Dugăeșescu Andrei 331CC

Tema 2 ASC - Code Optimization

Varianta blas: solver_blas.c

Implementarea acestei variante este cea mai scurtă din punct de vedere al codului folosit (dat fiind faptul că acesta se folosește de funcții deja existente) însa se dovedește a fi deosebit de performantă din punct de vedere al timpului de execuție. În ceea ce privește implementarea, aceasta presupune calculul a 3 înmultiri și efectuarea unei adunări "de mână". În cadrul soluției propuse sunt folosite 3 apeluri ale funcției cblas_dtrmm (funcție responsabilă pentru calculul produsului dintre o matrice superior triunghiulară și o matrice normală) precum și 3 variabile auxiliare: C - rezultatul final, valoarea întoarsă de funcție; A_2 - matrice în care este stocat rezultatul înmulțirii A*A si A_2B - matrice folosită pentru reținerea rezultatului calculului A_2 * B. Înainte de a calcula efectiv produsele necesare obținerii rezultatului final, matricile A_2 și A_2B sunt inițializate astfel: A_2 = A, A_2B = B.

Primul apel al funcției cblas_dtrmm calculează în interiorul lui A_2 produsul A*A: A_2 *= A, iar A_2 = A. Astfel în A_2 se va afla valoarea lui A^2. Al doilea apel al funcției cblas_dtrmm este responsabil pentru calculul lui A_2B = A^2 * B. Inițializarea lui A_2B cu B a fost făcuta tocmai pentru a se putea calcula A^2*B ținând cont de faptul că A^2 este de asemenea o matrice superior triunghiulară, practic A_2B = A_2 * A_2B, iar A_2B = B la început. Astfel, în urma apelului, A_2B va avea valoarea lui A_2 * B. Ultimul apel este responsabil pentru calculul lui B*At. Din cauza faptului că această funcție schimbă valoarea unuia dintre parametrii de intrare (a unei matrici cu ajutorul căreia se calculează produsul) acest apel este ultimul pentru că rezultatul va modifica valoarea lui B. Cu alte cuvinte, acest ultim apel efectuează B = B * At.

După ce au fost calculate toate înmulțirile de matrici, se adună rezultatele obținute și sunt salvate în cadrul matricei C, matrice care va reprezenta rezultatul final.

Varianta neoptimizată: solver_neopt.c

Pentru această variantă fără îmbunătățiri - "de mână", s-au folosit 3 "grupări" de for-uri unde este realizat calculul efectiv. Prima grupare (primele 3 for-uri) sunt responsabile pentru calculul matricei rezultate din A*A, rezultat stocat în interiorul matricei A_2 (nume provenit de la A^2). Pentru calculul lui A*A este parcursă partea superioară diagonalei principale a matricei A, deoarece A este o matrice superior triunghiulară iar pentru calculul efectiv al unei valori din A_2 se parcurg elementele din A care se știu că sunt nenule, adică până în diagonala principală. Cu alte cuvinte, pentru fiecare element ce se dorește a fi calculat din matricea rezultat A_2 sunt folosite doar acele valori aflate în partea nenulă a matricei A (din partea care conține numere diferite de 0). A doua grupare de for-uri este responsabilă pentru calculul (A*A)*B (A_2 * B). Rezultatul acestui calcul este păstrat în cadrul matricei C pentru ca restul calculelor să fie adunate direct în C pentru a avea rezultatul final direct în valoarea întoarsă de funcție - C (se evita astfel reținerea rezultatului A_2 * B într-o matrice care se va aduna la matricea C). Dacă în cazul lui A*A rezultatul era de asemena o matrice superior triunghiulară, în cazul A_2 * B rezultatul nu mai este nepărat tot o matrice superior triunghiulară (dacă B este

o matrice normală atunci rezultatul va reprezenta tot o matrice normală). Cu toate acestea, nu este nevoie să luam în calcul valorile de 0 cunoscute din interiorul matricei A_2. Ultima grupare de for-uri (a treia) este responsabilă pentru calculul B*At (At - transpusa lui A). Calculul respectiv este adunat la valorile deja calculate din C pentru a obține rezultatul final. În timpul înmulțirii acestor două ultime matrici: B și At putem ține cont de faptul că transpusa unei matrici superior triunghiulare este o matrice inferior triunghiulară, evitând astfel includerea zero-urilor cunoscute în calculul rezultatului final. Odată cu terminearea ultimei grupări de for-uri rezultatul expresiei B*At + A^2 + B se găsește în matricea C.

Varianta optimizată a versiunii neoptimizate (solver_neopt.c): solver_opt.c

Această implementare are la bază funcționalitatea descrisă mai sus în cadrul versiunii solver_neopt. Pentru a obține îmbunătățiri în ceea ce priveste performanțele, în principal scăderea timpului de execuție, au fost folosite o serie de optimizări. După cum a fost menționat și mai sus, din punct de vedere structural implementarea este împărțită în 3 "grupări" de for-uri, fiecare responsabilă pentru calculul unei înmulțiri de matrici. Ordinea și funcționalitatea acestora este identică variantei neoptimizate. Pentru prima grupare, cea responsabilă pentru calculul lui A*A, s-a optat în primul rând pentru o rearanjare/reordonare a buclelor pentru creșterea performanțelor. În cadrul acestei grupări ordinea folosită este i-k-j. Pe lângă aceasta, în locul folosirii accesului clasic la memoria matricilor, această implementare se folosește de pointeri pentru a reduce numărul de operații necesare pentru accesul la elementele matricei; de asemenea, pointerii folosiți cel mai frecvent sunt precedați de cuvântul cheie register, pentru a eficientiza cât mai mult operațiile cu aceștia. Cum proprietățile matricilor nu s-au schimbat, în continuare se evită lucrul cu elementele de sub diagonala principală (întrucat este cunoscut faptul că valorile acestora sunt 0).

