XI. Erdélyi Tudományos Diákköri Konferencia

Temporális videótömörítési modell GPU-n

Szerző:
Áfra Attila Tamás

Babeş-Bolyai Tudományegyetem
Matematika és Informatika Kar
Informatika szak, 3. évfolyam

Témavezető:
Dr. Darvay Zsolt

Babeş-Bolyai Tudományegyetem
Matematika és Informatika Kar
Programozási Nyelvek és Módszerek Tanszék

Kolozsvár, 2008. május 23-24.

Kivonat

A videótömörítési algoritmusok több különböző modell segítségével valósítják meg a hatékony kódolást. Ezek közül a temporális modell a legtöbb számítást igénylő, ezért kulcsfontosságú, hogy ez a lehető leghatékonyabban legyen implementálva. Általában a teljes tömörítési folyamatot a CPU végzi el, amely egy jó teljesítményt nyújtó, általános célú számítási egység, viszont nem tudja sebességben megközelíteni a párhuzamos feldolgozásra specializált GPU-t. A GPU eredetileg 3D-s grafikai számításokra lett tervezve, viszont az utóbbi pár év során egyéb problémák megoldására is alkalmassá vált. A dolgozat egy olyan speciális temporális videótömörítési algoritmust vezet be, amely a CPU helyett a GPU-ra van tervezve, így képes kihasználni annak hatalmas számítási kapacitását. A jelentős architekturális különbségek miatt ez az algoritmus nagymértékben eltér a tradicionálisaktól.

1. Bevezető

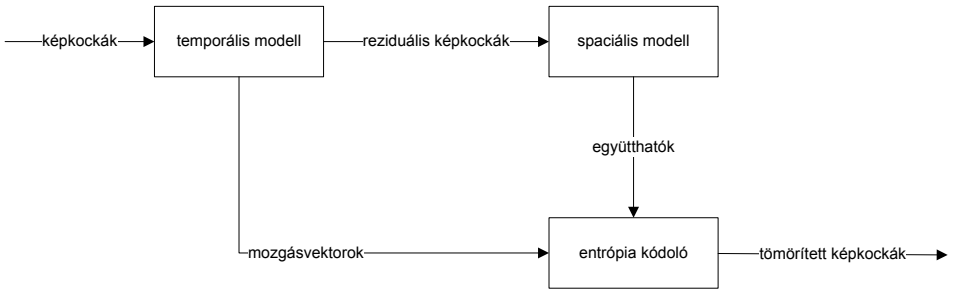
1.1. Videótömörítés

A videók hatékony kódolását nem lehet megvalósítani általános tömörítési algoritmusok segítségével. Ezért erre a célra számos speciális módszert és standardot fejlesztettek ki.

Egy videótömörítési rendszer két fő komponensből áll: egy kódolóból és egy dekódolóból. Ezt a rendszert CODEC-nek nevezzük (COder/DECoder). A kódoló megpróbálja csökkenteni a redundanciát a bemeneti digitális videó jelben, és a kapott adatokat összetömöríti. A dekódolónak ugyanezeket a műveleteket kell elvégeznie fordított sorrendben. A dekódolt videó az eredeti másolata, vagy annak egy megközelítése. Ha a kapott videó bitre pontosan megegyezik az eredetivel, akkor veszteségmentes tömörítésről beszélünk; ha különbözik attól, akkor veszteségesről.

A CODEC az eredeti videószekvenciát egy *modell* segítségével reprezentálja (egy hatékony kódolt reprezentáció, amelyet felhasználva rekonstruálni lehet az eredeti adatokat vagy azoknak megközelítését). A cél, hogy a modell minimális mennyiségű bitet használjon, és ugyanakkor maximális minőséget nyújtson (amennyiben veszteséges a tömörítés).

Egy videó kódoló három fő komponensből áll (1. ábra): egy *temporális modell*ből, egy *spaciális modell*ből és egy *entrópia kódoló*ból [3].



1. ábra. Egy videókódoló blokkdiagramja

A temporális modell bemenetként egy tömörítetlen videószekvenciát kap. Ez a modell megpróbálja csökkenteni a temporális redundanciát, kihasználva a szomszédos képkockák közti hasonlóságot. Általában két egymás utáni képkocka között kicsi a különbség, mivel időben közel állnak egymáshoz, így a mozgás minimális. Ennek a tulajdonságnak köszönhetően egy képkockát bizonyos mértékben meg lehet *jósolni* az előző (vagy akár jövőbeli) képkockák alapján. A modell kimenete egy *reziduális képkocka* és egy paraméterhalmaz (általában *mozgásvektor*okból áll), amely a jóslást írja le. A reziduális képkocka a jósolt és a tényleges képkocka különbsége, tehát a jóslási hibákat tartalmazza.

A spaciális modell bemenetét az említett reziduális képkocka képezi. Ez a modell a redundancia csökkentése érdekében a szomszédos minták közötti hasonlóságot használja ki, amit úgy valósít meg, hogy először transzformálja a képkockát (pl. DCT vagy wavelet transzformációt alkalmazva), és utána kvantálja a kapott együtthatókat. A kvantálás mértéke határozza meg elsősorban a minőséget, és így a tömörítés erősségét is. A modell kimenete a kvantált együtthatók összessége.

A temporális modell paramétereit és a kvantált együtthatókat egy entrópia kódoló tömöríti össze. Ez a statisztikai redundanciát minimalizálja (pl. a gyakran előforduló mozgásvektorokhoz vagy együtthatókhoz rövid bináris kódokat rendel). A kapott adatokat el lehet tárolni vagy lehet továbbítani, és utána a dekódoló segítségével meg lehet kapni az eredeti videószekvencia közelítését.

Az említett komponensek közül a legtöbb számítást igénylő a temporális modell, mivel ez minél pontosabban meg kell határozza a képkockák közötti hasonlóságokat. Ezért kulcsfontosságú, hogy ez a komponens a lehető leghatékonyabban legyen implementálva. A temporális modellnél használt algoritmusok párhuzamosíthatóak, így nagyobb teljesítmény lehet elérni, ha a célhardver támogat párhuzamos feldolgozást.

Manapság a CPU-k jelentős része rendelkezik több maggal és vektoriális feldolgozási egységekkel is, melyeket kiaknázva fel lehet gyorsítani a párhuzamos algoritmusokat. Ennél viszont sokkal nagyobb teljesítménynövekedést lehetne elérni, ha ezeket az algoritmusokat egy teljesen párhuzamos architektúrájú feldolgozó egységre implementálnánk. A legnépszerűbb és legolcsóbb ilyen hardver a GPU, amely csak az utóbbi pár évben vált alkalmassá ilyen feladatok megoldására. Ezért egyelőre nagyon kevés GPU-ra tervezett algoritmus létezik, és ezek között nagyon kevés videótömörítéssel kapcsolatos módszer van. A továbbiakban ennek a hiányosságnak a kiküszöböléséről lesz szó.

