Tehnici de Optimizare de Cod – Inmultirea Matricelor

Obiective

In acest laborator vom exemplifica o serie de optimizari de cod pe una dintre cele mai simple, si in acelasi timp utilizate probleme, si anume, inmultirea matricelor.

De ce inmultirea matricelor?

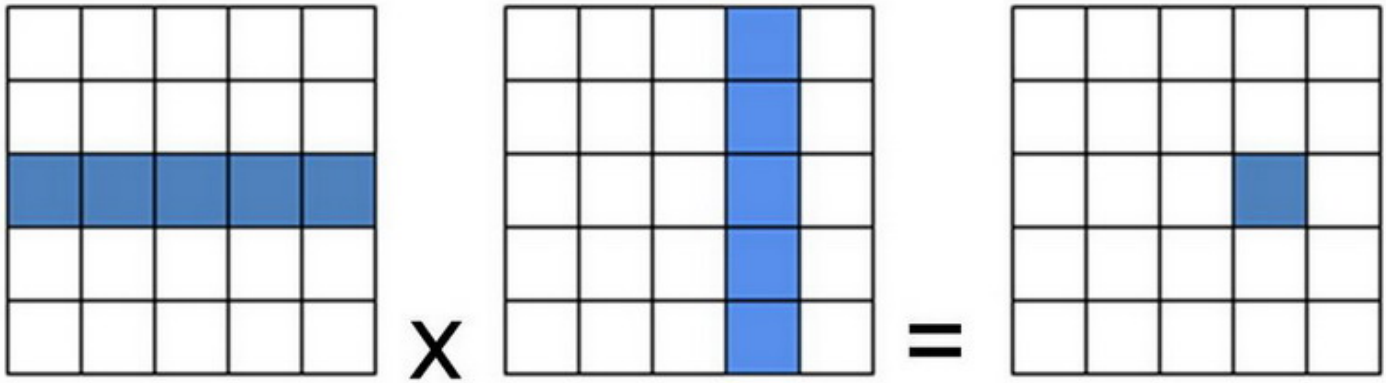
Este o operatie fundamentala si elementara in algebra liniara ce serveste la rezolvarea unui numar extrem de mare de probleme, cum ar fi: rezolvarea sistemelor liniare de ecuatii in majoritatea domeniilor stiintifice si economice (operatiile cu matrice sunt practic prezente pretudindeni); calcule si operatii cu grafuri; inversari de matrice. Problema inmultirii matricelor este in mod cert cea mai bine studiata problema in HPC (High Performance Computing), ea beneficiind de o multitudine de algoritmi inteligenti si implementari performante pe toate arhitecturile existente astazi. Pentru a simplifica lucrurile, in acest laborator ne vom ocupa doar de inmultirea matricelor patratice.

Cel mai simplu algoritm

Intuitiv, cel mai simplu algoritm, urmeaza formularea matematica:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Matricele A = [aij], i,j=1,…,N si B = [bij], i,j=1,…,N sunt salvate ca vectori bidimensionali de marime N x N. Matricea rezultat C = A x B = [cij], i,j=1,…,N, avand fireste aceeasi dimensiune.



A x B = C

Cum este si de asteptat, similar cu majoritatea operatiilor din algebra liniara, formula de mai sus se transforma in urmatorul program extrem de simplu:

```
int i,j,k;
double a[N][N], b[N][N], c[N][N];
// initializarea matricelor a si b
for (i=0;i<N;i++){
   for (j=0;j<N;j++){
      c[i][j] = 0.0;
      for (k=0;k<N;k++){
        c[i][j] += a[i][k] * b[k][j];
      }
   }
}
```

Cat de bun este acest algoritm?

Algoritmul este bun pentru ca:

- Se poate specifica in doar cateva linii;
- Este o mapare directa a formulei de calcul pentru Cij (din algebra liniara); este usor de inteles si de urmarit de catre oricine poseda cunostinte minime de matematica;
- In sfarsit, in mod sigur nu contine bug-uri datorita simplitatii extreme pe care o manifesta algoritmul!

Algoritmul este prost pentru ca:

- Are performante extrem de reduse!

De aceea ne vom ocupa in acest laborator de optimizarea acestei operatii din punctul de vedere al performantei.