Pentru cea de a doua grupare, responsabilă pentru calculul A_2 * B, se aplică aceeași strategie cu cea menționată mai sus - reordonarea buclelor din i-j-k în i-k-j, folosirea de pointeri pentru a eficientiza accesul la memorie și evitarea calculului pentru valorile care se cunosc a fi 0. În ceea ce privește ultima grupare, cea în care se calculeaza B*At, aceasta prezintă ordinea clasică a buclelor (i-k-j) din cauza unei dependințe a indicilor. Pe lângă pointerii folosiți în acest caz, dată fiind structura buclelor, este folosită o variabila (suma) pentru a calcula valoarea finală ce va trebui adăugată în rezultat, limitând astfel pe cât posibil accesul la memorie. O altă optimizare valabilă pentru toate cele 3 grupări este reprezentată de folosirea tehnicii de loop unrolling pentru bucla cea mai interioară pentru a efectua în total un număr semnificativ mai mic de iterații, fiecare iterație însă având totuși o serie mai mare de instrucțiuni de executat, însă per total se efectuează mai puține verificări de control a buclei, se elimină din penalitățile de branch opținându-se un trade-off între dimensiunea codului - prin introducerea de linii adiționale de cod, în acest caz 10 pentru fiecare grupare dat fiind unroll-ul de 10, și timpul de execuție.

Fiecare dintre optimizările mai sus menționate au contribuit în ceea ce privește reducerea timpului de execuție însa nu în ponderi egale. Folosirea de pointeri a redus substanțial timpul de executie (aproximativ 10s vs 5s - timpi locali), iar loop unrolling-ul a redus timpul de execuție suficient de mult cât aproape să îl înjumătățească (beneficiile au fost vizibile de la un unroll de 4 la un unroll de 10).

Varianta de optimizare cu ajutorul flag-urilor de compilare: opt_f_extra

Această variantă de optimizare are la bază codul descris în cadrul sursei solver_neopt.c a cărui funcționalitate a fost descrisă într-un paragraf anterior. Optimizarea în sine constă în folosirea unor flag-uri de compilare care vizează reducerea timpului de execuție. În această secțiune se va discuta despre flag-urile folosite care nu sunt deja incluse în -O3, adică: -mtune=native -funroll-loops -fno-math-errno -funsafe-math-optimizations -ffinite-math-only și -fexcess-precision=fast.

