Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Heterogén számítási rendszerek házi feladat dokumentáció

Szilágyi Gábor
Neptun: NOMK01

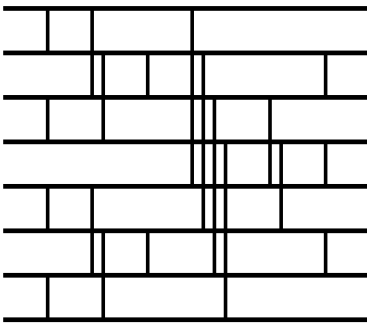
Budapest, 2023. január 7.

Tartalomjegyzék

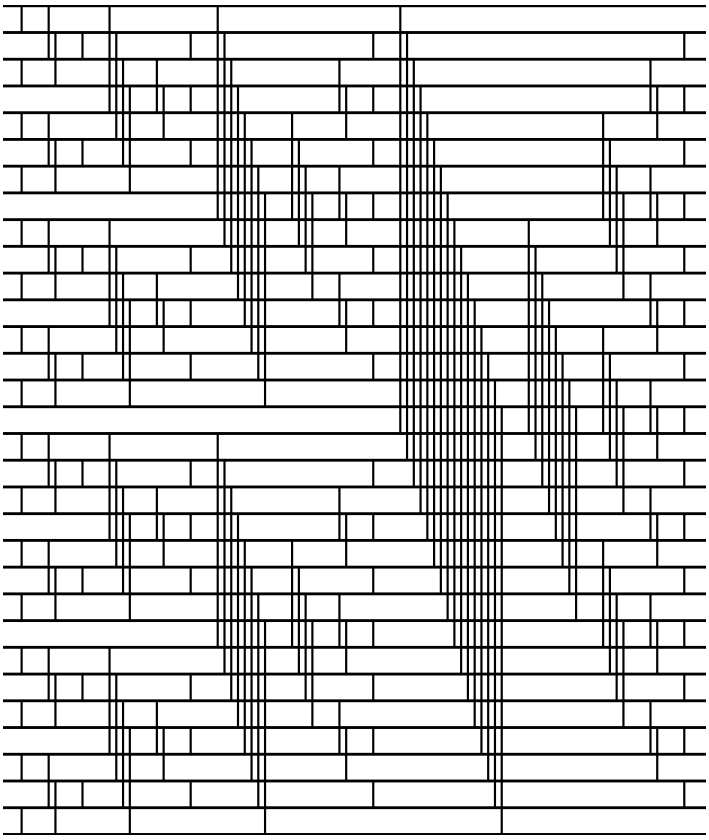
1. A használt algoritmus ismertetése

A házi feladat egy kép pixelein történő medián szűrés 5×5-ös ablakkal, ami 25 elemből a medián kiválasztását jelenti, minden kimeneti pixelhez a három színcsatornára különkülön.

A házi feladatomhoz a Batcher odd-even mergesort algoritmust választottam [4]. Ez az algoritmus eredeti formájában egy $n = 2^k$ elem fölötti rendezés, nem csak mediánkiválasztás, ezzel valamennyi fölösleges része is van, ezt el lehet hagyni az én esetemben. Az algoritmus a rendezendő elemek értékétől független szerkezetű, ezért reprezentálható egy rendezési hálózattal (sorting network). A rendezési hálózat $n = 8$ és $n = 32$ esetén az 1. ábrán látható.



