UNIVERSITATEA DIN CRAIOVA
FACULTATEA DE AUTOMATICĂ, CALCULATOARE ȘI ELECTRONICĂ

DEPARTAMENTUL DE CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI

RAPORT FINAL

Radu-Mihai Capalb

MODELAREA ȘI EVALUAREA PERFORMANȚELOR

CRAIOVA

CUPRINS

1. PROPUNERE DE PROIECT

Proiectul practic constă în selectarea unui șir de operații matriceale și implementarea acestora printr-un algoritm rudimentar, în faza inițială. Ulterior, algoritmul va fi optimizat succesiv, rezultând alte două versiuni ale acestuia, scopul final fiind efectuarea unei analize comparative din punct de vedere al performanței.

Calculul matriceal care trebuie realizat este următorul:

$$R = A^t \times A + A \times B \times B^t, \text{ unde:}$$

- A și B sunt matrice pătratice, de tip double, de dimensiune N x N
- A este o matrice superior triunghiulară
- A^t este transpusa lui A și B^t este transpusa lui B
- $\times$ este operația de înmulțire
- $+$ este operația de adunare

Cele trei implementări ale calculului matriceal de mai sus sunt următoarele:

- **neopt** - o variantă brută, care efectuează operațiile matriceale în modul clasic.
- **opt** - o variantă îmbunătățită a versiunii **neopt**. Îmbunătățirea are în vedere exclusiv optimizarea codului pentru a obține performanțe mai bune.
- **blas** - o variantă care folosește una sau mai multe funcții din biblioteca BLAS (Basic Linear Algebra Subprograms) (BLAS fără an) pentru realizarea operațiilor de înmulțire și adunare.

De asemenea, fiecare dintre cele trei implementări va ține cont de faptul că A este o matrice superior triunghiulară.

1.1. Specificații

Proiectul are drept produse finite trei fișiere executabile, câte unul pentru fiecare implementare a calculului.

Din punctul de vedere al specificațiilor funcționale, proiectul trebuie să fie capabil să primească un argument, ce reprezintă numele fișierului de intrare în care se află datele necesare rulării programului. Structura fișierului de intrare este următoarea:

- Pe prima linie se află numărul de teste.
- Pe următoarele linii se află descrierea fiecărui test: valoarea lui N, seed-ul folosit la generarea datelor de test, calea către fișierul de ieșire în care să se scrie matricea rezultat.

Din punctul de vedere al specificațiilor non-funcționale, proiectul trebuie să îndeplinească următoarele criterii:

- Să calculeze într-un mod corect din punct de vedere matematic operațiile matriceale.
- Să calculeze într-un mod eficient din punct de vedere al timpului de execuție operațiile matriceale.
- Să folosească într-un mod corect accesul la memorie - fără leak-uri de memorie.
- Să folosească într-un mod eficient accesul la memorie.

2. INSTRUMENTE SOFTWARE UTILIZATE

Proiectul va fi implementat folosind limbajul de programare C. Am luat această decizie întrucât C permite lucrul cu structuri low-level, precum pointerii și regiștri.

Pentru implementarea variantei **blas** voi folosi biblioteca BLAS. Basic Linear Algebra Subprograms furnizează rutine care stau la baza operațiilor vectoriale și matriceale. BLAS este împărțit pe trei niveluri:

- Level 1 BLAS - tratează operațiile scalare, vectoriale și între vectori.
- Level 2 BLAS - tratează operațiile între matrice și vectori.
- Level 3 BLAS - tratează operațiile între matrice și matrice.

Biblioteca BLAS este eficientă, portabilă, disponibilă pe mai multe platforme, fiind destul de utilizată pentru dezvoltarea aplicațiilor de algebră liniară de înaltă performanță.

Pentru analiza accesului corect al memoriei voi folosi tool-ul Memcheck (Memcheck manual fără an) al utilitarului valgrind. În arhiva finală destinată proiectului sunt atașate trei fișiere cu extensia .memory, câte unul pentru fiecare implementare, reprezentând output-urile rulării valgrind (Memcheck).