Optimizarea algoritmului de inmultire a doua matrice

Detectarea constantelor din bucle

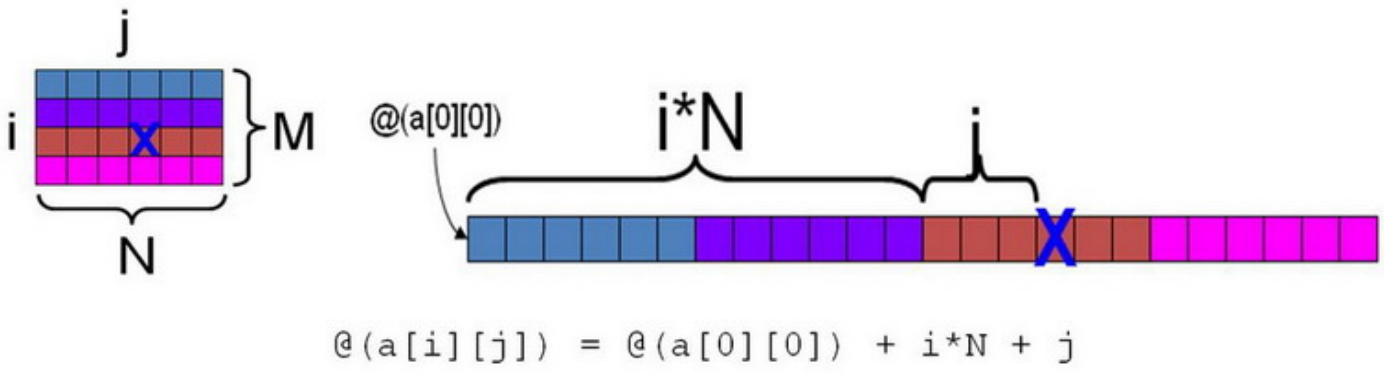
Prima optimizare, consta in a observa ca c[i][j] este o constanta in cadrul ciclului interior k. Totusi, pentru un compilator acest fapt nu este neaparat evident deoarece c[i][j] este o referinta in cadrul unui vector. Astfel, o prima optimizare va arata asa:

```
for (i=0;i<N;i++){
   for (j=0;j<N;j++){
      register double suma = 0.0;
      for (k=0;k<N;k++) {
         suma += a[i][k] * b[k][j];
      }
      c[i][j] = suma;
   }
}
```

In acest mod, compilatorul va putea avea grija ca variabila suma sa fie tinut intr-un registru, permitand astfel o utilizare optima a acestei resurse. Astfel utilizarea keyword-ului “register” este util de folosit ca hint pentru compilator, atunci cand socotiti ca acest lucru este util.

Accesul la vectori

Un alt aspect care necesita resurse din plin, este utilizarea si accesul variabilelor de tip vectorial. De fiecare data cand programul face o referinta la un obiect de tipul X[i][j][k] compilatorul trebuie sa genereze expresii aritmetice complexe, pentru a calcula aceasta adresa, in cadrul vectorului muldimensional X. De exemplu, iata cum arata un vector bidimensional in limbajul C (salvat row-major):



Astfel, pentru N = 6, M = 4: a[2][3] = a[0][0] + 2*6 + 3 = a[0][0] + 15

In limbaje de programare ca FORTRAN-ul, formula este inversata, deoarece aceste limbaje salvează vectorii în format column-major:

a[i][j] = a[0][0] + j*M + i

Oricare ar fi asezarea vectorilor in memorie, accesele la vectori sunt scumpe din punctul de vedere al performantelor. Noi vom considera de aici inainte o asezare row-major, ca in limbajul C. Conform acestei formule, pentru vectori bidimensionali (matrice), fiecare acces presupune doua adunari si o inmultire (de numere intregi). Evident, pentru vectori cu mai multe dimensiuni, aceste costuri cresc considerabil. Astfel, in momentul in care compilatorul intalneste instructiunea:

suma += a[i][k] * b[k][j]

se vor efectua implicit, suplimentar inmultirii si adunarii in virgula mobila implicata de codul de mai sus, patru adunari si doua inmultiri in numere intregi pentru a calcula adresele necesare din vectorii a si b. Se intampla astfel destul de frecvent ca procesorul sa nu aiba date disponibile pentru a lucra in continuu, din cauza faptului ca overhead-ul pentru calculul adreselor este semnificativ.