(a) $n = 8$



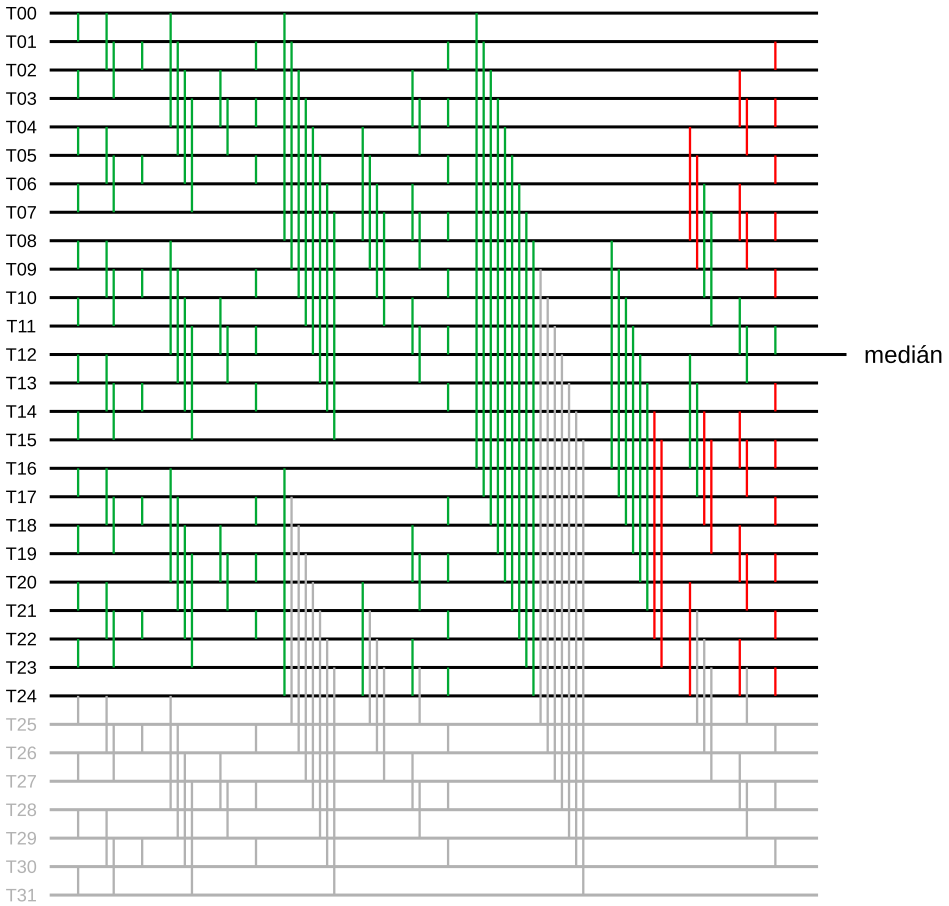
(b) $n = 32$

1. ábra

A hálózatban a vízszintes vonalak az egyes tárolókat jelölik, amikben a rendeződő értékek vannak, a függőleges vonalak az összehasonlítás és csere műveleteket, ami után az összekötött két elem rendezve van. Egy ilyen pár rendezése után vagy a felső a nagyobb, vagy az alsó, minden pár esetén egyformán, hogy konkrétan melyik, az a medián szempontjából mindegy. Az idő „jobbra telik”.

A 25 elem mediánjának kiválasztásához a 32 elemet rendező hálózatból indultam ki. Ebből kitöröltem azokat az összehasonlításokat, amelyekre nincs szükség. Ez az átalakított hálózat a 2. ábrán látható.

Az n elem rendezéséhez szükséges összehasonlítások sorozata algoritmikusan generálható $n = 2^k$-ra [4]. Ez alapján írtam egy egyszerű C-programot, ami az összehasonlítás-



2. ábra. A 25-ös mediánkiválasztáshoz szükséges hálózat.
(zöld: szükséges rendezés, szürke: 32 helyett 25 elem miatt szükségtelen rendezés, piros: teljes rendezés helyett mediánkiválasztás miatt szükségtelen rendezés)

sorozat kódját generálja a különböző megoldásokhoz ($m = 25$-re), de ez a sorozat még tartalmazza a mediánkiválasztáshoz szükségtelen összehasonlításokat.

```
#include <stdio.h>
int main(void)
{
  int n=32, m=25;
  for(int p=1; p<n; p*=2)
    for(int k=p; k>=1; k/=2)
      for(int j=k%p; j<=n-k-1; j+=2*k)
        for(int i=0; i<k; i++)
          if((i+j)/(p*2)==(i+j+k)/(p*2) && i+j+k<m)
            printf("SORT(%d,%d)\n", i+j, i+j+k);
  return 0;
}
```

2. Sztenderd C implementáció

A szűrőprogram első megvalósítása egy egyszerű skalár C program, ami referenciaként szolgál a többi megvalósításhoz, így meg lehet tudni, hogy az egyes hardveres gyorsítási módszerek hogyan teljesítenek. A kép a memóriában egy `char` tömbként van tárolva, három egymásmelletti bájt jelenti egy pixel három színkomponensét.

