

EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

Statikus videók elemzése mozgásdetektálás céljából párhuzamosítás segítségével

TDK dolgozat

Készítette: Botos Ádám
ELTE IK
Programtervező informatikus
MSc

Témavezető: Prof. Csetverikov Dmitrij
Egyetemi tanár
ELTE IK
Algoritmusok és
Alkalmazásaik Tanszék
MTA SZTAKI
Laborvezető

Témavezető: Kovács Péter
Tanársegéd
ELTE IK
Numerikus Analízis Tanszék



Budapest, 2014.11.23.

Tartalomjegyzék

1. Bevezetés	2
2. Matematikai áttekintés	5
2.1. PCA	5
2.2. SVD	10
2.3. Running SVD	14
2.4. Képfeldolgozás	16
3. Párhuzamosítási lehetőségek	19
3.1. OpenCV	19
3.2. CUDA	21
3.3. Tárolási lehetőségek	22
4. Eredmények	25
4.1. Motiváció	25
4.2. Golub - Reinsch SVD	26
4.3. Szekvenciális Running SVD	27
4.4. Párhuzamos Running SVD	27
4.5. Összefoglalás	29
5. Felhasználói dokumentáció	30
6. Fejlesztési lehetőségek	32
Függelék	35

1. fejezet

Bevezetés

A számítógépes látás elsősorban a digitális képi tartalom feldolgozásával foglalkozik, ezen belül objektumok felismerésével és mozgásuk detektálásával, egymáshoz viszonyított geometriai tulajdonságainak meghatározásával. Széleskörű elterjedését főleg a 20. század végén bekövetkező technikai innovációnak köszönheti. Példaként említhetjük a fényképezőgépek, okostelefonok képstabilizáló, objektumkövető alkalmazásait, de ide tartozik a facebook megjelölés opciójához használt arcdetektálás is.

A számítógépes látás célja, hogy a 3 dimenziós világból információkat, tulajdonságokat nyerjen ki digitális képek feldolgozásával. A számítógépes látás, a névből eredően, egyfajta digitális látásra törekszik. Amit az emberi szem lát, azt az agy egy pillanat alatt fel tudja dolgozni. Ugyanez a folyamat a gépi feldolgozásban teljesen különböző módon történik. Például amint megpillantunk egy vázát, meg tudjuk mondani, milyen alakja van, ugyanakkor, ha egy csapat embert látunk egy parkban, képesek vagyunk megszámlálni, hogy közülük hányan állnak vagy ülnek. A számítógépnek ezeket az információkat kell meghatároznia a rendelkezésre álló képekből különböző transzformációk és matematikai módszerek segítségével. Azonban a képek és videók elemzése közben számos probléma adódhat, amiknek a megoldása komoly matematikai háttérrel igényel. Ilyen például a környezetben lévő 3D-s alakzatok felismerése, a tárgyak mozgásának és irányának meghatározása, valamint hasonló emberek és objektumok felismerése.

A számítógépes látást – sokszínűsége és sokrétűsége miatt – rengeteg helyen alkalmazzák. Ide sorolható a gyártósorokon az alkatrészek felismerése; az orvostudományban a daganatok detektálása és az ultrahangos felvételeken történő objektumok észlelése, körberajzolása. Számos területen alkalmazzák a hadseregben is ezt a technológiát: például az ellenséges csapatok és járművek detektálására, esetleg fegyverzet felismerésére; rakéták irányítására, melyeket egy adott területre lőnek ki, melyek csak a terület feltérképezése után választják ki a célpontot. De a legújabb

technológiák, mint amilyen például az Ön-Vezetés, vagyis a vezető nélküli járművek gyártása, illetve az automatizálás sokat köszönhetnek e tudományágnak.

Az arcfelismerő rendszerek implementálása is a számítógépes látás feladatai közé tartozik. Ekkor az arcok tulajdonságai, az ún. „feature”-ök, egy adatbázisban vannak eltárolva. Ezek segítségével a kamera – mozgás hatására – az aktuális arcot, vagy az arc egy tulajdonságát az adatbázisban lévő információkra illeszti.

Ma már egy egyszerű fényképezőgép szoftverében is találunk olyan alkalmazást, melynek működése a számítógépes látáson alapszik. Ilyen például a mosoly funkció. Ebben az esetben például a fénykép csak akkor készül el, ha az objektív előtt álló személy arcvonásai között a gép mosolyt érzékel.

A TDK dolgozatomban, a számítógépes látás alkalmazásai közül a statikus, szűrkeárnyalatos videófelvételeken történő mozgás detektálást valósítottam meg. Statisztikus videó felvétel az, ahol a kamera egy stabil állapotban áll a felvétel során, ezáltal a háttér fixnek tekinthető és csak a frame-k közötti változásokkal kell foglalkozni. Ez történhet úgy, hogy a háttérrel valamilyen módszerrel megbecsüljük, majd megvizsgáljuk, mennyiben tér el a háttér és az aktuális képkocka. Léteznek gyors, de pontatlan módszerek, amelyek nem veszik figyelembe a háttérhez tartozó objektumokat, csak az egymást követő frame-ek közti különbséget vizsgálják [1]. Azonban ezek a módszerek olyan, feladat szempontjából irreleváns, környezeti változásokat is detektálnak, mint például a falevelek, eső, vízfelszín mozgása, stb. Ezzel szemben vannak nagy pontosságú módszerek, amelyek viszont nem alkalmasak valós idejű feldolgozásra. Ilyen például az Szinguláris Érték Felbontás (Singular Value Decomposition - SVD). Egy egyszerű videóra gondolva – ami kisméretű a jelenleg használt felbontásokhoz képest – nagy adatszerkezetekkel kell dolgozni. Ezért a fő feladat az algoritmusok optimalizálása, párhuzamos implementálása volt.

A BSC-s szakdolgozatom [2] keretein belül egy SVD alapú mozgásdetektáló algoritmust valósítottam meg. Ennek alapkonceptiója, hogy először adott számú frame-re meghatározza a háttérrel, majd minden egyes új frame-nél frissíti a háttér mátrixot, melyre ismét lefuttatja az algoritmust. Az így kapott módszer számításigénye nagy, csak offline video feldolgozást tesz lehetővé. Az SVD futásidejének csökkentését az irodalom több cikkben is tárgyalja. Példaként említhető, az eredeti videó felvétel átméretezése [3], blokkokba darabolása [4], stb. Ezzel azonban csökken az SVD alapú módszerek robusztussága. A TDK dolgozatban egy olyan SVD algoritmust használok melynek bonyolultsága csak lineárisan függ az egyes framek méretétől [5,6]. Az eredeti módszert grafikus kártyán párhuzamosítottam, majd alkalmaztam statikus videófelvételek elemzésére. Így egy olyan iterációs módszert implementáltam, amely hasonlóan robusztus, viszont jól alkalmazható a videóelemzés területén.

A dolgozatban nagy figyelmet kaptak az implementálási problémák, tesztelési

esetek és a matematikai háttérrel foglalkozó fejezetek. Ez elsősorban azért volt fontos, hogy átláthatóbb és érthetőbb formában tudjam prezentálni az implementált módszereket. Ennek megfelelően először ismertetem az alkalmazott módszerek megértéséhez szükséges matematikai háttérrel, amely a megvalósított műveletek és eljárások teljes megértéséhez elengedhetetlen. Ezek után, a fejlesztői rész keretein belül a program részletes és átfogó működését, osztályhierarchiáját fogom tárgyalni. Majd pedig, egy fejezet erejéig kitérek a megvalósításkor előforduló problémákra, amelyek egyrészt a nagy adatszerkezetek, másrészt az implementált algoritmus robusztuságának volt köszönhető. Ezek után a párhuzamosítási lehetőségeket veszem sorra, kijelölve az alkalmazott párhuzamos számítási platformot. Ezt követően részletesen értékelem a dolgozat főbb eredményeit. Nevezetesen összehasonlítom a BSc-s szakdolgozatom algoritmusát, illetve az új módszer párhuzamos és párhuzamosítás nélküli változatát. Az összehasonlítás alapja az algoritmusok futási ideje és az általuk felhasznált memória mennyisége lesz. A megvalósított algoritmus vizsgálata több konfiguráción történik, amely a hardveres technológia fejlődését is jól reprezentálja. Ezt követően a felhasználói oldalt fogom bemutatni. Ide tartoznak az alkalmazás telepítésével, futtatásával kapcsolatos információk, valamint azok az opcionális paraméterek, melyek a program működését befolyásolják. A dokumentáció végén szerepelnek a további fejlesztési lehetőségek.

2. fejezet

Matematikai áttekintés

2.1. PCA

A Főkomponens analízis (Principal Component Analyzis - PCA) egy matematikailag jól definiált ortogonális lineáris transzformáció, melyet sikeresen alkalmaztak kezdeti lépésként számos számítógépes látással kapcsolatos feladatban [7]. Például arc felismerésnél vagy akár objektum detektálásnál is.

A PCA alkalmazási területei közül, a TDK dolgozatban a feature vektorok dimenziójának csökkentésével foglalkoztam, ezen belül is statikus videókban történő mozgás detektálással. A videók képek sorozataiból állnak össze. Az egymás utáni frame-k pixelértékei között ott látható eltérés, ahol a háttér megváltozik, vagyis mozgás történik. Ha például biztonsági vagy traffipaxos felvételekre gondolunk, a pixelek értéke ott változik, ahol egy autó áthalad, vagy egy ember megjelenik a kamera előtt. Tehát az összes frame közül csak bizonyos tulajdonsággal bíró képekre van szükségünk. Matematikailag az egyes framek megfelelnek egy-egy $N \times M$ méretű mátrixnak. Ekkor egy frame megfelel egy pontnak az NM dimenziós térben. A számunkra szükséges képek pedig ennek a térnek az alterét fogják alkotni.

A háttér frame adaptív becsléséhez a dolgozatban PCA-t használtam. Így ebben a fejezetben [8] alapján vázlatosan bemutatom a PCA matematikai modelljét, illetve a megvalósításhoz használt szinguláris felbontást is.

Egy videofelvételen az egyes framek által rögzített információ általában redundáns. Tekintsük például az 2.1 ábrán látható 256×256 felbontású képet, mely a következő mátrixsal reprezentálható:

$$\mathbf{A} := \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix}.$$

Jelen esetben $n = 256$, illetve $a_{i,j} \in \{0, 1, \dots, 255\}$, mivel egy 8 bites szürkeárnyaltos képet vizsgálunk.

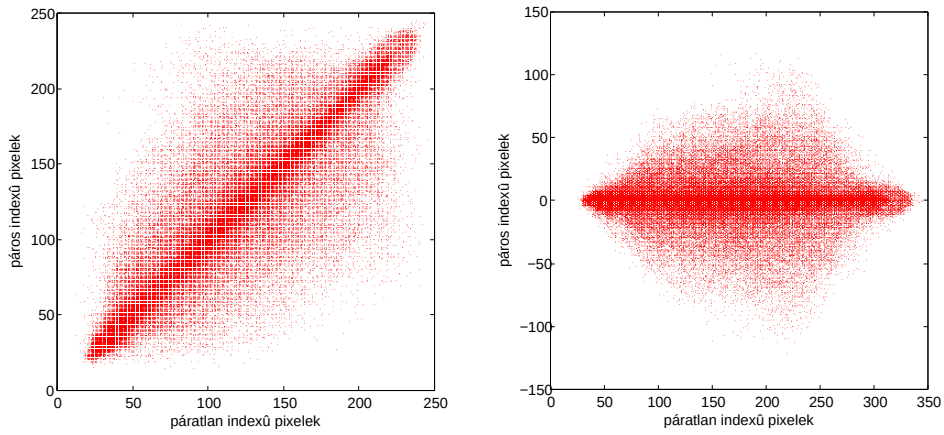


2.1. ábra. Eredeti, redundáns kép (Barbara image).