Astfel, un mod de a spori viteza programului este renuntarea la accesele vectoriale prin derefentiere utilizand in acest scop pointeri. De exemplu:

```
for (j=0;j<N;j++)
   a[i][j] = 2;         // 2*N adunari si N inmultiri
```

se va inlocui cu:

```
double *ptr=&(a[i][0]); // 2 adunari si o inmultire
for (j=0;j<N;j++) {
   *ptr = 2;
   ptr++;               // N adunari in numere intregi
}
```

In mod similar se procedeaza si pentru cazul in care indexul incrementat este cel al liniilor si nu cel al coloanelor. In ambele cazuri, practic se va calcula “de mana” adresa in cadrul vectorului, exact in modul in care ar face-o compilatorul limbajului folosit. Totusi, rezolvarea noastra este mai rapida, deoarece ea tine cont de pozitia in care ne aflam in cadrul vectorului, lucru destul de complicat de facut automat. De exemplu, pentru a trece la urmatoarea coloana, e suficient sa adunam N pointer-ului, fata de recalcularea pornind de la @(a[0][0]) ce necesita doua inmultiri si o adunare in intregi. Evident, facilitatile oferite de limbaje ca C-ul, ne vin in ajutor: astfel incrementarile de pointeri de tip char * vor face incrementarea cu un byte, in vreme ce pentru int * se va face cu patru bytes. Ca urmare a aspectelor prezentate mai sus, iata forma optimizata in care ajunge algoritmul nostru:

```
for(i = 0; i < N; i++){
  double *orig_pa = &a[i][0];
  for(j = 0; j < N; j++){
    double *pa = orig_pa;
    double *pb = &b[0][j];
    register double suma = 0;
    for(k = 0; k < N; k++){
      suma += *pa * *pb;
      pa++;
      pb += N;
    }
    c[i][j] = suma;
  }
}
```

Atentie! Codul de mai sus va da rezultate corecte doar daca matricile sunt declarate global sau pe stivă pentru că în felul acesta sunt stocate continuu în memorie (și are sens pb += N). Dacă alocați dinamic, atunci folosiți matrici liniarizate și adaptați acest cod pentru cazul lor.
Din primele doua optimizari se pot desprinde cateva concluzii. Prima ar fi ca optimizarea unui cod (din punct de vedere al performantelor), presupune utilizarea a cat mai putine constructii complexe (high-level), puse la dispozitie de limbajul folosit. Aceasta concluzie poate suna extrem de ciudat pentru cineva care porneste de la ideea ca facilitatile limbajelor de programare sunt acolo pentru a fi folosite. Da, este adevarat acest lucru, insa atunci cand vrei performanta, trebuie sa stii ce constructii sa eviti! Astfel, apare concluzia a doua: vectorii sunt concepte mai abstracte decat pointerii (ca implementare), asadar, utilizati pointeri cand vreti viteza. Viteza crescuta insa, va fi obtinuta cu pretul unui cod mult mai dificil de urmarit si de inteles, mai rau, mult mai greu de debug-at. Un cod complex si performant, de multe ori poate contine bug-uri extrem de subtile si greu de depistat. Asadar, e util sa stii exact ceea ce faci cand incepi sa faci astfel de optimizari!

Activitate practica - Optimizare constantelor si al accesului la vectori

Intrebarea este acum: aduc ceva imbunatatiri optimizarile 1 si 2? Pentru a afla raspunsul la aceasta intrebare, va invitam sa implementati problema, cu optimizarile sugerate, si sa observati singuri ce se intampla.

Optimizarea pentru accesul la memorie

Dupa cum ar trebui sa va fie destul de evident pana acum, din experienta voastra de programatori, memoria este in general cel mai problematic bottleneck. Optimizarile prezentate mai sus reduc timpul de executie intr-o oarecare masura, insa ele nu schimba in nici un fel modul in care memoria este accesata in cadrul algoritmului. Cu alte cuvinte, aceleasi locatii de memorie sunt accesate in aceeasi ordine, indiferent daca am operat sau nu optimizarile prezentate. O intrebare interesanta ar fi acum: ce se intampla, daca am schimba ordinea in care se executa buclele? S-ar obtine performante diferite?
Pentru problema noastra, care contine trei bucle, exista asadar sase secvente posibile, si anume: i-j-k, i-k-j, j-i-k, j-k-i, k-i-j, si k-j-i. Fiecare dintre aceste secvente corespunde unui tip diferit de acces la memorie pentru matricele considerate. Deoarece bucla interioara este cea mai des executata, ne vom concentra acum atentia un pic asupra ei. Operatia executata acolo ramane:

c[i][j] += a[i][k] * b[k][j]

