Nastasoiu Florina 335 CA
Tema 2 ASC

IMPLEMENTARE BLAS

Am folosit Double DGEMM de la nivel 3 (**matrix-matrix product**) din BLAS Atlas care realizeaza inmultirea a doua matrice. Se foloseste similar pentru a calcula At x B si Bt x A.

Semnatura functiei este urmatoarea:

```
cblas_dgemm(CblasRowMajor,CblasTrans,CblasNoTrans, m, n, k, alpha, A, k, B, n, beta, C, n);
```

- primii 3 parametrii sunt tipul matricei: *transpusa*, *row-major*
- m, n, k sunt *dimensiunile matricelor* = *N*
- pentru a simula adunarea celor doua produse, **beta = 1.0** aduna produsul curent obtinut la cel anterior (la outputul apelului functiei anterioare)

```
cblas_dgemm(CblasRowMajor,CblasTrans,CblasNoTrans, N, N, N, 1.0, A, N, B, N, 0.0, left, N); //  left = At x B

cblas_dgemm(CblasRowMajor,CblasTrans,CblasNoTrans, N, N, N, 1.0, B, N, A, N, 1.0, left, N); // left += Bt + A
```

In concluzie, implementarea se realizeaza cu 3 instructiuni de tipul **cblas_dgemm.**

IMPLEMENTARE NEOPTIMIZATA

Pentru calculul transpusei, am folosit formula **transA[N*(i%N) + (i/N)] = A[i]** avand in vedere ca matricea are reprezentare liniara.

Pentru produsul a doua matrice, am folosit algoritmul clasic cu 3 loop-uri in cadrul caruia am adaptat formula interioara pentru forma liniara:

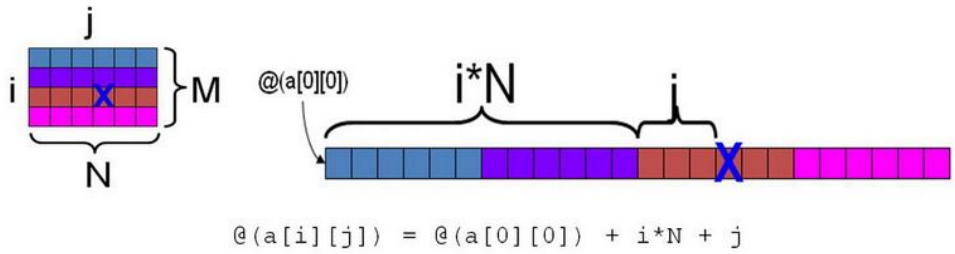
```
suma += a_trans[(k * N) + j] * B[(j * N) + i];
```

Am realizat adunarea celor doua doar pentru triunghiul superior, apoi am revenit la produsul de mai sus pentru operatia finala.

In concluzie, implementarea se realizeaza in O(N^3) insa este evident un aspect care necesita resurse din plin: accesul la vectori prin indecsi. Pentru fiecare astfel de referinta sunt necesare expresii aritmetice complexe (generate de compilator) pentru a afla adresa.

Asadar, algoritmul clasic de inmultire de matrice folosit de mine (care e bazat pe o constructie bidimensioanala) este o abordare este destul de costisitoarea din punct de vedere al performantei.

Accesul presupune doua adunari si o inmultire, iar in cazul de mai sus s-a modificat doar formula de acces la elemente, dar in esenta se simuleaza exact acelasi model in inmultire pentru vectori bidimensionali.



IMPLEMENTARE OPTIMIZATA

Am exploatat faptul ca utilizarea si accesul variabilelor de tip vectorial necesita resurse. Cand programul face o referinta de tipul X[i][j][k], compilatorul trebuie sa genereze expresii aritmetice complexe pentru a calcula adresa. Considerand asezarea row-major, pentru vectori bidimensionali, fiecare acces presupune doua adunari si o inmultire.

Astfel, am sporit viteza programului prin renuntarea la accesele vectoriale prin derefentiere, utilizand in acest scop pointeri.

Nu am folosit indecsi, retinand adresa primului element de pe linie (`double *orig_pa = &a[i][0])` si adresa primului element de pe coloana `(double *pb = &b[0][j])`.

