне на елементите на полето. Функциите, които могат да се изпълняват върху регистри, условно могат да бъдат разглеждани като леки (побитови операции, събиране, изваждане, сравнение и др.) и тежки (умножение, пермутация) функции. Тежките функции използват по-голям изчислителен ресурс от леките, като при работа с регистри могат да съдържат в своята имплементация изпълнението на няколко хардуерни инструкции. В допълнение към типичните операции с вектори като събиране и изваждане, тези разширени инструкции предоставя много допълнителни възможности, които значително улесняват или подобряват ефективността в програмното изпълнение на някои алгоритми. Тук са описани някои от използваните функции за имплементация на представените алгоритми:

- Операции за сравнение - покомпонентно се сравняват съответните елементи на два регистъра за операции $=, >, <, \geq, \leq$. Резултата от сравнението се записва в регистър със същата дължина, като резултата за всеки елемент е 0 (резултата от сравнението за съответния елемент е лъжа) или F (резултата от сравнението за съответния елемент е истина), където под 0 и F се разбира елемент от съответната дължина в битове, съдържащ единствено нули или единици в своя двоичен запис.

  Пример 1.1. *Сравнение на два регистъра от 16-битови цели числа с операция* $>$*:*

  ```
  a = (120,0,30,260,-10,6,30,-100), b = (0,0,0,0,0,0,0,0)
  c = _mm_cmpgt_epi16(a,b) = (F,0,F,F,0,F,F,0)
  ```

- Операции за съчетаване на вектори - в резултатния регистър се записва координата, избрана между два дадени регистъра, за които се изпълнява функцията, спрямо стойността на съответната координата от трети регистър (маска).

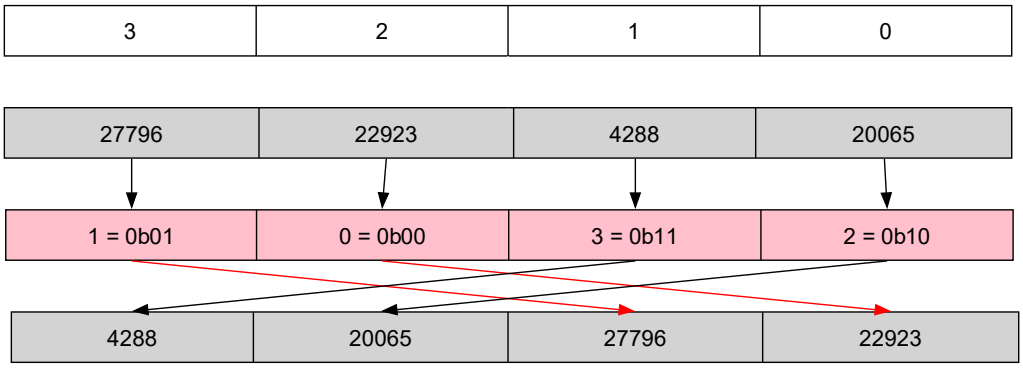
  Пример 1.2. *Съчетаване на два регистъра от 8-битови цели числа с операция* blendv*:*
  $a = (120, 0, 17, 8, 70, 60, 45, 90, 33, 11, 0, 6, 61, 80, 16, 5)$
  $b = (59, -61, -44, -53, 9, -1, -16, 29, -21, -50, -11, -55, 0, 19, -45, -56)$
  $c = \_mm\_blendv\_epi8(a, b, b) = (59, 0, 17, 8, 9, 60, 45, 29, 33, 11, 0, 6, 0, 19, 16, 5)$,
  където $c_i = \begin{cases} a_i & \text{ако } b_i < 0 \\ b_i & \text{ако } b_i \geq 0 \end{cases}$

- Операции за пермутация - в зависимост от двоичното представяне на контролно 8-битово цяло число, може да се извърши пермутация на елементите на даден регистър. В зависимост от типа на елементите в регистъра, контролната стойност може да се представи като масив от цели числа, записани в един или два бита. Регистърът може да се разгледа като масив от еднотипни елементи с подходяща дължина (в зависимост от големината на масива и типа на данните, записани в него). Елементите на така описания масив, получен от контролната стойност, показват в коя позиция трябва да се измести съответния елемент на регистъра. Фигура 1.2 показва изпълнението на функцията *_mm_shuffle_epi32*, която извършва разместване на 32-битовите елементи, записани в 128-битов регистър с контролна стойност 78 (побитово представяне: 01 00 11 10).

Намиране на броя на ненулевите битове в компютърна дума

Бързо намиране на ненулевите битове в компютърна дума или регистър е съществено за бързодействието на описаните в Глава 2 алгоритми. За целта съществуват различни алгоритмични и хардуерни подходи. Един подход е да се използват предварително изчислени тегла на двоичните вектори с определена дължина, както е описано в [2]. Алгоритъм за намирането на теглото на вектор, който използва маски, е представен в [30]. За съвременните архитектури е разработена хардуерна имплементация на функция, която намира броя на ненулеви битове в компютърна дума, известна като *population count* (*popcnt*). При работа с 64-битови компютърни думи, *popcnt* е по-бърза от изчисления чрез предварително намерени тегла на двоични вектори [2] и алгоритъма описан в [30]. В представените алгоритми в Глава 2 за намирането на тегло на

Фигура 1.2: пермутация на 32-битови цели числа от 128-битов регистър спрямо контролна стойност 78



вектор е необходимо намирането на броя на ненулеви битове в 64-битова компютърна дума, като в описанието им се използва именно *popcnt* функцията.

Глава 2

Основни алгоритми за изчисление на тегловен спектър на линейни кодове и тяхната векторизация

В тази глава са описани разработените алгоритми за намиране на тегловен спектър на линеен код. Представените алгоритми условно са разделени на алгоритми от високо ниво и алгоритми от ниско ниво. Алгоритмите от високо ниво, описани в Раздел 2.1, определят начина за генериране на следваща кодова дума като линейна комбинация на редовете на пораждащата матрица на кода. Основната оптимизация е в генерирането на нова кодова дума единствено чрез събиране на вектори. В Раздел 2.2 са представени два основни типа алгоритми - за събиране на вектори над крайно поле и за намиране на теглото на вектор над крайно поле. Тези алгоритми изпълняват тежката изчислителна част и именно за тях се използва паралелизация чрез векторизация и разширени регистри. В описанието на алгоритмите и примерите са използвани функции от SSE разширения набор от инструкции за x86 архитектури. В Раздел 2.3 са представени получените експериментални резултати за бързодействието на разработените алгоритми. Направено е сравнение между функциите за намиране на тегловен спектър на софтуерните пакети Magma и GAP и разработените от нас векторизирани алгоритми. Също така е представено сравнение между ефективността на алгоритмите, имплементирани чрез различни инструкции. Анализирана е ефективността на различните компилатори при използването на векторизация чрез разширени векторни регистри.

2.1 Алгоритми от високо ниво

Известно е, че проблема за намиране на тегловен спектър на линеен код е NP-пълен [10], като за решаването на задачата е необходимо да се генерират всички кодови думи, като линейни комбинации на редовете на пораждащата матрица на кода. Сред най-използваните подходи за генериране на всички линейни комбинации е използване на код на Грей [46]. Този подход е подходящ за двоични кодове. За кодове над други полета при използването на код на Грей ще се генерират всички q^k кодови думи. По-ефективен подход е генериране само на непропорционалните $\frac{q^k-1}{q-1}$ кодови думи, като един такъв подход е представен в Алгоритъм 1. Очевидно е, че този алгоритъм е практически неприложим. Алгоритъм 2 представя рекурсивната му реализация, като се използва подход за емулиране на вложени цикли, представен в [17]. В основните алгоритми 1 и 2 елементите на полето се обхождат чрез променливите $e_i, i = 2, \ldots, k$ за Алгоритъм 1 и e за Алгоритъм 2. За стойностите на $l(e_i)$ (съотв. $l(e)$) имаме $l(e_i) = e_i$ (съотв. $l(e) = e$) за прости полета и $l(e_i) = \alpha^{q-e_i}$ (съотв. $l(e) = \alpha^{q-e}$) при съставни полета, където α е примитивен елемент на полето. В реда i матрицата T от Алгоритъм 2 съдържа линейна комбинация на i реда от пораждащата матрица. Така генерирането на нова кодова дума се получава чрез добавяне на ред от пораждащата матрица умножен с елемент на полето. Генерирането само на непропорционални кодови думи се извършва чрез прескачане на линейни комбинации, чиито първи ненулев коефициент е различен от 1 (ред 10, където qf представлява горна граница за обхождането на елементите на полето). В представения основен алгоритъм се изпълняват $\theta = \frac{q^k-1}{q-1}$ събирания на вектори, θ умножения на вектор с елемент на полето и θ изпълнения на функцията *get_ weight*, която намира теглото на вектор.

Основната оптимизация на представения алгоритъм се състои в заменяне на операцията умножение на вектори, която е изчислително тежка, с операция за събиране на вектори. При работа над прости полета това е лесно постижимо, тъй като елементите на просто поле са остатъци по модул p. Операциите събиране и умножение се извършват по модул p. Тогава умножението с елемент от полето може да се замени с добавяне на един и същ ред от пораждащата матрица към текуща линейна комбинация. Алгоритъм 3 представя оптимизираната функция *Linear_ Combinations* за прости полета, като функцията *add* събира два реда от матриците G и T. Раздел 2.2 описва имплементацията на функциите *add* и *get_ weight* чрез векторизация за различните полета. Основната идея за оптимизация може да се опише по следния начин: ако текущият ред от пораждащата матрица се добавя към линейна комбинация на i реда

Algorithm 1 Основен алгоритъм за генериране на непропорционални кодови думи

```
1: global definition
2:     int G[k][n] - глобален двумерен масив, съдържащ k × n пораждащата матрица
3:     int T[k+1][n] - глобален двумерен масив, съдържащ (k+1) × n временна матрица от линейни комбинации
4: end global definition
5: function LINEAR_COMBINATIONS_SEQUENTIAL(int n, int k)
6:     for (i_1 = 1; i_1 <= k; i_1 ++) do T[1] = G[i_1];
7:         for (i_2 = i_1 + 1; i_2 <= k; i_2 ++) do
8:             for (e_2 = 1; e_2 < q; e_2 ++) do T[2] = T[1] + l(e_2)G[i_2];
9:                 ...
10:                for (i_k = i_{k-1} + 1; i_k <= k; i_k ++) do
11:                    for (e_k = 1; e_k < q; e_k ++) do T[i_k] = T[i_{k-1}] + l(e_k)G[i_k];
```

с коефициент 1 (добавя се за първи път към някоя линейна комбинация на $i-1$ реда), то той се добавя към ред $(i-1)$ на временната матрица. Така се получава линейна комбинация на i реда. В противен случай (текущият ред на G се добавя с коефициент $\neq 1$ в линейната комбинация), редът се добавя към съществуваща линейна комбинация на i реда. При поле с два елемента всички кодови думи са непропорционални и $\theta = 2^k - 1$. Тогава редовете 6 и 7 в Алгоритъм 3 могат да бъдат заменени с израза $T[r] = add(T[r-1], G[i])$. При просто поле с 3 елемента линейната комбинация може да се изчисли чрез следния израз $T[r] = add(T[r+e-2], G[i])$, който заменя редовете 6 и 7 от Алгоритъм 3.

Елементите на съставно поле $\mathbb{F}_q$, където $q = p^m$, могат да се представят като полиноми с коефициенти над крайното поле $\mathbb{F}_p$ и степен по-малка от m, на които се съпоставят вектори. Тогава събирането на вектори над полето е лесно за имплементиране. Умножението на елементи над съставно поле е с по-голяма сложност, като проблема за намаляването ѝ е широко разглеждан [8, 28, 29]. За оптимизация в представения алгоритъм се използват предварителни изчисления, които се извършват еднократно преди започване на основните изчисления. Предварително се генерира множеството от пораждащи матрици $M = \{G, xG, \ldots, x^{m-1}G\}$. Използва се примитивен пораждащ полином и следователно за примитивния елемент имаме $\alpha = x$. Това множество от матрици може да се намери чрез използване на таблица за умножение. При наличието на предварително изчисленото множество M, умножението на

Algorithm 2 Основен алгоритъм за изчисление на тегловен спектър на q-ичен линеен код чрез рекурсия

```
1: global definition
2:     int G[k][n] - глобален двумерен масив, съдържащ k × n пораждащата матрица
3:     int T[k+1][n] - глобален двумерен масив, съдържащ (k+1) × n временна матрица от линейни комбинации
4:     int A[n + 1] - глобален масив, съдържащ тегловния спектър
5:     int n, k, q - глобални променливи, показващи параметрите на кода
6:     int get_ weight(v) -външна функция за изчисляване на теглото на вектор v
7: end global definition
8: function LINEAR_COMBINATIONS(int r, int h)
9:    int qf = q;
10:    if h==1 then qf = 2
11:    for (int i = h; i <= k; i++) do
12:        for (int e = 1; e < qf; e++) do
13:            T[r] = T[r − 1] + l(e) ∗ G[i];
14:            int w = get_weight(T[r]);
15:            A[w]++;
16:            if r < k then Linear_Combinations(r + 1, i + 1);
17: function WEIGHT_DISTRIBUTION(G_curr, n_curr, k_curr, q_curr)
18:    G = G_curr; n = n_curr; k = k_curr; q = q_curr;
19:    for (int i = 0; i <= n; i++) do A[i] = 0;
20: /* предварителни изчисления, необходими за модифицираните алгоритми */
21:    Linear_Combinations(1,1);
```

Algorithm 3 Алгоритъм за намиране на спектър на линеен код над просто поле

```
1: function Linear_Combinations_prime(r, h)
2:     int qf = q;
3:     if h==1 then qf = 2
4:     for (i = h; i <= k; i++) do
5:         for (int e = 1; e < qf; e++) do
6:             if e==1 then T[r] = add(T[r − 1], G[i]);
7:             else T[r] = add(T[r], G[i]);
8:           int w = get_weight(T[r]);
9:           A[w]++;
10:            if r<k then Linear_Combinations_prime(r + 1, i + 1);
```

ред от пораждащата матрица с елемент от полето може да бъде заменено с използването на съответен ред от подходяща матрица от M. Този подход е представен в Алгоритъм 4. За избиране на матрица от M се използва редица на прехода на p-ичен код на Грей, означена с TS. По този начин генерирането на елементите на полето се получава в наредба от вида код на Грей чрез добавяне на ред на пораждащата матрица, умножен по примитивен елемент на полето. Генерирането на линейни комбинации чрез множеството M за полето $\mathbb{F}_8$ е представено в Таблица 2.1. За по-голяма яснота в таблица 2.1 отбелязваме редовете r, $r-1$ и i на матриците T и G съответно като t_r, t_{r-1} и g_i.

Algorithm 4 Алгоритъм за намиране на спектър на линеен код над съставно поле

```
1: function Linear_Combinations_composite(r, h)
2:     int qf = q;
3:     if h==1 then qf = 2
4:     for (i = h; i<=k; i++) do
5:         for (int e = 1; e < qf; e++) do
6:             if e==1 then T[r] = add(T[r − 1], G[i]);
7:             else
8:               int j = TS[e] − 1;
9:               T[r] = add(T[r], x^j G[i]);        // i-ти ред на матрицта x^j G
10:            int w = get_weight(T[r]);
11:            A[w]++;
12:            if r<k then Linear_Combinations_composite(r + 1, i + 1);
```

Таблица 2.1: Пример за генериране на линейни комбинации за полето $\mathbb{F}_8$

Грей	TS	e	Генериране	Лин. комбинация
000	0	0	-	-
001	1	1	$t_r = t_{r-1} + g_i$	$t_{r-1} + g_i$
011	2	2	$t_r + xg_i = t_{r-1} + g_i + xg_i$	$t_{r-1} + (x+1)g_i$
010	1	3	$t_r + g_i = t_{r-1} + (x+1)g_i + g_i$	$t_{r-1} + xg_i$
110	3	4	$t_r + x^2g_i = t_{r-1} + xg_i + x^2g_i$	$t_{r-1} + (x^2+x)g_i$
111	1	5	$t_r + g_i = t_{r-1} + (x^2+x)g_i + g_i$	$t_{r-1} + (x^2+x+1)g_i$
101	2	6	$t_r + xg_i = t_{r-1} + (x^2+x+1)g_i + xg_i$	$t_{r-1} + (x^2+1)g_i$
100	1	7	$t_r + g_i = t_{r-1} + (x^2+1)g_i + g_i$	$t_{r-1} + x^2g_i$

Анализ на сложността на представените алгоритми

В Алгоритъм 2 се изпълняват θ събирания на вектори, θ умножения на вектор с елемент от полета и θ изчисления на теглото на вектор. Модифицираните Алгоритъм 3 и Алгоритъм 4 не изпълняват операциите за умножение. Алгоритъм 4 има по-голяма сложност по памет, тъй като се използват m пораждащи матрици. В алгоритмите не са представени реализациите на функциите *add* и *get_ weight*, които извършват основните изчисления. За тяхната имплементация са използвани разширените векторни регистри. Различните реализации зависят от начина за представяне на елементите на полето. Те са обсъдени в следващият Раздел 2.2.

2.2 Алгоритми от ниско ниво

В този раздел представяме подходите за реализиране на алгоритмите от ниско ниво за събиране на вектори и изчисляване на теглото на вектор (функции *add* и *get_ weight*). Те са различни в зависимост от стойностите на q. В основата на тези алгоритми е подходящото представяне на елементите на полето за записване в регистър. В зависимост от дължината n на вектора регистъра може да се допълни с нули, ако е необходимо, или да се използва множество от регистри, ако векторът не може да бъде записан в един. За всяка реализация първо обсъждаме представянето на елементите и след това съответните алгоритми. В описанието на алгоритмите от ниско ниво за простота използваме вектори, които се събират в един регистър с дължина 128 бита.

2.2.1 Побитови имплементации за прости полета

Използването на побитово представяне на елементите и побитови операции може да се разглежда като най-естествения подход за паралелизация. За полетата с 2, 3 и 4 елемента са разработени различни побитови представяния и подходи за извършване на операции [1, 16, 30, 48, 56]. В този раздел са разгледани подходи за побитово представяне на вектори над простите полета $\mathbb{F}_2$ и $\mathbb{F}_3$ и реализиране на функциите за събиране и намиране на теглото на вектор чрез разширени регистри.

Подходи за оптимизация при $\mathbb{F}_2$

Векторите над поле с два елемента имат естествено двоично представяне - всеки бит може да отговаря за една координата на вектора. Тогава за записването на n-мерни двоични вектори в 128-битови регистри са необходими $\lfloor \frac{n-1}{128} + 1 \rfloor$ регистъра, които могат да бъдат разгледани като масив от регистри. При необходимост последния регистър може да се допълни с нули. Операцията събиране на вектори се реализира чрез една XOR операция за всяка двойка регистри. Интересен случай е n-мерното векторно пространство, където $n \leq 64$. В този случай повече от половината битове на регистъра не носят информация. Тогава може да се използва подходящо записване на данните в масива T от алгоритъм 2, което да реализира допълнителна паралелизация като се генерират две кодови думи едновременно.

