

TDK dolgozat

Böröcz Péter

RITKÁS ADATSZERKEZETEK IMPLEMENTÁCIÓINAK TELJESÍTMÉNYANALÍZISE

Szerző: Böröcz Péter

Szak / évfolyam: Mérnökinformatikus BSc. III. évf.

Témavezetők:
Smidla József, PhD hallgató
Tar Péter, PhD hallgató



Pannon Egyetem, Műszaki Informatikai Kar
Rendszer és Számítástudományi Tanszék
Operációkutatási Kutatólaboratórium

Absztrakt

Napjainkban már megbízható algoritmusok állnak rendelkezésre nagyméretű lineáris programozási problémák megoldására, de futási sebességük még mindig kulcsfontosságú tényező. A megoldók hatékonyságát jelentősen befolyásolják a használt adatszerkezetek. A megfelelő adatszerkezetek implementációja nem triviális, mivel nagy mennyiségű lineáris algebrai művelet kerül elvégzésre a megoldás során, így a rendelkezésre álló adatszerkezetek teljesítményanalízise értékes segítséget nyújthat a hatékonyság növeléséhez.

Munkánk során ritkás adatszerkezetek több nyílt forráskódú implementációját, illetve saját kernelünket hasonlítottuk össze a Blazemark rendszerre épülő új elemzőszoftverünk segítségével. Célunk azon kulcsfontosságú tényezők feltárása, amelyek döntő szerepet játszanak az implementációk különböző műveleteinek teljesítményében.

Dolgozatunkban ismertetjük a ritkás lineáris algebrai adatszerkezeteket és műveleteket, valamint azok tulajdonságait és sajátosságait. Kitérünk továbbá ezeknek hagyományos, sűrű adatszerkezeti változataiktól való különbségére is. Bemutatjuk a Blazemark felépítését, amely lehetővé teszi különböző lineáris algebrai könyvtárak műveleteinek hatékonyság szempontjából való összehasonlítását. Ismertetjük PanOpt lineáris algebrai könyvtárainak integrálását a Blazemark rendszerbe, továbbá a műveletek hatékonyságát mérő tesztek implementálásának folyamatát.

Bemutatjuk saját szoftverünk, a BlazemarkAnalyser felületét és működését, kiemelve az adatszerkezetek elemzését, valamint műveleteinek profilozását szolgáló funkciókra. Ismertetjük, hogy méréseink alapján mik azok a kulcsfontosságú tényezők, amelyek döntő szerepet játszanak az implementációk hatékonyságában, mint például a cache kihasználása, vagy a processzor speciális utasításainak használata. Végül prezentáljuk eredményeinket és az ezekből levont következtetéseinket, amelyeket felhasználtunk saját adatszerkezetek fejlesztésére, kitérve a különböző implementációs tényezők műveleti teljesítményre gyakorolt hatásaira.

Kulcsszavak: Lineáris programozás, lineáris algebra, ritkás adatszerkezetek

Tartalomjegyzék

1. Bevezetés.....	2
2. Irodalmi áttekintés	3
2.1. Lineáris algebrai műveletek	3
2.2. Ritkás adatszerkezetek és műveletek	5
2.3. Lineáris algebrai műveletek numerikus pontossága.....	7
2.4. Nyílt forrású lineáris algebrai könyvtárak.....	8
2.5. A Pannon Optimizer (PanOpt)	9
2.5.1. A PanOpt Vector osztálya	9
2.5.2. A PanOpt Matrix osztálya	10
2.6. A Blazemark keretrendszer	10
2.6.1. A Blazemark felépítése	10
2.6.2. Konfigurálás és paramétereizhetőség.....	11
2.6.3. Az adatszerkezetek inicializálása	11
2.6.4. A teljesítmény mérése	12
3. A Blazemark rendszer kibővítése.....	14
3.1. A Blazemark kiterjesztése új modullal.....	14
3.1.1. A Blazemark osztályainak módosítása.....	14
3.1.2. A Makefile kiegészítése	15
3.2. A Pannon Optimizer tesztjeinek integrálása	15
3.2.1. A mérési osztályok	15
3.2.2. A PanOpt könyvtármodul által támogatott tesztek	17
4. A BlazemarkAnalyser elemző szoftver	19
4.1. A BlazemarkAnalyser adatbázisa.....	19
4.2. A BlazemarkAnalyser működése	22
4.2.1. Adatbázis-kezelés.....	22
4.2.2. A Blazemark keretrendszer kezelése.....	24
4.3. A BlazemarkAnalyser kezelőfelülete.....	26
4.3.1. A Run fül.....	26

4.3.2. A Solvers fül.....	27
4.3.3. Az Analyse fül.....	28
5. Mérési eredmények.....	31
5.1. Tesztelési körülmények.....	31
5.2. Teszteredmények és következtetések.....	31
5.3. A teljesítményt befolyásoló paraméterek.....	34
5.4. A PanOpt új adatszerkezetei	38
5.4.1. A PanOpt DenseVector osztálya	38
5.4.2. A PanOpt IndexedDenseVector osztálya	40
5.4.3. A PanOpt SparseVector osztálya	42
5.5. Fejlett implementációs tényezők.....	43
5.5.1. Blokkhatárra igazítás.....	43
5.5.2. Loop unrolling.....	45
5.5.3. A processzor utasításkészletének hatékony kihasználása	46
6. Összefoglalás	48
Irodalomjegyzék	49
Köszönetnyilvánítás.....	50

1. Bevezetés

Nagy számításigényű problémák megoldása során fontos szempont a megoldó algoritmus gyorsasága. Ez optimalizálási feladatok megoldása esetén kiemelkedő fontosságú. A lineáris optimalizálási feladatok matematikai modellje vektorokból épül fel, annak megoldásait lineáris terekben keressük, így a feladat megoldása során nagy számban kerülnek kiszámításra különféle lineáris algebrai műveletek. Napjainkban egy ilyen modell akár egy többszázézerszer többszázezres mátrixot is tartalmazhat, aminek már a memóriában való eltárolása is problémákba ütközik hagyományos tárolási módszerekkel. Ennek értelmében mind a feladat tárolása, mind pedig a megoldása során szükség van a modell egyéni jellemzőinek kihasználására. Egy ilyen jellemző tulajdonság a ritkasság, ritkás feladatok esetén az együtthatók között a nem-nulla értékek aránya nagyon alacsony (általában 1% alatti). Ennek kihasználása elengedhetetlen a megoldó algoritmusok szempontjából. A ritkasságot kihasználó adatszerkezetek és az általuk elvégzett műveletek implementációja azonban nem triviális, komoly tervezést igénylő feladat.

Napjainkban számos nyílt forráskódú lineáris algebrai könyvtár létezik, amely alkalmas ezen feladatok ellátására, azonban mérhető különbség van műveleteik hatékonysága között. A különböző ritkás adatszerkezeti implementációk egyes jellemzői és az általuk elvégzett lineáris algebrai műveletek egy csoportja közötti összefüggés meglete nyilvánvaló. Munkánk során ezen implementációkat vizsgáltuk meg és hasonlítottuk össze annak érdekében, hogy feltárjuk a teljesítményeik különbségében szerepet játszó kulcsfontosságú tényezőket, és mindezt felhasználhassuk kutatócsoportunk saját fejlesztésű lineáris programozási megoldó szoftverének, a Pannon Optimizer (PanOpt) adatszerkezeteinek implementációjában.

Dolgozatunkban először ismertetjük a ritkás adatszerkezeteket és műveleteket, kitérve ezeknek a hagyományos, sűrű adatszerkezeti változataiktól való különbségére is. Bemutatjuk az általunk vizsgált nyílt forrású lineáris algebrai könyvtárakat, valamint a PanOpt lineáris algebrai kernelét, illetve ezek jellemzőit. Ismertetjük a Blazemark tesztelő keretrendszer felépítését, amely segítségével lehetséges különböző lineáris algebrai könyvtárak műveleteinek összehasonlítása azok hatékonysága alapján. Bemutatjuk, hogyan terjesztettük ki a Blazemark rendszert egy új, a PanOpt lineáris algebrai könyvtárait magába foglaló modullal, ismertetjük az ehhez tartozó tesztek integrálásának menetét, illetve a Blazemark rendszerre épülő, saját fejlesztésű elemző szoftverünk, a BlazemarkAnalyser működését. Prezentáljuk tesztjeink eredményeit és az ezekből levont következtetéseinket, illetve az ezen eredmények alapján fejlesztett új adatszerkezeteinket. Végül bemutatjuk a vizsgált implementációs tényezők, köztük az architektúra-specifikus tulajdonságok kihasználásának hatását a különböző műveletek hatékonyságára.

2. Irodalmi áttekintés

Ezen fejezetben a korábban publikált eredményekre és a téma szakirodalmára hivatkozva bemutatjuk a munkánk során vizsgált lineáris algebrai műveleteket és adatszerkezeteket, kitérve ezek specifikációira és fontosságukra. Áttekintést nyújtunk a sűrű és ritkás adatszerkezetek és műveleteik közötti különbségekről, illetve az implementálás során figyelembe veendő tényezőről.

Bemutatjuk továbbá a tesztjeink során használt és vizsgált nyílt forráskódú lineáris algebrai könyvtárakat, az általuk támogatott műveleteket, kitérve a ritkás lineáris algebrai adatstruktúrákra kiterjedő támogatásukra. Külön alfejezetben tárgyaljuk kutatócsoportunk saját fejlesztésű lineáris programozási megoldó szoftverének, a Pannon Optimizer-nek lineáris algebrai könyvtárát, bemutatva az ezekben implementált adatszerkezeteket, illetve műveleteik működését.

A fejezet végén a Blazemark keretrendszer működéséről és tesztjeinek menetéről nyújtunk áttekintést, kitérve a Blazemark könyvtárak segítségével végzett mérések inicializálására és pontosságára.

2.1. *Lineáris algebrai műveletek*

Ebben az alfejezetben az általunk vizsgált és gyakran használt lineáris algebrai műveleteket mutatjuk be [1]. Ezen műveletek olyan skalár-, vektor-, illetve mátrixműveletek, amelyek implementálása során fontos szempont a művelet végrehajtásának gyorsasága, mivel az azokat felhasználó algoritmusok számára ez kritikus lehet. Minden vektor leírható egy rendezett n -esként, ahol n az adott vektor feletti vektortér dimenziója.

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \quad (1)$$

Egy mátrix, melynek m sora és n oszlópa van, az alábbi módon írható le:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \quad (2)$$

Az általunk vizsgált műveletek esetében $\lambda \in \mathcal{R}$; $\mathbf{u}, \mathbf{v} \in \mathcal{R}^n$; $\mathbf{A} \in \mathcal{R}^n \times \mathcal{R}^n$. A továbbiakban a vektorműveletek leírásánál feltételezzük, hogy $\dim(\mathbf{u}) = \dim(\mathbf{v})$. Az első alapvető művelet két vektor összeadása:

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} u_1 + v_1 \\ u_2 + v_2 \\ \vdots \\ u_n + v_n \end{bmatrix} \quad (3)$$

Ezen művelet gyors végrehajtása azért is fontos, mert több másik műveletet (vektorok különbsége, lineáris kombinációja, mátrixműveletek) is a vektorok összeadására visszavezetve oldunk meg. A második művelet egy vektor skalárral való szorzása:

$$\lambda \mathbf{v} = \lambda \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \quad (4)$$

Ezt a műveletet vektorok különbségének és lineáris kombinációinak számításakor is alkalmazzuk, ezért kiemelten fontos a művelet hatékony implementálása. A harmadik alpművelet a vektorok skaláris szorzása. Ezen művelet elvégzése során két vektor szorzataként egy skalárt kapunk eredményül:

$$\mathbf{u}^T \mathbf{v} = \sum_{i=1}^n u_i v_i \quad (5)$$

Ezt a műveletet használjuk fel a mátrix-vektor, illetve a mátrix-mátrix szorzások megoldása során. Vektorok különbségét a -1 értékű skalárral való szorzás és a vektorösszeadás alkalmazásával számíthatunk:

$$\mathbf{u} - \mathbf{v} = \mathbf{u} + (-1)\mathbf{v} \quad (6)$$

Vektorok lineáris kombinációját szintén a skalárral való szorzás és a vektorösszeadás műveletekre visszavezetve számolhatjuk. Legyen $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k$ azonos dimenziójú vektorok halmaza, és $\lambda_1, \lambda_2, \dots, \lambda_k$ skalárok halmaza, ekkor $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k$ vektorok lineáris kombinációja:

$$\lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2 + \dots + \lambda_i \mathbf{v}_i = \sum_{i=1}^k \lambda_i \mathbf{v}_i \quad (7)$$

A transzponált vektor-mátrix szorzás (amely akkor értelmezhető, ha a mátrix sorainak száma egyenlő a vektor dimenziójával), illetve a mátrix-vektor szorzás (amely akkor értelmezhető, ha a mátrix oszlopainak száma megegyezik a vektor dimenziójával) eredményei vektorok, amelyek elemeit skaláris szorzással számoljuk ki:

$$\mathbf{v}^T \mathbf{A} = [v_1 \quad \dots \quad v_m] \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} = [\sum_{i=1}^m v_i a_{i1} \quad \dots \quad \sum_{i=1}^m v_i a_{in}] \quad (8)$$

$$\mathbf{A} \mathbf{v} = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^n a_{1i} v_i \\ \sum_{i=1}^n a_{2i} v_i \\ \vdots \\ \sum_{i=1}^n a_{mi} v_i \end{bmatrix} \quad (9)$$

A mátrix-mátrix szorzást szintén skaláris szorzás műveletek segítségével hajthatjuk végre. Itt $\mathbf{A}$ mátrix oszlopainak száma megegyezik $\mathbf{B}$ mátrix sorainak számával:

$$\mathbf{AB} = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & \dots & b_{1o} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{no} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^n a_{1i} b_{i1} & \dots & \sum_{i=1}^n a_{1i} b_{io} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^n a_{mi} b_{i1} & \dots & \sum_{i=1}^n a_{mi} b_{io} \end{bmatrix} \quad (10)$$

A továbbiakban ezen műveletek implementációinak vizsgálatával foglalkozunk.

2.2. Ritkás adatszerkezetek és műveletek

Vektorok és mátrixok esetében gyakran előfordul, hogy az adatszerkezetben tárolt elemek nagy része nulla, ezt a tulajdonságot nevezzük ritkasságnak. Kutatási területünkön, a lineáris programozás (LP) által használt modellek mátrixai esetében ez egy jellemző tulajdonság, amelynek kihasználása elengedhetetlen nagyméretű feladatok megoldásához [2]. Az matematikai modell megoldása során, ezt a lehető legnagyobb mértékben ki kell használni, mivel a problémák méretéből adódóan már a modellek memóriában való tárolása is problémát jelenthet, az optimális megoldás megtalálása pedig rengeteg számítást igényel. Ezen alfejezetben bemutatjuk a sűrű és ritkás adatszerkezetek közti különbségeket, illetve a ritkás adatszerkezetek műveleteinek jellemzőit és speciális eseteit.

Egy hagyományos (sűrű) vektor implementáció a vektor összes elemét lineárisan, egy, legalább a vektor dimenziójának megfelelő nagyságú tömbben tárolja. Ennek a tárolási módnak az előnyei közé tartozik az elemekhez való direkt hozzáférés (a tömb elemeinek indexei megegyeznek a vektor elemeinek indexeivel), a vektorműveletek megvalósítása egyszerű, illetve kihasználhatjuk, hogy a vektor mérete előre ismert. Ezzel szemben egy ritkás vektor implementációban a vektornak csak a nem-nulla elemei kerülnek eltárolásra $(i, v_i), i \in \bar{Z}$ párként, ahol $\bar{Z}$ a $\mathbf{v}$ vektorban szereplő nem-nulla elemek indexeinek halmaza. Legyen m a vektor dimenziója, és z a vektor nem-nulla elemeinek száma. A vektor implementációk adattömbjében sűrű vektorok esetében összesen m , ritkás vektorok esetében pedig $2z$ (mely mennyiségnek a fele egész, fele pedig lebegőpontos szám) elem kerül eltárolásra. Nagy lineáris programozási feladatok modelljei esetében átlagosan az elemek 0.01-1%-a nem-nulla,

könnyen látható, hogy egy LP modell esetében a létrehozott adatszerkezet memóriaigénye akár 0.02% alá csökkenhet ritkás adatszerkezet használata esetén [2].

Egy ritkás vektor esetében vizsgálhatjuk a nem-nulla indexek rendezettségét. Ha egy vektorunk rendezett és meg szeretnénk tartani az elemek index szerinti rendezettségét, akkor minden esetben, amikor megváltozik a vektor, szükséges rendezni is az elemeket. Mivel ez több erőforrást emésztene fel, mint maguk a vektorral végzett műveletek, nem vesszük alapértelmezettnek a ritkás vektorok rendezettségét, arra csak ritkán van szükség.

A vektorműveletek implementációja is függ a műveletben szereplő vagy vektorok típusaitól, az egyes műveleteket mindkét vektortípusra külön kell implementálni. A vektorösszeadás implementációjában például külön esetként kell kezelni két sűrű, két ritkás, illetve egy sűrű és egy ritkás vektor összeadását. Két sűrű vektor esetében egy ciklus segítségével a vektorok összes elemének összeadására van szükség, míg egy ritkás és egy sűrű vektor esetében elegendő a ritkás vektor nem-nulla elemeit hozzáadni a sűrű vektor megfelelő indexű elemeihez. Két ritkás vektor esetében pedig célszerű létrehozni egy sűrű segédvektort, majd a ritkás vektorok nem-nulla elemeit hozzáadni a segédvektorhoz. Látható tehát, hogy ezek az eljárások alapvetően különbözőek, az összeadás mellett a többi művelet esetén is hasonló szempontokat kell figyelembe vennünk.

Egy ritkás mátrix implementációja általában ritkás oszlop- vagy sorvektorokból, vagy a nem-nulla elemek és a hozzájuk tartozó koordináták halmazából áll. Az utóbbi módszer előnye, hogy nincs megszabva az elemek sorrendje, így a mátrix feltöltése egyszerű és gyors folyamat, azonban hátrányai közé tartozik, hogy a mátrixműveletek implementálása nehézkes, és a hatékony műveletvégzés érdekében az így tárolt mátrixot egy másik tárolási módszert használó adatszerkezetre kell átkonvertálni. Ha a mátrixot sorvektorainak halmazaként tároljuk, akkor sorfolytonos, ha oszlopvektorainak halmazaként, akkor oszlopfolytonos tárolásról beszélünk. Ezen tárolási módszerek esetében a sorok (oszlopok) nem-nulla elemei folytonosan tárolhatóak egyazon tömbben (az elemek és a hozzájuk tartozó indexek külön-külön), azonban tárolni kell minden vektor első elemének pozícióját. Ha nem biztos, hogy a sorokat (oszlopokat) index szerint rendezve tároljuk, akkor minden vektor esetében célszerű tárolni a vektorban lévő nem-nulla elemek számát is. Ezzel a tárolási móddal kapcsolatos probléma a vektorokban lévő nem-nulla elemek számának megváltozása, azaz az adatszerkezet dinamikus használata. Ez jelentősen felgyorsítható a mátrixban tárolt vektorok után hagyott üres memóriapozíciókkal, amelyek helyére beszűrhetők az esetlegesen hozzáadott új nem-nulla elemek. A mátrixokkal végzett műveletek esetében az implementációnak tudnia kell külön kezelni a ritkás és sűrű mátrixokat, sorfolytonos, illetve oszlopfolytonos tárolás esetén is [2].

2.3. *Lineáris algebrai műveletek numerikus pontossága*

A legtöbb LP megoldó szoftver, köztük a PanOpt is dupla pontosságú lebegőpontos számokkal dolgozik, amelyek képesek nagyon nagy és nagyon kicsi racionális számok reprezentálására. Mivel ezen változók véges memóriaterületen kerülnek eltárolásra, ezért az ábrázolható számok száma a valós számok halmáznak számosságával ellentétben véges. Ha egy művelet elvégzése során eredményül kapott szám nem ábrázolható a lebegőpontos aritmetikával, az adott szám a hozzá legközelebb eső ábrázolható értékre kerekítődik. Több művelet elvégzése után ezek az egyébként elhanyagolható kerekítési hibák összeadódása jelentős számítási hibát eredményezhet. Ha $x > 0$ és $y > 0$ lebegőpontos számok abszolút hibája Δ_x és Δ_y , relatív hibájuk pedig δ_x és δ_y , a velük elvégzett műveletek hibái:

Összeadás elvégzése esetén:

$$\Delta_{x+y} = \Delta_x + \Delta_y \quad (11)$$

$$\delta_{x+y} = \max\{\delta_x, \delta_y\} \quad (12)$$

Kivonás elvégzése esetén:

$$\Delta_{x-y} = \Delta_x + \Delta_y \quad (13)$$

$$\delta_{x-y} = \frac{|x|}{|x-y|} \delta_x + \frac{|y|}{|x-y|} \delta_y \quad (14)$$

Szorzás elvégzése esetén:

$$\Delta_{x \cdot y} = |y| \cdot \Delta_x + |x| \cdot \Delta_y \quad (15)$$

$$\delta_{x \cdot y} = \delta_x + \delta_y \quad (16)$$

Osztás elvégzése esetén:

$$\Delta_{\frac{x}{y}} = \frac{|y| \cdot \Delta_x + |x| \cdot \Delta_y}{y^2} \quad (17)$$

$$\delta_{\frac{x}{y}} = \delta_x + \delta_y \quad (18)$$

Látható, hogy a kivonás művelet során keletkezhet a legnagyobb relatív numerikus hiba, azon esetekben, amikor $|x - y|$ érték sokkal kisebb $|x|$ és $|y|$ értékeknél. A szimplex módszert implementációinak esetében a lebegőpontos számábrázolásból adódó számítási hibák komoly problémát jelenthetnek, így azok kezelése elengedhetetlen [3].