Pentru analiza accesului eficient al memoriei voi folosi tool-ul Cachegrind (Cachegrind manual fără an) al utilitarului valgrind. În arhiva finală destinată proiectului sunt atașate trei fișiere cu extensia .cache, câte unul pentru fiecare implementare, reprezentând output-urile rulării valgrind (Cachegrind).

Pentru vizualizarea timpilor de execuție specifici fiecărei implementări voi folosi biblioteca Matplotlib pentru limbajul de programare Python. Prin intermediul acesteia, voi realiza un grafic comparativ - pe axa X se va afla dimensiunea matricelor pătratice (N), iar pe axa Y se va afla timpul de execuție.

3. PROIECTARE ȘI IMPLEMENTARE

3.1. Varianta neopt

Varianta **neopt** se bazează pe algoritmul clasic de înmulțire a două matrice.

```
 1. int i, j, k;
 2. double a[N][N], b[N][N], c[N][N];
 3. // Inițializarea matricelor a și b.
 4. for (i = 0; i < N; i++){
 5.     for (j = 0; j < N; j++){
 6.         c[i][j] = 0.0;
 7.         for (k = 0; k < N; k++){
 8.             c[i][j] += a[i][k] * b[k][j];
 9.         }
10.     }
11. }
```

Singura optimizare de care această implementare beneficiază este luarea în considerare a faptului că matricea A este superior triunghiulară. Astfel, a treia buclă for, cea cu counter-ul k, nu va avea întotdeauna N iterații pentru înmulțirile care implică matricea A, ci N - i sau j + 1 iterații, precum în exemplele următoare.

```
1. // Compute A * B * B' (A * BBt). The result is stored in
first_term.
2. for (i = 0; i < N; i++) {
3.     for (j = 0; j < N; j++) {
4.         for (k = N - 1; k >= i; k--) {
5.             first_term[i][j] += *(A + i * N + k) * BBt[k][j];
6.         }
7.     }
8. }
```

```
1. // Compute A' * A (At * A). The result is stored in second_term.
2. for (i = 0; i < N; i++) {
3.     for (j = 0; j < N; j++) {
4.         for (k = 0; k <= j; k++) {
5.             second_term[i][j] += At[i][k] * *(A + k * N + j);
6.         }
7.     }
8. }
```

Fișierul de intrare cu ajutorul căruia au fost realizate testele de performanță are următoarea structură:

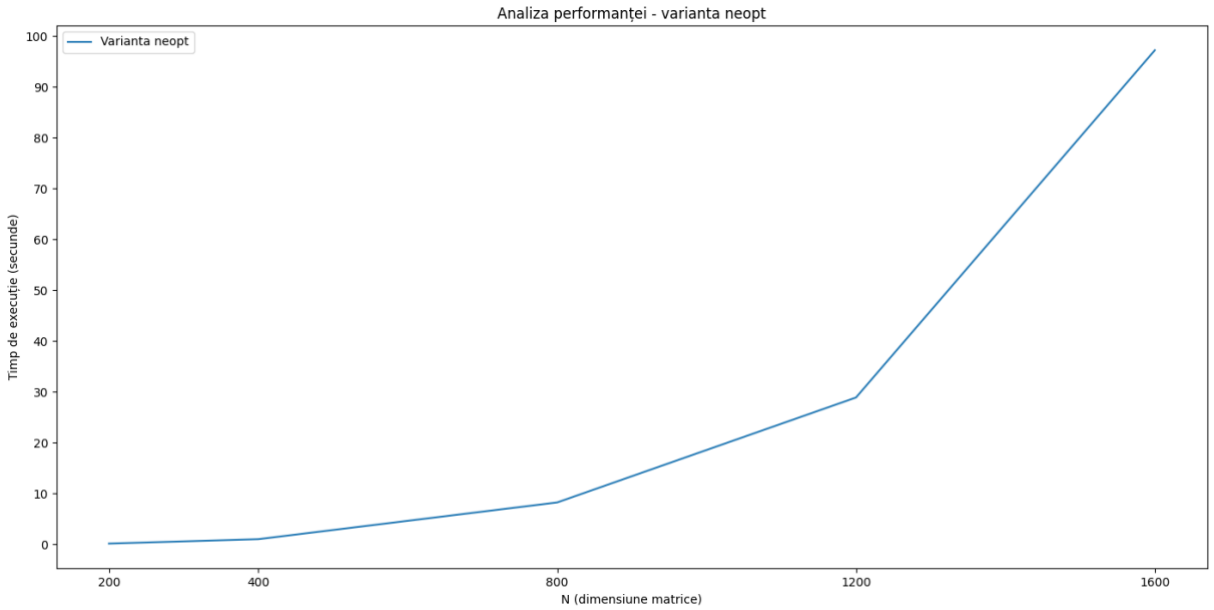
```
1. 5
2. 200 147 out0
3. 400 123 out1
4. 800 456 out2
5. 1200 789 out3
6. 1600 258 out4
```