Ennél a változatnál nem foglalkoztam az összehasonlítások mediánszámítás miatti lecsökkentésével, csak a 25 db elemen kívüliekre vonatkozókat hagytam el. A fentebb leírt algoritmus szerint generálom futás közben a következőnek rendezendő elempár indexeit. Ezen kívül egyszerűen egyesével végiglépkedek a kép pixelein, ehhez a belső ciklus az x irányú, a külső az y irányú.

A sztenderd C implementáció forráskódja a B.4 függelékben látható.

3. AVX2 + OpenMP implementáció

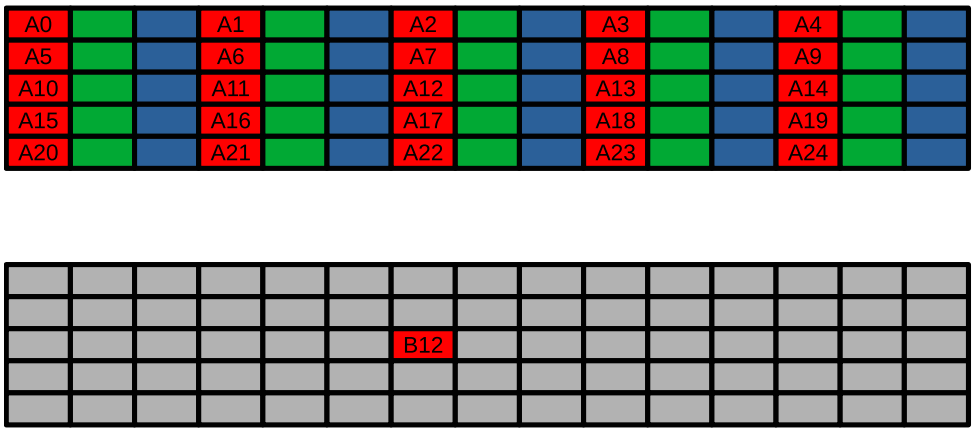
Ehhez a megoldáshoz egyrészt az AVX2 utasításkészlet vektorutasításait használtam fel, másrészt az OpenMP API-t, aminek a segítségével többszálúsítható a képfeldolgozás.

3.1. AVX2

Az AVX2 utasításkészlet 256 bites vektorutasításokat tartalmaz többféle vektoradattípusra, többek között 32 db `uint8`-ból, más néven `char`-ból álló vektorra is. Fontos, hogy létezik maximum, és minimumkiválasztó vektorutasítás `uint8` vektorokra, ezek adják az algoritmus lelkét. Két vektorregiszter (`a` és `b`) elemeinek páronkénti rendezése a következő séma szerint történik egy harmadik ideiglenes vektorregiszter (`tmp`) segítségével. Ezután `a` i-edik eleme kisebb vagy egyenlő, mint `b` i-edik eleme.

```
1  temp = _mm256_max_epu8(a,b);
2  a = _mm256_min_epu8(a,b);
3  b = temp;
```

Az AVX2-es gyorsítás alapgondolata az volt, hogy mindig, amikor a skalár kódban egy bájton dolgozik a program, akkor ehelyett a vektorizált kódban 32 db, egymás melletti bájton dolgozzon párhuzamosan. Ezzel egy mediánkiválasztás végrehajtása gyakorlatilag ugyanúgy zajlik, mint skalár esetben, csak a végén nem egy bájt az eredmény, hanem 32 egymás melletti bájt. Ez a 32 egymás melletti bájt nem feltétlenül pixelhatáron kezdődik vagy végződik, de ez nem okoz gondot, mivel a különböző színcsatornákra teljesen függetlenül kell elvégezni a számítást. A memóriahozzáférés szemléltetéséhez segít a 3. ábra.