Erre a problémára részben megoldást jelenthet a megoldás során a megengedettségi és az optimalitási feltételek toleranciás vizsgálatának megvalósítása, azonban a vektorok feltöltődhetnek számítási hibákból adódó értékekkel, amelyek jelentős teljesítménycsökkenést eredményezhetnek. Ez bizonyos mértékben kezelhető az összeadás toleranciás

megvalósításával, amely egy abszolút és egy relatív toleranciát használva nullára állítja az elhanyagolhatóan kicsi értékű eredményeket:

$$x + y = \begin{cases} 0, & \text{ha } |x + y| < \varepsilon_r \max\{|x|, |y|\}, \\ 0, & \text{ha } |x + y| < \varepsilon_a \\ x + y & \text{egyébként} \end{cases} \quad (19)$$

2.4. Nyílt forrású lineáris algebrai könyvtárak

A lineáris algebrai műveleteket megvalósító implementációk (Basic Linear Algebra Subprograms – BLAS) funkcionalitása 3 szintre osztható: az első szintű műveletek közé a különböző vektor-vektor, illetve a vektor skalár műveletek, a második szintű műveletek a vektor-mátrix, a harmadik szintűek pedig a mátrix-mátrix műveleteket foglalják magukba. Ezen műveletek hatékony elvégzésére számos nyílt forráskódú könyvtár használható, amelyek implementációi azonban lényegesen eltérőek. Ezen alfejezetben bemutatjuk az általunk vizsgált nyílt forráskódú lineáris algebrai könyvtárakat, illetve fő jellemzőiket.

Boost uBLAS

Az uBLAS egy általános C++ lineáris algebrai adatszerkezet implementáció, amely ritkás és sűrű adatszerkezetekkel is képes dolgozni, ellátva az 1-es, 2-es és 3-as szintű BLAS funkcionalitást. Az uBLAS könyvtár a szintén nyílt forrású Boost C++ könyvtárcsomag része, eredeti szerzői Joerg Walter és Mathias Koch [4].

Blaze

A Blaze egy nyílt forrású, nagy teljesítményű matematikai C++ könyvtár, sűrű és ritkás adatszerkezeti műveletekre. Külön kezel sor- és oszlopvektorokat, illetve sor- és oszlopfolytonos mátrixokat. Számításainak felgyorsítása érdekében kihasználja a modern számítógépek párhuzamosított műveletekre való alkalmasságát. Fejlesztői Klaus Iglberger, Georg Hager, Christian Godenschwager, és Tobian Scharpf [5].

Blitz++

A Blitz++ egy C++ könyvtár teljesítményoptimalizált vektor- és mátrixműveletekre. Külön adatszerkezetet használ sorfolytonos és oszlopfolytonos mátrixokra, és csak sűrű adatszerkezetekkel dolgozik. A projekt alapítója és fejlesztője Todd Veldhuizen [6].

GMM++

A GMM++ sűrű és ritkás vektorokat, illetve mátrixokat és az ezekkel végzett műveleteket tartalmazó implementáció. Rendelkezik külön olvasásra és írásra optimalizált vektor adatszerkezettel, szintén külön kezel sorfolytonos és oszlopfolytonos mátrixokat. A könyvtár fejlesztői Yves Renard és Julien Pommier [7].

Eigen

Az Eigen egy C++ header könyvtár lineáris algebrai, vektor és mátrix műveletekhez, numerikus megoldókhöz, illetve ezekhez kapcsolódó algoritmusokhoz. A vektorokat 1 soros, illetve 1 oszlopos mátrixként kezeli, tartalmaz sűrű és ritkás adatszerkezeteket is. Fejlesztői Benoît Jacob és Gaël Guennebaud [8].

MTL

A Matrix Template Library (MTL) egy lineáris algebrai könyvtár C++ programnyelven, ritkás és sűrű adatszerkezeti implementációt tartalmazva. Az MTL-t fejlesztő cég a SimuNova [9].

2.5. A Pannon Optimizer (PanOpt)

Ezen alfejezetben kutatócsoportunk saját fejlesztésű lineáris programozási megoldó szoftverének, a Pannon Optimizer-nek lineáris algebrai könyvtárát és annak sajátosságait mutatjuk be.

2.5.1. A PanOpt Vector osztálya

A PanOpt-ban implementált vektor adatszerkezet egyedülálló módon képes sűrű és ritkás módon is tárolni egy vektort, ritkásságának mértéke alapján váltva a tárolási módok között. A vektor adattömbjének létrehozásakor néhány elemmel nagyobb tömb kerül lefoglalásra a memóriában, mint ami a tároláshoz szükséges lenne. Ennek köszönhetően felgyorsulnak a vektorok dimenziójának, vagy a nem-nulla elemek számának változását eredményező műveletek, mivel a tömb által tartalmazott elemek számának növekedése esetén nincs szükség a memória folyamatos újrafoglalására. Az adatszerkezet tárolja, hogy az adott vektor elemei index szerinti sorrendben szerepelnek-e az adattömbben, illetve a nem-nulla elemeknek azt az arányszámát, amely fölött a vektort sűrű vektorként reprezentálja. Sűrű vektor esetében a tömb, amely az adattömbben lévő elemek indexeit tárolja, nincs használatban.

Műveletei úgy vannak implementálva, hogy minél gyorsabban el tudják végezni a PanOpt által megvalósított simplex algoritmus számításait. Az egyik ilyen gyakori művelet a transzformációk során gyakran alkalmazott $\mathbf{u} := \mathbf{u} - \lambda \mathbf{v}$, így ez alaplóműveletként szerepel az osztály függvényei között. Speciális esete az egyszerű vektorösszeadás: $(\mathbf{u} + \mathbf{v} = \mathbf{u} - (-1)\mathbf{v})$. Szintén alaplóműveletként szerepelnek a skaláris szorzás, a skalárral való szorzás. Támogatja továbbá az algoritmus által gyakran használt elemi Forward Transformation (FTRAN) műveletet, ami a vektor balról való megszorzása egy elemi transzformációs mátrixszal, ami egy, az egységmátrixtól pontosan egy oszlopban különböző speciális mátrix:

$$u := Eu, \quad E = \begin{bmatrix} 1 & & a_{1i} & & \\ & \ddots & \vdots & & \\ & & a_{ii} & & \\ 0 & & \vdots & \ddots & \\ & & a_{ni} & & 1 \end{bmatrix} \quad (20)$$

2.5.2. A PanOpt Matrix osztálya

A PanOpt-ban implementált mátrix adatszerkezet sajátosságai közé tartozik a mátrix redundáns tárolása, a sor- és oszlopfolytonos tárolás egyaránt felépítésre kerül. Bár a mátrixok létrehozása és elemeinek módosítása számításigényesebb, de bizonyos intenzíven használt mátrixműveletek jelentősen gyorsabbak. Ez egy LP megoldó esetében azért hatékony, mivel az együtthatómátrix módosításra a megoldás során általában nincs szükség, azonban sorainak és oszlopainak lekérésére gyakran, amelyek bármelyike így közvetlenül elérhető.

Az adatszerkezet képes az elemek soronkénti, illetve oszloponkénti skálázására, amelyre a numerikus számítások pontosságának javítása miatt van szükség. Ez igen fontos nagyméretű LP-k megoldása esetén, valamint az LP előfeldolgozó által elvégzendő reduktív műveletek is támogatással rendelkeznek. A PanOpt Matrix osztályának jelenlegi verziójában számos műveletgyorsítási fejlesztés még nincs implementálva, így a mátrix-mátrix műveletek, és a mátrix-vektor műveletek nagy része nagyságrendileg nem összehasonlítható a műveletek sajátosságait már kihasználó könyvtárakéval.

2.6. A Blazemark keretrendszer

A Blazemark a Blaze lineáris algebrai tesztelő keretrendszere, amely különböző könyvtárak sűrű és ritkás vektor- illetve mátrixművelet implementációinak teljesítménymérésére alkalmas. Ezen alfejezetben a Blazemark keretrendszer működését és teljesítmény-analizálási lehetőségeit mutatjuk be.

2.6.1. A Blazemark felépítése

A Blazemark keretrendszer tesztjeinek alapját egy minden teszthez külön megírt tesztelő főosztály, illetve a tesztelt könyvtárakhoz tartozó modulok képezik, ezek tartalmazzák az egyes könyvtárak támogatott tesztjeinek futtatásához szükséges függvényhívásokat. A főosztály feladatai közé tartozik a tesztek paraméterezése, a teszt során elvégzendő lebegőpontos műveletek számának kiszámítása, a jobb mérhetőség érdekében a szükséges tesztismétlések kiszámítása, illetve a teszt megfelelő ismétlésszámban való futtatása minden olyan könyvtármodullal, ami támogatja az adott tesztet.

A könyvtármodulok egy inicializáló osztályból és a könyvtár által implementált tesztműveletek mérési osztályaiból állnak. Az inicializáló osztály a könyvtár adatszerkezeteinek (ritkás és sűrű vektorok, illetve mátrixok) paraméterezhető inicializáló

függvényeit, egy mérési osztály pedig a tesztelés során végrehajtott egy műveletet és az annak mérésére szolgáló függvényt foglalja magában.

2.6.2. Konfigurálás és paraméterezhetőség

A Blazemark, a Blaze könyvtár részeként nyílt forráskódú, C++ nyelven íródott rendszer, konfigurálása egy teszt és egy fordítási konfigurációs fájl segítségével történik:

- A teszt konfigurációs fájlban a tesztekre vonatkozó beállítások szerepelnek. Ide tartozik a tesztek kívánt és maximális ideje, az adatszerkezetek feltöltése során használt lebegőpontos változók pontossága, a tesztek végrehajtási száma, illetve a véletlenszám-generátor által használt inicializáló értékek. Megadhatjuk továbbá, hogy ritkás mátrixok esetén azok milyen struktúrában kerüljenek feltöltésre (véletlenszerű, vagy csak a főátlóval párhuzamos sávban feltöltött).
- A fordítási konfigurációs fájlban a használni kívánt fordítóprogram és a hozzá tartozó fordítási paraméterek, illetve a tesztelés során használt könyvtárak elérési útvonala adható meg.
- A fordításhoz szükséges Makefile-t egy konfigurációs script hozza létre a fordítási konfigurációs fájl segítségével. Ezen Makefile segítségével lehetőség van az egész keretrendszer egyben való lefordítására, illetve arra is, hogy a tesztekétkülön-külön fordítsuk le. (Erre a teljes rendszer hosszú fordítási ideje miatt lehet szükség.)

A Blazemark keretrendszer lehetőséget nyújt minden teszt paraméterezésére, amelyet egy tesztenként különböző paraméterfájl segítségével lehet megtenni. Mivel a teszt futtatása során az adott teszt főosztálya dolgozza fel a paraméterfájlban tárolt adatokat, ezek módosítása esetén nincs szükség a teszt újrafordítására.

A paraméterfájlokban megadhatjuk a tesztelés során használt adatszerkezetek méretét, illetve (ritkás adatszerkezetek esetében) az adatszerkezetben lévő nem-nulla elemek számát, lehetőség van továbbá a tesztismétlések számának felülírására is, amennyiben nem szeretnénk, hogy ez a rendszer által automatikusan meghatározott érték legyen.

2.6.3. Az adatszerkezetek inicializálása

Az adatszerkezetek paraméterezés szerinti inicializálása a könyvtáranként eltérő adatszerkezetek miatt minden könyvtármodul saját inicializáló osztálya segítségével történik. Ez tartalmazza az összes felhasznált adatszerkezet (tesztjeink során a sűrű vektor, a ritkás vektor, a sűrű mátrix, illetve a ritkás mátrix) inicializálásához szükséges függvényeket.

Az adatszerkezetek feltöltése 0 és 10 közötti (a 0 és a 10 már nem megengedett) véletlen lebegőpontos számokkal történik a Blaze véletlenszám-generátorának segítségével. Ritkás adatszerkezetek esetében ügyelni kell arra is, hogy a tesztelt adatszerkezetekben szereplő nem-nulla elemek minden teszt eset során azonos helyen szerepeljenek, mivel így biztosított a

művelet elvégzése után eredményül kapott vektorok vagy mátrixok azonos ritkássága, így a különböző könyvtárak eredményeinek összehasonlíthatósága is. Ezt a feladatot a Blazemark *Indices* segédosztálya látja el, amely a tesztek megkezdése előtt véletlenszerűen (ritkás mátrix esetében a konfigurációban megadott struktúra szerint) generálja az adatszerkezetekben feltölteni kívánt indexek listáját, így ugyanaz a lista kerül felhasználásra az összes inicializálás folyamán.

2.6.4. A teljesítmény mérése

A Blazemark az eredmény megbízhatóságának növelése érdekében minden tesztesetet többször hajt végre, ezt az iterációszámot úgy határozza meg, hogy a teszt a Blaze könyvtár esetében a másodperces nagyságrendet érje el, így (amennyiben a tesztelt könyvtárak eredményei nem térnek el több mint 2-3 nagyságrenddel a Blaze könyvtár eredményeitől) pontos eredményeket és elfogadható futási időket kaphatunk.

A tesztesetek iterálása és az időeredmény mérése nem tartalmazza az inicializálást, a mérés csak magára a tesztelt műveletre terjed ki. A mérés a *Blaze Timer* osztályának segítségével történik, amely a processzor specifikációinak és a platformnak a függvényében akár *ns* pontosságú eredményt is adhat.

A teszt teljesítményének meghatározásához a teszt főosztálya előre kiszámítja az elvégzendő lebegőpontos műveletszámot. A mért futási időeredmény alapján a program kiszámítja az adott könyvtár művelet-implementációja által másodpercenként elvégzett millió lebegőpontos műveletek számát (MFlop/s). Ez a mérőszám szolgál majd a könyvtárak összehasonlítására. A Blazemark rendszer feltételezi a könyvtárak által implementált műveletek szemantikai hibátlanságát, így csak korlátozott visszajelzést ad az eredmények helyességéről, csak az eredményül kapott adatszerkezetek méretét ellenőrzi, azok tartalmát nem.

Mivel a tesztet a rendszer a konfiguráció alapján többször is megismétli, jelezni tudja a mérési eredmény pontatlanságát, ha bármelyik ismétlés eredménye több mint 5%-al eltér a legjobb eredménytől. Ilyen hiba akkor fordulhat elő, ha a futtatott teszteken kívül más nagy erőforrás-igényű program terheli a processzort.

A tesztek eredményei a standard kimeneten kerülnek kiírásra. A kimenet első sorába a teszt megnevezése szerepel, ezt követően modulonként (illetve ritkás adatszerkezeti teszt esetében feltöltöttség szerint) külön-külön a mért műveleti gyorsaságok alapján számított MFlop/s eredmények következnek, minden paraméterezésre (**1. ábra**).

```

Sparse Vector/Dense Vector Addition:
Blaze <1% filled> [MFlop/s]:
1000      44.9805
1000000    8.43703
Blaze <10% filled> [MFlop/s]:
1000      297.173
1000000    51.8299
Boost uBLAS <1% filled> [MFlop/s]:
1000      2.14553
1000000    1.8936
Boost uBLAS <10% filled> [MFlop/s]:
1000      19.853
1000000    15.3758
PanOpt <1% filled> [MFlop/s]:
1000      1.97092
1000000    1.5839
PanOpt <10% filled> [MFlop/s]:
1000      19.4065
1000000    13.0223
GMM++ <1% filled> [MFlop/s]:
1000      33.9521
1000000    6.3062
GMM++ <10% filled> [MFlop/s]:
1000      276.947
1000000    40.1897

```

1. ábra. A Blazemark tesztek kimenete egy műveletre

Miután az összes modul tesztje befejeződött, kiírásra kerül egy összegzés, amely a paraméterezésben megadott összes tesztesetre kiírja a teszteset végrehajtásának számát, majd a könyvtárak egymáshoz viszonyított relatív gyorsaságát (a legjobb eredményt 1-nek véve). Ezen eredmények mögé kerül zárójelben a teszt elvégzéséhez szükséges tényleges idő másodpercben megjelenítve (2. ábra).

```

N=1000, fill level=1%, steps=8945864
PanOpt      = 22.822   (45.3892)
Blaze       = 1       (1.98883)
Boost uBLAS = 20.9647 (41.6953)
GMM++       = 1.32482 (2.63485)
N=1000000, fill level=1%, steps=1538
PanOpt      = 5.32673 (9.71019)
Blaze       = 1       (1.82292)
Boost uBLAS = 4.45556 (8.12211)
GMM++       = 1.33789 (2.43887)
N=1000, fill level=10%, steps=5319346
PanOpt      = 15.313  (27.4101)
Blaze       = 1       (1.78998)
Boost uBLAS = 14.9687 (26.7937)
GMM++       = 1.07303 (1.92071)
N=1000000, fill level=10%, steps=756
PanOpt      = 3.9801  (5.80544)
Blaze       = 1       (1.45862)
Boost uBLAS = 3.37088 (4.91682)
GMM++       = 1.28963 (1.88108)

```

2. ábra. a Blazemark tesztek kimenetének összegzése szakasza

3. A Blazemark rendszer kibővítése

Ezen fejezetben bemutatjuk, hogyan terjesztettük ki a Blazemarkot egy új könyvtármodullal, amely lehetővé teszi a PanOpt-hoz fejlesztett lineáris algebrai könyvtárak tesztelését a rendszer keretei között. Ezután ismertetjük a PanOpt műveleteinek az új Blazemark modulunkhoz szükséges mérési osztályait, valamint azok integrálásának folyamatát. Felsoroljuk továbbá a legfontosabb műveletek tesztjeit, amelyeket részletesen megvizsgáltunk munkánk folyamán.

3.1. A Blazemark kiterjesztése új modullal

Mint azt már említettük a Blazemark keretrendszer képes a legnépszerűbb nyílt forráskódú lineáris algebrai könyvtárak, a hagyományos tömb implementáció, illetve a standard vektorok műveleti teljesítményének összehasonlítására. A program kódjának bővítése azonban annak bonyolultsága miatt nehézkes, saját adatszerkezetek tesztelésére viszont csak egy új könyvtármodul hozzáadása útján van lehetőség. Ebben az alfejezetben a PanOpt könyvtármodul Blazemark-ba való beépítését mutatjuk be.

3.1.1. A Blazemark osztályainak módosítása

A Blazemark számos osztályában hivatkozik a beépített könyvtármodulokra, illetve sok helyen külön kezeli azokat. Új modul beépítése során az összes főbb osztály kódját módosítani kell, kiegészítve azokat úgy, hogy azok figyelembe vegyék az új könyvtárat. A keretrendszer által használt könyvtárak headerfájlokban lévő makrók útján vannak definiálva, és a teszt konfigurációs fájl által deklarált konstans változók állítják be azok használatát.

Egy új makrót tároló headerfájl és az új konfigurációs beállítás hozzáadása után a tesztek modulonkénti iterálásért felelős osztályokat kiegészítettük az új modulra vonatkozó változókkal és függvényekkel. Ezek az osztályok a Blazemark-ban a *SolverRun*, a *DynamicDenseRun*, a *DynamicSparseRun*, a *StaticDenseRun* és a *StaticSparseRun*, amelyek közül a tesztelt adattípustól függően alkalmaz egyet a rendszer a futtatandó teszt iterálásához. Az iteráló osztályok hasonló felépítésűek, mindegyikben létrehoztuk a PanOpt modul eredményét tároló változót, ennek lekérdező (getter) és beállító (setter) függvényét, illetve inicializáltuk azokat az osztályok konstruktoraiban. Mivel az új modul eredményének kiírását is ezek az osztályok végzik, módosítanunk kellett a `<<` operátorokat is.

A tesztek futtatása során azok parancssori paramétereinek feldolgozásáért és a futtatandó modulok kiválasztásáért a Blazemark *Benchmarks* osztálya a felelős. Itt egy új futtatási argumentumot vezettünk be, valamint az osztály *parseCommandLineArguments* függvényét módosítottuk a többi modul esetében is alkalmazott argumentumkezelés mintája alapján. Mindez lehetővé teszi, hogy parancssori paraméterek alapján tudjunk tesztek futtatni az új

könyvtármodul felhasználása nélkül (-no-panopt), vagy annak kizárólagos felhasználásával (-only-panopt).

3.1.2. A Makefile kiegészítése

Mivel a Blazemark keretrendszer fordítását lehetővé tevő Makefile egy konfigurációs script alapján jön létre, ennek a scriptnek pedig része a Blazemark-ba épített összes könyvtármodul minden teszthez való hozzárendelése, a kiegészített rendszer működéséhez módosítani kell a konfigurációs scriptet, vagy közvetlenül az általa létrehozott Makefile-t.

Mind a script, mind pedig a Makefile szerkesztése igen bonyolult, azok szerkezete nehezen áttekinthető. Bár a fájlok könnyen azonosítható minták alapján épülnek fel (minden egyes tesztet hasonlóképpen kell fordítani), a Makefile több mint 3000, a konfigurációs script pedig csaknem 8500 sorból áll. Mivel a fordítói konfiguráció minden beállítása módosítható magában a Makefile-ban is, ezért ennek kiegészítése mellett döntöttünk. Módosításaink után tehát már nincs szükség a konfigurációs script futtatására, jelenleg a kiegészített Makefile-unk elején található változók módosításával konfigurálható a fordító.

