Optimizacija algoritama povećanjem prostorne lokalnosti pristupa memoriji

Mislav Đomlija

25.1.2023.

1 UVOD

Performansa modernih računala uvelike ovisi o brzini memorije. U početcima proizvodnje i dizajniranja računala, brzine procesora i memorije nisu se značajno razlikovale. Smanjenje tranzistora i rafiniranje proizvodnih procesa rezultirali su višestrukim ubrzanjem procesora u odnosu na brzinu memorije. Na mnogim modernim sustavima, dohvaćanje podatka iz radne memorije (*engl.* Random access memory) rezultira trošenjem više stotina ciklusa procesora na čekanje. Moderna računala taj problem pokušavaju riješiti korištenjem više razina priručnih memorija (*engl.* Cache). Priručne memorije su puno brže, ali i skuplje od radne memorije te imaju značajno manji kapacitet. Prijenos podataka između priručnih memorija i između priručne memorije zadnje razine i radne memorije odvija se u linijama tipične veličine 64B. To znači da uslijed potrebe za dohvatom jednog podatka, u priručne memorije se učitavaju i njemu susjedni podatci. Procesor direktno može pristupiti samo priručnoj memoriji prve razine (*engl.* L1 cache). Pošto je logički adresni prostor značajno veći od fizičke veličine priručne memorije postoji mogućnost da traženi podatak nije prisutan. U tom slučaju se podatak traži u memoriji viših razina koje su veće i udaljenije te posljedično sporije. Brzina priručne memorije je veliki motivator za iskorištavanje lokalnosti pristupa.

2 OPTIMIZACIJA

Za mnoge stvarne primjene performansa je izrazito bitna. Osim kroz poboljšanja procesora i ostalih računalnih sklopova, performansu možemo povećati optimizacijom samog algoritma koji se izvodi. Za algoritam je, osim asimptotske složenosti, od presudne važnosti i prilagodba za sklopovlje na kojem se izvodi. Moderna računala imaju standardiziranu arhitekturu memorijskog sustava te se algoritmi mogu optimzirati za maksimalno iskorištenje priručnih memorija. Različita računala će imati različite specifikacije, ali sličnu arhitekturu. Prevodilac, može saznati specifikacije sustava[1] u trenutku prevođenja izvornog programa i dopuniti informacije o specifikacijama. Takav pristup rezultira strojnim kodom koji je prilagođen računalu na kojem će se izvoditi

2.1 Prostorna lokalnost pristupa memoriji

Prostorna Lokalnost (*engl.* spatial locality) je ideja da će se memorijske lokacije blizu korištenih lokacija također koristiti. Ideja prostorne lokalnosti primjenjuje se prilikom oblikovanja memorijskog sustava računala. Blokovskim prijenosom podataka

1 npr. instrukcijom CPUID na Intelovim arhitekturama

povećava se propusnost, a latencija ostaje približno jednaka. Uz pretpostavke da je kvant prijenosa veličine n i da je potrebno dohvatiti $2n$ podatka, u slučaju slijednih podatka taj prijenos se može obaviti uz samo dva pristupa višoj memorijskoj razini.

2.2 Algoritmi s optimiziranim pristupom memoriji

Algoritmi se često izvode nad strukturama koje su kontinuirano pohranjene u memoriji (polje, array list). Osim što takve strukture omogućuju pristup proizvoljnom podatku u složenosti $O(1)$, one bolje iskorištavaju svojstvo lokalnosti pristupa od struktura zasnovanih na pokazivačima (povezane liste, grafovi). Uz pretpostavku da neki algoritam a mora pristupiti n slijednih lokacija jednom i da podatci nisu međusobno ovisni, idealna implementacija algoritma a bi svaki podatak samo jednom učitavala u priručnu memoriju najniže razine. U stvarnom svijetu, istim memorijskim lokacijama se mora više puta pristupiti (npr. množenje matrica) i podatci su međusobno ovisni (npr. sortiranje) no ideja je i dalje ista. Algoritam će biti brži ako može minimizrati potrebu za dohvaćanjem podatka koji su međusobno daleko pohranjeni

2.3 OpenBLAS

Biblioteka OpenBLAS je jedna od najpopularnijih implementacija programskog sučelja BLAS (Basic linear algebra subroutines). Sučelje BLAS je podijeljeno u tri razine. Prva razina implementira funkcije između dva vektora, druga razina između matrice i vektora, a treća razina između dvije matrice. Biblioteka je implementirana pomoću programskih jezika C i Fortran, a kritični dijelovi su pisani u strojnom jeziku. Bibloteka koristi blokovske algoritme koji značajnu smanjuju memorijske promašaje i vektorske instrukcije te ostvaruje izvrsne performanse. Bibloteka se preuzima kao izvorni kod, te se prevodi na računalu kako bi se sklopovlje računala maksimalno moglo iskoristiti.