-mtune=native: folosirea acestui flag provine de la observarea îmbunătățirii semnificative a performanței prin intermediul folosirii unui alt flag, care îl include pe acesta și anume: -march=native. -march=cpu-type este un flag responsabil pentru generarea de instrucțiuni specifice tipului de procesor specificat (native presupune selectarea procesorului întâlnit pe mașina pe care se realizează compilarea). Intuitiv, specializarea codului pentru mașina pe care va rula pare o idee bună iar rezultatele confirmă acest lucru. Din cauza faptului că -march va conduce la generarea unui cod specific pentru procesorul cu ajutorul căruia a fost compilat codul, acest lucru însemnă faptul că odată compilat pe un anumit tip de procesor executabilul obținut în final nu va rula pe altă mașină dacă aceasta nu dispune de același tip de procesor. Din aceste considerente s-a optat pentru folosirea -mtune, flag care optimizează codul generat, este mai puțin agresiv din punct de vedere al specializării codului pentru un anumit procesor, iar performanțele obținute cu ajutorul acestui flag par să nu se fi diminuat față de folosirea -march. Cu toate acestea, este indicat ca înainte de rularea executabilului pentru a observa timpii de rulare, codul să fie compilat pe mașina de testare, în cazul acestei teme, coada nehalem. Pentru facilitarea acestui lucru în cadrul arhivei este regăsit un script de testare care va compila codul pe coadă și va genera câte un fișier de output pentru fiecare variantă, fișier care va conține timpii obținuți de varianta respectivă. Pentru folosirea acestui script se dorește folosirea urmatoarei comenzi:

```
////////////////////////////////////////////////////////////////////////////
////////////////////// RULARE SCRIPT //////////////////////
////////////////////////////////////////////////////////////////////////////
   qsub -cwd -q ibm-nehalem.q -b n check.sh
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//////////////// RULAREA UNEI VARIANTE ////////////
////////////////////////////////////////////////////////////////////////////
      qsub -cwd -q ibm-nehalem.q -b y ./exec input
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