A PanOpt elérési útvonalának és a lefordított binárisok helyének megadása után a tesztek fordítását végző parancsokhoz hozzáadtuk a PanOpt mérési osztályainak fordítását is. Amennyiben az adott tesztet nem támogatja a PanOpt és a kapcsolódó mérési osztály nem létezik, a fordító csak az azt támogató könyvtárak moduljait használja fel.

3.2. A Pannon Optimizer tesztjeinek integrálása

A PanOpt lineáris algebrai könyvtárainak a Blazemark segítségével való teszteléséhez szükség van a PanOpt könyvtármoduljához megírt mérési osztályokra, amelyek a tesztek által vizsgált elemi művelet-implementációk teljesítményének méréséért felelősek. Ezen alfejezetben bemutatjuk egy mérési osztály megírásának folyamatát, és a PanOpt könyvtármoduljához megírt mérési osztályainkat.

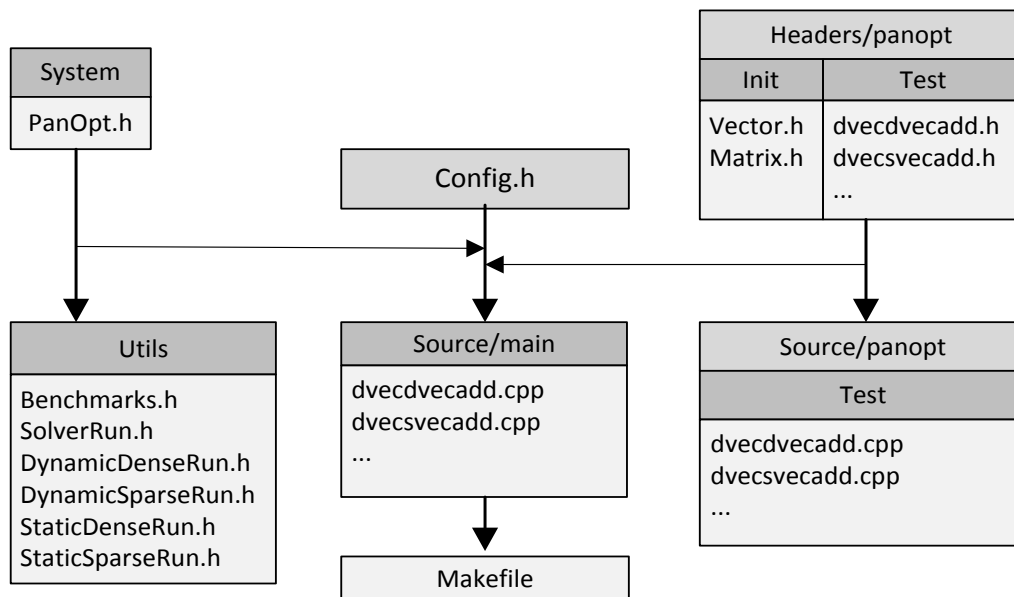
3.2.1. A mérési osztályok

Mivel minden mérési osztály a könyvtár egyik adatszerkezetének egy műveletét teszteli, ezért meg kellett írunk a PanOpt adatszerkezeteinek inicializáló függvényeit, amelyek a Blazemark osztályait használják az adatszerkezetek feltöltésére. Mint azt már említettük, a PanOpt adatszerkezetei képesek váltani sűrű és ritkás tárolási mód között, így csak két osztály, a Vector és a Matrix inicializáló függvényeit kellett elkészíteni. Ezekhez azonban két implementáció is tartozik, mivel a ritkás és sűrű feltöltést külön kezeli a rendszer.

A Blazemark rendszer felépítéséből adódóan minden mérési osztályhoz tartozik egy header (ami csak a könyvtár tesztjéhez tartozó függvény deklarációját tartalmazza) és egy forrásfájl, amelyeket az összes PanOpt által támogatott tesztre külön-külön elkészítettünk. A

forrásfájlban az adott művelet PanOpt által implementált verzióját mérő függvény található. Ez az inicializáló függvények meghívásából, majd a iteratív elvégzéséből áll. Ennek idejét a *Blaze Timer* osztályának segítségével rögzítve kapunk meg egy részeredményt. Egy teszt eredményeként az ismétlések során kapott legjobb időeredmény kerül eltárolásra. Ha az időeredmények átlaga több mint 5%-al eltér a legjobb eredménytől, a rendszer esetleges mérési hibát jelez. Ugyanígy jelzi, ha a művelet eredményeképpen kapott adatszerkezet mérete nem egyezik az inicializálás során megadott mérettel, ebben az esetben az adott művelet implementációja hibás.

A PanOpt által támogatott tesztek integrálásához a Blazemark iterációs osztályait alkalmazó fő tesztfüggvényét is módosítani kellett, mivel az gyűjti össze a megfelelő iterációs osztályoktól a mért eredményeket, majd az dolgozza fel a beérkezett adatokat és írja ki azokat a kimenetre. Ezen fő tesztfüggvény kiegészítése után a Blazemark képessé vált a PanOpt eredményeinek lemerésére az adott teszt esetében. (3. ábra)



3. ábra. A Blazemark PanOpt modul hozzáadása miatt módosított fájljainak vázlata

3.2.2. A PanOpt könyvtármodul által támogatott tesztek

A Blazemark lineáris algebrai tesztjei közül a PanOpt könyvtármodulja 66-ot támogat. Az alábbi táblázatok tartalmazzák a támogatott teszteket. A vektorműveletek leírásánál sűrű vektorokra az $\mathbf{a}$ és $\mathbf{b}$, ritkás vektorokra a $\mathbf{c}$ és $\mathbf{d}$ jelölést használtuk. (1. táblázat)

1. táblázat. A PanOpt modul vektorműveleti tesztjei

Blazemark név	Teszt megnevezése	Művelet
dvecdvecadd	sűrű vektor – sűrű vektor összeadás	$\mathbf{a} + \mathbf{b}$
dvecsvecadd	sűrű vektor – ritkás vektor összeadás	$\mathbf{a} + \mathbf{c}$
svecdvecadd	ritkás vektor – sűrű vektor összeadás	$\mathbf{c} + \mathbf{a}$
svecsvecadd	ritkás vektor – ritkás vektor összeadás	$\mathbf{c} + \mathbf{d}$
dvecdvecsub	sűrű vektor – sűrű vektor kivonás	$\mathbf{a} - \mathbf{b}$
dvecscalarmult	sűrű vektor skalárral való szorzása	$\lambda \mathbf{a}$
svecscalarmult	ritkás vektor skalárral való szorzása	$\lambda \mathbf{c}$
daxpy	sűrű vektorok daxpy szorzata	$\mathbf{a} + \lambda \mathbf{b}$
dvecnorm	sűrű vektor Euklideszi normája	$ \mathbf{a} = \sqrt{\mathbf{a}^T \mathbf{a}}$
tdvecdvecmult	sűrű vektor – sűrű vektor skaláris szorzás	$\mathbf{a}^T \mathbf{b}$
tdvecsvecmult	sűrű vektor – ritkás vektor skaláris szorzás	$\mathbf{a}^T \mathbf{c}$
tsvecdvecmult	ritkás vektor – sűrű vektor skaláris szorzás	$\mathbf{c}^T \mathbf{b}$
tsvecsvecmult	ritkás vektor – ritkás vektor skaláris szorzás	$\mathbf{c}^T \mathbf{d}$

A mátrixműveletek leírásánál az **A** és **B** sűrű, a **C** és **D** pedig ritkás mátrixokat jelöl. (

2. táblázat) Egyes könyvtárak lineáris algebrai művelet-implementációi külön kezelnek sorfolytonosan, illetve oszlopfolytonosan tárolt mátrixokat, amely miatt a műveletekben szereplő transzponált mátrix nem egy mátrix transzponálásával, hanem egy külön adatszerkezet használatával kapható meg.

2. táblázat. A PanOpt modul vektor – mátrix műveleti tesztjei

Blazemark név	Teszt megnevezése	Művelet
dmatdvecmult	sűrű mátrix – sűrű vektor szorzás	Aa
tdmatdvecmult	transzpontált sűrű mátrix – sűrű vektor szorzás	$A^T a$
dmatsvecmult	sűrű mátrix – ritkás vektor szorzás	Ac
tdmatsvecmult	transzponált sűrű mátrix – ritkás vektor szorzás	$A^T c$
smatdvecmult	ritkás mátrix – sűrű vektor szorzás	Ca
tsmatdvecmult	transzponált ritkás mátrix – sűrű vektor szorzás	$C^T a$
smatsvecmult	ritkás mátrix – ritkás vektor szorzás	Cc
tsmatsvecmult	transzponált ritkás mátrix – ritkás vektor szorzás	$C^T c$
tdvecdmatmult	transzpontált sűrű vektor – sűrű mátrix szorzás	$a^T A$
tdvectdmatmult	transzponált sűrű vektor – transzponált sűrű mátrix	$a^T A^T$
tdvecsmatmult	transzponált sűrű vektor – ritkás mátrix szorzás	$a^T C$
tdvectsmatmult	transzponált sűrű vektor – transzponált ritkás	$a^T C^T$
tsvecdmatmult	transzponált ritkás vektor – sűrű mátrix szorzás	$c^T A$
tsvectdmatmult	transzponált ritkás vektor – transzponált sűrű	$c^T A^T$
tsvecsmatmult	transzponált ritkás vektor – ritkás mátrix szorzás	$c^T C$
tsvectsmatmult	transzponált ritkás vektor – transzponált ritkás	$c^T C^T$

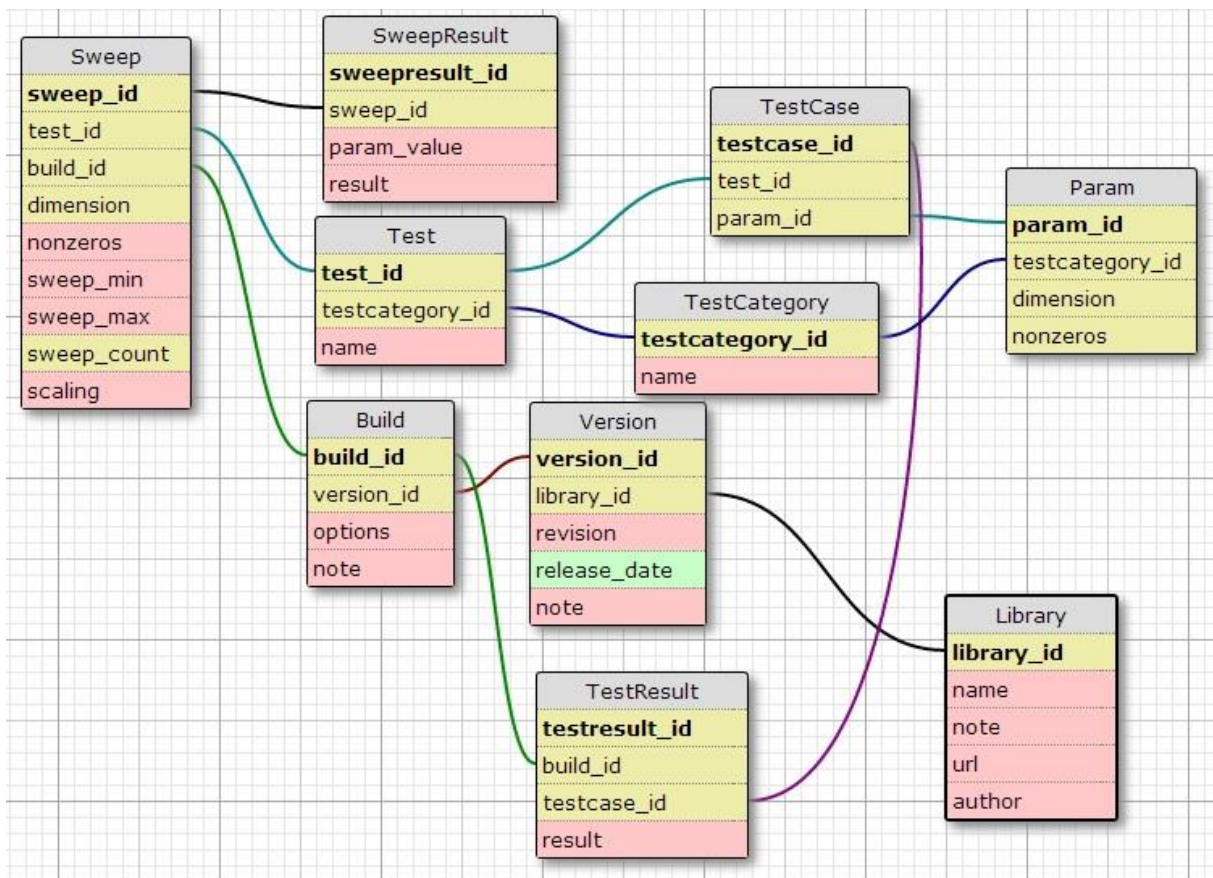
A PanOpt modul mátrixműveleti tesztjei tartalmazzák a sűrű és ritkás mátrixok transzponálását, skalárral való szorzását, a sűrű mátrix – sűrű mátrix kivonást, illetve a sűrű és ritkás mátrixok és transzponált mátrixok bármilyen párosításban való összeadását és szorzását. Mivel a PanOpt jelenlegi verziójában a mátrixműveletek még fejlesztés alatt állnak, és számos adatszerkezeti sajátosságot kihasználó gyorsító módszer nincs implementálva, ezen tesztek vizsgálata csak a PanOpt Matrix osztályának későbbi, fejlettebb verzióiban ad felhasználható eredményt.

4. A BlazemarkAnalyser elemző szoftver

Mint azt már korábban említettük, a Blazemark keretrendszer korlátai közé tartozik az eredmények nehézkes áttekinthetősége, illetve nincs lehetőség azok rendszerezésére, elemzésére. Ebben a fejezetben bemutatjuk a Blazemark rendszerre épülő, annak lehetőségeit kiterjesztő saját fejlesztésű szoftverünk, a BlazemarkAnalyser működését. Részletesen tárgyaljuk továbbá a könyvtárakat, tesztek, és teszteredményeket tartalmazó és rendszerező adatbázisunkat, valamint az eredmények felhasználóbarát elemzését támogató grafikus felhasználói felületet is. Végül prezentáljuk a szoftver segítségével elvégzett méréseink eredményeit és az ezekből levont következtetéseinket.

4.1. A BlazemarkAnalyser adatbázisa

A BlazemarkAnalyser által használt és kezelt SQLite adatbázisunk 10 táblából áll. Az adatbázis szerkezete a következő ábrán (4. ábraHiba! A hivatkozási forrás nem található.) látható, melyen az egyes adattípusokat külön színnel jelöltük. Az adatbázis segítségével könnyen tárolhatók, rendszerezhetők és vizsgálhatók a mért eredmények, valamint az adatbázis kész a Blazemark esetleges bővítése esetén új könyvtárak támogatására. Ezen alfejezetben ennek az adatbázisnak a felépítését tárgyaljuk, melynek táblái a következők:



4. ábra. A BlazemarkAnalyser adatbázis-struktúrája

Library

A Library tábla a különböző könyvtárakat tárolja, amelyek műveleti gyorsaságát tesztelhetjük. Mezői a *library_id* elsődleges kulcs, amely a könyvtár egyedi azonosítója az adatbázisban, a *name*, amelyben a könyvtár neve tárolható, a *note*, amelyben a könyvtárhoz fűzött saját megjegyzést tároljuk, az *url*, amely a könyvtár hivatalos honlapjának a linkjét tárolja, illetve az *author*, amely a könyvtár szerzőjét, illetve szerzőit jelöli.

Version

A Version tábla a könyvtárak verzióit tárolja, azok közzétételi dátumával. A verziók külön kezelésével mérhető például a könyvtár régebbi verziója és a friss verzió közötti teljesítménybeli eltérés. Mezői a *version_id* elsődleges kulcs, amely a verzió egyedi azonosítója az adatbázisban, a *library_id*, amely a verzió könyvtárának azonosítója, a *revision*, amely a verziószámot jelöli, a *release_date*, amely a közzététel idejét, illetve a *note*, amely a verzióhoz fűzött egyéni megjegyzést tárolja.

Build

A Build tábla a könyvtárak verzióinak egyes fordításaihoz tartozó különböző fordítói beállításait tárolja. Ennek a táblának a bevezetésével van lehetőségünk különböző fordítók, illetve különböző beállítások mellett összehasonlítani egy adott könyvtárverziót. Mezői a *build_id* elsődleges kulcs, amely a beállítási konfiguráció egyedi azonosítója az adatbázisban, a *version_id*, amely a bejegyzéshez tartozó verzió azonosítója, az *options*, amely a fordítót meghívó parancsot és a hozzá tartozó beállítási paramétereket tárolja, illetve a *note*, amelyben a beállításokhoz tartozó egyéni megjegyzés tárolható.

TestCategory

A TestCategory tábla a futtatható tesztek különböző kategóriáit tárolja. Jelenleg négy ilyen kategóriába soroljuk a tesztek az általuk felhasznált adatszerkezetek szerint beszélhetünk sűrű vektor, ritkás vektor, sűrű mátrix és ritkás mátrix tesztekéről. Tesztkategóriák bevezetésére a tesztek eltérő paraméterezhetősége miatt van szükség, egy ritkás vektorokkal dolgozó teszthez szükséges például az adatszerkezetekben szereplő nem-nulla elemek számának megadása, míg sűrű vektorok esetében nem. Mezői a *testcategory_id* elsődleges kulcs, amely a tesztkategória egyedi azonosítója az adatbázisban, és a *name*, amely a tesztkategória neve.

Test

A Test tábla a Blazemark által implementált futtatható tesztek tárolja. Mezői a *test_id* elsődleges kulcs, amely a teszt egyedi azonosítója az adatbázisban, a *testcategory_id*, amely a azon tesztkategória azonosítója, amelyikhez a teszt tartozik, illetve a *name*, amely a teszt Blazemarkon belüli neve.

Param

A Param tábla a tesztkategóriák alapértelmezett paraméterezéseit tárolja. A paraméterezések azonban tesztenként is specializálhatók, így előfordulhat, hogy egy paraméterezés csak egy konkrét tesztre van felhasználva. Mezői a *param_id*, amely a paraméterezés egyedi azonosítója az adatbázisban, a *testcategory_id*, amely a paraméterezéshez tartozó tesztkategória azonosítója, a *dimension*, amely a paraméterezett teszt futtatása során használt adatszerkezetek dimenziója, illetve a *nonzeros*, amely az adatszerkezetekben szereplő nem-nulla elemek száma (sűrű adatszerkezeteknél ez az érték -1).

TestCase

A TestCase tábla a felparaméterezett teszteseteket tárolja. Minden tesztesetnél tárolásra kerül a *testcase_id* elsődleges kulcs, amely a teszteset adatbázison belüli egyéni azonosítója, továbbá a *test_id* és a *param_id*, amelyek egy tesztet és egy paraméterezést kapcsolnak össze azonosító alapján.

TestResult

A TestResult tábla a különböző buildek különböző tesztesetek futási eredményeit tárolja. Mezői az *testresult_id* elsődleges kulcs, amely az eredmény adatbázison belüli egyedi azonosítója, a *build_id*, amely a tesztelés során használt buildet, a *testcase_id*, amely a tesztesetet, illetve a *testresult*, amely a teszt eredményét tárolja. Mivel az eredmény a mért MFlop/s értéken kívül hibaüzenet is lehet (például, ha a kiválasztott build-hez tartozó könyvtár nem támogatja az adott tesztet) a testresult mező szöveges típusú.

Sweep

A Sweep tábla az elemzések megkönnyítése céljából készült, paraméterek kis változtatásával készített méréssorozatok leírását tárolja. Mezői a *sweep_id* elsődleges kulcs, a *test_id* és a *build_id*, amelyek a sorozatban elvégzett teszt és a tesztelt build azonosítóját tárolják, a *dimension*, és a *nonzeros*, amelyek a méréssorozat paramétereit jelölik, ezek jelölik a sweep során változtatott paramétert is. A *sweep_min* és *sweep_max* értékek a tesztsorozat változtatott paraméterének minimális és maximális értékét tárolják, a *sweep_count*, a méréssorozat által végrehajtandó tesztesetek számát, illetve a *scaling*, amely a paraméterek megjelenítésének skáláját (lineáris, vagy logaritmikus) tárolja.

SweepResult

A SweepResult tábla a sweep-ek változtatott paraméter-értékekhez tartozó eredményeit tárolja. Mezői a *sweepresult_id*, amely elsődleges kulcsként a SweepResult egyedi azonosítója, a *sweep_id*, amely az eredményhez tartozó sweep azonosítója, a *param_value*, amely a sweep változtatott paraméterének aktuális értéke, illetve a *result*, amely az ehhez tartozó eredményt (MFlop/s) tárolja.

4.2. A BlazemarkAnalyser működése

A BlazemarkAnalyser program C++ nyelven, a Qt 5.0.2 fejlesztői könyvtárainak segítségével készült. Működése során el tudja látni az adatbázis kezelésének, az eredmények rendszerezésének, a Blazemark rendszer átkonfigurálásának feladatát, valamint megkönnyíti az eredményeink feldolgozását, egy felhasználóbarát grafikus felhasználói felület segítségével.