Rendezzük az $\mathbf{A}$ mátrix soraiban a szomszédos elemeket 1×2 blokkokba. Az így kapott kétdimenziós pontokat egy derékszögű koordináta-rendszerben ábrázoltuk a 2.2(a) ábrán. Látható, hogy a szomszédos pixelek értékei egymással erősen korrelálnak. Tehát ha ismerjük egy képpont intenzitását, akkor tudunk következtetni a szomszédos pixelek értékére is. Így tehát nem érdemes a blokkokat a hagyományos $\mathbf{e}_1 = [1, 0]$, $\mathbf{e}_2 = [0, 1]$ kanonikus bázisban felírni. Ezt szemlélteti a 2.3 ábra is, ahol a páros és páratlan pixelek hisztogramját rajzoltuk ki. A diszkrét sűrűség függvények hasonlósága mellett, jól látszik az is, hogy az adatok mindkét esetben kitöltik a rendelkezésre álló $[0, 255]$ tartományt. Tehát a páros és páratlan pixelek értékeit is 8 biten kell tárolnunk. A redundancia csökkentéséhez egy $\Phi : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ ortogonális bázistranszformációt fogunk használni. Nevezetesen, tekintsük az alábbi ún. Givens forgató mátrixot:

$$\mathbf{G} := \begin{bmatrix} \cos \varphi & -\sin \varphi \\ \sin \varphi & \cos \varphi \end{bmatrix}. \quad (2.1)$$

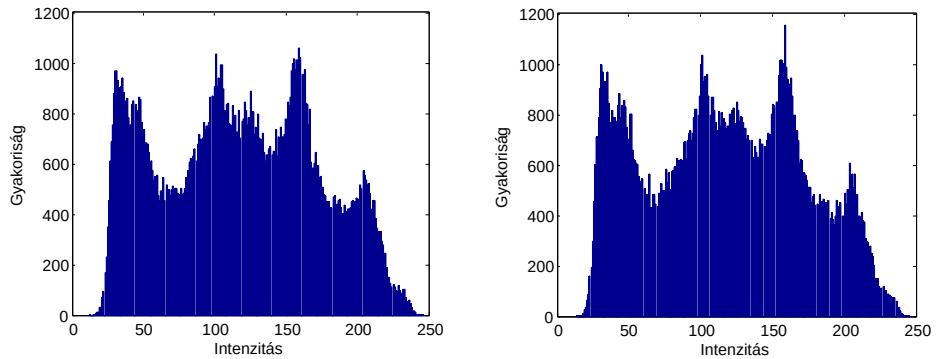
Mivel a pontok az $y = x$ egyenes mentén sűrűsödnek, ezért legyen $\varphi = \pi/4$. Ezek után a $\Phi = \mathbf{G}$ megkötéssel végezzük el az 1×2 méretű blokkokon a bázistranszformációt. Az eredmény egy 45° -os forgatás, melyet a 2.2(b) és a 2.4 ábrákon szemléltetünk. Utóbbin jól látható, hogy a transzformált adatsorozat páratlan indexű pixeleinek hisztogramja lényegében nem változott. A páros indexűeké viszont teljesen más. Itt az eredeti $[0, 255]$ -ös tartomány lényegében a $[-50, 50]$ intervallumra redukálódott. Ez még önmagában nem jelenti a tárolási kapacitás csökkenését, hiszen a 2.4(a) ábrán ez a tartomány $[0, 350]$. Tehát a páratlan indexek esetében 1 bitet veszítettünk, míg a párosak esetében 1 bitet nyertünk. Utóbbinál azonban az elemek szórása kicsi, a pixelek 90%-a a $[-30, 30]$ tartományba esik. Így veszteséges kódolás esetén, a páros indexű képpontokat 6 biten tárolva, ℓ^2 normában csak kicsit hibá-



(a) Eredeti adatsorozat (korreláló). (b) Elforgatott adatsorozat (dekorrelált).

2.2. ábra. Szomszédos pixelek korrelációjának szemléltetése.

zunk. Vegyük észre, hogy az új bázis rendszer elemei a 2D ponthalmaz legnagyobb kiterjedései felé azaz szórásirányba mutatnak. A jobb approximációs tulajdonság is ennek köszönhető, mely tovább javítható, ha $2 \leq n$ egymást követő pixel közötti korrelációkat vizsgálunk.



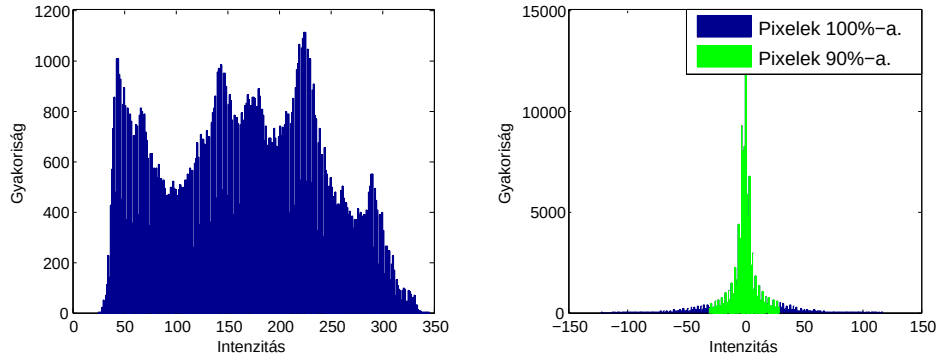
(a) Páratlan indexű pixelek hisztogramja. (b) Páros indexű pixelek hisztogramja.

2.3. ábra. Eredeti adatsorozat hisztogramjai.

A fenti, kétdimenziós példa után, most megadjuk a PCA általános definícióját. Ehhez vegyünk n darab egymást követő pixelt. Ez tekinthető az $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$ valószínűségi vektorváltozó (vvv) egy realizációjának. A célunk az, hogy a különböző koordináták korrelációja 0 legyen, vagyis

$$E(x_i x_j) = \begin{cases} 0 & \text{ha } i \neq j \\ \sigma_i^2 & \text{ha } i = j \end{cases}, \quad (2.2)$$

ahol σ_i^2 az i . valószínűségi változó szórásnégyzete. A feladat megoldásához szüksé-



(a) Páratlan indexű pixelek hisztogramja. (b) Páros indexű pixelek hisztogramja.

2.4. ábra. Transzformált adatsorozat hisztogramjai.

günk lesz az $\mathbf{x}$ vvv. kovariancia mátrixára is:

$$\mathbf{C}_{\mathbf{x}} := E((\mathbf{x} - \bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}})^T),$$

ahol $\bar{\mathbf{x}}$ az $\mathbf{x}$ vvv. várható értékét jelöli. A dolgozat további részében feltesszük, hogy $\bar{\mathbf{x}} = 0$. Ez nem jelent megszorítást, hiszen a feldolgozás előtt a diszkrét adatokból a mintaátlagot kivonva 0 várható értékű sorozatot kapunk. Így tehát $\mathbf{C}_{\mathbf{x}} = E(\mathbf{x} \mathbf{x}^T)$. A feladatunk az, hogy találjunk egy olyan $\Phi : \mathbb{R}^n \rightarrow \mathbb{R}^n$ lineáris ortogonális transzformációt, mellyel az $\mathbf{y} := \Phi^T \mathbf{x}$ vvv. teljesíti az (2.2) egyenlet feltételeit. Tehát

$$\begin{aligned} \mathbf{C}_{\mathbf{y}} &= E(\mathbf{y} \mathbf{y}^T) = E(\Phi^T \mathbf{x} (\Phi^T \mathbf{x})^T) \\ &= E(\Phi^T (\mathbf{x} \mathbf{x}^T) \Phi) = \Phi^T \mathbf{C}_{\mathbf{x}} \Phi, \end{aligned}$$

ami ekvivalens alábbival

$$\mathbf{C}_{\mathbf{x}} \Phi = \Phi \mathbf{C}_{\mathbf{y}}.$$

Vegyük észre, hogy fenti egyenlet i . sora meghatározza a $\mathbf{C}_{\mathbf{x}}$ mátrix i . saját vektor sajátérték párját (λ_i, Φ_i) :

$$\mathbf{C}_{\mathbf{x}} \Phi_i = \lambda_i \Phi_i \quad (i = 1 \dots n).$$

Tehát az (2.2)-ben definiált feladat a $\mathbf{C}_{\mathbf{x}}$ kovariancia mátrix diagonalizálásával ekvivalens. Könnyű belátni azt is, hogyha $|\lambda_1| \geq |\lambda_2| \geq \dots \geq |\lambda_n|$, akkor az $\mathbf{x}$ -et ℓ^2 normában legjobban közelítő vektor a következő alakban írható fel

$$\mathbf{x} \approx \hat{\mathbf{x}} := \sum_{i=1}^k \langle \mathbf{x}, \Phi_i \rangle \Phi_i.$$

Mivel a $\{\Phi_i : i = 1 \dots k\}$ rendszer ortogonális, így $\hat{\mathbf{x}}$ éppen az első k legnagyobb

sajátértékhez tartozó sajátaltérbe vett projekcióval egyenlő. Vegyük észre, hogy a $\{\Phi_i : i = 1 \dots k\}$ rendszer az $\mathbb{R}^n$ tér egy optimális bázisa az ℓ^2 normára nézve. Ez azt jelenti, hogy az $\mathbf{x}$ vvv-vel azonos eloszlású mintákat egy alacsony dimenziós $k \ll n$ sajátaltérben jól tudjuk közelíteni. Így például egy statikus videofelvételen, ahol az egymást követő framek erősen korrelálnak, elegendő csak az első néhány legnagyobb sajátértékhez tartozó sajátvektorra kiszámolni a vetületet. Az így kapott képen az erősen korreláló részek (pl. háttér) hibája vélhetően kicsi lesz. Az alacsony sajátértékekhez tartozó sajátvektorokra vett vetület $\mathbf{x}^\perp = \mathbf{x} - \widehat{\mathbf{x}}$ pedig a kép kevésbé korreláló részeit tartalmazza (pl. mozgó objektumok, zaj, stb.). Az így kapott eljárást nevezzük főkomponens analízisnek, Karhunen–Loève vagy Hotelling transzformációnak.

A PCA lényege, hogy az egymással többé-kevésbé korreláló sávok lineáris transzformációját úgy végezzük el, hogy a kapott főkomponensek korrelálatlanok legyenek. Az eljárás során az intenzitás vektorok eloszlását jellemző kovariancia mátrixból meghatározzuk a sajátértékeket, sajátvektorokat, és a képet a sajátvektorok által meghatározott koordináta-rendszerbe transzformáljuk. A transzformációt hívják fő-tengely transzformációnak is. E lényegkiemelő átalakítás után a kép első két sávjában kaphatjuk a teljes variancia több mint 90%-át, így a kép osztályozásához elegendő az első néhány sáv használata. Természetesen a főkomponens transzformáció megfordítható. A nem korreláló zaj az utolsó főkomponensekbe kerül, így a komponensek simításával, majd a főkomponensek vissza transzformálásával a kép speciális zajszűrését végezhetjük el. A PCA globális jellemzőket keres minimális hibát elérve.