1.2. GPU programozás

A GPU (*graphics processing unit*) egy olyan speciális feldolgozó egység, amely valós idejű 2D-s és 3D-s grafikai számításokra lett tervezve. Általában videókártyákra vagy alaplapi chipkészletekbe van integrálva, és gyakran saját memóriával rendelkezik (*videómemória*).

A grafikai műveletek nagyon sok, relatív egyszerű, egymástól független számításokból állnak, így ezeket párhuzamosan is el lehet végezni. A GPU a lehető legjobb teljesítmény érdekében pontosan így jár el. A CPU-val ellentétben nagyon sok, egyszerű feldolgozó egységgel rendelkezik. Így a rendelkezésre álló tranzisztorokat hatékonyabban ki lehet használni. Ennek viszont az a következménye, hogy a GPU műveletvégző képesség szempontjából jóval korlátozottabb, mint a CPU.

A GPU nem tud önmagában működni: szükség van egy CPU-ra, amely vezérli azt. Ezt speciális grafikai API-k segítségével lehet megvalósítani. Két ilyen API létezik: az *OpenGL* és a *Direct3D*. Ezek utóbbi pár verziói már olyannyira szabadon programozhatóak, hogy akár grafikától eltérő feladatokat is meg lehet oldani velük. Ettől függetlenül a megoldandó problémát muszáj visszavezetni egy grafikai problémára, ami az esetek többségében nem könnyű feladat. Ezt a programozási technikát *GPGPU*-nak (*general-purpose computing on graphics processing units*) nevezzük [2]. Szerencsére az API által nyújtott funkcionalitásoknak csak egy töredékére van szükség.

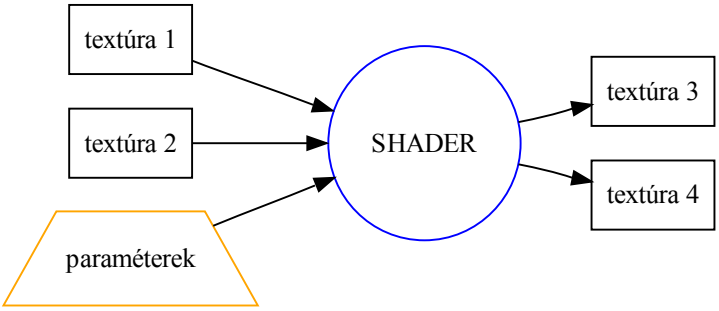
A GPU több fajta feldolgozásra képes: *vertex*-, *primitív*- és *pixel*feldolgozásra. Az adatok *erőforrás*okban vannak tárolva, a számolásokat pedig a *shader* programok hajtják végre. Az erőforrások ideális esetben a videómemóriában vannak tárolva, így a GPU-nak gyors hozzáférése lehet azokhoz. A shadereket általában egy C-szerű nyelvben kell megírni. GPGPU esetében elsősorban csak a pixelfeldolgozásra van szükség, mivel abban az esetben a bemeneti és kimeneti adatok egyaránt többdimenziós tömbök, amelyekkel könnyedén meg lehet oldani a legtöbb párhuzamos problémát.

A GPU a feldolgozandó pixeladatokat *textúrák*ban tárolja, a feldolgozást pedig a *pixel shader*ek végzik.

A textúrák lehetnek egy-, két- vagy háromdimenziósak, és rendelkeznek egy *pixelformátum*mal. A pixelformátum meghatározza, hogy a textúrában lévő pixeleknek (*texel*ek) milyen komponenseik vannak. Négy fajta komponens létezik: a három alapszín (*R*, *G* és *B*) és az *alpha* (*A*). A leggyakrabban használt formátum az *RGBA8*, amely mindegyik fajta komponenst tartalmazza, és ezek *jelöletlen 8-bites egész* típusúak.

A shaderek elsősorban lebegőpontos számításokra lettek tervezve, ezért a be- és kimeneti egész típusú pixelértékek a shadereken belül 0 és 1 közötti lebegőpontos értékekre konvertálódnak.

A pixel shader azt határozza meg, hogy *egyetlen* kirajzolandó pixel milyen értéket kapjon. A



2. ábra. Egy pixel shader be- és kimenetei

GPU egy primitív kirajzolása során minden egyes pixelre meghívja ezt a shadert. A shader bemenetei paraméterek és textúrák lehetnek, kimenetei pedig – GPGPU esetében – textúrák (2. ábra). A bemeneti textúrákból tetszőlegesen tud olvasni, viszont a kimeneti textúrák esetében csak az aktuális pixelnek megfelelő pozícióba írhat. Ugyanakkor a kimeneti textúrák kétdimenziósak vagy háromdimenziós textúrák kétdimenziós szeletei kell legyenek, és nem lehetnek egyszerre bemeneti textúrák is. Továbbá ezek a textúrák méretben és formátumban is meg kell egyezzenek. Általában csak az RGBA formátumú textúrákat lehet használni kimenetként.

A GPU-val tehát olyan kétdimenziós tömböket lehet generálni, melyeknek elemeit ugyanazzal a kóddal lehet egymástól független, párhuzamosan kiszámolni. Ez a számolási módszer a funkcionális nyelvekben megtalálható *map* függvényre emlékeztet [2]. A generálást egy rajzolási függvényhívással lehet előidézni. Az esetek döntő többségében egy olyan téglalapot kell rajzolni, amely teljesen lefedi a kimeneti textúrákat, így azoknak minden eleme tud módosulni a shader által. Annak érdekében, hogy a kiszámolt tömbelemek értékei különbözőek lehessenek, a shaderek hozzáférhetnek a kimeneti koordinátákhoz, és azokat figyelembe vehetik a számítások során.

Általában egy GPU-ra tervezett algoritmust nem lehet egyetlen tömbgenerálással megoldani, így az algoritmust fázisokra kell bontani. Minden számolási fázis egy tömböt generál ki egy shader segítségével.

A számolási fázisok előtt ajánlott létrehozni az összes szükséges textúrát. Ennek két előnye van. Az egyik az, hogy az időigényes memóriaallokálások nem fogják lassítani a számítási folyamatot. A másik előny akkor mutatkozik meg, amikor sokszor, különböző adatokkal szeretnénk lefuttatni az egész műveletsorozatot (*streaming*). Mivel az elvégzendő számítások nem változnak, az előre létrehozott textúrákat minden futtatásnál fel lehet használni változatlanul.

Ha szükség van bemeneti tömbökre, azokat természetesen futtatás előtt fel kell tölteni a videómemóriába. Futtatás közben vagy az után a végső kimeneti tömböket vissza kell másolni a rendszermemóriába. A videómemória és rendszermemória közti másolások időigényesek, ezért a lehető legnagyobb mértékben csökkenteni kell a két memóriaegység közti adatforgalmat.