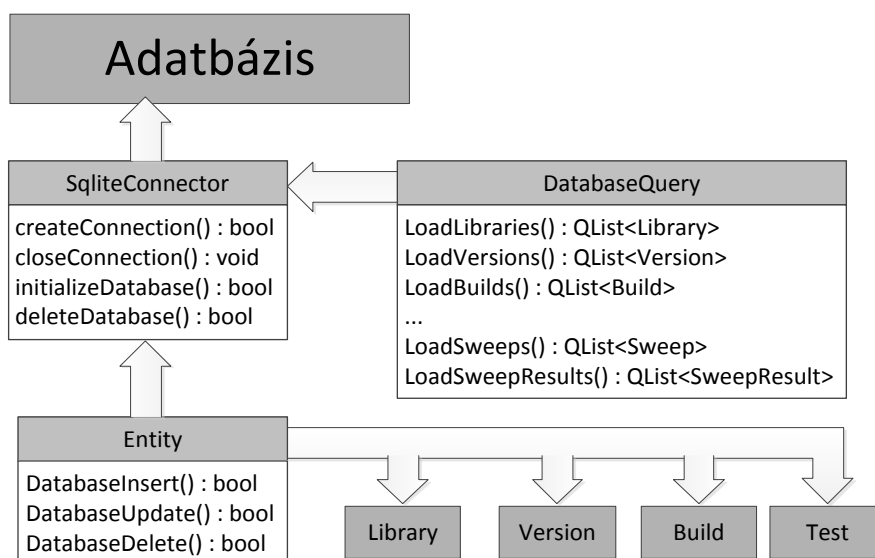
Ezen alfejezetben áttekintést nyújtunk a program készítése során felhasznált módszereinkről, illetve a főbb feladatok megvalósításáról.

4.2.1. Adatbázis-kezelés

Az adatbázis kezelésére két osztályt és egy ezek által használt entitásrendszert alkalmaztunk, amely az adatbázis tábláinak megfelelő oszályokból áll. (5. ábra)

Az *SqliteConnector* osztály

Az *SqliteConnector* osztály kezeli az SQLite adatbázissal való kapcsolatfelvételt, a lekérdezések futtatását az adatbázison, az adatbázis esetleges törlését, illetve a kapcsolat lezárását lehetővé. Amennyiben az adatbázis még nem létezik, képes azt egy előre tárolt script alapján létrehozni és inicializálni azt. Az *SqliteConnector* osztályt használhatják tehát a program egyéb osztályai és az adatbázis közötti kommunikációra. Ha hiba lép fel az adatbázishoz való kapcsolódás, vagy a lekérdezések végrehajtása során, függvényei hibajelző értékekkel térnek vissza.



5. ábra. A BlazemarkAnalyser adatbázis-kezelő osztályai (részlet)

A *DatabaseQuery* osztály

A *DatabaseQuery* osztály feladata a megfelelő lekérdezések segítségével betölteni az adatbázisból kinyert információkat a program objektumaiba. A *DatabaseQuery* az imént

ismertetett *SqliteConnector* objektum segítségével kommunikál az adatbázissal, bármelyik tábla bejegyzéseinek lekérdezését meg tudja valósítani. Az információkat a lekérdezett táblának megfelelő entitások listájaként adja vissza. Nem csak egy adott tábla lekérdezésére képes, hanem rendelkezik az egymáshoz kapcsolódó táblák összekapcsolt mezői alapján való lekérésre alkalmas függvényekkel is, ilyen lehet például egy adott könyvtárhoz tartozó verziók lekérdezése.

Az entitásosztályok

Az entitások (6. ábra) tárolják a hozzájuk tartozó adatbázistábla egy bejegyzésének minden adatát, illetve ezen adatok futás közbeni módosításait, melyeket az *SqliteConnector* objektum segítségével tudnak menteni az adatbázisba. Ezek az entitásosztályok azt a célt szolgálják, hogy a program dinamikusan és egyszerűen kezelhesse az adatbázis adatait, azokat közvetlenül a memóriából érje el az adatbázisból való állandó lekérdezés helyett. Az összes ilyen osztály egy általános *Entity* osztályból van származtatva. Örökölt függvényei a *DatabaseInsert* függvény, ami az entitást új bejegyzésként írja be az adatbázisba, majd lekéri annak kapott egyedi azonosítóját, és beállítja az entitás megfelelő adattagját erre az értékre, a *DatabaseUpdate* függvény, ami az entitás adattagjai alapján frissíti az adatbázisban található bejegyzését, így a program által alkalmazott minden módosítás mentésre kerül, valamint a *DatabaseDelete* függvény, ami törli az entitás bejegyzését az adatbázisból.

Version : Entity	
Adattagok	Függvények
m_versionId : int m_libraryId : int m_revision : QString m_releaseDate : QDateTime m_note : QString m_parentLibrary : Library*	DatabaseInsert() : bool DatabaseUpdate() : bool DatabaseDelete() : bool Adattagok getter függvényei Adattagok setter függvényei

6. ábra. A Version entitásosztály felépítése

Az entitások kezelése

A BlazemarkAnalyser a program elindításakor csatlakozik az adatbázishoz, és annak teljes tartalmát (a SweepResult táblán kívül) betölti entitások listájaként a memóriába. Mivel a listák elemei közül a kiválasztás gyorsabb, mint az adatok adatbázisból való beolvasása (kivéve a SweepResult tábla esetében, mivel itt több ezer bejegyzést tárol az adatbázis). A DatabaseQuery adatbázis-beolvasó függvényeit a program indulásán kívül tehát csak a Sweepek grafikus megjelenítése, illetve új elemek létrehozása vagy törlése esetén használja a program. A táblák összefüggései alapján az entitásokban pointereket alkalmaztunk, így az adatbázistáblák közötti hivatkozások könnyen és gyorsan kezelhetők, az entitáslistákban való keresés nélkül.

4.2.2. A Blazemark keretrendszer kezelése

Paraméterezés és tesztfuttatás

A Blazemark rendszer konfigurálását és működtetését a programunk `BlazemarkProcessor` osztálya látja el, amelynek adattagjai tartalmazzák az összes olyan beállítást, amely a Blazemark újrakonfigurálásához szükséges: egy `Test` (`m_selectedTest`), egy `Library` (`m_selectedLibrary`), egy `Version` (`m_selectedVersion`), egy `Build` (`m_selectedBuild`) és egy `Param` lista (`m_params`) formájában. Rendelkezik továbbá egy saját `QProcess` objektummal (`m_process`), amely segítségével végrehajthatók a Blazemark fordításához és a megfelelő teszt futtatásához szükséges parancsok.

A `BlazemarkProcessor` osztály először kiválasztja a végrehajtani kívánt tesztet, majd ennek Blazemark paraméterfájlját a megadott paraméterezések alapján szerkeszti. Ezt követően a Blazemark `Makefile`-jában módosítja a fordítói beállításokat, illetve a tesztelt könyvtár elérési útvonalát a tárolt adatok alapján. Ezen módosítások után a Blazemark főkönyvtárában futtatja a "make clean" parancsot, amely törli az esetlegesen régebbi beállításokkal fordított tesztek fordított állományait, majd a "make" parancs után a megfelelő paramétert használva, lefordítja a kiválasztott tesztet az új beállításokkal.

A teszt fordítása után a `BlazemarkProcessor` futtatja a teszt elindítását végrehajtó parancsot. Ezzel a módszerrel egy futtatással csak egyetlen könyvtár tesztelhető, viszont jóval egyszerűbb a hibák megfelelő jelzése és a kimenet feldolgozása. Hibajelzésként megvalósítottuk, hogy amennyiben egy teszt futása nem fejeződik be 30 perc alatt, a `BlazemarkProcessor` automatikusan megszakítja annak futását, és hibát jelez.

Az eredmények feldolgozása

Az Blazemark teszt kimenetének értelmezéséért a program `parseBlazemarkOutput` függvénye, az eredmények feldolgozásáért pedig a `processResult` és `processSweepResult` függvények a felelősek. A teszt kimenete az azt futtató `BlazemarkProcessor` objektumtól kérhető le. Mivel egyszerre csak egy könyvtár tesztelése folyhat, a kimenetből elég a különböző paraméterezésekhez tartozó eredményeket kiválasztani. Ha a `BlazemarkProcessor` hibával tért vissza, az összes kiválasztott paraméterezés teszteredményének helyére a hiba típusa kerül.

A paraméterezés és a teszt ismeretében a `processResult` függvény megkeresi az ezekhez tartozó `TestCase` entitást, vagy létrehoz egy újat, majd a tesztelt `Build` és a `TestCase` párosításával létrejövő `TestResult` eredményét rögzíti az adatbázisba. A `processSweepResult` függvény esetében nincs szükség a `TestCase` és `TestResult` objektumok kiválasztására és vizsgálatára, így annak eredménye rögtön bekerülhet az adatbázisba.

A tesztek ütemezése

Mivel egy ütemben csak egy Buildet lehet tesztelni, több teszt, vagy Build kiválasztása esetén szükség van a tesztek ütemezésére. Ezt a BlazemarkAnalyser-ben egy FIFO (First-In First-Out) listával valósítottuk meg.

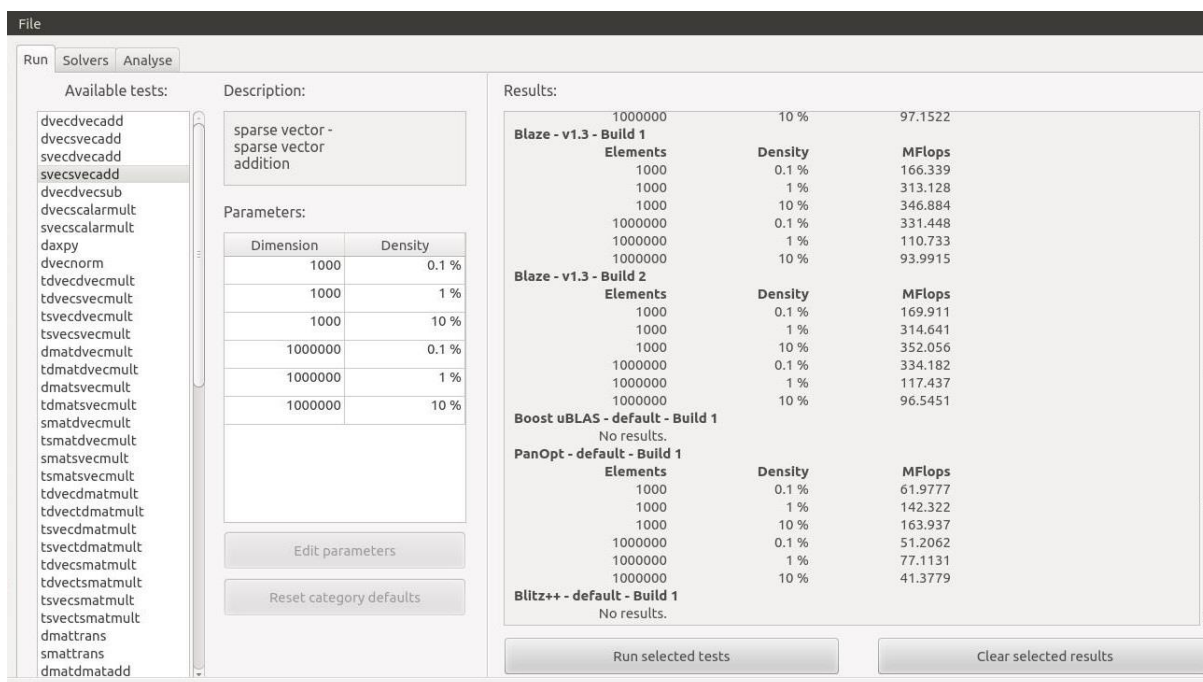
Mivel egy teszt futtatása közben a program egyéb funkciói nem állhatnak le, a BlazemarkProcessor objektum egy külön szálon működik, és futásának befejezése után meghívja *processReady* függvényt, amely az eredmény-feldolgozó függvények végrehajtása után elindítja a következő ütemezett tesztet.

4.3. A BlazemarkAnalyser kezelőfelülete

A BlazemarkAnalyser grafikus felhasználói felülete három különböző fülből áll, amelyek a felületet a Run, a Solvers és az Analyse részekre osztják. A Run fül a tesztek futtatásához és az eredmények megtekintéséhez készítettük, a Solvers fül a könyvtárakhoz tartozó verziók és buildek menedzselését szolgálja, az Analyse fül az eredmények grafikus szemléltetéséhez és elemzéséhez nyújt nagy segítséget, illetve itt található a sweepek futtatásához szükséges kezelőfelület is.

4.3.1. A Run fül

A Run fül (7. ábra) két listából áll a tesztek kiválasztásához és paraméterezéseik menedzseléséhez, illetve egy különálló részben a kiválasztott teszthez tartozó összes teszteredmény kerül megjelenítésre. Ez elsőként a buildek, azokon belül pedig paraméterek szerint növekvő sorrendben történik. A teszt kiválasztásakor a program kiválasztja a tesztkategóriához tartozó paraméterezéseket, és megjeleníti a paraméterek kijelzésére szolgáló listában, valamint a teszt leírását a "Description" dobozban.



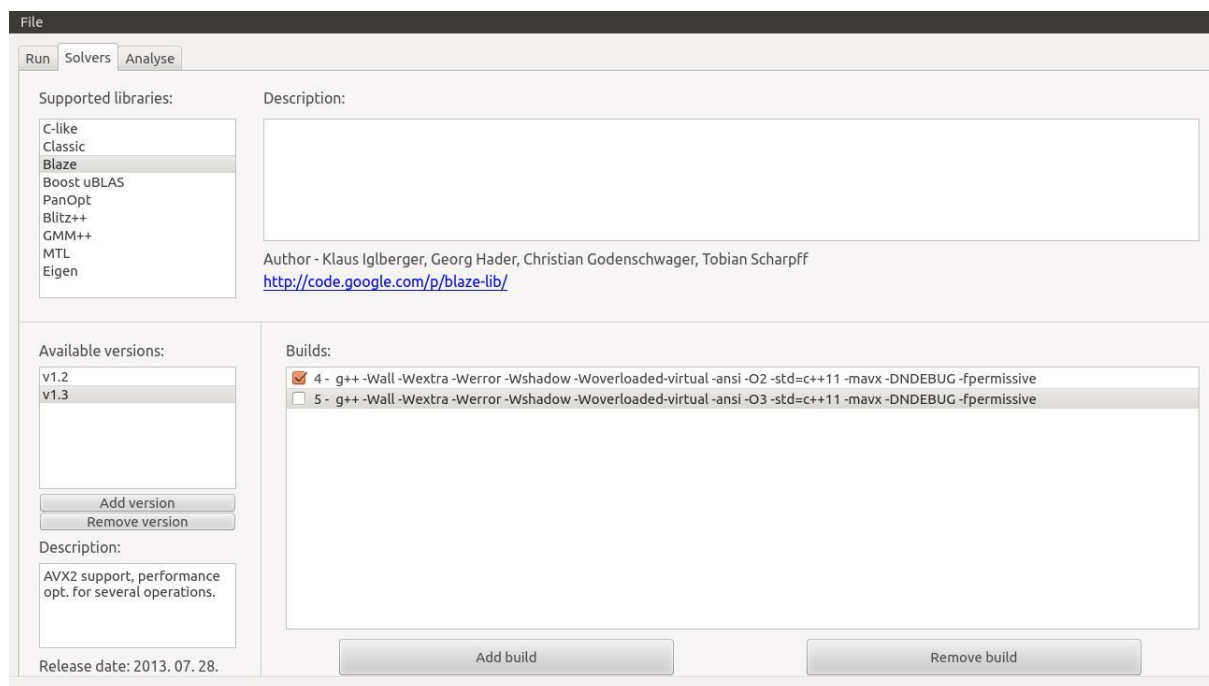
7. ábra. A BlazemarkAnalyser Run fül alatti kezelőfelülete

A paraméterlistát az "Edit parameters" gombbal lehet szerkeszteni, amely egy párbeszédablakot jelenít meg paraméterek hozzáadásához vagy törléséhez. A listához jelenleg maximum 10 különböző paraméterezés adható, a paraméterezések hozzáadása, illetve törlése esetén a változtatások rögzítve lesznek az adatbázisban is. A "Reset category defaults" gomb pedig visszaállítja az adott tesztkategóriához alapértelmezetten hozzárendelt paraméterezéseket.

A beállított buildeket az adott paraméterezésekkel a „Run selected test” gombbal lehet futtatni. Ekkor a gomb inaktívvá válik, és egy "Working..." felirat jelenik meg rajta, jelezve, hogy tesztelés van folyamatban. A teszt befejeződése után a program frissíti az eredményeket a Results részben, és újra aktívvá válik a tesztfuttatás gombja. A „Clear selected results” gomb egy felugró ablakon kért nyugtázás után a beállított buildek kiválasztott teszttel való eredményeit törli az adatbázisból.

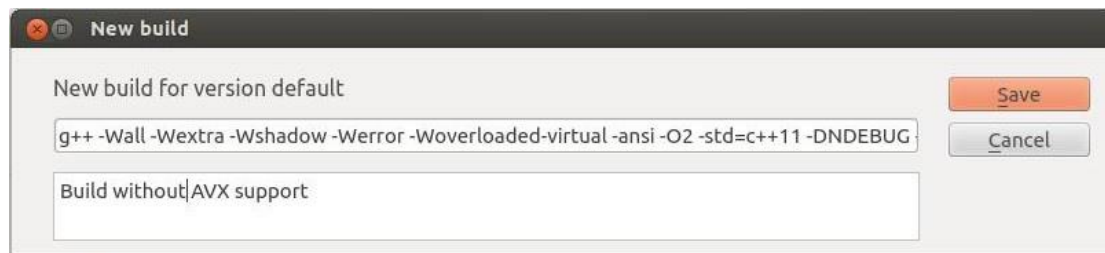
4.3.2. A Solvers fül

A Solvers fül (8. ábra) a könyvtárakat, a kiválasztott könyvtárhoz tartozó verziókat, és a kiválasztott verzióhoz tartozó buildeket tartalmazó listákból, valamint az ezekhez tartozó adatokat kijelző elemekből áll. Ez a könyvtárak esetében ezek a felületelemek a szerkeszthető megjegyzést tartalmazó szövegdoz, a könyvtár szerzőjének (vagy szerzőinek) neve, és a könyvtár hivatalos honlapjára hivatkozó link. Verziók esetében pedig a verzió leírását tartalmazó szövegdoz, és az annak megjelenési idejét jelző címke.



8. ábra. A BlazemarkAnalyser Solvers fül alatti kezelőfelülete

Verziók és buildek esetében lehetőség van új elemek hozzáadására, illetve meglévő elemek eltávolítására. Új elemek hozzáadása esetén egy felugró párbeszédablakban van lehetősége a felhasználónak az új verziót, vagy buildet meghatározni. (9. ábra) Meglévő elemek törlése esetén a törlést nyugtázni kell egy felugró ablakban, továbbá a törlés gomb teszt futása közben inaktív, hiszen a futtatott teszt által tesztelt verzió, vagy build törlése esetén a teszt eredményei nem lennének használhatóak.



9. ábra. A BlazemarkAnalyser új Build létrehozására szolgáló párbeszédablaka

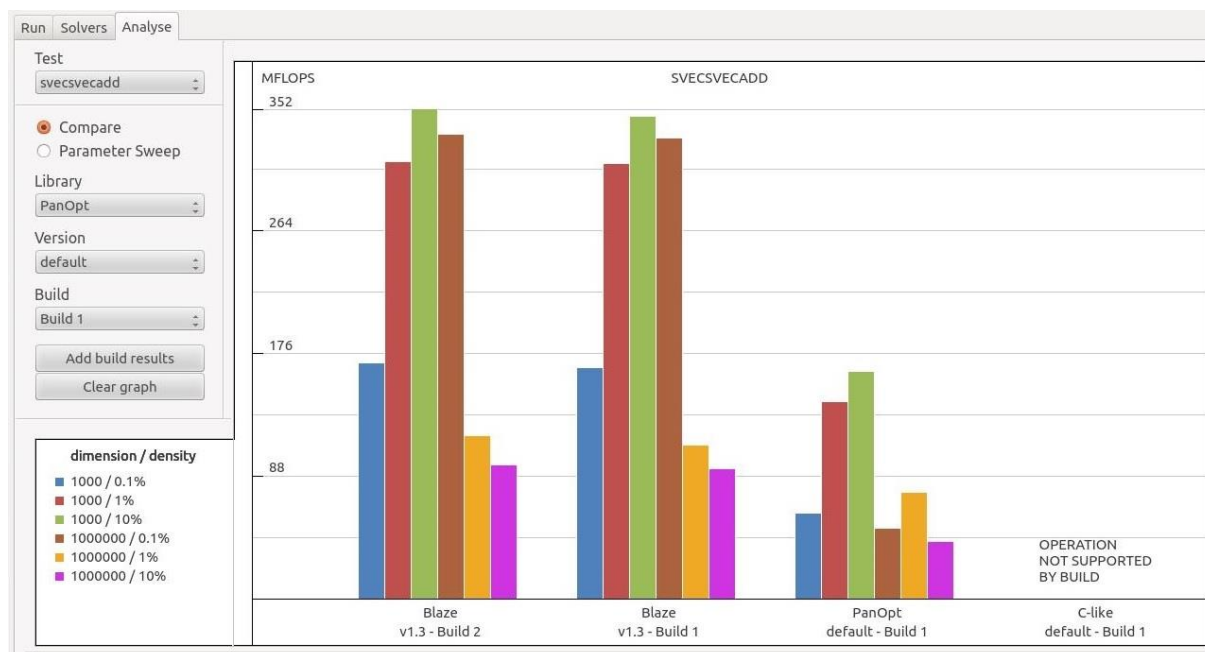
Egy új verzió első használata esetén a program bekéri a könyvtár adott verziójának elérési útvonalát. A buildeket tartalmazó lista elemei előtt lévő választódoboz segítségével lehet kiválasztani, hogy tesztelés során a program mely build teljesítményét mérje le.