A főkomponens analízist gyakran használják, mint az adattömörítés módszerét. Ezzel lehetővé válik a redundáns adatok kevesebb főkomponenssel történő ábrázolása, amely az adatok dimenziószámának csökkentését jelenti. A PCA főkomponensei nem-korrelálnak, függetlenek és gyakran könnyebb azokat ábrázolni, mint az eredeti forrásadatokat.

Bár a főkomponens analízis eredményeként n kimenő komponens jön létre, az első néhány ilyen tartalmazza a teljes adathalmaz információtartalmának 100 %-át az adatok nagy varianciája miatt. Ezért a főkomponens analízis alkalmas az adatok néhány sávba való tömörítésére. Másrészt a főkomponensek elemzése után megtalálhatjuk azt a képsávot, amely a legnagyobb kontrasztú az eredeti képhez képest. Utóbbit okozhatják a mozgó objektumok, de a szenzor hibájából származó zaj is. Ennek az eldöntésére a szakdolgozatomban az OpenCV függvénykönyvtár [9] magasabb szintű képfeldolgozási eljárásait használtam.

2.2. SVD

Az előző fejezetben láttuk, hogy a PCA előállításához szükségünk van a $\mathbf{C}_x \in \mathbb{R}^{n \times n}$ kovariancia mátrixra. A gyakorlatban ezt általában nem ismerjük pontosan, ezért ennek csak egy közelítését használhatjuk. Tekintsük az alábbi $\mathbf{X} \in \mathbb{R}^{n \times m}$ mátrixot, melynek oszlopait a kérdéses $\mathbf{x} \in \mathbb{R}^n$ vvv. m -szeres mintavételezésével kaptuk:

$$\mathbf{X} := \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_m \end{bmatrix}.$$

Ennek segítségével megadható az $\mathbf{x}$ vvv-hez tartozó kovariancia mátrix egy közelítése:

$$\mathbf{C}_x \approx \frac{1}{m} \mathbf{X} \mathbf{X}^T. \quad (2.3)$$

Esetünkben az $\mathbf{X} \in \mathbb{R}^{n \times m}$ mátrix oszlopait a videofelvételek egymást követő framejei alkotják majd. Ehhez tekintsük az $A_j \in \mathbb{R}^{N \times M}$ ($j = 1, \dots, m$) egycsatornás 8 bites szürkeárnyaltos képeket. Majd rendezzük az A_j pixeleit oszlopfolytonosan az $\mathbf{x}_j \in \mathbb{R}^n$ oszlopvektorba, ahol $n = N \cdot M$. Vegyük észre, hogy ekkor $\mathbf{X} \mathbf{X}^T \in \mathbb{R}^{n \times n}$, ami videofelvételek esetén egy nagyméretű, nem ritka mátrix. Például egy 640×480 felbontású kép esetén, $m = 15$ egymás utáni frame-t véve $\mathbf{X} \in \mathbb{R}^{(640 \cdot 480) \times 15}$, a kovariancia mátrix pedig $n^2 = (640 \cdot 480)^2$ elemet tartalmaz. Mivel minden pixelhez egy 8 bites intenzitás érték tartozik, így az $\mathbf{X} \mathbf{X}^T$ tárolásához kb. 88 GB memóriára lenne szükség. Ekkora adatmennyiséget valós időben egyelőre lehetetlen kezelni, nem is beszélve a sajátérték feladat megoldásáról. A probléma kiküszöböléséhez az $\mathbf{X}$ mátrix szinguláris felbontását fogjuk használni.

1. Tétel (SVD). Minden $\mathbf{X} \in \mathbb{R}^{n \times m}$ mátrix felírható egy $\mathbf{U} \in \mathbb{R}^{n \times n}$, $\mathbf{D} \in \mathbb{R}^{n \times m}$ és egy $\mathbf{V} \in \mathbb{R}^{m \times m}$ mátrix szorzataként. Az $\mathbf{U}$ oszlopai az $\mathbf{X} \mathbf{X}^T$, a $\mathbf{V}$ oszlopai pedig az $\mathbf{X}^T \mathbf{X}$ ortonormált sajátvektorai. A $\mathbf{D} = [\mathbf{\Sigma} \mathbf{0}]$, ahol $\mathbf{\Sigma} \in \mathbb{R}^{m \times m}$ és $\mathbf{\Sigma} := \text{diag}(\sigma_i)$. Továbbá, a σ_i ($i = 1, \dots, m$) számokat szinguláris értékeknek nevezzük. Ha feltesszük, hogy $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n > 0$, akkor a felbontás egyértelmű is.

Ha most visszahelyettesítjük az $\mathbf{X} = \mathbf{U} \mathbf{D} \mathbf{V}^T$ felbontást az (2.3) egyenletbe akkor kapjuk, hogy

$$\mathbf{C}_x \approx \frac{1}{m} \mathbf{U} \mathbf{D} \mathbf{V}^T (\mathbf{U} \mathbf{D} \mathbf{V}^T)^T = \frac{1}{m} \mathbf{U} \mathbf{D} \mathbf{D}^T \mathbf{U}^T.$$

Innen adódik, hogy $\lambda_i = \sigma_i^2/m$ és $\Phi = \mathbf{U}$. Tehát a PCA-t visszavezettük az $\mathbf{X}$ szinguláris felbontásának meghatározására. Felhívjuk a figyelmet arra, hogy a probléma nem lett egyszerűbb, hiszen $\mathbf{U} \in \mathbb{R}^{n \times n}$ továbbra is fennáll. Az SVD segítségével azonban lehetőségünk van arra, hogy az $\mathbf{U}$ mátrixnak csak az első néhány legnagyobb szinguláris értékhez tartozó oszlopát (sajátvektorát) számoljuk ki. Tegyük

fel, hogy a felbontásban szereplő $\mathbf{D}$, $\mathbf{V}$ mátrixokat már meghatároztuk. Ekkor

$$\mathbf{U} = \mathbf{XVD}^{-T}, \quad \mathbf{D}^{-T} = \begin{bmatrix} \boldsymbol{\Sigma}^{-1} \\ \mathbf{0} \end{bmatrix}, \quad (2.4)$$

feltéve, hogy $\sigma_i \neq 0$ ($i = 1, \dots, m$). Vegyük észre, hogy a kisebbik dimenziónál, az $\mathbf{X}^T \mathbf{X} \in \mathbb{R}^{m \times m}$ -hez tartozó sajátvektorokat könnyen meg tudjuk határozni. Így a $\mathbf{D}$, $\mathbf{V}$ mátrixok kiszámolása egyszerű. Ezek ismeretében pedig $\mathbf{U}$ első néhány oszlopa az (2.4) egyenlet segítségével könnyen meghatározható (bővebben lásd [10]). Megjegyezzük, hogy a dolgozatban implementált algoritmus [6] sorfolytonos elrendezést használ az $\mathbf{X}$ előállításához. Így a kérdéses sajátvektorokat a $\mathbf{V} \in \mathbb{R}^{n \times n}$ mátrix tartalmazza. A feladat az eredetivel ekvivalens, a fenti összefüggések $\mathbf{X}^T$ -ra analóg módon levezethetők. Ennek megfelelően a továbbiakban a célunk $\mathbf{V}$ oszlopainak meghatározása.

A lineáris algebra területén az SVD egy valós vagy komplex mátrix faktorizálását jelenti. Többek között használható mátrixok pszeudoinverzének kiszámolására (nem négyzetes mátrixok inverzének meghatározása), homogén lineáris egyenletrendszer (LER) megoldására, valamint elsősorban nagy méretű mátrixok rangszámának csökkentésére. Továbbá rengeteg hasznos alkalmazása ismert a digitális jel- és képfeldolgozásban, illetve az ezekből kinyert statisztikai adatok analízisében [11]. Ebben az esetben folyamatosan változó mátrixok valós idejű feldolgozása történik. A legtöbb alkalmazás a számítógépes látás területén például ilyen jellegű.

Általánosabban, az SVD felbontásban lévő $\mathbf{V}$ mátrix oszlopvektorai jelölik a főirányokat. Az adathalmaz pontjainak változása ezen irányok mentén a legszignifikánsabb. Ennek megfelelően a $\mathbf{D}$ mátrixban lévő σ_i szinguláris értékek az adathalmaz varianciáit mutatják a megfelelő főirányok mentén.

A BSc-s szakdolgozat keretein belül a Golub - Reinsch SVD [10] algoritmust implementáltam, amely két fő lépésből áll:

1. Bidiagonizálás
2. Golub - Kahan SVD step

Az implementációban nagy figyelmet szenteltem a BSc-s Numerikus analízis [12] tárgy keretein belül tanult forgatási és tükrözési algoritmusok megvalósítására, amely az SVD algoritmus alapját képezik. A bidiagonizálás lépése kialakítja a kezdetleges

$\mathbf{A} = \mathbf{UBV}^T$ felbontást, ahol a $\mathbf{B}$ mátrix a következő alakú:

$$\mathbf{B} := \left[\begin{array}{ccccc} d_1 & f_1 & 0 & \dots & 0 \\ 0 & d_2 & f_2 & & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & & d_{n-1} & f_{n-1} \\ 0 & \dots & & & d_n \end{array} \right] \in \mathbb{R}^{m \times n}$$

$$\begin{array}{c} 0 \end{array}$$

Az $\mathbf{U}$ és $\mathbf{V}$ mátrixokat Householder mátrixok szorzataként számolhatóak ki. A bidiagonalizálás művelet igénye $\mathcal{O}(4mn^2 - \frac{4n^3}{2})$.

A következő lépés a $\mathbf{B}$ bidiagonális mátrix szuperdiagonálisából kinullázni az értékeket. Ezt a Givens féle forgató mátrixok segítségével valósítottam meg. A forgatás lényege, hogy mátrix szorzások segítségével jelentősen csökkentjük a szuperdiagonális elemeinek az értékét. Ez egy iterációs folyamat, melyben a Golub - Kahan SVD lépést addig ismétljük, míg az átlón kívüli elemek abszolútértéke elég kicsi nem lesz. Utóbbi eldöntéséhez a következő egyenlőtlenségeket használjuk:

$$|f_i| \leq \epsilon (|d_i| + |d_{i+1}|) \quad (2.5)$$

$$|d_i| \leq \epsilon \|\mathbf{B}\| \quad (2.6)$$

ahol ϵ egy előre adott érték. Az algoritmus vizsgálata során az $\epsilon = 10^{-6}$ értéket használtam. A Golub - Reinsch SVD aszimptotikus lépésszáma $\mathcal{O}(4mn^2 + 8n^3)$.

