Heterogén számítási rendszerek gyakorlatok (2019.)

Tartalom

1. 2D konvolúció megvalósítása C-ben

A gyakorlat során a cél egy kétdimenziós konvolúció megvalósítása C-ben, majd ezen implementáció gyorsítása OpenMP pragmákkal, illetve SSE/AVX intrinsic-ekkel.

1.1 C implementáció

A konkrét megvalósítás egy 5x5-ös Laplace szűrés, amely a bemeneti képen élkeresést hajt végre a három színkomponensre (R, G, B) külön-külön. A bemenet egy 4992x3744 pixelt tartalmazó JPG file, a kimenet ugyanilyen formátumú. Az 5x5-ös Laplace szűrő együtthatói:

-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	24	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

1. ábra Együtthatók

A bemenet és az elvárt eredmény:

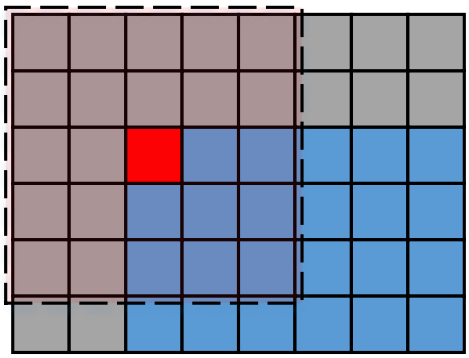
2. ábra Bemeneti és kimeneti kép

Az 5x5 2D konvolúció formálisan:

$$o = \sum_{fy=-2}^{2} \sum_{fx=-2}^{2} h(fy, fx) * p(y + fy, x + fx)$$

ahol h() az együtthatókészlet, p a bemenet, o pedig a kimenet.

Mint minden konvolúciós eljárásnál, itt is problémába ütközünk a kép széleinek kezelésekor: ahhoz, hogy a bemenettel megegyező számú kimeneti pixelt tudjunk generálni, nem létező pixelekre is szükség van. Az 1. ábra a bal felső sarok számításához szükséges pixeleket mutatja:

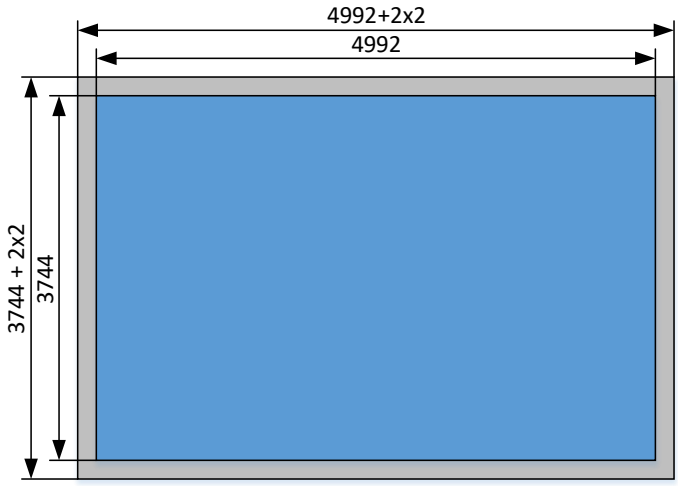


3. ábra Konvolúció a kép bal felső sarkában

A kék pixelek jelölik a bemeneti (és kimeneti) kép tényleges pixeleit, a piros a számítandó pixel, míg szürkével a számításhoz szükséges hiányzó pixeleket jelöltük. Az 5x5 pixeles szűrő ablakot szaggatott vonal mutatja. A képfeldolgozásban a probléma megoldására legelterjedtebb módszerek:

- A nem létező pixelekre a bemenet legszélső sorát/oszlopát másoljuk.
- A szélső sorokat/oszlopokat tükrözve duplikáljuk.
- A nem létező pixeleket feketével helyettesítjük.

Egyszerűsítendő a gyakorlat során a feladatot, a harmadik opcióval élünk, ráadásul a feketével történő kiegészítést nem a tényleges számítás során tesszük meg, hanem a fájl beolvasás után, de a konvolúció futtatása előtt. Azaz a feladatot tekinthetjük úgy, hogy a konvolúciós függvény tényleges bemenete minden irányban két pixellel ki van terjesztve, míg a kimenet az eredeti bemeneti kép méretével egyezik meg.



4. ábra Kiterjesztett bementi (szürke), illetve eredeti méretű kimeneti (kék) kép

A konvolúciót végző függvény deklarációja:

```
void conv_filter(int imgHeight, int imgWidth, int imgWidthF,
                 short *filter, unsigned char *imgSrcExt, unsigned char *imgDst)
```

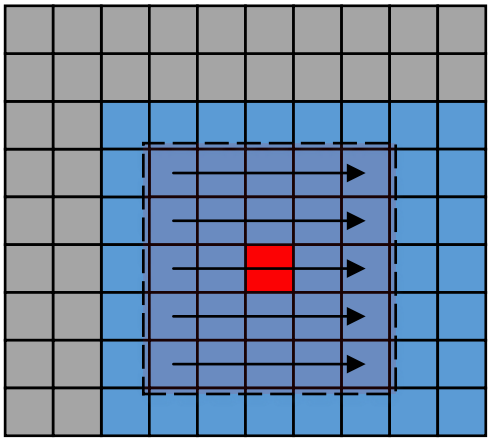
A változók jelentése:

- imgHeight: az eredeti bemeneti kép magassága pixelben.
- imgWidth: az eredeti bemeneti kép szélessége pixelben.
- imgWidthF: a kiterjesztett kép szélessége pixelben.

- *filter: a szűrőegyütthatókra mutató pointer.
- *imgSrcExt: a kiterjesztett bemeneti kép pixeleire/komponensire mutató pointer. Minden pixel 3 darab 8 bites komponenst tartalmaz (R, G, B).
- *imgDst: a kimeneti kép pixeleire mutató pointer. Nincs kiterjesztve, és ugyancsak 3 darab 8 bites komponenst tartalmaz pixelenként.

Adott kimeneti pixel számítása során azalábbi „lépkedés” történik:
1. A szűrőablak bal felső pixeléről indulunk
2. Végig lépkedünk a szűrőablak adott során
3. Visszalépünk az ablak első oszlopára, majd lelépünk egy teljes sort
4. Folytatjuk (2)-vel

Azaz grafikusan ábrázolva:



5. ábra A szűrőablak letapogatása

Gondolja végig:
- Hogyan számítható ki egy adott pixel lineáris, pixelben számolt címe felhasználva a (kiterjesztett) vízszintes képméretet és a pixel pozícióját (azaz adott pixel abszolút értékben hányadik a képen)?
- Hogyan képezhető ez le memória címmé, tudván, hogy egy pixel 3 színkomponenst tartalmaz?
- Hogyan oldható meg a szűrőablakon belül a következő sor elejére lépés a lehető legegyszerűbb aritmetikai műveletekkel?
- A konvolúciót megvalósító egymásba ágyazott for() ciklusoknak memória elérés szempontjából mi az ideális szervezése, azaz mi legyen a legbelső ciklus?

A konvolúció ciklusának implementációja után az alábbi kimeneti kép generálódhat:

6. ábra Majdnem megfelelő kimenet

Gondolja végig, hogy a Laplace szűrővel történő konvolúció során előfordulhat-e olyan eset, amikor az eredményül adódó színkomponens kívül esik a JPG fájlban (ami a bemenetünk és kimenetünk is) ábrázolható 0…255 tartományon. Ha igen, milyen lépésre van még szükség a kimenet írása előtt?

1.2 Futtatás több szálon OpenMP-vel

A konvolúció számítása egy gyakorlatilag triviálisan párhuzamosítható feladat, hiszen az egyes kimeneti értékek csak a bemenettől függenek, így egymástól függetlenül számíthatók (miközben természetesen ugyanazt a bemeneti adatot több kimenethez is felhasználjuk). Ennek megfelelően a számítást végző program több szálon történő futtatása is egyszerű: a for ciklusok egyes iterációja egymástól függetlenül számítható más-más szálon.

Alkalmazza a for() ciklusok párhuzamosítására szolgáló pragma-t, ügyelve a változók láthatóságára (shared/private).

Vizsgálja meg, hogy alakul a futási idő, ha más-más ciklust párhuzamosít. Ügyeljen a változók shared/private beállítására, illetve a megfelelő kezdőértékekre!