4.3.3. Az Analyse fül

Az Analyse fül két rajzolófelületről áll, melyeken a grafikus eredménykijelző, valamint az ehhez tartozó jelmagyarázat kapott helyet. Tartalmazza továbbá a mód kiválasztására szolgáló gombokat (összehasonlító mód / sweep mód kiválasztása) és négy legördülő listát (teszt, könyvtár, verzió és build), amelyek segítségével az összehasonlítani kívánt eredményeket választhatjuk ki.

Összehasonlító módban (10. ábra) a teszt kiválasztásakor a program azon könyvtárakkal tölti fel a felületet, amelyekhez már van az adott tesztből futtatási eredmény. A könyvtár kiválasztása után a verzióválasztó lista, valamint a buildek listáját hasonlóképp kerül feltöltésre.

A kijelző felülethez a mérések eredményeit az „Add build results” gombbal lehet hozzáadni, amely már hozzáadott build kiválasztása esetén „Remove build results” gombra vált át, így azok eltávolításra is van lehetőség. Ehhez a kijelző felülethez tetszőleges számú build eredménye hozzáadható, amely ezek futási eredményeit oszlopdiagramként jeleníti meg. A skálázás a legmagasabb mérési eredményhez igazodik, a kijelző baloldai tengelyén jelezve a főbb értékeket. Minden build esetében annyi oszlopot rajzol a programunk, ahány különböző paraméterezéssel lett lefuttatva az adott teszt.

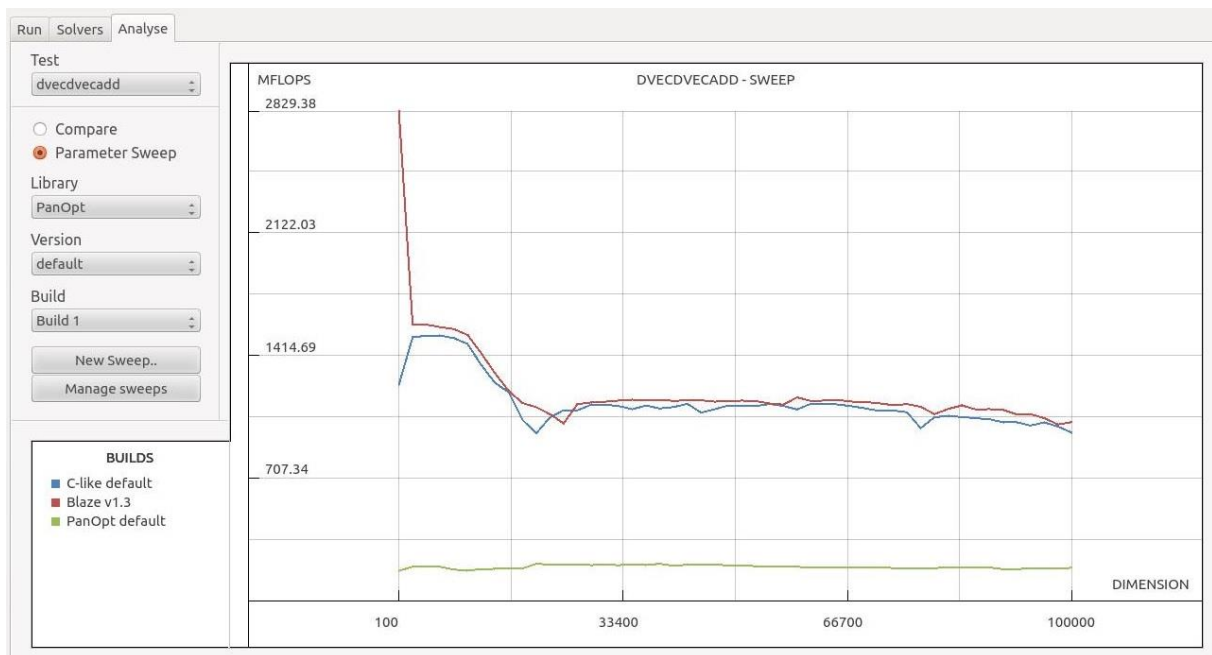


10. ábra. A BlazemarkAnalyser Analyze fül alatti felülete összehasonlító módban

Az eltérő paraméterezések eltérő színű oszlopokként kerülnek megjelenítésre, a jelmagyarázatban pedig automatikusan megjelenik a színezéshez tartozó paraméterezés. Ha egy build nem támogatja az adott tesztet, a felülethez adva ez is kiírásra kerül. Az oszlopokcsoportok alatt megjelenítjük, hogy az adott eredmények mely könyvtár melyik verziójához és buildjéhez tartoznak. A „Clear graph” gomb eltávolítja a grafikonról az összes hozzáadott eredményt, így alaphelyzetbe állítja azt.

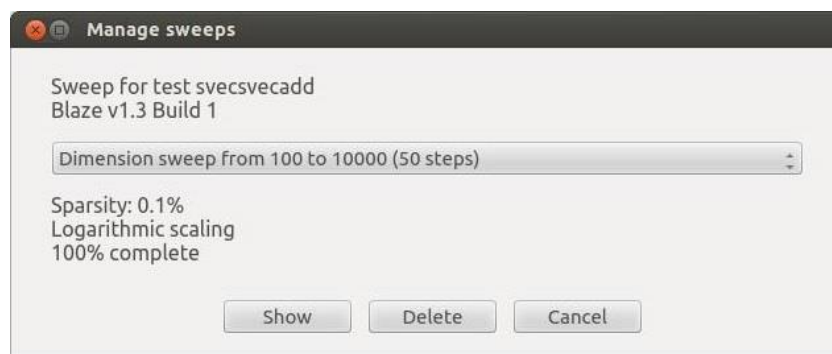
Sweep módban (**11. ábra**) a legördülő listák segítségével bármelyik könyvtárverzió bármelyik buildje kiválasztható. A "New sweep" gomb segítségével a kiválasztott builddel és tesztel egy új tesztorozat indítható, amelyet egy felugró párbeszédablakban lehet beállítani. Beállítás után a program automatikusan elkezd a sweep futtatását, a gomb feliratát "Cancel sweep"-re változtatva, továbbá megjelenítjük a tesztorozat pillanatnyi elkészültségét jelezve, százalékban. Ez a tesztorozat bármikor megszakítható, és később a megszakított tesztől újratekzdve folytatható, így ha szükségünk van a számítógép kapacitására, szüneteltethetjük a számításokat.

Ha a kiválasztott párosítással már indult sweep korábban, akkor a "Manage sweeps" gombbal megjelenített párbeszédablak (**12. ábra**) segítségével kiválaszthatjuk azt. Befejezett tesztorozat esetén megjeleníthetjük, ellenkező esetben pedig folytathatjuk a futtatást. Ugyanezen párbeszédablakon van lehetőség sweepeek törlésére is.



11. ábra. A BlazemarkAnalyser Analyse fül alatti felülete sweep módban

A sweepepek megjelenítése során az összehasonlító módhoz hasonlóan kerül megjelenítésre a függőleges tengely skálázása és a jelmagyarázat (sweepepek esetében a különböző színek a különböző buildek eredményeit jelölik), viszont a program oszlopdiagram helyett grafiont rajzol, amire a sweep fokozatos paraméter-változtatása miatt van lehetőség. A vízszintes tengely a változtatott paraméter értékét jeleníti meg, amelynek növekvése a megjelenített sweepepek beállításaitól függően lehet lineáris, vagy logaritmikus. Egyszerre csak ugyanazon skálázási beállításokkal és paraméterekkel rendelkező sweepepeket lehet megjeleníteni.



12. ábra. A BlazemarkAnalyser sweepepek menedzselésére szolgáló párbeszédablaka

5. Mérési eredmények

Ezen alfejezetben a BlazemarkAnalyser segítségével mért eredményeinket prezentáljuk. Ismertetjük a mérések körülményeit, kitérünk a ritkás adatszerkezeti implementációk elemzésére, és a csaknem kétszáz teszteredmény vizsgálata során levont következtetéseinkre. Az eredményeket szemléltető összes ábra a BlazemarkAnalyser grafikus megjelenítőjének segítségével készült.

5.1. *Tesztelési körülmények*

A tesztek egy Intel Core i5-3230M CPU-n végeztük, mely órajelét 2.6 GHz-en fixáltuk. Ez a processzor rendelkezik az AVX (Advanced Vector Extensions) utasításkészlettel, amely segítségével jelentősen gyorsabb vektorműveleti teljesítmény érhető el. A CPU képes az órajelének automatikus növelésére nagy terhelés esetében, de a megbízható eredmények mérése érdekében ezt letiltottuk. Az adatszerkezetek elemei minden mérésünkönél dupla pontosságú lebegőpontos változók voltak.

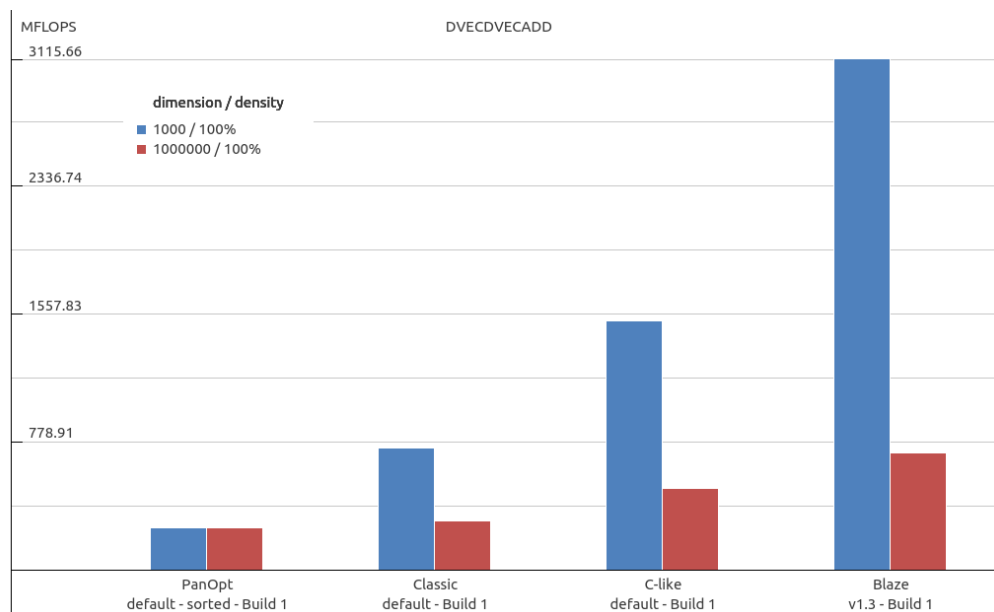
Fordítóprogramként a GNU GCC 4.8.1-es verzióját használtuk, az összes build esetében az alábbi fordítóparamétereket alkalmazva:

```
-Wall -Wextra -Werror -Wshadow -Woverloaded-virtual -ansi -O2 -std=c++11 -mavx  
-DNDEBUG -fpermissive
```

5.2. *Teszteredmények és következtetések*

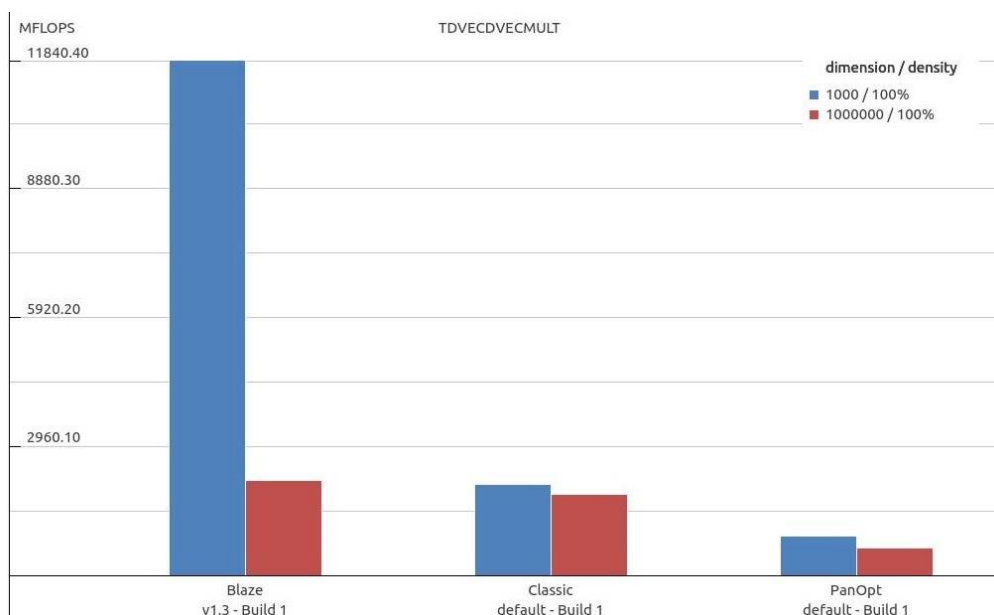
Bár a PanOpt jelenlegi verziójának lineáris algebrai osztályai folyamatos fejlesztés alatt állnak, a legtöbb vektor-vektor és mátrix-vektor művelet összemérhető hatékonyságú az AVX használatára is képes leggyorsabb könyvtár, a Blaze implementációjával, azaz legfeljebb két nagyságrenddel tér el attól [10].

A PanOpt *Vector* és *Matrix* osztályainak műveletei lassabbak a hagyományos, kis adatszerkezetek által implementált műveleteknél, mivel sok, a módosított szimplex módszer adatszerkezeteiként fontos tulajdonsággal bírnak. Ezen adatszerkezeteknek csak másodlagos célja az alapvető lineáris algebrai műveletek gyors végrehajtása, mivel elsődlegesen a PanOpt által követelményeinek kell megfelelniük. Az egyik ilyen lassító tényező az additív műveletek során alkalmazott *stableAdd* stabil összeadó művelet, amelyet a numerikus problémák ismertetése során már bevezettünk.. Ha az eredmény átlépi a toleranciát, és korrigálásra nincs szükség, a programnak akkor is meg kell vizsgálnia a szükséges feltételeket, így az egyszerű összeadás is jóval több időt vesz igénybe, mint a hagyományos módszerekkel.



13. ábra. Az 1000 és 1000000 elemű sűrű vektor–sűrű vektor összeadás implementációk összehasonlítása

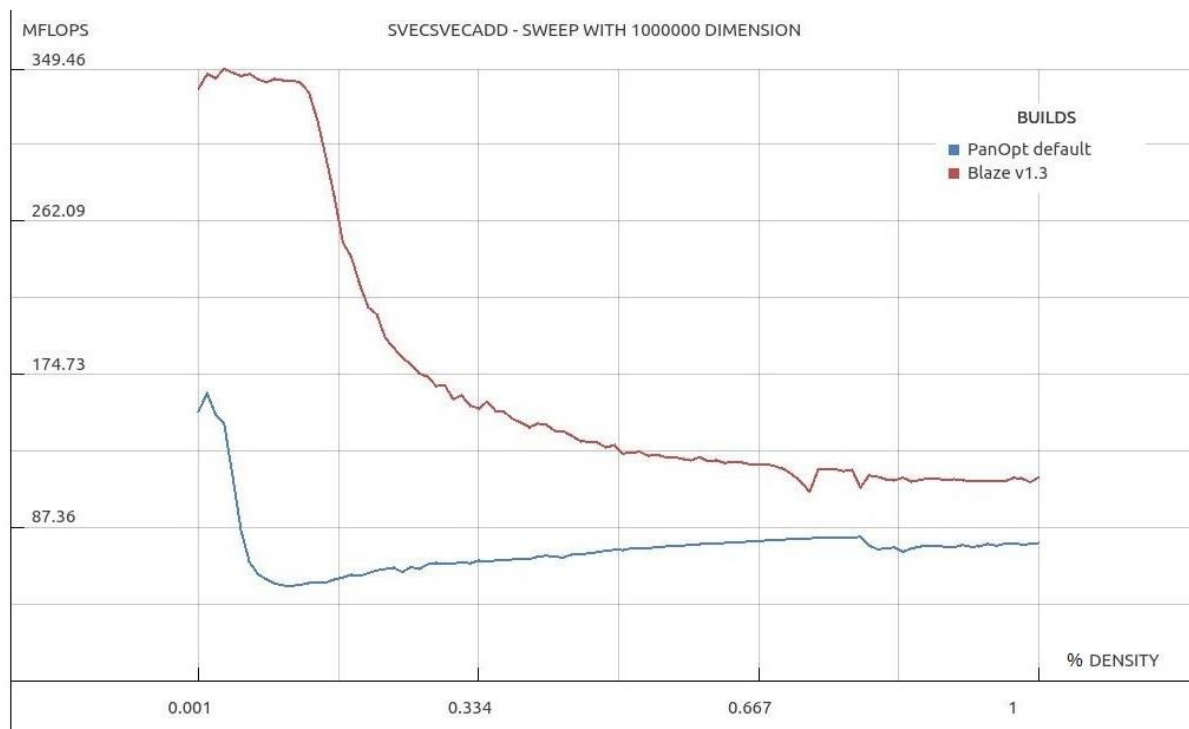
Megfigyeltük, hogy az egyszerű alpműveletek (például a sűrű vektorok összeadása) hatékonysága erősen függ az adatszerkezet bonyolultságától. Mivel egy ilyen művelet esetén a műveletnek kevés olyan sajátossága van, amit fel lehetne használni a végrehajtás felgyorsítása érdekében, arra a következtetésre jutottunk, hogy a hatékonyságot az $=$ operátor meghívásakor (a művelet tesztelése során a $c = a + b$ kifejezést vizsgáltuk) meghívódó konstruktor végrehajtási ideje befolyásolja. Az $=$ operátort nem konstruktor meghívásával használó Blaze könyvtár ezen művelet végrehajtása során jóval hatékonyabb, mint a többi könyvtár, amelyek közül a legegyszerűbb implementáció a leggyorsabb. (13. ábra)



14. ábra. Az 1000 és 1000000 elemű sűrű vektor–sűrű vektor skalárszorítás implementációk összehasonlítása

Ezt a feltételezést megerősíti az a tény, hogy minél több elemből áll egy vektor, annál kevesebb az eltérés a különböző könyvtárak eredményei között. Ilyenkor a teszt kevesebb iterációból áll, így kevesebbszer hívódik meg a konstruktor. Igaz továbbá, hogy olyan sűrű művelet esetében, ahol az eredmény skalár, a teljesítménybeli különbségek két nagyságrendről egyre csökkennek. **(14. ábra)** Ezen művelet esetében a különbségeket nagy valószínűséggel a toleranciafeltételes műveletvégzés okozza, illetve a Blaze által implementált “Smart Expression Template”-ek használata. [5] Tesztjeink alapján a Blaze vektor műveleteinek hatékonysága 1000 elemű adatszerkezetek esetében a legnagyobb, míg a PanOpt könyvtáré nem rendelkezik hirtelen kimagasló csúcserővel, hanem egyenletes teljesítményt nyújt.

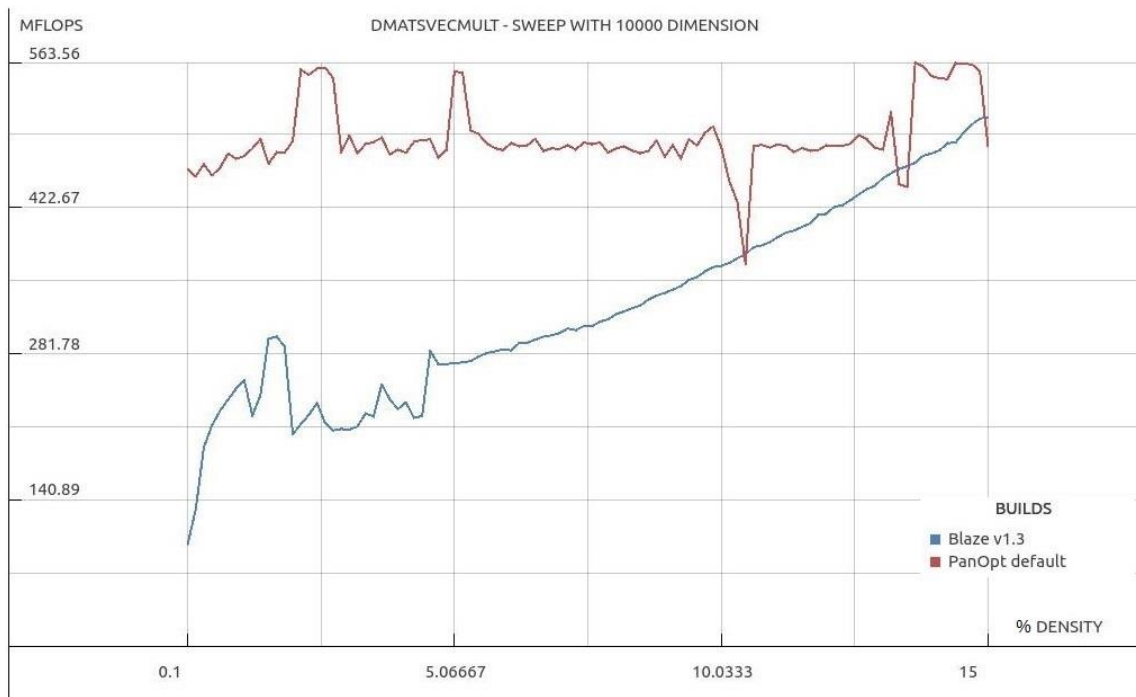
A ritkasságot kihasználó műveletek esetében, nagy adatszerkezeteknél a PanOpt könyvtár műveleti sebessége már átlagosan egy nagyságrendbe esik a Blaze adatszerkezetével. A konstruktor miatt létrejövő lassulás az adatszerkezet mérete miatt nem játszik nagy szerepet a művelet elvégzésének teljesítményében. Megfigyelhető, hogy egy bizonyos feltöltöttség alatt a teljesítmény hirtelen megugrik. **(15. ábra)** Ennek oka valószínűleg a ritkás tárolásmóddal járó bonyolultabb adatszerkezet előnyeinek (például csak a nem-nulla elemeken való gyors iterációra való lehetőségnek) előtérbe kerülése, valamint a cache memória gyorsító hatása. Ez a ritkasság-küszöb a PanOpt esetében körülbelül 0,1%, a Blaze esetében pedig 0.2%.



