Tema 2 ASC

Corcodel Florina-Denisa, 336CC

April 15, 2020

1 Implementare BLAS

Am împărțit operația $C = B * A^t + A^2 * B$ în
$C = B * A^t$
$A2xB = A^2 * B$
$C = C + A2xB$

Pentru a înmulți o matrice am folosit funcția cblas-dgemm. Pentru a nu mai calcula A transpus am schimbat al treilea parametru al funcției cblas-dgemm.

Ex: cblas-dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, N, N, N, 1, B, N, A, N, 0, C, N); Codul de mai sus înmulțește matricea B cu A transpus, iar rezultatul este stocat în matricea C.

Pentru a face adunarea finală am folosit un algoritm clasic, cu două loop-uri for și complexitate $O(N^2)$.

2 Implementare neoptimizată

Această implementare folosește algoritmi simpli, iar codul este neoptimizat.

Pentru a genera A transpus se folosește un algoritm cu complexitate $O(N^2)$ Pentru a înmulți două matrici se folosește un algoritm cu trei loop-uri, cu complexitate $O(N^3)$ Calcularea matricii A^2 se face cu o complexitate de $O(N/2 * N^2)$.

Această versiune este de departe cea mai lentă.

3 Implementare optimizată manual

Algoritmii sunt asemănători cu cei de la varianta neoptimizată, dar apar unele diferențe care schimbă timpii de execuție.

Nu se mai calculează A transpus, se înmulțește B direct cu A, dar A va avea indicii de linie și coloană inversați.

Listing 1: Codul neoptimizat pentru a înmulți pe B cu At

```
        int i,j;
        for(i=0; i<N; i++)
        {
                for(j=0; j<N; j++)
                {
                        C[(i*N)+j] = 0;
                        for (k = 0; k < N; k++)
                                C[(i*N)+j] += B[(i*N)+k]
                                        * At[(k*N)+j];
                }
        }
```

Listing 2: Codul optimizat

```
register double* B_p = B;
        register double* C_p = C;

        register int i,j,k;

        for(i=0; i<N; i++)
        {

                register double* A_p = A;
                register double* C_pt = C_p;

                for(j=0; j<N; j++)
                {
                        register double result = 0;
                        register double* B_pt = B_p;
                        register double* A_pt = A_p;

                        for (k = 0; k < N; k++)
                        {

                                result += (*B_pt) *
                                        (*A_pt);

                                A_pt++;
                                B_pt++;
                        }

                        A_p += N;
                        *C_pt = result;
```

```
                C_pt++;
            }

            B_p += N;
            C_p += N;
        }
```

Cum am menționat mai sus, înmuțirea în cazul optimizat se face direct între B și A, fără a avea un A transpus calculat intermediar.

Se folosesc pointeri pentru a accesa liniile și coloanele mai optim. Se folosesc variabile de tip register.
În codul neoptimizat , pentru a afla valoarea înmulțirii se adună direct pe matrice, acest lucru va cauza un număr de scrieri suplimentare de memorie.
Pentru a optimiza acest lucru am declarat o variabilă result, care memorează rezultatul. Variabila result este de tip register, iar scrierea în memorie va avea loc o singură dată per rezultat. Celelalte înmulțiri urmează aceleași reguli de optimizare ca exemplul de mai sus.

Pentru a face adunarea finală, folosim un singur loop for de numărul de repetări N*N. Practic matricea este tratată ca un vector. Timpul obținut pentru această variantă este mult mai bun decât în cazul variantei neoptimizate.

Implementare optimizată extra Flagurile folosite sunt: ftree-vectorize ftree-loop-distribution fno-math-errno funsafe-math-optimizations
fno-trapping-math ffinite-math-only fno-rounding-math fno-signaling-nans fcx-limited-range mtune=nehalem

Flagurile ftree-vectorize și ftree-loop-distribution incearcă paralelizarea loop-urilor. Există o ușoară creștere în performanță în urma acestor optimizări. Flagul mtune=nehalem îi transmite compilatorului să optimizeze și mai mult codul pentru Intel Nehalem.
Celelalte flaguri sunt flaguri care flaguri care afectează operațiile numerelor cu virgulă flotantă.

Flagul funsafe-math-optimizations face optimizări care asumă că nu pot apărea excepții în operațiile cu numere cu virgulă mobilă. Se dezactivează suportul pentru NaN. În urma acestor optimizări operațiile cu numere cu virgulă mobilă nu mai sunt în conformitate cu IEEE, dar totuși programul funcționează cum trebuie.

Există o mică îmbunătățire față de versiunea cu -O3. Timpul este exprimat în secunde.

În urma rulării a mai multe teste pe coada ibm.nehalem, acesta este timpul mediu obținut pentru cele 5 teste:

Nume executabil	N		
	400	800	1200
tema2_blas	0.056	0.404	1.22
tema2_neopt	1.488	11.45	39.8
tema2_opt_m	0.482	3.68	12.23
tema2_opt_f	0.325	2.15	7.8
tema2_opt_f_extra	0.306	2.02	7.5

În urma rulării a mai multor teste cu N variabil, am făcut un grafic care arată timpul de execuție estimat în funcție de N.