1.3 Vektorizáció

A vektorizáció egy kissé bonyolultabb:

- Az alapműveletünk a szorzás és összeadás 8 bites bemeneti adatokon, de ügyelve arra, hogy a számítás során ennek értéktartománya nem elegendő. Az SSE/AVX utasításkészletek nem tartalmaznak 8 bites szorzó utasítást, de az értéktartomány növekedése miatt ez amúgy sem lenne jó. Lehetőségeink:
    - Float szorzás (_mm256_mul_ps) és összeadás használata. AVX esetében ez 8 művelet elvégzését jelenti órajelenként. Ha általános konvolúciós függvényt írnánk, akkor ez lenne a megfelelő választás, hiszen nem lenne semmilyen információnk a használt együtthatókról. Jelen esetben viszont ez nem igaz, így van gyorsabb megoldás.
    - Short műveletvégzés. 16 bites adatok szorozhatók úgy, hogy az eredmény alsó 16 bitjét tartjuk meg (_mm256_mullo_epi16). Mivel a Laplace szűrés során a bemeneti értéktartománynál legfeljebb 48-szor nagyobb érték állhat elő, így a 16 bites változó értéktartománya elegendően nagy a számítás túlcsordulás nélküli elvégzésére.
- Típus konverziót kell végeznünk, amire kétféle lehetőségünk van
    - Megtehetjük, hogy a konvolúció előtt a teljes képet átkonvertáljuk float vagy short színkomponensre. Ennek előnye, hogy minden komponenst csak egyszer konvertálunk, van azonban néhány hátránya: szükségünk van egy teljes képnyi átmeneti memóriára

(ráadásul dupla akkora változóval); a teljes képet be kell olvasnunk, majd konvertálás után ki kell írnunk; a konvolúció során kétszer akkora adatot olvasunk → jelentősen növekszik a memória sávszélesség igény.
    - Elvégezhetjük a konvertálást a konvolúció bemeneti adatainak olvasásakor, azaz 8 bites komponenseket olvasunk, konvertálunk, majd konvolválunk. Ennek előnye a jó memória sávszélesség kihasználás, hátránya, hogy minden egyes komponenst annyiszor konvertálunk, ahányszor használjuk (25!).
    - Tipikus konvertáló utasítások:
        - A short feldolgozás esetén szükséges 8 bit unsigned → 16 bit signed konverzió az __m256i _mm256_cvtepu8_epi16 utasítással tehető meg. A bemenet itt egymást követő 16 darab 8 bites érték (_m128i típus).
        - Amennyiben float feldolgozást választunk, úgy konvertálni az _mm256_cvtepi32_ps utasítással lehet, de ennek bemenete egy 32 bites integer, így a 8 bites adatainkat előbb erre a formátumra kell konvertálni az _mm256_cvtepu8_epi32 utasítással.
- Amennyiben short műveletvégzést és AVX utasításokat használunk, úgy egy vektor utasítással 16 elemen tudjuk végrehajtani ugyanazt a műveletet. Ennek megfelelően egyszerre 16 kimeneti színkomponenst számolunk. A számítás lépései:
    - Az első lépésében 16 egymás melletti 8 bites bemeneti értéket (színkomponenst) töltünk be, ezeket mind a bal felső együtthatóval kell megszorozni.
    - A következő lépésben a bemenet olvasását egy pixelnyivel, azaz 3 8 bites elemmel kell odébb léptetni.
        - Ezt megtehetjük úgy, hogy unaligned olvasással megint beolvasunk minden adatot.
        - Vagy pedig felhasználhatjuk a permutálás és/vagy shuffle utasításokat, ami bonyolultabb, de memória sávszélesség igény szempontjából kedvezőbb lehet. Sajnálatos, hogy sem 8 bites, sem pedig 16 bites adatra nincs két operandusú „multiplexálás” jellegű utasítás, így a feladat csak operanduson belüli permutálással, maszkolással és vagy-olással oldható meg, ami jelentős mennyiségű utasítást igényel – így végeredményben hiába spórolunk memória sávszélességet, előfordulhat, hogy lassabb eredményt kapunk.
    - Az előző lépést ismételjük az összes szűrőegyütthatóra, megfelelően lépkedve a bemenet oszlopain és sorain.
    - A konvolúció elvégzése után az eredményt szaturáljuk a 0…255 tartományra.
    - Végül a 16 db 8 bites értékből előállítunk 16 db 8 bites értéket, amit kiírunk a kimeneti memória területre.
        - Ezt megtehetnénk úgy, hogy a kiírás előtt az _mm256_cvtepi8_epi16 konvertáló utasítást használjuk, de sajnos a Visual Studio 2015-ből ez hiányzik.
        - Alternatív megoldás, hogy az __mm256i típusú vektor kimeneti változó (fval) címéből készítünk egy __mm256i típusra mutató pointert, majd ezt cast-oljuk short-ra mutató pointerré, aminek egyesével végiglépkedve a 16 elemén megtehető a kiírás. (Érdemes megnézni, hogy mi lesz a fordítás eredménye.)

A feladat tehát a következő. Vektorizálja az eredeti C kódot:

- Használjon 16 bites vektor elemeket a számításhoz. A típus konvertálást a konvolúcióval együtt végezze.
- A pixeleken történő lépkedéskor használjon unaligned load utasítást.

Ha ez a megoldás működik, akkor további esetleges optimalizációs elehetőségek:

- A Visual Studio nem igazán hatékony a ciklusok kiterítésében (unroll), „kézzel” kiterítve a kódot jelentősen növelhető a feldolgozási teljesítmény.
- A pixelenkénti lépkedésnél a folyamatos töltés helyett használjon minimális számú töltést, permutálást (pl. _mm_shuffle_epi8), maszkolást és vagy-olást.
- Még több kimenet egyidejű számításával növelhető a tényleges számítást végző utasítások aránya, de ez csak addig hatékony, amíg nem (nagyon) futunk ki a rendelkezésre álló regiszterekből.

SSE és AVX utasítás intrinsic-ek:
Intel intrinsic guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

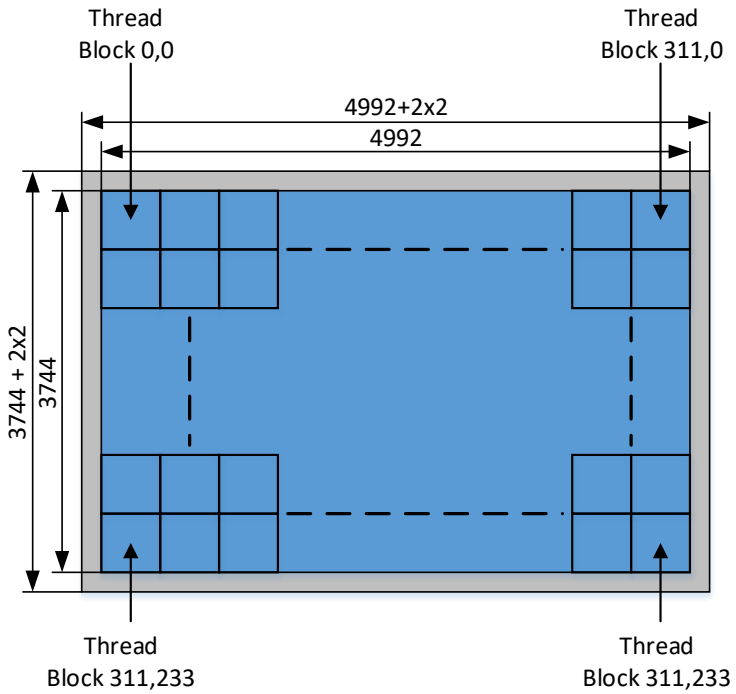
2. 2D konvolúció GPU-val

A GPU-s gyakorlatokon megvalósítandó feladat ugyanaz, mint az első gyakorlat feladata: Laplace szűrés egy képen, melynek pixelei 3 8 bites komponenst (RGB) tartalmaznak.

2.1 Global Memory használata

A CPU megvalósítás esetében a teljes kép feldolgozását két ciklus biztosította, amely „végiglépkedett" a kimeneti kép összes pixelén. A GPU megvalósítás esetén ugyanezt az indított szálak megfelelő száma biztosítja. Első megközelítésünk legyen az, hogy minden egyes, a GPU-n elindított szál a kép egyetlen pixelének számításért felel, azaz összesen 4992x3744 szálat kell elindítani (azaz a kimeneti kép pixelszámának megfelelőt). A Thread Block méretet a szokásos ökölszabály alapján határozhatjuk meg, azaz legyen 32 (warp méret) többszöröse, ne legyen túl kis érték, de ne legyen nagyobb, mint az adott GPU maximuma (~1024). 16x16, 32x16 vagy 32x32 például jó választás, induljunk ki az elsőből. A teljes kép feldolgozásához ekkor (4992/16)x(3744/16)=312*234 Thread Block szükséges. Mivel mindkét hányados egész szám, ennél a képnél minden szál érvényes munkát fog végezni – amennyiben nem így lenne, úgy egyes szálakat a kernel kódban le kellene tiltani.

A kernel megvalósítását tekintve tehát a kernelnek (ami egy szál kódja) az 5x5-ös konvolúció kiszámítása a feladata, így a kernel kódja nagy vonalakban megegyezik a C implementáció kódjával. A lényeges különbség a memória címzés alapja: míg a C kódban ez a sor és oszlop ciklusváltozók alapján (akár inkrementálisan) történt, addig a kernel kódban a címet a szál azonosító alapján határozhatjuk meg. A 7. ábra a kimeneti kép Thread Block-okra történő felosztását mutatja.