2. Temporális modell

2.1. Egyszerű temporális jóslás

A temporális jóslást legegyszerűbben úgy valósíthatjuk meg, hogy a jósolt képkockát az előző képkockával egyenlőnek vesszük, azaz azt feltételezzük, hogy nincs mozgás az előző képkockához képest. Így a reziduális képkocka egyenlő az aktuális és előző képkocka különbségével.

(a) Előző képkocka (b) Aktuális képkocka (c) Reziduális képkocka

3. ábra. Egyszerű temporális jóslás

A 3. ábrán meg lehet figyelni ezt a jóslási módszert. A reziduális képkockát ábrázoló képen a középszürke árnyalat a 0 különbséget jelzi, a világosabb és sötétebb árnyalatok pedig a pozitív és negatív különbségeket. Ezen az ábrán a 0 közeli különbségek dominálnak, amelyek azt jelzik, hogy az illető zónákban csak minimális mozgás van. Így a reziduális képkocka sokkal jobban tömöríthető, mint az eredeti. Ahol viszont már egy kicsit nagyobb mozgás van, ott elég nagy különbségeket lehet tapasztalni.

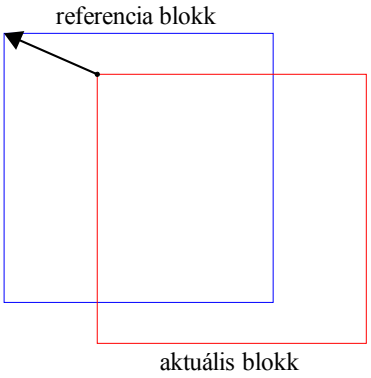
A valóságban elég gyakoriak az olyan képzónák, melyek főleg síkban mozognak, és pixelértékeik alig változnak. Ahhoz, hogy ezt a jelenséget ki lehessen aknázni tömörítés során, egy olyan jóslási algoritmusra van szükség, amely megpróbálja beazonosítani ezeket a zónákat, és meghatározni mozgásuk irányát.

2.2. Mozgásbecslés

A *mozgásbecslés* és *mozgáskompenzáció* együttese a jósolt képkockák előállításának egyik leghatékonyabb és legnépszerűbb módja.

A *blokkalapú mozgásbecslés* abból áll, hogy fel kell osztani az aktuális képkockát régiókra, és meg kell határozni, hogy ezek a *referencia képkockában* (egy már kódolt, az aktuálissal szomszédos képkocka) hova vannak elmozdulva. Általában ezen régiók egymást nem fedő téglalapok. Ezeket *blokk*oknak nevezzük. Minden blokkhoz hozzá kell rendelni egy *mozgásvektor*t (4. ábra), amely megmutatja a *relatív* elmozdulását pixelekben mérve. A mozgásvektor egy olyan tetszőleges

pozíciójú *referencia blokk*ra kell mutasson a referencia képkockán belül, amely a lehető legjobban hasonlít az aktuális blokkra. Ennek a régiónak a mérete természetesen meg kell egyezzen az aktuális blokkéval. Mozgásbecslést csak a videókódolónak kell végeznie.



4. ábra. Mozgásvektor

A mozgáskompenzáció a blokkfelosztást és a mozgásvektorokat felhasználva felépíti a jósolt képkockát. Ezt egyszerű és gyors blokkmásolásokkal meg lehet oldani. Mozgáskompenzációt egyaránt a kódoló és a dekódoló is kell alkalmazzon, mivel a jósolt képkockára mindkét félnek szüksége van.

A mozgásbecslés egy nehéz és számításintenzív feladat, mivel a mozgásvektorokat csak kereséssel lehet meghatározni. Ideálisan az aktuális blokkot össze kellene hasonlítani az összes létező referencia blokkal, viszont ez annyira sok számítást igényel, hogy a gyakorlatban nem használható. A megoldás tehát az, hogy szűkítenünk kell a keresést, ezzel viszont elveszítjük az optimalitást.

16

(1, 1)	(1, 2)	(1, 3)	(1, 4)
(2, 1)	(2, 2)	(2, 3)	(2, 4)
(3, 1)	(3, 2)	(3, 3)	(3, 4)
(4, 1)	(4, 2)	(4, 3)	(4, 4)

16

5. ábra. Makroblokk

Hatalmas teljesítménynövekedést érhetünk el, ha csak egy adott méretű téglalap/négyzet alakú régión belül keresünk. Ezt *keresési tartomány*nak nevezzük. Általában ennek középpontja az aktuális blokk pozíciójában van. Mivel általában a legjobb találatok a blokk szűk környezetében vannak, ezzel a módszerrel csak nagyon ritkán nem lehet megtalálni a legjobb mozgásvektort. Az eseteket döntő többségében ez a régió nem kell nagyobb legyen, mint 64x64 pixel. Ha a keresési

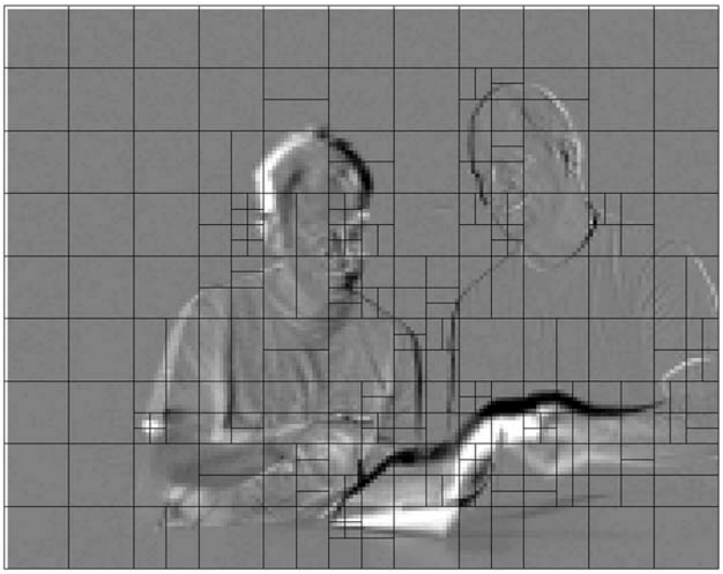
tartományon belül minden lehetséges pozíciót kipróbálunk, *kimerítő keresés*ről (*ESA*, azaz *exhaustive search algorithm*) beszélünk. Léteznek gyorsabb (heurisztikus) algoritmusok is, amelyek nem próbálják ki az összes lehetőséget, viszont így ritkábban találják meg a legjobb mozgásvektorokat.