Lehetséges gyakorlati problémák

Gyakorlati példa korlátok feloldására

Vizsgáljunk meg egy 640×480 felbontású, szürkeárnyaltos videó felvételt. Olvassunk be $m = 50$ darab frame-t és tároljuk el sorfolytonosan az $\mathbf{A} \in \mathbb{R}^{50 \times (640 \cdot 480)}$ mátrixban. A kapott mátrixnak határozzuk meg az SVD felbontását. Ekkor az $\mathbf{A}$ mátrix szinguláris felbontása az $\mathbf{A} = \mathbf{UDV}^T$, ahol $\mathbf{D} \in \mathbb{R}^{50 \times (640 \cdot 480)}$, $\mathbf{U} \in \mathbb{R}^{50 \times 50}$. A jobb oldali $\mathbf{V}$ mátrixot pedig a következő formula segítségével számoltuk ki:

$$\mathbf{V} = (\Sigma^{-1} \mathbf{U}^T \mathbf{A})^T = \mathbf{A}^T \mathbf{U} \Sigma^{-1}, \quad \Sigma \in \mathbb{R}^{50 \times 50}, \quad \Sigma = \text{diag}(\mathbf{D}). \quad (2.7)$$

A képlet alapján a bal oldali mátrix, $\mathbf{V} \in \mathbb{R}^{(640 \cdot 480) \times 50}$ méretű lesz. Ezt az adathalmazt már tudja kezelni a számítógép, valamint egyszerűbbé válik a számolás is. Azonban (2.7) egyenlet számolása még mindig nem elég hatékony. Egyrészt tetszőle-

ges $\mathbf{A} \in \mathbb{R}^{m \times n}$ mátrix SVD felbontása még így sem egyértelmű, másrészt lehetséges, hogy a $\mathbf{D}$ diagonális elemei 0 közeli értékűek lesznek. Utóbbi miatt a Σ^{-1} kiszámolása numerikusan instabil, mivel előfordulhat, hogy az átlóbeli elemeket 0-val kell osztanunk. Ezért a következő módosításokat hajtjuk végre:

1. A $\mathbf{D}$ mátrix diagonális elemei legyenek pozitívak. Tehát $\mathbf{A} = \mathbf{U} \mathbf{D} \mathbf{I}_s \mathbf{V}^T$, ahol $(\mathbf{I}_s)_{ii} = \text{sign}(\mathbf{D}_{ii})$ egy diagonális előjel mátrix. Innen a $\hat{\mathbf{D}} = \mathbf{D} \mathbf{I}_s$ mátrix átlója már pozitív és $\hat{\mathbf{V}}^T = \mathbf{I}_s \mathbf{V}^T$.
2. A $\hat{\mathbf{D}}$ mátrix diagonális elemeit rendezzük csökkenő sorrendbe. Ennek megfelelően változtassuk a $\hat{\mathbf{V}}^T$ mátrix oszlopait is.
3. Legyen $\hat{\Sigma} \in \mathbb{R}^{r \times r}$ a $\hat{\mathbf{D}}$ mátrix csak pozitív elemeit tartalmazó diagonális részmátrixa. Jelöljük $\tilde{\mathbf{U}} \in \mathbb{R}^{m \times r}$ és $\tilde{\mathbf{V}}^T \in \mathbb{R}^{r \times n}$ -val azokat a részmátrixokat, melyeket úgy kapunk, hogy elhagyjuk $\hat{\mathbf{U}}$ és $\hat{\mathbf{V}}^T$ utolsó $n - r$ oszlopát és sorát, továbbá legyen $\tilde{\mathbf{D}} = \hat{\Sigma}$ (lsd. 2.5. ábra).

Az így kapott $\mathbf{A} = \tilde{\mathbf{U}} \tilde{\mathbf{D}} \tilde{\mathbf{V}}^T$ felbontás az előjelektől eltekintve már egyértelmű és numerikusan stabil.

$$\left[\begin{array}{c} \mathbf{A} \end{array} \right] = \left[\begin{array}{c} \tilde{\mathbf{U}} \end{array} \right] \left[\begin{array}{c|c} \tilde{\mathbf{D}} & \mathbf{0} \end{array} \right] \left[\begin{array}{c} \tilde{\mathbf{V}}^T \\ \hline \end{array} \right]$$

2.5. ábra. Redukált SVD felbontás.

Minimális adategységek

Az előző probléma jól mutatja, hogy a dolgozat írása során a hatalmas adatszerkezetek eltárolása volt a legnehezebb feladat, amire folyamatosan kellett figyelni és mindig a leoptimálisabb tárolásra törekedni. Emellett az adatok mozgatására és a számítási idő minimalizálására is nagy figyelmet fordítottam. Ekkora adatmennyiség mellett ez bonyolult feladat, ezért a lehető legkisebb egységekre történő memória lefoglalásra és felszabadításra törekedtem.

Az egyik ilyen megoldás, hogy az SVD algoritmus minél kisebb mátrixot diagonalizáljon. Ezért a bidiagonalizálásból kapott $\mathbf{B} \in \mathbb{R}^{m \times n}$ mátrixot négyzetessé alakítottam, aminek köszönhetően már csak egy $\tilde{\mathbf{B}} \in \mathbb{R}^{m \times m}$ alakú mátrix diagonalizálása volt a feladat, ahol az m érték jelzi a beolvasott framek darabszámát.

Folyamatos figyelés

Az algoritmus tehát hatalmas adatokhalmazokkal dolgozik, és emiatt az optimalizációra is kellő figyelmet kellett fordítanom az implementálás során. További problémát jelentett, hogy a ciklikus ismétlések következtében a bidiagonalizáló, illetve forgató mátrixok, felszabadítás nélkül a memóriában maradhattak. Emiatt, a futás során a dinamikus memória hamar megtelt, ami a program abortálását eredményezte. Ezért a már nem használatos adattagok felszabadítását folyamatosan felügyelni kellett, főképp azokat a tárterületeket, melyek a számolási fázis szempontjából feleslegessé váltak. Ennek a problémának egy lehetséges, gyors tesztelési módja, ha egy nagyobb videót analizálunk az algoritmussal, vagy a háttér approximációjához több ($0 \ll m$) frame-t használunk fel. Ekkor a számításokban szereplő mátrixok elég nagyok ahhoz, hogy hamar felhasználjuk a rendelkezésre álló memória területet.

Az implementáció során többször is memória szivárgásba ütköztem, melynek keresése jelentősen megnehezítette a fejlesztés előrehaladását. A megvalósítás során fontos szempont volt a program skálázhatósága. Ezért a tesztelési fázist nagy méretű videókra, nagyméretű mátrixokkal, jelentős mennyiségű frame-en hajtottam végre. Erről bővebben a 4. fejezetben lesz szó.

2.3. Running SVD

Az előző fejezetben bemutatott, a BSc-s szakdolgozatomban implementált SVD felbontás és az ezzel történő mozgásdetektálás nem bizonyult valós időben használható modellnek. Ennek oka a háttérbecsléshez használt frameszám (amely a detektálás pontosságát befolyásolja). Viszont a ma használatos videó fájlok felbontása és a használt képkockák száma hatalmas adatmennyiséget eredményez. Az eddig lemért futási eredmények azt mutatták, hogy egy 768×576 felbontású videó, 30 képkockából történő háttérbecslése után a további 20 frame feldolgozásához közel ~ 20 percet volt szükség.

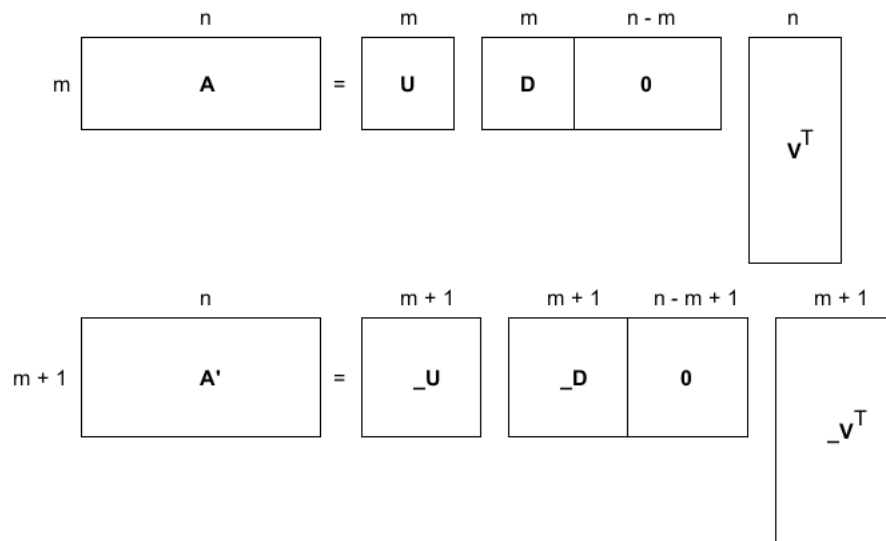
Egy lehetséges futás idejű gyorsítás, ha a videó méretét redukáljuk. Ezzel képi információkat veszítünk, továbbá az eredeti algoritmussal a valós idejű feldolgozás így sem lesz elérhető. Ezért a matematikai modellt kell tovább fejleszteni. Az eredeti algoritmus tesztelésekor körvonalazódott, hogy a modell leginkább számítást igénylő része az SVD felbontás. Az algoritmus működéséből adódóan, minden iteráció tartalmazza a háttér becslését, amely egy $m \times framesize$ méretű mátrix felbontását jelenti (m nagysága befolyásolja a detektálás pontosságát). Ezért a matematikai modellt úgy érdemes átalakítani, hogy a kezdeti háttérbecslés után, a felbontandó mátrixra ne legyen szükség. Más szóval az új frame és a háttér közti eltérés meghatározásához elegendő az iterációk előtt kiszámolt SVD felbontás mátrixainak és az

aktuális képkockának a további használata [5, 6]. Az algoritmus előnye, hogy nem szükséges blokkokra bontani a képmátrixot, szemben más mozgásdetektáló megoldásokkal [3, 4]. Így a feldolgozás ideje jelentősen csökkenthető. Továbbá könnyedén párhuzamosítható, amely fontos lépés a valós idejű feldolgozás felé. A 3. fejezetben látni fogjuk, milyen párhuzamosítási lehetőségek vannak a képfeldolgozás területén. Az GPU implementációval elért eredményeket a 4. fejezet tárgyalja.

A dolgozatban alkalmazott ún. Running SVD algoritmus minden új frame érkezésekor két lépést hajt végre.

- **Update** - Frissíti az SVD felbontás három mátrixát az aktuális frame szerint.
- **Downdate** - Törli a legrégebbi frame értékeit a mátrixokból.

Mindkét lépés méretváltoztatást hajt végre a mátrixokon. Az *Update* eljárás megnöveli mindhárom mátrix dimenzióját eggyel, ezzel elhelyezve az új frame-et a mátrixokban. A Running SVD algoritmusban a bemeneti és kimeneti változók dimenziójának meg kell egyeznie, ezért az *Downdate* eljárásban eltávolítja a feleslegessé vált frame értékeit a mátrixokból, valamint visszaállítja az eredeti méretet. Annak érdekében, hogy az eljárás kimenete és bemenete különbözzön, a *Downdate* előtt szükséges egy shift művelet az **U** vagy **V** mátrixra. Így az első lépésben a kiegészített mátrix SVD-jét, a második lépésben pedig a redukált mátrix SVD-jét kapjuk. Az algoritmus két lépésének számítási bonyolultsága $\mathcal{O}((m+n)m^2)$, ahol m a háttérbecsléshez használt frame-ek darabszáma, míg az n a képen lévő pixelek száma. A lépések során keletkező mátrixokat a 2.6 ábra szemlélteti. Az algoritmus pszeudokódját az 1. és a 2. függelék tartalmazza.



2.6. ábra. Az *Update* és a *Downdate* műveletek mátrixai.

