ИЗПОЛЗВАНЕ НА AVX ИНСТРУКЦИИ ЗА ОПТИМИЗАЦИЯ НА АЛГОРИТМИ ЗА ТЕГЛОВИ ХАРАКТЕРИСТИКИ НА ЛИНЕЕН ДВОИЧЕН КОД
UTILIZING AVX INSTRUCTION SET FOR OPTIMIZING ALGORITHMS FOR WEIGHT CHARACTERISTICS OF BINARY LINEAR CODE

Maria Pashinska, Iliya Bouyukliev
Institute of Mathematics and Informatics,
Bulgarian Academy of Sciences
mariqpashinska@math.bas.bg
iliyab@math.bas.bg

Abstract
We present methods of using Advanced Vector Extensions (AVX, AVX2, AVX512) for calculation of weight characteristics of binary linear codes. Some experimental results for different lengths and dimensions are presented.
Keywords: Linear codes, Weight distribution, Algorithms, AVX - 512

ВЪВЕДЕНИЕ

Тегловите характеристики на линейните кодове са важни както от теоретична гледна точка, така и от практическа. Известно е, че тяхното намиране е NP-трудна задача [1]. Съществуват различни алгоритми, базирани на код на Грей или допълнителни пораждащи матрици, използвани в различни софтуери за математически изчисления [2, 3, 4]. Известно е че за намирането на тегловите характеристики на един код е необходимо бързо генериране на всички кодови думи. Една естествена оптимизация, която следва от факта, че компютрите са базирани на двоична аритметика, е побитовото представяне на кодовите думи в една компютърна дума. Съвременните компютри разполагат с допълнителни регистри, чиято големина е кратна на стандартната дължина на компютърна дума (64 бита). Чрез тези регистри се отварят нови възможности за оптимизация също базирани на побитовото представяне на кодовите думи и двоичната аритметика, която е хардуерно реализирана в работата на централните процесори.

В текущата разработка ще се разгледа по-конкретно задачата за намиране на теглови спектър на линеен двоичен $[n, k]$ код C с пораждаща матрица $G_{k\times n}$. Можем да считаме, че пораждащата матрица е представена като двумерен масив с от 64 битови компютърни думи с k реда и s стълба, където $s = \frac{n-1}{64} + 1$. От друга страна, съвременните централни процесори за масово потребление разполагат с регистри с дължина 256 бита, които надграждат 128 битови регистри, присъстващи в централните процесори, разработени от Intel още през 1999 година. Последните поколения на процесорите, разработени от Intel, също така разполагат с 512 битови регистри, които присъстват и в ускорителите Intel Xeon Phi, предназначени за работа със суперкомпютри. Тези регистри могат да бъдат използвани чрез специализираните инструкции Streaming SIMD Extensions (SSE, SSE2,

SSE3, SSE4, където са дефиниране функции за работа със 128 и 256 битови регистри) и Advanced Vector Extensions (AVX, AVX2, AVX-512, където са дефиниране функции за работа със 256 и 512 битови регистри), като всяко ново поколение надгражда функциите на предходното.

Целта на текущата разработка е да се разгледат различни стратегии за оптимизация на алгоритъм за намиране на теглови спектър, като се използва разполагаемият ресурс (Intel Core i5-1035G1CPU), който притежава 512 битови регистри и набора от инструкции AVX-512 [5]. Ще се опишат получените експериментални резултати и ще се анализира полученото ускорение в зависимост от използваните стратегии и големината на използваните регистри.