За генериране на кодовите думи на код C с дължина $n \leq 64$, може да се използва $[n, k-1]$ подкод C' с пораждаща матрица G', получена от G като е премахнат последния ред g_k. Също така се разглежда съседният клас $g_k + C' = \{g_k + c | c \in C'\}$. Тогава първоначалният код C може да се представи като множеството $C' \cup (g_k + C')$. В реализацията реда g_k се записва във вторите 64 бита на нулевия ред на матрицата T (първите 64 бита са нули). Основната идея е да се генерират кодови думи от C' и съседния клас $(g_k + C')$ с една операция върху целия регистър. За целта ред от матрицата G' се съхранява два пъти в регистър (едно копие в първите 64 бита и още едно във вторите 64 бита). Така с всяко добавяне на ред от матрица G' се получава кодова дума от C' и нейния съседен клас едновременно. За извършване на операцията за намиране на теглото на вектор, един регистър от 128 бита може да разгледа като масив от две 64-битови компютърни думи. Това представяне за матрицата T може да се запише като $t_r = (t_r[0], t_r[1])$, където с t_r се означава реда r на матрицата. Намирането на теглото на кодови думи от C' и $(g_k + C')$ се извършва

чрез изпълнение на *popcnt* функцията за $t_r[0]$ и $t_r[1]$. Алгоритъм 5 представя реализираните модификации на Алгоритъм 3 за полето $\mathbb{F}_2$ при $n \leq 64$, като функциите *add* и *get_ weigth* са представени чрез техните реализации. За $n > 64$ намирането на теглото на вектор може да се използва същото представяне на $t_r = (t_r[0], t_r[1])$, като теглото на вектора е равно на сбора от теглата на $t_r[0]$ и $t_r[1]$.

Algorithm 5 Алгоритъм за намиране на спектър на линеен код над $\mathbb{F}_2$ при $n \leq 64$

```
1: function LINEAR_COMBINATIONS_GF2(r, h)
2:     for (i = h; i < k; i++) do
3:         t_r = t_{r-1} XOR g_i;
4:         int w = popcnt(t_r[0]);    A[w]++;
5:         w = popcnt(t_r[1]);     A[w]++;
6:       if r<k-1 then Linear_Combinations_GF2(r + 1, i + 1);
```

Подходи за оптимизация при $\mathbb{F}_3$

За полета с три елемента съществуват различни подходи за побитово представяне на елементите на полето [1, 16, 30, 48, 56]. Едно такова представяне дава възможност за записване на елемент на полето в два бита, като събирането на елементите се извършва чрез шест побитови операции [30]. Това представяне се изразява чрез следното изображение $\Pi : \mathbb{F}_3 \to \mathbb{F}_2^2$, където

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

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

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

Изображението Π може да се разшири за вектори над полето до $\pi : \mathbb{F}_3^n \to \mathbb{F}_2^{2n}$, където

$$\pi(v) = (\alpha_1, \alpha_2, \ldots, \alpha_n, \beta_1, \beta_2, \ldots, \beta_n),$$
$$v = (v_1, v_2, \ldots, v_n) \in \mathbb{F}_3^n, \Pi(v_i) = (\alpha_i, \beta_i).$$

Също така може да се разгледа вектора

$$\overline{\pi(v)} = (\beta_1, \beta_2, \ldots, \beta_n, \alpha_1, \alpha_2, \ldots, \alpha_n).$$

Използвайки тези представяния, операцията за събиране на вектори над $\mathbb{F}_3$ може да се реализира чрез 5 операции над регистри, представен в Алгоритъм

6. При имплементирането на алгоритъма стойностите на $\overline{\pi(b)}$ и $\bar{t}$ могат да бъдат изчислени чрез векторна функция за разместване на елементи в регистър *shuffle* (виж 1.2.2), а векторите са редове на пораждащата матрица G и помощната матрица T от Алгоритъм 3. Възможно е събирането да бъде извършено чрез 4 операции, като предварително се изчисли матрица $\overline{G}$, съдържаща изображенията $\overline{\pi(g_i)}$ за всеки ред от пораждащата матрица. Това от своя страна ще увеличи необходимата памет за реализирането на алгоритъма.

Algorithm 6 Алгоритъм за събиране на вектори над $\mathbb{F}_3$

1: **function** ADDF3(a, b)
2: $t, u, r \in \mathbb{F}_2^{2n}$;
3: $t = \pi(a) \oplus \overline{\pi(b)}$;
4: $u = t \oplus \overline{\pi(b)}$;
5: $r = \bar{t} \vee u$;
6: **return** r

Изображението π също така позволява лесно изчисление на броя на ненулевите елементи на даден вектор. Тъй като елемента 0 се изобразява в $(1, 1)$, за изчислението на теглото на вектор v при използване на изображението

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

е необходимо да се изпълни XOR операция за α и β частите на $\pi(v)$. Тогава Алгоритъм 7 изчислява теглото на вектор като използва *popcnt* функция.

Коректността на функциите *addF3* и *get_ weightF3* следва от зададеното изображение $\Pi : \mathbb{F}_3 \to \mathbb{F}_2^2$, описано в [30].

Algorithm 7 Алгоритъм за изчисление на теглото на вектор над $\mathbb{F}_3$

1: **function** GET_WEIGHTF3($\pi(v)$)
2: $a, b \in \mathbb{F}_2^n$;
3: $a = (\alpha_1, \ldots, \alpha_n)$;
4: $b = (\beta_1, \ldots, \beta_n)$;
5: $t = a \oplus b$;
6: int r = *popcnt*(t);
7: **return** r

2.2.2 Побайтови имплементации за прости полета

Елементите на простото поле $\mathbb{F}_p$ могат да бъдат представени като цели

числа в интервала $[0, p-1]$. Следователно в случаите, когато $p \neq 2, 3$ може да се използват 8-битови цели числа при $p < 128$. Събирането на вектори върху $\mathbb{F}_p$ обикновено се реализира чрез операция събиране по модул, която отнема повече изчислително време. Представената реализация има за цел да изпълни събирането на вектори с множество от различни инструкции. Методът, показан в Алгоритъм 8, изпълнява три инструкции - събиране на два вектора, изваждане на вектора $P = (p, p, \ldots, p)$ с дължина n от резултата на предишна операция и подходящо смесване на двата получени вектора. При тази операция се вземат положителните стойности от r_sub, а останалите от r_add. Последната операция се изпълнява от специалната инструкция *blendv(a,b,m)* от разширения набор от инструкции (виж 1.2.2).

Algorithm 8 Алгоритъм за събиране на вектори над $\mathbb{F}_p$ чрез *blendv*

1: **function** ADD_FP_BLENDV($u, v \in \mathbb{F}_p^n$)
2: $\_\_m128i\ r\_add, r\_sub, r\_blendv,\ P = (p, p, \ldots, p)$;
3: $r\_add = u + v$;
4: $r\_sub = r\_add - P$;
5: $r\_blendv = blendv(r\_add, r\_sub, r\_sub)$;
6: // $r\_blendv_i = \begin{cases} r\_add_i & \text{if } r\_sub_i < 0 \\ r\_sub_i & \text{if } r\_sub_i \geq 0 \end{cases}$
7: **return** r_blendv;

Има два основни подхода за изчисляване на теглото на вектор. За простота в представените алгоритми се приема, че $n \leq 16$, тъй като единичен 128-битов регистър ($n \leq 32$ за 256-битов регистър, $n \leq 64$ за 512-битов регистър) съдържа до 16 елемента от желания вид. Ако дължината на кода е по-малка от 16, всички кодови думи се разширяват с $16 - n$ нули. Първият алгоритъм маркира позициите на регистъра, съдържащ ненулеви елементи и след това изчислява теглото на дадения вектор, както е показано в Алгоритъм 9.

Algorithm 9 Алгоритъм за изчисление на теглото на вектор над $\mathbb{F}_p$ (версия 1)

1: **function** GET_WEIGHTFP_VERSION1($v \in \mathbb{F}_p^n$)
2: $\_\_m128i\ \ r = (0, 0, \ldots, 0)$
3: $r = cmpgt(v, 0)$; // $r_i = \begin{cases} FF & \text{ако } v_i > 0 \\ 0 & \text{ако } v_i \leq 0 \end{cases}$
4: **return** $(popcnt(r_1, \ldots r_{n/2}) + popcnt(r_{(n/2)+1}, \ldots, r_n))\ >>\ 3$; // (>> 3) - бързо деление на 8

Вторият подход използва броя на нулевите елементи w_0 за изчисляване на теглото. За целта се конструира компютърна дума, която има w_0 различни от нула битове. Това се постига чрез сравнение с нулев регистър, маска h и побитови операции. Алгоритъмът 10 показва изпълнението на втората версия при използването на 128-битови регистри и дължини на кода ≤ 16. Реализацията за 256-битови регистри разширява h до вектора

$$(8,8,8,8,8,8,8,8,4,4,4,4,4,4,4,4,2,2,2,2,2,2,2,2,1,1,1,1,1,1,1,1).$$

Изпълняват се още две операции - преместване надясно с 128 бита и операция *OR*. Основното предимство на Алгоритъм 10 е минимизирането на използването на инструкция *popcnt*. При използването на тази инструкция регистъра се разглежда като масив от 64-битови компютърни думи, като за целта всяка от тях трябва да се запише в променлива с подходяща дължина, което е относително бавен процес.

Algorithm 10 Алгоритъм за изчисление на теглото на вектор над $\mathbb{F}_p$ (версия 2)

function GET_WEIGHTFP_VERSION2($v \in \mathbb{F}_p^n$)
  $\_\_m128i\ r1, r2, r, m\_cmpl,$
  $h = (2,2,2,2,2,2,2,2,1,1,1,1,1,1,1,1);$
  $m\_cmpeq = cmpeq(v, 0);$ // $m\_cmpeq_i = \begin{cases} FF & \text{ако } v_i = 0 \\ 0 & \text{ако } v_i \neq 0 \end{cases}$
  $r1 = m\_cmpeq \wedge h;$
  $r2 = srli(r1, 8);$ //изместване надясно на битовете на r1 с 8 байта
  $r = r1 \vee r2;$
  $w_0 = popcnt(r_{(n/2)+1}, \ldots, r_n));$
return $16 - w_0;$

2.2.3 Съставни полета

За всяко просто p и положително цяло число m има поне един неразложим полином $g(x) \in \mathbb{F}_p[x]$ от степен m. Този полином може да се използва като пораждащ за съставното поле с $q = p^m$ елемента. Тогава може да се разгледа $\mathbb{F}_q = \{r(x) \in \mathbb{F}_p[x],\ \deg r(x) < m\}$, където операциите събиране и умножение се извършват по модул $g(x)$ (това е адитивното представяне на елементите на полето). Елементите на F_q могат да бъдат представени като вектори в $\mathbb{F}_p^m$, използвайки съответствието $r(x) = r_0 + r_1 x + \cdots + r_{m-1}x^{m-1} \mapsto (r^{(0)}, r^{(1)}, \ldots, r^{(m-1)})$,

където $r^{(i)} = r_i$ за всяко естествено $i \in [0; m-1]$. Представеният Алгоритъм 4 реализира изчисленията за намиране на тегловния спектър на кода като се използва единствено събиране на вектори над съставно поле. Следователно представянето на елементите на полето като вектори с елементи в простото поле p дава възможност изчисленията до бъдат сведени до изчисления над вектори в простото поле. Нека разгледаме векторното пространство F_q^n, където $q = p^m$ и елементите $r(x) \in F_q$ са представени като вектори $(r^{(0)}, r^{(1)}, \ldots, r^{(m-1)})$. Тогава векторите в $F_{p^m}^n$ могат да бъдат представени като вектори в F_p^{mn} по следният начин:

$$(u_0, \ldots, u_{n-1}), \mapsto (u_0^{(0)}, \ldots, u_{n-1}^{(0)}, u_0^{(1)}, \ldots, u_{n-1}^{(1)}, \ldots, u_0^{(m-1)}, \ldots, u_{n-1}^{(m-1)}),$$

където $u_i \in F_{p^m}, u_i^{(j)} \in F_p$. При така зададеното съответствие, събирането на вектори $u, v \in F_{p^m}$ при побитови и побайтови представяния в паметта се реализира чрез представените методи за събиране над прости полета. Намиране на теглото на вектор $u \in F_{p^m}^n$ чрез представянето му като вектор $u' \in F_p^{mn}$ използва допълнителни стъпки, които зависят и от представянето му в паметта. Нека се разгледа следното означение за $u' \in F_p^{mn}$:

$$u' = (u'_0, \ldots, u'_{m-1}), \text{ където } u'_i = (u_0^{(i)}, \ldots, u_{n-1}^{(i)}), i = 0, 1, \ldots, m-1.$$

За намирането на теглото на вектора $u \in F_{p^m}^n$ можем да разгледаме следните подходи в зависимост от $p \leq 64$ и m:

- При $p = 2$, $m = 2$ и побитово представяне на елементите в полето имаме следното представяне на $u' = (u'_0, u'_1)$. Тогава за намиране на теглото на u е необходимо да се извърши следната побитова операция $u'_0 \vee u'_1$, като резултата е вектор от F_2^n, чието тегло е равно на търсеното тегло, тъй като $u_i = 0 \iff u_i^0 = u_i^1 = 0$.

- При $p = 2$ и $2 < m \leq 6$ може да се използва побайтово представяне. Тук вектора $(u^{(0)}, \ldots, u^{(m-1)})$, съответстващ на елемент на полето, се записва в един байт. Събирането отново се реализира чрез побитова $\oplus$ операция. За намиране на теглото на вектора могат да се използват методите, представени в 2.2.2 за намиране на теглото на вектор над просто поле, тъй като нулев байт в това представяне съответства на нулев елемент на полето.

- При $p = 3$ и $m = 2, 3$ може да разгледаме побитово представяне за u'. Тогава за намирането на теглото на вектор при $m = 2$ се използва една побитова операция $u'_0 \vee u'_1$ (за $m = 3$ използваме две операции $u'_0 \vee u'_1 \vee u'_2$). Аналогично на случай, където $p = 2, m = 2$, резултатът от побитовите

операции е вектор от $\mathbb{F}_3^n$, чието тегло е равно на теглото на първоначалния вектор u.

- При $p > 3$ и $m = 2$ ($p^m \leq 64$) използваме побайтово представяне, като $u' = (u'_0, u'_1)$. Тогава за u_i е в сила следното: $u_i = 0 \iff u_i^0 = u_i^1 = 0$, където всяко u_i^0 и u_i^1 е записано в един байт. Ако разгледаме Алгоритми 9 и 10 виждаме извършването на сравнение с *0* съответно в редове 3 и 4. При съставните полета, посочените сравнения трябва да бъдат извършени за векторите u'_0 и u'_1. Резултатите от тези сравнения са регистри, които означаваме с m_0 и m_1. Те имат стойност FF в позиция i, тогава и само тогава, когато резултата от сравнените за u'_0 (съответно u'_1) е истина. За Алгоритъм 9 след изпълнение на сравнението се изпълнява допълнително операцията $m_0 \vee m_1$, тъй като елемента на полето не е 0 ако поне един от съответните елементи на m_1 и m_2 не е 0 (сравняваме елементите на u'_0 и u'_1 с оператор $>$). При Алгоритъм 10 се изпълнява побитовата операция $m_0 \oplus m_1$, тъй като е необходимо двата съответни елемента на m_0 и m_1 да са едновременно 0 (броят се нулевите координати на вектори).

2.3 Експериментални резултати

Представените алгоритми са включени в библиотека за изчисление на тегловни характеристики на линейни кодове над полета с $q \leq 64$ елемента. Тяхната ефективност е анализирана като е използвана функция за намирането на тегловен спектър на линеен код. Направено е сравнение между времената за изчисление на две векторизирани имплементации с регистри с дължина 128 и 256 бита. Времето за работа на имплементацията с 256-битови регистри е сравнено с невекторизирана имплементация на Алгоритъм 2, която използва таблици за събиране и умножение. Също така, времето за работа на разработените функции е сравнено с времето за изчисление на аналогични функции, налични в два широко използвани пакета за компютърна алгебра - Magma и GAP. Експерименталните резултати са изпълнени на операционна система Windows 10 с процесор Intel Core I5 1035G1 @1.00 GHz и компилатор gcc 8.1. Изчисленията, използващи пакета Guava за GAP, са извършени с помощта на същата платформа. Експерименталните резултати със софтуерния пакет Magma са извършени на онлайн калкулатора, работещ на виртуална машина с процесор Intel Xeon Processor E3-1220 @3.10GHz. В Таблица 2.2 са представени времената за работа в секунди на съответните функции за намиране на тегловно разпределение. В първите три колони са дадени параметрите на кодовете, за които се извършват

изчисленията. Следващите две колони дават времето за работа на функциите със софтуерните пакети Magma и GAP, съответно. Колони 5 и 6 дават времето за изчисления за векторизираните имплементации съответно с SSE4.1 и AVX2, докато последната колона представя времето за работа с невекторизирани версии на алгоритмите, представени в раздел 2.1. Изчисленията са извършени за кодове над прости и съставни полета с побитово и побайтово представяне, като са разгледани дължините 60 и 500.

Таблица 2.2 показва, че в случай на побитово представяне ($\mathbb{F}_2$,$\mathbb{F}_4$, $\mathbb{F}_3$,$\mathbb{F}_9$, $\mathbb{F}_{27}$) 256-битовата версия е сравнима със 128-битовата версия. При прости полета ($\mathbb{F}_2, \mathbb{F}_3$), имплементацията с 256-битови регистри е за $n > 100$. Анализирането на кода с Visual Studio Profiling Tools показва, че повече от 75% от изчислителното време се използва за изчисляване на теглото на векторите. Поради представянето на елементите на полето и наличните инструкции, в тези случаи се изпълняват множество *popcnt* функции. Данните също се прехвърлят между големи и малки 64-битови регистри, което е бавен процес. Допълнителна проверка на генерираната асемблерна версия показва, че във функциите за изчисляване на теглото на вектор, данните се прехвърлят от 256-битови регистри в 128-битови и след това в 64-битов регистър. Това също увеличава времето за изпълнение. При съставни полета се изпълняват по-малко *popcnt* функции, като по този начин времето за изчисляване на теглото на вектора, намалява до приблизително 60% от общото време за изпълнение. Въпреки това, тази функция е тежка и другите ограничения продължават да съществуват. Следователно времето за изпълнение с AVX2 е с между 10% и 30% по-малко в сравнение с SSE4.1. В случай на побайтово представяне ($\mathbb{F}_5$, $\mathbb{F}_{25}$), 256-битовата версия е между 1,2 и 1,7 пъти по-бърза от 128-битовата версия.

Нека сравним векторизираната версия с 256-битови регистри с функциите на Magma и GAP. Magma използва побитово представяне на елементите на полетата за $q \leq 5$ [80]. При простите полета $\mathbb{F}_2$, $\mathbb{F}_3$ и $\mathbb{F}_5$ се наблюдават между 1,4 и 2,5 по-бързи изчисления в сравнение с функцията на Magma, като ускорението се увеличава с увеличаването на дължината на кода. При съставните полета, представената векторизирана версия е между 1,9 ($\mathbb{F}_4, n = 500$) и 43,4 ($\mathbb{F}_{27}, n = 500$) пъти по-бърза в сравнение с Magma. Пакетът Guava за GAP е софтуер с отворен код и е включен в математическия софтуер SAGE. Според неговата документация оптимизацията за целевата функционалност се прилага само за поле $\mathbb{F}_2$. Това се вижда и от представените експериментални резултати в Таблица 2.2. При сравнение с GAP експерименталните резултати представят ускорение между 3,3 пъти ($\mathbb{F}_2$) и 1480 пъти ($\mathbb{F}_{27}$). Тук отново по-големи ускорения се наблюдават при съставните полета.