7. ábra Kép felosztása Thread Block-okra

A memória (kezdő)cím kiszámításához arra van szükségünk, hogy egy (blockIdx.x, blockIdx.y) X, Y indexű Thread Block-on belüli (threadIdx.x, threadIdx.y) indexű szálnak (jelenleg == pixel, PX és PY) az abszolút sorszámát ki tudjuk számolni. X és Y irányban:

$$PX = blockIdx.x * blockDim.x + threadIdx.x$$

$$PY = blockIdx.y * blockDim.y + threadIdx.y$$

Ebből pedig az adott pixel sorszáma a képen (hasonlóan a CPU megvalósításhoz):

$$P = PY * kép\ szélesség + PX$$

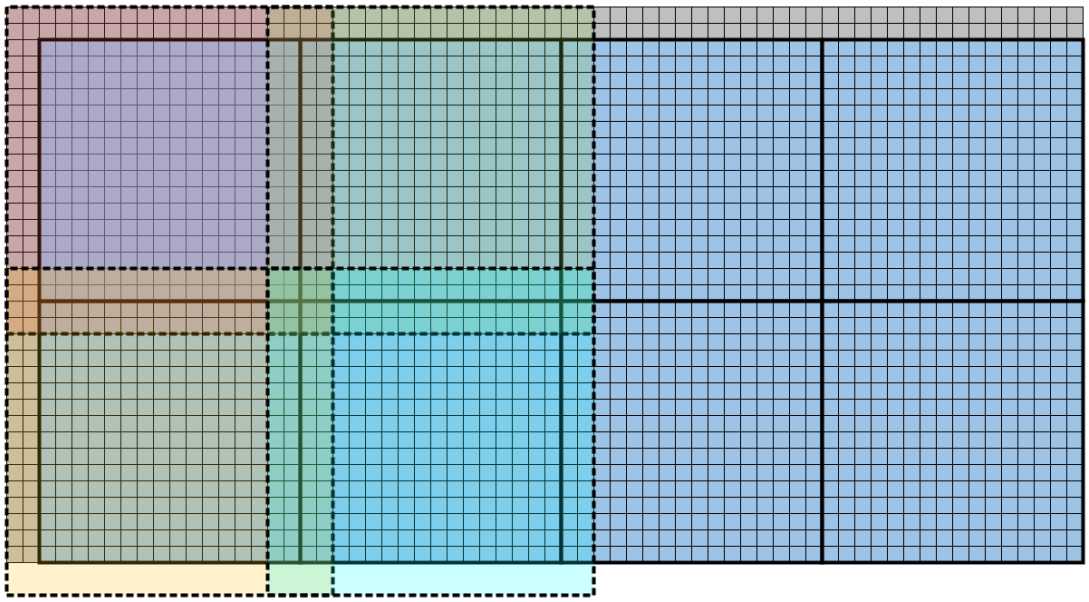
Mivel egy pixel 3 komponensből áll, így adott pixel első komponensének címe a fenti érték háromszorosa. Ezzel meghatároztuk, hogy adott szálnak milyen memória címre kell írni, valamint hogy a bemeneti képen milyen címről kell kezdenie az 5x5 pixeles ablak beolvasását. Az ablakon belüli lépkedés megegyezik a C implementációval.

Implementálja a Global Memory-t használó kódot, ellenőrizze a kimenetet, majd vizsgálja meg az NVIDIA Visual Profiler által szolgáltatott adatokat!

2.2 Shared Memory használata, int-tel történő számítás

Mint minden olyan esetben, amikor az egy blokkon belüli szálak adatelérése redundáns, a belső memóriába történő „kézi" bufferelés nagymértékben tudja javítani a teljesítményt (a Global Memory sávszélessége a számítási teljesítményhez képest relatíve alacsony, így a többszörös adatelérés kerülendő, bár ezen a cache azért segít). Jelen esetben az egymás melletti szálak 5x5 méretű ablaka jelentős mértékben átfed, így ugyanazt a bemeneti pixelt sok szál használja (két egymás melletti kimeneti pixel számításánál 20 közös bemeneti pixel van). A chip-en belüli Shared Memory sávszélessége jóval nagyobb, így ha a feldolgozás megkezdése előtt a szükséges bemeneti adatokat átmásoljuk a Global Memory-ból ide, akkor a konvolúció elvégzése során elegendő a Shared Memory-t olvasni.

Egy Shared Memory blokkot egy Thread Block használ, így annyi bemeneti pixelt kell ide betölteni, amennyi a Thread Block-ban futó szálak számára szükséges. Hasonlóan a teljes képhez, itt sem a szálak számával megegyező (16x16) képpontra van szükség, hanem Thread Block területét is ki kell terjeszteni minden irányban 2-2 pixellel, így a betöltendő terület 20x20 pixeles, azaz 60x20 byte-os. A 8. ábra az első néhány Thread Block által lefedett kimeneti területet (nagyobb kék négyzetek), illetve az ezek számításához szükséges bemeneti képtartományt (nagyobb színes, szaggatott vonallal jelölt négyzetek) mutatja. A kimeneti blokkok (==Thread Block) értelemszerűen nem, a bemeneti blokkok (20x20 pixeles kiterjesztett ablakok) azonban átfedik egymást.



8. ábra Thread Block-ok és a számításhoz szükséges bemeneti terület

2.2.1 Shared Memory írás

A beolvasandó terület meghatározását követő kérdés, hogy hogyan olvassuk be a 20x20 pixelt (60x20 byte) a rendelkezésre álló 16x16=256 szállal. Első megvalósításként egy áttekinthető megoldást választunk: a lehető legtöbb szálat felhasználva, byte-osával olvasunk. Ez azt jelenti, hogy az első sorban szükséges 60 byte-t 60 szállal olvassuk, majd a második sort további 60 szállal, és így tovább. Így egy utasítással (amelyet 240 szál hajt végre) 4 sort tudunk beolvasni, tehát a műveletet ötször meg kell ismételni, így összességében az olvasásban résztvevő 240 szál mindegyike 5 byte-t fog beolvasni.
Megjegyzések:

- A kernel kódja egyetlen szál által végrehajtott utasításokat tartalmaz (kivéve Shared Memory allokáció). Így a kernel kód 5 byte betöltését tartalmazza, NEM pedig a teljes 60x20 byte töltését. Az, hogy adott szál mely byte-okat tölti be, a Thread Block-on belüli threadIdx.x és threadIdx.y szál azonosítótól függ, a töltés bázis címe (Thread Block bal felső pixelének címe) pedig a Thread Block azonosítótól (blockIdx.x és blokcIdx.y).
- Vegyük észre, hogy **az adott szál által feldolgozott pixel és a Shared Memory töltés során betöltött adat között nincs összefüggés, a két művelet egymástól független**! A Thread Block 16x16-os dimenziója a számítási struktúrának felel meg, a betöltéskor más indexelésre van szükség. A Thread Block-on belüli lineáris szál azonosító a threadIdx.y*BlockDim.x+threadIdx.x képlettel generálható (ez a szokásos 2D → lineáris index számítás).

A töltéshez tehát a 16x16-os Thread Block 2D-s szál azonosítójából egy 1D-s 0…255 értékkészletű azonosítót generálunk:

$$th1d = threadIdx.y * blockDim.x + threadIdx.x$$

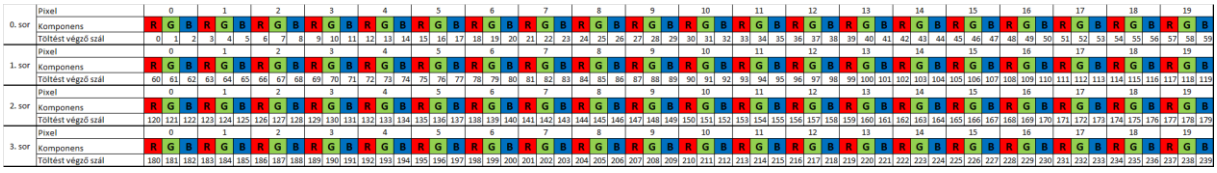
A töltendő terület bal felső pixelét a Thread Block azonosítók alapján lehet számítani mind X, mind Y irányban, majd ebből számítható a lineáris cím. Értelemszerűen a Thread Block által betöltendő kép-terület legfelső pixelének meghatározásában csak a Thread Block azonosítók szerepelnek, a Thread Block-on belüli threadIdx nem. Tehát:

$$base\_x = blockIdx.x * blockDim.x$$

$$base\_y = blockIdx.y * blockDim.y$$

$$base_{pixel} = base_y * ImgWidthF + base\_x$$

Mint fentebb megállapítottuk, a Thread Block-on belüli töltés minden szál esetében 5 byte mozgatását jelenti, az első 4 sor töltését az alábbi ábra szemlélteti.



9. ábra Shared Memory töltés 240 szállal

Amennyiben a Shared Memory deklarációja [20][20][3] (Y, X, komponens), úgy a fent meghatározott egy dimenziós szál azonosítóból (th1D) ki kell számítani az alábbiakat:

- Adott indexű szál a 4 sor méretű blokkon belül melyik sor töltésében vesz részt.
- Soron belül melyik pixel töltése a szál feladata.
- A pixelen belül mely komponenst tölti a szál.
- A globális memóriából történő olvasáshoz nincs szükség a pixel- és a komponens indexek szétválasztására, hiszen az egymást követő szálak egymást követő byte-okat töltenek.

(Megjegyzés: amennyiben a Shared Memory deklaráció [20][20*3] tömb, úgy ugyanez az írási cím számítására is érvényes.)

- A következő 4 sor betöltéséhez minden szálnak 4 teljes bemeneti képsorral „lejjebb” kell folytatnia a bement olvasását és 4 Shared Memory sorral „lejjebb” az írást.