Mai exact, pentru evaluarea timpilor de execuție au fost rulate cinci teste, cu dimensiuni ale matricelor pătratice cuprinse între 200 și 1600.

Timpii de execuție (exprimați în secunde) obținuți de varianta neopt sunt următorii:

```
NEOPT SOLVER
Run=./neopt: N=200 - Time=0.169884
Run=./neopt: N=400 - Time=1.035024
Run=./neopt: N=800 - Time=8.256198
Run=./neopt: N=1200 - Time=28.900076
Run=./neopt: N=1600 - Time=97.214600
```

Rezultatele obținute, exprimate sub formă grafică:



3.2. Varianta opt

Varianta **opt** reprezintă o implementare care îmbunătățește timpul de execuție și accesul la memorie în mod considerabil.

Prima optimizare constă în a observa faptul că c[i][j] este o constantă în cadrul ciclului interior k. Totuși, pentru un compilator acest lucru nu este neapărat evident, deoarece c[i][j] este o referință în cadrul unui vector bidimensional. Astfel, o primă optimizare arată în felul următor:

```
1. for (i = 0; i < N; i++){
2.     for (j = 0; j < N; j++){
3.         register double suma = 0.0;
4.         for (k = 0; k < N; k++) {
5.             suma += a[i][k] * b[k][j];
6.         }
7.         c[i][j] = suma;
8.     }
9. }
```

În acest mod, compilatorul va avea grijă ca variabila suma să fie ținută într-un registru, permițând astfel o utilizare optimă a acestei resurse. Utilizarea keyword-ului “register” este benefică atunci când sunt identificate constante în cadrul unor bucle.

Un alt aspect care necesită resurse din plin este utilizarea și accesul variabilelor de tip vectorial. De fiecare dată când programul face o referință la un obiect de forma c[i][j], compilatorul trebuie să genereze expresii aritmetice complexe pentru a calcula această adresă din cadrul vectorului bidimensional c. De exemplu, calculul pentru aflarea adresei variabilei c[i][j] este următorul:

$$@(c[i][j]) = @(c[0][0]) + i * N + j$$

Conform acestei formule, fiecare acces presupune două adunări și o înmulțire de numere întregi. Astfel, în momentul în care compilatorul întâlnește instrucțiunea *suma += a[i][k] * b[k][j]* se vor efectua implicit, suplimentar înmulțirii și adunării în virgulă mobilă implicate de cod, patru adunări și două înmulțiri de numere întregi pentru a calcula adresele necesare din vectorii a și b. Se întâmplă destul de frecvent ca procesorul să nu aibă date disponibile pentru a lucra în continuu, din cauza faptului că overhead-ul pentru calculul adreselor este semnificativ. Astfel, un mod de a spori viteza programului este renunțarea la accesele vectoriale prin referință, utilizând în schimb pointeri.

```
1. for (j = 0; j < N; j++)
2.     a[i][j] = 0;          // 2 * N adunări și N înmulțiri
3.
4. // Versus
```

```
 5.
 6. double *ptr = &(a[i][0]); // 2 adunări și o înmulțire
 7. for (j = 0; j < N; j++) {
 8.     *ptr = 0;
 9.     ptr++;                // N adunări de numere întregi
10. }
```