Egy fontos tényező, ami jelentősen befolyásolja a tömörítés hatékonyságát: a blokkméret. Ha a blokkok túl nagyok, akkor a mozgásokat nehéz megjósolni, mivel kisebb zónák egyedi mozgását nem lehet reprezentálni. Ha túl kicsik, akkor a jóslás nagyon jó, viszont a rengeteg mozgásvektor eltárolásával elveszítjük ezt az előnyt. A tapasztalat azt mutatja, hogy a legnagyobb használható blokkméret 16x16 pixel, a legkisebb pedig 4x4 pixel. A 6. ábrán jól látható, hogy különböző blokkméretű mozgásbecslések hogyan befolyásolják a reziduális értékek nagyságát.

(a) Mozgásbecslés nélkül (b) Mozgásbecsléssel (16x16) (c) Mozgásbecsléssel (4x4)

6. ábra. Reziduális képkockák különböző jóslásokkal

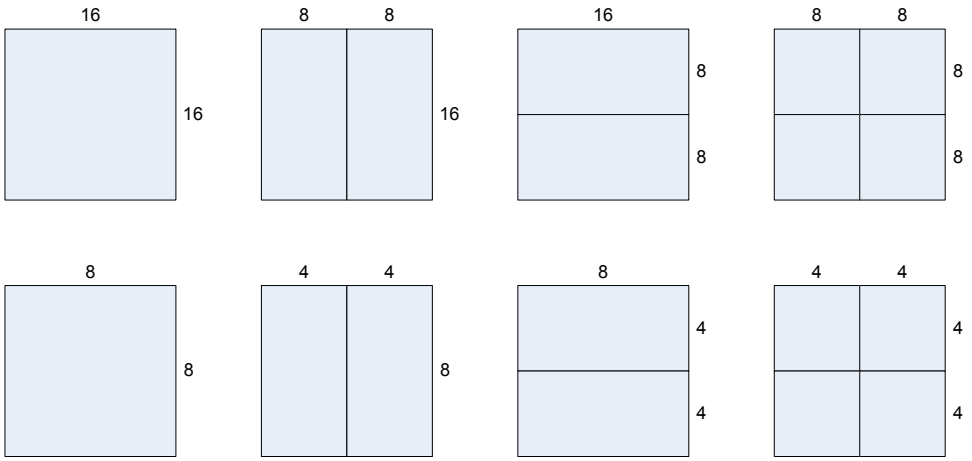
Egyes videótömörítési standardok konstans méretű blokkokat használnak (pl. MPEG-2), viszont az újabbak szinte kivétel nélkül változó méretűeket is megengednek. Így egyaránt egyszerű és komplex mozgásokat is hatékonyan meg lehet jósolni.



7. ábra. Reziduális képkocka (mozgásbecslés nélkül) egy lehetséges blokkfelosztással

Jelenleg a legtöbb blokkméret konfigurációt a *H.264* standard engedi meg. A legnagyobb blok-

kok 16x16 pixelből állnak, amelyeket a kódoló tetszés szerint feloszthat, particionálhat kisebb blokkokra [3] (pl. a 7. ábrán látható módon). Az ilyen blokkokat *makroblokk*oknak (5. ábra) hívjuk. A megengedett blokkméretek a következők: 16x16, 8x16, 16x8, 8x8, 4x8, 8x4 és 4x4 (8. ábra). Az ideális particionálás megtalálása nehéz feladat, és ezt csak akkor lehet megvalósítani, ha az összes particionálás esetén megkeressük a legjobb mozgásvektorokat. Ha ezek a kódoló rendelkezésére állnak, akkor el tudja dönteni, hogy mennyire érdemes felaprózni a blokkokat.



8. ábra. H.264 blokkfelosztások

Ahhoz, hogy meg tudjuk találni a legjobb mozgásvektort, szükségünk van egy hibafüggvényre, amely segítségével meg lehet mérni két pixelrégió hasonlóságát. Ha két régió megegyezik, a hibafüggvény értéke 0 kell legyen. Tulajdonképpen keresés során ezt a függvényt kell minimalizálni. Számos ilyen hibafüggvény létezik, de videótömörítésnél messze a legelterjedtebb az *SAD* (*sum of absolute differences*), ami a pixelkülönbségek abszolút értékeinek összegével egyenlő:

$$SAD\,(A,B) = \sum_{i=1}^{M}\sum_{j=1}^{N} |A_{ij} - B_{ij}| \tag{1}$$

3. Mozgásbecslés GPU-n

3.1. Vázlatos felépítés

A továbbiakban egy új, GPU-ra tervezett mozgásbecslési algoritmusról lesz szó. Mivel ez a komponens minden videótömörítési standard esetében egymástól kicsit eltérő követelményekkel kell rendelkezzen, a bevezetendő algoritmus a legkomplexebb standardra van specializálva: a H.264-re, amely mellesleg egyike a legfontosabb standardoknak. Így az algoritmust könnyen lehet adaptálni más standardokra is, mivel csak egyszerűsítésekre van szükség.

Keresési stratégiaként az ESA-t választottuk. Ez a létező leglassabb módszer, viszont egye-

dül ez vezet optimális megoldáshoz (ha eltekintünk attól, hogy csak egy adott tartományon belül keresünk). Ez elsősorban professzionális környezetben lényeges, ezért a szóban forgó algoritmus elsősorban azt a szegmenset célozza meg. Ugyanakkor, a gyorsabb módszerekkel ellentétben, az ESA nagymértékben párhuzamosítható, így a GPU hatalmas számítási ereje kompenzálja az ESA eredendő lassúságát.

Az algoritmus bemenetként két képkockát kap: az aktuálisat és egy referenciát. Az egyszerűség kedvéért csak *luma* (fényerő) csatornájú képkockákról lesz szó; a *chroma* (szín) csatornákra való kibővítés triviális.

A kimenet két fajta adatból áll: a legjobb mozgásvektorokból és a hozzájuk tartozó hibaértékekből. A mozgásvektorokat az összes lehetséges blokkparticionálás esetében meg kell találni.

A be- és kimeneti adatok textúrák formájában vannak tárolva. Ezeken kívül szükség van temporális textúrákra is.

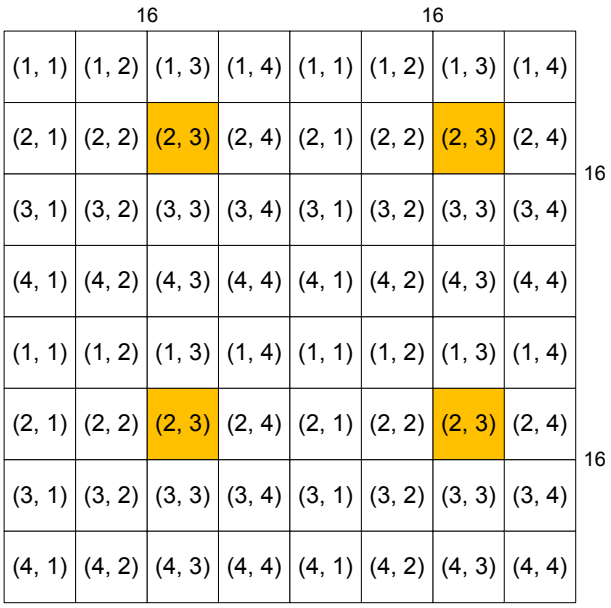
Az algoritmust két fő részre lehet bontani. Az egyik a CPU-n futó *vezérlő komponens*, amely a GPU inicializálásáért, az erőforrások kezeléséért és a számítási/rajzolási műveletek meghívásáért felelős. A másik a shaderekből áll, amelyek a tulajdonképpeni számításokat végzik el. Az algoritmus három shadert foglal magában.