Fișierele generate se vor numi: blas.txt, neopt.txt, opt_m.txt, opt_f.txt și opt_f_extra.txt.

-funroll-loops: dat fiind faptul că soluția în sine constă într-o serie de loop-uri ale căror capete (indici de start și de sfârșit) variază, un flag care vizează o tehnica de optimizare care s-a dovedit deosebit de eficientă în cadrul versiunii optimizate "de mână" (solver_opt.c) părea o alegere naturală. Îmbunătățirea în ceea ce privește performanța a fost totuși minoră - deși timpul de execuție a scăzut, speed up-ul a fost marginal.

Următoarele flag-uri fac parte dintr-o colecție de flag-uri prin intermediul căreia au fost identificate. Flag-ul -ffast-math conține restul de flag-uri folosite dar mai activează și alte flag-uri decât cele menționate.

-fno-math-errno: acest flag, conform documentației, ar trebui să conducă la o creștere a vitezei de rulare dat fiind faptul că dezactivează flag-ul errno pentru anumite instrucțiuni matematice. Tot în cadrul documentației găsite pentru acest flag se sugerează faptul că pentru un cod care nu necesită folosirea exactă a implementării IEEE, folosirea acestui flag ar trebui să îmbunătățească performanțele.

-funsafe-math-optimizations: acest flag s-a dovedit a fi unul important pentru reducerea timpului de execuție. Acest flag permite folosirea unor anumite optimizări pentru operațiile în virgulă mobilă presupunând faptul că argumentele (valorile din cadrul operațiilor în virgulă mobilă) sunt valide (de unde și numele de unsafe). Presupunând totuși că în cadrul temei sunt generate matrici valide din punct de vedere al conținutului (presupunem ca matricile conțin doar numere) acest flag aduce îmbunătățiri vizible în timpul de execuție.

-ffinite-math-only: asemănător flag-ului de mai sus, acesta permite optimizări pentru operațiile în virgulă mobilă (în cazul acesta calculul valorilor matricelor) presupunând faptul că valorile cu care se lucrează (valorile din matrici) nu sunt NaN sau Inf (Not a Number sau infinit - numărul invalid obținut spre exemplu în urma unei operații de împărțire la 0). Cum presupunerea faptului ca matricile conțin valori valide pentru tipul de date folosit (double) nu pare a fi una foarte ambițioasă s-a optat pentru folosirea acestui flag.