ИЗЛОЖЕНИЕ

В текущата работа ще бъдат използвани допълнителни регистри и съответните набори от инструкции (SSE, AVX). В един регистър могат да се запишат няколко еднотипни променливи, като броят им зависи от техния тип и големината на регистъра. Така се позволява паралелно извършване на операции за няколко променливи (Single Instruction Multiple Data - SIMD), като съответната операция се извършва за целия регистър. За работа с регистрите се използват допълнителни библиотеки (<immintrin.h> за 512 битови регистри, <emmintrin.h> за 256 битови регистри, <smmintrin.h> за 128 битови регистри), в които са дефинирани специализирани типове данни и функции. Голяма част от декларираните функции отговарят на единствена машинна инструкция за централния процесор. Съществуват различни функции, които изпълняват еднотипни операции (напр. събиране на два вектора), които се различават по големината на регистъра и типа данни, записани в него. Това дава сравнително голям набор от възможности за използването на регистрите, като същевременно изисква много добро познание на инструкциите. Програмите, работещи с такива регистри, са написани на програмните езици C и C++, като те могат да бъдат компилирани за работа както на операционна система Windows, така и на Linux базирани системи. Друга особеност при работата с регистрите е необходимостта да се провери информацията за процесора, който ще изпълнява програмата. За всеки процесор съществуват четири 32 битови регистри (EAX, EBX, ECX, EDX), които съдържат информация за поддържаните от процесора функции. Всеки от битовете на тези регистри показва присъствието (отсъствието) на определени функции. Чрез инструкцията __cpuid информацията от тези регистри може да бъде прочетена.

Ще бъдат разгледани две стратегии за оптимизация – за кодове с малка дължина и за кодове с голяма дължина. И двете стратегии използват побитовото представяне на думите на един двоичен код и хардуерно реализираното събиране над полето с два елемента, като се използват регистри с дължина по-голяма от 64 бита. Ще бъдат генерирани всички кодови думи на един линеен код C, като се използва пораждащата му матрица, записана в масив от регистри G[k+1] и допълнителен масив от регистри H[k+1], който във всеки ред h съдържа линейни комбинации на h реда на пораждащата матрица. Тегловият спектър на кода е записан в масив W[n]. Алгоритъм 1 описва начина за намиране на линейните комбинации, когато пораждащата матрица на линейния код е записана като масив G[k+1] от 64 битови числа и дължина на кода n < 64. Тогава побитовото представяне на един елемент на този масив отговаря на един ред на пораждащата матрица. Събирането на два реда на пораждащата матрица отговаря на

побитовата операция XOR. Алгоритъмът е реализиран чрез рекурсивна функция, която приема като параметри h – броя на събраните редове на G и i – ниво на рекурсията. Функцията се извиква с първоначални стойности h = 1, i = 1, H[0] = 0 (номерата на редовете на матрицата започват от 1).

Алгоритъм 1:

```
void spec(int i, int h){
    for (int j = h; j < =k; j++) {
    H[h] = H[h - 1] xor G[j];
    W[ wt( H[h] ) ] ++;
    if (i < k){ spec (i+1, j+1);}
}
```

Стратегия 1:

Нека разгледаме линеен двоичен $[n, k]$ код C с пораждаща матрица $G_{k\times n}$, където $n \leq 128$. Тогава кодовите думи на C могат да бъдат разгледани като обединение на кодовите думи на C' с пораждаща матрица $G'_{(k-1)\times n}$, получена от G като е премахнат последният ред g_k, и съседния клас на кода C', получен чрез събиране на всички кодови думи на G' с g_k. Регистрите с големина 256 бита ни позволяват да получим тези две множества паралелно. Матрицата G' е записана два пъти в масив от регистри, като всеки ред на масива, съдържа две копия на съответния ред на матрицата G'. В първите 128 бита на масива H ще бъдат записани кодовите думи на C' за текущият броя на събраните редове на G', докато във вторите 128 бита ще бъдат записани кодовите думи, получени след събиране с g_k. Това е постигнато, като нулевият ред на масива H има вида H [0] = $\langle 0|g_k\rangle$. В основния алгоритъм масива G' се обхожда до ред k. Чрез тази стратегия сложността на алгоритъма се свежда от $\sum_{j=1}^{k}\binom{k}{j}$ до $\sum_{j=1}^{k-1}\binom{k-1}{j}$. Операцията XOR в алгоритъма се реализира чрез функцията _mm256_xor_si256(…), която извършва побитово събиране върху целите регистри. Намирането на теглото на текущата линейна комбинация също става чрез единствена функция _mm256_popcnt_epi64(v1), която записва в 256 битов регистър теглата на четирите компютърни думи, записани в регистъра __m256i v1. Резултатът се записва в 256 битов регистър, като се използва дефиниране на потребителски тип чрез ключовата дума union. Такива потребителски типове позволяват разглеждането на една и съща памет, като различни типове данни. Те също така са използвани за дефиниране на масивите от регистри G и H. Така един 256 битов регистър може да се използва едновременна като масив от 4 цели числа тип unsigned long long int. Тогава за намиране на теглото е необходимо събирането на първите и вторите две числа, които отговарят на wt (H[h]) и wt (g_k + H[h]) съответно.