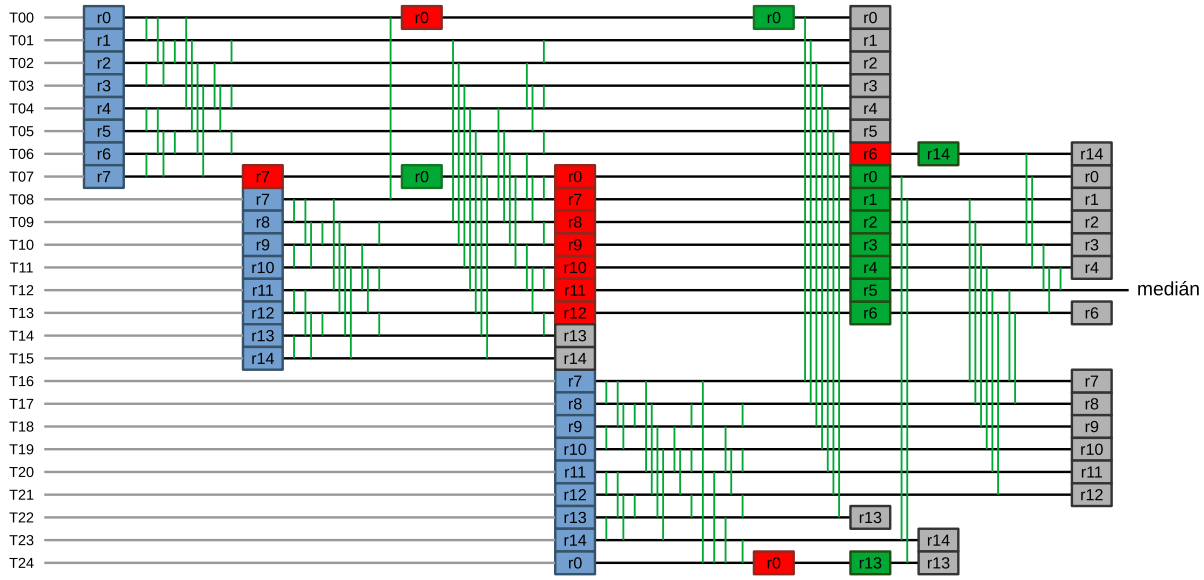


3. ábra. A memóriahozzáférés szemléltetése.

A 3. ábrán az A_x-szel jelölt téglalapok azt a 25 címet jelölik a memóriában, ahonnan a bemeneti képből olvasni kell, hogy előállíthassuk a kimeneti kép B_{12} című színkomponensét. Például $A_1 = A_0 + 3$ és $A_5 = A_0 + 3 \times W_F$, ahol W_F a kiterjesztett bemeneti

kép szélessége pixelben. A skalár kódban ténylegesen ezekről a címekről kellene olvasni egy-egy bájtot, majd a 25 db beolvasott bájt mediánját, ami szintén egy bájt, kellene kiírni a B_{12}-es címre. Ehelyett a vektorizált kódban az A_x-től az $A_x + 31$-ig terjedő 32 bájt hosszú tartományt olvasom be és a mediánok vektorát a B_{12}-től a $B_{12} + 31$-ig terjedő területre írom ki. Ezután a következő olvasássorozatban a bal felső sarokhoz tartozó olvasás a jelenlegi $A_0 + 32$-es címen fog kezdődni, tehát 32 bájtosával lépegetek. A sorvégekkel nem kell foglalkozni, mert a kimeneti kép szélessége pixelben 32-vel osztható.

Mivel az AVX2-es utasításkészletet támogató, de AVX-512 nélküli processzorokban csak 16 db vektorregiszter van, ezért nem fér el mind a 25 db beolvasott, plusz a rendezéshez használt ideiglenes tároló vektorváltozó regiszterekben, így a medián keresése során bizonyos változókat ideiglenesen ki kell írni memóriába majd visszaolvasni. Ez a kiírás-visszaolvasás akkor is megtörténne, ha csak rábíznám a fordítóra, de valamit lehet javítani a futási időn, ha a saját kezembe veszem a dolgot és explicit módon leírom a kódba, hogy mikor melyik regiszterben melyik vltozó legyen és mikor történjenek a kiírások és a visszaolvasások, mert így csökkenthető a memóriahozzáférések száma. Ezzel az is együtt jár, hogy gyakorlatilag kézzel unrollolnom kell az összehasonlítások sorozatát generáló kódrészt és még ezt is meg kell variálnom a vektorregiszterek kezelése miatt. Ennek az unrollolásnak az a nagy előnye, hogy az összehasonlítandó változók indexeinek kiszámításának overheadje teljesen megszűnik és a mediánkiválasztás közben nincs elágazás a kódban, ami elég hatékony végrehajtást tesz lehetővé. A vektorregiszterek menedzselését a 4. ábra szemlélteti.