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.

Ordinea buclelor este si ea importanta.

IMPLEMENTARE CU FLAGURI DE OPTIMIZARE

Pentru gcc:

- **`-Ofast`** permite toate optimizarile de tip **`-O3`**. In plus, seteaza pe on **`-ffast-math`** si alte optimizari care creeaza un tradeoff spatiu-timp in favoarea timpului, cum sunt operatiile cu loop-uri. Binarul devine mai mare din pricina acesteia.
- **`-funroll-loops`** este util pentru ca matricea este stocata liniar si este util la operatiile de tip loop pentru a realiza o traversare secventiala a datelor

Pentru icc:

- `-xHost` seteaza un set de optimizari specifice procesorului, iar -ipo permite compilatorului sa analizeze codul pentru a determina unde ii pot fi utile anumite optimizari specifice

ANALIZA PERFORMANTEI

1) Rulari succesive ale unei metode

% in documentul **results.pdf** se gasesc grafice pentru rulari succesive
% rezultatele se gasesc si la urmatoarele **linkuri**, cu informatii in plus

- gcc_neoptimizat
- icc_neoptimizat

- gcc_blas
- icc_blas

- gcc flag optimisation
- icc flag optimisation

- gcc optimisation
- icc optimisation

2) GNU Plots - Analiza comparativa blas vs neopt vs opt_m vs opt_f, gcc și icc

Am folosit gnuplot in cadrul Makefile si Makefile.icc.
Pentru afisare grafic, se poate da una din comenzile:

- **make plot_gcc /* cele 4 metode cu gcc */**
- **make plot_icc /* cele 4 metode cu icc*/**

- **make plot_blas_gcc_vs_icc**
- **make plot_neopt_gcc_vs_icc**

- **make plot_opt_m_gcc_vs_icc**
- **make plot_opt_f_gcc_vs_icc**

ANALIZA PERFORMANTEI - OBSERVATII

1) blas (gcc_blas vs icc_blas)

➔ conform rezultatelor, blas este cel mai rapid din cele 4 metode, avand mai putine instructiuni, deci mai putini cicli de procesor datorita optimizarii cache si paralelismului

- functiile de nivel 3 (DGEMM) folosesc optimizarea pentru code generation impreuna cu alte tehnici, avand O(N^3) operatii pe O(N^2) data
- lucrand pe o platforma cu ierarhie cache, se accelereaza performanta pentru ca optimizarile sunt cache friendly
- matricea patratica de dimensiune N ajuta foarte mult optimizarilor folosite de nivelul 3 blas

Rezultatele gcc_blass (time) vs icc_blas (time) tind sa fie asemanatoare in medie.

2) flag optimisation (icc vs gcc)

➔ conform rezultatelor si in linie cu asteptarile, icc (time) are cu cel putin x2 speed fata de gcc (time)

➔ pe langa optimizarile comune pentru code-speed,aggressive loop transformations (-O3/-Ofast), icc dispune de flag-ul -ipo care este un proces multi-step ce permite compilatorului sa analizeze codul si sa determine unde anume se poate interveni cu optimizari specifice; in plus, -xHost activeaza cel mai inalt nivel de vectorizare disponibil pe procesorul unde se compileaza;

3) opt_m (optimizat) vs neopt (neoptimizat)

conform rezultatelor, opt_m este mai rapid:

➔ opt_m foloseste pointeri pentru accesul la elemente, reducand in mod substantial nr de instructiuni realizate de catre procesor, adica nr de cicli

➔ cele mai accesate valori sunt retinute in register, reducand numarul de accese la memoria cache; astfel avem mai putine cache miss-uri, deci mai putin timp folosit pentru a aduce valori din memoria principala (ceea ce ar fi consumat foarte multi cicli de procesor)

4) opt_m (optimizat) vs opt_f (flag)

conform rezultatelor, opt_f este (putin) mai rapid:

- ➔ se folosesc flag-urile pentru loop unrolling, loop interchange si function inlining care sunt mult mai eficiente
- ➔ numarul de instructiuni executate scade