Pentru fiecare dintre cele trei matrice, a, b si c, fiecare element poate fi accesat in trei moduri diferite, si anume:

- Constant: accesul nu depinde de indexul buclei interioare
- Secvential: aceesul la memorie este contiguu (adica in celule succesive de memorie)
- Nesecvential: accesul la memorie nu este contiguu (celulele de memorie logic succesive, sunt de fapt adresate cu pauze de dimensiune N)

Astfel, pentru cele sase configuratii, se obtine:

Loop order	c[i][j] +=	a[i][k]	* b[k][j]
i-j-k:	Constant	Secvential	Nesecvential
i-k-j:	Secvential	Constant	Secvential
j-i-k:	Constant	Secvential	Nesecvential
j-k-i:	Nesecvential	Nesecvential	Constant
k-i-j:	Secvential	Constant	Secvential
k-j-i:	Nesecvential	Nesecvential	Constant

Care sunt totusi, comparativ, performantele celor trei moduri de acces? In mod clar, accesul constant este mai bun decat cel secvential – aceste constante in cadrul unor bucle, sunt in general puse in registri, ducand la imbunatatirea performantelor algoritmului, dupa cum s-a aratat in optimizarea 1. Accesul secvential la randul sau, este mai bun decat cel nesecvential, in principal pentru ca utilizeaza considerabil mai bine cache-ul.

Luand in considerare aceste observatii, putem concluziona ca:

- Configuratiile k-i-j si i-k-j ar trebui sa aiba cele mai bune performante
- Configuratiile i-j-k si j-i-k ar trebui sa fie mai proaste decat primele, si
- Configuratiile j-k-i si k-j-i ar trebui sa fie cele mai proaste!

Activitate practica - Ordinea buclelor

Efectiv, care este adevarul? Construiti singuri aceste scenarii si analizati aceasta problema!

Pentru a studia mai in detaliu problema, sa analizam un pic configuratia i-j-k (desi nu este cea mai buna configuratie, cum vedem de mai sus):

```
for (i=0;i<N;i++){
   for (j=0;j<N;j++){
      sum=0;
      for (k=0;k<N;k++)
         sum+=a[i][k]*b[k][j];
      c[i][j] = sum;
   }
}
```

Cate cache-miss-uri sunt generate in acest algoritm, cu aceasta secventa de acces la memorie? In mod evident, aceasta nu este o intrebare usoara. De exemplu: daca fiecare matrice ar fi de doua ori mai mare decat cache-ul, ar avea loc multe incarcari si eliberari de linii, ducand astfel la o formula complicata. Astfel, cel mai simplu aproximam, si consideram ca dimensiunea matricei este mult mai mare decat cea a Cache-ului. Astfel, fie C, numarul de elemente din matrice ce intra in Cache.
Astfel, considerand algoritmul de mai sus (fara optimizarea pentru constante):

```
for (i=0;i<N;i++){
   // Citeste linia i pt a in Cache (Ra)
   // Scrie linia i a lui c in Memorie (Wc)
   for (j=0;j<N;j++){
      // Citeste coloana j a lui b in Cache (Rb)
      for (k=0;k<N;k++){
         c[i][j] += a[i][k] * b[k][j];
      }
   }
}
```

Astfel, daca L este dimensiunea unei linii de Cache: pentru (Ra) obtinem aproximativ N*(N/L) cache-miss-uri, pentru (Wc) la fel, iar pentru (Rb) un dezastruos N*N*N! Acest lucru se intampla deoarece, desi accesul la b este secvential intr-o coloana, matricea este salvata in memorie utilizand row-major! Concluzia este descurajatoare: $2N^2/L + N^3 \rightarrow N^3$ cache-miss-uri! Se adauga la acest aspect si cele $2N^3$ operatii aritmetice, si se ajunge la raportul: operatii aritmetice / operatii cu memoria $\rightarrow 2$. Acest lucru este extrem de rau, deoarece noi stim de la (curs) si de la alte materii, ca arhitecturile calculatoarelor NU sunt echilibrate, si ca operatiile aritmetice sunt de ordine de marime mai rapide decat operatiile cu memoria. De aceea, memoria ramane in continuare bottleneck-ul pentru aceasta implementare a inmultirii de matrice. Pentru a obtine performante mai bune, este necesara obtinerea unui raport considerabil mai mare.

Cum se face insa, ca pentru N^2 elemente intr-o matrice, ajungem la N^3 cache-miss-uri? Pai am stabilit ca acest lucru se datoreaza accesului ineficient al lui b, deoarece se incearca incarcarea coloana cu coloana a matricei!