15. ábra. Az 1000000 elemű ritkás vektor-ritkás vektor összeadás gyorsasága a ritkasság függvényében

A PanOpt Matrix osztályának egyedülálló tulajdonsága, hogy sorfolytonosan és oszlopfolytonosan is el van tárolva. Ez azon műveletek során, ahol szükséges a mátrix

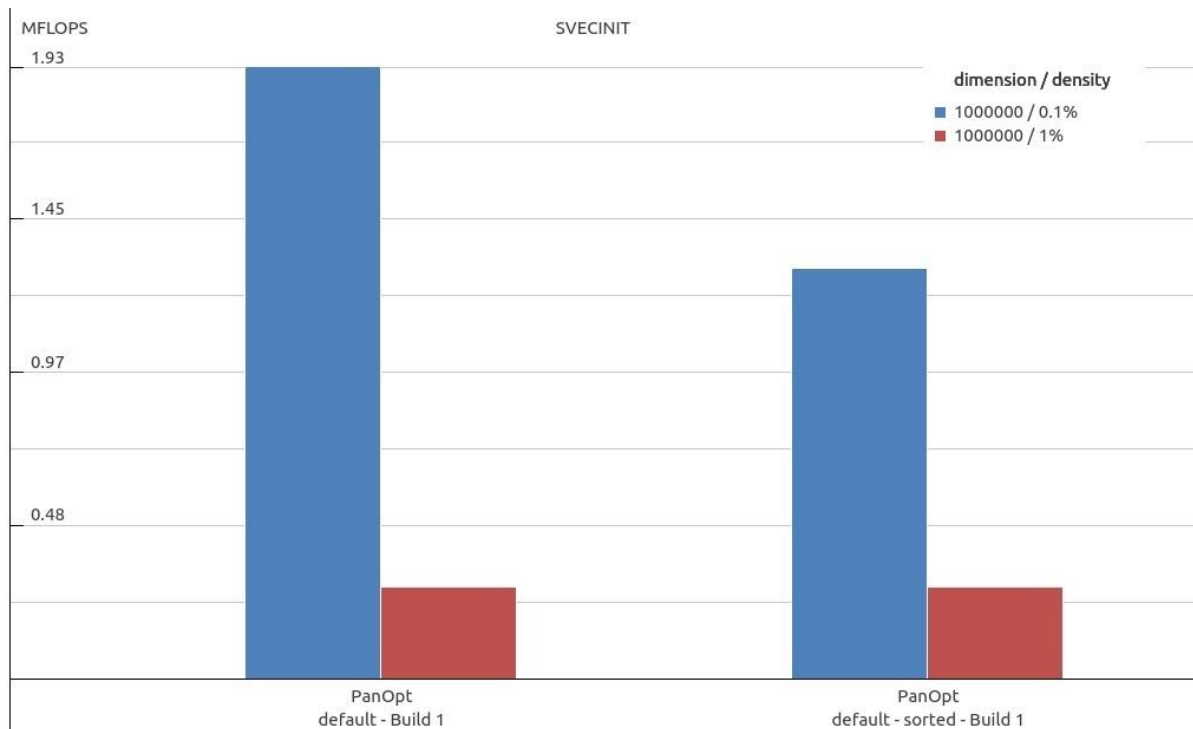
transzponálása kiemelkedően hatékony megoldás. A sűrű mátrix-ritkás vektor szorzás műveleténél ez nagyon reprezentatívan megjelenik, (16. ábra) ezen kívül látható a vektorok konstruktorai által okozott lassulás is. A műveleti sebesség konzisztenciájából következtethető, hogy ennek a műveletnek az implementációja jól kihasználja a benne szereplő vektor ritkasságát, mivel kevesebb nem-nulla elemet tartalmazó vektorral arányosan kevesebb idő alatt szorozza meg a mátrixot, így a műveleti sebesség nem változik a nem-nulla elemek számának változtatásával.



16. ábra. Az 10000 elemű sűrű mátrix-ritkás vektor szorzás gyorsasága a ritkasság függvényében

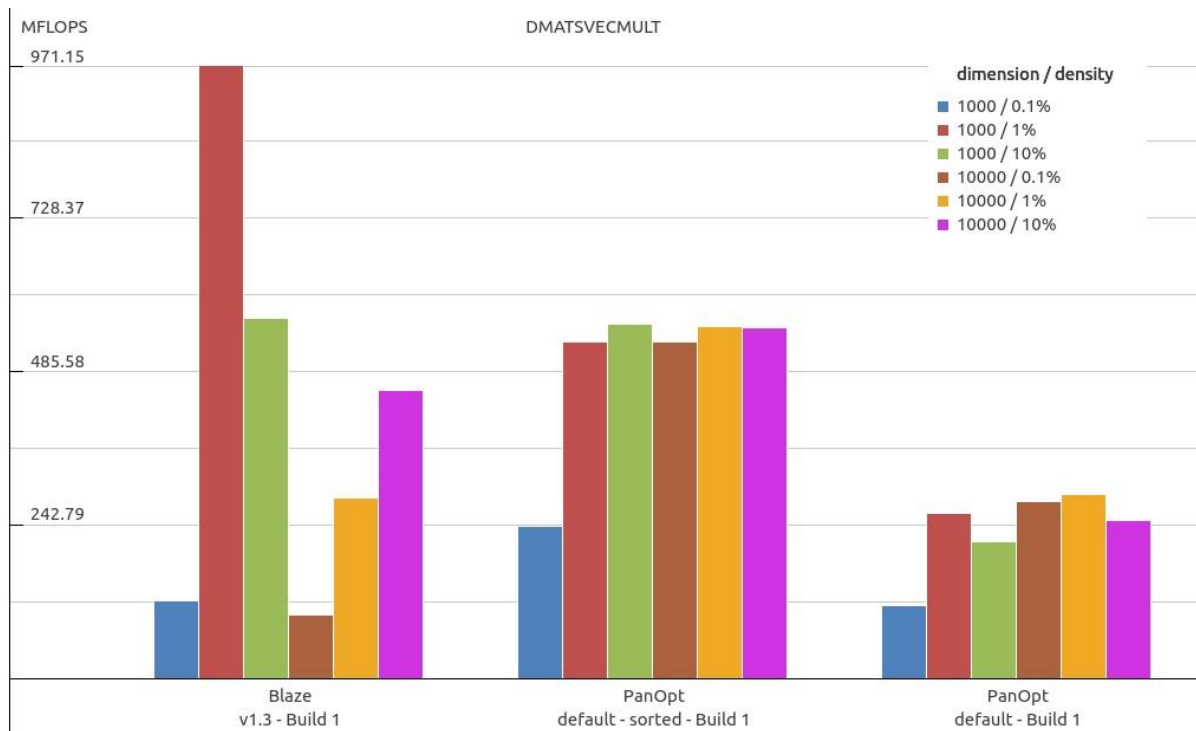
5.3. A teljesítményt befolyásoló paraméterek

A PanOpt lineáris algebrai osztályainak elemzését abból a célból kezdtük el, hogy az eredmények segítségével a megoldó szempontjából hatékonyabban működő adatszerkezeteket fejlesszünk. Az implementációban számos olyan tényező szerepel, amely bizonyos műveleteknél hatékonyabb, más műveleteknél viszont kevésbé hatékony működést eredményez. Ezeknek a tényezőknek a megfelelő beállítása jelentősen gyorsíthatja a megoldó működését. Ezen alponthan kiemelünk néhány ilyen tényezőt, és bemutatjuk az azokkal kapcsolatos mérési eredményeinket.



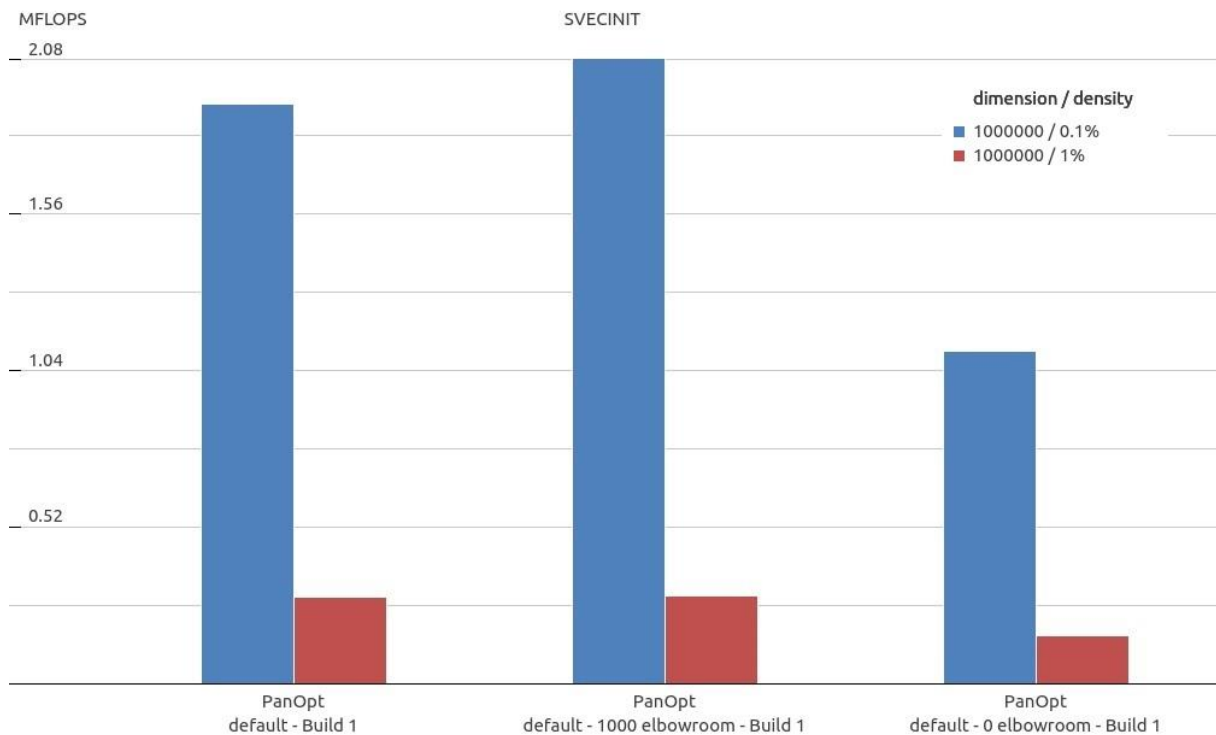
17. ábra. Az 1000000 elemű ritkás vektorok inicializálása rendezetlen és rendezett indexek esetén

Az első ilyen a ritkás vektorok index szerinti rendezettsége. Ennek a rendezettségnek a megtartása többszámításokat igényel minden olyan művelet során, amely új nem-nulla elemeket generálhat a vektorban, azonban néhány művelet esetében jelentős gyorsulást eredményez. A vektor feltöltése után is rendezni kell az elemeket, amely lassabb inicializációt eredményez. (17. ábra). Az elemek rendezettsége az alapvető vektorműveletek közül a vektorok skaláris szorzatának számítása során használható ki a legjobban. A skaláris szorzásra visszavezethető, gyakran alkalmazott művelet a mátrix-vektor szorzás. Ezen művelet esetében a ritkás vektor rendezettségének megtartásával a rendezetlen vektorok használatához képest körülbelül kétszeres gyorsaságot érhetünk el. (18. ábra) Mivel a rendezettség megtartása nagyon sok műveletnél igényelne többszámításokat, hatékonyabb a vektorokat rendezetlenül tárolni, azonban a több skaláris szorzást felhasználó műveletek elvégzése előtt rendezve jobb teljesítmény érhető el. Fontos megjegyezni továbbá, hogy konstans vektorok esetében érdemes rendezni azokat, ez tipikusan előfordul LP feladatok megoldása során.



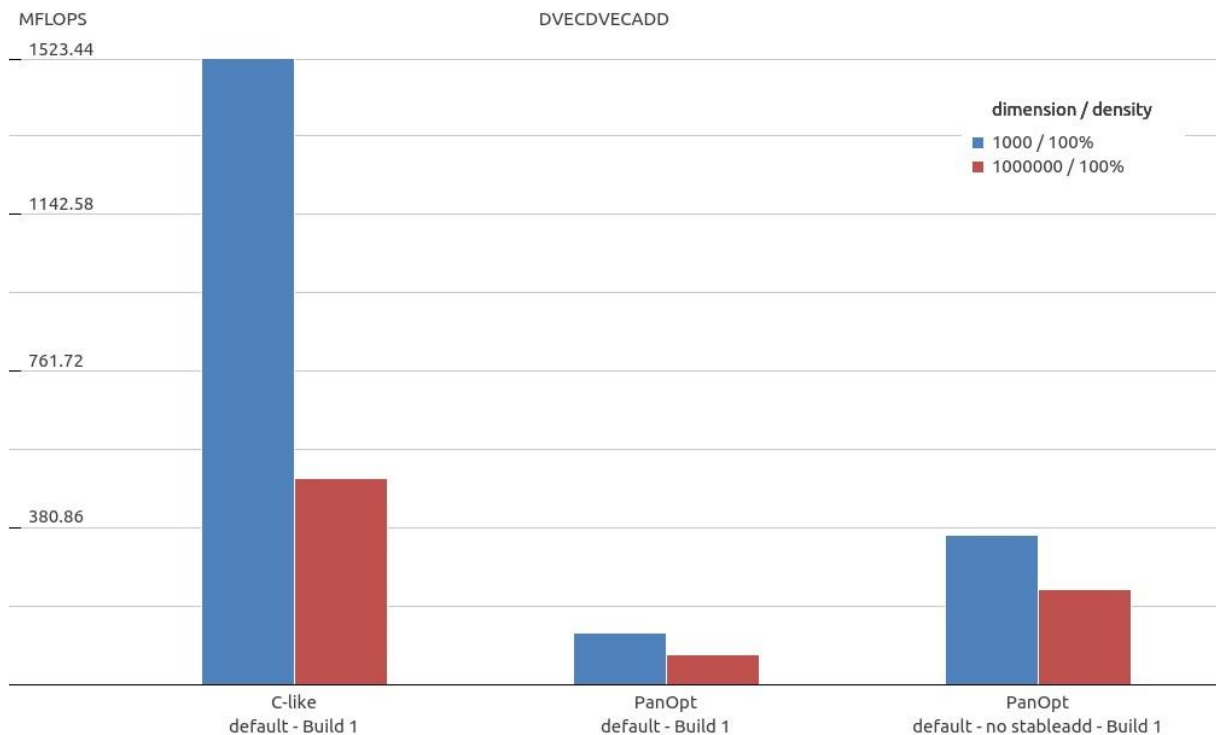
18. ábra. A sűrű mátrix-ritkás vektor szorzás gyorsulása a vektor elemeinek rendezésével

A ritkás vektorok esetében lehetőség van a vektor méreténél több memóriát lefoglalni abból a célból, hogy a vektor méretét esetleges későbbi műveletek során újabb memórafoglalás nélkül lehessen növelni. A PanOpt Vector osztályának esetében ezeknek a plusz lefoglalt elemeknek a számát az *ELBOWROOM* nevű paraméter adja meg. A paraméter nullára állítása esetén a ritkás vektor adat része által elfoglalt hely a memóriában a vektor nem-nulla elemeinek számától függ, az általa elfoglalt memória egészét kihasználja. Ilyenkor minden újonnan keletkező nem-nulla elem beírásakor újra kell foglalni a vektor számára fenntartott memóriát. Az *ELBOWROOM* alapértelmezett értéke jelenleg 5. Nagy érték esetén nem érünk el jelentősebb teljesítménycsökkentést (19. ábra), viszont a memóriagény jelentősen megnő.



19. ábra. A PanOpt ritkás vektor inicializációjának teljesítménye különböző ELBOWROOM értékek esetén

A PanOpt lineáris algebrai könyvtárai a többi nyílt forráskódú könyvtárral ellentétben stabil összeadást használ a numerikus hibák kiküszöbölése céljából. Ennek a módszernek a használata mindenképpen szükséges, mivel a PanOpt által implementált LP megoldó működése során az egyik elsődleges szempont a nagy numerikus pontosság. A stabil összeadó jelenlegi verziója a mérések alapján a műveletek teljesítményét több, mint háromszorosan csökkenti (**20. ábra**), ezért folyamatban van egy új, alacsony szintű programozással megvalósított, hatékonyabb stabil összeadó fejlesztése.



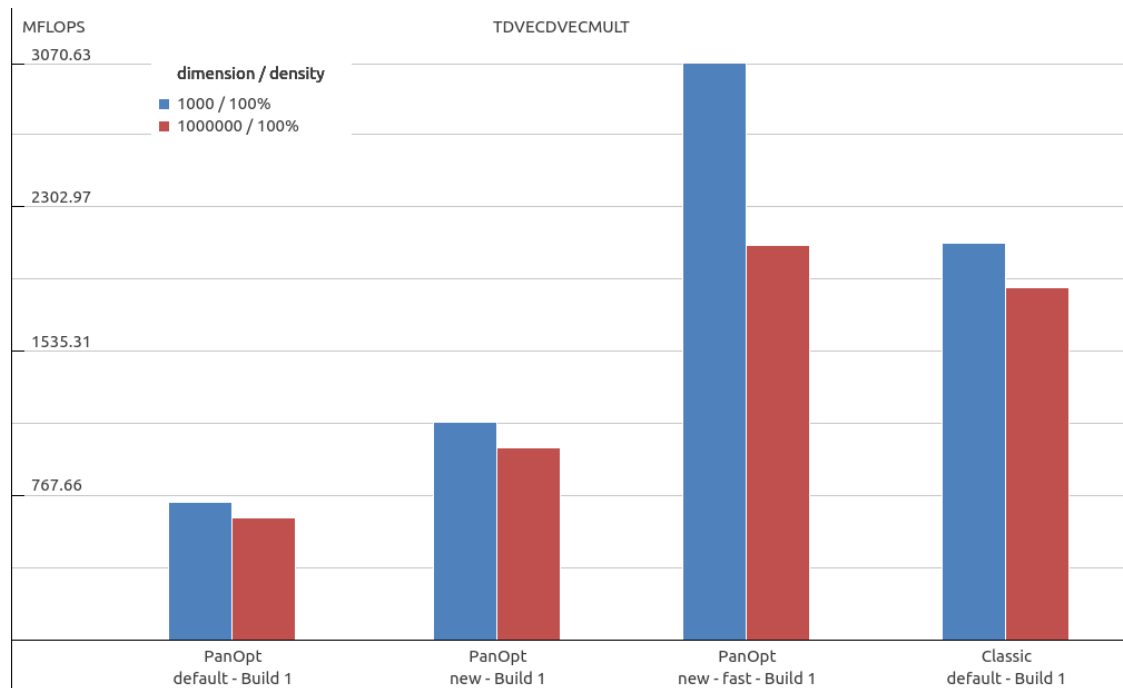
20. ábra. A stabil összeadó alkalmazásának hatása a műveleti teljesítményre

5.4. A PanOpt új adatszerkezetek

Mérési eredményeink alapján a PanOpt Vector osztálya által implementált tárolási módok között váltani képes adatszerkezet kevésbé hatékony, mint a ritkás és sűrű vektor-reprezentációk külön osztályban való kezelése. Mivel a megoldó szempontjából a felhasznált adatszerkezetek hatékonysága kulcsfontosságú, ezért a korábbi adatszerkezetek újratervezésével, hatékonyabb adatszerkezetek implementálása vált szükségessé, melyek fejlesztése során felhasználtuk eddigi eredményeinket. Az új adatszerkezetek továbbra is fejlesztés alatt állnak, azonban a megoldó szempontjából legfontosabb műveletek, a skaláris szorzás és a daxpy ($a = a + \lambda b$) szorzás már működőképesek.