2.2.2 Shared Memory olvasás

További fontos kérdés, hogy a Shared Memory-ban hogyan tároljuk a beolvasott adatokat. Itt a legfontosabb kritérium, hogy a feldolgozás során az egy warp-ban található szálak adott időpillanatban bankütközés mentesen tudják olvasni a Shared Memory-t. Ha a természetesen adódó módon ugyanolyan 8 bites RGB RGB RGB… formátumban tároljuk az adatokat, mint azok a Global Memory-ban találhatóak, akkor a 10. ábrán látható Shared Memory elrendezéshez jutunk.
A Shared Memory olvasása a kernelben az alábbi módon történik:

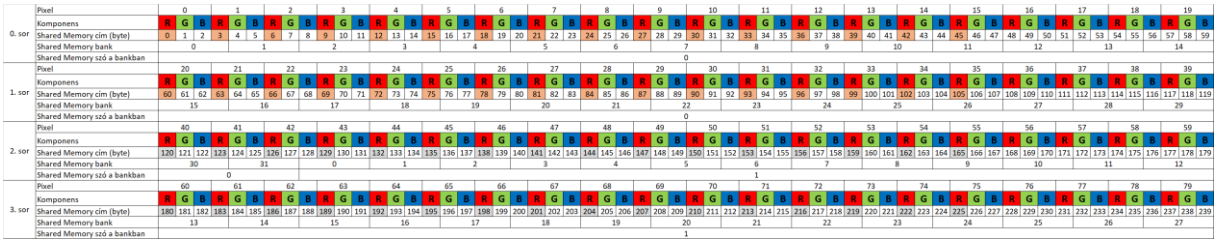
```
__shared__ unsigned char shmem[20][20][3];
……
……
for (int fy=0; fy<5; fy++)
  {
  for (int fx=0; fx<5; fx++)
  {
    for (int rgba=0; rgba<3; rgba++)
    {
      int rd_data = (int)(in_shmem[threadIdx.y+fy][threadIdx.x+fx][rgba]);
    }
  }
}
```

Amikor egy 16x16 szál méretű Thread Block 0. warp-ja (azaz az első két szál-sor) kiolvassa a számára szükséges első pixel első komponensét (azaz a szálhoz tartozó szűrőablak bal felső elemét), a fenti kód az alábbira egyszerűsödik:

```
int rd_data = (int)(in_shmem[threadIdx.y+0][threadIdx.x+0][0]);
```

Ez az a memória töltés (load) utasítás, amit a warp szálai egy adott órajelben végrehajtanak – az ennek megfelelő memória címeket a 10. ábra narancssárga jelöléssel mutatja.
Az ábrán szereplő jelölések/értékek tehát:

- Pixel: a töltandő pixel sorszáma. Egy sorban 20 pixel, összesen 20 sor (az ábra az első 4 sort mutatja).
- Komponens: adott pixelen belüli R, G, B komponens.
- Shared Memory bank: az adott komponens melyik Shared Memory bank-ban lesz. A bankok 4 byte-osak, így a bank index 4 byte-onként változik.
- Narancssárgával jelölt komponensek: lásd fentebb.
- A „Shared Memory szó a bankban” azt mutatja, hogy az addott érték a bankon belül hányadik szóban található



10. ábra Shared Memory elrendezés byte-os komponensek esetén

Mint az az ábrán látható a 20 pixel széles rész-kép egy-egy sora 60 byte, azaz 20*3/4=15 32 bites szó. Ez a Shared Memory-ban 15 bank-ot foglal el. Mint fentebb említésre került, a konvolúció első lépése, hogy minden szál beolvassa a saját kimeneti pixeléhez tartozó szűrő ablak bal felső pixelének R komponensét – a fentebb részletezett 0. warp esetén (0…31 szálak) ez a 10. ábrán narancssárgával jelölt komponenseket jelenti, azaz a 0…11 (0. sor) és 15…26 (1. sor) bankokból történik olvasás. Hasonlóan az 1. warp (32…63 szálak) a 30…9 és 13…24 bankokból olvasnak 1-1 byte-ot. Ugyan előfordul olyan eset, amikor egy bank adott szavából más-más szál más-más byte-ot olvas, de ez a bank ütközés nélkül megtehető – olyan eset, amikor egy warp szálai ugyanannak a banknak más-más szavát olvassák nincs, tehát bank ütközés sincs.

Arra is törekedni kell, hogy a Shared Memory írása során se legyen bank ütközés, de ez – mivel ritkább, és általában Global Memory sávszélesség limitált – kevésbé kritikus, mint az olvasások optimalizálása.

Implementálja a Shared Memory-t használó kódot, ellenőrizze a kimenetet, majd vizsgálja meg az NVIDIA Visual Profiler által szolgáltatott adatokat!

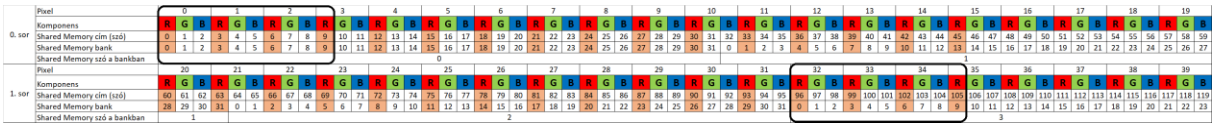
2.3 Shared Memory használata, float-tal történő számítás

Számos GPU architektúra esetén az int adattípussal történő számítás számottevően lassabb, mint a float adattípus használata, ezért megfontolandó gondolat, hogy a konvolúció számítása során float adattípust használjunk. Vegyük észre azonban, hogy ez azzal jár, hogy a Shared Memory-ban tárolt adatokat a számítás előtt float-tá kell konvertálni, ami azt jelenti, hogy minden egyes Shared Memory olvasást egy típus konverzió követ – ezen utasítás sebességétől függően nem biztos, hogy végeredményben gyorsabb végrehajtást kapunk, mint az első esetben.

A kód megírása és a futtatása után hasonlítsa össze az NVIDIA Visual Profiler által szolgáltatott eredményeket az előző implementációval!

2.4 Shared Memory használata, float tárolás, float-tal történő számítás

A 2.3 verzióban a típus konverziót minden szál minden komponensre elvégezte. Mivel a pixelek felhasználása redundáns, így a konverziót is feleslegesen sokszor végezzük el. Ez elkerülhető, ha a float-tá történő konvertálást nem a Shared Memory olvasás, hanem a Shared Memory írás során végezzük el – ezt a műveletet amúgy is Global Memory sávszélessége korlátozza, így a típuskonverzió által jelentett plusz munka gyakorlatilag elhanyagolható. Hátrány viszont, hogy a Shared Memory-ban minden komponenst float-ként tárolunk a byte helyett, így a Thread Block-onként szükséges Shared Memory méret négyszereződik, és ugyanaz igaz a sávszélesség-igényre is. Az adatszélesség változása a Shared Memory elrendezést is megváltoztatja, így a bank ütközés kérdését is újra meg kell vizsgálni. A 11. ábra jelölései megegyeznek a 10. ábra jelöléseivel.



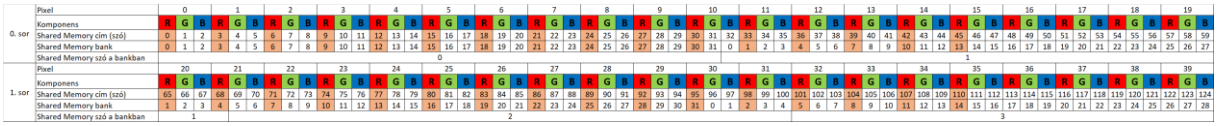
11. ábra Shared Memory elrendezés float komponensek esetén

A 11. ábrából látható módon a 0. warp első 16 szála a számítás kezdetén az N*3 címről olvas ezen címek az (N*3)%32 bankban foglalnak helyet. Ebből következően egy pixel soron belül nincs bank ütközés, még úgy sem, hogy a 0…10 pixelek adata az adott bank 0. szavában található, míg a 11. pixeltől kezdődően az adatok az adott bank 1. szavában vannak, hiszen az olvasott értékek más-más bankban találhatók. Nem igaz ugyanez, ha a második sort (16…31 szálak) is figyelembe vesszük, hiszen itt az olvasás címzése nem lineárisan folytatódik (ennek oka, hogy a soraink nem 16, hanem 20 pixelt

tartalmaznak), pl. a 32. pixel R komponense ugyanúgy a 0. bank-ban van, mint a 0. pixel R komponense, ezek olvasása tehát bank ütközéshez vezet.

Implementálja az elvben olvasási bank ütközéssel rendelkező kódot, és vizsgálja meg az eredményt NVIDIA Visual Profiler-ben!

A bank ütközés elkerülésére két megoldás kínálkozik.
Az első, hogy minden egyes sort kiegészítünk „dummy adatokkal” úgy, hogy a következő sor megfelelő Shared Memory bankban kezdődjön. A kiegészítés kód szempontjából relatíve egyszerű, csak megfelelő méretű Shared Memory-t kell foglalni, aminek lesznek nem használt részei. Ennek hátránya, hogy a szükséges dummy adatok mennyiségétől függően jelentősen növelheti a Shared Memory használatot. Egy megfelelő memória kiosztást a 12. ábra mutat. Vegyük észre, hogy az 1. sor az 59. címen végződik, a második sor pedig a 65. címen kezdődik, azaz 5 szónyi „dummy adat” kerül beszúrásra soronként, ami nem egész számú többszöröse a pixelek számának, így a memória deklaráció esetében nem élhetünk a triviális, több dimenziós tömb deklarációval. E helyett megfelelő méretű 1D tömb deklarációjára van szükség, az ezen belüli címzést megvalósítása a C kódunk feladata.