2.4. Képfeldolgozás

A kezdeti mátrix felbontás, illetve az aktuális képkockával történő frissítés után elkezdhetjük a tényleges képfeldolgozási folyamatot. Az eddig használt adatstruktúrák kizárólag lebegőpontos értékeket használnak, viszont a kapott eredmények még nem voltak megjeleníthetőek. A továbbiakban a képi feldolgozásban használt, alapvető fogalmakkal ismerkedünk meg.

Az implementált algoritmus képfeldolgozási fázisa öt lépésből áll. Legelőször ki-nyertük, a háttér becslése után, a valós mozgást jelentő objektumokat. Ezek után bináris képek segítségével az objektumok kontúrját határoztuk meg. Majd utolsó fázisként az eredeti képkockára helyeztük rá az így megkapott körvonalakat. A képelemzés egyes lépéseit az alábbi felsorolás részletezi:

Vetület kép az a frame, ahol a háttér már fixnek tekinthető. Az eredeti képhez képest a különbség annyiban látszik, hogy a mozgó objektumok homályosabbak.

Különbség kép az a frame, ami az eredeti kép és a vetület kép különbségéből keletkezik. Vizuálisan leírva egy közel fekete képet kapunk, ahol a mozgó objektumok nagyobb intenzitásban jelennek meg. Utóbbiak tehát világosak, ezzel szemben a háttér sötét vagy szinte teljesen fekete.

Küszöbölés hatására egy bináris képet kapunk, ahol a mozgó objektumok pixellei 255 értékűek, míg a háttérhez tartozó pixelértékek 0-k. Az eljárás a kép minden egyes pixelben ellenőrzi, hogy az intenzitás kisebb-e, mint a küszöbölési érték. Ha a feltétel teljesül, akkor ezt 0-ra állítja, egyébként pedig 255-re.

Opening - Closing operátor segítségével a küszöbölés után kapott "lyukas" objektumokat lehet kitölteni. A két operátort egymás után kell használni. A closing segítségével elimináljuk a küszöbölés után visszamaradt zajt, majd ezután az opening operátorral az egymáshoz közel álló, egybetartozó területeket kapcsoljuk össze.

Sobel operátor segítségével kiemeljük a képen az éleket, így a detektált objektum körberajzolása leegyszerűsödik. A kitöltött objektumok helyett az objektumok körvonalai jelennek meg.

A 4. fejezetben elsősorban a program futási idejét illetve a futás során felhasznált memória mennyiségét hasonlítom össze, a feldolgozás eredményeként kapott képek pontosságát nem. Ezért érdemes egy kisebb kitekintést tenni a futás részeredményeire, ezzel egyben bemutatva az előbb részletezett képfeldolgozási fogalmakat.

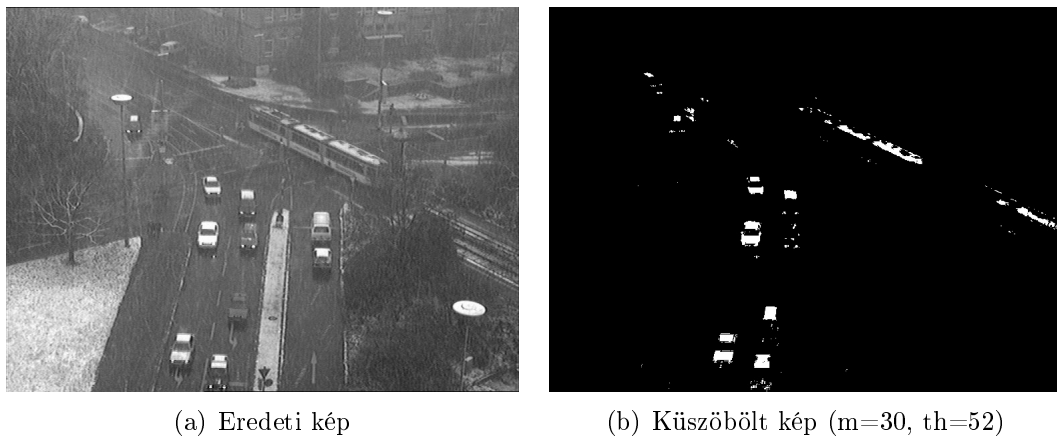
Először érdemes a vetület képet tekintenünk, hiszen ez lesz a háttér becslése után az első eredmény, amit kapunk. A vetület kép jelentősége, hogy a háttér fix, a mozgó

objektumok pedig homályosak. A következő lépés a különbség kép küszöbölése. Így



2.7. ábra. Teszt normál fényviszonyok között.

eliminálhatjuk a képen megmaradt zajokat, amely függ a küszöb mértékétől. Az eljárás után egy bináris fekete/fehér képet kapunk. A detektált kép tartalmazza



2.8. ábra. Rossz időjárási viszonyok melletti tesztelés.

az eredeti képkockát, valamint a valós mozgást előidéző objektumok kontúrját. Az ábrán látható, hogy a detektálás pontos, a meghatározott körvonalak illeszkednek az emberekre, továbbá jól elkülöníti a háttérrel a mozgó objektumoktól. Továbbá az esőcseppek és más irreleváns objektumok mozgását nem detektáltuk. A 2.10 képről – amellett, hogy zajos felvételt ábrázol – elmondható, hogy a háttérbecsléshez használt képkockák száma minimális volt. Ezért a detektálás pontatlannak bizonyult. Ezt az objektumok részleges körvonalai szemléltetik.



(a) Eredeti kép



(b) Detektált kép ($m=30$, $th=69$)

2.9. ábra. Normál fényviszonyok melletti tesztelés.



(a) Eredeti kép



(b) Detektált kép ($m=5$, $th=50$)

2.10. ábra. Teszt zajos felvételen.

3. fejezet

Párhuzamosítási lehetőségek

A számítógépes látás és a képelemzés területén a valós idejű adatfeldolgozáshoz szükséges - az optimális algoritmus mellett - a megvalósítandó módszer párhuzamosítása. A képfeldolgozást támogató programcsomagok között több olyan is található, amely a párhuzamos programozás alapos ismerete nélkül, könnyű párhuzamos implementációs megoldásokat biztosít [13, 14].

A következőkben a képelemzéssel foglalkozó OpenCV és CUDA programcsomagok kerülnek bemutatásra. A releváns különbség a két programcsomagban, hogy míg az OpenCV iteratív módszerek megvalósítására alkalmas, valamint a CPU memóriájában tárolja el az osztály adattagjait, addig a CUDA egy párhuzamosított környezetet biztosít, amelyben a memória foglalkozás a videokártya által biztosított memóriában történik. Mindemellett a két platform között az is különbség, hogy míg az OpenCV sorfolytonosan tárolja az adatokat a memóriában, addig a CUDA oszlopfolytonosan valósítja ezt meg. Így a két módszer együttes használatánál figyelni kell az adattárolás módjára, amely további adatmozgatást eredményez. Ezért fontos még az implementálás előtt eldönteni, hogy milyen koncepció alapján fejlesszük az alkalmazást.

3.1. OpenCV

Az OpenCV – Open Computer Vision – egy nyílt forráskódú programcsomag (licenc változata is létezik), amely a számítógépes látás, valamint a valós idejű képelemzés alapvető funkcióit, operátorait foglalja magában és egyszerű formában támogatja a fejlesztői munkát. Az egyszerűség azon alapszik, hogy a fejlesztő részletes leírást kap a programkönyvtár függvényeiről a dokumentációban [9], példákkal, kimeneti hibákkal és opcionális lehetőségekkel. Ezen felül példaprogramokon keresztül ismerteti meg a fejlesztőt a kezdeti lépésekkel.

Az OpenCV sajátossága a számítási hatékonyság. A képelemzésben valamint

a számítógépes látás területén a futási idő a legrelevánsabb tényezője a használt módszernek. Ennek oka a videók és képek méretének és minőségének fokozatos növekedése. Ez azért is fontos, mert az OpenCV erősen a valós idejű képelemzésre fókuszál. Azonban nem minden probléma oldható meg kizárólag az OpenCV segítségével. Ha egy szürkeárnyaltos videó elemzésére gondolunk, ahol a videó mérete 640×480 és pontos mozgásdetektálást szeretnénk végezni, akkor a 2.1 fejezetben említett kovariancia mátrix $(640 \cdot 480) \times (640 \cdot 480)$ dimenziójú lenne. Ebben az esetben $9,43 \cdot 10^{10}$ adatot kell kezelni. Ha az adatokat 4 byte-on tároljuk, akkor ~ 350 GB memóriára lenne szükség a feldolgozáshoz, amely a mai technológia mellett nem lehetséges, továbbá nem is lenne hatékony. Viszont a matematika biztosít olyan eszközöket és modelleket, például az SVD ilyen, amelyek ezt a memória mennyiséget minimalizálják. Mozgásdetektálást implementáló függvényeket az OpenCV is tartalmaz, viszont valós idejű videó feldolgozásra alkalmatlan, elsősorban futási idő és pontatlansági okok miatt.

A kezdeti lépésekhez hozzátartozik az OpenCV saját mátrix – illetve képtípusainak a részletes áttanulmányozása. A két legfontosabb struktúrája az `IplImage` illetve a `CvMat` (létezik C++ nyelvnek megfelelő struktúra is). Mindkét adatszerkezet legfontosabb paraméterei a méretek (szélesség, magasság) illetve az adattagra mutató pointer. Mivel az OpenCV csak dinamikus tagokkal dolgozik, emiatt a fejlesztőknek folyamatosan figyelniük kell az adattagok lefoglalására illetve felszabadítására. A továbbiakban felsorolunk néhány alkalmazást, ahol az OpenCV kiválóan alkalmazható megoldás:

- **Mozgáskövetés** - Videó felvételeken történő mozgások (általában statikus felvételekre alkalmazzák) detektálása, valamint objektumok követése.
- **Objektumazonosítás** - Megadott tulajdonságú objektumok detektálása képeken és videókon. A módszer előnye, hogy akkor is hatékonyan működik, ha az objektum takarásban van (az objektum csak egy része látható).
- **Szegmentálás** - A képeken megtalálható objektumok elkülönítése. A szegmentálással érhető el, hogy a bonyolultabban értelmezhető objektumok egy-egyét különválasszuk.
- **Arcfelismerés** - Elsősorban biztonsági kamerák működésének alapját képezi. A videó felvételek folyamatos elemzésével embereket lehet azonosítani az arcukat leíró tulajdonságok alapján. Ez az opció beléptető rendszerek esetén is használatos. Ebben az esetben az arcok leírását (az arcokat leíró tulajdonságokat) egy erre külön létrehozott adatbázis tárolja. Az ilyen típusú alkalmazásokkal képesek a videokamerák egy meghatározott embert az arcvonásai alapján azonosítani.

3.2. CUDA

A CPU magokra bontása a párhuzamosítás egyik fontos lépésének számít. A többmagos processzorokkal (kezdetben két mag szolgálta ki a kéréseket) megnyílt az út a párhuzamosítás felé, amely nagymértékben felgyorsította az operációs rendszereket, illetve csökkentette a programok futási idejét. Viszont nem minden esetben volt elegendő az egyprocesszoros, többmagos megoldás. Így ahhoz, hogy a párhuzamosítás hatékony legyen, több processzort építettek be a gépekbe. A módszer drágának bizonyult, emiatt a felhasználók inkább a 2 – 4 – 8 magos processzorok felé nyitottak.