Стратегия 2:

Нека сега разгледаме линеен двоичен $[n, k]$ код C с пораждаща матрица $G_{k\times n}$, където $n > 128$. Ако са налични регистри с големина 512 бита, отново е възможно да се приложи стратегия 1 за $n \leq 256$. В противен случай при $n > 256$ или когато са налични само 256 битови регистри, стратегия 1 не е приложима. Нека приемем наличието на 512 битови регистри. Тогава всеки ред на пораждащата матрица ще бъде записан в двумерен масив от регистри с k реда и $\frac{n-1}{512} + 1$ колони. Тук в основния алгоритъм трябва да се добави обхождане на масива от регистри по колоните му, като XOR операцията отново

се извършва за целия регистър. Намирането на теглото за текущия ред на помощния масив H[h] се извършва чрез две функции _mm512_popcnt_epi64(v1), която записва в 512-битов регистър теглата на 8 компютърни думи, записани в регистъра __m512i v1, и _mm512_reduce_add_epi64(…), която ще събере получените тегла. Важно е да се спомене, че тези две функции не са напълно оптимизирани, тъй като не отговарят на единствена машинна команда. Също така наличието на 512 битови регистри не гарантира присъствието на всички команди, обработващи тези регистри. За това е необходимо използването на __cpuid, където може да се намери информация за поддържаните функции за конкретния работен процесор.

Експериментални резултати

В Таблица 1 са показани получените експериментални резултати. В първата колона са описани параметрите на кода, за който е изчислен тегловият спектър. Във втората колона е записано времето за работа в секунди на пакета Q-Extension New Edition за един код със съответните параметри. В третата колона е записано работното време в секунди получено при използване на стратегия 1 за код с дължина по-малка от 256. Тук в зависимост от дължината на кода се използвани 128, 256 и 512 битови регистри съответно за $n \leq 64$, $n \leq 128$ и $n \leq 256$. При кодове с по-голяма дължина се изпълнява единствено стратегия 2 и за такива кодове не са записани работни времена. В четвъртата колона е записано работното време при използване на стратегия 2 и 512 битови регистри. Програмите са изпълнени на платформа с операционна система Windows 10 Pro (x64-based) и централен процесор Intel Core i5-1035G1 CPU @ 1.00GHz, 4 Core(s), 8 Logical Processor(s).

От Таблица 1 се вижда, че полученото ускорение в сравнение с пакета Q-Extension New Edition варира между 3,9 и 4,2 пъти за стратегия 1 и между 2 и 5,25 пъти за стратегия 2. При стратегия 2 се вижда увеличение на ускорението при увеличение на дължината на кода, докато при стратегия 1 ускорението се увеличава с увеличението на размерността.