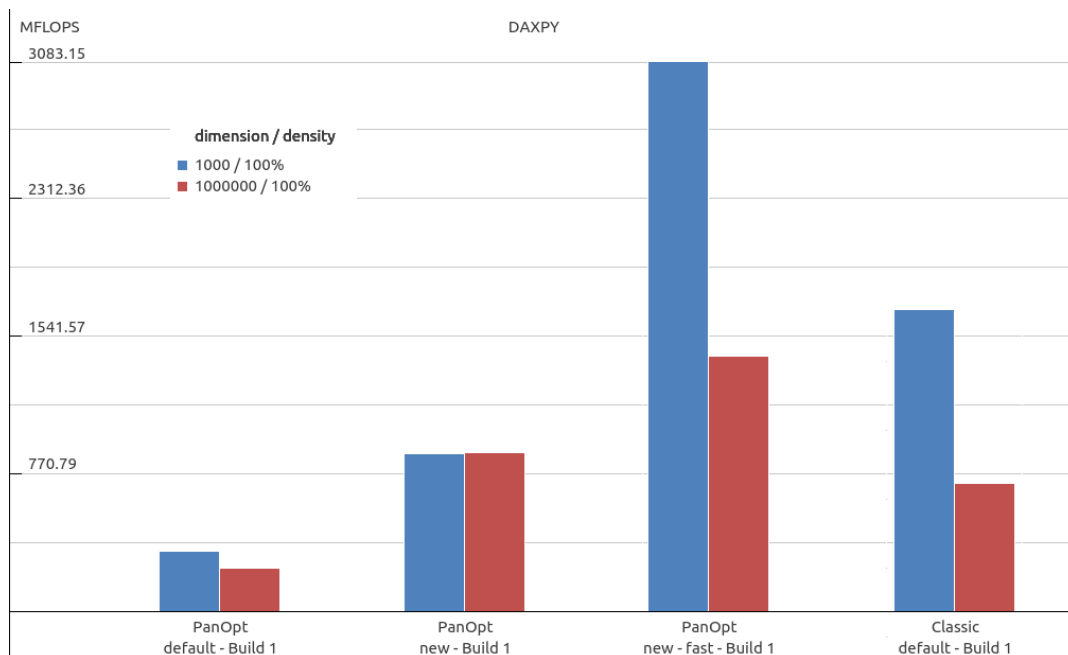
5.4.1. A PanOpt DenseVector osztálya

A PanOpt új, kizárólag sűrű vektor reprezentációra alkalmas DenseVector osztálya azon vektorok esetében a leghatékonyabb, amelyeknek index alapján való gyors elérése és módosítása fontosabb, mint a nem-nulla elemek hatékony bejárása.



21. ábra. A PanOpt DenseVector osztályának skaláris szorzás műveleti teljesítménye

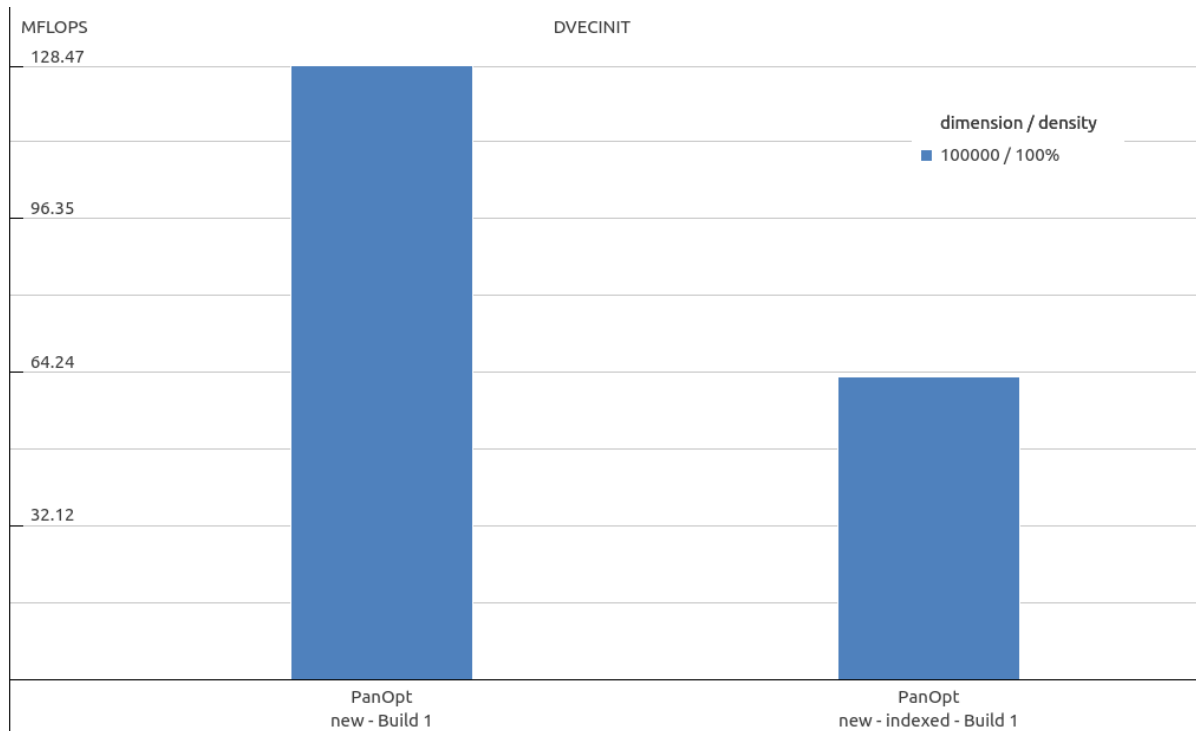
A DenseVector osztály műveletei során nem végez felesleges számításokat, amelyek esetleges többletműveletet okozhatnának, nem tárolja a vektorban lévő nem-nulla elemek számát, és az alapvető műveletek elvégzése közbeni numerikus pontosság megtartásáért felelős toleranciák vizsgálata is állítható. Az eredményekből látható, hogy az új, sűrű vektor reprezentációt implementáló adatszerkezetünk jelentősen hatékonyabb mind a réginél, mind pedig a hagyományos C++ `std::vector`-nál a vizsgált műveletek esetében. (21. ábra és 22. ábra)



22. ábra. A PanOpt DenseVector osztályának daxpy szorzat műveleti teljesítménye

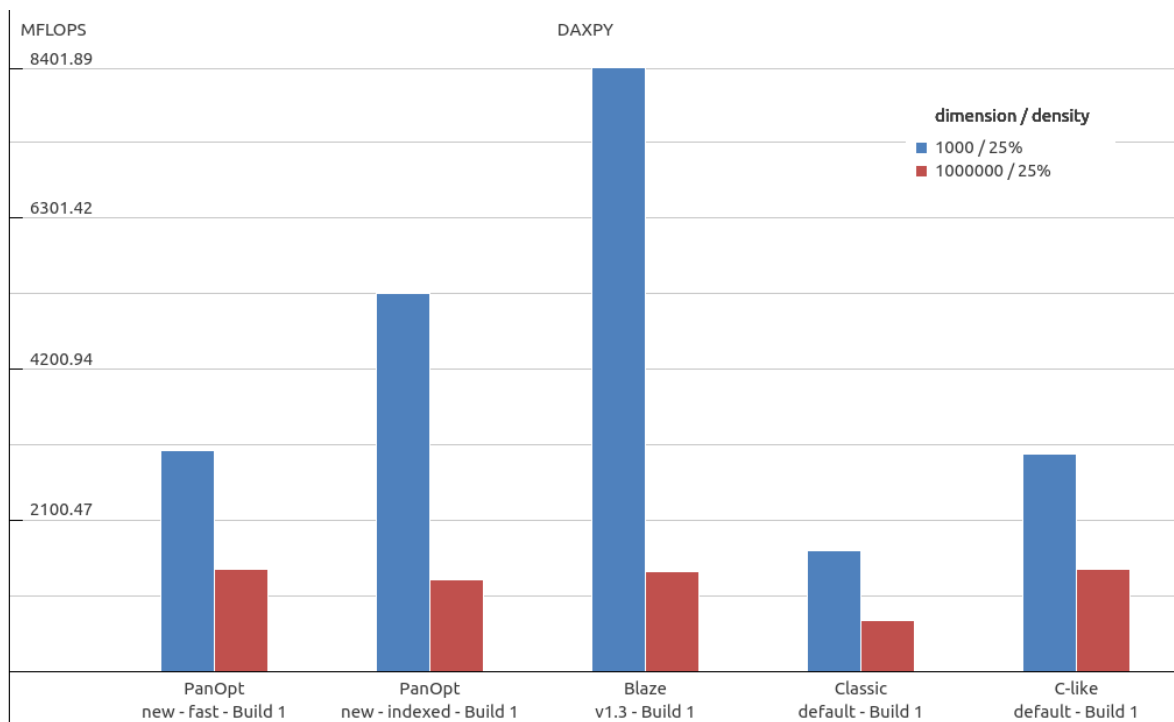
5.4.2. A PanOpt IndexedDenseVector osztálya

Az IndexedDenseVector osztály sűrű vektor reprezentációra alkalmas adatszerkezet, amely több memória árán gyorsabb műveletvégzést tesz lehetővé. Az adatszerkezet két redundáns vektort tárol az adatvektoron kívül. Az első a ritkás tárolásmódhoz hasonlóan a nem-nulla elemek indexeit tárolja, ennek alkalmazásával a nem-nulla elemek iterálása gyorsan elvégezhető. Ez növeli a legtöbb művelet sebességét. A második plusz adatszerkezet az összes nem-nulla elem tárolt indexére mutató pointereket tárolja, amelyek segítségével egy nem-nulla elem nullázódása esetén közvetlenül elérhető a törlendő index.



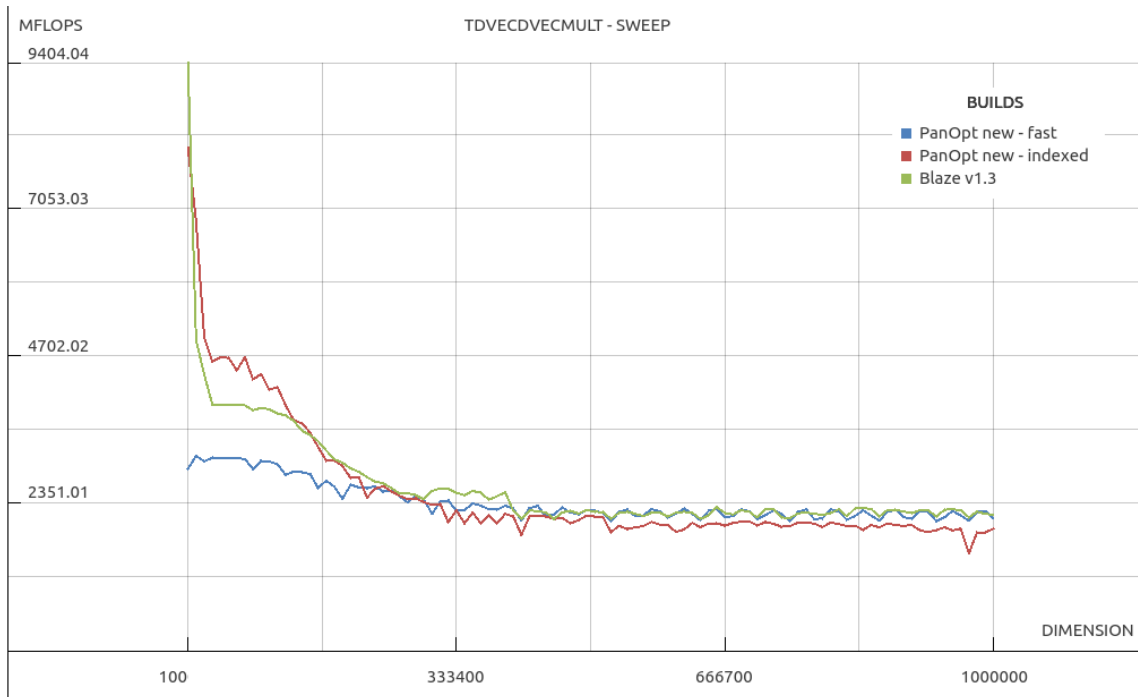
23. ábra. A PanOpt IndexedDenseVector osztályának inicializálási ideje a DenseVectorhoz képest

Ezen tulajdonságainak köszönhetően, bár a vektor inicializálása és elemeinek módosítása a redundáns adattárolás miatt több számítást igényel (**23. ábra**), műveleteinek elvégzése közben egyesíti a ritkás és sűrű vektor-reprezentációk előnyeit, így kevés nem-nulla elemmel rendelkező vektorok esetében jóval hatékonyabb teljesítményt nyújt, mint az egyszerű sűrű vektorok. (**24. ábra**) Ezt a vektortípust azon vektorok esetében célszerű alkalmazni, amelyekkel gyakran kell műveleteket végezni, és módosításukra is gyakran van szükség.



24. ábra. A PanOpt IndexedDenseVector osztályának daxpy szorzás műveleti teljesítménye 25%-os feltöltöttség esetén

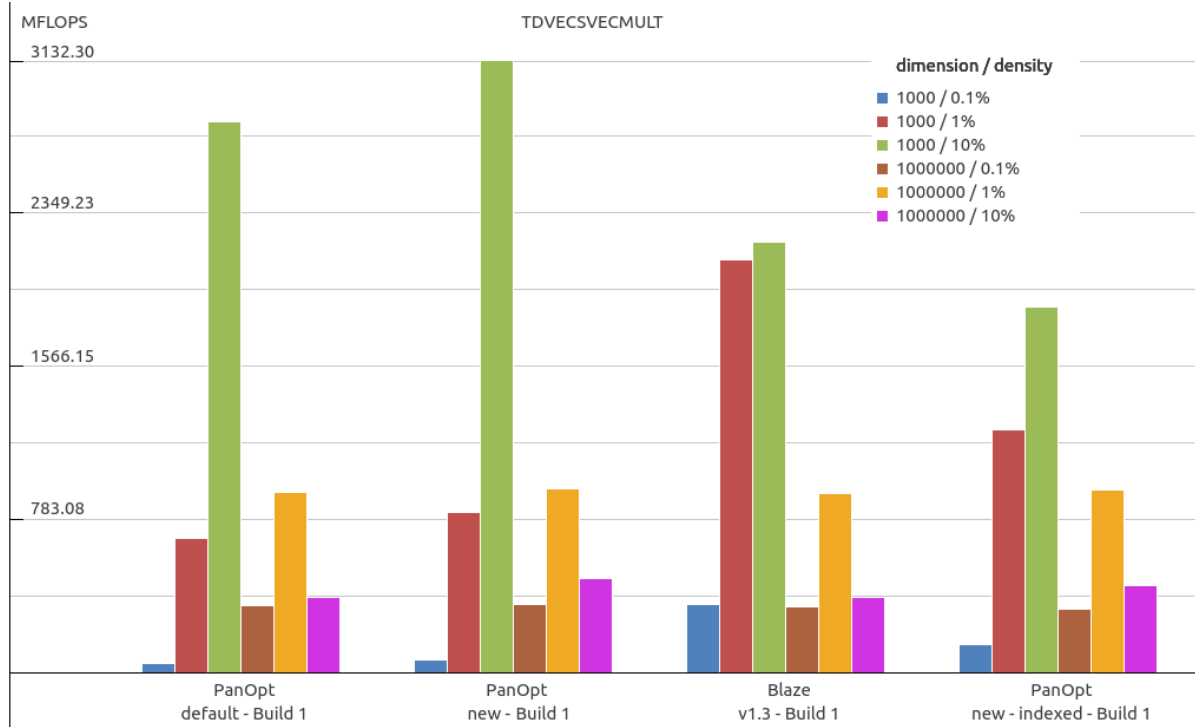
Mivel, amíg a vektor nem módosul, a redundáns adattárolás nem okoz számításigénybeli növekedést, sok nem-nulla elemmel rendelkező vektorok esetében az IndexedDenseVector egyenletes, a hagyományos sűrű vektor implementációkhoz hasonló teljesítményt nyújt. (25. ábra)



25. ábra. A PanOpt IndexedDenseVector osztályának skaláris szorzás műveleti teljesítménye 25%-os feltöltöttség esetén

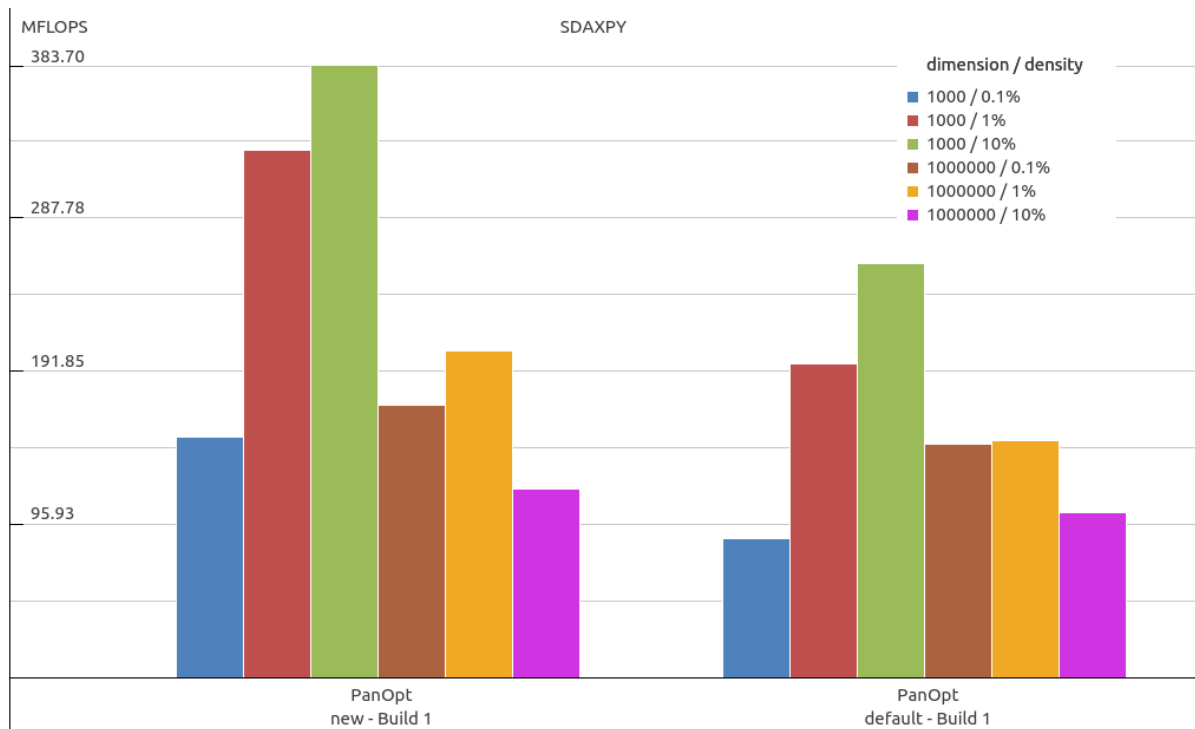
5.4.3. A PanOpt SparseVector osztálya

A SparseVector osztály a ritkás vektorok reprezentálásának céljából készült és elsődleges célja a megoldandó LP probléma mátrixát alkotó vektorok hatékony megvalósítása. Ezek a vektorok nem módosulnak a probléma megoldása során, így indexeik rendezve tárolhatók.



26. ábra. A PanOpt új vektor osztályainak sűrű vektor-ritkás vektor skaláris szorzás műveleti teljesítménye

Bár az előzőleg alkalmazott Vector osztály implementációja is elsődlegesen a nagy ritkasságú vektorok skaláris és daxpy szorzás műveleteinek minél gyorsabb elvégzése céljából készült, a SparseVector osztály kisebb redundanciataralma miatt hatékonyabb műveletvégzésre képes. (26. ábra és 27. ábra)



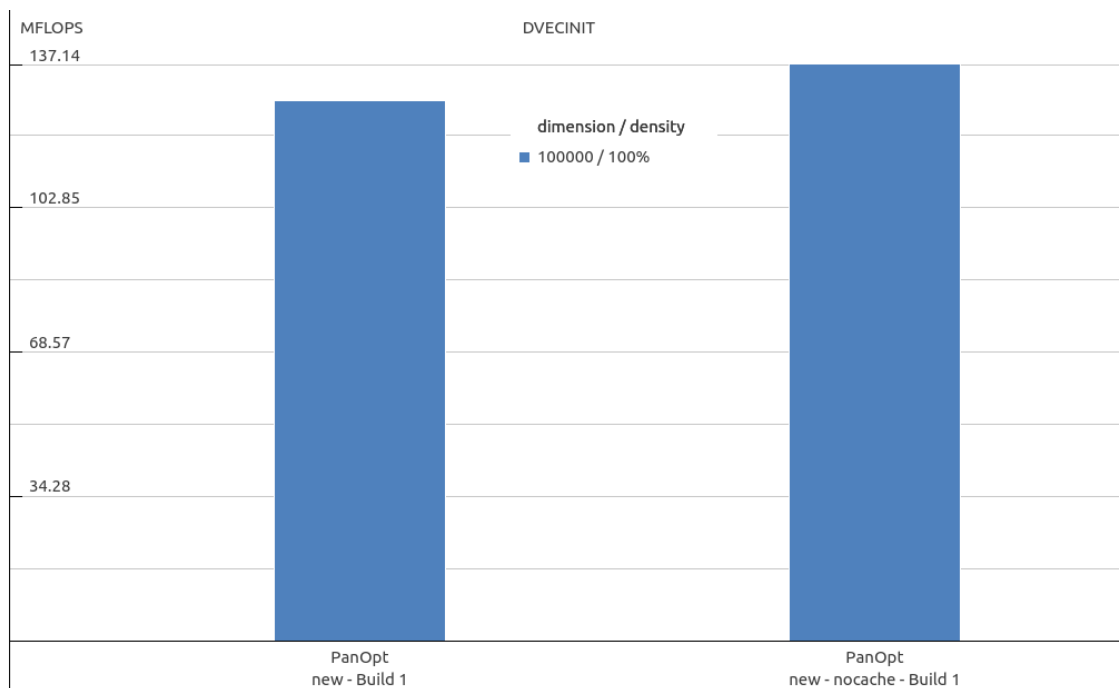
27. ábra. A PanOpt új vektor osztályainak sűrű vektor-ritkás vektor daxpy szorzás műveleti teljesítménye

5.5. Fejlett implementációs tényezők

Az adatszerkezetek teljesítményét nem csak az azokban szereplő redundancia és adattárolási mód befolyásolhatja, hanem a processzor közeli műveleti tényezőket befolyásoló implementációs technikák is. Az eredeti programkód alacsony szintű kódra konvertálását a fordító végzi el, amely beállításoktól függően optimalizációt is végez. Bizonyos esetekben a műveletek hatékonyabban valósíthatók meg ezen tényezők implementálás során való figyelembevételével.