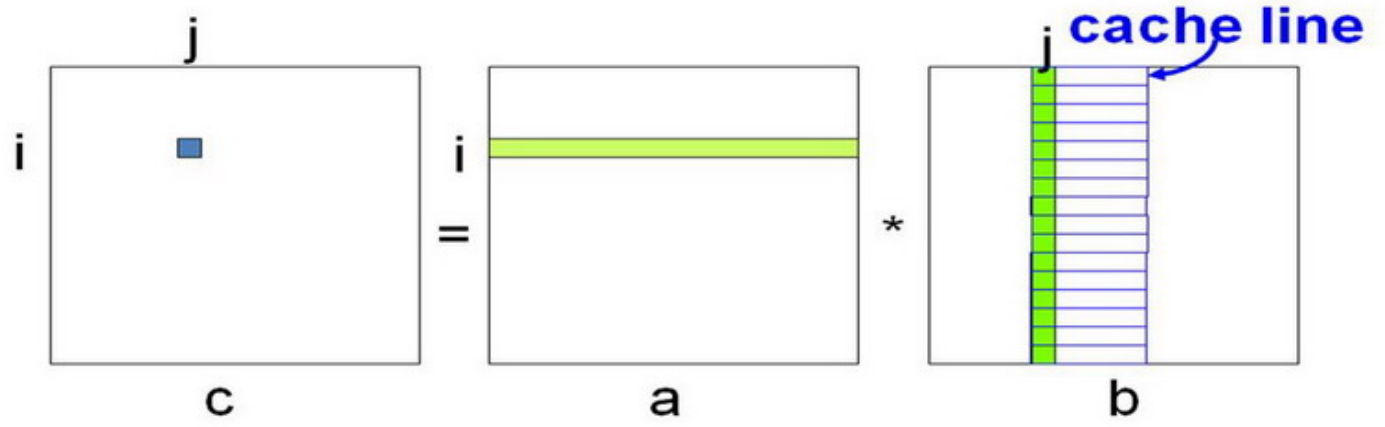
Concluzia acestei analize este ca nu putem spune, doar dupa numarul de operatii efectuate si dimensiunea datelor folosite, daca un algoritm va suferi sau nu din cauza unui bottleneck la memorie.

Solutia este: utilizarea mai ingenioasa a cache-ului.

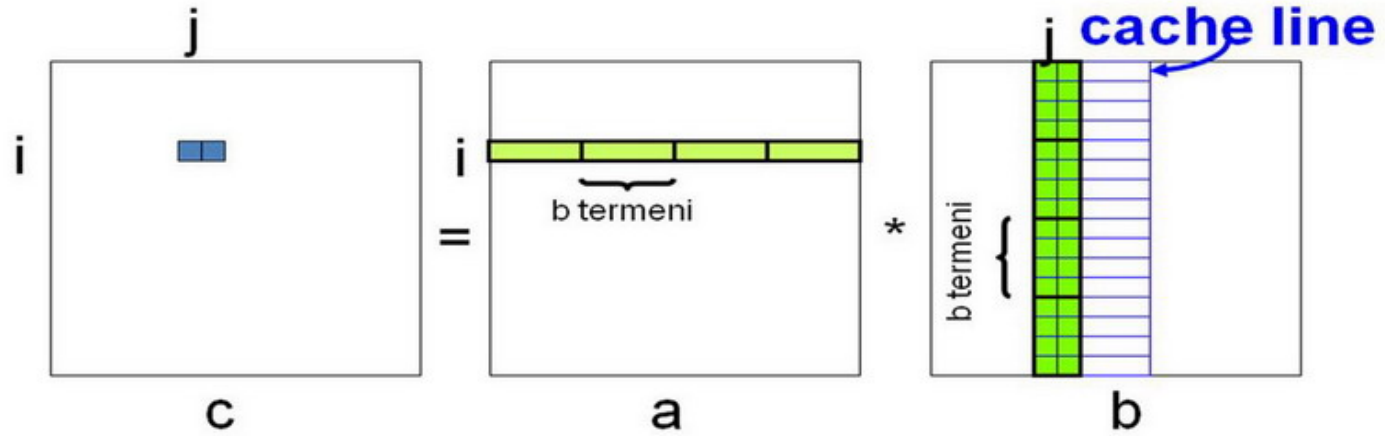
Acest lucru se poate realiza prin reorganizarea operatiilor din cadrul inmultirii de matrice pentru a obtine mai multe cache-hit-uri. Faptul ca adunarea si inmultirea sunt atat operatii asociative, cat si comutative face posibila aceasta reordonare a operatiilor. Acesta este un subiect de cercetare asupra caruia si-au indreptat atentia numerosi cercetatori de-a lungul timpului, generand o multitudine de algoritmi si de teoreme matematice care sa ii sustina. In orice caz, daca vom considera r = raportul intre operatiile aritmetice si operatiile la memorie (cu cache-miss-uri), este evident ca se doreste un r maxim, pentru a elimina bottleneck-ul de la memorie. S-a aratat ca orice reorganizare a acestui algoritm este limitata la r = O(sqrt(C)), unde C este dimensiunea Cache-ului (in numar de elemente ce intra in Cache). Acest lucru arata ca r nu scaleaza cu dimensiunea matricei N, indiferent de impartirea intuitiva a lui $2N^3$ la N^2...