3.2. SAD shader

Az *SAD shader* feladata az, hogy a bemenetként megadott aktuális- és referencia képkockák alapján számolja ki az összes lehetséges mozgásvektornak megfelelő hibaértéket. Figyelembe kell vegyük, hogy ezekre az összes blokkméret esetében szükségünk van. A hibafüggvény additív tulajdonságának köszönhetően viszont nem kell minden esetben elvégezni ezt a műveletet: elegendő csak a legkisebb blokkméretre, azaz a 4x4-esre elvégezni. Ha rendelkezésünkre áll két pixelrégió hibaértéke, és meg szeretnénk kapni az egyesített régió hibaértékét, akkor egyszerűen csak össze kell adnunk a két értéket. Mivel a nagyobb blokkokat össze lehet rakni 4x4-es blokkokból, összeadásokkal könnyedén meg tudjuk kapni az összes blokkméretnek megfelelő hibaértékeket. Ezt a műveletet nevezzük el *blokkfúzió*nak. Fontos, hogy mindig csak azonos relatív mozgásvektornak megfelelő hibákat adhatunk össze. Az SAD shader tehát csak 4x4-es blokkokkal kell dolgozzon, így lehet speciális optimizálásokat alkalmazni.

A GPU csak akkor tudja jól párhuzamosítani a számításokat, ha elegendően nagy méretű textúrákkal dolgozik. Minden blokkhoz annyi hibaérték tartozik, ahány mozgásvektor van. A keresési tartomány mindegyik pixeléhez tartozik egy mozgásvektor, tehát a hibaértékek száma egyenlő a keresési tartomány pixeleinek számával. Mivel ez a szám nem túl nagy (pl. 32x32 pixel), és a blokkok

száma viszont igen, az a praktikus, ha egyetlen textúrában tároljuk el az összes blokk hibaértékeit: a *hibatextúrában*. Így egyetlen rajzolási hívással ki tudjuk számolni az összes értéket. Ebben az esetben a hibatextúrát fel kell osztani annyi téglalap alakú részre, ahány blokk van. Ezeket nevezzük el *hibablokk*oknak. A hibablokkok az illető blokkokhoz tartozó hibaértékeket tartalmazzák, tehát akkorák, amekkora a keresési tartomány mérete. További előny az, hogy egyetlen blokkfúziós művelet során nem csak különálló blokkokat lehet egyesíteni, hanem egész *blokkcsoport*okat. Egy blokkcsoport azon blokkok összessége, amelyek a saját makroblokkjukon belül azonos indexűek (9. ábra).



9. ábra. Blokkcsoport (2, 3)

Ha a kimeneti textúra tartalmazza az összes hibablokkot, a blokkfúzió nehézkes, mivel az összeadások operandusai ugyanabban a textúrában vannak, és ezért címszámításokra van szükség, amelyek lassabbá és körülményesebbé teszik ezeket a műveleteket. További kellemetlenséget okoz az is, hogy a blokkfúziók során létrejött textúrák különböző méretűek (attól függően, hogy hány blokkcsoportot egyesítünk), nem egyeznek meg a bemeneti textúrákéval.

Szerencsére létezik egy elegáns megoldás: az SAD shader által létrehozott textúra makroblokkonként csak egyetlen hibablokkot kell tartalmazzon. Mivel egy makroblokk 16 darab 4x4-es blokkból áll, az összes hibablokk tárolásához 16 ilyen textúra szükséges. A shadernek egy paraméteren keresztül lehet megadni, hogy melyik blokkcsoportra (a makroblokkok mely indexű blokkjaira) végezze el a számításokat. Ennek a módszernek köszönhetően a blokkfúzió nagymértékben leegyszerűsödik és felgyorsul. Ez a művelet tulajdonképpen csak annyiból áll, hogy össze kell adni egyesítendő blokkcsoportoknak megfelelő textúrákat (az azonos indexű elemeket kell összeadni). Tehát a blokkfúzió kimeneti textúrája ugyanakkora, mint a bemeneti textúrái, és nem függ attól,

hogy hány blokkcsoportot fuzionálunk össze. Így az algoritmus során használt összes hibablokkokat tartalmazó textúra azonos méretű. A módszernek az egyedüli hátránya az, hogy a szomszédos hibablokkok különböző textúrákban találhatóak, és így egy kicsit nehézkesebb ezekkel dolgozni.

A képkockák pixelei általában 8-bites jelöletlen egész típusúak. Mivel a hibafüggvényt legfeljebb 16x16 pixelből álló blokkokra kell kiértékelni, a kapott érték egy 0 és 65280 ($255 \cdot 16 \cdot 16$) közötti egész szám lehet. Ennek az eltárolásához 16-bites egészre van szükség, viszont a GPU-k többsége nem támogat ilyen pixelformátumokat. Ezért a hibaértéket két darab 8-bites egészben kell eltárolni. Ez egy kicsit nehézkes, mivel olyan pixelformátumot kell használni, amely legalább két komponensből áll, és a shaderekben olvasásnál és írásnál konverziót kell alkalmazni.

A GPU-k elsősorban 4 elemű vektorokkal való számításokra lettek tervezve. Általában egy skaláris művelet elvégzése ugyanannyi időbe kerül, mint egy vektoriálisé, annak ellenére, hogy sokkal több adatról van szó. A hibafüggvény kiértékelését legkézenfekvőbben skaláris műveletekkel lehetne megoldani, viszont ezek helyett érdemes vektoriális műveleteket használni.

Ahhoz, hogy ki tudjuk számolni a hibaértéket, a shaderen belül a textúrákból ki kell olvassuk az aktuális- és referencia blokkot, majd ezekre ki kell értékelni a hibafüggvényt. Mivel a hibatextúrában egy pixel egyetlen hibaértéket reprezentál, a shaderen belül csak egyetlen blokkhasonlítást kell végezni. Így viszont nem tudjuk teljes mértékben vektorizálni az algoritmust, mivel a pixeleket egyenként kell kiolvassuk.