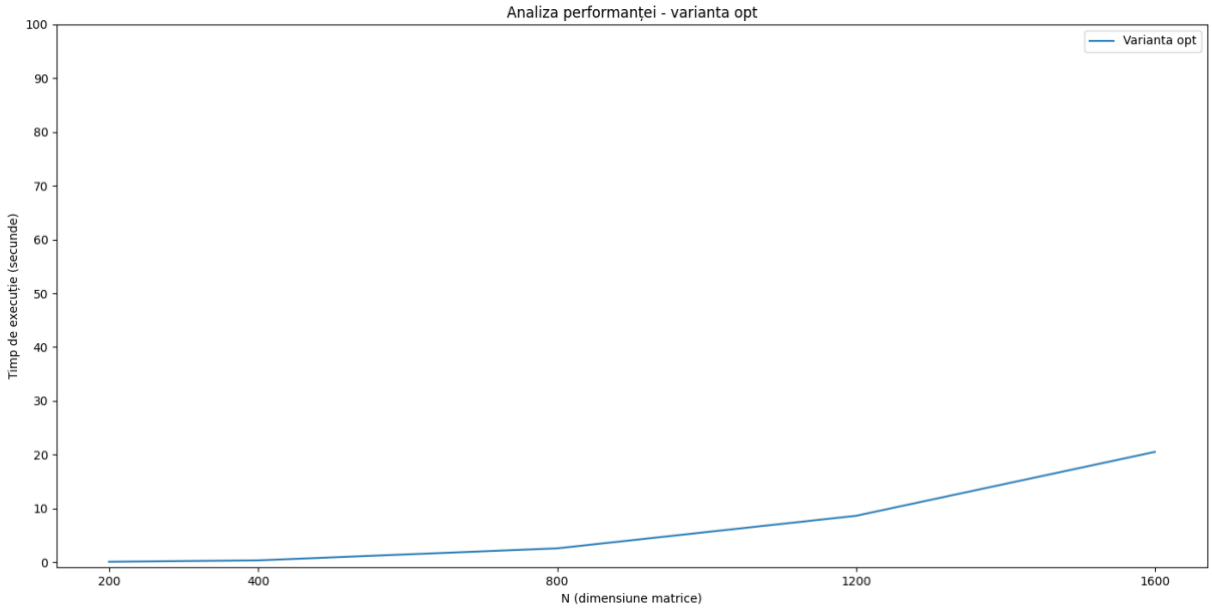
În ultimul rând, pentru înmulțirea matricelor am folosit algoritmul Blocked Matrix Multiplication. Ideea de bază a acestuia constă în refolosirea cât mai bună a elementelor aflate în memoria cache. Astfel, odată cu calcularea lui c[i][j], este indicat să calculăm și c[i][j+1] din moment ce coloana j+1 se află deja în cache. Acest lucru presupune însă reordonarea operațiilor în modul următor: calcularea primilor b termeni pentru c[i][j], calcularea primilor b termeni pentru c[i][j+1], calcularea următorilor b termeni pentru c[i][j], calcularea următorilor b termeni pentru c[i][j+1], etc. Versiunea înmulțirii de matrice utilizând metoda Blocked Matrix Multiplication devine:

```
1. for(bi=0; bi<n; bi+=blockSize)
2.     for(bj=0; bj<n; bj+=blockSize)
3.         for(bk=0; bk<n; bk+=blockSize)
4.             for(i=0; i<blockSize; i++)
5.                 for(j=0; j<blockSize; j++)
6.                     for(k=0; k<blockSize; k++)
7.                         c[bi+i][bj+j] +=
a[bi+i][bk+k]*b[bk+k][bj+j];
```