Solutia: “Blocked Matrix Multiplication”

Pentru a rezolva problema accesului in b pentru coloane intregi, se va trece la accesarea unui subset a unei coloane in b, sau a mai multor coloane la un moment dat. Pentru o mai buna intelegere, urmariti desenele de mai jos:

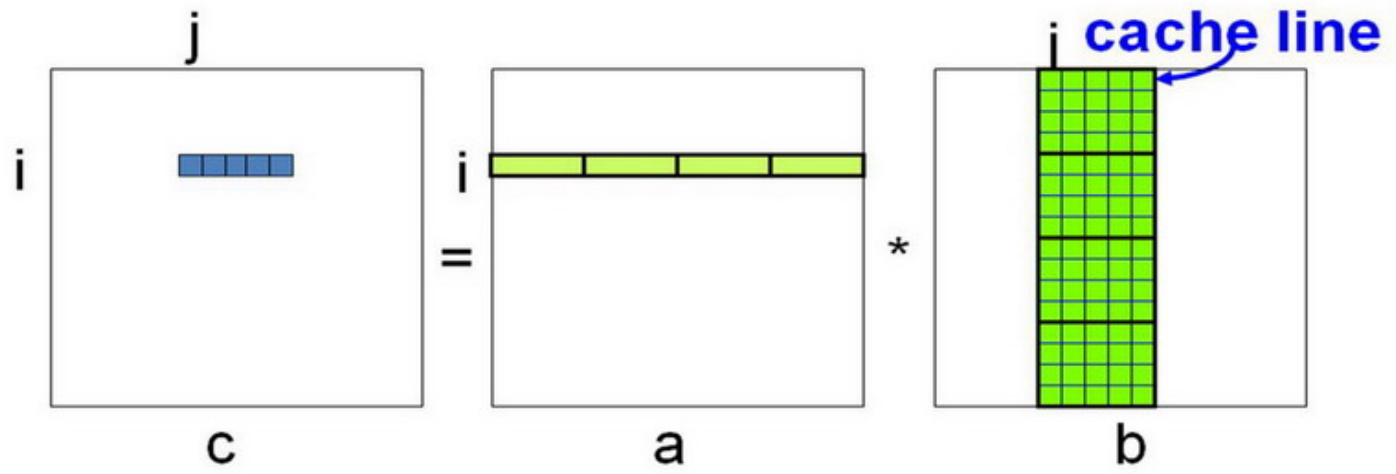


Ideea de baza este refolosirea cat mai buna a elementelor aflate in cache (pentru matricea b). Astfel odata cu calculul lui c[i][j], de ce nu am calcula si c[i][j+1], daca tot se afla in cache si coloana j+1. Acest lucru presupune insa reordonarea operatiilor astfel: calculeaza primii b termeni pentru c[i][j], calculeaza primii b termeni pentru c[i][j+1], calculeaza urmatorii b termeni pentru c[i][j], calculeaza urmatorii b termeni pentru c[i][j+1], etc.