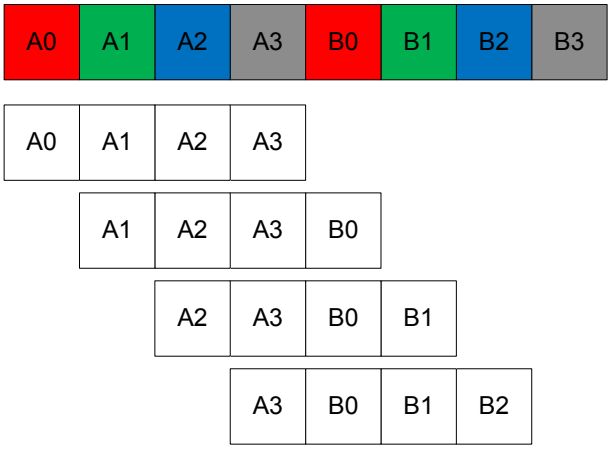
A shader az aktuális képkockából csak 4-el osztható kezdőkoordinátájú (0-tól kezdődő koordináták esetében) 4x4-es blokkokat kell kiolvasson. Emiatt a képkocka ne skaláris, hanem vektoriális, azaz RGBA, pixelformátumú textúra legyen [1]. Így egy RGBA pixel nem 4 színcsatornát, hanem 4 egymás melletti skaláris pixelt tárol. A textúrában lévő pixeleket nem kell átrendezni; ez csupán az adatok átértelmezése. Ezzel a tárolási móddal azt sikerült elérjük, hogy csupán 4 textúraolvasás szükséges egy 4x4-es blokk betöltéséhez, és így a hibaértékeket teljes mértékben vektoriális műveletekkel is ki lehet számolni. A következő *GLSL*[1] függvény a paraméterként megadott két blokkra kiszámolja a hibaértéket:

```
float sad(vec4[4] block1, vec4[4] block2)
{
    vec4 sad4 = abs(block1[0] - block2[0]) +
                abs(block1[1] - block2[1]) +
                abs(block1[2] - block2[2]) +
                abs(block1[3] - block2[3]);

    return sad4[0] + sad4[1] + sad4[2] + sad4[3];
}
```

[1] az OpenGL shader nyelve

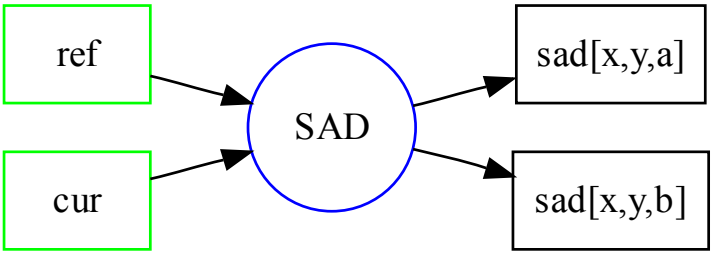
Viszont felmerül egy súlyos probléma: a referencia képkockából tetszőleges kezdőkoordinátájú blokkokat kell kiolvassunk, amit ilyen tárolási móddal nem lehet megvalósítani[2]. A függőleges koordináták továbbra is tetszőlegesek lehetnek, viszont a vízszintes koordináták 4 többszörösei kell legyenek.



10. ábra. Referencia blokkok előállítása két szomszédos blokkból (egy sor esetében)

Ezt a limitációt szerencsére meg lehet kerülni. Ha a referencia textúrából vízszintes irányban 2 egymás melletti blokkot olvasunk be, ezekből elő lehet állítani 4 darab referencia blokkot. Egy blokksor esetén a 10. ábra szerint kell eljárni. Ezzel a technikával hozzá lehet férni az összes szükséges referencia blokkhoz, viszont ez csak akkor lehetséges, ha a keresési tartomány mérete vízszintes irányban legalább 8, és osztható 4-el. Ez a limitáció a gyakorlatban nem jelent problémát. A hatékonyság érdekében (a shader ugyanazt a blokkpárt többször ne olvassa ki) a shader 1 helyett 4 hibaértéket kell kiszámoljon, amelyeket természetesen egyszerre kell eltárolni. Ezeket nem lehet egyetlen textúrába beírni, mivel 8 pixelkomponensre lenne szükség, és olyan formátum nem létezik. Ezért a shader két RGBA8 textúrába kell írjon: az egyikbe az első hibaértékpárt írja, a másikba pedig a második párt. Tehát az SAD shader egy *hibatextúrapár*t generál.

Az SAD shader be- és kimeneti textúráit a 11. ábra összegzi.



11. ábra. Az SAD shader be- és kimenetei

Az ábrán a kör a shadert, a téglalapok pedig a textúrákat szimbolizálják. A két zöld téglalap a

[2]Csak teljes pixeleket lehet kiolvasni egy textúrából.

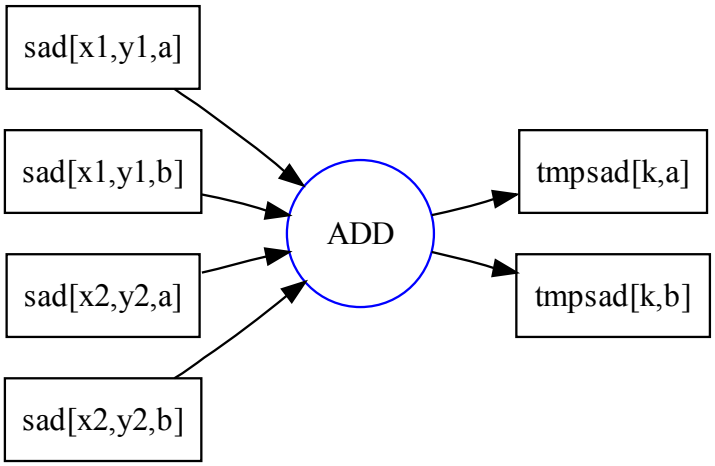
bemeneti textúrák: az aktuális- (*cur*) és referencia (*ref*) képkocka. Az ábrán látható *sad* egy háromdimenziós tömb, amelynek elemei a hibatextúrák. Ez az elrendezés könnyebbé és átláthatóbbá teszi a hibatextúrák kezelését. Az első két index, *x* és *y*, az illető blokkcsoportot azonosító makroblokkokon belüli indexpár. A harmadik index, amely *a* vagy *b* lehet, azt mutatja, hogy a hibatextúrapár melyik tagjáról van szó.

3.3. ADD shader

Az *ADD shader* a blokkfúzióért felelős. Bemenetként kap két hibatextúrapárt, kimenete pedig egy új hibatextúrapár. Mindegyik textúra mérete azonos kell legyen.

A shader csak annyit csinál, hogy összeadja a két-két hibatextúrát (az azonos koordinátájú hibaértékeket adja össze). Mivel egy hibaérték két 8-bites egészben van tárolva, az olvasásnál és írásnál konverzió szükséges.

A hibatextúrapárok azonos indexű tagjait össze lehetne adni egymástól függetlenül is, de az két rajzolási műveletet igényelne, ami rontaná a teljesítményt.