При сравнение с невекторизирана имплементация на алгоритъма се наблюдават ускорения между 7,8 ($\mathbb{F}_{25}$) и 58,9 ($\mathbb{F}_2$) пъти. Това се дължи на високия коефициент на векторизация [6]. Този коефициент показва максималния брой на координатите на даден вектор, които могат да бъдат записани в даден регистър. Той зависи от представянето на елементите в паметта и големината на използваният регистър. При побитово представяне коефициентът на векторизация за 128-битов регистър е между 21 и 128 в зависимост от полето. Тогава, при сравнение със скаларна имплементация, използваща таблици за събиране и умножение за операциите с елементи над полето, естествено се получава голямо ускорение.

Таблица 2.2: Изчислително време (сек.) на функция за намиране на тегловен спектър с различни софтуерни пакети

n	k	q	Magma	GAP	SSE4.1	AVX2	Без векторизация
60	29	2	1.780	6.832	1.431	1.252	60.936
500	29	2	12.910	24.067	5.920	7.311	430.856
60	16	4	14.190	825.245	5.998	7.389	173.034
500	16	4	56.450	6999.176	31.981	28.995	1141.309
60	19	3	4.880	311.815	2.871	3.224	67.744
500	19	3	25.600	2589.720	9.285	10.364	461.789
60	9	9	3.260	68.989	0.271	0.236	5.156
500	9	9	24.830	492.667	1.094	0.977	38.973
60	6	27	2.000	66.120	0.093	0.093	1.710
500	6	27	14.700	501.733	0.389	0.339	11.736
60	13	5	4.540	242.880	4.356	3.378	35.781
500	13	5	20.830	1989.983	20.568	14.979	241.332
60	6	25	1.320	41.891	0.166	0.141	1.105
500	6	25	10.060	321.037	0.921	0.541	8.093

2.4 Векторизация и ефективността на компилаторите

Векторизация на алгоритъм може да бъде постигната от компилатора без допълнителни инструкции, чрез директиви на компилатора и OpenMP [21, 79] или чрез директно използване на инструкции и съответните им функции от високо ниво за дадената архитектура на централния процесор. Първите два

подхода разчитат изцяло на използвания компилатор за постигане на векторизация. Често тези подходи не постигат достатъчно добра паралелизация на алгоритмите [6, 72]. Тогава е възможно да се използва директна векторизация чрез инструкциите на централните процесори. Основните изчисления в алгоритмите от високо ниво включват операции с вектори над крайни полета, което ги прави особено подходящи за векторизация. Също така, представените алгоритми от ниско ниво са разработени с идеята за имплементация чрез специализираните инструкции с разширени векторни регистри.

Изборът на компилатор може да повлияе на бързодействието и при директна векторизация. За да бъде анализирана ефективността на някои от често използваните компилатори (gcc, clang, msvc) за операционните системи Windows и Ubuntu са сравнени алгоритмите, представени в тази глава, като са използвани 128 и 256-битови регистри за x86 архитектури. Изчисленията са извършени за полето с два елемента. Сравнена е ефективността на три основни компилатора - gcc (GNU Compiler Collection), clang и msvc (Microsoft Visual C++). Компилаторът gcc традиционно се използва в Linux/Unix базирани операционни системи. Въпреки това, той има реализация за компилация на операционните системи Windows, известна като mingw [63]. Той е компилатор по подразбиране за интегрираната среда за разработка на софтуер CodeBlocks. Компилаторът msvc е разработен за Windows операционни системи и интегрираната среда Visual Studio, което го прави несъвместим с други среди за разработване на софтуер и различни операционни системи. Третият разгледан компилатор е clang, който основно е за операционните системи на Макинтош. Този компилатор може да бъде използван като алтернативен компилатор в Linix базирани операционни системи, тъй като поддържа флаговете за настройка на компилацията на компилатора gcc. Този компилатор също така може да бъде използван с Visual Studio чрез специален набор от инструменти, разработени за Visual Studio 2019. При използването на clang с Visual Studio предефинираните макроси и флаговете за оптимизация съвпадат с тези на компилатора по подразбиране.

В таблиците 2.3 и 2.4 са представени времена на изпълнение в секунди за различни n и k на Windows 10 OS, докато Таблица 2.5 показва времената на изпълнение в секунди за Ubuntu 20.04 OS. Таблица 2.3 дава времето за изпълнение чрез 128-битови регистри, докато Таблица 2.4 дава времето за изпълнение за 256-битови регистри. Изчисленията са извършени с процесор Intel Core i5 - 1035G1 с базова тактова честота 1,00 GHz и на Windows 10 OS като са генерирани произволни кодове за дадените параметри, за които изчисляваме тегловния спектър. Избрани са следните четири различни дължини, тъй

като за всяка от тях се използва различен брой регистри с дадена дължина. Експерименталните резултати показват, че никой компилатор не е значително по-бърз от останалите. Освен това не може да се каже категорично кой компилатор е по-добър за нашия специфичен проблем на Windows OS - различните компилатори се представят по-добре за различни параметри и регистри. Някои от наблюденията, които могат да бъдат направени след анализиране на експерименталните резултати от таблици 2.3 и 2.4, са следните:

- Функциите, компилирани с Clang, използван с Visual Studio, дават полоши времена за изпълнение в повечето от горните случаи в сравнение с msvc.

- Компилаторът msvc работи по-добре от компилатора mingw при използване на 128-битови регистри в повечето случаи. За случаите, когато е по-бавно, разликата във времето за изпълнение е под 10%.

- Функциите, компилирани с mingw, дават по-добро време за изпълнение от msvc и clang при използване на 256-битови регистри и за кодове с помалка дължина. Въпреки това, с увеличаването на дължината на кода, времето за изпълнение на msvc и clang се подобрява в сравнение с mingw.

- Времето за изпълнение на софтуера, компилиран с msvc, използващ 256-битови регистри, е по-бавно в сравнение със 128-битовата версия за кодове с $n = 50$. Разликата в този конкретен случай се състои в записването в паметта теглата на четири 64-битови компютърни думи, тъй като теглата се изчисляват за 4 кодови думи едновременно, използвайки съседни класове.

Таблица 2.5 показва експериментални резултати, изпълнени на Ubuntu 20.04 с Intel core i3-8145U с тактова честота 2,1 GHz. Параметрите на кодовете са показани в първите две колони и са същите като експериментите на Windows OS. Следващите две колони показват средното време за изпълнение в секунди след 10 повторения за изчисление с помощта на 128-битов регистър и компилиран софтуер с g++ и clang++ компилатор. Последните две колони показват средното време за изпълнение в секунди за изчисление с помощта на 256-битов регистър и същите компилатори. Изводите, които могат да се направят са аналогични на изводите, направени за таблиците 2.3 и 2.4. Разликите в средните времена на изпълнение са $\pm$ 10% между двата компилатора. Компилаторът clang++ се представя по-добре при работа със 128-битови регистри, докато g++ дава по-добри резултати за по-големите регистри. Освен това с двата компилатора не се получават толкова големи подобрения на времето за

Таблица 2.3: Време за изпълнение чрез 128-битови регистри

n	k	mingw	msvc	clang
50	26	0.202	0.186	0.234
	28	0.795	0.693	0.799
	30	3.158	2.799	3.267
100	26	0.323	0.335	0.369
	28	1.126	1.303	1.451
	30	7.148	5.163	5.755
200	26	0.452	0.483	0.551
	28	1.786	1.917	2.160
	30	8.228	7.339	8.638
600	26	0.753	0.759	0.732
	28	3.199	2.972	2.838
	30	11.931	11.939	11.440

Таблица 2.4: Време за изпълнение чрез 256-битови регистри

n	k	mingw	msvc	clang
50	26	0.134	0.182	0.157
	28	0.483	0.725	0.603
	30	1.893	3.111	2.624
100	26	0.195	0.215	0.280
	28	0.741	0.831	1.139
	30	2.943	3.319	4.587
200	26	0.372	0.396	0.438
	28	1.495	1.534	1.751
	30	5.881	6.103	7.179
600	26	0.736	0.648	0.697
	28	2.919	2.757	2.861
	30	11.636	10.148	10.974

Таблица 2.5: Време за изпълнение чрез 128- и 256-битови регистри и Ubuntu OS

		128-битов рег.		256-битов рег.	
n	k	g++	clang++	g++	clang++
50	26	0.204	0.195	0.116	0.123
	28	0.849	0.766	0.470	0.517
	30	3.156	2.964	1.780	2.504
100	26	0.311	0.349	0.176	0.213
	28	1.068	1.280	0.687	0.819
	30	4.304	5.490	2.745	3.265
200	26	0.401	0.291	0.359	0.369
	28	1.572	1.523	1.488	1.444
	30	6.279	6.047	5.687	6.203
600	26	0.710	0.626	0.655	0.707
	28	2.803	2.427	2.592	2.817
	30	11.188	9.748	10.323	11.474

изпълнение за $n = 600$, докато използват 256-битови регистри в сравнение със 128-битовите версии. Тези наблюдения насочват към заключението, че софтуер, компилиран с msvc на Windows OS, може да доведе до по-бавно време за изпълнение, когато е необходим чест достъп до паметта.

Коментари

Използваните подходи за двоични кодове, описани в тази глава, са представени в [P1]. Имплементациите на алгоритмите са включени в библиотека за намиране на тегловния спектър на линеен код и други тегловни характеристики. Алгоритмите от високо и ниско ниво и създадената библиотека са представени в [P6]. Анализът на ефективността на различните компилатори при използването на векторизация е представен в [P3]. Резултатите са представени на следните национални и международни научни форуми [D1, D2, D4, D6, D9].

Глава 3

Оптимизация на алгоритми чрез беззнакови типове от данни и разширените инструкции AVX512 и NEON

В представената глава са описани основни особености при векторизиране на алгоритми с разширените инструкции AVX512 и Neon, както и използването им за оптимизиране на алгоритми чрез беззнакови типове от данни. В Раздел 3.1 са описани особеностите на AVX512 инструкциите, които са налични в някои съвременни архитектури от вида x86. В Раздел 3.2 са представени някои особености на разширените инструкции за ARM процесорите и съответните регистри, известни като NEON. Представени са и някои основни разлики със съответните инструкции за x86 архитектурите. В Раздел 3.3 са представени алгоритми от ниско ниво, които реализират събиране на вектори над по-големи прости полета ($q \leq 128$), като се използват побайтовите представяния, описани в Раздел 2.2.2, и беззнакови типове от данни и инструкции със сатурация [70]. Най-съществената разлика в представените тук алгоритми е използването на същия голям фактор на векторизация, но при работа с прости полета с до 128 елемента. Раздел 3.4 представя сравнение между AVX512 и NEON инструкциите, като се използват алгоритмите от ниско ниво чрез беззнакови типове от данни. В този раздел също така е анализирана ефективността на AVX512 инструкциите спрямо останалите инструкции за x86 архитектури.

3.1 Разширени инструкции AVX512

Семейството от инструкции AVX512 е разделено на различни подкатегории. Те съдържат специализирани инструкции за работа с разширените регистри, като категоризацията им зависи от основните функции, които са включени в съответната подкатегория. Наличието на 512-битови регистри не гарантира наличието на всички инструкции. За проверка на наличните инструкции се използва вградената инструкция на централните процесори в x86 архитектурите *CPUID*. Чрез нея се извлича информация за централния процесор, която е записана в 4 константни регистъра. Информация за извличането и интерпретирането на данните, записана в тези регистри, може да се намери в [51].

В AVX512 също така са включени и осем допълнителни маскиращи регистри (opmask). Основното предназначение на тези регистри е свързано с ефективно сливане и условно записване в резултатните регистри. Дефинирани са функции, използващи тези маскиращи операции. Те имат основно значение при реализирането на алгоритмите от ниско ниво с 512-битови регистри, описани в Раздел 2.2. Основните функции за сравнение при инструкции от семействата SSE и AVX връщат като резултат регистър, съдържащ еднотипни елементи с големина, равна на големината на елементите, записани в целевите регистри. За разлика от тях, инструкциите за сравнение от основната подкатегория връщат като резултат маскиращ регистър с подходяща дължина, където бит i е 1, ако логическата операция е вярна за съответните елементи i, записани в целевите регистри и 0, когато логическата операция не е вярна. Следват някои от основните подкатегории от инструкции, които са използвани за имплементацията на алгоритмите от ниско ниво.

- Централните процесори, които разполагат с 512-битови регистри, имат основната подкатегория инструкции *AVX512Foundation*. В нея са включени основни операции за работа с регистрите (аритметични операции, логически операции, побитови операции, функции за записване на данни в паметта и др.).
- *AVX512VPOPCNTDQ* - функции, позволяващи намирането на теглото на целочислени елементи, записани в регистър. Функциите са дефинирани за 128-, 256- и 512-битови регистри и реализират намирането на ненулевите битове на целочислени 32- или 64-битови елементи в целевия регистър. Също така са дефинирани функции, които позволяват използването на регистър маска, който показва дали резултатната стойност за съответен елемент от регистъра да бъде записана. Тези функции позволяват

алгоритмите от ниско ниво за намиране на теглото на вектор, описани в раздел 2.2, да бъдат реализирани чрез по-малък брой функции. При побитовите представяния, функцията *popcnt*, която се изпълнява за 64-битови компютърни думи, може да се замени с *_mm512_popcnt_epi64*. Резултатният регистър може да се разгледа като вектор $x = (x_1, \ldots x_n)$, където n е броят на 64-битовите думи, които се съхраняват в регистъра (за 512-битов регистър $n = 512/64 = 8$) и x_i е броят на ненулевите битове на съответния елемент i на входния регистър. Следователно броят на ненулевите битове на целия регистър може да се изчисли чрез събиране (инструкция *_mm512_reduce_add_epi64*) на координатите в x.

- *AVX512VL*(*Vector Length*) - разширява основните функционалности, дефинирани в *AVX512Foundation*, за работа с по-малки регистри (128 и 256 бита). Такива са функциите за сравнение и различните инструкции, оперириращи с маскиращи регистри.

- *AVX512BW* (*Byte and Word Instructions*) - разширява основните функционалности, дефинирани в *AVX512Foundation*, като добавя функции за 8 и 16-битови цели числа. Такива функции се използват за имплементация на алгоритмите за събиране на вектори над прости полета с побайтово представяне (събиране, изваждане, сравнение и други функции за регистри, съдържащи 8-битови цели числа).

3.2 Разширени инструкции NEON

Архитектурите от вида ARM (Advanced RISC Machines)[7, 64] са все по-широко използвани поради тяхната енергийна ефективност. Процесори и копроцесори с тази архитектура се използват както в системи за видео обработка, така и в системи за персонална употреба. За тези архитектури са разработени т. нар. NEON регистри с дължина 128 бита и съответен набор от инструкции. Тези регистри могат да се разглеждат като аналог на 128-битовите регистри и SSE инструкциите в x86 архитектурите. Някои от съвременните компилатори позволяват "автоматична" векторизация на даден софтуерен код (GCC, Arm Compiler for HPC и др.), като също така е възможно изрично да се укаже "желанието" за векторизация на кода чрез директиви на компилатора (OpenMP). В тези случаи компилатора "превежда" дадения код на машинен език, който да използва разширените инструкции за конкретната архитектура. Аналогично на разширените регистри в x86 архитектурите, по-ефикасен подход за векторизация с ARM процесори е директното използване на функциите и типовете от

данни, създадени за работа с тези разширени векторни регистри в езиците от високо ниво. Въпреки, че разширените регистри за x86 и ARM архитектурите и съответните функции и типове от данни имат сходна цел - лесно осъществяване на векторизация - те се различават в голяма степен в своята имплементация. Следователно, ускорението на алгоритми, разчитащо на изрична векторизация, може да има различна имплементация в зависимост от архитектурата. В този раздел са представени някои основни разлики между разширените инструкции NEON и SSE инструкциите за x86 архитектури, които са ключови при имплементацията на алгоритмите от ниско ниво.

- Синтаксис на типовете от данни и функции - за работа с Neon регистрите са разработени специални типове от данни. Те позволяват записването на еднотипни елементи в 64- или 128-битови регистри. Всеки един от тях описва типа на данните, които ще бъдат записани в регистъра, неговата големина и броя на елементите, които ще бъдат записани в регистъра. Например int8x16_t съдържа 16 целочислени елемента с големина 8 бита. Броят на елементите, които могат да бъдат записани в регистъра зависи от големината на регистъра, както и от големината на базовият тип данни (128-битов регистър може да съдържа 16 елемента от типа char, 8 елемента от типа short int, 4 елемента от типа int и т.н.). При инструкциите за x86 архитектури се описва единствено големината на регистъра и типа данни (например __m128i съдържа целочислени данни, записани в 128-битов регистър). Функциите, дефинирани в специални библиотеки, улесняват работата с регистрите в езиците C/C++. Тези функции и типовете от данни са дефинирани в библиотеката arm_neon.h и се използват като стандартни. По време на компилацията функциите се заменят от последователност от инструкции на ниско ниво. Това означава, че може да проследи архитектурно поведение от ниско ниво на език от високо ниво. Дефинираните функции имат стандартизирани наименования, които описват операцията на функцията, големината на регистъра (наличието на символа “q” след кодовата дума за операцията показва, че ще бъде използван 128-битов регистър), типа на данните и тяхната големина. Всяка функция започва със символа “v” (виж Пример 3.1).

  Пример 3.1. *vaddq_s64 е функция, която събира два 128-битови регистъра, които съдържат 64-битови цели числа със знак в NEON.*

- Операции за сравнение на вектори от целочислени елементи - в NEON инструкциите са дефинирани операциите ($<, >, =, \leq, \geq$) за сравнение на целочислените елементите, записани в регистър. Операциите сравняват

едновременно съответните елементи, записани в два регистъра. Резултатът се записва в трети регистър, като за всеки съответен елемент, резултата е променлива с подходяща големина, съдържаща само битове 1, ако сравнението е истина, и само 0 в противен случай. Инструкциите от семействата SSE, AVX и AVX512 притежават аналогични функции, позволяващи сравнение на елементите в два регистъра. За разлика от NEON инструкциите, при тях не са налични всички операции за сравнение. Също така, наличието на дадена операция за едно от семействата, не гарантира наличието им за останалите.

- Операции за смесване и пермутации - някои от наборите от инструкции за x86 архитектури притежават функции за смесване на два регистъра (*blendv*) и извършване на пермутации на елементите в един регистър (*shuffle, permute*). Такива инструкции не са налични за набора от инструкции NEON.