-fexcess-precision=fast: această opțiune permite o formă de control asupra preciziei pentru operațiile în virgulă mobilă. Folosirea acestei opțiuni (fast) înseamnă că operațiile pot fi efectuate cu o precizie mai largă decât tipurile specificate; acest lucru poate conduce la un cod mai rapid însă mai puțin previzibil din punct de vedere al momentului în care se realizează rotunjiri la tipul specificat. Din punct de vedere al preformanțelor acest flag este de ajutor însă nu are un impact la fel de mare precum -mtune sau -funsafe-math-optimization.

În concluzie, pentru obținerea unui speed-up vizibil față de folosirea doar a opțiunii -O3, cele mai importante flag-uri folosite sunt reprezentate de -mtune=native și -funsafe-math-optimizations. Prin presupunerea validității datelor (o presupunere care nu ar trebui să fie una foarte îndrăzneață) și prin optimizarea instrucțiunilor pentru codul oferit s-a putut atinge un timp de execuție mai rapid (din testele efectuate pe coadă cu aproximativ 11-14% mai rapid).

În continuare vor fi prezentate câteva grafice descripitive pentru performanțele referitoare la timpul de execuție pentru cele 5 moduri de calcul ale expresiei $C=B\times A_t + A_2\times B$. Pentru realizarea graficelor au fost înregistrate valori ale timpului de execuție pentru 6 rulări ale fiecăreia dintre cele 5 variante (blas, neopt, opt_m, opt_f și opt_f_extra). Tabelele cu aceste valori sunt următoarele:

N	blas run 1	blas run 2	blas run 3	blas run 4	blas run 5	blas run 6	Average
400	0.048624	0.042877	0.045068	0.03277	0.048315	0.04769	0.044224
800	0.203501	0.241703	0.242778	0.207297	0.202471	0.203004	0.21679233
1200	0.656465	0.656673	0.657331	0.658274	0.67508	0.657013	0.66013933
1400	1.03096	1.031224	1.03203	1.035478	1.045062	1.046176	1.03682167
1600	1.525725	1.525481	1.52485	1.531224	1.523821	1.525621	1.52612033

N	neopt run 1	neopt run 2	neopt run 3	neopt run 4	neopt run 5	neopt run 6	Average
400	0.648051	0.681577	0.675707	0.630148	0.672361	0.681375	0.66486983
800	5.109627	5.116908	5.121133	5.038324	5.11961	5.034413	5.0900025
1200	17.055424	16.75197	16.788427	16.965994	16.809999	17.04055	16.9020607
1400	27.588505	27.6542	27.753664	27.563389	27.597616	27.567673	27.6208412