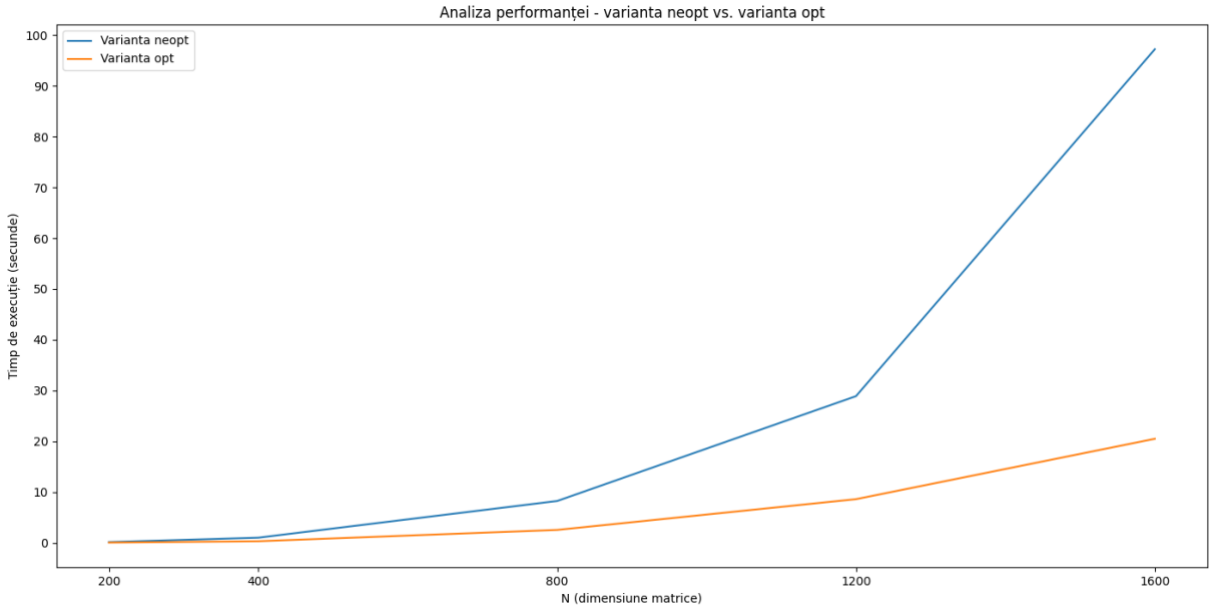
Timpii de execuție (exprimați în secunde) obținuți de varianta opt sunt următorii:

```
OPT SOLVER
Run=./opt: N=200 - Time=0.071030
Run=./opt: N=400 - Time=0.332529
Run=./opt: N=800 - Time=2.562476
Run=./opt: N=1200 - Time=8.617030
Run=./opt: N=1600 - Time=20.503532
```

Rezultatele obținute, exprimate sub formă grafică:



Rezultatele obținute de varianta neopt, comparate cu varianta opt:



3.3. Varianta blas

Biblioteca Basic Linear Algebra Subprograms (BLAS) reprezintă o colecție de funcții matematice de nivel scăzut pentru efectuarea operațiilor de bază cu vectori și matrice. Aceasta oferă o interfață standardizată pentru operațiile comune de algebră liniară, cum ar fi înmulțirea vectorilor și matricilor, produsul scalar și factorizările matriceale.