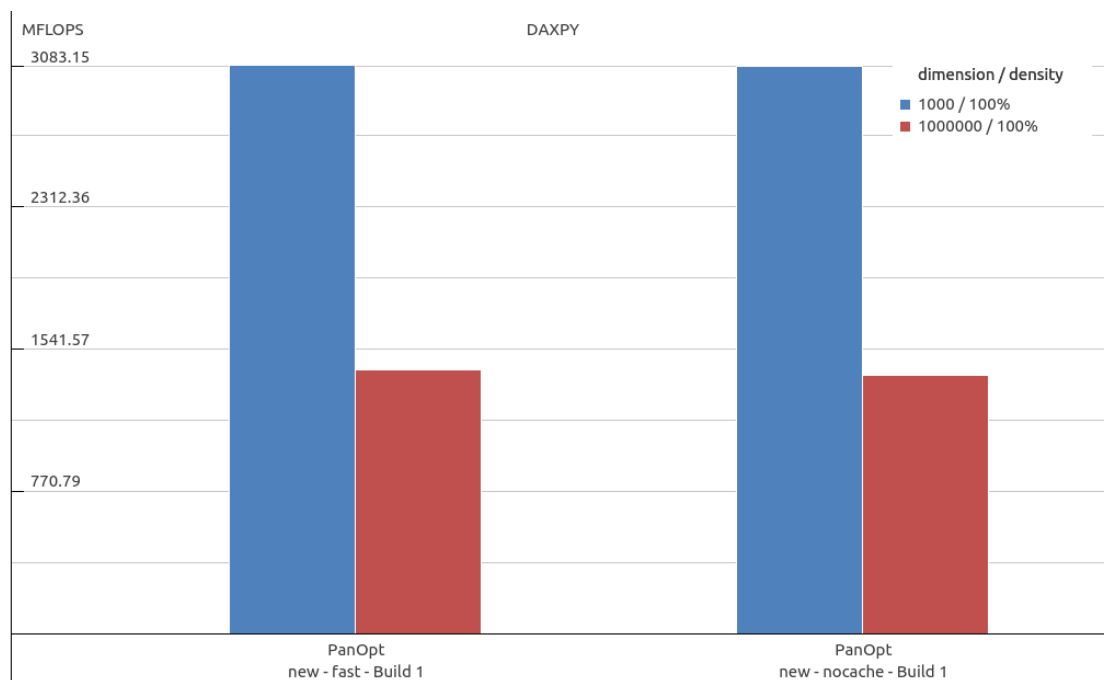
5.5.1. Blokkhatárra igazítás

A műveletek végrehajtása során a processzornak be kell töltenie a szükséges adatokat a gyorsítótárába. Ezek az adatok a memória és a processzor közti buszon töltődnek be, azonban ez a busz csak a memória meghatározott blokkjait képes átadni. Így, ha a egy, a végrehajtandó művelethez szükséges értékek ilyen blokkhatárra esik, a processzornak két blokkot kell betöltenie egy helyett.



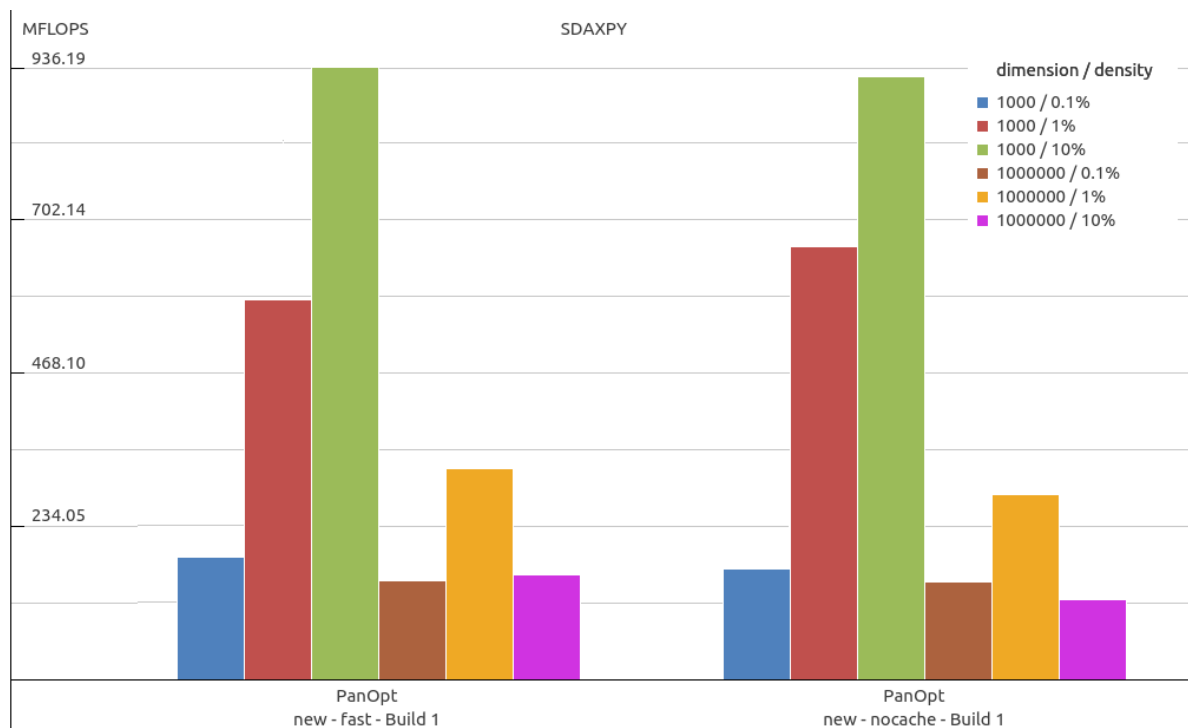
28. ábra. A PanOpt DenseVector osztályának inicializálási ideje az értékek blokkhatárra igazításával és anélkül

Az adatszerkezet megvalósítása során van lehetőség arra, hogy működés során a tárolt értékek méreténél több memória kerüljön lefoglalásra, így az értékek első bitje a blokkhatár utáni memóriabitre essen. Ezzel a módszerrel a több memóriafoglalás és számítási művelet miatt az inicializálási idő megnő (28. ábra), azonban a kevesebb cache művelet miatt a műveletek gyorsabban hajthatók végre.



29. ábra. A PanOpt DenseVector osztályának daxpy szorzás műveleti teljesítménye blokkhatárra igazítással és anélkül

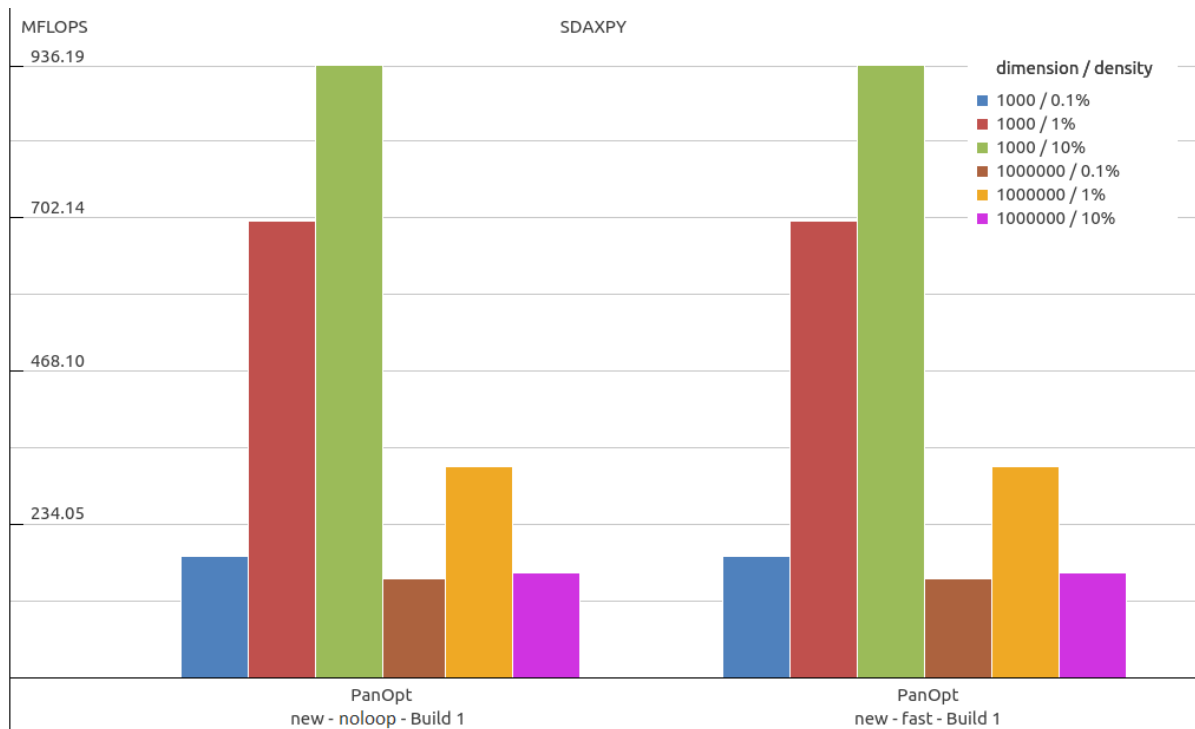
A mérési eredmények alapján az értékek blokkhatárra igazítása során az inicializáció ideje a vektor feltöltöttségétől függően 5-10%-al megnő, azonban a műveleti teljesítmény sűrű vektorok esetében 2-3%-al (29. ábra), ritkás vektorok esetében pedig 10-15%-al (30. ábra) javul.



30. ábra. A PanOpt SparseVector osztályának daxpy szorzás műveleti teljesítménye blokkhatárra igazítással és anélkül

5.5.2. Loop unrolling

Az implementáció során használt ciklusok magjának minden végrehajtása után a processzornak egy feltételvizsgálatot kell végrehajtania, amely kiértékelése után dől el, hogy a ciklus folytatódik-e. Bár a fordító is végez ilyen téren optimalizálást, bizonyos esetekben, amikor előre látható, hogy a ciklusfeltétel teljesülni fog, célszerű a ciklusmagot egymás után többször végrehajtani, és így ritkábban végezni feltételvizsgálatot. Ennek a módszernek a hátránya a kód méretének növekedése, illetve bonyolult, sok utasításból álló ciklusmag esetén a processzornak többször kell lekérnie az utasításokat a memóriába, mivel az egész mag nem fér be az utasításmemória gyorsítótárába.

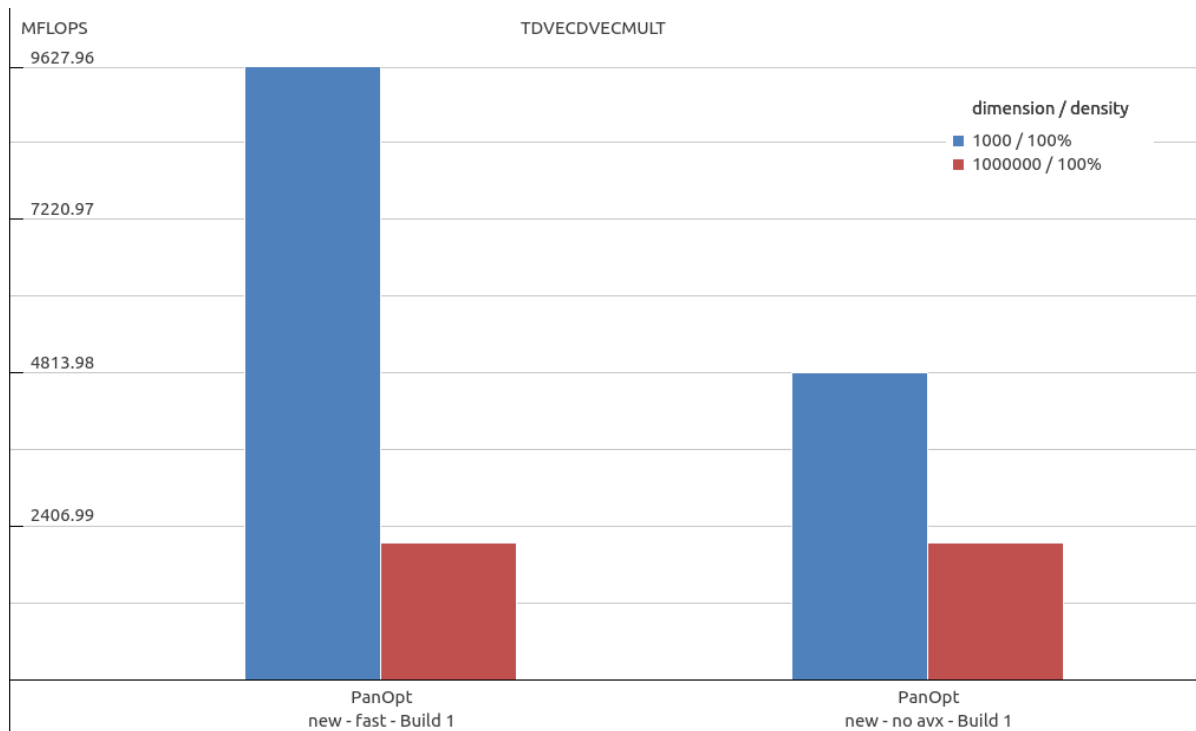


31. ábra. A PanOpt SparseVector osztályának daxpy szorzás műveleti teljesítménye loop unrolling alkalmazása esetén és anélkül

Mérési eredményeink alapján az általunk használt fordító által alkalmazott dinamikus loop unrolling (a ciklusfeltételként vizsgált változó értéke fordítási időben nem meghatározható) közel azonos eredményt ér el az általunk alkalmazotthoz képest.

5.5.3. A processzor utasításkészletének hatékony kihasználása

Bizonyos processzorok kiegészítő utasításkészleteikkel támogatják a vektorműveletek hatékonyabb végrehajtását több művelet egyidejű elvégzésével. Ilyen utasításkészletek például az SSE, AVX és AVX2. Ezek alkalmazása jelentősen felgyorsíthatja az adatszerkezetek műveleti teljesítményét. Mivel az SSE 64 bites, az AVX 128 bites, az AVX2 pedig 256 bites regisztereket használ, 64 bites rendszerek esetében az AVX kétszeresére, az AVX2 pedig négyszeresére növeli a teljesítmény elméleti maximumát [11].



32. ábra. Az AVX utasításkészlet hatása a PanOpt DenseVector osztályának skaláris szorzás műveletének teljesítményére

A PanOpt új vektor implementációi már kihasználják az AVX utasításkészlet által nyújtott előnyöket, amennyiben a processzor támogatja azt. Az eredmények alapján kis dimenziójú vektorok esetében érvényesül a különleges processzorműveletek eredményéül kapott kétszeres műveleti sebesség, nagy vektorok esetében azonban nem figyelhető meg nagy gyorsulás. (32. ábra) Ennek az oka valószínűleg az, hogy nagyon nagy elemszámú vektorok esetében a memóriaműveletek veszik el a művelet által igénybevert számítási idő legnagyobb részét.

6. Összefoglalás

Munkánk során azzal a célkitűzéssel végeztük kutatásainkat, hogy az Operációkutatási Kutatólaboratórium munkájának keretében fejlesztett lineáris programozási megoldó lineáris algebrai adatszerkezeteit elemezhessük, illetve összehasonlíthassuk egyéb hasonló célú adatszerkezetekkel. Mivel egy ilyen megoldó működése során fontos a gyors és hatékony működés, valamint ezek az adatszerkezetek az egész rendszer implementációjának alapjához tartoznak, a tesztjeink által összegyűjtött információ és annak rendszerezése kulcsfontosságú a fejlesztés szempontjából.

Dolgozatunkban áttekintést nyújtottunk a főbb lineáris algebrai műveletekről, a ritkás és sűrű adatszerkezetek közti különbségről, a ritkás műveletek implementációs kihívásairól és a ritkáság kihasználásának fontosságáról. Bemutattuk a legnépszerűbb nyílt forráskódú könyvtárakat, majd ismertettük, hogy a megoldóhoz fejlesztett lineáris algebrai osztályaink miként oldják meg a ritkás adatszerkezetek implementációjával kapcsolatban felmerülő problémákat. Áttekintést nyújtottunk a Blazemark keretrendszer működéséről, teljesítményelemzési lehetőségeiről és korlátairól.

Ismertettük a Blazemark rendszer kiterjesztését az általunk implementált adatszerkezetek tesztelésére. Bemutattuk a saját fejlesztésű, Blazemarkon alapuló elemző szoftverünket, a BlazemarkAnalysert, kitérve az ehhez tervezett eredmény-rendszerező adatbázisra, a Blazemark rendszert működtető motorra, valamint az eredmények áttekintését szolgáló grafikus felületekre.

A dolgozatunkban ismertettük az adatszerkezeteken futtatott tesztjeinket, illetve a már több mint 200 teszteredményt tartalmazó adatbázisunk adatainak vizsgálata során levont következtetéseinket. Mindezt szemléltető példákkal támasztottuk alá. Következtetéseink alapján a Pannon Optimizerhez új adatszerkezeteket terveztünk, majd azok implementációinak is elvégeztük hatékonysági vizsgálatát. Megállapítottuk, hogy az új adatszerkezetek jelentősen gyorsabbak a korábbiaknál, valamint további fejlesztési irányokat határoztunk meg.

Tesztjeink elvégzése hasznos információt szolgáltatott a lineáris algebrai adatszerkezetek fejlesztésével kapcsolatban, és tervezzük azok folyamatos monitorozását a további fejlesztések folyamán. Terveink között szerepel a tesztelő keretrendszert kiegészíteni az adatszerkezetek konstruktorának, illetve a megoldónk által implementált módosított szimplex módszerre jellemző összetett műveleteknek mérésére szolgáló tesztekkel.

Jövőbeni tervünk a BlazemarkAnalyser továbbfejlesztése tetszőleges könyvtár és művelet elemzésére, majd ennek publikálása, ami jelentősen megkönnyíti bármilyen C++ nyelvű lineáris algebrai könyvtár fejlesztését. Mindemellett szeretnénk szoftverünket profilozó felülettel is kiegészíteni, ami segítene az eredmények még hatékonyabb elemzésében.

Irodalomjegyzék

- [1] **I. Maros**, Essentials of computational linear algebra, Veszprém: Department of Computer Science, Faculty of Information Technology, University of Pannonia, 2009, pp. 3-21.
- [2] **I. Maros**, Computational Techniques of the Simplex Method, Springer, 2003.
- [3] **F. Hartung**, Bevezetés a numerikus analízisbe, Veszprém: Veszprémi Egyetemi Kiadó, 1999.
- [4] **B. Kempf**, "The Boost.Threads Library," *C/C++ Users Journal*, május 2002.
- [5] **K. Iglberger, G. Hager, J. Treibig and U. Rüde**, "High performance smart expression template math libraries," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on High Performance Computing*, Madrid, Spain, 2012.
- [6] **J. Cummings and T. Veldhuizen**, "The Blitz++ meta-template library," 2011. [Online]. Available: <http://blitz.sourceforge.net/>.
- [7] **Y. Renard and J. Pommier**, "Gmm++ Library," 2013. [Online]. Available: <http://download.gna.org/getfem/html/homepage/gmm/>.
- [8] **G. Guennebaud**, "Eigen: a c++ linear algebra library," in *Ecole Polytechnique*, 2013.
- [9] **SimuNova UG**, 2014. [Online]. Available: <http://www.simunova.com/mtl4>.
- [10] **K. Iglberger, G. Hager, J. Treibig and U. Rüde**, "Expression Templates Revisited: A Performance Analysis of Current Methodologies," *SIAM Journal on Scientific Computing*, vol. 34., no. 2., pp. C42-C69, 2012.
- [11] **P. Gepner, V. Gamayunov and D. L. Fraser**, "Early performance evaluation of AVX for HPC," *Procedia Computer Science*, vol. 4., pp. 452-460, 2011.

Köszönetnyilvánítás

Ez úton szeretnék köszönetet mondani mindazoknak, akik lehetővé tették, hogy ez a dolgozat elkészülhessen. Dr. Maros István professzor úrnak, az Operációkutatási Kutatólaboratórium vezetőjének, aki biztosította számomra a kutatócsoporthoz való csatlakozás lehetőségét, és végig támogatott munkám során. Tar Péter és Smidla József PhD hallgatóknak, akik témavezetőmként nem csak irányították munkámat és értékes szakmai tanácsokkal láttak el, de végig ösztönöztek és útmutatást nyújtottak kutatásaim során. Köszönöm továbbá tanárainnak, kollégáimnak és hallgatótársaimnak, hogy segítettek eredményeim elérésében.

„A publikáció az Európai Unió, Magyarország és az Európai Szociális Alap társfinanszírozása által biztosított forrásból a TÁMOP-4.2.2.C-11/1/KONV-2012-0004 azonosítójú "Nemzeti kutatóközpont fejlett infokommunikációs technológiák kidolgozására és piaci bevezetésére" című projekt támogatásával jött létre.”