1600	47.677017	46.839638	47.710045	47.514465	47.313126	47.658009	47.45205

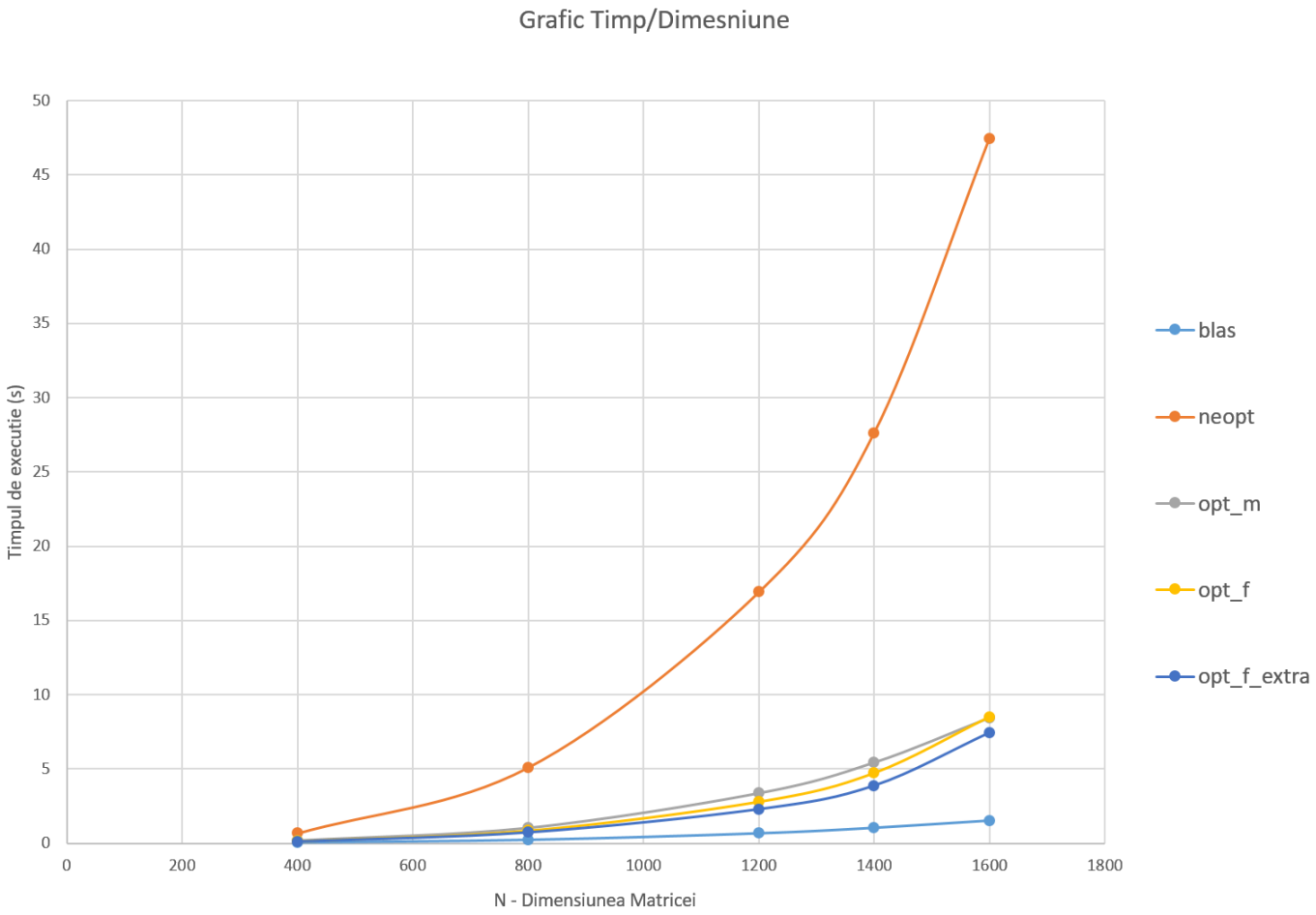
N	opt_m run 1	opt_m run 2	opt_m run 3	opt_m run 4	opt_m run 5	opt_m run 6	Average
400	0.170987	0.12623	0.128568	0.139826	0.173467	0.164529	0.15060117
800	1.037989	0.999634	0.996068	1.000347	1.010438	0.996241	1.00678617
1200	3.345425	3.346522	3.37187	3.348186	3.343791	3.340089	3.34931383
1400	5.397864	5.405167	5.441648	5.428729	5.403303	5.441298	5.41966817
1600	8.475395	8.387436	8.46864	8.437127	8.439124	8.469693	8.44623583

N	opt_f run 1	opt_f run 2	opt_f run 3	opt_f run 4	opt_f run 5	opt_f run 6	Average
400	0.130954	0.134062	0.151201	0.112352	0.144985	0.140081	0.13560583
800	0.816232	0.877832	0.854238	0.817754	0.857105	0.874581	0.84962367
1200	2.788615	2.748814	2.789375	2.806575	2.827855	2.788992	2.79170433
1400	4.73082	4.683425	4.758782	4.751099	4.703902	4.707545	4.7225955
1600	8.527223	8.387289	8.562209	8.50433	8.415972	8.519985	8.486168

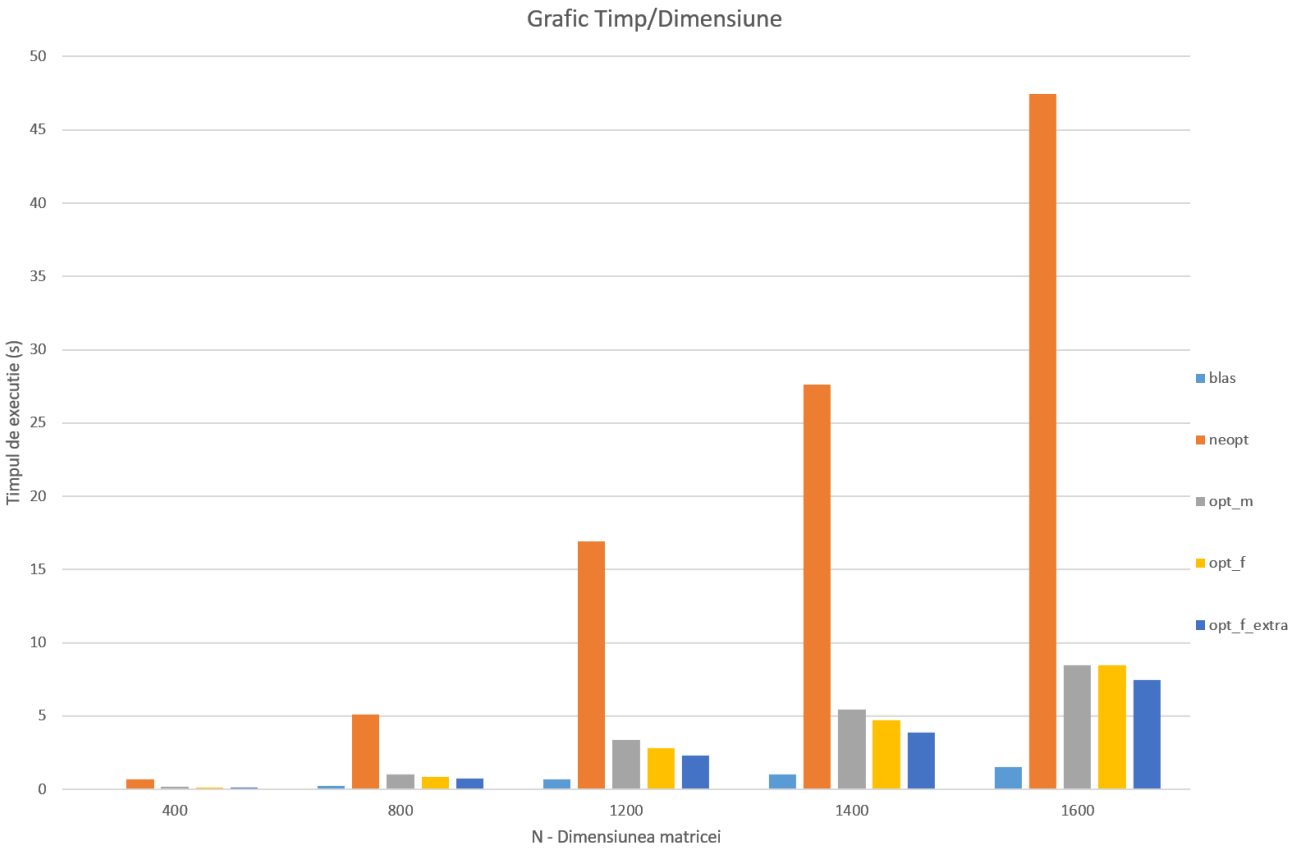
N	opt_f_extra run 1	opt_f_extra run 2	opt_f_extra run 3	opt_f_extra run 4	opt_f_extra run 5	opt_f_extra run 6	Average
400	0.108673	0.109343	0.100707	0.095547	0.119901	0.108277	0.10707467
800	0.710415	0.751874	0.694472	0.691801	0.746264	0.741502	0.72272133
1200	2.376057	2.22867	2.318405	2.312344	2.226591	2.249702	2.28529483
1400	3.857394	3.870419	3.884011	3.832379	3.910993	3.884848	3.87334067
1600	7.492222	7.288666	7.456815	7.441002	7.510702	7.43888	7.43804783

Pentru realizarea graficelor au fost folosite valorile medii ale timpilor de rulare pentru fiecare dintre cele 5 variante (pentru fiecare dimensiune a matricii). Aceste valori se regăsesc în tabelul următor:

N	blas	neopt	opt_m	opt_f	opt_f_extra
400	0.044224	0.664869833	0.150601167	0.135605833	0.107074667
800	0.216792333	5.0900025	1.006786167	0.849623667	0.722721333
1200	0.660139333	16.90206067	3.349313833	2.791704333	2.285294833
1400	1.036821667	27.62084117	5.419668167	4.7225955	3.873340667
1600	1.526120333	47.45205	8.446235833	8.486168	7.438047833



Grafic 1



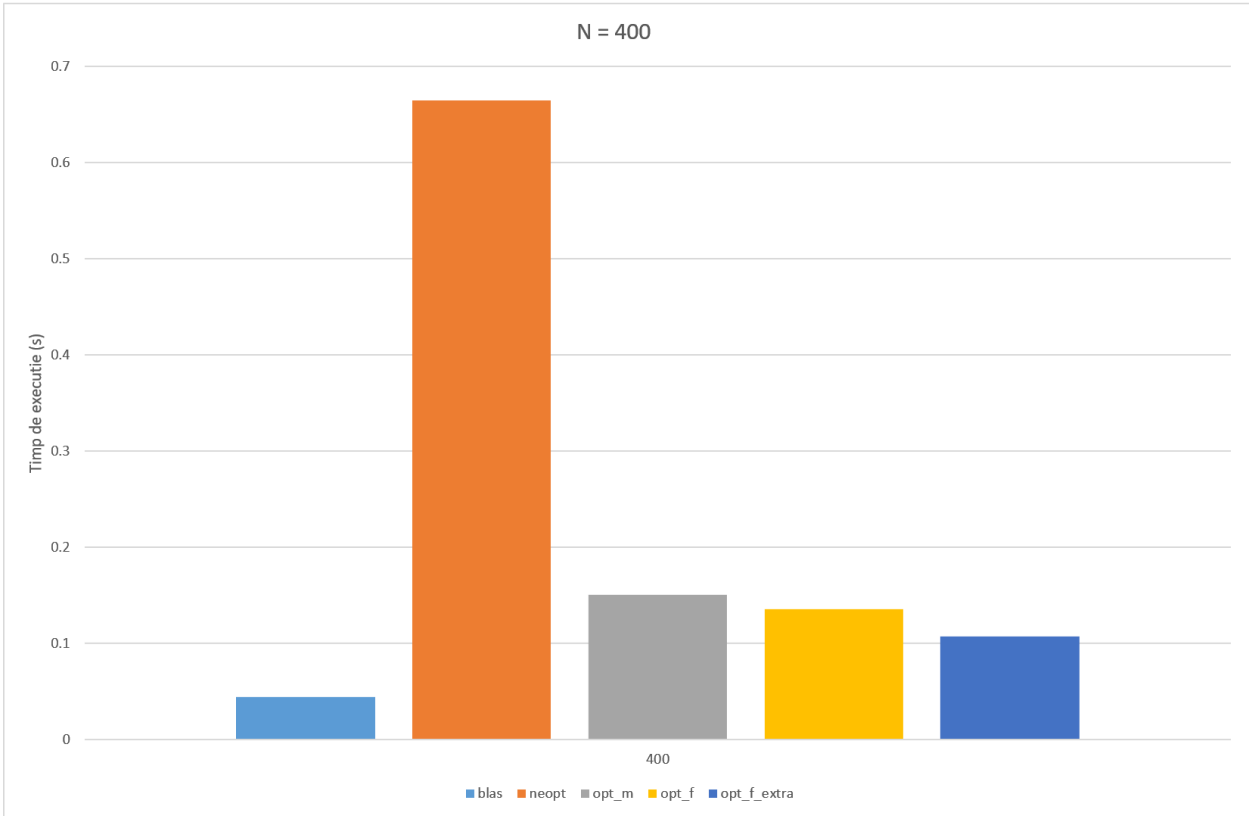
Grafic 2