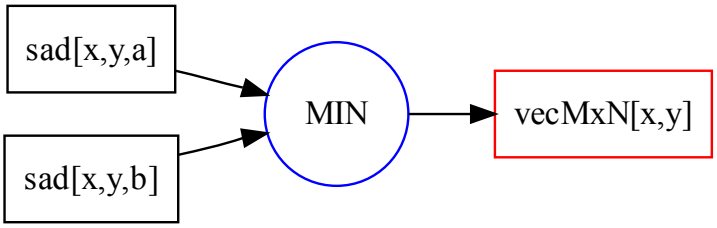
12. ábra. Az ADD shader be- és kimenetei

A 12. ábrán a *tmpsad* egy olyan tömböt jelöl, amely temporális hibatextúrapárokat tartalmaz. A *k* a hibatextúrapár indexe.

3.4. MIN shader

A *MIN shader* a hibatextúrákból előállítja a *vektortextúrát*, amely tartalmazza a legjobb mozgásvektorokat és a hozzájuk tartozó hibaértékeket. Ezt úgy valósítja meg, hogy megkeresi a legkisebb hibaértéket minden hibablokk esetében. A mozgásvektort a legkisebb érték relatív pozíciója (a hibablokkhoz viszonyítva) alapján kapja meg.

A kimeneti textúra RGBA8 formátumú. A mozgásvektort az első két 8-bites komponensben tárolja, amelyek elegendőek akár egy 256x256 méretű keresési tartomány esetén is. Mivel ezek a komponensek jelöletlen egészek, és a vektorkoordináták lehetnek negatívak is, a shader a koordinátákat *eltolt nullpontú formában* kódolja. Az utolsó két pixelkomponensbe a hibaértéket írja be a már említett módon.



13. ábra. A MIN shader be- és kimenetei

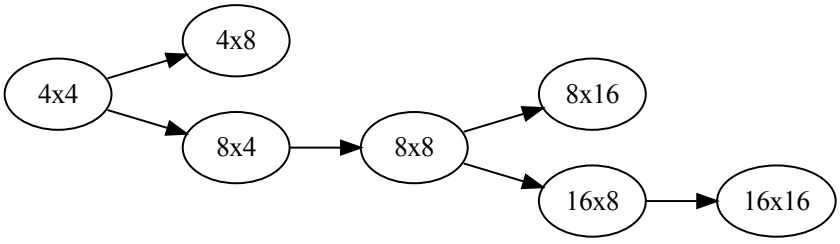
A 13. ábrán a *vecMxN* a vektortextúrákból álló tömb, ahol *M* és *N* a blokkméretet jelentik.

3.5. Vezérlő komponens

A vezérlő komponens bemenetként megkapja a két képkockát, kimenetként pedig az összes blokkméretnek megfelelő vektortextúrákat kell előállítsa.

A számítások előtt létrehozza az összes ideiglenes adatokat tároló textúrát, amelyeket ismételt futtatások során újra fel lehet használni.

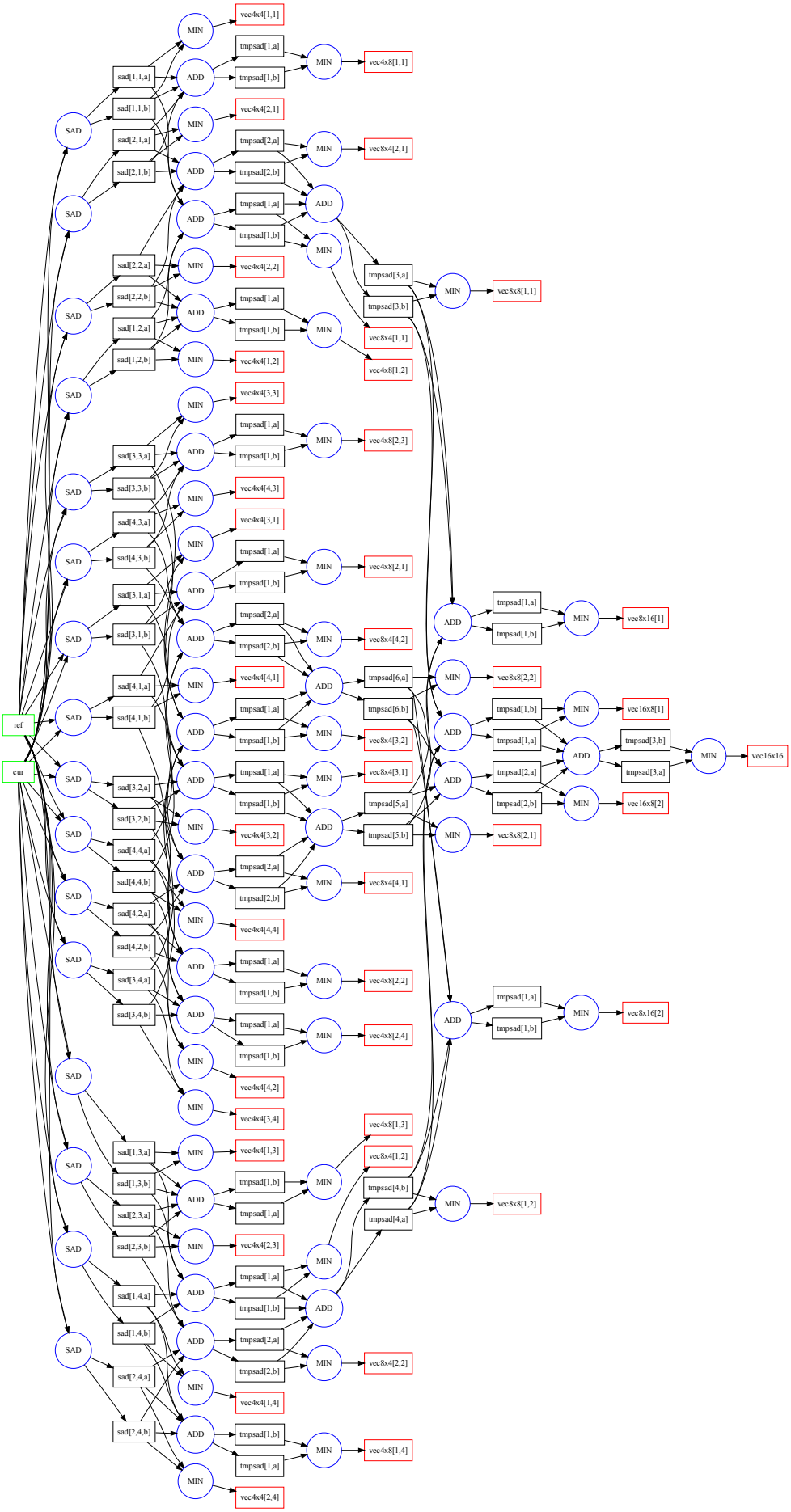
Első lépésben az SAD shadert többször meghívva kiszámolja az összes 4x4-es blokkcsoport hibatextúráit. Ehhez 16 pár temporális textúrát használ fel. Ezekből a MIN shader segítségével rögtön elő lehet állítani a vektortextúrákat.