- Функции за намиране на броя на ненулевите битове на целочислени променливи - голяма част от алгоритмите за намиране на тегловни инварианти изискват намирането на теглото на вектор. При използването на векторизация и подходящо представяне на елементите на полето, намирането на тегло на вектор може да се сведе до намирането на ненулевите битове в компютърна дума или регистър. Такива функции са известни като *population count* или *popcnt*. В NEON такива инструкции намират броя на ненулевите битове на всеки байт, записан като вектор в регистър с големина 64 или 128 бита. Резултата за съответните елементи се записват в друг регистър на съответни позиции. За намирането на теглото на цял векторен регистър е необходимо да се съберат резултатните стойности, което е възможно чрез една функция. Някои процесори с x86 архитектура имат специална *popcnt* функция, която намира теглото на 32 или 64 битова компютърна дума.

3.3 Оптимизация чрез беззнакови типове от данни и разширени регистри

Разширените векторни инструкции за x86 и ARM архитектури разполагат с функции със сатурация. Две основни такива функции са събиране и изваждане. При изпълняване на някоя функция, използвайки сатурация, резултатът от операцията е равен на резултата от стандартното изпълнение на операцията,

когато не се излиза от диапазона на дадения тип от данни. Ако резултата от операцията е извън диапазона на типа от данни, записан в регистъра, за някой елемент, то в изходния регистър за съответния елемент се записва променлива, съдържаща само нулеви битове, когато стойността е по-малка от долната граница за валидните стойности на дадения тип от данни. Когато резултатът е по-голям от горната граница на валидните стойности, в регистъра на съответната позиция се записва променлива с подходяща дължина, съдържаща само ненулеви битове.

Операциите събиране и изваждане на регистри се изпълняват, като елементите в регистъра могат да се интерпретират само като типове от данни със знак. Операциите със сатурация могат да се изпълняват като записаните данни се интерпретират като цели числа с или без знак. Начина на интерпретиране на данните може да промени изходния резултат при операциите със сатурация, тъй като интервала на стойностите е различен. Пример 3.2 представя резултата от изпълнението на стандартно изпълнение на операция изваждане, където елементите се интерпретират като тип от данни със знак, операцията изваждане като се използва сатурация и последователно изваждане на елементите, където данните се интерпретират като цели числа, записани в беззнаков тип от данни. При изпълнение на операцията без сатурация (чрез типове от данни със и без знак) резултата от изпълнението е един и същ, което е видимо при разглеждането на побитовото представяне на данните. Разликата се състои единствено в начина на интерпретиране на информацията (битовете), записана в дадената променлива.

Пример 3.2. $A = (0, 14, 31, 75)$ $B = (31, 31, 31, 31)$
$(A - B) = (-31, -17, 0, 44)$ - *стандартно изваждане без сатурация*
$(A - B)_{saturation} = (0, 0, 0, 44)$ - *изваждане на беззнакови числа със сатурация*

Функциите, реализирани чрез сатурация, позволяват алгоритмите за събиране на вектори да бъдат реализирани за вектори над крайни прости полета с $q < 128$ елемента и с голям фактор на векторизация. Това позволява да се извършат изчисления за по-големи полета, като се използва същото представяне и ресурс, използвани за реализирането на алгоритмите от ниско ниво от Раздел 2.2. За целта елементите на полето се записват в беззнакови типове от данни. Алгоритъм 8 реализира събиране на вектори над просто поле с $q < 64$, като се използва инструкция за x86 архитектурите, която позволява съчетаването на два различни регистъра, в зависимост от стойностите в трети регистър. При този алгоритъм това е възможно, тъй като се използват типове от данни със знак и резултата от изваждането може да бъде отрицателно число. Ал-

горитъм 11 представя подход за имплементация на събиране на вектори чрез разширени векторни регистри, базиран на използването на беззнакови типове от данни и инструкции за събиране и изваждане чрез сатурация за 128- и 256-битови регистри. Инструкцията *blend* е заменена от функции за сравнение, чиито резултат се използва като маска за реализирането на съчетаването на резултатите от операциите за събиране и изваждане, като се използват побитови операции. Алгоритъм 11 също така може да се използва за реализирането на събиране на вектори над прости полета чрез набор от инструкции, които нямат функция *blend*, като NEON инструкциите. При имплементацията на този алгоритъм с NEON инструкции, алгоритъмът може да се изпълни в 5 стъпки (с две по-малко), тъй като е налична инструкция за сравнение на елементите с логическа операция "по-малко"(<). Тогава ред 5 може да се замени с инструкция *m1 = cmplt(r1,P)*, редове 5 и 6 се премахват, а ред 7 побитовата операция се изпълнява за операндите $r1$ и $m1$.

Algorithm 11 Алгоритъм за събиране на вектори, използвайки сатурация и SSE, AVX или NEON инструкции

1: **function** ADD_SATURATION(a, b)
2: const P = $(p, p, \ldots, p)$, ZERO = (0,0,. . . , 0)
3: r1 = A + B // покомпонентно събиране на вектори чрез сатурация
4: r2 = r1 - P; // покомпонентно изваждане на вектори чрез сатурация
5: m1 = cmpeq(r1,P); $//m1_i = \begin{cases} FF & \text{if } r1_i = p \\ 0 & \text{if } r1_i \neq p \end{cases}$
6: m2 = cmpeq(r2,ZERO); $//m2_i = \begin{cases} FF & \text{ако } r2_i = 0 \\ 0 & \text{ако } r2_i \neq 0 \end{cases}$
7: r3 = m1 $\oplus$ m2 // побитова операция XOR
8: r4 = r3 AND r1 // побитова операция AND
9: c = r4 OR r2 //побитова операция OR
10: **return** c

Алгоритъм 11 при използване на AVX512 също може да се имплементира като се използва логическа операция "по-малко"(<), която е налична за подкатегория *AVX512BW*. Резултата от инструкциите за сравнение в AVX512 са маскиращи регистри. При изпълнението на инструкция за сравнение за 512-битов регистър, който съдържа 64 елемента с дължина 8 бита, резултата е 64-битов регистър маска, където всеки бит съответства на резултата от логическото сравнение за съответните елементи. Може да се изпълнят допълнителни команди, за да се съпостави на този регистър, такъв с големина 512 бита

и изчисленията да бъдат сведени до изпълнението на Алгоритъм 9. Друг подход при използването на AVX512 позволява свеждането на изчисленията до изпълнение на 4 инструкции. Този подход е представен в Алгоритъм 12 чрез функция *blend*.

Algorithm 12 Алгоритъм за събиране на вектори чрез сатурация и AVX512

1: **function** ADD_SATURATION(a, b)
2: const P = $(p, p, \ldots, p)$, ZERO = (0,0,. . . , 0),
3: __mmask64 m; // маскиращ 64-битов регистър
4: r1 = A + B // покомпонентно събиране на вектори чрез сатурация
5: r2 = r1 - P; // покомпонентно изваждане на вектори чрез сатурация
6: m = cmplt(r1,P); $//m_i = \begin{cases} 1 & \text{ако } r1_i < p \\ 0 & \text{ако } r1_i \geq p \end{cases}$
7: c = blend(m,r2,r1); $//c_i = \begin{cases} r1_i & \text{ако } m_i = 1 \\ r2_i & \text{ако } m_i = 0 \end{cases}$
8: **return** c

3.4 Експериментални резултати

В този раздел ще бъде разгледана ефективността на разширените инструкции AVX512 и NEON. Нека първо се разгледа имплементацията на Алгоритъм 3 за намиране на тегловен спектър на линеен код. Функциите за събиране на вектори над крайно поле реализират Алгоритми 11 и 12, като при използването на NEON инструкциите е имплементирана оптимизирана версия с изпълнение на една инструкция за сравнение. В Таблица 3.1 са представени времената за работа на представените алгоритми в секунди. Първите три колони дават параметрите n, k и p на кодовете, за които се изпълняват функциите. Във втора, трета и четвърта колона са представени времената за изпълнение на функция за намиране на тегловен спектър на линеен код, имплементирани съответно с инструкции SSE, AVX512 и NEON. Изчисленията са извършени на следните две платформи:

- x86 архитектура - централен процесор Intel Xeon Gold CPU 5118 с 24 ядра, 48 нишки, 2.3 GHz базова работна честота и Linux операционна система Red Hat Enterprise 7.9 и компилатор gcc 9.2;

- ARM архитектура - централен процесор от серията M1 на Apple с 8 ядра, 8 ГБ оперативна памет, операционна система MacOS Sonoma 14.5 и компилатор clang 15.0.0

От експерименталните резултати, представени в Таблица 3.1 могат да се направят следните изводи за ефективността на различните инструкции и архитектури:

- Ускорението, което се получава при сравнение на времената за работа на SSE и AVX512 инструкциите е между 1,5 и 4,5 пъти. Ускорението също така се увеличава с нарастване на дължината на кодовете. Това показва добрата скалируемост на представените имплементации. Едно от основните причини за постигането на ускорение по-голямо от 4 пъти е опростяването на алгоритмите за събиране и намиране на теглото на вектор при инструкциите от семейството AVX512. Събирането на вектори се реализира с по-малко инструкции (Алгоритъм 12), докато намирането на теглото на вектор се реализира чрез 2 инструкции (сравнението с 0 връща като резултат 64-битов регистър маска, чието тегло се намира с една инструкция $popcnt$).

- Сравнявайки времената за изпълнение на оптимизираните алгоритми с инструкциите, разработени за 128-битови регистри за двете основни архитектури, се наблюдава между 1,38 и 2,93 по-бързи изчисления при работа с NEON инструкции.

- За всички видове инструкции се наблюдава подобряване на ефективността с нарастване на дължината на кода. Това е така, защото при използването на повече регистри, времето за последователни инструкции като зареждане на данни в регистъра, се компенсира от постигнатото ускорение.

Нека разгледаме експерименталните резултати за алгоритмите, описани в раздел 2.2 чрез инструкции от вида AVX512 и NEON. Фигура 3.1 показва времето за изпълнение на изчисленията чрез AVX512 за различни стойности на k за полета $\mathbb{F}_2$, $\mathbb{F}_3$, $\mathbb{F}_5$ и $\mathbb{F}_{27}$ и различни стойности на n между 100 и 3000. Фигура 3.2 показва времето за изпълнение за същите параметри при имплементация с NEON инструкции. От Фигура 3.1 се вижда, че времената за изпълнение при дължина на кода между 200 и 500 за $\mathbb{F}_2$ и между 100 и 200 за $\mathbb{F}_3$ са близки. Това се обяснява с факта, че в тези случаи коефициентът на векторизация е повече от 256 и се използва един регистър за изчисленията (при $q = 2$ и $n = 100$ се използват съседни класове). При $\mathbb{F}_{27}$ изпълнението на функцията за намиране на

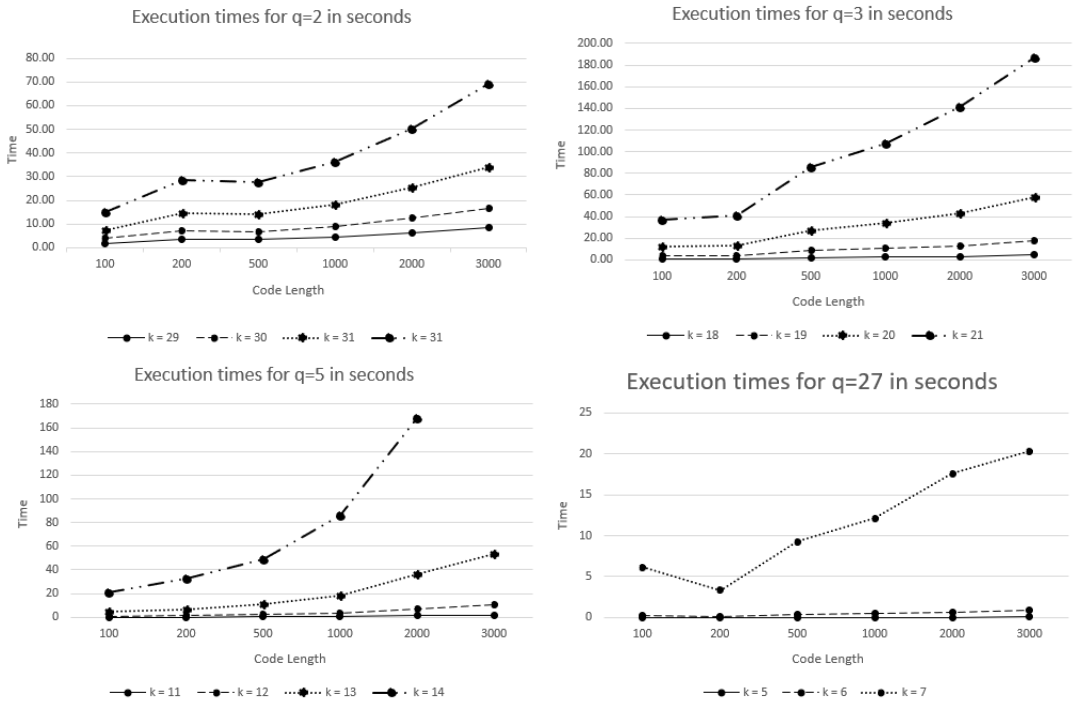
Таблица 3.1: Ефективност на AVX512 и NEON инструкции

n	k	p	SSE	AVX512	NEON
40	8	17	5.57	3.86	3.84
120	8	17	9.83	4.62	5.42
240	8	17	15.36	6.10	6.28
500	8	17	35.86	8.24	12.34
40	5	101	1.25	0.82	0.96
120	5	101	2.23	1.01	1.39
240	5	101	3.74	1.39	1.45
500	5	101	8.53	1.89	2.91
40	5	127	3.13	1.99	2.33
120	5	127	5.80	2.47	3.39
240	5	127	9.10	3.44	3.56
500	5	127	20.87	4.67	7.21

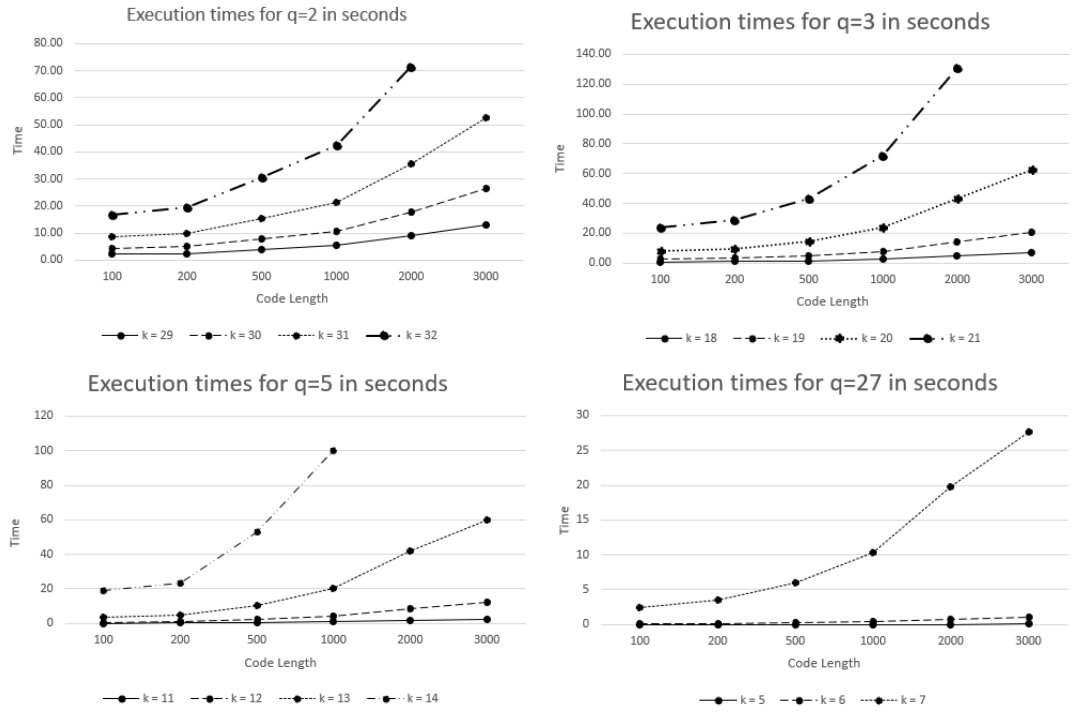
спектър при $n = 100$ е по-бавно в сравнение с $n = 200$. В този случай коефициентите на вектора са записани в един регистър, но за изпълнение на основните векторизирани функции се изпълняват няколко тежки операции (пермутация на елементите, записани в регистър). За останалите дължини, както се вижда от графиката, времето за работа се увеличава с увеличаване на дължината. При $\mathbb{F}_5$ се използва побайтово представяне на елементите на полето в паметта (виж Раздел 2.2.2). Тогава коефициентът на векторизация за 512-битови регистри е 64. Следователно, за избраните стойности на n времето за работа се увеличава пропорционално на дължината на кода.

От Фигура 3.2 се вижда, че при по-малък коефициент на векторизация поради по-малка дължина на регистъра, времето за изпълнение нараства пропорционално на увеличаване на дължината на кода. Също така не се наблюдава по-бавно изпълнение за $\mathbb{F}_{27}$ и $n = 100$, поради представянето на елементите в паметта и липсата на необходимост от използване на тежки функции за пермутация на елементите в регистър. При сравняване на изпълнението с NEON инструкции с това чрез AVX512 се вижда, че при по-малки дължини и размерности, времената за изпълнение са близки. При нарастване на стойностите на параметрите, времето за изпълнение при AVX512 е по-бързо в сравнение с това на имплементацията с NEON. Описаните зависимости и за двата вида инструкции са по-добре изразени при увеличаване на размерността на кода и следователно броят на генерираните кодови думи.

Фигура 3.1: Време за изчисление за AVX512



Фигура 3.2: Време за изчисление за NEON



3.4.1 Избор на оптимална дължина на регистър при x86 архитектури

За избор на оптимална дължина на регистър при x86 архитектури е необходимо да бъде анализирано времето за работа при по-голям набор от дължини. В Таблица 3.2 са представени усреднени времена за работа при x86 архитектури. В първите три колони са представени стойностите за n, q и k, съответно. Следващите три колони дават времето за работа съответно за инструкциите SSE4.1, AVX2 и AVX512. От таблицата се вижда, че времето за работа при реализацията чрез AVX2 инструкции и побитово представяне е близко до това на SSE4.1. При побайтово представяне тези инструкции могат да бъдат до 2 пъти по-бързи в зависимост от дължината на кода. Имплементацията чрез AVX512 инструкции може да бъде до 6 пъти по-бърза при побитово представяне и до 3 пъти при побайтово представяне.

Анализ на експерименталните резултати показва, че освен представянето на елементите на полето, при избор на оптимална дължина на регистъра е необходимо да бъде разгледана и каква част от регистъра реално се използва за извършване на изчисленията. За целта може да се въведе понятието *коефициент на използване (Utilization Factor)*, който показва колко бита от регистъра са използвани при изчисленията. Той варира от 0 до 1. При 1 се използва целият регистър, при 1/2 - половината, а при 0 не се използва регистър. Този коефициент зависи от представянето на елементите и дължината на кода, за който се извършват изчисленията. Когато коефициента на използване е по-малък от 1/2 за дадена стойност на n и дължина на регистъра, експерименталните резултати показват, че е по-добре да бъде използван по-малък регистър. Могат да бъдат разгледани следните случаи за избор на дължина на регистър при изчисления с x86 архитектури:

- Тъй като времената за изчисление с SSE и AVX2 са близки, при избор между тях и използване на побитово представяне е по-подходящо да бъдат използвани 128-битови регистри.
- За $\mathbb{F}_2$ координатата на вектор се съхранява в един бит, докато в случая на $\mathbb{F}_4$ се съхранява в два бита. Тогава за $n \leq 64$ е подходящо да бъде използван 128-битов регистър вместо 512-битов регистър, тъй като коефициента на използване е по-малък от 1/2.
- За крайни полета $\mathbb{F}_{3^m}$, $m \geq 1$, елемент от полето се съхранява в $2m$ бита. Тогава за $n \leq 64$ подходящо да бъдат използвано 128-битови регистри. Изключение прави случаят, в който $q = 27$, където при по-малки дъл-

жини се извършват тежки операции за разместване на елементите в регистър при намиране на теглото на вектор. Тогава 512-битови регистри е подходящо да бъдат използвани при $n > 128$.