12. ábra Bank ütközés elkerülése „dummy adatok” beiktatásával

A másik megoldás a fentebb már megállapított észrevételen alapul. Azaz, ha tudjuk, hogy egy soron belül nincs bankütközés, akkor érjük el, hogy az egy warp-ban levő szálak adott olvasási utasítás végrehajtásakor ugyanazt a sort olvassák. Ez megtehető, ha megváltoztatjuk a Thread Block vízszintes méretét 32-re, így egy warp a kép-részlet egy sorának feldolgozásáért felel. Ez természetesen maga után vonja a Shared Memory méretének változását is, pl. 32x8-as Thread Block esetén (32+4)*(8+4) méretű ablakot kell beolvasni egy Thread Block-hoz.

Implementálja ezt a verziót is, és vizsgálja meg az eredményt Visual Profiler-ben!

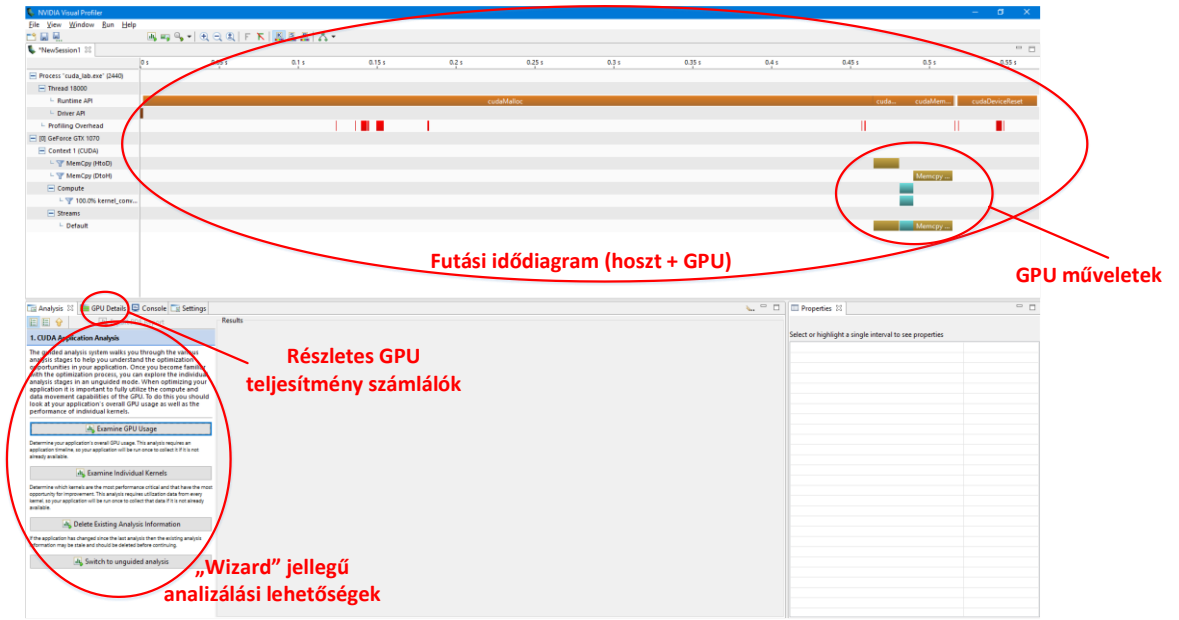
2.5 További gyorsítási lehetőségek

A további gyorsítási lehetőségek hasonlóak, mint a Mátrix szorzás példa esetében:

- Optimálisabb Global Memory hozzáférést kell kialakítani, azaz minimalizálni kell a Read Request-ek és Write Request-ek számát, ami úgy érhető el, ha egy Warp szálai egymást követő 128 byte-ot írnak vagy olvasnak lehetőleg 128 byte-os határra eső kezdőcímről.
- Növelni kell a tényleges feldolgozást jelentő aritmetikai utasítások és a többi utasítás arányát, valamint – GPU-tól függően – az utasítás szintű párhuzamosságot. Ezek bizonyos határig megvalósíthatók pl. úgy, hogy növeljük az egy szál által végzett számítások mennyiségét.

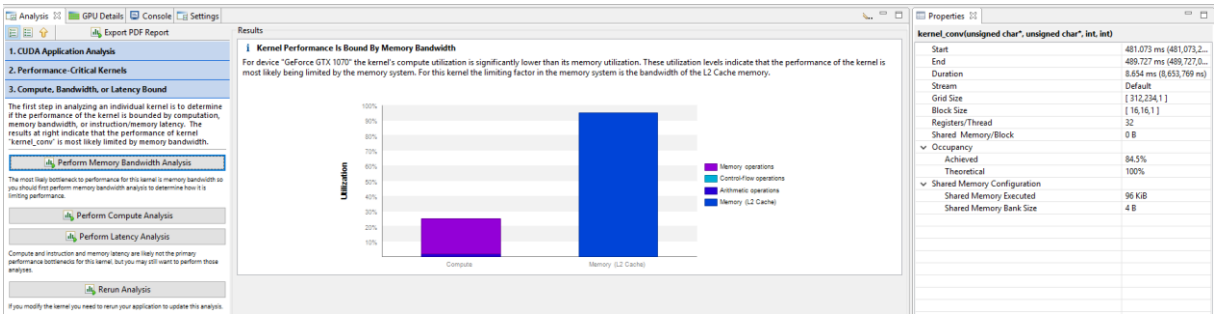
2.6 NVIDIA Visual Profiler

Az NVIDIA Visual Profiler lehetőséget nyújt a megírt kód teljesítményének analizálására, ami a GPU-ból kiolvasott „Performance Counter”-eken alapul. Ez tipikusan azzal jár, hogy a lefordított .exe fájlt a profiler jónéhányszor lefuttatja. A Visual Profiler ablakot a 12. ábra mutatja. A bal oldali, „Wizard jellegű” analizálás során a Profiler a kódot különböző aspektusokból vizsgálja, és tanácsot is ad a gyorsításra (de ez nem mindig betartandó!). A „GPU Details” fülön a GPU-ból kiolvasott értékek táblázatos formában tekinthetőek meg, az egyes „Performance Counter”-ek jelentését a Help tartalmazza.



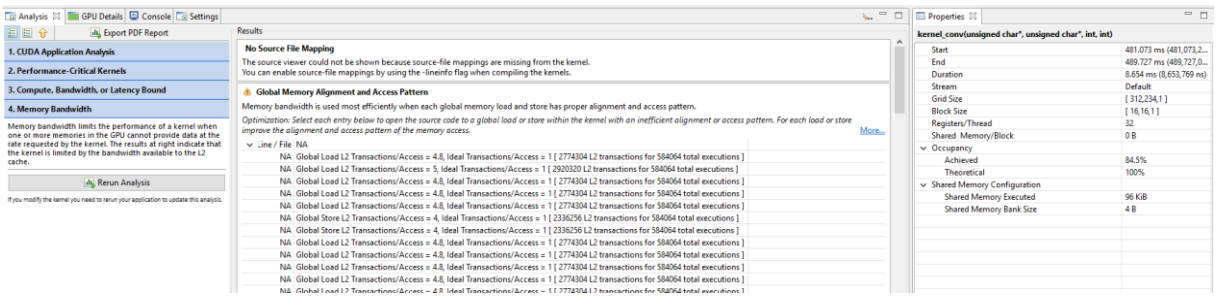
13. ábra A Visual Profiler főablaka

A legelső (Global Memory) kódunk „Examine Individual Kernels”, majd „Memory Bandwidth”, „Compute Analysis” és „Latency Analysis” eredményei a következők.



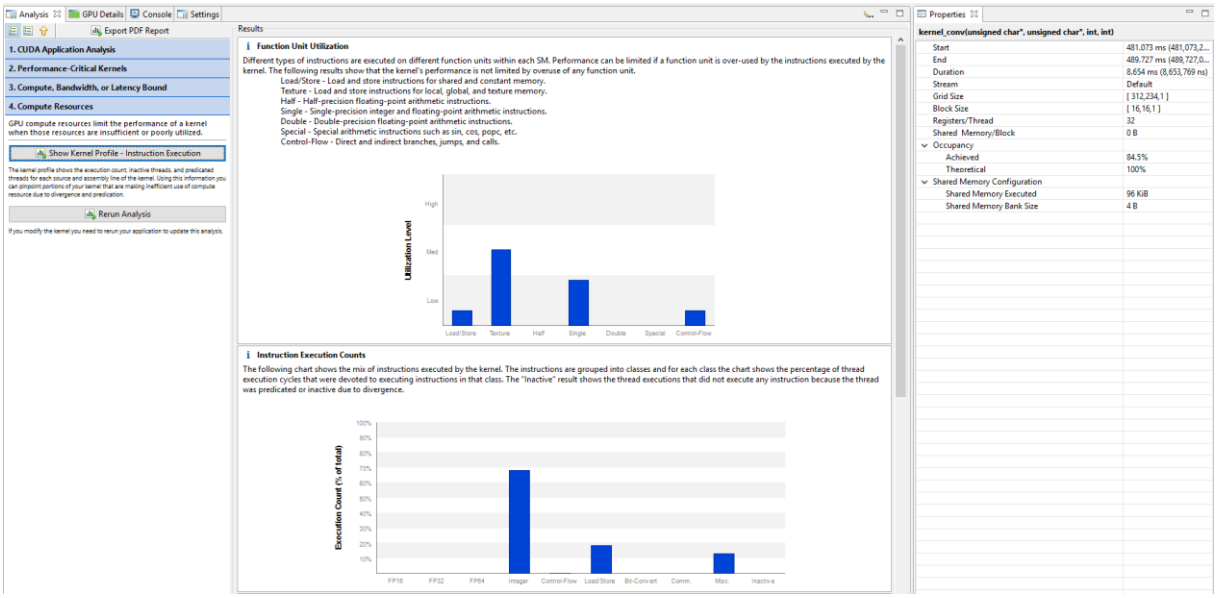
14. ábra Kernel analízis eredménye

Az ábrából azt látjuk, amire amúgy is számíthattunk: a teljesítményt a memória műveletek korlátozzák. A memória átvitel kihasználtsága ugyan magas, a Streaming Multiprocessor-oké viszont alacsony, és az általuk végrehajtott utasítások nagy része is memória művelet, azaz az idő nagy részében az aritmetikai elemke nem csinálnak semmit.