4. ábra. A vektorregiszterek menedzselése.

A 4. ábrán T_{XX} és a mellette lévő vízszintes vonal jelöli az egyes vektorváltozókat, a függőleges zöld vonalak pedig a rendezéseket két ilyen változó között. A kódban az egyes regiszterek szigorúan véve egy-egy változóként vannak deklarálva, de itt változó alatt azokat az adatvektorokat értem, amik közben néha ki is íródnak a memóriába és az elemeik össze-vissza cserélődnek, de magát az adatvektort végig számon kell tartani. A kék téglalap, benne egy regiszter nevével (r_{XX}) azt jelenti, hogy akkor olvassom be a bemeneti képből az adott vektorváltozót, ami egy unaligned load utasítást jelent. A piros téglalap benne egy regiszter nevével azt jelenti, hogy a regiszter tartalmát (ami az aktuális vízszintes vonalhoz tartozó változó) kiírom egy kis ideiglenes tároló tömbbe egy aligned

store utasítás segítségével, hogy ez a regiszter használható legyen egy másik változó tárolására. A szürke téglalap hasonló dolgot jelent, mint a piros, ez is az adott regiszter felszabadulását jelenti, azzal a különbséggel, hogy itt nincs szükség store utasítással kiírni az ideiglenes tömbbe az adott r_{XX} regiszter tartalmát, mert az aktuálisan benne tárolt változóra innentől már nincs szükség, egyszerűen felülírható egy másik változóval. Végül pedig a zöld téglalap, ami a piros párja, az aligned load utasítást jelenti, amikor az aktuális vízszintes vonalnak megfelelő változót beolvasom a zöld téglalapban megjelölt r_{XX} regiszterbe. Logikus, hogy minden piros store után ugyanabban a sorban van egy zöld load, hiszen eleve azért írom ki a változót, hogy később visszaolvasva még használjam.

Összesen egy mediánvektor (32 bájt) kiszámolásához 25 db unaligned loadra (kék), 11 db aligned store-ra (piros), 11 db aligned loadra (zöld) és 1 db unaligned store-ra van szükség. És emellett természetesen 113 db páronkénti rendezésre a vektorváltozók között.

A vektorutasításokat az Intel intrinsic függvényeken keresztül tudom használni. A kódomban a következő intrinsic-eket használtam vagy próbáltam ki [2].

```
_mm256_max_epu8(a, b)          // maximum selection
_mm256_min_epu8(a, b)          // minimum selection
_mm256_lddqu_si256(addr)       // unaligned load
_mm256_storeu_si256(addr, a)   // unaligned store
_mm256_load_si256(addr)        // aligned load
_mm256_store_si256(addr, a)    // aligned store
_mm_malloc(size, align)        // aligned malloc
_mm_free(addr)                 // aligned free
_mm256_stream_si256(addr, a)   // aligned store to RAM, bypassing cache
_mm_prefetch(addr, i)          // prefetch cache line containing addr
  // prefetch into non-temporal cache: i = _MM_HINT_NTA
```

Az aktuális mediánszámításhoz szükséges cache line-okat az algoritmus elkezdése előtt prefetch-elem, hogy potenciálisan kevesebb időt kelljen várni a loadokra a számolás közben. Ennek a változtatásnak tapasztalatom szerint nagyon kicsi, de pozitív hatása van. Kipróbáltam különböző prefetch módokat is (az `i` változó értéke), minimális különbséget tapasztaltam, a fent látható non-temporal cache-be való olvasás nyert. Ahol lehet, ott unaligned load helyett aligned load-ot használok a kódban, de tapasztalatom szerint ennek is csak minimális hatása van a futási időre. Az egyik leginkább kritikus pontja a programnak a kiszámolt mediánok kirása a memóriába. Itt ismert, hogy mindig 32 bájtra aligned címre kell írni, így használható valamilyen aligned store. A `_mm256_stream_si256()` és `_mm256_store_si256` intrinsic-ek közül az előbbit, a cache-t kikerülő verziót találtam kicsivel gyorsabbnak.