В останалите случаи е подходящо да бъдат използвани регистрите с най-голяма дължина. При избор на дължина на регистъра е необходимо да се вземе предвид и начина за използване на функциите. Известно е, че при използване на AVX512 в програми, работещи с повече нишки или процеси, работната тактова честота намалява [44]. Тогава при използване на функциите в програми с многоядрени изчисления, е възможно оптималната дължина на регистър да бъде по-малка.

Коментари

Разработените алгоритми и експерименталните резултати, представени в тази глава, са публикувани в [P5]. Резултатите са докладвани на 14-та Международна конференция Large-Scale Scientific Computations [D7].

Таблица 3.2: Време за работа в сек. при x86 архитектури

n	q	k	SSE4.1	AVX2	AVX512
100	2	30	5,87	4,79	3,80
500	2	30	13,69	15,53	6,65
1000	2	30	21,86	26,90	8,89
2000	2	30	39,23	50,59	12,46
100	4	15	1,88	2,10	1,77
500	4	15	8,41	8,74	3,25
1000	4	15	14,23	15,08	4,03
2000	4	15	28,60	24,46	5,41
100	3	20	20,64	13,76	8,21
500	3	20	33,44	37,58	19,07
1000	3	20	52,65	62,39	23,85
2000	3	20	83,31	105,62	30,31
100	9	10	4,93	2,75	2,53
500	9	10	9,78	8,81	4,86
1000	9	10	15,37	15,52	6,02
2000	9	10	27,57	26,22	8,77
100	27	7	4,85	3,80	5,91
500	27	7	12,38	8,77	8,83
1000	27	7	18,79	15,26	11,61
2000	27	7	34,78	28,70	16,92
100	5	12	1,20	1,03	0,75
500	5	12	4,70	3,02	1,81
1000	5	12	8,68	5,96	3,12
2000	5	12	16,17	11,27	5,96
100	25	7	6,73	4,29	3,53
500	25	7	27,17	13,07	9,34
1000	25	7	48,94	25,32	17,51
2000	25	7	92,16	48,71	33,27

Глава 4

Самодопълнителни кодове, достигащи границата на Грей-Ранкин и свързани с тях фамилии от кодове с две и три тегла

В тази глава са разгледани линейни самодопънителни кодове, достигащи границата на Грей-Ранкин, представена в (1.1). Линейните кодове с размерност $k \leq 7$, достигащи равенство в (1.1), и някои семейства от кодове, свързани с тях, са добре изучени. Те са свързани със силнорегулярни графи [23, 33], двойнотегловни кодове [76], квазисиметрични СДП дизайни [53]. Кодове от този тип могат също да се разглеждат като кодове с четни тегла [78] или самоортогонални кодове [18]. Друга важна връзка на тези кодове е с бент функции [26, 34] и векторни бент функции [36]. Параметрите (дължина, размерност, минимално разстояние, тегловен спектър) на линейните самодопълнителните кодове с четна дължина, които достигат равенство в (1.1), са известни [62]. За кодове от този тип с размерност $k > 7$ има само частични резултати. В тази глава са разгледани връзките между производни кодове на кодове, достигащи на границата на Грей-Ранкин. Тези връзки са използвани за конструирането на кодове от по-високи размерности, като в тази глава са разгледани четнотегловни, проективни, линейни самодопълнителни кодове. Дефинирани са шест основни фамилии от дву- и тритегловни линейни кодове и връзките им с раз-

гледаните самодопълнителни кодове. Представени са конструкции, описващи връзките между отделните фамилии от кодове. Благодарение на описаните връзки и конструкции са получени частични класификационни резултати за самодопълнителни кодове, достигащи границата на Грей-Ранкин.

4.1 Подкодове, проективнодопълнителни кодове

4.1.1 Подкодове с размерност $k-1$

Нека разгледаме $[n,k]$ линеен двоичен проективен код C и неговите подкодове с размерност $k-1$. Ако $a \in \mathbb{F}_2^k$, тогава aG е кодова дума в кода C. Следователно всеки вектор от k-мерното векторно пространство $\mathbb{F}_2^k$ дефинира кодова дума в C. Ако разгледаме всички ненулеви вектори на k-мерното векторно пространство като стълбове на $k \times (2^k-1)$ матрица S_k, получаваме пораждаща матрица на добре известния симплексен код $\mathcal{S}_k$. Нека $a=(a_1,a_2,\ldots,a_k) \in \mathbb{F}_2^k$ е ненулев вектор и B_a е множеството от всички решения на уравнението $a_1x_1+a_2x_2+\cdots+a_kx_k=0$. Тогава B_a е линейно подпространство на $\mathbb{F}_2^k$ с размерност $k-1$. Използвайки това подпространство, получаваме подкод на C като

$$C_a=\{bG,\ b \in B_a\},\quad \dim C_a = k-1.$$

Обратно, всеки подкод с размерност $k-1$ може да се разглежда като множеството $\{w=vG,\ v \in \mathbb{F}_2^k\}$, където векторите v са решенията на линейно уравнение с k променливи. По този начин получаваме биекция между множеството от ненулеви вектори в $\mathbb{F}_2^k$ и множеството от всички подпространства на C с размерност $k-1$. Следователно броят на тези подпространства е 2^k-1. Освен това, ако фиксираме $b \in \mathbb{F}_2^k$, можем да го разглеждаме като решение на $2^{k-1}-1$ линейни уравнения с k променливи. От това следва, че всяка кодова дума bG принадлежи към точно $2^{k-1}-1$ подкода с размерност $k-1$. Тъй като C е проективен двоичен код, ефективната дължина на неговия подкод с размерност $k-1$ не може да бъде по-малка от $n-1$. Ако C_i е множеството от всички кодови думи в C, които имат 0 в i-та координата, тогава C_i е подкод на C с размерност $k-1$ и ефективна дължина $n-1$. Така подкодовете C_1, $C_2,\ldots,C_n$ са всички подкодове на C с размерност $k-1$ и ефективна дължина $n-1$. От това следва, че подкодовете с размер $k-1$ и ефективна дължина n са 2^k-1-n. Интересно наблюдение е, че симплексният код е единственият проективен код, за който всички подкодове с размерност $k-1$ имат ефективна дължина $n-1$.

За самодопълнителен код C, броят на подкодовете, съдържащи вектора $\mathbf{1} = (1, \ldots, 1)$ е $2^{k-1} - 1$. Тогава броят на подкодове, които не съдържат вектора $\mathbf{1}$ е $2^k - 1 - (2^{k-1} - 1) = 2^{k-1}$. Ако $n < 2^{k-1}$, то подкодовете на C с ефективна дължина n, които не съдържат вектора $\mathbf{1}$, са $2^{k-1} - n$. Ако $n \geq 2^{k-1}$, то всички подкодове с ефективна дължина n съдържат вектора $\mathbf{1}$. Нека разгледаме кода на Рид-Малер $\mathcal{RM}(1, m)$ с дължината $n = 2^m$ и размерност $k = m + 1$ (следователно $n = 2^{k-1}$). Тогава всичките му $[2^m, m]$ подкодове или съдържат вектора за $\mathbf{1}$, или имат ефективна дължина $n - 1$.

4.1.2 Проективнодопълнителни кодове и самодопълнително еквивалентни кодове

Нека C е проективен $[n, k]$ код с пораждаща матрица G. Стълбовете на матрицата S_k могат да бъдат пренаредени до получаване на матрица $S'_k = (G|\overline{G})$. Матрицата S'_k генерира симплексен код (по-точно, матриците S_k и S'_k генерират еквивалентни кодове като и двата наричаме симплексни кодове). Тогава матрицата $\overline{G}$ генерира код с дължина $2^k - 1 - n$, наречен проективнодопълнителен код на C и означен с $\overline{C}$. Ако C има две ненулеви тегла w_1 и w_2, тогава теглата на $\overline{C}$ са $2^{k-1} - w_1$ и $2^{k-1} - w_2$, тъй като всички кодови думи на симплексния код имат тегло 2^{k-1}. Тъй като разглежданите кодове са проективни и линейни, се знае, че минималното тегло на дуалния код е поне 3. Тогава, за кодове с две и три тегла, могат да бъдат изчислени боят на кодовите думи със съответните тегла A_{w_1} и A_{w_2} като се използва системата от уравнения, извесни като *Pless power moments* [71]. Тази система от уравнения включва биномни коефициенти и числа на Стърлинг и свързва тегловните спектри на C и $C^\perp$. Системата за двутегловни кодове е:

$$\begin{array}{rcrcl} A_{w_1} & + & A_{w_2} & = & 2^k - 1, \\ w_1 A_{w_1} & + & w_2 A_{w_2} & = & 2^{k-1} n. \end{array} \tag{4.1}$$

Могат да бъдат разгледани следните конструкции за дадена стойност на n:

- Нека C е $[n, k-1]$ линеен код, който не е самодопълнителен и $\widehat{C} = C \cup (\mathbf{1} + C)$. Ако G е пораждаща матрица на C, то матрицата $\widehat{G} = \begin{pmatrix} 1 & \ldots & 1 \\ & G & \end{pmatrix}$ е пораждаща матрица на $\widehat{C}$.
- Нека C е $[n-1, k-1]$ линеен код, който не е самодопълнителен и $\widehat{C'} =$

$(0|C) \cup (1|\mathbf{1}+C)$. Ако G е пораждаща матрица на C, то матрицата $\widehat{G'} = \begin{pmatrix} 1 & \dots & 1 \\ & G & 0 \end{pmatrix}$ е пораждаща матрица на $\widehat{C'}$.

Използвайки тези конструкции може да дефинира следната релация на еквивалентност:

Дефиниция 4.1 (Самодопълнително еквивалентни кодове)**.** *Нека разгледаме двоични линейни кодове, които не съдържат вектора* $\mathbf{1}$*.*

- *Ако* C_1 *и* C_2 *са* $[n, k-1]$ *кодове, то те се наричат самодопълнително еквивалентни, ако кодовете* $\widehat{C_1}$ *и* $\widehat{C_2}$ *са еквивалентни самодопълнителни кодове с дължина* n*.*

- *Ако* C_1 *и* C_2 *са съответно с параметри* $[n-1, k-1]$ *и* $[n, k-1]$*, то те са самодопълнително еквивалентни, ако съответните им кодове* $\widehat{C_1'}$ *и* $\widehat{C_2}$ *са еквивалентни самодопълнителни кодове с дължина* n*. Аналогично, ако* C_1 *и* C_2 *са съответно с параметри* $[n, k-1]$ *и* $[n-1, k-1]$*, то те са самодопълнително еквивалентни, ако съответните им самодопълнителни кодове с дължина* n *са еквивалентни.*

- *Ако* C_1 *и* C_2 *са* $[n-1, k-1]$ *кодове, то те са самодопълнително еквивалентни, ако кодовете* $\widehat{C_1'}$ *и* $\widehat{C_2'}$ *са еквивалентни самодопълнителни кодове с дължина* n*.*

Разглеждайки тази релация на еквивалентност, за самодопълнителен код C, всички подкодове, които не съдържат вектора $\mathbf{1}$, са в един клас на самодопълнителна еквивалентност. Концепциите за самодопълнително еквивалентните кодове и проективнодопълнителните кодове се използват за дефиниране на фамилии от линейни двоични кодове, свързани със самодопълнителни кодове, които достигат равенство в (1.1).

4.2 Фамилии от кодове с две и три тегла и връзките им със самодопълнителни кодове, достигащи границата на Грей-Ранкин

Нека C двоичен линеен самодопълнителен код, който достигат равенство в (1.1). За C могат да бъдат дефинирани четири семейства от двоични линейни

кода с две тегла и две семейства от двоични линейни кодове с три тегла. За целта използваме следните означения, където $k = 2s$ и $s \geq 2$:

$$\begin{array}{ll} t_k = 2^{k-2}, & t_{k^\pm} = t_k \pm 2^{s-1}, \\ T_k = 2t_k = 2^{k-1}, & T_{k^\pm} = T_k \pm 2^{s-1}, \\ T_{k+1} = 2^k. & \end{array}$$

За така дефинираните T_k и t_k са в сила следните равенства:

$$T_{k^+} + T_{k^-} = 2^k = T_{k-1}, \tag{4.2}$$

$$t_{k^+} + t_{k^-} = 2^{k-1} = t_{k+1}. \tag{4.3}$$

Могат да бъдат дефинирани фамилии от двутегловни кодове Φ_k и тритегловни кодове Ψ_k, като се използват дадените означения:

- Φ_{k^-} с параметри $[T_{k^-}, k; \{t_{k^-}, t_k\}]$.
- Φ_{k^+} с параметри $[T_{k^+}, k; \{t_k, t_{k^+}\}]$.
- Φ'_{k^-} с параметри $[T_{k^-} - 1, k; \{t_{k^-}, t_k\}]$.
- Φ'_{k^+} с параметри $[T_{k^+} - 1, k; \{t_k, t_{k^+}\}]$.
- Ψ_k с параметри $[2^k, k+1; \{T_{k^-}, 2^{k-1}, T_{k^+}\}]$.
- Ψ'_k с параметри $[2^k - 1, k+1; \{T_{k^-}, 2^{k-1}, T_{k^+}\}]$.

За така дефинираните кодове от фамилиите Φ_k и Ψ_k може да бъдат направени следните наблюдения:

- Тегловните спектри и съответно тегловните функции на кодовете от семействата Φ_k и Ψ_k могат да бъдат пресметнати, като те са:

$$\begin{aligned} \Phi_{k^-} &: \ 1 + T_{k^-} y^{t_{k^-}} + (T_{k^+} - 1) y^{t_k}, \\ \Phi_{k^+} &: \ 1 + (T_{k^-} - 1) y^{t_k} + T_{k^+} y^{t_{k^+}}, \\ \Phi'_{k^-} &: \ 1 + T_{k^+} y^{t_{k^-}} + (T_{k^-} - 1) y^{t_k}, \\ \Phi'_{k^+} &: \ 1 + (T_{k^+} - 1) y^{t_k} + T_{k^-} y^{t_{k^+}}, \\ \Psi_k &: \ 1 + T_{k^-} y^{T_{k^-}} + (2T_k - 1) y^{T_k} + T_{k^+} y^{T_{k^+}}, \\ \Psi'_k &: \ 1 + T_{k^+} y^{T_{k^-}} + (2T_k - 1) y^{T_k} + T_{k^-} y^{T_{k^+}}. \end{aligned}$$

Таблица 4.1: Класификация на кодовете от семействата Φ_k

		$s=2$	$s=3$	$s=4$
Φ_{k^-}	$[n,k;\{w_1,w_2\}]$	$[6,4;\{2,4\}]$	$[28,6;\{12,16\}]$	$[120,8;\{56,64\}]$
	$W(y)$	$1+6y^2+9y^4$	$1+28y^{12}+35y^{16}$	$1+120y^{56}+135y^{64}$
	#	1	7	
Φ_{k^+}	$[n,k;\{w_1,w_2\}]$	$[10,4;\{4,6\}]$	$[36,6;\{16,20\}]$	$[136,8;\{64,72\}]$
	$W(y)$	$1+5y^4+10y^6$	$1+27y^{16}+36y^{20}$	$1+119y^{64}+136y^{72}$
	#	1	5	
Φ'_{k^-}	$[n,k;\{w_1,w_2\}]$	$[5,4;\{2,4\}]$	$[27,6;\{12,16\}]$	$[119,8;\{56,64\}]$
	$W(y)$	$1+10y^2+5y^4$	$1+36y^{12}+27y^{16}$	$1+136y^{56}+119y^{64}$
	#	1	5	
Φ'_{k^+}	$[n,k;\{w_1,w_2\}]$	$[9,4;\{4,6\}]$	$[35,6;\{16,20\}]$	$[135,8;\{64,72\}]$
	$W(y)$	$1+9y^46y^6$	$1+35y^{16}+28y^{20}$	$1+135y^{64}+120y^{72}$
	#	1	7	

Таблица 4.2: Класификация на кодовете от семействата Ψ_k

		Ψ_k	Ψ'_k
$s=2$	$[n,k;\{w_1,w_2,w_3\}]$	$[16,5;\{6,8,10\}]$	$[15,5;\{6,8,10\}]$
	$W(y)$	$1+6y^6+15y^8+10y^{10}$	$1+10y^6+15y^8+6y^{10}$
	#	1	1
$s=3$	$[n,k;\{w_1,w_2,w_3\}]$	$[64,7;\{28,32,36\}]$	$[63,7;\{28,32,36\}]$
	$W(y)$	$1+28y^{28}+63y^{32}+36y^{36}$	$1+28y^{28}+63y^{32}+36y^{36}$
	#	4	4
$s=4$	$[n,k;\{w_1,w_2,w_3\}]$	$[256,9;\{120,128,136\}]$	$[255,9;\{120,128,136\}]$
	$W(y)$	$1+120y^{120}+255y^{128}+136y^{136}$	$1+136y^{120}+255y^{128}+120y^{136}$

- Ако се разгледа код C, който е от някой от класовете Φ_{k^-}, Φ_{k^+} или Ψ_k, то кодът $\widehat{C}=C\cup(\mathbf{1}+C)$ има същите ненулеви тегла като кода C с допълнително тегло n, съответстващо на вектора $\mathbf{1}$, където n е дължината на кода. Аналогично, ако C е код от някое от семействата Φ'_{k^-}, Φ'_{k^+} или Ψ'_k, то кодът $\widehat{C'}=((0,C')\cup(1,\mathbf{1}+C))$ има същите ненулеви тегла като кода C с допълнително тегло n.
- За $s<4$ е известен броя на нееквивалентни кодове за всяка от фамилии. Таблица 4.1 дава параметрите, тегловните разпределения и броя на нееквивалентните кодове от фамилиите Φ_k, докато Таблица 4.2 представя параметрите, тегловните разпределения и броя на нееквивалентните кодове за Ψ_k, като за $s=4$ няма пълна класификация.

Твърдения 4.1 и 4.2 описват връзките между кодове в класовете Φ и Ψ.

Твърдение 4.1. *От код в едно от семействата Φ_{k^-}, Φ_{k^+}, Φ'_{k^-}, Φ'_{k^+} може да се конструират кодове от другите три семейства.*

Доказателство. Нека A е код от семейството Φ_{k^-}. Тогава неговият проективнодопълнителен код $\overline{A}$ има дължина $2^k - 1 - T_{k^-} = T_{k^+} - 1$ и тегла $2^{k-1} - t_k = 2^{k-1} - 2^{k-2} = t_k$ и $2^{k-1} - t_{k^-} = t_{k^+}$. Следователно $\overline{A} \in \Phi'_{k^+}$.