După cum era de așteptat, varianta neoptimizată se dovedește a avea cele mai slabe performanțe din punct de vedere al timpului de rulare, fiind de altfel varianta pentru care timpul de execuție se înrăutățește cel mai rapid odată cu creșterea dimeniunii matricilor de lucru. După cum se poate observa din graficul 1, timpul de execuție pentru varianta neoptimizată (neopt) descrie o creștere cu un caracter mult mai puțin liniar decât celelalte variante, având mai degrabă un caracter exponențial. Pentru a putea determina cu certitudine comportamentul implementării neoptimizate ar fi nevoie de o serie de teste mai ample, cu matrici tot mai mari, însă chiar și din testele efectuate se poate observa impactul pe care optimizările îl au asupra timpului de execuție. Prin comparație, varianta opt_m descrie exact aceeași complexitate cu vairanta neoptimizată însă prin reducerea numarului de accese la memorie, prin eficientizarea folosirii cache-ului și prin ajutarea compilatorului de a pregăti instrucțiuni prin tehnici precum loop-unrolling se remarcă o scădere drastică a timpului necesar pentru efectuarea aceluiași calcul.

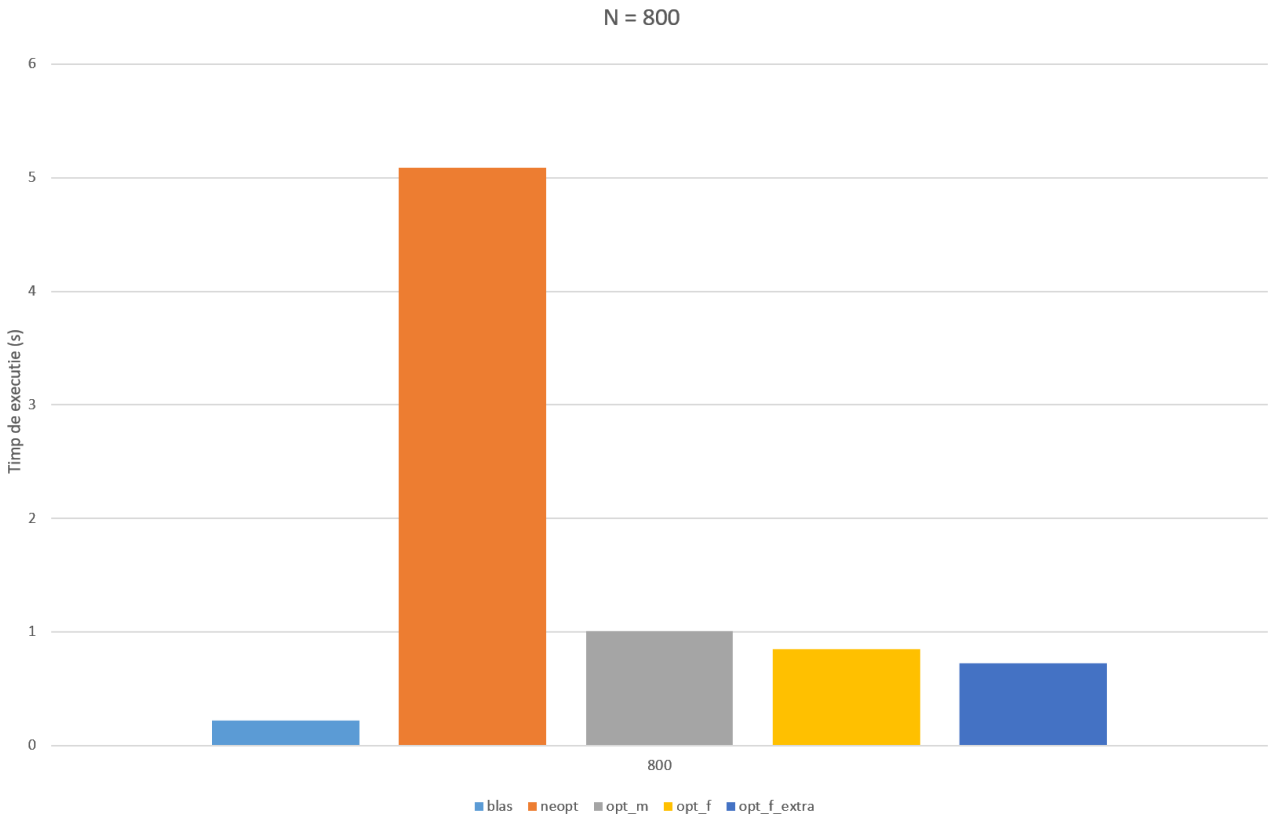
Chiar dacă performanțele obținunite cu ajutorul optimizărilor aduse prin intermediul codului sursă sunt impresionante prin comparație cu varianta neoptimizată, un lucru remarcabil este nivelul de optimizare adus de către compilator în cazul variantelor opt_f și opt_f_extra. Dacă varianta opt_m prezintă un speed-up de aproximativ 80% (pentru N = 1200 folosind timpii medii de rulare) cu ajutorul unor modificări semnificative ale codului sursă, opt_f prezintă un speed up de aproximativ 83% doar cu ajutorul unui flag de compilare (-O3) pornind de la codul corespunzător variantei neoptimizate.

Pentru variantele opt_m, opt_f și opt_f_extra rezulatele sunt relativ apropiate, fapt ce reiese și din graficele 1 și 2 însă pentru implementarea folosind BLAS timpul de execuție este considerabil mai mic, lucru de așteptat din partea unei biblioteci de specialitate. Datorită timpilor foarte mici de execuție obținuți pentru implementarea BLAS, în continuare vor urma o serie de grafice pentru fiecare dimensiune de testare a matricilor de intrare pentru evidențierea ceva mai clară a performanței BLAS. O cuantificare a vitezei BLAS în comparație cu celelalte implementări se poate observa din faptul că timpul necesar completării ultimului test (N = 1600) este de 1.5 ori mai rapid decât cea mai rapidă implementare pentru N = 1200 (opt_f_extra); față de varianta neoptimizată speed up-ul este de 96% (pentru N = 1200 folosind timpii medii de rulare).