BLAS a fost dezvoltată la sfârșitul anilor '70 și începutul anilor '80 pentru a furniza implementări eficiente și fiabile ale acestor operații de bază, care sunt fundamentale pentru multe aplicații științifice și inginerești. Scopul BLAS este

de a oferi rutine puternic optimizate care pot beneficia de caracteristicile specifice ale diferitelor arhitecturi hardware, cum ar fi procesoarele vectoriale, procesoarele multicore și sistemele de calcul paralel.

BLAS a devenit un standard adoptat pe scară largă pentru operațiile de bază cu algebră liniară, iar multe pachete și biblioteci software se bazează pe aceasta. Biblioteci de calcul numeric de nivel înalt, cum ar fi LAPACK (Linear Algebra PACKage), se bazează adesea pe BLAS pentru operațiile de nivel inferior.

De-a lungul anilor, implementările BLAS au fost optimizate pentru diferite platforme hardware, iar diferite versiuni ale BLAS, cum ar fi BLAS-1, BLAS-2 și BLAS-3, au fost dezvoltate pentru a se adresa nivelurilor specifice de funcționalitate. În plus, există mai multe biblioteci BLAS optimizate disponibile, cum ar fi Intel MKL, OpenBLAS și ATLAS, care oferă implementări extrem de eficiente pentru diferite arhitecturi hardware.

Prin utilizarea BLAS, dezvoltatorii pot efectua calcule de algebră liniară de performanță ridicată fără a trebui să implementeze operațiile de nivel scăzut în mod individual, permițându-le să se concentreze pe algoritmi și aplicații de nivel superior.

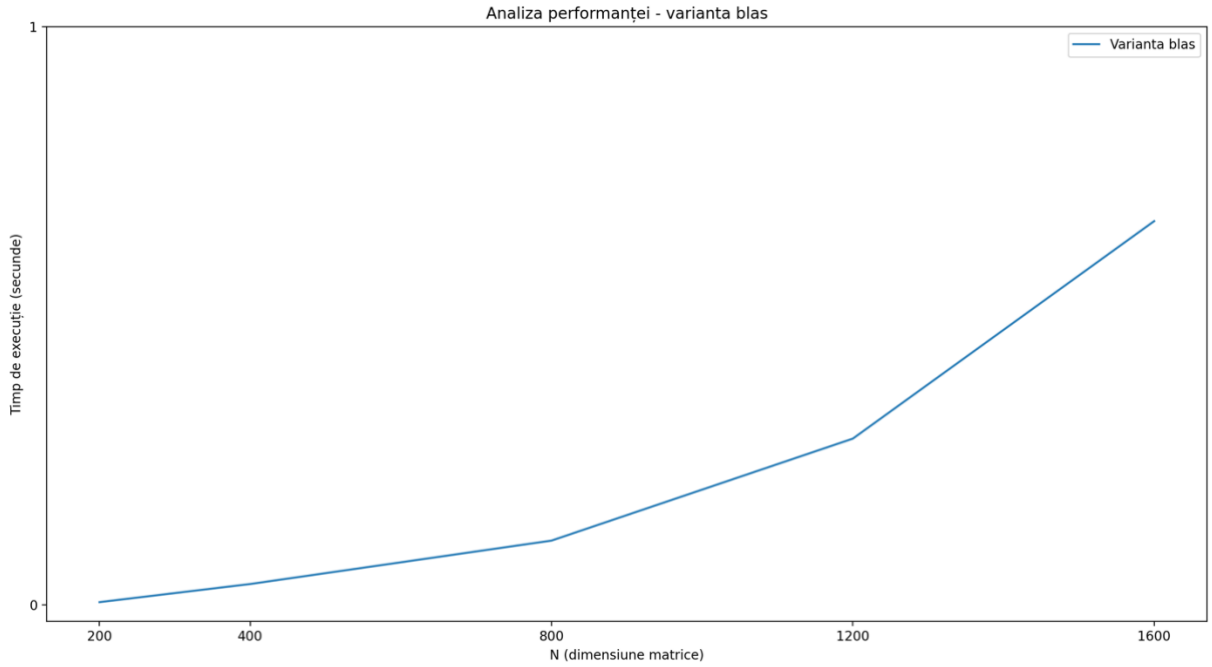
Din cadrul bibliotecii BLAS am folosit următoarele funcții:

- cblas_dcopy - copiază un vector în alt vector;
- cblas_dtrmm - scalează o matrice triunghiulară și o înmulțește cu altă matrice;
- cblas_dgemm - înmulțește două matrice.