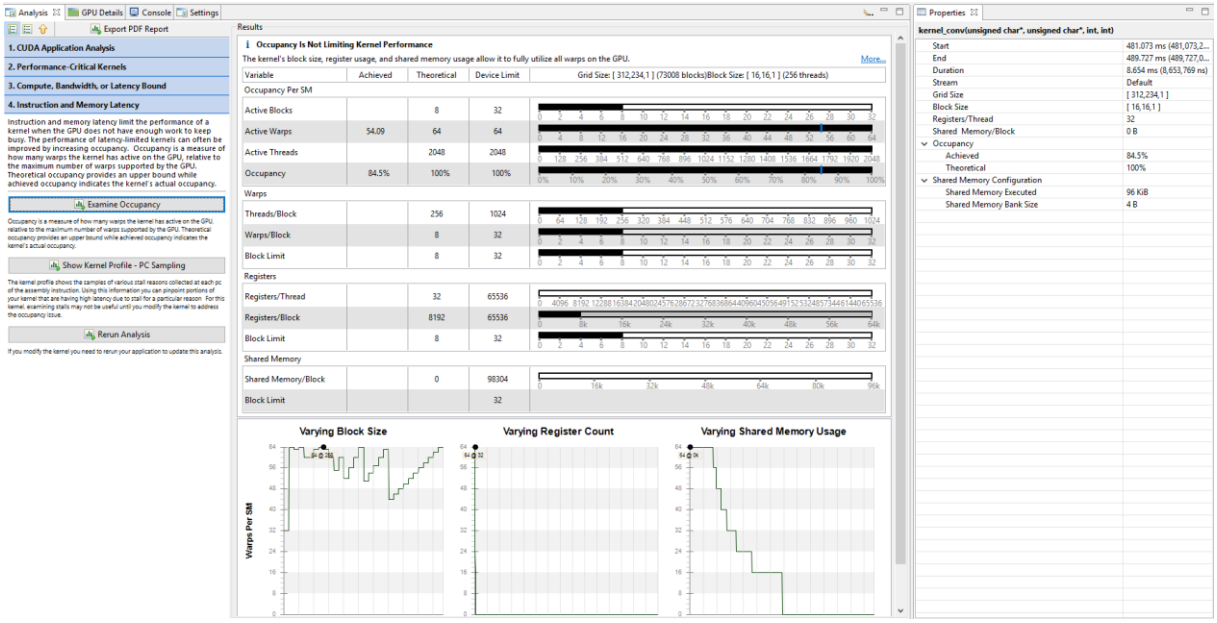
15. ábra Memory Bandwidth analízis eredménye

A memória műveletek elemzése azt mutatja, hogy a Global Memory hozzáférés nem ideális, ami ismerve a kódunkat egyértelmű. Az ideális az lenne, ha minden warp 128 byte-os határról kezdődően végezne 128 byte-os memória műveletet, ez esetünkben nagyon nem teljesül.



16. ábra Compute Analysis eredménye

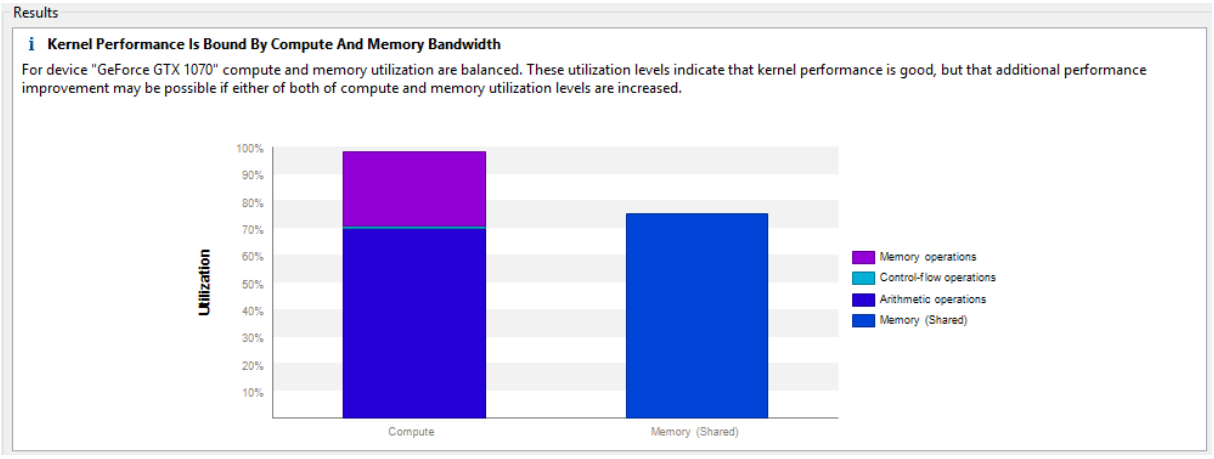
A „Compute Analysis” eredményéből azt látjuk, hogy az utasítások nagy része „Texture” utasítás, ami jelenthet Local, Global és Texture Memory hozzáférést, esetünkben a másodikról van szó. A „Load/Store” utasítások a Constant Memory-ból történő együttható töltésekre vonatkoznak (Shared Memory-t nem használunk), míg a „Single” oszlop az integer műveletvégzésekért felelős utasítások kihasználtságát mutatják.



17. ábra Latency Analysis eredménye

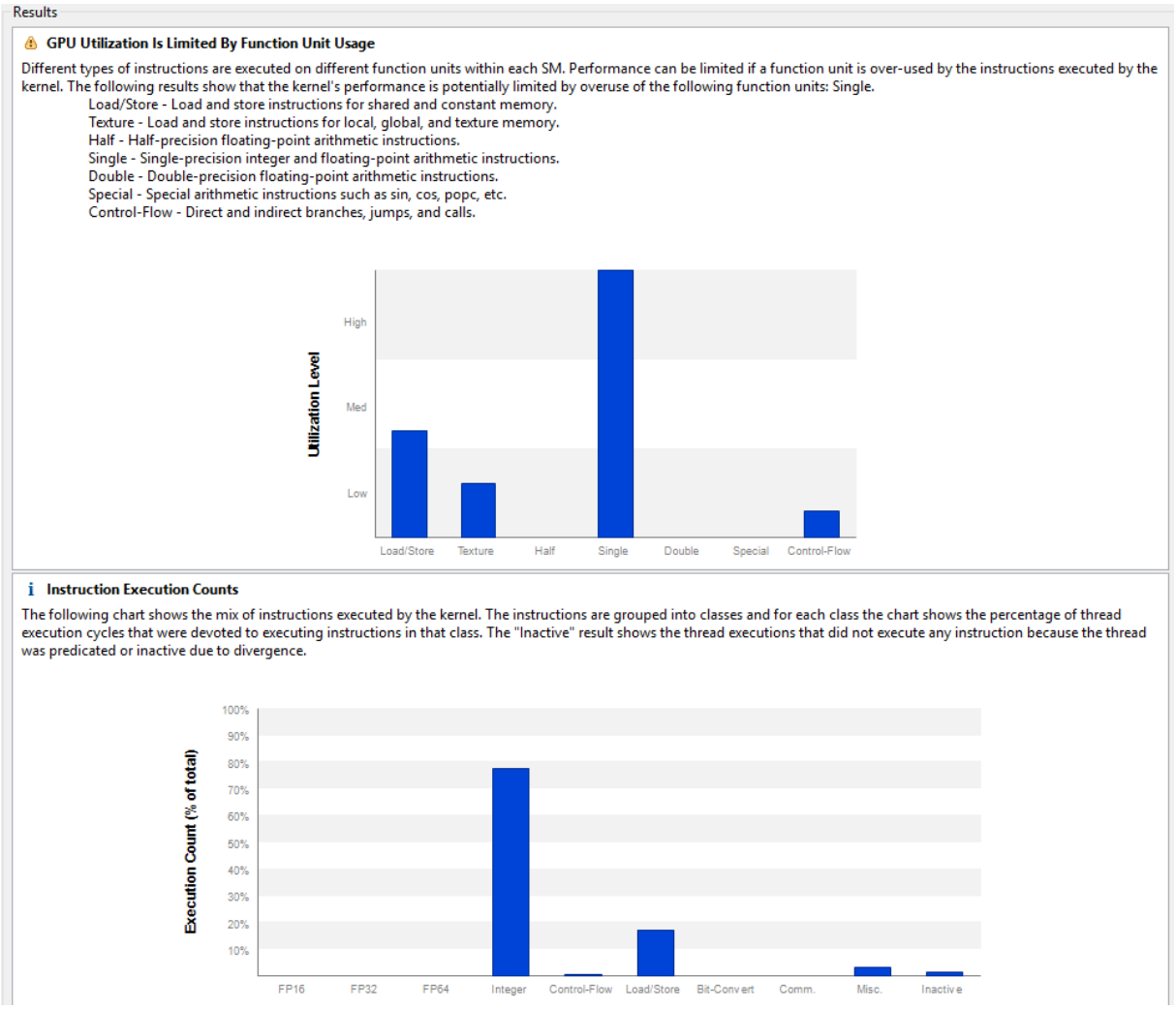
A „Latency Analysis” eredménye az, hogy elegendően nagy számú szál tud egyszerre futni, hogy a memória és pipeline késleltetések ne befolyásolják negatívan a teljesítményt: az „Occupancy” 84,5%, elegendően magas érték.