Нека разгледаме кода $\widehat{A} = A \cup (\mathbf{1} + A)$. Той има T_{k^-} подкода с размерност k и ефективна дължина $T_{k^-} - 1$ и нито един от тях не съдържа вектора $\mathbf{1}$. Тъй като $t_k + t_{k^-} = T_{k^-}$, ненулевите тегла в $\widehat{A}$ са t_k, t_{k^-} и T_{k^-} и следователно t_k и t_{k^-} са ненулевите тегла в подкодовете с ефективна дължина $T_{k^-} - 1$. От това следва, че тези подкодове принадлежат към семейството Φ'_{k^-}. Ако $\widehat{A}'$ е един от тези подкодове, то неговият проективнодопълнителен код принадлежи на семейството Φ_{k^+}. Аналогично, за код от Φ_{k^+}, могат да се получат кодове в другите три семейства.

Ако B е код от семейството Φ'_{k^+}, тогава неговият проективнодопълнителен код принадлежи на семейството Φ_{k^-}. Следователно от B могат да се конструират кодове от другите три семейства. Аналогично е доказателството за кодовете в Φ'_{k^-}. □

Твърдение 4.2. *От код от семейството Ψ_k може да се конструира код в Ψ'_k и обратно.*

Доказателство. Нека разгледаме код $C \in \Psi_k$. Тогава подкодовете на $\widehat{C} = C \cup (\mathbf{1} + C)$ с ефективна дължина $2^k - 1$ и размерност $k + 1$ принадлежат към семейството Ψ'_k.

Ако $C' \in \Psi'_k$, може да се разгледа $[2^k, k+2]$ кода $\widehat{C'} = ((0, C') \cup (1, \mathbf{1} + C'))$. $\widehat{C'}$ е самодопълнителен код с ненулеви тегла T_{k^-}, 2^{k-1}, T_{k^+} и 2^k. Броят на неговите подкодове с ефективна дължина 2^k и размерност $k + 1$, които не съдържат вектора $\mathbf{1}$, е $2^{k+1} - 2^k = 2^k$ (раздел 4.1). Тези подкодове принадлежат на семейството Ψ_k. □

Твърдения 4.3 и 4.4 представят връзките между самите класове Φ и Ψ.

Твърдение 4.3. *Нека $A \in \Phi_{k^\pm}$ е код с пораждаща матрица G_A и $G_{\overline{A}}$ е пораждаща матрица на неговия проективнодопълнителен код. Тогава матрицата*

$$\hat{G}_A = \begin{pmatrix} 1 \dots 1 & 0 \dots 0 \\ G_A & G_{\overline{A}} \end{pmatrix}$$

генерира код в семейството Ψ'_k.

Доказателство. Очевидно кодът с пораждаща матрица от вида $\hat{G}_A$, има същата дължина като симплексния код $\mathcal{S}_k$, а именно $2^k - 1$. Тъй като $(G_A|G_{\overline{A}})$ симплексния код с размерност k, кодът $\langle (G_A|G_{\overline{A}}) \rangle$ има само едно ненулево тегло, а именно 2^{k-1}. Ако v е кодова дума в този код, тогава $v = (v_1, v_2)$, където $v_1 \in A$, $v_2 \in \overline{A}$. Ако $\mathrm{wt}(v_1) = t_k$ тогава $\mathrm{wt}(v_2) = 2^{k-1} - t_k = t_k$ и следователно $\mathrm{wt}(\mathbf{1} + v_1, v_2) = T_{k^\pm} - t_k + t_k = T_{k^\pm}$. Ако $\mathrm{wt}(v_1) = t_{k^\pm}$, то $\mathrm{wt}(v_2) = 2^{k-1} - t_{k^\pm} = t_{k^\mp}$ и $\mathrm{wt}(\mathbf{1} + v_1, v_2) = T_{k^\pm} - t_{k^\pm} + t_{k^\mp} = t_k + t_{k^\mp} = T_{k^\mp}$. Следователно кодът $\hat{A}$ има следните ненулеви тегла: $2^{k-1} = T_k$, T_{k^+} и T_{k^-}. Следователно $\hat{A} \in \Psi'_k$. □

Твърдение 4.4. *Нека $A' \in \Phi'_{k^\pm}$ с пораждаща матрица $G_{A'}$ и $G_{\overline{A'}}$ е пораждащата матрица на проективнидопълнителния код. Тогава матрицата*

$$\hat{G}_A = \begin{pmatrix} 1 & 1 \dots 1 & 0 \dots 0 \\ 0 & & \\ \vdots & G_{A'} & G_{\overline{A'}} \\ 0 & & \end{pmatrix}$$

е пораждаща матрица на код в Ψ_k.

Доказателство. Доказателството на това твърдение е аналогично на доказателството на Твърдение 4.3. Дължината на кода с пораждаща матрица $\hat{G}_A$ е 2^k. Този код може да се означи с $\hat{A}$. Тъй като $(G_{A'}|G_{\overline{A'}})$ поражда симплексния код с размерност k, той има само едно ненулево тегло, а именно 2^{k-1}. Ако $v \in \langle (G_{A'}|G_{\overline{A'}}) \rangle$, тогава $v = (v_1, v_2)$, където $v_1 \in A'$, $v_2 \in \overline{A'}$. Ако $\mathrm{wt}(v_1) = t_k$ тогава $\mathrm{wt}(v_2) = 2^{k-1} - t_k = t_k$ и така $\mathrm{wt}(1, \mathbf{1} + v_1, v_2) = 1 + T_{k^\pm} - 1 - t_k + t_k = T_{k^\pm}$. Ако $\mathrm{wt}(v_1) = t_{k^\pm}$ тогава $\mathrm{wt}(v_2) = 2^{k-1} - t_{k^\pm} = t_{k^\mp}$ и така $\mathrm{wt}(1, \mathbf{1} + v_1, v_2) = 1 + T_{k^\pm} - 1 - t_{k^\pm} + t_{k^\mp} = t_k + t_{k^\mp} = T_{k^\mp}$. Следователно кодът $\hat{A}$ има следните ненулеви тегла $2^{k-1} = T_k$, T_{k^+} и T_{k^-}. От това следва, че $\hat{A} \in \Psi_k$. □

Кодовете от фамилиите Φ_k и Φ'_k, които са получени чрез конструкциите, описани в Твърдения 4.3 и 4.4, съдържат като подкод симплексния код. Те могат да бъдат означени с Ψ_{S_k} и Ψ'_{S_k}.

Теорема 4.5. *Ако C е код от семейството $\Phi_{k^\pm}$ (или $\Phi'_{k^\pm}$), тогава неговият остатъчен код по отношение на кодова дума с тегло $t_{k^\pm}$ принадлежи към семейството Ψ_k (съответно Ψ'_k).*

Доказателство. Ако $C \in \Phi_{k^-}$ тогава $d = t_{k^-}$. Следователно размерността на разглеждания остатъчен код е $k-1$. В другия случай, ако $C \in \Phi_{k^+}$, $t_{k^+} = 2^{k-2}+2^{s-1} < 2.2^{k-2} = 2d$, то от теорема 1.1 може да се докаже, че размерността на остатъчния код по отношение на кодова дума с тегло t_{k^+} също е $k-1$.

Да разгледаме теглата на тези кодове и да допуснем $c \in C$, $\mathrm{wt}(c) = t_{k^\pm}$ и $c = (\underbrace{11\ldots1}_{t_{k^\pm}}\underbrace{00\ldots0}_{t_k})$. За остатъчния код $Res(C,c)$, ако $v = (v_1, v_2) \in \Phi_{k^\pm}$, $v_1 \in \mathbb{F}_2^{t_{k^\pm}}$, $v_2 \in \mathbb{F}_2^{t_k}$, то $v_2 \in Res(C,c)$ и

$$\begin{aligned} \mathrm{wt}(v) = \mathrm{wt}(v_1) + \mathrm{wt}(v_2) &= t_k \text{ или } t_{k^\pm}, \\ \mathrm{wt}(c+v) = \mathrm{wt}(\mathbf{1}+v_1) + \mathrm{wt}(v_2) = t_{k^\pm} - \mathrm{wt}(v_1) + \mathrm{wt}(v_2) &= t_k \text{ или } t_{k^\pm}. \end{aligned}$$

От това следва, че

$$\begin{aligned} \mathrm{wt}(v_1) + \mathrm{wt}(v_2) &= t_k \text{ или } t_{k^\pm}, \\ -\mathrm{wt}(v_1) + \mathrm{wt}(v_2) &= 0 \text{ или } \mp 2^{s-1}. \end{aligned}$$

Следователно $\mathrm{wt}(v_2) = t_{(k-2)}, t_{(k-2)^-}$ или $t_{(k-2)^+}$. Тогава остатъчният код $Res(C,c)$ има дължина $T_{k^\pm} - T_{k^\pm} = 2^{k-2}$, размерност $k-1$ и тегла $t_{(k-2)}$, $t_{(k-2)^-}$ и $t_{(k-2)^+}$. Следователно $Res(C,c) \in \Psi_k$. □

Теорема 4.6. *Ако C е код от семейството Ψ_k (съответно Ψ'_k), тогава неговият остатъчен код по отношение на кодова дума с има най-много пет ненулеви тегла. Ако $C \in \Psi_{S_k}$ (съответно $C \in \Psi'_{S_k}$) и $v \in C$ има тегло $T_{k^\pm}$, тогава $Res(C,v) \in \Phi_{k^\mp}$ (съответно $\Phi'_{k^\mp}$).*

Доказателство. Тъй като кодовете в семействата Ψ_k и Ψ'_k са с тегла, кратни на 2^{s-1}, от Лема 1.1 следва, че техните остатъчни кодове са с тегла кратни на 2^{s-2}. Освен това минималното тегло на $Res(C,c)$ е поне $T_{k^-}/2 = 2^{k-2} - 2^{s-2}$, а максималното тегло е най-много $2^{k-2} + 3.2^{s-2}$. В този интервал има точно пет цели числа $[2^{k-2} - 2^{s-2}, 2^{k-2} + 3.2^{s-2}]$, които се делят на 2^{s-2}.

Ако $C \in \Psi'_{S_k}$, то $C = \mathcal{S}_k \cup (v + \mathcal{S}_k)$ и произволна кодова дума в съседния клас $v + \mathcal{S}_k$ има тегло T_{k^+} или T_{k^-}. Освен това, всички кодови думи с тегло $T_{k^\pm}$ принадлежат към съседния клас $v + \mathcal{S}_k$. Ако $y \in Res(C,v)$, тогава $(x,y) \in \mathcal{S}_k$

и $v + (x, y) \in v + \mathcal{S}_k$ за подходящ вектор x, аналогично на доказателството на Лема 1.1, може да се разгледа v като $v = (\underbrace{11\ldots1}_{T_{k^\pm}}\underbrace{00\ldots0}_{T_{k^\mp}})$. Следователно

$$\begin{aligned} \mathrm{wt}(x) + \mathrm{wt}(y) &= 2^{k-1}, \\ -\mathrm{wt}(x) + \mathrm{wt}(y) &= T_{k^-} - T_{k^\mp} \text{ или } T_{k^+} - T_{k^\mp}. \end{aligned}$$

От това следва, че $\mathrm{wt}(y) = t_{k^-}$ или t_k, ако $\mathrm{wt}(v) = T_{k^+}$ и $\mathrm{wt}(y) = t_k$ или t_{k^+}, ако $\mathrm{wt}(v) = T_{k^-}$. Следователно остатъчните кодове на $C \in \Psi'_{S_k}$ по отношение на кодова дума с тегло T_{k^-}, принадлежат на Φ'_{k^+}, и остатъчните кодове по отношение на кодова дума с тегло T_{k^+} принадлежат на Φ'_{k^-}. □

4.3 Конструкции за построяване на кодове от семействата Φ_{k+2} чрез кодове с размерност k

За кодовете от Φ_k може да се разгледат конструкции, които позволяват построяването на кодове от размерност $k+2$ като се използват кодове с размерност k. Чрез тази конструкция може да се дефинира безкрайна фамилия от кодове. Твърдение 4.7 представя подход за конструиране на кодове от $\Phi'_{(k+2)^\pm}$ чрез кодове от $\Phi_{k^\pm}$. Твърдение 4.8 представя аналогична конструкция за построяване на кодове от $\Phi_{(+2)^\pm}$ чрез кодове от $\Phi'_{k^\pm}$. Тази конструкции са означени с общото наименование *Self-complementary Lifting (SCL)* конструкция.

Твърдение 4.7. *Нека $A \in \Phi_{k^\pm}$ е код с пораждаща матрица G_A и B е код с пораждаща матрица*

$$G_B = \begin{pmatrix} G_A & G_A & G_A & \overline{G_A} \\ 1\ldots1 & 1\ldots1 & 0\ldots0 & 0\ldots0 \\ 0\ldots0 & 1\ldots1 & 1\ldots1 & 0\ldots0 \end{pmatrix}.$$

Тогава $B \in \Phi'_{(k+2)^\pm}$.

Доказателство. Очевидно B е проективен код с дължина $2T_{k^\pm} + 2^k - 1 = 2(2^{k-1} \pm 2^{s-1}) + 2^k - 1 = 2^{k+1} \pm 2^s - 1 = T_{(k+2)^\pm} - 1$. За теглата на кодовите думи в B имаме, че ако $v \in B$, то $v = (w, w, w, \overline{w})$, $(\mathbf{1}+w, \mathbf{1}+w, w, \overline{w})$, $(w, \mathbf{1}+w, \mathbf{1}+w, \overline{w})$ или $(\mathbf{1}+w, w, \mathbf{1}+w, \overline{w})$, където $w \in A$ и $(w, \overline{w}) \in \mathcal{S}_k$. Тъй като

$$\mathrm{wt}(\mathbf{1}+w, \mathbf{1}+w, w, \overline{w}) = \mathrm{wt}(w, \mathbf{1}+w, \mathbf{1}+w, \overline{w}) = \mathrm{wt}(\mathbf{1}+w, w, \mathbf{1}+w, \overline{w}),$$

е необходимо да бъдат изчислени само теглата на $(w, w, w, \overline{w})$ и $(\mathbf{1}+w, \mathbf{1}+w, w, \overline{w})$. За първият вектор е в сила $\mathrm{wt}(w, w, w, \overline{w}) = 2\mathrm{wt}(w) + 2^{k-1}$. Следователно:

$$\mathrm{wt}(w, w, w, \overline{w}) = \begin{cases} 2t_k + 2^{k-1} = 2^k = t_{k+2}, & \text{ако } \mathrm{wt}(w) = t_k, \\ 2t_{k^\pm} + 2^{k-1} = 2^k \pm 2^s = t_{(k+2)^\pm}, & \text{ако } \mathrm{wt}(w) = t_{k^\pm}. \end{cases}$$

За първият вектор е в сила

$$\begin{aligned} \mathrm{wt}(\mathbf{1}+w, \mathbf{1}+w, w, \overline{w}) &= 2(T_{k^\pm} - \mathrm{wt}(w)) + 2^{k-1} = 2^k \pm 2^s + 2^{k-1} - 2\mathrm{wt}(w) \\ &= 3.2^{k-1} \pm 2^s - 2\mathrm{wt}(w). \end{aligned}$$

Следователно:

$$\mathrm{wt}(\mathbf{1}+w, \mathbf{1}+w, w, \overline{w}) = \begin{cases} 3.2^{k-1} \pm 2^s - 2t_k = 2^k \pm 2^s = t_{(k+2)^\pm}, & \text{ако } \mathrm{wt}(w) = t_k \\ 3.2^{k-1} \pm 2^s - 2t_{k^\pm} = 2^k \pm 2^s = t_{(k+2)^\pm}, & \text{ако } \mathrm{wt}(w) = t_{k^\pm} \end{cases}$$

Тогава,

$$\mathrm{wt}(\mathbf{1}, \mathbf{1}, \mathbf{0}, \mathbf{0}) = \mathrm{wt}(\mathbf{1}, \mathbf{0}, \mathbf{1}, \mathbf{0}) = \mathrm{wt}(\mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{0}) = 2^k = t_{k+2}.$$

Следователно ненулевите тегла в B са t_{k+2} и $t_{(k+2)^\pm}$. От това следва, че $B \in \Phi'_{(k+2)^\pm}$. □

Твърдение 4.8. *Нека $A \in \Phi'_{k^\pm}$ е код с пораждаща матрица G_A и B е код с пораждаща матрица*

$$G_B = \begin{pmatrix} G_A\ 0 & G_A\ 0 & G_A\ 0 & \overline{G_A} \\ 1 \ldots 1 & 1 \ldots 1 & 0 \ldots 0 & 0 \ldots 0 \\ 0 \ldots 0 & 1 \ldots 1 & 1 \ldots 1 & 0 \ldots 0 \end{pmatrix}.$$

Тогава $B \in \Phi_{(k+2)^\pm}$.

Доказателство. Тук може да се приложи същият подход като този в доказателството на Твърдение 4.7. Кодовете в $\Phi_{(k+2)^\pm}$имат параметри $[T_{(k+2)^\pm}, k+2; \{2^{2s}, 2^{2s} \pm 2^s\}]$. Тъй като $A \in \Phi'_{k^\pm}$ матрицата G_A може да бъде разширена с нулев стълб, за да се получи необходимата дължина. Това не променя теглата. Тъй като добавяме векторите $(\mathbf{1}, \mathbf{1}, \mathbf{0}, \mathbf{0})$ и $(\mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{0})$ към пораждащата матрица, B е проективен код. Останалите изчисления са аналогични на тези в доказателството на Твърдение 4.7. Тогава за $A \in \Phi'_{k^+}$ резултатният код B принадлежи на $\Phi_{(k+2)^+}$ и за код $A \in \Phi'_{k^-}$, кодът B принадлежи на $\Phi_{(k+2)^-}$. □

Теорема 4.9. *От всеки код $C \in \Phi_k$ могат да бъдат конструирани кодове в Φ_{k+2l} за всички цели числа $l \geq 1$.*

Доказателство. Следва директно от Твърдения 4.1, 4.7 и 4.8. □

Тази конструкция е приложена към кодовете с параметри $[119, 8; \{56, 64\}]$, за да се получат кодове с параметри $[496, 10; \{240, 256\}]$ и след това към кодове с параметри $[2015, 12; \{992, 1024\}]$. В тези случаи нееквивалентните стартови кодове водят до нееквивалентни.

4.4 Изчислителни резултати

Дефинираните в 4.2 фамилии от кодове с две и три тегла и връзките между тях могат да бъдат използвани за получаване на нови класификационни резултати за самодопълнителни кодове, достигащи равенство в (1.1). Нека да разгледаме самодопълнителните кодове с параметри $[28, 7; \{12, 16, 28\}]$. Съществуват точно четири нееквиваленти самодопълнителни кода, които достигат границата [37, 67]. Следователно техните подкодове с размерност 6 са разделени в четири класа на самодопълнителна еквивалентност. Получените подкодове с размерност 6 и ефективни дължини 27 и 28 са представени в Таблица 4.3. Първата колона в таблицата показва пораждаща матрица на самодопълнителен код. Втората и третата колона представят броя на нееквивалентните $[27, 6; \{12, 16\}]$ и $[28, 6; \{12, 16\}]$ кодове, които са подкодове на съответния самодопълнителен код. Очевидно е, че кодовете с дължа 28 са от семейството Φ_{k^-}, а кодовете с дължина 27 са от семейството Φ'_{k^-}.