Az NVIDIA kutatói kifejlesztették és 2007 februárjában publikálták a CUDA programozási architektúrát [15], amely a párhuzamosítási feladatok között áttörést jelentett. A CUDA (Compute Unified Device Architecture) a számítógépbe épített videó kártyájának memóriáját használva ad lehetőséget a párhuzamos feladatok gyors és hatékony elvégzésére. A CUDA kialakításakor a SIMT (Single Instruction Multiple Thread) architektúrát vették alapul, amely a SIMD (Single Instruction Multiple Data) egy speciális változata [16]. A SIMD megvalósításkor a program egy időben képes ugyan azt az utasítást több adaton végrehajtani. Ezzel szemben a SIMT arra hivatott, hogy minél több, egymástól elkülönített szál fusson azonos időben. A CUDA az egyes szálakat blokkokba csoportosítja, a blokkokat pedig grid-be.

A GPU-s párhuzamosítás a CPU-s megoldással vetekszik. Sok összehasonlítást lehet találni, amelyeken azt érzékeltetik, hogy a GPU-val történő párhuzamos megoldások gyorsabbak, hatékonyabbak, illetve, jobban optimalizáltak. Egy fontos kritériuma van ennek a hatékonysági faktornak, amivel a két párhuzamosítás nagymértékben elkülönül. Olyan feladatok megoldásaira alkalmas a CUDA, ahol nagymennyiségű adaton kell numerikusszámítást elvégezni. A nagymennyiségű adathalmazt jól érzékelteti az OpenCV-nél említett mozgásdetektálási probléma. Az ilyen típusú feladatok megoldásához a CUDA megfelelő választást jelent.

A CUDA működésének alapelve:

- A bemenő adatot a CPU memóriájából átmásolja a GPU memóriájába.
- A CPU utasítást küld, hogy végezze el a megadott utasításokat a GPU-n.
- Ezt az utasítást elküldi a feldolgozó egységnek. A virtuális számítási magok elvégzik az utasításokat az adatokon.
- A végrehajtás után az eredményt a kimenetre kell küldeni (GPU memóriájából visszamásolódik az adat a CPU memóriájába).

Habár a CUDA a párhuzamos végrehajtás miatt gyorsabb végrehajtást tesz lehetővé, az OpenCV-vel ellentétben nincsenek előre implementálva az alapvető képelemző algoritmusok. Amit a CUDA tartalmaz és a képelemzéshez szükséges, az a megfelelő adatszerkezet. Olyan struktúrákkal tudjuk felépíteni a programot, amivel a GPU-n tárolt adatokat közvetlenül el lehet érni. Ha csak egy egyszerű tömböt használunk ehhez, akkor a GPU memóriájában történő helyfoglalás után csak procedurális műveleteket lehet rajtuk végezni, viszont adatot nem tudunk belőlük kiolvasni. Ezt a problémát oldja fel a CUDA Thrust adatszerkezete [17].

A CUDA használatához több külső programkönyvtár is társulhat:

- **cuBLAS** - CUDA Basic Linear Algebra Subroutines [18]
- **Cula** – CUDA Linear Algebra [19]

A két könyvtár között a lényeges különbség, hogy az előbbi az alapvető mátrixműveleteket tartalmazza. Ahogy azt korábban említettük, az OpenCV is mátrixokkal dolgozik, mátrixokon végzi el a műveleteket szekvenciális módszerekkel. A cuBLAS ugyanennek a párhuzamosított megvalósítása. Az Intel is kifejlesztett egy hasonló könyvtárat, az MKL BLAS-t (Math Kernel Library). Utóbbi a lineáris algebra mátrix felbontásainak párhuzamos változatait tartalmazza. A cuBLAS azonban 5-7-szeres teljesítmény növekedés biztosított az Intel megoldásához képest [20].

A dolgozat implementálásában mindkét külső könyvtárat használtam. Az előbbit a mátrixműveletekkel kapcsolatos lépésekben (mátrixszorzások, mátrixok normáinak kiszámolása), az utóbbit pedig az SVD eljárás implementálásánál.

3.3. Tárolási lehetőségek

Pár évvel ezelőtt még nem volt releváns kérdés a megfelelő háttértár típusának a kiválasztása. Ennek oka, hogy adattárolásra csak a mágneses lemezeket lehetett használni, vagy még nem érték el azt az árkategóriát az újabb technológiák, melyek megfeleltek volna a hétköznapi konfigurációkhoz. A háttértárak legjellemzőbb tulajdonságai a csatlakozó típusa (SATA), a fordulatszám illetve az adatsűrűség volt. Most azonban érdemes a technológia egy újabb ágára tekintenünk és összehasonlítani a háttértárak egy következő állomását, az SSD-t. Az algoritmusok futási idejének tesztelését különböző környezetekben végeztem, amely közül az egyik – a legújabb és egyben legerősebb – konfiguráció tartalmazott SSD háttértárat. Látni fogjuk, hogy a mágneses adattárolástól való elszakadás gyorsabb feldolgozást eredményez. Az eredményeket később a 4. fejezetben részletezem.

A HDD (Hard Disk Drive) mágnesezhető lemezekből felépülő, olvasásra és írásra alkalmas fejekkel ellátott mechanikus periféria. A gyártók egyre inkább fessegetik

ennek a technológiának a határait, bár ez a pont még nem jött el. A HDD-k teljesítményének fejlődését a fordulatszám, az interfész-ek fejlesztése, a fejtechnológiai újítások és az adatsűrűség növelése befolyásolta. A merevlemezek nagy mennyiségű adatok tárolására kiválóan alkalmasak, notebook és asztali gépekben egyaránt. A piacon megtalálható merevlemez kapacitások az átlagos 500 GB-tól a nagyobb 4 TB-ig megtalálhatóak. A HDD hátránya, hogy az olvasási/írási fej folyamatos mozgást végez, így az adatok elérésére külön időt kell biztosítani.

Az SSD (Solid State Disk) a flashmemóriák, azon belül is a NAND memóriák közé tartozik. Ebből adódón, a HDD-vel szemben, az SSD nem tartalmaz mozgó komponenst. Nincs benne írási/olvasási fej. Így az átlagos 0.0012 mp-es várakozás 10^{-9} nagyságrendre csökken. A kezdeti magas árak miatt inkább csak a nagyobb cégek érdeklődtek az SSD iránt jobban. Az eltelt évek során a gyártók annyira lecsökkentették az árakat, hogy már érdemes az SSD használatát fontolóra venni. Az SSD teljesítményét két faktor is jól mutatja. Az egyik a HDD-nél tapasztalt 200 MB/s-os írási/olvasási sebességet 500 MB/s-ra emelték. Ezen kívül a notebook-ok akkumulátorának az élettartalma is megnő, ha az operációs rendszert ezen tároljuk. Az előbbi kihasználva az operációs rendszereket érdemes erre a platformra telepíteni. Bármelyik operációs rendszerről elmondható, hogy a rendszerfelállítás és indítási idő lecsökken, viszont a Windows rendszer indítási ideje az ötödére csökkenthető ennek alkalmazásával. Ugyan ez az érték jelen van az SSD árában is, csak a másik irányban. A hagyományos merevlemezekhez képest jóval költségesebb a flash meghajtók beszerzése. Ezért érdemes megfontolni, hogy melyik lehetőséget választjuk az adatok tárolására.

Másik hátránya az SSD-nek a rövid élettartam. A HDD-be épített lemezekre történő írások száma szinte korlátlan (minimális lemezkopásra kell számítani az évek során, de ennek csekély a jelentősége). Ezzel szemben az SSD-ben található flash korlátos ilyen szempontból. Egy ilyen háttértár ötévente elhasználódik, valamint a maximális írási és olvasási korlát a százezres nagyságrendet éri el.

Egy köztes választás is lehetséges, amely a HDD és az SSD kombinációja, ez egyfajta hibrid megoldást ad. Ebben nem található külön SSD. A HDD tartalmaz egy kisebb, átlagosan 4 – 8 GB méretű SSD puffert, amelyen a rendszer működéséhez legfontosabb fájlokat tárolja. A módszer lényege, hogy ezen az SSD pufferen tárolt fájlok mellett, a rendszer statisztikát is készít. Így tudja bővíteni vagy rendszerezni a releváns rendszerfájlokat. A hibrid megoldásnál a sebesség elérheti az SSD-k sebességét, viszont a HDD-nál megszokott áramfogyasztási és zajfaktor megmarad. Másik jellemzője, hogy a hibrid megoldás csak a rendszer működésében jelent teljesítménynövekedést, viszont az adatok másolásánál ez nem érzékelhető.

Mindezeket figyelembe véve a videó feldolgozási feladatok esetén mérlegelni kell

a megfelelő adattárolási módot, mivel a fájlok írása és olvasása nagyban befolyásolhatja a program futási idejét. A megfelelő háttértár kiválasztását nagymértékben befolyásolja a feladat jellege. Ha a feladat számításigényes, érdemes az SSD mellett dönteni, hiszen az olvasási és írási sebessége megnöveli a rendszer teljesítményét a hagyományos merevlemezhez képest. A másik lehetőség, ha a feladat inkább a nagy adattárolást igényli, kisebb számítási feladatok megoldására. Ekkor a hagyományos merevlemezeket érdemes választani. Egyes esetekben a vegyes megoldás is optimális választás lehet. Az adatokat nagy kapacitású HDD-en tároljuk, míg a számítási feladatok SSD meghajtókon történnek.

4. fejezet

Eredmények

4.1. Motiváció

Az implementáció legfontosabb fázisa a tesztelés és az eredmények kiértékelése. A tesztek során a 2. fejezetben bemutatott matematikai modelleket és azok párhuzamos megvalósítását hasonlítjuk össze (elsősorban a Golub - Reinsch SVD és a Running SVD futási idejét tekintve). Elmondható, hogy a detektálás pontossága nem változott az egyes modellekben, így elsősorban a futási idő és a memória tekintetében végeztem méréseket. Az implementált algoritmushoz a Karlsruhe [21] videó adatbázist használtam fel. Az algoritmus képes szürkeárnyaltos, illetve háromcsatornás videofelvételek feldolgozására egyaránt. A teszteléshez felhasznált videók mpeg és avi kiterjesztésűek.

Mielőtt rátérnénk a megvalósítás egyes fázisainak tesztelésére illetve az elért eredmények bemutatására, konkretizálni kell több, fontos jellemzőjét a tesztelésnek. A megvalósított algoritmusok tesztelését három különböző konfiguráción végeztem el. A tesztelési környezetet mindegyik konfigurációnál hasonlóan alakítottam ki. Az algoritmus skálázhatóságát kihasználva egy nagyobb felbontású, 768×576 videó állományon, 10 frame/mp és 600 frame mellett történt az algoritmus tesztelése. Általánosan elmondható az eredmények kapcsán, hogy az egyes implementációk közti különbség a nagyobb frame feldolgozás esetén vált érezhetővé.