A 2.2 (Shared Memory) verzióban a kihasználtság sokkal kedvezőbben alakul:



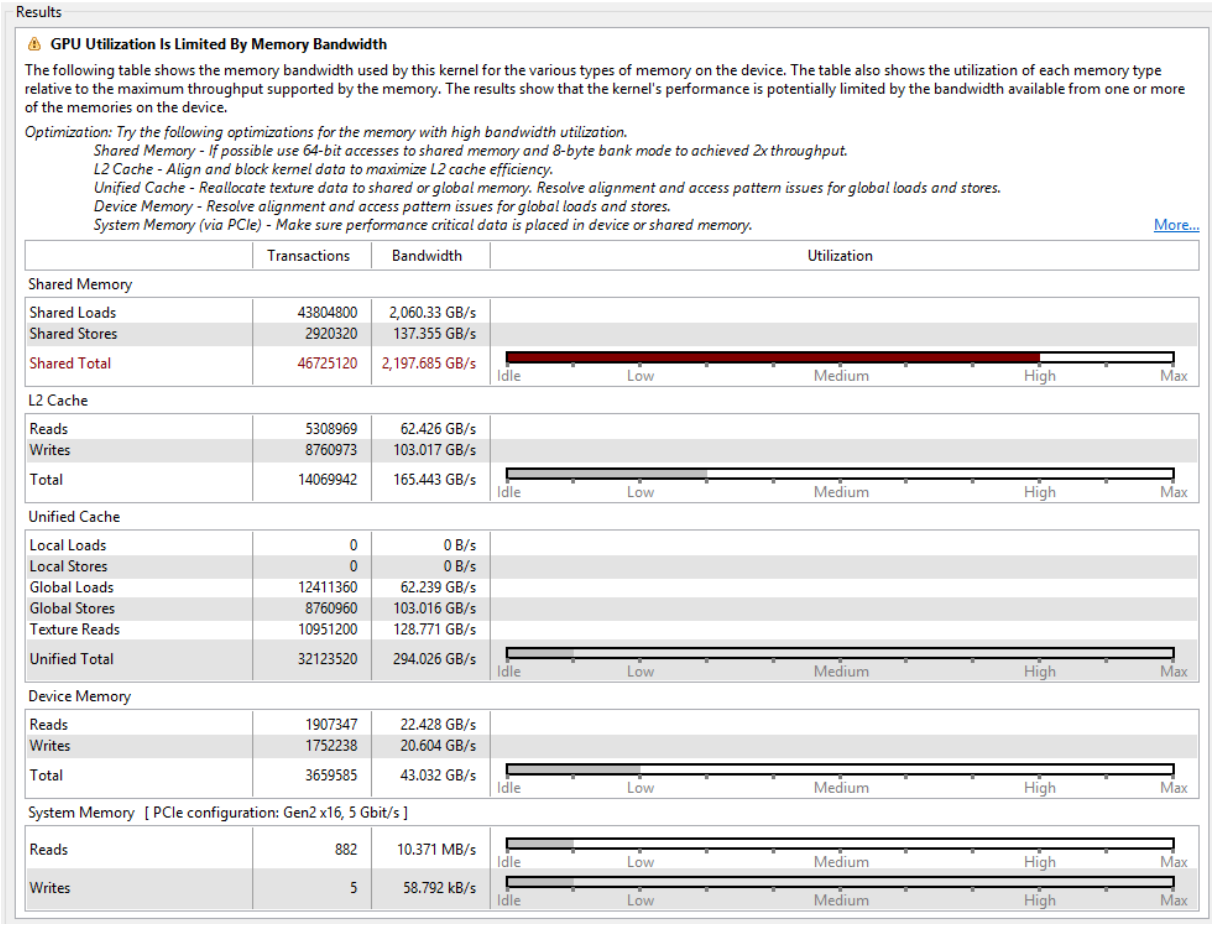
18. ábra Kihasználtság a 2.2 verzióban

Látható, hogy az aritmetikai egységek kihasználtsága jelentősen megnőtt.



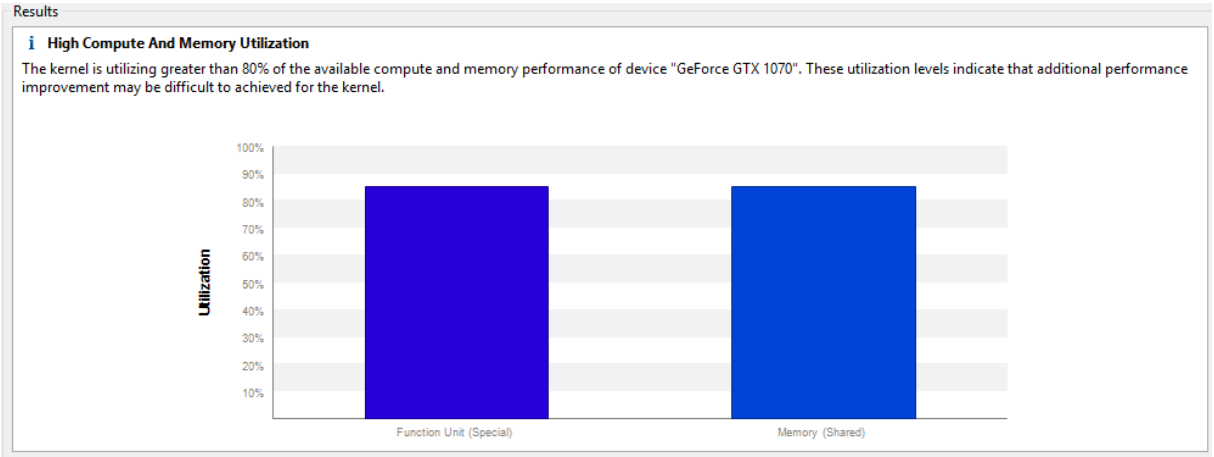
19. ábra Kernel és Compute Analysis a 2.2-es kódon

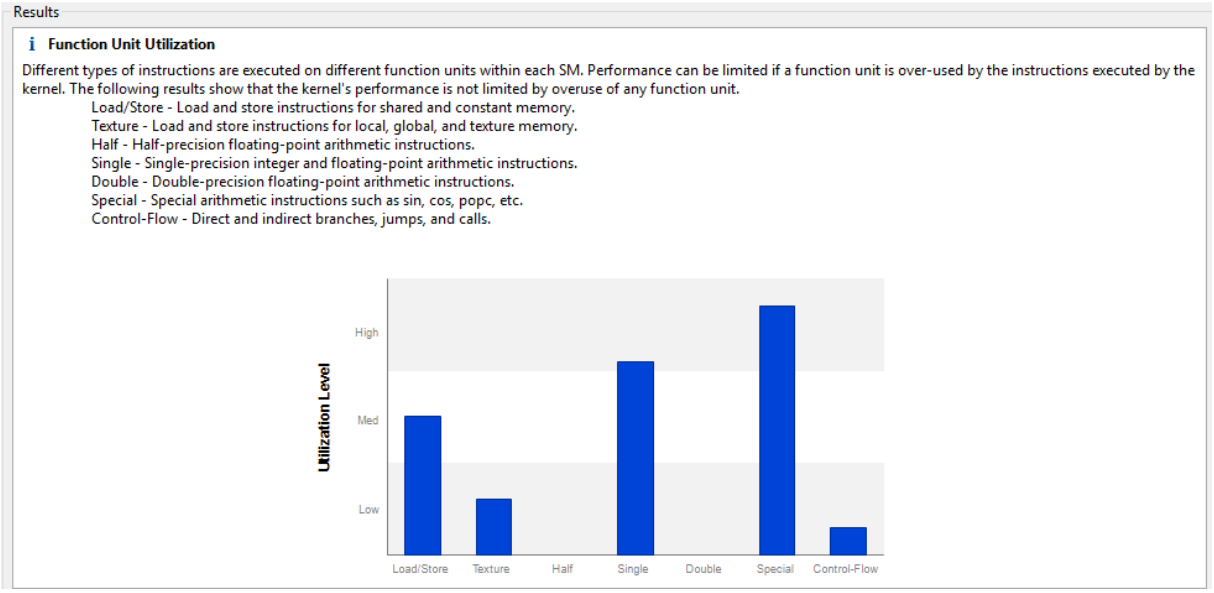
Az egyes végrehajtók kihasználtsága erősen a „Single” típusú utasítások felé mozdult, ide tartoznak az integer utasítások. A memória műveletek relatív mennyisége jelentősen csökkent.