3.2. OpenMP

Az algoritmus többszálúsítása már valamivel egyszerűbb az OpenMP pragmakészlet segítségével. A kód párhuzamosítással foglalkozó része a következő.

```
register __m256i r00, r01, r02, r03, r04, r05, r06, r07,\
         r08, r09, r10, r11, r12, r13, r14, tmp;
int y_out, x_rgb_out;

#pragma omp parallel private( y_out, x_rgb_out, \
  r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14, tmp) \
  shared(imgHeight, imgWidth, imgWidthF, imgSrcExt, imgDst)
{
  // array to temporarily store register contents when registers need to be freed
  __m256i* arr = (__m256i*)_mm_malloc(25 * 32, 32);

#pragma omp for schedule(dynamic)
  for (y_out = 0; y_out < imgHeight; y_out++)
  {
    for (x_rgb_out = 0; x_rgb_out < imgWidth * 3; x_rgb_out += 32)
```

```
16      {
17        // calculate median
18        // store median to output
19      }
20    }
21    _mm_free(arr);
22  }
```

A fenti kódrészletben az első `#pragma` a párhuzamos rész kezdete, itt explicite megadom a változókról, hogy melyik shared és melyik private. A képen belüli „navigáláshoz" használt ciklusváltozókból minden szálnak sajátra van szüksége, mert ezzel indexelik azt a részt a bemeneti és kimeneti memóriaterületeken ahol éppen számolnak. Vektorregiszterekből (`r00` - `r14`, `tmp`) minden szálnak saját példányokra van szüksége, mert független számításokat végeznek más-más adatokon, így ezek is privátok lesznek. A bemeneti és kimeneti kép címei és méretei pedig jó, ha shared változók, mert ezeket csak olvassák az egyes szálak és a bennük tárolt információkra mindegyiknek szüksége van. Elvileg lehetne probléma abból, hogy több szál akar ugyanonnan olvasni, de hamar minden szál becache-eli ezt a néhány adatot magának, így nem interferálnak egymással az olvasások során. Annyit még megjegyzek, hogy az `arr` ideiglenes tömb lefoglalása és felszabadítása is a párhuzamos blokkon belül zajlik, tehát ebből a tömbből is minden szálnak saját, független példánya van.

A következő `#pragma` a külső, kép sorain iteráló ciklusra vonatkozik, annak az iterációit osztja fel a szálak között. A `schedule(dynamic)` pragma a ciklus iterációinak szálak közötti szétosztását vezérli. Gyakorlatilag n db szál és H sorból álló kép esetén minden szálnak egy folytonos, $\lceil H/n \rceil$ sorból álló sáv jut a képből. Ezáltal minimális lesz azoknak a memóriaterületeknek a mérete két ilyen sáv határánál (határonként 4 sor), amelyeket a bemeneti képből mindkét szál olvas valamikor, így esélye van az ütközésnek. Egyébként az ütközés gyakorlatilag kizárt, mert amennyire értem, a saját sávjukon belül minden szál fentről lefelé halad, így amikor egy határvonal fölött dolgozó szál a határ közelébe ér, a határvonal alatt dolgozó szál már régen eltávolodott onnan.

Kipróbáltam azt is, amikor az egy soron belüli iterálásra vonatkozik a párhuzamosítás és egyben ez a külső ciklus, így körülbelül 30%-kal romlott a program teljesítménye, így az előző verziónál maradtam. Azért lehet jogos, hogy romlik a teljesítmény, ha függőleges sávokra osztom a képet, mert amikor egy szál már végigért a sávja egy során és éppen kezdi a következőt, de a tőle balra lévő sávhoz tartozó szál még éppen a sora végén jár, akkor átlapolódás van a két szál által olvasott memóriaterületek között. Ennek fényében nem gondolom, hogy ennek a problémának 30%-os sebességromlást kellene okoznia.

Az AVX2 + OpenMP implementáció forráskódja a B.4. függelékben látható.