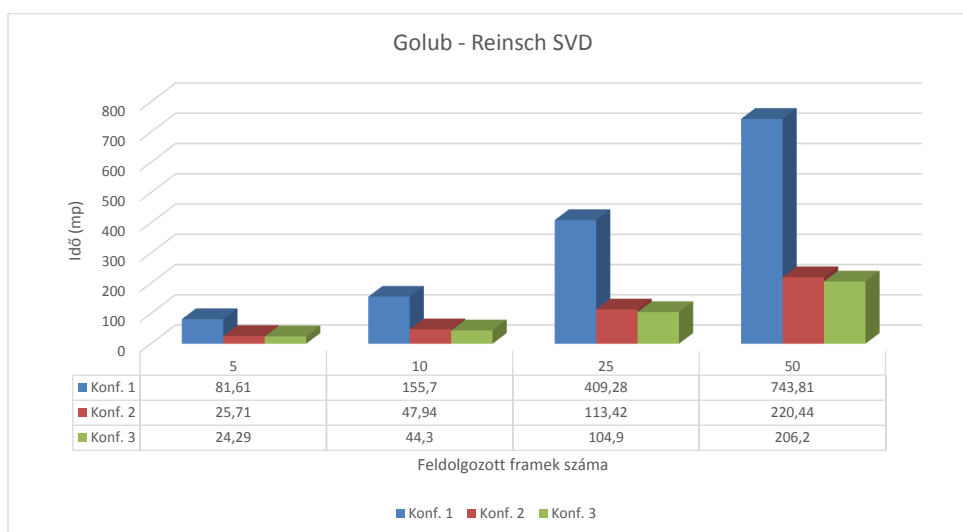
A konfigurációk felépítését a 4.1. táblázat részletezi. Megjegyezzük, hogy a harmadik konfiguráció, amitől a legjobb eredményt várjuk, abban egyedi, hogy a tesztelés Plextor N3-L2128 Sata SSD háttértáron történt.

	Konfiguráció 1	Konfiguráció 2	Konfiguráció 3
CPU	Intel Pentium Dual Core 2.16 GHz	Intel Core i3 2310M 2.1 GHz	Intel Core i5 4200U 1.6 GHz
GPU	NVIDIA GeForce 9500M 510 MB	NVIDIA GeForce GT 520M 1 GB	NVIDIA GeForce GT 730M 2GB
Memória	DDR 2 - 4 GB	DDR III - 4 GB	DDR III- 8 GB
Compute Capability	1.0	2.1	3.0
Évjárat	2009	2012	2014

4.1. táblázat. A teszteléshez használt konfigurációk

4.2. Golub - Reinsch SVD

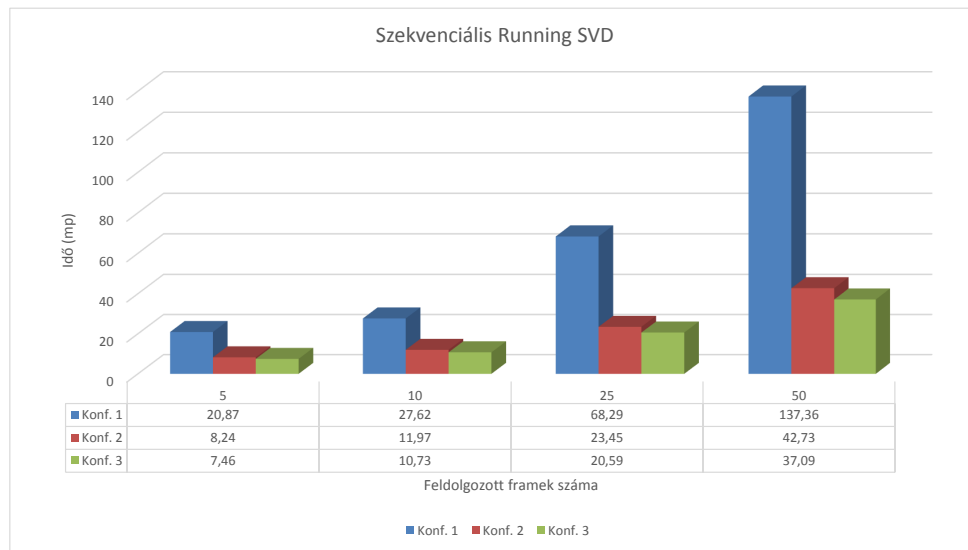
Ahogy azt a 2. fejezetben említettük, a mozgásdetektálást megvalósító módszerek közül a Golub - Reinsch SVD bizonyult a leglassabbnak mindegyik tesztkörnyezetben. A párhuzamos SVD esetében, a teszteléskor maximum 600 képkocka feldolgozását mértem le. Ezzel szemben, a Golub - Reinsch (GR) SVD-t csak maximum 50 frame feldolgozásával lehetett belátható időn belül lefuttatni. Az eredményekből tisztán látszik, hogy ez a módszer valós idejű feldolgozásra nem alkalmas. Ezzel ellentétben, a Running SVD két változata eredményesebbnek bizonyult. A két implementáció között releváns különbség nincs, viszont a kettőt külön összevetve a GR SVD-vel, már jelentős gyorsulás tapasztalható.



4.1. ábra. Golub - Reinsch SVD futási ideje.

4.3. Szekvenciális Running SVD

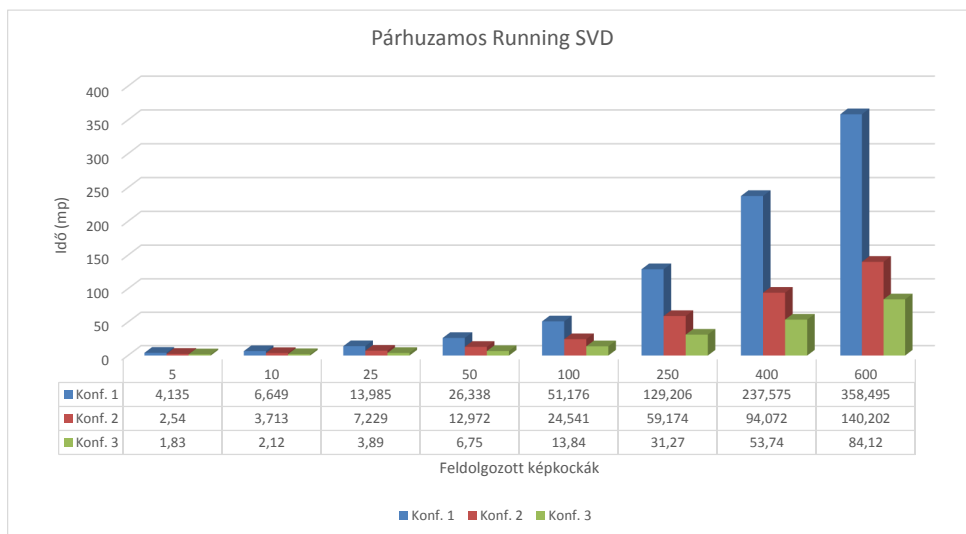
A következő a Running SVD szekvenciális változatának tesztelése volt. Az eredményekből az előző fejezethez hasonló következtetés vonható le, a konfigurációk sorrendje a futásidőt tekintve nem változott. Ebben az esetben sem volt értelme 50 iterációnál többet vizsgálni, hasonlóan a nagy futási idő miatt. Azonban összehasonlítva az előző módszerrel, még a leggyengébb konfiguráción történő tesztelés is átlagosan 4 – 6-szoros csökkenést eredményezett a program futási idejében.



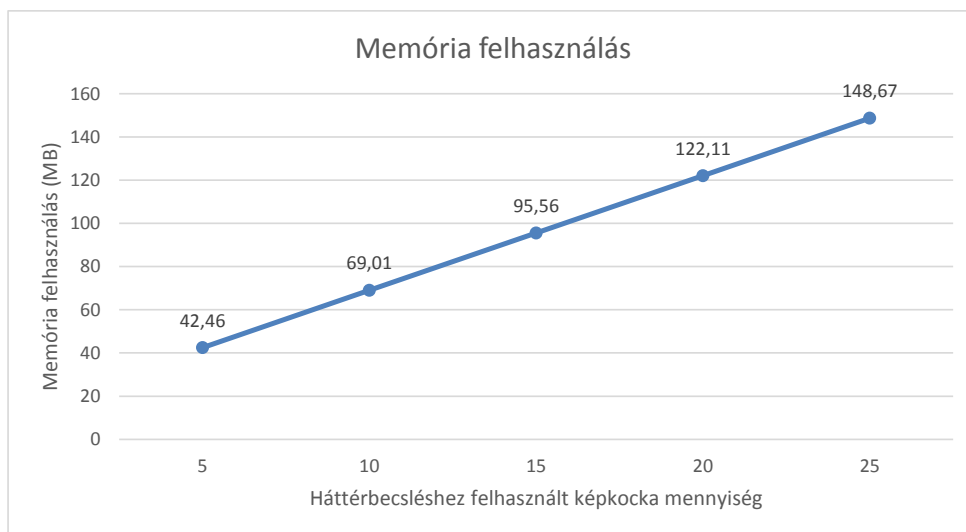
4.2. ábra. Szekvenciális Running SVD futási ideje.

4.4. Párhuzamos Running SVD

Végül a párhuzamos megvalósítás eredményét vizsgáljuk. A GR SVD-vel összehasonlítva, 50 frame esetén a leggyengébb konfiguráción 28, míg a legerősebben 34-szeres gyorsulást tudtunk elérni. A video feldolgozásának sebességét illetően, az egyes konfigurációk (konfigurációk szerint növekvő sorrendben) ~ 2 frame/mp, 4 frame/mp, és 7 frame/mp feldolgozást tudtunk elérni. Az értékeket a legnagyobb feldolgozott frame szám – 600 – és az adott algoritmus futási idejének hányadosából kapjuk. A legjobb eredmény még nem éri el a valós idejű feldolgozási lehetőséget, viszont a 6. fejezetben kitérek a további fejlesztési lépésekre, amelyekkel az offline feldolgozást valós idejű alkalmazásra lehet cserélni. Érdekességgéppen a párhuzamos változat memória igényét is megvizsgáltuk. A program a legkevesebb memóriát $m = 5$, a legtöbbet pedig $m = 25$ képkocka mellett igényli. A háttérbecsléshez használt frame mennyiség maximum $m = 25$, ezzel az értékkel pontosan lehet detektálni a mozgást. A felhasználtsági szintek között lineáris növekedés figyelhető meg.



4.3. ábra. Párhuzamos SVD futási ideje.



4.4. ábra. Párhuzamos SVD memória igénye.

4.5. Összefoglalás

A képfeldolgozáshoz implementált algoritmusok nem adnak minden esetben pontos eredményt, ezért a körberajzolt objektumok élei pontatlanok lehetnek. Ezen felül a teszteken olyan pontok, kisebb objektumok is láthatóak, amelyek nem mozognak. Ezek detektálása a küszöbérték növelésével vagy a háttér jobb becslésével eliminálható. Ugyanakkor elmondható, hogy egy olyan adaptív eljárást sikerült implementálni, mely automatikusan alkalmazkodik a fény és időjárási viszonyokhoz. Emberek által végzett mozgások detektálásának tesztelésekor az algoritmus pontosnak bizonyult, jól körvonalmazódtak az objektumok, kevés volt az ebből adódó anomália. Ezzel szemben egy traffipaxos felvételen, ahol több különböző méretű, esetleg távoli objektumok is szerepelnek, az algoritmus kisebb pontatlanságot eredményezett. A körvonalak néhol pontatlanok, illetve statikus objektumok is detektálva lettek.