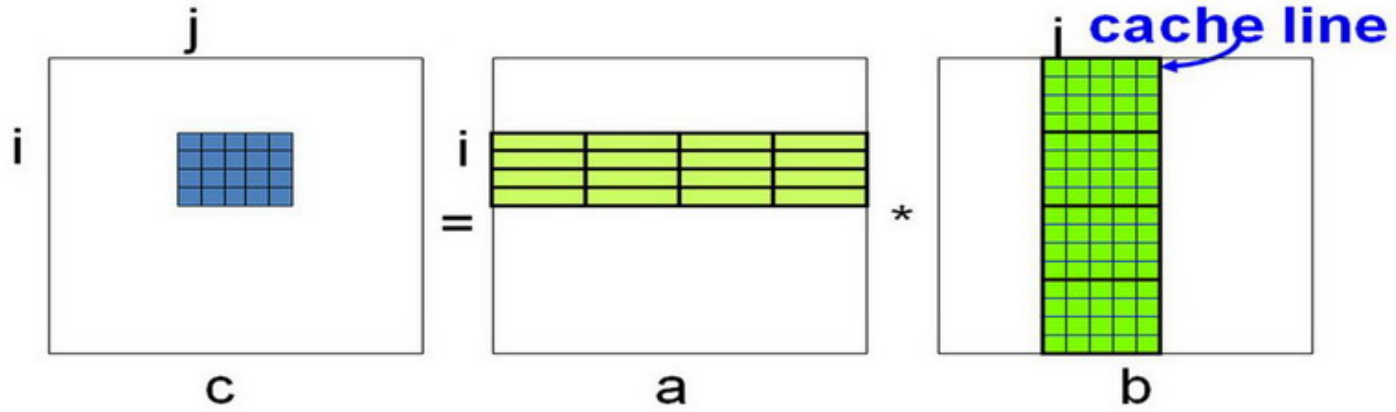


In acest mod, de ce nu am calcula o intreaga sectiune de linie din c, folosind aceste reordonari de operatii?

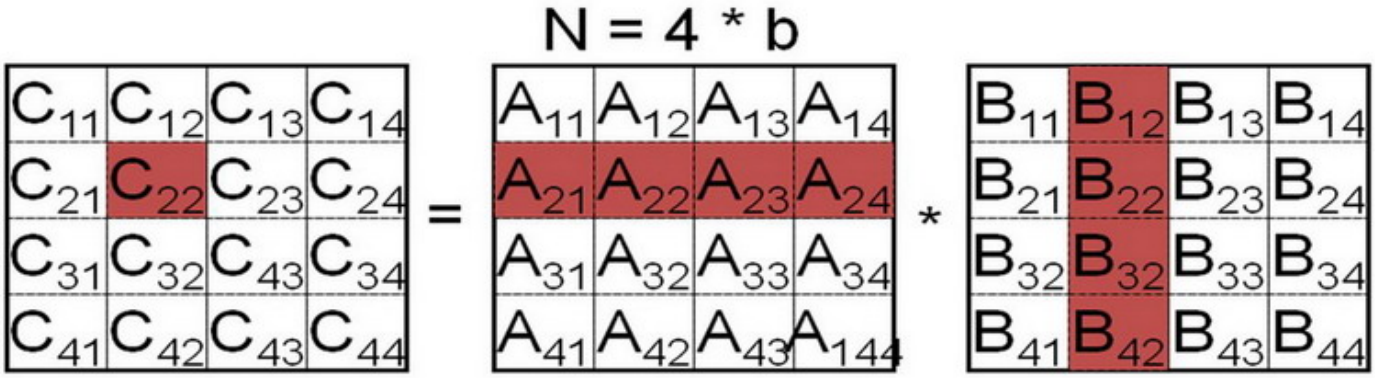
Ce s-ar intampla daca am incerca sa calculam o intreaga linie din c?



Ar insemna ca trebuie sa incarcam toate coloanele lui b in memorie (cache), lucru pe care am incercat sa il evitam aici! Astfel, se vor refolosi doar acele blocuri din b ce au fost deja incarcate. De aici nu ne mai ramane decat sa utilizam intreaga linie de cache din b, si obtinem ideea de baza a algoritmului “Blocked Matrix Multiplication”:



Operatiile trebuie reordonate astfel: calculeaza primii b termeni pentru c[i][j] din blocul C, calculeaza urmatorii b termeni pentru c[i][j] din blocul C, …, calculeaza ultimii b termeni pentru c[i][j] din blocul C. Generalizand:



Pentru a calcula blocul C_{22} folosim formula:

$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42}$

ce presupune patru inmultiri si patru adunari de matrice. Ideea este ca fiecare inmultire opereaza pe un block suficient de mic ca dimensiune astfel incat sa intre in Cache!