Теореми 4.5 и 4.6 показват връзките на кодовете от класовете Φ_k и Ψ_k, базирани на остатъчни кодове. Използването на вече класифицирани остатъчни кодове е подходящо в случаите, когато се класифицират линейни кодове със значително по-голяма дължина от размерността. В повечето случаи се използват остатъчните кодове по отношение на кодова дума с минимално тегло. Това е така, защото в този случай известната част от матриците, които ще бъдат конструирани, е най-голямата за дадените параметри.

За класификация на самодопълнителни $[120, 9; \{56, 64, 120\}]$ кодове, които достигат (1.1), може да бъде използван алгоритъм за разширяване на остатъчни кодове [15]. За кодовете с параметри $[119, 8; \{56, 64\}]$ от Φ_{8^-} с е известно, че техните остатъчни кодове, относно кодова дума с тегло 56 имат параметри $[63, 7; \{23, 32, 36\}]$, които принадлежат на семейството Ψ'_6. Ако се раз-

гледат остатъчните на тези кодове, ще бъдат получени кодове с параметри $[35, 6; \{14, 16, 18, 20, 22\}]$. Тези кодове могат да бъдат класифицирани, като се използва [15]. Получени са 267370 нееквивалентни кода, от които 7 са от класа Φ'_{6^-} с параметри $[35, 6; \{16, 20\}]$. Тези кодове могат да бъдат разширени до кодове с параметри $[63, 7; \{28, 32, 36\}]$, като са получени 3220339 нееквивалентни кода. От тях могат да бъдат разгледани нееквивалентните кодове от фамилията Ψ'_{S_6}, които са точно 4. Тези кодове могат да бъдат получени и от четири представителя на класовене на самодопълнителна еквивалентност, дадени в Таблица 4.3. След последващо разширение на четирите кода от Ψ'_{S_6}, могат да бъдат получени кодове с параметри $[119, 8; \{56, 64\}]$.

Таблица 4.4 представя получените резултати. Във втората колона са дадени пораждащи матрици за тези четири $[63, 7; \{28, 32, 36\}]$ кода. Третата колона съдържа броя на еквивалентните $[119, 8; \{56, 64\}]$ кодове, конструирани от матрицата в същия ред. Общият брой на конструираните $[119, 8; \{56, 64\}]$ кодове е 91397, като 91337 от тях са нееквивалентни. Добавяйки нулев стълб и след това вектора $\mathbf{1}$ като ред към пораждащите матрици на всички тези кодове, се получават точно 2946 нееквивалентни самодопълнителни $[120, 9; \{56, 64, 120\}]$ кода. Тези 2946 кода съдържат, като подкодове на размерност 8, точно 175213 кода с ефективна дължина 120 и 156763 кода с ефективна дължина 119. Следователно множеството от всички нееквивалентни 331976 подкодове с размерност 8 е разделен на 2946 класа на самодопълнителна еквивалентност.

Коментари

Резултатите, представени в тази глава, са публикувани в [P4] и докладвани на Национален семинар по теория на кодирането "Проф. Стефан Додунеков"[D5].

Таблица 4.3: Самодопълнителни $[28, 7; \{12, 16, 28\}]$ кодове

G	$[27, 6; \{12, 16\}]$	$[28, 6; \{12, 16\}]$	Общо:
$\begin{pmatrix} 1111111111111111111111111111 \\ 0000000111111111111111100000 \\ 0011111000000000111111010000 \\ 0100011000001111001111001000 \\ 1001100000110011110011000100 \\ 0110001011010001011101000010 \\ 1010111101000101000101000001 \end{pmatrix}$	1	1	2
$\begin{pmatrix} 1111111111111111111111111111 \\ 0000000111111111111111100000 \\ 1111111000000011111111010000 \\ 0000111000011100011111001000 \\ 0001011001100101100111000100 \\ 0110001010001110111001000010 \\ 1010001100100111101010000001 \end{pmatrix}$	1	2	3
$\begin{pmatrix} 1111111111111111111111111111 \\ 0000000111111111111111100000 \\ 1111111000000011111111010000 \\ 0000111000011100011111001000 \\ 0001011001100101100111000100 \\ 0110001010001110111001000010 \\ 1010001110100001101011000001 \end{pmatrix}$	2	2	4
$\begin{pmatrix} 1111111111111111111111111111 \\ 0000000111111111111111100000 \\ 1111111000000011111111010000 \\ 0000111000011100011111001000 \\ 0011001001100111100011000100 \\ 0100011010101001100111000010 \\ 1001100100011010101101000001 \end{pmatrix}$	1	2	3

Таблица 4.4: [119,8;{56,64}] конструирани от остатъчни кодове

No	[63,7;{28,32,36}]	[119,8;{56,64}]
1	$\begin{pmatrix} 11111111111111111111111111111100000000000000000000000000000000000000000 \\ 00000000000000001111111111111000000000000111111111111111111111100000 \\ 00000000011111100000011111100111111111000000000011111111111010000 \\ 00000111100001100001100111010000011110000011110000111111001000 \\ 00011000101110000110101001100000110111001110011001100011100010 \\ 01100001010110001011010010100011001011010110101010100101100001 \\ 10101010000010111000011101001101010001100010011010101110100000 \end{pmatrix}$	63836
2	$\begin{pmatrix} 11111111111111111111111111111100000000000000000000000000000000000000000 \\ 00000000000000001111111111111000000000000111111111111111111111100000 \\ 00000000011111100000011111100111111111000000000011111111111010000 \\ 00000111100001100001100111010000011110000011110000111111001000 \\ 00011000101110000110101001100000110111001110011001100011100010 \\ 01100001010110001011010010100011001011010110101010100101100001 \\ 10101010000010101001001110101101010001101010010101101100000 \end{pmatrix}$	4406
3	$\begin{pmatrix} 11111111111111111111111111111100000000000000000000000000000000000000000 \\ 00000000000000001111111111111000000000000111111111111111111111100000 \\ 00000000011111100000011111100111111111000000000011111111111010000 \\ 00000111100001100001100111010000011110000011110000111111001000 \\ 00111000100010100010101011101001110001001110001011100011100010 \\ 01001001001110001101010001100100101110101100111001001011000010 \\ 10010010010110010101010010100100101011101010101100101010000001 \end{pmatrix}$	23075
4	$\begin{pmatrix} 11111111111111111111111111111100000000000000000000000000000000000000000 \\ 00000000000000001111111111111000000000000111111111111111111111100000 \\ 00000000011111100000011111100111111111000000000011111111111010000 \\ 00000111100001100001100111010000011110000011110000111111001000 \\ 00111000100010100010101011101001110001001110001011100011100010 \\ 11001001000100100100101101101110010010110010010101101100010 \\ 01011011101000101000110000101010110111010110111111011010100000001 \end{pmatrix}$	80
	Общо:	91397
	Нееквивалентни:	91337

Глава 5

Библиотека за изчисление на тегловни инварианти

Разработените алгоритми в Глава 2 са включени в библиотека за намиране на тегловни инварианти на линейни кодове **LinCodeWeightInv** над поле $\mathbb{F}_q$, където $q \leq 64$. Разработената статична библиотека съдържа основни функционалности за намиране на някои тегловни инварианти, базирани на намирането на тегловния спектър на кода. За работа с библиотеката са създадени и два основни модула - интерфейсна програма, която има за цел да улесни използването на библиотеката и модул за тестване на бързодействието на разработената библиотека. Създадено е подробно ръководство за използването на библиотеката, което включва описание на основните ѝ компоненти и начините за компилиране, инсталиране и тестване. За разработването на библиотеката са изучени и използвани софтуерни продукти, предназначени за разработването на софтуер с езиците C/C++. В Раздел 5.1 са описани основните интерфейсни функции, предоставени на крайния потребител, както и начините за препредаване на входните данни. Раздел 5.2 описва двата основни модула, разработени за работа с библиотеката - интерфейсен модул и модул за тестване и верификация. Раздел 5.3 представя използвания софтуер при разработката на библиотеката и описва начините за компилиране и инсталиране на самата библиотека.

5.1 Основни функционалности

Основната функционалност на разработената библиотека се състои в намирането на тегловен спектър на линеен код над поле $\mathbb{F}_q$, където $q \leq 64$. За целта са

използвани разработените алгоритми, представени в Глава 2. Тези алгоритми са реализирани като са използвани различни набори от инструкции и могат да бъдат изпълнени на централни процесори с ARM и x86 архитектури. Библиотеката включва следните шест основни функции:

- Изчисление на тегловен спектър на линеен код.
- Намиране на минималното разстояние на линеен код.
- Намиране на броя на кодовите думи с дадено тегло w.
- Намиране на броя на кодовите думи с тегло по-малко от w.
- Определяне дали съществуват кодови думи с дадено тегло w.
- Определяне дали съществуват кодови думи с тегло по-малко от дадено w.

Основните данни, необходими за изчисленията, са параметрите n, k и q и пораждаща матрица на кода. За всяка от интерфейсните функции са създадени следните подходи за въвеждане на данните:

- Четене на входните данни от файл. Функцията приема като параметър масив от елементи тип *char*, представляващ името на файла. Изчисленията се извършват за всички кодове, записани във файла чрез пораждащи матрици. Входните данни трябва да бъдат записани във файла като се използва следната структура - символ, показващ дали елементите на полето са записани чрез адитивен (изпозва се символа *"?"*) или чрез мултипликативен запис (изпозва се символа *"!"*), стойности за параметрите k, n, q и пореден номер на кода, за който се извършват изчисленията. След тези параметри на нов ред е записана пораждаща матрица на кода. Резултатите се записват във изходен файл с подходящо наименование спрямо интерфейсната функция.
- Генериране на псевдослучаен код по зададени стойности на k, n, q. Функцията приема като параметри стойности за k, n, q, като за тях се генерира пораждаща матрица. Пораждащата матрица се записва в текстов файл *"EXAM"*.
- Изчисление за записана пораждаща матрица в паметта като двумерен масив. Функцията получава като параметри динамичен двумерен масив от тип *int*, параметрите на кода n, k, q и булева променлива, която показва дали елементите на полето в пораждащата матрица са записани чрез

адитивен или мултипликативен запис (променливата има стойност *true*, ако е използван мултипликативен запис).

Функциите за намиране и изброяване на кодовите думи с дадено тегло също така получават като параметър стойността на търсеното тегло, докато функциите за изброяване имат допълнителен параметър, който показва дали кодовите думи да бъдат записани в текстов файл. За същинските изчисления са имплементирани версии на алгоритмите от ниско и високо ниво в зависимост от интерфейсната функция, която ги използва (функциите за търсене приключват работа при намиране на кодова дума с даденото тегло и е възможно да не изчислят целия спектър), броя на елементите в полето и дължината на кода. Изходните данни на функциите зависят от интерфейсната функция. За функциите за намиране на спектър, резултата е записан в глобална променлива *weights*, която съдържа тегловният спектър на кода. Функциите за намиране на минимално разстояние и брой на кодови думи с дадено тегло връщат като резултат цяло число. Функциите за търсене връщат булева стойност, показваща дали са намерени кодови думи с даденото тегло. Когато входните данни се четат от файл, всички интерфейсни функции записват резултата в текстов файл с подходящо име.

Представяне на данните

Входните данни за всички функции могат да бъдат прочетени от текстов файл с определена структура, в динамична матрица, заредена в паметта на програмата или да бъде създадена псевдослучайна матрица по зададени от терминала стойности за n, k и q. За записване на пораждащата матрица в паметта се използва представяне на елементите на дадено поле като цели числа. Когато се разглежда код над просто поле $\mathbb{F}_p$, елементите могат да бъдат представени като остатъците от делението на p (елементите на полето са целите числа от 0 до $(p-1)$). Когато $q = p^m$, $m > 1$, елементите могат да бъдат представени като полиноми от степен по-малка то m и коефициенти от простото поле F_p. Съществува естествено представяне на елементите на всяко крайно поле като пореден номер (или индекс), който позволява лесно индексиране в компютърна памет като цяло число. При съставно поле, индексът на всеки елемент се изчислява по формулата

$$index = \sum_{i=0}^{m-1} \alpha_i p^i, \tag{5.1}$$

където α_i е коефициента в полинома пред x^i.

Елементите на полето могат също да бъдат представени като 0 и степени на примитивен елемент на полето. Следователно могат да се разгледат три различни представяния на елементите на едно съставно поле – адитивно, мултипликативно и като пореден номер (индекс). Адитивното представяне запазва коефициентите на полинома като m-мерен вектор върху простото поле. Това представяне се използва за основните изчисления и оптимизации. Мултипликативният запис се представя от степените на примитивните елементи, докато индексирането е представено с $index$. Записването като пореден номер и в мултипликативната форма се използват за въвеждане на пораждащата матрица.

За оптимизациите при съставни полета (Алгоритъм 4) се използва множеството от пораждащи матрици $M = \{G, \alpha G, \ldots, \alpha^{m-1}G\}$. Изчислението на множеството M се извършва еднократно при първоначална инициализация за дадения код, като се използва представянето на елементите на полето като полиноми. Пораждащата матрица (или множеството M) се записват в статични променливи като се използва подходящ тип от данни за използваното побитово или побайтово представяне.

5.2 Интерфейсна програма и тестване

За работа с библиотеката е разработена интерфейсна програма, която предоставя възможност за работа с всички интерфейсни функции. В нея са включени два подхода за въвеждане на входните данни - чрез четене от файл или чрез генериране на пораждаща матрица по зададени стойности на n, k, q. След избор на метода за въвеждане на данни от командния ред се чете името на файла или стойностите на параметрите на кода. Следващото меню дава възможност за избор на интерфейсна функция, като резултата се изписва в терминала. При въвеждане на грешни данни или стойности (задаване се име на несъщесвуващ файл и др.) се изписва подходящо съобщение.

В главното меню освен две опции за избор на метод за въвеждане на данните, може да се избере и използване на модула за тестване и верификация. Този модул позволява тестване на всички функционалности за коректност на изчисленията и времето за изпълнението им. За целта се използва предварително подготвен входен файл съдържащ линейни кодове и съответните тегловни спектри във вида $\{A_i\}$, където $i = 0, \ldots, n$, генериран с различен софтуер. За тестване на библиотеката са създадени два файла, съдържащи по 100 линейни кода за всяко възможно поле с $q \leq 64$ (стойностите за n и k са еднакви за фиксирано q), зададени чрез техните пораждащи матрици и тегловни спектри.

Изчисленията се извършват последователно за записаните кодове във входния файл и приключват при получаване на грешен резултат или при достигане на края на файла. В изходния файл *Results* се записва средно време за изпълнение на изчисленията за всяко поле при получаване на правилен резултат. При възникване на грешка в изчисленията се изписва подходящо съобщение в терминала и в текстов файл *error.txt*.

При стартиране на модула за тестване и верификация първо автоматично се определя целевата архитектура, върху която се изпълнява програмата. Спрямо нея се задава избор на множество от инструкции за изпълняване на изчисленията или извършване на изчисления без векторизация. При x86 архитектури се предоставят възможности за избор на SSE4.1, AVX2 или AVX512. Ако някой от наборите не е наличен се изписва подходящо съобщение. При изпълнение върху ARM архитектура са възможни използването на NEON инструкции или изпълнение без векторизация. При избор на изпълнение на изчисленията и при двете архитектури следва да бъде въведено името на входния файл. За тестване на програмата са създадени два тестови файла - *TestDataSmall* и *TestDataBig*, които съдържат съответно кодове с по-малки или по-големи размерности. Следователно времето за изпълнение при *TestDataBig* е значително повече.

5.3 Използван софтуер

CMake

До момента бяха представени основните инструкции за векторизация на алгоритми в архитектурите x86 и ARM за централни процесори. Представените алгоритми в Глава 2 са имплементирани с инструкции от семействата SSE, AVX и AVX512 и NEON инструкции за централни процесори с ARM архитектура. Разработените имплементации са използвани за разработването на оптимизирана библиотека за намирането на тегловни характеристики на линейни кодове, базирани на изчислението на тегловния спектър. За разработването на библиотеката са изучени различни софтуерни продукти за разработване и документация на преносим софтуер (CMake, Doxygen). CMake е продукт, който позволява генерирането на проекти за различни интегрирани системи за разработка от едни и същи изходни файлове. Разработен е за повечето основни операционни системи (OSX, Ubuntu, Windows) и придружаващите ги IDE (Xcode, Codeblocks, Visual Studio и др.) и системи за компилиране (Make). CMake дава възможност да се избере една от наличните интегрирани системи за целева-

та операционна система. Основно предимство е възможността за промяна на компилатора, когато е възможно, задаване на флагове на компилатора и управление на други параметри чрез кеширани променливи на програмата. CMake има свой собствен скриптов език, който се използва за контролиране на тези променливи и описване на структурата на целевия софтуер. Той също така може да уведоми, ако липсва ключов компонент за компилацията, като например компилатор или основна библиотека. Може да се използва за уеднаквяване на параметрите при компилация на даден софтуер за различни операционни системи. Някои от основните възможности на платформата са:

- Софтуера е разработен за да бъде свободно достъпен за най-популярните операционни системи. Автоматично се разпознава платформата и наличните необходими компилатори.
- CMake използва собствен език за създаването на проект. Този език разполага със собствени команди и променливи, нужни за описанието на платформата и проекта.
- В текстов файл се описва структурата на проекта и външните библиотеки, необходими за компилирането. Така се изяснява процеса на компилиране и свързване на отделните части на проекта, като същевременно се придобиват умения за правилно моделиране на един софтуерен продукт.
- Дава се възможност за копирането на изпълнимите файлове в нови директории, което позволява да се правят независими промени и да се контролират версиите на разработения софтуер.
- Възможно е избирането на конкретен компилатор, при наличието на няколко такива, като лесно се добавят необходими флагове за компилиране в зависимост от операционната система и избрания компилатор. В резултатния проект, това са допълнителни настройки, за които няма да бъде необходима допълнителна конфигурация на средата.
- Могат да се добавят задължителни компоненти, необходими за паралелно изпълнение, като CMake автоматично открива необходимите файлове.
- Могат да бъдат създадени тестови скриптове, които заедно с предварително зададени входни данни и очаквани изходни резултати, проверяват поведението на ключови компоненти (функции, библиотеки, класове и др.) на разработвания софтуер. Това позволява откриване на грешки при добавяне на нови функционалности или промени в началните (source) файлове за компилация. Тази функционалност на платформата също та-

ка позволява използването на разработка, управлявана от тестове (test driven development).