Az implementált algoritmus pontossága mellett a valós idejű alkalmazhatóság is látható az eredményekből. A mért adatok alapján, a program futási ideje jelentősen csökkent az egyes fejlesztési lépések között. A kezdeti, a BSc-s szakdolgozatomban megvalósított szekvenciális Golub - Reinsch SVD alkalmatlannak bizonyult a valós idejű felhasználáshoz. A fejlesztés következő fázisa a Running SVD implementálása volt, amely egyszerre használta fel az előbb említett SVD-t, illetve a képkockák frissítési iterációs eljárását (Running SVD). A mért eredmények mellett elmondható, hogy a matematikai modell változtatásával közel valós idejű felhasználhatóságot kaptam.

Az implementáció utolsó lépése az így kialakított SVD párhuzamosítása volt. Mint minden párhuzamos algoritmusnál, itt is jelentős időbeli csökkenésre lehetett számítani. Az eredmények jól mutatják a videó gyors feldolgozását, a háttérbecsléshez használt nagyméretű frame mennyiség mellett is. Továbbá a párhuzamosítással elértem a 7 frame/mp-es feldolgozást, amely az eredeti, 2010-ben publikált cikk [6] futási idejéhez képest hatszoros gyorsulást jelent.

Az eredmények több konfiguráción lettek bemutatva, így nem csak az algoritmus, hanem a hardveres technológia fejlődését is jól szemléltetik.

5. fejezet

Felhasználói dokumentáció

A következőkben a TDK dolgozathoz elkészített program működésének menétét mutatom be, definiálom a kezdeti paraméterek listáját, tovább a futtatáshoz szükséges rendszerkövetelményeket is sorra veszem. A program indítása Windows operációs rendszeren lehetséges.

A program indításának két fontos előfeltétele van. Az első, hogy az implementációt olyan konfiguráció alatt futtassuk, amely támogatja a CUDA párhuzamosítást. Más szóval kizárólag NVIDIA GPU használata mellett futtatható az alkalmazás. Ennek hiánya rendszerüzenettel és program leállással jár. A futtatáshoz az alábbi DLL kiterjesztésű fájlok megléte szükséges:

cublas32_60.dll	cuda32_60.dll
cula_lapack.dll	opencv_ffmpeg249d.dll
opencv_imgproc249.dll	opencv_highgui249.dll
opencv_core249.dll	MSVCP110D.dll
libiomp5md.dll	MSVCR110D.dll

Az első három a CUDA, cuBLAS és a Cula programkönyvtárak bináris állományai. Ezt követi a képi és videó tartalom feldolgozáshoz használt OpenCV könyvtár. A maradék két fájlt a Windows és a fejlesztői környezet követeli meg.

A TDK dolgozathoz elkészített program konzolos alkalmazásként futtatható. A futáshoz szükséges bemenő paraméterek a következők:

- **Videó fájl** - Három vagy egysatornás, avi vagy mpeg kiterjesztésű videó állomány, amelyben a háttér statikus. A futáshoz ez utóbbi feltétel nem szükséges, viszont a feladat az ilyen videók feldolgozására készült, ezért a program eredményét befolyásolja a bemeneti videó fájl ezen tulajdonsága.
- **Háttér framek száma** - A 2. fejezetben bemutatott háttérbecslési folyamat meghatározó értéke. Minél több frame alkotja a hátteret, annál stabilabb

lesz az algoritmus és a detektálás során a mozgó objektumok körvonalai is részletesebben rajzolódnak ki. A dolgozat kiértékelésekor 20 háttérframe-et használtam, amely már biztosítja a pontos detektálást.

- **Feldolgozandó framek száma** - A háttérbecslés után feldolgozandó képkockák számát adja meg, amely nem lehet nagyobb, mint a videót alkotó framek száma. Megadja a program iterációinak a számát.
- **Küszöbérték** - A 2.4 alfejezetben bemutatott küszöbölési eljáráshoz használt küszöbérték. A bináris kép generálásánál van nagy szerepe, értékét az $[1, 255]$ intervallumból veheti fel.

A rendszerkövetelmények biztosítása és a bemeneti paraméterek helyes megadása után a program az `ApproxSVD_CULA.exe` állomány futtatásával indul és megadott képkocka mennyiségű iterációt hajt végre. A feldolgozás eredményét, a detektált képeket a `CulaSVDFrames` mappába menti ki. Az így megkapott kimeneti fájlokat, például VirtualDub programmal egyesíteni lehet egy videó állományba.

6. fejezet

Fejlesztési lehetőségek

Az egyes konfigurációkon mért eredmények jól mutatják a hardveres fejlődést, illetve a matematikai modellek használhatóságát.

Tanulmányaim során szeretném a programot további opciókkal bővíteni, valamint a fejlesztési területen még nem alkalmazott módszereket implementálni. A bevezetőben részletesen bemutattam, hogy a számítógépes látás számtalan lehetőséget rejt nem csak a hétköznapiakban, de akár a tudományos életben is. Ezért szándékomban áll további fejlesztések implementálása. Ezzel kapcsolatban néhány ötletet fogok felvázolni, melyeket MSc tanulmányaim során diplomamunka keretein belül tervezek megvalósítani.

A CUDA GPU-s párhuzamosítás esetén, a GPU memóriájába bemásolt adat nem érhető el közvetlenül. Ebből kifolyólag minden olyan esetben szükséges egy CPU - GPU memória másolás az adatok közvetlen eléréséért, amely a futási időt növeli. A 3. fejezetben említettük a CUDA Thrust könyvtárát. Ennek segítségével az adatok közvetlen elérhetőek és módosíthatóak a GPU memóriájában. A program ilyen jellegű fejlesztése nagyobb munkát igényel, mivel a Thrust egy saját adatszerkezettel támogatja a párhuzamos feladatok implementálását.

Egy másik lehetőség az összes detektált, vagy egy opcionálisan kinevezett objektum, követése, esetleg ezek azonosítása, megjelölése, illetve a feldolgozás során történő vizuális megjelenítése. A feladat sokkal komplexebb, mint azt elsőre gondolnánk. Hiszen például az objektumok takarhatják egymást, elhaladhatnak a háttér egyes részei előtt vagy mögött is. Ebben az esetben már csak két vagy több különböző objektumként detektálhatóak, melyeket regisztrálni kell. A további fejlesztésekkel e problémák kezelése az elsődleges célom.

Az előző ötletből kiindulva megszámozhatjuk a képen az egybetartozó objektumokat is. Gyakorlati példaként említhetjük a traffipaxos felvételeket, vagy akár egy focimeccset, ahol statisztikai adatokat számolnak a kamerák. Az objektum azonosítás megvalósítása után ez a fejlesztési lehetőség már kevés időt venne igénybe.

Célom elsősorban az olyan algoritmusok implementálása, melyeknek alapja valamilyen életszerű, gyakorlati probléma. Mivel az ilyen módszerek főképp ipari alkalmazásokban fordulnak elő, ezért megismerésük későbbi karrierem során is hasznos lehet.

Függelék

Algorithmus 1 SVD Update ([6])

```
function [_U, _S, _V1] = svdUpdate(U, S, V1, a)
N = size(V1)(1);
M = size(U)(1);
D = S(:, 1 : M);
z1 = V1' * a;
w = a - V1 * z1;
[v, eta] = qr(w, 0);
L = [D, zeros(M, 1); z1', eta];
[Q, Omega, W] = svd(L);
_U = [U, zeros(M, 1); zeros(1, M), 1];
_S = Omega;
_V1 = [V1, v] * W;
```

Algorithmus 2 SVD Downdate ([6])

```
function [_U, _S, _V] = svdDowndate(U, S, V)
N = size(V)(1);
M = size(U)(1);
mu = U(M, M);
if mu < 0 then
    U = -1 * U;
    V = -1 * V;
    mu = -1 * mu;
end
D = S(:, 1 : M);
U_11 = S(M, M);
x = U(1 : M - 1, 1 : M - 1);
u1 = U(M, 1 : M - 1);
MM = eye(M - 1) - (1/(1 + mu)) * u1 * u1';
X = U_11 * MM - x * u1';
C = [MM, -u1] * D;
[Q, Omega, W] = svd(C);
Omega = Omega(:, 1 : M - 1);
_U = X * Q;
_S = Omega;
_V = (V * W)(:, 1 : M - 1);
```

Irodalomjegyzék

- [1] Widyawan, M. I. Zul, and L. E. Nugroho, „Adaptive motion detection algorithm using frame differences and dynamic template matching method,” *The 9th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pp. 236 – 239, 2012.
- [2] Á. Botos, *Numerikus módszerek alkalmazásai a Számítógépes látás területén*, Bsc thesis, Eötvös Lóránd Tudományegyetem, Budapest, Hungary, 2012.
- [3] F. Kahl, R. Hartley, and V. Hilsensteen, „Novelty detection in image sequence with dynamic background,” *Springer LNCS*, vol. 3247, pp. 117–128, 2004.
- [4] A. Monnet, A. Mittal, N. Paragios, and V. Ramesh, „Background modeling and subtraction of dynamic scenes,” *Proceedings of International Conference on Computer Vision*, vol. 2, pp. 1305–1312, 2003.
- [5] M. Gu, *Studies in numerical linear algebra*, Ph.d. thesis, Yale University, Connecticut, USA, 1993.
- [6] D. Csetverikov and A. Axt, „Approximation-free running svd and its application to motion detection,” *Pattern Recognition Letters*, vol. 31, pp. 891–897, 2010.
- [7] I. T. Jolliffe, *Principal component analysis*, Springer Series in Statistics. Springer-Verlag, New York, USA, 2nd edition, 2002.
- [8] K. R. Rao, *The Transform and Data Compression Handbook*, CRC Press, New York, USA, 2001.
- [9] B. Gary and K. Adrian, *Learning OpenCV*, O’Reilly Media, Inc., Sebastopol, USA, 2nd edition, 2008.
- [10] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The John Hopkins University Press, London, UK, 3rd edition, 1996.
- [11] R. Liu and T. Tan, „A svd-based watermarking scheme for protecting rightful ownership,” *IEEE Transactions on Multimedia*, vol. 4, pp. 121–128, 2002.

- [12] L. Gergó, *Numerikus módszerek: Kidolgozott példák - feladatok (Egyetemi jegyzet)*, Eötvös Loránd Tudományegyetem, Informatikai Kar, Digitális Könyvtár, 2010.
- [13] NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, NVIDIA Corporation, 2007.
- [14] G. Bradski, „The opencv library,” *Dr. Dobb's Journal of Software Tools*, 2000.
- [15] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, Morgan Kaufmann, Massachusetts, USA, 1st edition, 2013.
- [16] M. McCool, J. Reinders, and A. Robinson, *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann, Massachusetts, USA, 1st edition, 2012.
- [17] J. Hoberock and N. Bell, „Thrust: A parallel template library 1.7.0,” 2010, Available: <http://thrust.github.io>.
- [18] NVIDIA Corporation, California, USA, *Cublas library 6.0*, 2014, [Online], Available: <http://docs.nvidia.com/cuda/cublas>.
- [19] EM Photonics, Inc., *CULA Reference Manual R17*, 2014, [Online], Available: <http://www.culatools.com>.
- [20] P. Estival and L. Giraud, „Performance and accuracy of the matrix multiplication routines : CUBLAS on NVIDIA Tesla versus MKL and ATLAS on Intel Nehalem,” Tech. Rep., 2010, Available: <https://hal.inria.fr/>.
- [21] Institute of Algorithms University of Karlsruhe and Cognitive Systems, „Image Sequence Server,” http://i21www.ira.uka.de/image_sequences, 1998.