3 PRIMJER

Množenje matrica je od velike koristi u mnogim područjima te postoji velika potreba za optimizacijom istoga. Biblioteka OpenBLAS sadrži funkciju *dgemm* koja izvodi matrično množenje nad matricama brojeva dvostruke preciznosti. Prezentirat ćemo i usporediti performansu školskog algoritma, optimiziranog školskog algoritma i funkcije *dgemm*.

3.1 Školski algoritam

Želimo izračunati

$$A \times B = C$$

Školski algoritam skalarni produkt retka i matrice A i stupca j matrice B zapisuje na mjesto (i,j) matrice C. To matematički možemo zapisati ovako

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in} + b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Nedostatak ovog algoritma proizlazi iz načina spremanja matrica u memoriju. One se pohranjuju kao jedan dugački niz brojeva, na način da se redci ulančavaju. To znači da su podatci $A[i][j]$ i $A[i][j+1]$ jedan pored drugog a između podataka $A[i][j]$ i $A[i+1][j]$ se nalazi n podataka. Ako je matrica dovoljno velika, ta dva podatka neće biti u istim linijama priručne memorije. Zbog tri ugniježđene petlje složenost ovog algoritma je $O(n^3)$ U programskom jeziku C ovo bismo mogli ovako implementirati:

```
double **multiplyMatrix(double **matrix1, double **matrix2,
    int m1, int n1, int m2, int n2)
{
    double **product = initMatrix(m1, n2);

    for (int i = 0; i < m1; i++)
    {
        for (int j = 0; j < n2; j++)
        {
            double element = 0.0;
            for (int k = 0; k < n1; k++)
            {
                element += matrix1[i][k] * matrix2[k][j];
            }
            product[i][j] = element;
        }
    }

    return product;
}
```

3.2 Optimiziran školski algoritam

U prethodnom algoritmu nailazimo na neefikasan pristup stupcima matrice B. U slučaju velikih matrica, gotovo za svaki pristup elementu stupca, trebali bi izbacivati linije iz priručne memorije. Za svaki pristup učitavamo jedan podatak koji koristimo i više podataka koje ne možemo iskoristiti. Jednostavno poboljšanje je transponiranje matrice B. U tom slučaju je matrica B pohranjena "po stupcima". U trenutku potraživanja podatka $B[i][j]$ učitat ćemo još podataka iz istog stupca[2]. Asimptotska složenost algoritma se ne mijenja zato što transpoziciju možemo izvesti u $O(n^2)$. U programskom jeziku C to bismo mogli ovako implementirati[3]:

```
double **multiplyTransposedMatrix(double **matrix1, double **matrix2, int m1, int n1, int
     m2, int n2)
{
    assert(n1 == m2 && m2 == n2);
    double **product = initMatrix(m1, n2);
    double **matrix2Transposed = transpose(matrix2, m2, n2);

    for (int i = 0; i < m1; i++)
    {
        for (int j = 0; j < n2; j++)
        {
            double element = 0.0;
            for (int k = 0; k < n1; k++)
            {
                element += matrix1[i][k] * matrix2Transposed[j][k];
            }
            product[i][j] = element;
        }
    }

    return product;
```

2 B predstavlja originalnu matricu, ona će u memoriji biti zapisana transponirano te ćemo u programskom kodu morati obrati pažnju na to

3 radi jednostavnosti promatramo samo kvadratne matrice

```
21 }
```

3.3 dgemm

Funkcija dgemm koristi blokovski algoritam. Algoritam prvo podijeli matrice u blokove koji stanu u priručnu memoriju druge razine (*engl.* L2 cache), zatim podijeli blokove u pruge od 8 elemenata[4]. Međurezultati se spremaju u vektorske registre. Složenost ovog algoritma je također $O(n^3)$