Versiunea inmultirii de matrice utilizand metoda bloc si ordonarea i-j-k devine:

```
for (i=0;i<N/b;i++){
   for (j=0;j<N/b;j++){
      for (k=0;k<N/b;k++){
         C[i][j] += A[i][k]*B[k][j]
      }
   }
}
```

unde:

- b este dimensiunea blocului (presupunem ca b divide N)
- C[i][j] este un bloc al matricei C pe linia i si coloana j
- “+=” inseamna adunare de matrice
- si “*” inseamna inmultire de matrice

Ce se intampla cu Cache-miss-urile acum?

```
for (i=0;i<N/b;i++){
   for (j=0;j<N/b;j++){
      // Scrie blocul C[i][j] al lui c in Memorie (Wc)
      for (k=0;k<N/b;k++){
         // Citeste blocul A[i][k] pt a in Cache (Ra)
         // Citeste blocul B[k][j] pt b in Cache (Rb)
         C[i][j] += A[i][k] * B[k][j];
      }
   }
}
```

Pentru (Wc) avem acum (N/b)*(N/b)*b*b Cache-miss-uri, in vreme ce pentru (Ra) si (Rb) avem (N/b)*(N/b)*(N/b)*b*b, astfel ducand la $N^2 + 2N^3/b \rightarrow 2N^3/b$ Cache-miss-uri pentru intregul algoritm. Combinand acest calcul cu faptul ca avem $2N^3$ operatii aritmetice, rezulta un raport $r = 2N^3/b / 2N^3 \rightarrow b$. Dupa cum am stabilit, r trebuie sa fie maxim (mai mare oricum decat 2-ul obtinut in varianta anterioara). Daca mergem pana la cazul extrem, il vom face pe b = N, dar asta nu este viabil, pentru ca atunci suntem din nou la cazul fara blocuri, de la care tocmai venim…

Astfel, acest algoritm functioneaza doar daca blocurile intra in Cache. Acest lucru inseamna ca trei blocuri diferite, de dimensiune b*b, trebuie sa intre in Cache, pentru toate cele trei matrice (a, b si c). Daca C este dimensiunea Cache-ului in elemente de matrice, atunci trebuie sa fie $3b^2 \leq C$ sau $b \leq \sqrt{(C / 3)}$. Astfel, in cel mai bun caz, r-ul trebuie sa fie si el $\sqrt{(C / 3)}$.

Putem astfel spune, pentru diverse procesoare, cunoscand rata de operatii aritmetice la cache-miss-uri r, care este dimensiunea necesara a Cache-ului, pentru a rula acest algoritm, astfel incat procesorul sa NU astepte niciodata memoria:

Activitate practica - BMM & Optimizare pentru Cache

De aceea incercati sa experimentati cele prezentate in acest laborator, in C. Pentru cei interesati, incercati completarea tabelului de mai sus cu dimensiunea Cache-ului pentru procesoarele voastre personale. Acest lucru presupune evident, si o documentare asupra caracteristicilor sistemului propriu (determinarea r-ului, a dimensiunii Cache-ului etc.).

In loc de concluzie

Intelegerea reala a comportamentului unei aplicatii (algoritm), din punctul de vedere al utilizarii cache-ului (si al performantelor in general), este o chestiune complexa, ce necesita multa rabdare si cunostinte diverse. Deseori, aproximatii utile pot fi folosite pentru a imbunatati unele aspecte ale implementarii curente. Utilizarea blocurilor este intalnita deseori in algoritmi si aplicatii ce necesita performante crescute.

Exercitii

1. Optimizarea constantelor si al accesului la vectori (**3p**) folosind matrici liniarizate.
2. Ordonarea buclelor folosind matrici liniarizate. (**3p**)
3. Optimizari pentru Cache folosind matrici liniarizate. (**4p**)
4. In general, nu recomandam alocarea matricelor ca vectori de vectori. Ca bonus va sugeram sa realizati un test unde se face acest tip de alocare si se verifica “performantele” obtinute. Bonus (**2p**).

Resurse

- Responsabilul acestui laborator: Emil Sluşanschi [mailto:emil.slusanschi@cs.pub.ro]
- PDF laborator

Discutii interesante

- De ce este mai rapida procesarea unui vector ordonat? [http://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-an-unsorted-array]

- What every programmer should know about memory.pdf

Valgrind

- http://valgrind.org/docs/manual/cg-manual.html [http://valgrind.org/docs/manual/cg-manual.html]

asc/lab5/index.txt · Last modified: 2017/03/21 09:27 by adriana.draghici