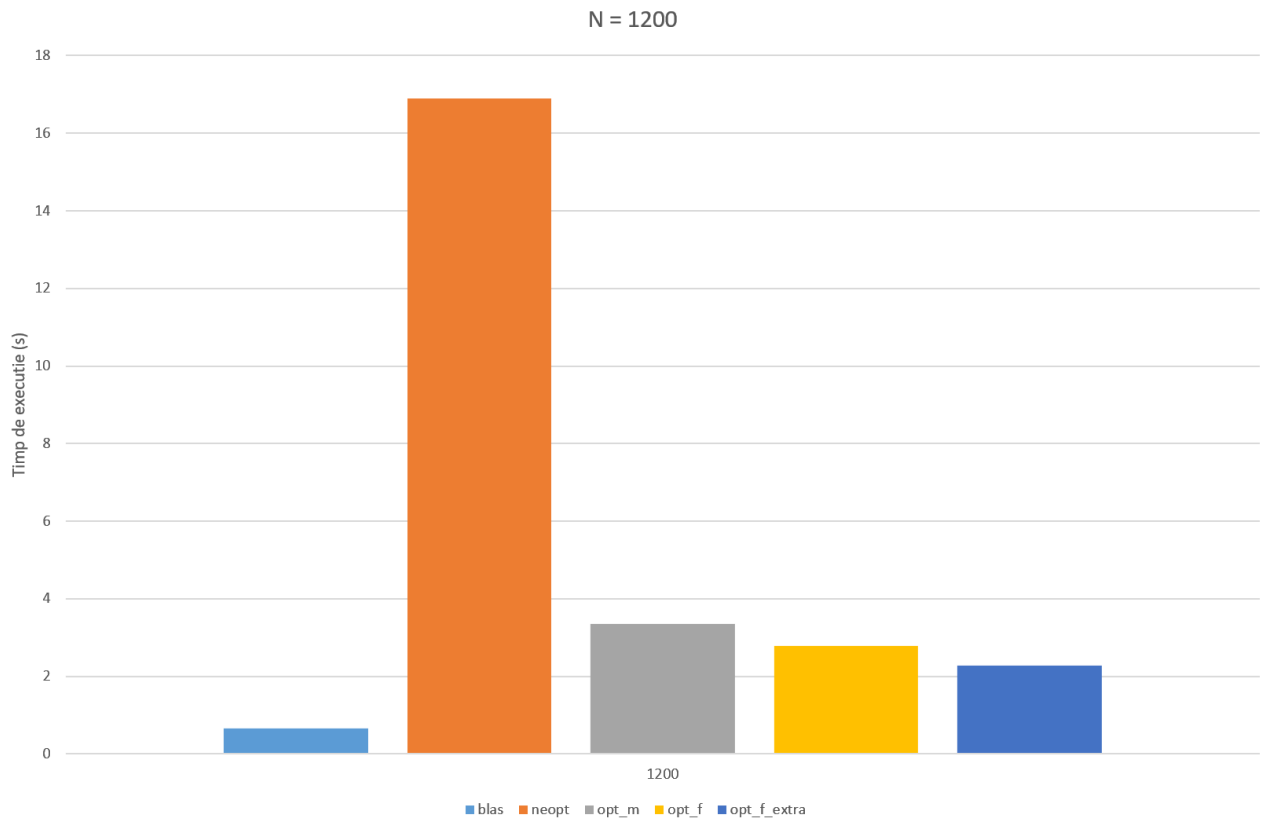
Pentru următoarele grafice este recomandată insepctarea valorilor de pe axa OY întrucât diferențele de timp nu apar neapărat proporționale de la un grafic la următorul. Pentru o privire asupra proporționalității timpilor de rulare de la o dimensiune la alta pentru cele 5 variante este recomandată folosirea graficului 2.



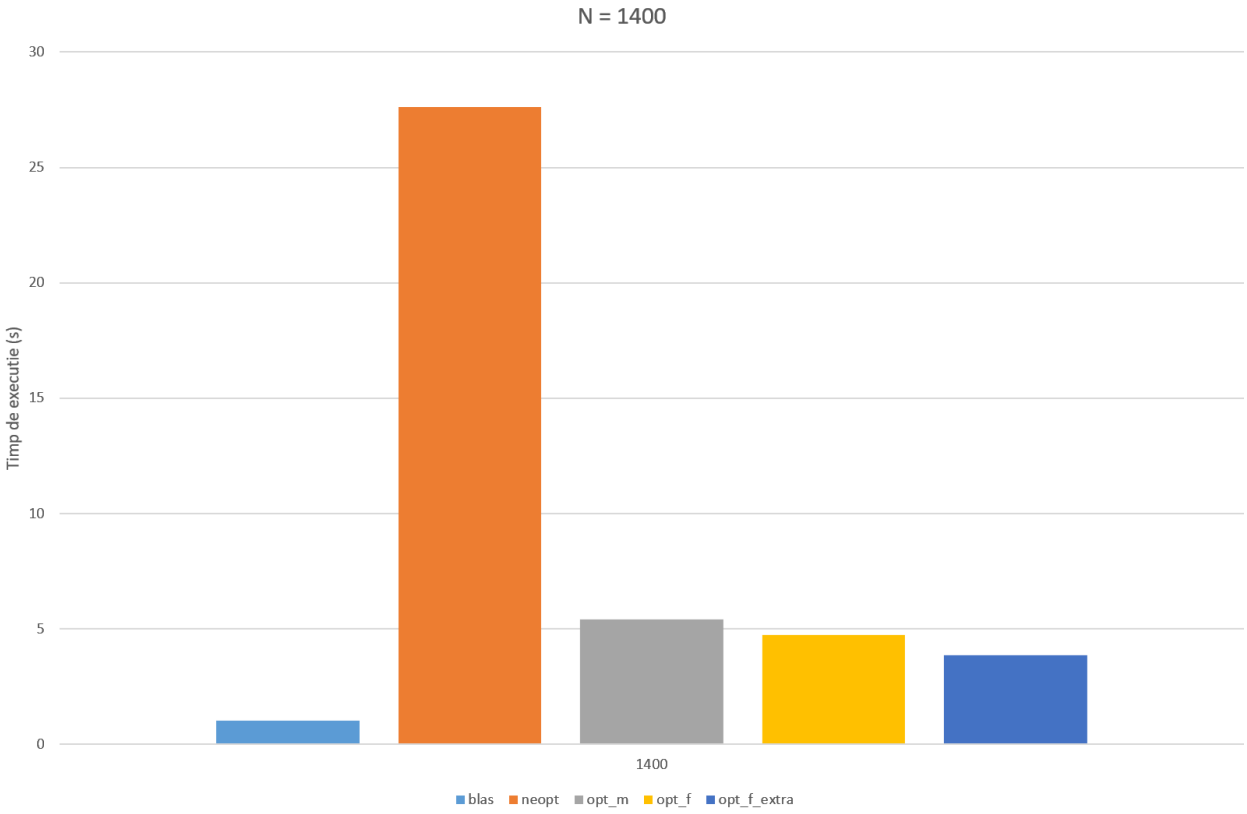
Grafic 3


N = 800
6
5
4
3
2
1
0
Timp de executie (s)
800
blas
neopt
opt_m
opt_f
opt_f_extra

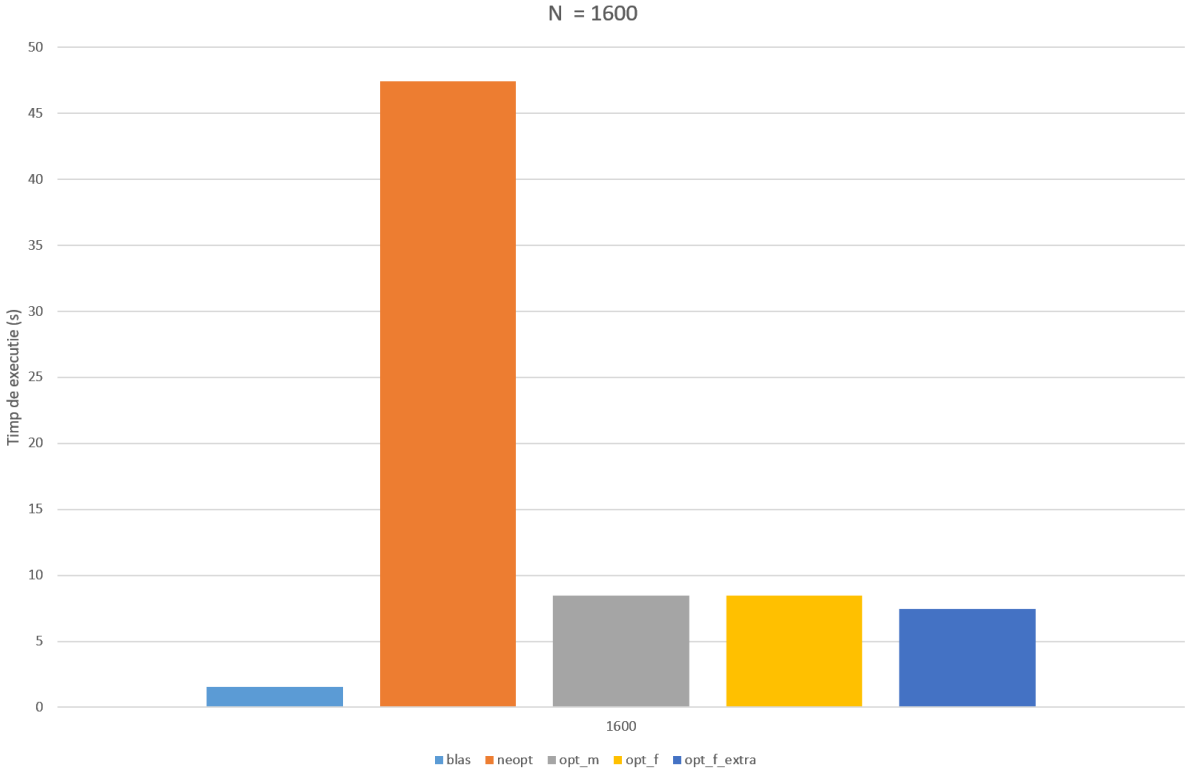
Grafic 4


N = 1200
18
16
14
12
10
8
6
4
2
0
Timp de executie (s)
1200
blas
neopt
opt_m
opt_f
opt_f_extra

Grafic 5



Grafic 6



Grafic 7

Pentru realizarea acestor grafice au fost folosiți timpi medii de rulare. Un astpect important este faptul că în timpul înregistrării datelor pentru aceste grafice nu s-au constatat variaținii semnificative de la o rulare la alta, iar pentru fiecare test diferențele relative de la o metodă de implementare la alta au rămas constante (pentru fiecare testare s-a păstrat același clasament al performanțelor: blas, opt_f_extra, opt_f, opt_m, neopt).

În concluzie, pe baza rezultatelor obținute se poate observa importanța și impactul pe care îl poate avea optimizarea unui program. Fiecare variantă propusă realizează, în esență, același calcul folosind aceleși date de intrare însă timpul necesar comlpetării acestui task poate fi drasctic îmbunătățit pornind de la o implementare de bază, în cadrul acestei teme solver_neopt. În funcție de implementare, programul final se poate dovedi eficient din punct de vedere al timpului de execuție, al timpului de scriere al programului efectiv, poate fi scalabil sau nu, performanțele care se doresc a fi atinse dictând ce fel de optimizări ar trebui aduse programului; în cadrul acestei teme a fost vizată obținerea unui timp de execuție cât mai bun.