20. ábra Memory Bandwidth a 2.2-es kódon

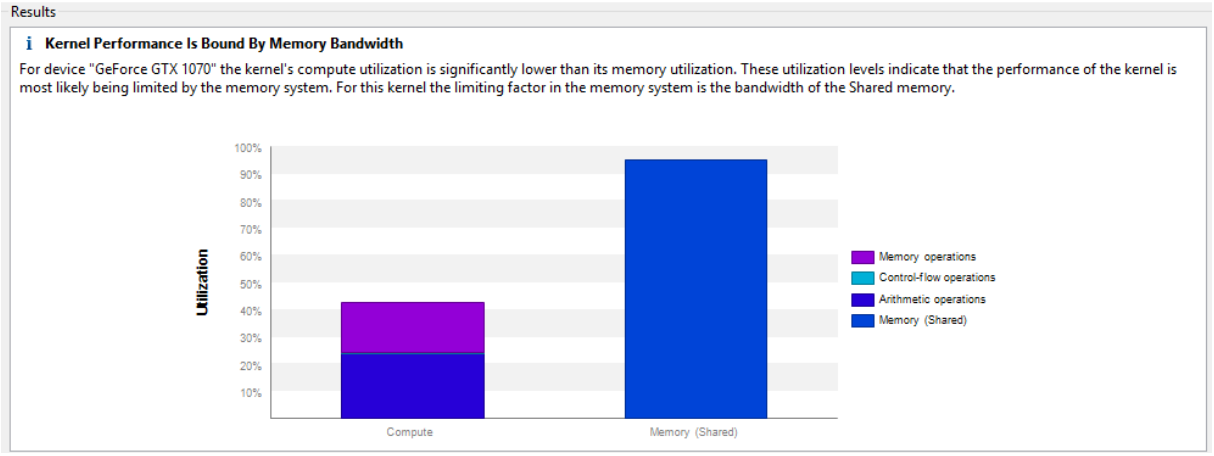
Áttérve a 2.3 kódra, (Shared Memory olvasás során float-ba konvertálás) a profiling eredményéből azt vehetjük észre, hogy a speciális utasítások kihasználtsága (ide tartozik a típuskonverzió) nagyon megnőtt.





21. ábra Kihasználtság a 2.3 kódban

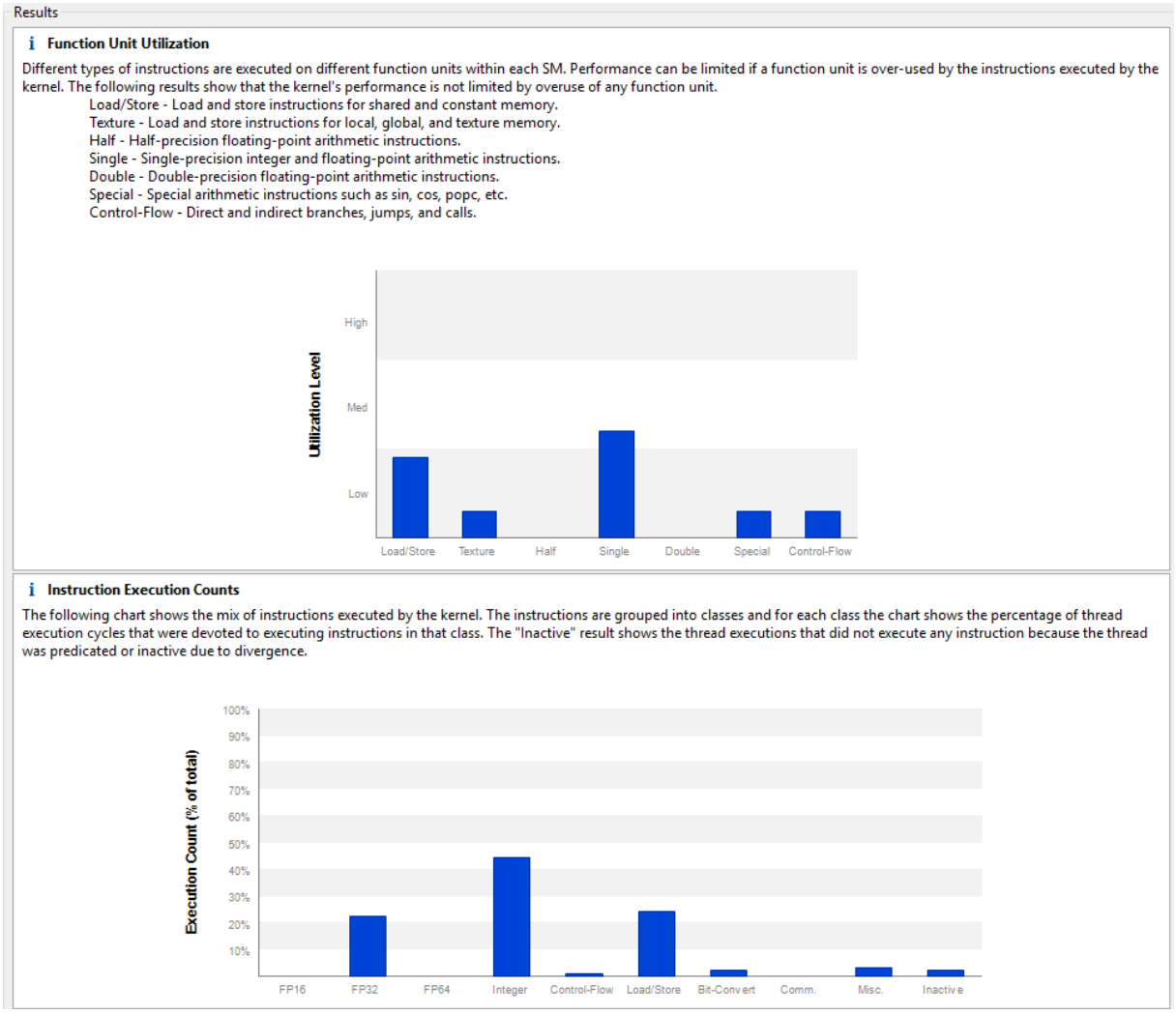
Módosítva a kódot úgy, hogy a típus konverziót a Shared Memory írás közben végezzük (2.4), az alábbi eredményre jutunk.



22. ábra Kihasználtság a 2.4 verzióban

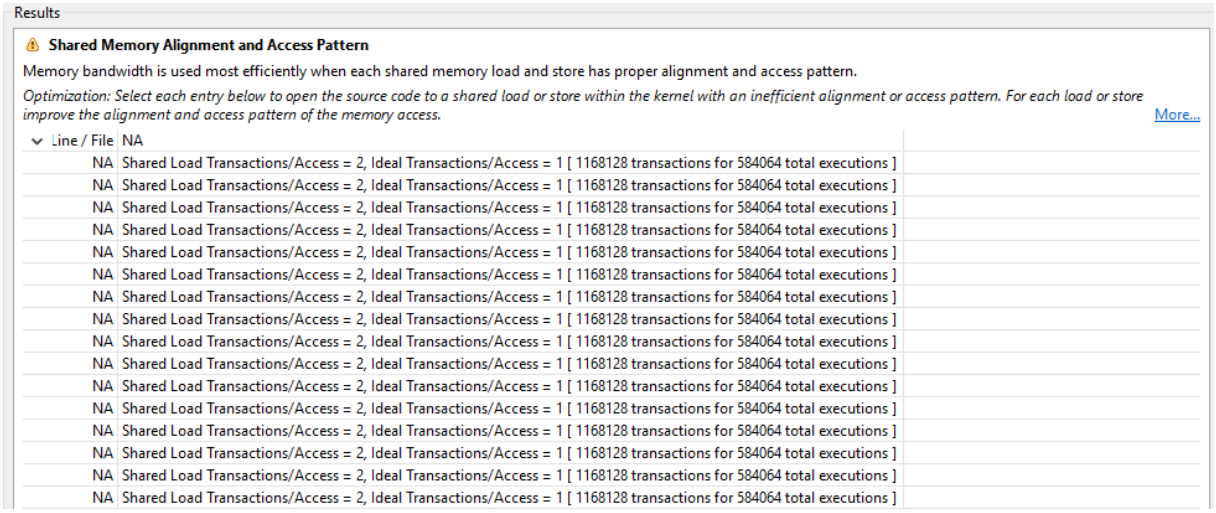
Mint látható, a memória műveletek mennyisége jelentősen megugrott, ez többek között annak köszönhető, hogy most minden komponenst float-ban tárolunk, így a szükséges Shared Memory sávszélesség a négyszeresére nőtt.

A részletesebb eredményekben ugyanakkor azt is láthatjuk, ami a várakozásunk volt: a típus konverziós utasítások mennyisége jelentősen csökkent.



23. ábra Utasítások aránya a 2.4 verzióban

Végül a „Memory Bandwidth” analízis felhívja figyelmünket a legnagyobb gondra, a bank ütközésre.



24. ábra Bank ütközés a 2.4 verzióban