***Табл. 1**_Експериментални резултати*

Параметри	Q-Extension (сек.)	Стратегия 1 (сек.)	Стратегия 2 (сек.)
[64, 25]	0,316	0,081	0,159
[64, 30]	9,884	2,438	3,370
[64, 33]	80,099	19,210	33,691
[128, 25]	0,349	0,086	0,123
[128, 30]	11,262	2,687	3,855
[128, 33]	92,044	21,550	31,631
[256, 25]	0,448	0,116	0,130
[256,30]	14,274	3,451	4,208
[256, 33]	115,854	27,747	34,463
[512, 25]	0,636		0,133
[512, 30]	19,781		4,640
[512, 33]	157,363		32,500

Предимства и недостатъци

Работата с допълнителните регистри има своите недостатъци. Известно е че при стартирането на повече от един работни процеси е възможно намаляване на тактовата честота на процесора. Следователно алгоритми, оптимизирани за работа върху много нишки, могат да не постигнат оптимални резултати при работа с допълнителните регистри. Също така споменатите команди в тази работа са дефинирани единствено за централните процесори на Intel. Някои от тези команди имат еквивалент за централни процесори на други производители за регистри с големина 128 и 256 бита. Това води до голяма зависимост на програмния код от конкретния производител на централния процесор. Друг недостатък, който е удачно да бъде споменат, е наличието на функции, които съответстват на няколко машинни инструкции. За това е необходимо много добро познание на работата на самите функции при работата с тях.

От друга страна, използването на допълнителни регистри позволява ускорение на алгоритми за намиране на теглови характеристики, което се постига чрез едновременното обработване на до 512 бита. Наличието на тези регистри позволява лесно преобразуване на вече реализирани алгоритми, използващи побитово представяне и xor операция, като имплементацията на кода остава лесно разбираема. Дефинирането на потребителски тип чрез union ни позволява да разглеждаме тези регистри едновременно като масив от 64 битови числа. Така кода остава лесно четим, като не се извършва допълнително копиране на данните.

ЗАКЛЮЧЕНИЕ

Масово последните поколения процесори разполагат с допълнителни регистри с дължина по-голяма от стандартната компютърна дума. Възможно е тези регистри да бъдат използвани ефективно за оптимизирането на алгоритми за намиране на теглови характеристики на линейни кодове. Според получените експериментални резултати може да се постигне до 5 кратно ускорение в сравнение с програмни пакети, разработени и оптимизирани за работа с линейни кодове. От друга страна, всеки процесор разполага с различни по големина регистри и различен набор от функции за работа с тях. Така използването на конкретни функции и регистри може да ограничи преносимостта на програмния код. За това тези регистри трябва да се използват при много добро познание на целевия хардуер или съответния програмен код да описва всички възможности за използването им, като се избира най-подходящата за конкретния централен процесор.

ACKNOWLEDGEMENTS

The research of Maria Pashinska was supported, in part, by National program "Young scientists and postdoctoral students" РМС №577 от 17.08.2018. The research of Iliya Bouyukliev was supported, in part, by the Bulgarian Ministry of Education and Science by Grant No. DO1-221/03.12.2018 for NCHDC, a part of the Bulgarian National Roadmap on RIs.

ЛИТЕРАТУРА

[1] E. R. Berlekamp, R. J. McEliece and H. C. van Tilborg. 1978. On the inherent intractability of certain coding problems, IEEE Trans.Inform. Theory, 24, 384386.

[2] W. Bosma, J. Cannon, and C. Playoust, 1997. The Magma algebra system I: The user language, J. Symbolic Comput. 24, 235265.
https://doi.org/10.1006/jsco.1996.0125
[3] R. Baart, T. Boothby, J. Cramwinckel, J. Fields, D. Joyner, R. Miller, E. Minkes, E. Roijackers, L. Ruscio, C. Tjhai, GAP package GUAVA.
https://www.gap-system.org/Packages/guava.html
[4] I. Bouyukliev, "QEXTNEWEDITION - GENERATION module," Online available at
http://www.moi.math.bas.bg/moiuser/_data/Software/QextNewEdition.html
[5] Intel Intrinsics Guide, Available at:
https://software.intel.com/sites/landingpage/IntrinsicsGuide/