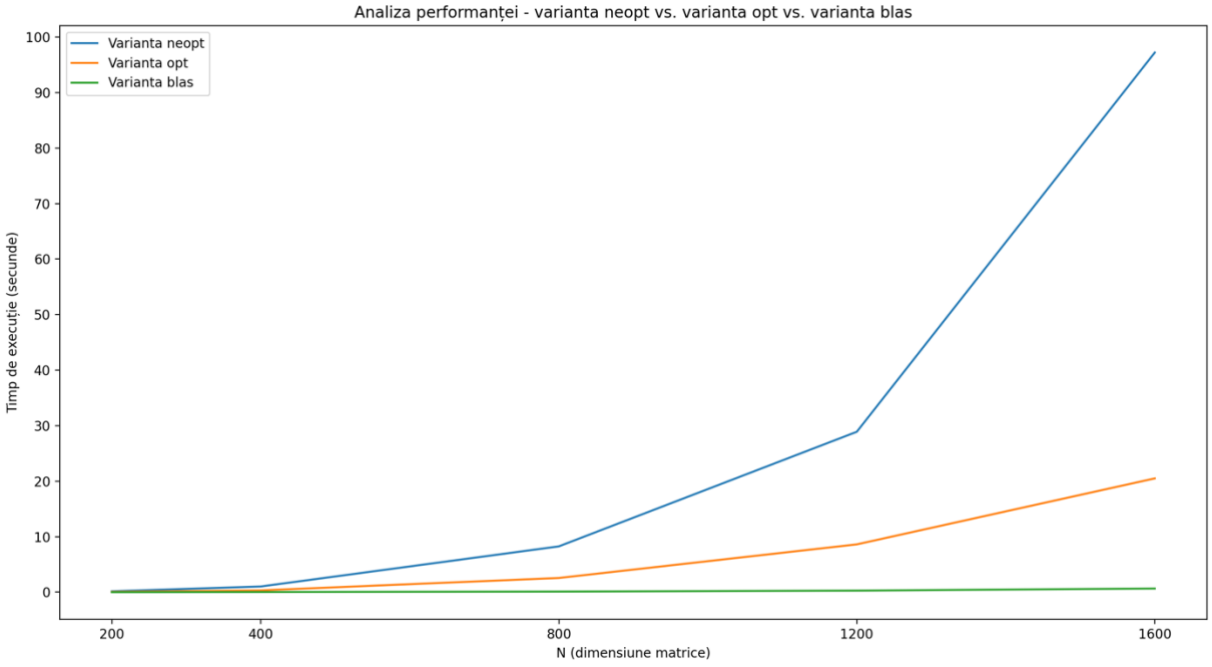
Timpii de execuție (exprimați în secunde) obținuți de varianta blas sunt următorii:

```
1. BLAS SOLVER
2. Run=./blas: N=200: Time=0.004747
3. Run=./blas: N=400: Time=0.036089
4. Run=./blas: N=800: Time=0.111179
5. Run=./blas: N=1200: Time=0.287569
6. Run=./blas: N=1600: Time=0.663836
```

Rezultatele obținute, exprimate sub formă grafică:



Rezultatele obținute de varianta blas, comparate cu varianta neopt și opt:



4. BIBLIOGRAFIE

fără an. *Memcheck manual.* https://valgrind.org/docs/manual/mc-manual.html.

fără an. *BLAS.* https://netlib.org/blas/.

fără an. *Cachegrind manual.* https://valgrind.org/docs/manual/cg-manual.html.