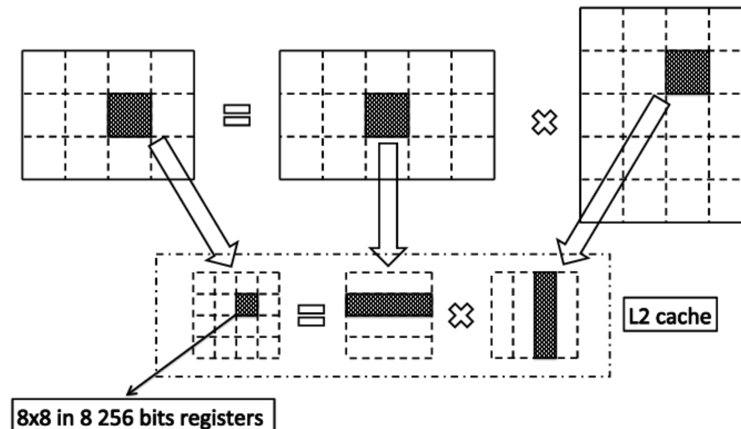


Figure 1: blokovski algoritam (https://www.researchgate.net/publication/308844880_Sparse_Convolutional_Neural_Networks).

U programskom jeziku C ovu funkciju pozivamo ovako:

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                m, n, k, 1.0, A, k, B, n, 0.0, C, n);
```

Prije samog poziva potrebno je preuzeti i prevesti biblioteku[5]. C programski kod moramo prevesti s navođenjem lokacije biblioteke[6]:

```
    gcc -o my_benchmark my_benchmark.c \
    -I /opt/OpenBLAS/include/ \
    -L /opt/OpenBLAS/lib \
    -Wl,-rpath,/opt/OpenBLAS/lib -lopenblas -O3
```

3.4 Postupak

Za usporedbu brzine algoritama koristit ćemo standardnu biblioteku programskog jezika C *time.h*. Za mjerenje broja pristupa i promašaja priručne memorije koristit ćemo funkcionalnost *cachegrind*[7] alata *Valgrind*. *Cachegrind* simulira stvarnu memorijsku arhitekturu te analizira pristupe i promašaje memorije. Također može izvesti analizu predikcije grananja. Svaki algoritam prevodimo uz zastavicu -O3 koja označava najveći stupanj optimizacije koda.

4 Ako je biblioteka prevedna za računalo koje podržava AVX512 instrukcije
5 https://github.com/xianyi/OpenBLAS
6 Također koristimo zastavicu -O3 koja agresivno optimizira strojni kod, iako se to ne odnosi na samu funkciju ostavljamo je radi uniformnosti
7 https://valgrind.org/docs/manual/cg-manual.html

3.5 Rezultati

Rezultate našeg mjerenja performansi (*engl.* Benchmark) ćemo prezentirati kroz tablice. Prikazani su samo rezultati promašaja i pristupa priručnoj memoriji prve razine. *Cachegrind* simulira samo prvu i zadnju priručnu memoriju, a promašaji zadnje priručne memorije su se samo razlikovali za najveću matricu pa nisu uključeni[8].

Veličina matrice	Vrijeme/s	Broj pristupa priručnoj memoriji	Broj promašaja priručne memorije
200	0.12	2.5×10^7	1.1×10^6
500	0.79	3.4×10^8	1.6×10^8
1000	2.41	2.6×10^9	9.8×10^8
1500	10.16	8.8×10^9	3.5×10^9
2000	37.65	$2.1 \times 10^1 0$	8.4×10^9

Table 1: Školski algoritam.

Veličina matrice	Vrijeme/s	Broj pristupa priručnoj memoriji	Broj promašaja priručne memorije
200	0.09	1.4×10^7	1.1×10^6
500	0.69	1.6×10^8	1.6×10^7
1000	1.26	1.1×10^9	1.3×10^8
1500	4.34	3.7×10^9	4.3×10^8
2000	10.25	8.5×10^9	1.1×10^9

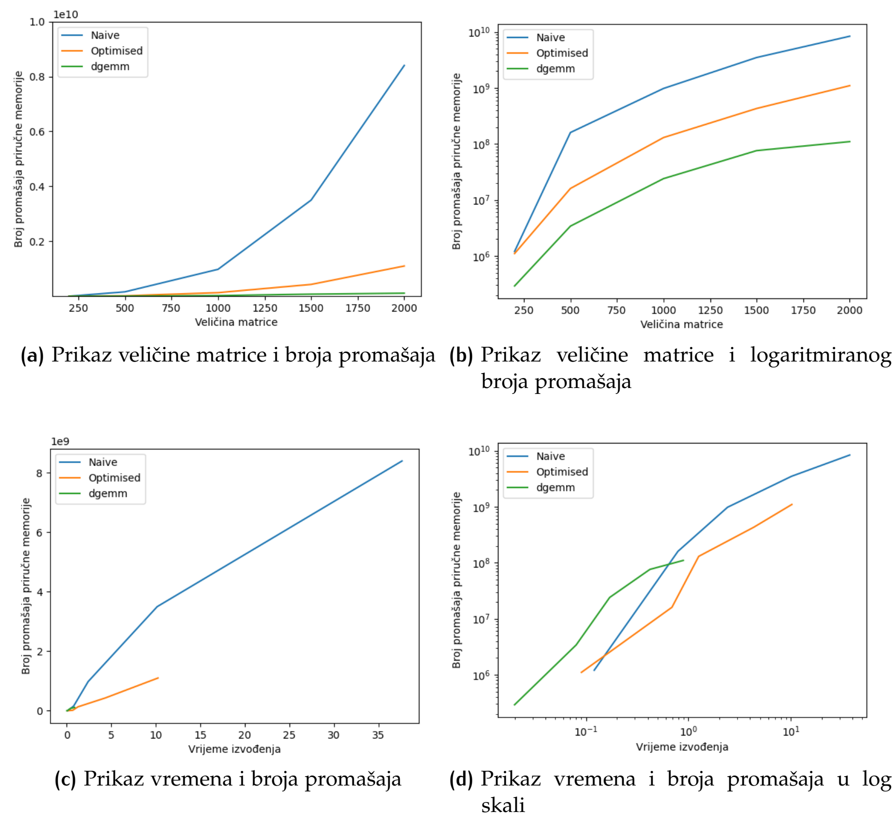
Table 2: Optimizirani školski algoritam.

Veličina matrice	Vrijeme/s	Broj pristupa priručnoj memoriji	Broj promašaja priručne memorije
200	0.02	7.5×10^6	2.9×10^5
500	0.08	5.3×10^7	3.4×10^6
1000	0.17	2.9×10^8	2.4×10^7
1500	0.42	8.5×10^8	7.6×10^7
2000	0.89	1.8×10^9	1.8×10^8

Table 3: dgemm.

8 Ovisno o veličini priručnih memorija ovo, može a i ne mora biti slučaj. Za veće matrice ovo može predstavljati ozbiljan problem

3.6 Grafički prikaz mjerenja



(a) Prikaz veličine matrice i broja promašaja

(b) Prikaz veličine matrice i logaritmiranog broja promašaja

(c) Prikaz vremena i broja promašaja

(d) Prikaz vremena i broja promašaja u log skali

Figure 2: Vizualizacija

4 ZAKLJUČAK

Eksperimenti ukazuju na ovisnost između broja promašaja priručne memorije i vremena izvođenja algoritma. Također postoji ovisnost između broja pristupa priručnoj memoriji i vremenu izvođenja. Funkcija *dgemm* osim što bolje iskorištava priručnu memoriju povećanjem lokalnosti pristupa koristi i vektorske instrukcije. Iako sva tri algoritma imaju istu asimptotsku složenost, stvarno vrijeme izvođenja se značajno razlikuje. Ovaj rezultat upućuje na isplativost optimizacije programa u smjerovima povećanja prostorne lokalnosti i povećanje udjela vektorskih instrukcija gdje je to moguće.

5 LITERATURA

https://valgrind.org/docs/manual/cg-manual.html
https://github.com/xianyi/OpenBLAS/wiki
https://pubs.opengroup.org/onlinepubs/7908799/xsh/time.h.html
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
Hennessy, Patterson. Computer Architecture A Quantitative Approach (5th edition)
http://acs.pub.ro/~cpop/SMPA/Computer%20Architecture%20A%20Quantitative%20Approach%20(5th%20edition).pdf