Doxygen

Doxygen е софтуер, предназначен за генериране на документация за софтуерни проекти. Този софтуер създава подробно описание на целева програма във формата на LaTeXили HTML файлове. За генерирането на документацията на целева програма се анализира дървото на програмния код на програмата, създадена чрез езици от високо ниво с синтаксис, сходен на езиците C/C++(C#, Objective-C, Java, PHP, Python, Fortran и др.). За целта се използват специално структурирани блокове от коментари, които съдържат информация за описаната програмна единица (клас, функция, структура и др.). В документацията могат да бъдат визуализирани връзките между различни програмни единици. Doxygen може да се използва чрез графичен интерфейс или команди в терминала. За създаването на документацията в текстов файл се описват основните характеристики и настройки за генерирането, като се използват дефинирани за софтуера променливи. Тези променливи могат да приемат точно определени стойности, като те често са булеви. Графичният интерфейс всъщност дава възможност за промяна на стойностите на променливите в конфигурационния файл, като също така дава краткото им описание. Възможно е създаването на шаблонен файл чрез програмата за улеснение на потребителя - чрез команда от терминала или от графичния интерфейс при стартиране. По-долу са описани някои основни настройки (параметри) в шаблонния файл, които са използвани при генерирането на документацията на библиотеката **LinCOdeWeightInv** чрез графичен интерфейс:

- В главното меню, съдържащо основните настройки, се задава директорията на проекта, който ще бъде документиран и директорията, в която да бъде записана документацията.

- При документиране на големи проекти, съдържащи повече файлове с програмен код, в основните настройки на конфигурационния файл се маркира флага *Scan Recursively*, съответстващ на променливата *RECURSIVE*, за да бъдат сканирани рекурсивно всички файлове, участващи в проекта.

- В подменюто *Mode* се избира програмният език на проекта и кои програмни единици да бъдат включени в описанието. В създадената документация са включени само документираните чрез коментари файлове и функции.

- В подменюто *Diagrams* се избира пакет за генериране на графично представяне на връзките между отделните програмни единици, както и кои връзки да бъдат описани. Сред възможните опции са графика на включените библиотечни файлове (използвано за описанието на библиотеката), йерархия на класовете, граф на извикваните функции и др.

- В менюто *Expert* могат да бъдат променени стойностите на допълнителни променливи, като скриване на недокументирани класове, генериране на документацията за определен програмен език (C/C++, Java, Fortran и др.) използван стил на коментарите и др.

За документирането на програмния код могат да се използват различни стилове на коментарите. Коментарите се поставят непосредствено преди програмната единица. Програмните единици могат да имат кратко и подробно описание. По подразбиране първото изречение на коментара се счита за кратко описание на програмната единица, докато останалият текст се счита за подробно описание. Три са основните възможни начина за описващ коментар:

- /** Кратко описание. */

- /*! Кратко описание.*/

- /// Кратко описание.

Различни команди могат да бъдат използвани в структурираните коментари, за да бъде генерирано по-описателна документация на функциите. Тези команди могат да започват със символа "@"(стил на коментара JAVADOC) или символа "\"(стил на коментара QT). Основните използвани команди за генериране на документацията на библиотеката са:

- @brief - кратко описание на файл или функция. За да бъда използвана тази команда, променливата JAVADOC_AUTOBRIEF трябва да бъде със стойност *true* чрез маркиране на съответното поле.

- @file - показва, че текущият файл трябва и описаните в него функции трябва да бъдат добавени в документацията. Задава се в началото на файла и се използва за файлове с разширение *.h*, тъй като по подразбиране тези файлове не се включват в подробното описание на библиотеката.

- @param - описание на параметър на функция

- @note - бележка към описаната програмна единица

5.4 Структура и компилиране

Файлова структура на библиотеката

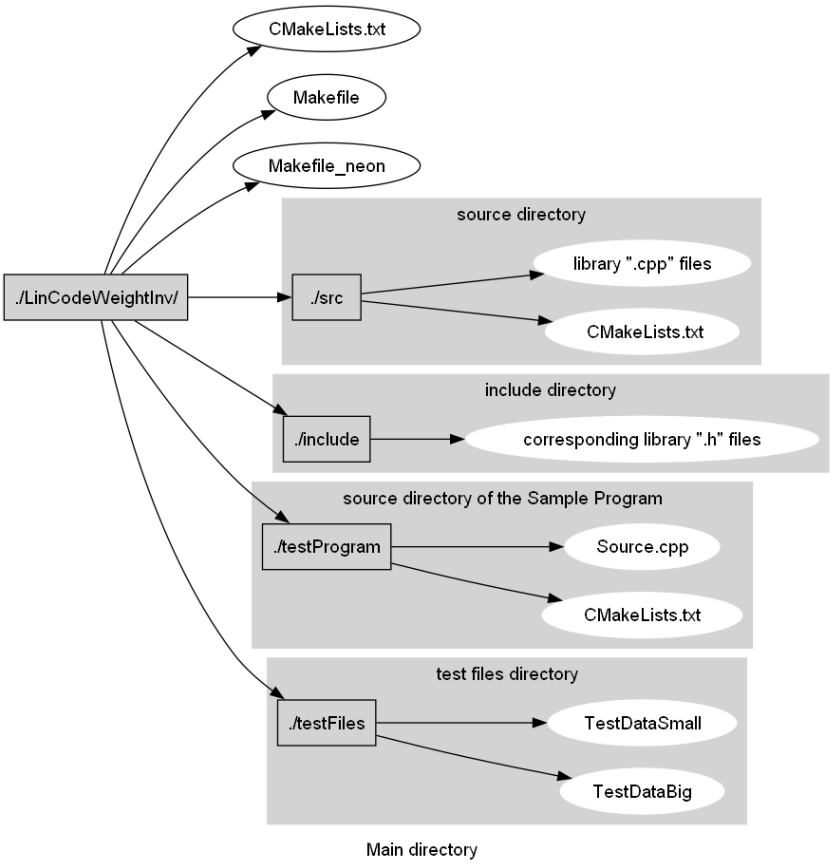
Разработената библиотека се състои от две основни части - статична C/C++ библиотека, която може да се използва самостоятелно в други проекти, и интерфейсна програма, която има за цел да предостави и покаже основните начини за използване на библиотеката. Файловете, съдържащи програмния код на библиотеката са разделени в две директории, съдържащи файлове с дефиниции (*header* файлове, директория *./include*) и файлове с имплементации на функции *source* файлове, директория *./src*). Тези директории се съдържат в основната директория на библиотеката *LinCodeWeightInv*. Основната директория също така съдържа необходимите CMake файлове за създаването на проект за компилиране на библиотеката и тестовата програма и два файла, позволяващи компилирането под Linux и MacOS чрез платформата Make. Останалите поддиректории съдържат съответно файловете с програмния код на програмата за тестване (*./testProgram*) и файлове с входни данни (*./testFiles*). Също тъка, всяка поддиректория, съдържаща файлове с програмен код, съдържа и съответен CMakeLists файл, който се използва за генериране на проект и компилация (променливи на средата, флагове и др.). Фигура 5.1 визуализира описаната структура.

Използване на CMake за компилиране

За създаване на проект със CMake, структурата на целевия софтуер се описва в текстови файлове с името *CMakeLists.txt*. Всяка директория, съдържаща файл с програмен код за компилиране, съдържа текстов файл с това име. За целевата библиотека са разработени следните три основни *CMakeLists.txt* файла (фиг. 5.1):

- Основен файл, съдържащ името на проекта, минималната версия на CMake, необходима за създаване на проекта и командите, показващи поддиректориите, където се намират останалите файлове CMakeLists.txt. Този файл се намира в основната директория.
- Файл, описващ структурата и начина на компилиране на самата библиотека. Този файл се намира в поддиректорията *./src*, която също съдържа всички необходими файлове за изграждане на текущата библиотека. Поддиректорията *./include* на главната директория с файловете съдържа

Фигура 5.1: LinCodeWeigntInv основни директории



файлове с дефинициите на основни функции на библиотеката.

- Файл, описващ структурата на приложената интерфейсна програма, който описва начина за добавяне на библиотеката. Този файл и изходният файл на програмата се намират в поддиректорията *./testProgram* на главната директория. Добавянето на библиотеката към друг проект може да бъде осъществени чрез минимална модификация на този файл - замяна на файла *Source.cpp* с файл, съдържащ програмния код на целевия проект. Този файл трябва да бъде добавен към директорията.

Създаването на проект чрез CMake за целевата платформа и среда може да бъде изпълнено чрез графичният интерфейс или чрез изпълнението на команди в команден терминал (подходящо при Linux базирани операционни системи). Командата

```
cmake -G <build\_system> -B <IDE\_project\_dir> -S <source\_dir>
```

създава проект за избрана интегрирана среда за компилиране (*build_ system*) в директория *IDE_ project_ dir*, като главният CMakeLists.txt файл се намира в директория *source_ dir*. Пример 5.1 показва генерирането на проект за операционна система Windows, MacOS (съответно системи за компилиране Visual Studio 16 2019 и XCode) и Linux базирана операционна система (система за компилиране по подразбиране Make).

Пример 5.1. `cmake -G "Visual Studio 16 2019" -B .\msvc -S .\`

```
cmake -G XCode -B .\xcode -S .\
```

```
cmake -B .\ -S .\
```

Командата

```
cmake -D CMAKE\_C\_COMPILER=<full\_compiler\_path>
```

може да се използва преди командата за конфигуриране за задаване на конкретен C компилатор. Аналогично, командата

```
cmake -D CMAKE\_CXX\_COMPILER=<full\_compiler\_path>
```

може да се използва за избиране на конкретен C++ компилатор. Пример 5.2 показва избирането на gcc/g++ компилатор с помощта на името му (пътят на компилатора е зададен в променливата PATH) като се използва система по подразбиране:

Пример 5.2. `cmake -D CMAKE\_CXX\_COMPILER=g++ -B .\buildDir -S .\`

Инсталиране

Под инсталиране на програма или библиотека се разбира създаването на съответния изпълним файл или статична/динамична библиотека в избрана директория. Проекта за създадената библиотека освен двата основни компонента - статичната библиотека и придружаващата програма за тестване, в средата за компилиране се създава и компонент (target) за инсталиране. Задаването на примерната програма като компонент за компилиране по подразбиране ще компилира библиотеката и изпълнимата примерна програма. Компилирането на компонента за инсталиране ще генерира библиотека в поддиректорията *lib* на главната директория, докато компилираната примерна програма

е в поддиректорията *build* на избраната директория за генератора (среда за разработване, Makefile и други). Той също така ще копира тестовите файлове *TestDataSmall* и *TestDataBig* в директорията *build* . В Unix система с генератор по подразбиране, зададен като Unix Makefiles, библиотеката и придружаващата примерна програма могат да бъдат конфигурирани, компилирани и инсталирани (Makefile се съхранява в поддиректорията make) със следните команди:

cmake -B .\make -S .
cd make
make all
make install

Коментари

Проблема, свързан с генерирането на преносим софтуер, е представен в [P2]. По тематиката е изнесен доклад на Международна конференция *"Иновативно STEM образование"* [D3]. Описаните техники в тази глава са използвани при създаването на оптимизирана библиотека, представена в [P6].

Библиография

[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] Бакоев, В. *Генериране на множества в лексикографска наредба и чрез минимално изменение.* Университетско издателство "Св. св. Кирил и Методой Велико Търново, 2023.

[3] Биков, Д. *Криптографски свойства на някои векторни булеви функции и паралелни алгоритми с CUDA.* PhD thesis, Великотърновски университет "Св. св. Кирил и Методий 2017.

[4] Буюклиев, И. *Алгоритмични подходи за изследване на линейни кодове.* Дисертация за присъждане на образователна и научна степен "доктор на математическите науки Институт по математика и информатика, Българска академия на науките, 2008.

[5] Марков, М. *Ефективни алгоритмии с приложение в криптографията с публичен ключ и теория на кодирането.* Дисертация за присъждане на образователна и научна степен "доктор Институт по математика и информатика, Българска академия на науките, 2024.

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

[7] ARM. Arm architecture instruction sets guide, 2023.

[8] Ballet, S., Bonnecaze, A., and Tukumuli, M. On the construction of elliptic chudnovsky-type algorithms for multiplication in large extensions of finite fields. *Journal of Algebra and Its Applications 15*, 01 (2016), 1650005.

[9] Barg, A., and Dumer, I. On computing the weight spectrum of cyclic codes. *IEEE Transactions on Information Theory 38*, 4 (1992), 1382–1386.

[10] 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.

[11] 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.

[12] 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.

[13] Bosma, W., Cannon, J., and Playoust, C. The magma algebra system i: The user language. *Journal of Symbolic Computation 24*, 3-4 (1997), 235–265.

[14] 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.

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

[16] 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.

[17] 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.

[18] Bouyukliev, I., Bouyuklieva, S., and Dodunekov, S. On binary self-complementary [120, 9, 56] codes having an automorphism of order 3 and associated sdp designs. *Problems of Information Transmission 43*, 2 (2007), 89–96.

[19] Bouyukliev, I., Bouyuklieva, S., Maruta, T., and Piperkov, P. Characteristic vector and weight distribution of a linear code. *Cryptography and Communications 13* (2021), 263–282.

[20] 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.

[21] Bradley, C., and Gaster, B. R. Exploiting loop-level parallelism for simd arrays using openmp. In *International Workshop on OpenMP* (2007), Springer, pp. 89–100.

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

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

[24] 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.

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

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

[27] Cebrian, J. M., Natvig, L., and Jahre, M. Scalability analysis of avx-512 extensions. *The Journal of supercomputing 76*, 3 (2020), 2082–2097.

[28] Cenk, M., and Özbudak, F. On multiplication in finite fields. *Journal of Complexity 26*, 2 (2010), 172–186.

[29] Chudnovsky, D., and Chudnovsky, G. Algebraic complexities and algebraic curves over finite fields. *Journal of Complexity 4*, 4 (1988), 285–316.

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

[31] Cramwinckel, J., Roijackers, E., Baart, R., Minkes, E., Ruscio, L., Miller, R., Boothby, T., Tjhai, C., and Joyner, D. Gap package guava.

[32] Dai, J., Yin, H., Liu, S., and Lv, Y. Avx-512 based, high-throughput ldpc decoders. In *2021 6th International Conference on Image, Vision and Computing (ICIVC)* (2021), pp. 431–435.

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

[34] 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.

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

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

[37] Dodunekov, S. M., Encheva, S. B., and Kapralov, S. N. On the [28, 7, 12] binary self-complementary codes and their residuals. *Designs, Codes and Cryptography 4*, 1 (1994), 57–67.

[38] Dodunekova, R., and Dodunekov, S. Sufficient conditions for good and proper error-detecting codes. *IEEE Transactions on Information Theory 43*, 6 (1997), 2023–2026.

[39] Flynn, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers C-21*, 9 (1972), 948–960.

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

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

[42] Fog, A. Vcl—c++ vector class library manual.

[43] Fujiwara, T., and Kusaka, T. The weight distributions of the (256, k) extended binary primitive bch codes with k$\leq$ 71 and k$\geq$ 187. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 104*, 9 (2021), 1321–1328.

[44] Gottschlag, M., Schmidt, T., and Bellosa, F. Avx overhead profiling: how much does your fast code slow you down? In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (2020), pp. 59–66.

[45] 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.

[46] Gulliver, T. A., Bhargava, V. K., and Stein, J. M. Q-ary gray codes and weight distributions. *Applied mathematics and computation 103*, 1 (1999), 97–109.

[47] Günther, S. M., Appel, N., and Carle, G. Galois field arithmetics for linear network coding using avx512 instruction set extensions. *arXiv preprint arXiv:1909.02871* (2019).

[48] 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.

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

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

[51] Intel. Intel® 64 and ia-32 architectures software developer's manual, 2023.

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

[53] 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.

[54] Kapralov, Stoyan; Christov, P., and Bogdanova, G. The new version of qlc—a computer program for linear codes studying. In *International Workshop on Optimal Codes and Related Topics* (1995), pp. 15–20.

[55] Kaski, P., and Ostergard, P. *Classification Algorithms for Codes and Designs*. Springer, 2006.

[56] 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.

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

[58] Lim, R., Lee, Y., Kim, R., and Choi, J. An implementation of matrix–matrix multiplication on the intel knl processor with avx-512. *Cluster Computing 21* (2018), 1785–1795.

[59] Lin, S., and J., C. D. *Error Control Coding: Fundamentals and Applications.* Prentice Hall, 1983.

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

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

[62] McGuire, G. Quasi-symmetric designs and codes meeting the grey–rankin bound. *journal of combinatorial theory, Series A 78*, 2 (1997), 280–291.

[63] Miller, F. P., Vandome, A. F., and McBrewster, J. *Cross Compiler: Compiler, MinGW, Executable, Computing platform, Embedded system, Microcontroller, Operating system, Paravirtualization, Source-to-source... Server farm, Bootstrapping (compilers).* Alpha Press, 2010.

[64] Mitra, G., Johnston, B., Rendell, A. P., McCreath, E., and Zhou, J. Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum* (2013), pp. 1107–1116.

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

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

[67] Parker, C., Spence, E., and Tonchev, V. D. Designs with the symmetric difference property on 64 points and their groups. *Journal of Combinatorial Theory, Series A 67*, 1 (1994), 23–43.

[68] Pashinska, M., and Bouyukliev, I. A parallel algorithm for computing the weight spectrum of binary linear codes. In *2020 Algebraic and Combinatorial Coding Theory (ACCT)* (2020), pp. 1–5.

[69] Pashinska, M., and Bouyukliev, I. A parallel algorithm for computing the weight spectrum of binary linear codes. In *2020 Algebraic and Combinatorial Coding Theory (ACCT)* (2020), pp. 1–5.

[70] Peleg, A., and Weiser, U. Mmx technology extension to the intel architecture. *IEEE Micro 16*, 4 (1996), 42–50.

[71] Pless, V. Power moment identities on weight distributions in error correcting codes. *Inf. Control. 6*, 2 (1963), 147–152.

[72] Pohl, A., Cosenza, B., Mesa, M. A., Chi, C. C., and Juurlink, B. An evaluation of current simd programming models for c++. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing* (2016), pp. 1–8.

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

[74] Robert, J.-M., and Véron, P. Faster multiplication over f 2 [x] using avx512 instruction set and vpclmulqdq instruction. *Journal of Cryptographic Engineering 13*, 1 (2023), 37–55.

[75] Shoup, V., et al. Ntl: A library for doing number theory.

[76] Tonchev, V. D. The uniformly packed binary [27, 21, 3] and [35, 29, 3] codes. *Discrete Mathematics 149*, 1-3 (1996), 283–288.

[77] 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.

[78] Ward, H. Divisible codes-a survey. *Serdica Mathematical Journal 27*, 4 (2001), 263p–278p.

[79] Wende, F., Noack, M., Steinke, T., Klemm, M., Newburn, C. J., and Zitzlsberger, G. Portable simd performance with openmp* 4. x compiler directives. In *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22* (2016), Springer, pp. 264–277.

[80] White, G. Enumeration-based algorithms in linear coding theory, 2006.

[81] 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.

[82] Yao, H., Fazeli, A., and Vardy, A. A deterministic algorithm for computing the weight distribution of polar codes. In *2021 IEEE International Symposium on Information Theory (ISIT)* (2021), IEEE Press, p. 1218–1223.

[83] Zhong, D., Cao, Q., Bosilca, G., and Dongarra, J. Using advanced vector extensions avx-512 for mpi reductions. In *Proceedings of the 27th European MPI Users' Group Meeting* (2020), pp. 1–10.

Изнесени доклади

[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

Публикации

[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, IEEE Xplore, 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, indexed in WoS **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, indexed in Scopus **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

Списък на цитирания

[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.