14. ábra. Blokkfúziók

A többi blokkméret esetében már blokkfúziókra van szükség. Ha optimális sorrendben végezzük el a fúziókat, további 6 pár temporális textúrára van szükség. A nagyobb méretű blokkok fúziója során fel lehet használni a kisebb blokkok esetében kapott eredményeket (14. ábra).

A vezérlő komponens működését egy gráf segítségével is lehet ábrázolni. Ez a 15. ábrán tekinthető meg. A gráf építő elemei a shaderek, amelyek megfelelően összekapcsolva együttesen elvégzik a kívánt komplex műveletsorozatot.



15. ábra. A vezérlő komponens

4. Implementáció

4.1. GPGPU könyvtár

A GPU programozása közvetlenül grafikai API-n keresztül nehézkes, amiatt, hogy grafikai koncepciókkal kell dolgozni (pl. textúra, shader, rajzolás). Ezért kifejlesztettek több speciális *GPGPU könyvtár*at, amelyek megkönnyítik az ilyen jellegű programozást, elrejtve a programozó elől a grafikával kapcsolatos részleteket [2].

A GPGPU-ra legalkalmasabb grafikai API az OpenGL, mivel tartalmaz olyan fontos funkcionalitásokat (pl. aszinkron textúramásolás), amelyekkel más konkurens API-k nem rendelkeznek, és ugyanakkor platformfüggetlen is. Sajnos egyelőre nem létezik olyan GPGPU könyvtár, amely kellőképpen támogatja az OpenGL-t, és egyidőben nyitott forráskódú is. A nyitott forráskódúság azért nagyon fontos, mert bárki meg tudja vizsgálni a működését, és szükség szerint testre tudja szabni. Tudományos kutatásnál ezek nem elhanyagolható tényezők.

A felsorolt okok miatt kifejlesztettünk egy saját, nyitott forráskódú, keresztplatformos (egyelőre Windows és Unix), OpenGL 2.1[3]-re épülő, *GPUC* nevű GPGPU könyvtárat[4]. A maximális hordozhatóság és kompatibilitás érdekében a C programozási nyelvben implementáltuk, és a shadereket GLSL-ben írtuk meg. Ugyanakkor készítettünk egy C++ *wrapper*t is. A könyvtár sajátossága, hogy rendkívül kicsi és egyszerű.

4.2. Az algoritmus implementálása

Magát az algoritmust a már említett GPUC segítségével valósítottuk meg. A vezérlő komponenst C++-ban implementáltuk, felhasználva a wrappert.

A fejlesztés *Ubuntu Linux 7.10 x86-64* és *Windows Vista 64-bit* alatt történt. Az elsődleges fejlesztési környezet *Eclipse CDT* volt, a C/C++ fordító pedig *GCC 4.2*. Windows alatt *Visual C++ 2008*-at használtunk.

4.3. Eredmények

Az algoritmus a mozgáskeresési problémára minden esetben *optimális* megoldást talál, ezért minőségi szempontból fölösleges elemezni.

A teljesítmény méréséhez egy tesztprogramot készítettünk. A teszteléseket egy *NVIDIA GeForce Go 7600 SE* notebook GPU-n bonyolítottuk le. Ez a GPU az előző generáció tagja, és csök-

[3] a jelenlegi legújabb verzió

[4] A könyvtár forráskódja elérhető a `http://code.google.com/p/gpuc/` címen.

kentett órajelen fut a kisebb hőtermelés érdekében. Ezek miatt teljesítményben jóval lemarad a jelenlegi *high-end* GPU-któl, viszont jól tükröz egy átlagos esetben elvárható sebességszintet.

A tesztelés során felhasznált képkockák mérete 512x512 pixel volt. A tesztet többször futtattuk le, és a futási időket átlagoltuk. Az első futtatás eredményét nem vettük figyelembe, mivel akkor a GPU a háttérben időigényes késleltetett inicializálásokat hajt végre. A kapott eredményekbe (1. táblázat) nem számoltuk bele a textúrák fel- és letöltéséhez szükséges időt, mivel ezeket a műveletek el lehet végezni aszinkron módon, akár számolás közben is [4].

Keresési tartomány (pixel)	Futási idő (ms)
8x8	30,3284
16x16	99,3946
32x32	528,58
64x64	3236,97

1. táblázat. Teszteredmények

A kapott futási idők nagyjából az elméletben elvárható módon függnek a keresési tartomány méretétől. Az algoritmus sajátosságai miatt gyakorlatilag teljesítmény szempontjából nem számít, hogy a képkockák vagy a keresési tartomány méretét skálázzuk egy adott mértékben.

5. További kutatások

A bemutatott algoritmus csak egész koordinátájú mozgásvektorokat támogat, viszont a legtöbb tömörítési standard megenged fél- vagy akár negyedpixeles (*half-pel*, *quarter-pel*) pontosságúakat is. További kutatások során az algoritmust ki lehetne egészíteni ezzel a fontos funkcionalitással.

Sok esetben nincs szükség az ESA által nyújtott minőségre. Meg lehetne vizsgálni, hogy mely más nagyobb teljesítményű keresési stratégiát lehetne jól átültetni GPU-ra. A legígéretesebbnek tűnő ilyen algoritmus a *hierarchikus keresés*, amely, az ESA-hoz hasonlóan, determinisztikus és könnyen párhuzamosítható.

6. Konklúzió

Egy olyan speciális mozgásbecslési algoritmust sikerült kidolgozni, amely hatékonyan ki tudja aknázni a GPU párhuzamos architektúráját. Az optimális megoldás keresése mellett döntöttünk, így az algoritmus megfelel a legmagasabb professzionális igényeknek is.

Az algoritmust be lehet építeni akár a legkomplexebb videóstandardokra tervezett tömörítőkbe

is. Az optimális teljesítmény érdekében a CPU és a GPU egyidőben foglalkozhat a tömörítés különböző fázisaival, így egyik feldolgozási egység sem marad kihasználatlanul.

Hivatkozások

[1] Chi-Wang Ho, Oscar C. Au, S.-H. Gary Chan, Shu-Kei Yip, and Hoi-Ming Wong. Motion estimation for H.264/AVC using programmable graphics hardware. In *ICME*, pages 2049–2052. IEEE, 2006.

[2] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[3] Iain E. G. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. John Wiley & Sons, 2003.

[4] Richard S. Wright, Benjamin Lipchak, and Nicholas Haemel. *OpenGL SuperBible*. Addison-Wesley Professional, fourth edition, 2007.