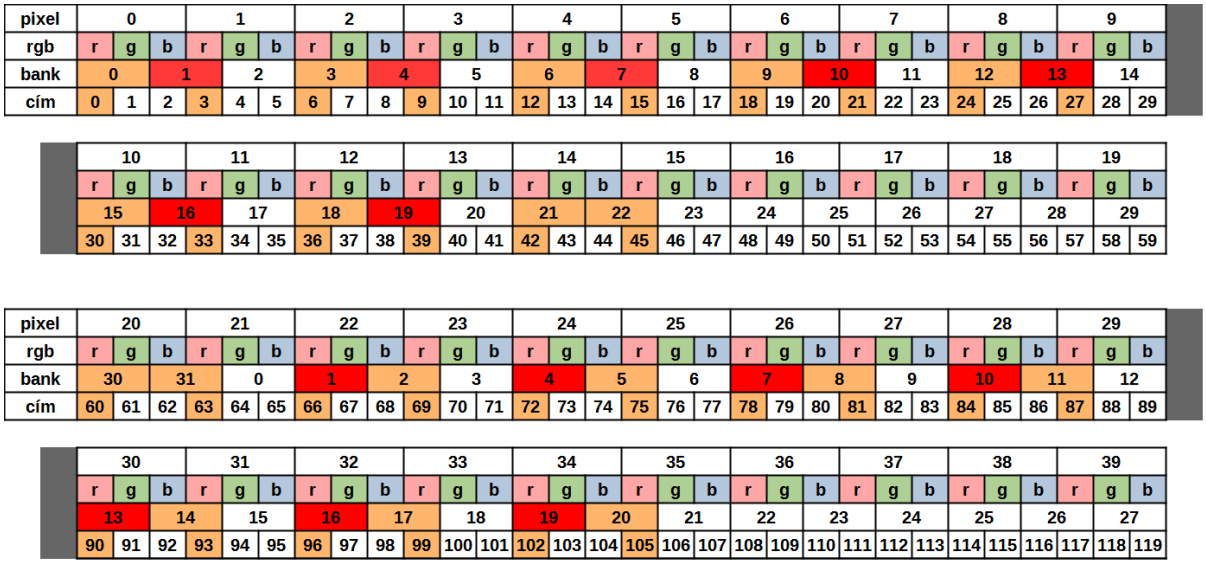
4. OpenCL implementáció

Az OpenCL verzió half precision float (`half`) típust használ, mert a teszteléshez használt GPU is és az osztályozáshoz használt GPU is ilyen típussal tud a leggyorsabban számolni. Az előbbi a teszteléshez használt processzorba integrált GPU (Intel UHD Graphics for 11th Generation Intel Processors, UHD 730 [1]), az utóbbi pedig egy NVIDIA TITAN Xp videókártya lesz, ha minden igaz (Pascal architektúra, [3]). Ezen kívül az is szempont volt a `half` típus használatánál, hogy viszonylag kis fejfájással és kicsivel több shared memória felhasználásával meg lehet oldani, hogy elvben ne történjen bankütközés a shared memória olvasásakor.

A számítás menete a gyakorlaton megismerttel gyakorlatilag megegyezik. Először a thread block (workgroup) szálai bemásolják a globális memóriából a szükséges részt a shared (local) memóriába, közben átkonvertálják a beolvasott `char` értékeket `half` típusra, a shared memóriában már így tárolódik a beolvasott adat. Ezután bevárják egymást, majd minden szál a saját regisztereibe olvas 25 db `half` értéket a shared memóriából, ezeken megkeresi a mediánt, majd az eredményt visszakonvertálja `char` típusra és kiírja a globális memóriába. Ezt a shared memória feltöltés utáni részt minden szál a hozzá tartozó pixel 3 színcsatornájára végzi el egymás után, vagyis szálanként 3 bájtnyi kimenet keletkezik.

A bankütközés elkerülése `half` adattípusok esetén a következőképpen történik. Ha egyszerűen egy háromdimenziós `half` tömböt deklarálok a következőképpen, akkor az 5. ábrán látható módon lesznek ütközések. (Az ábrákon a sorokat félbe törtem, hogy olvasható legyen. A szürkített részeknél kell összeragasztani a félsorokat.)

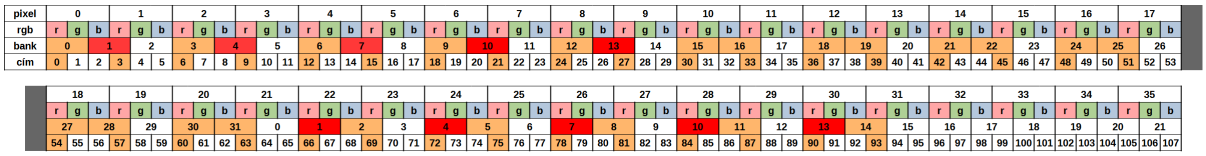
```
1 __local shmem[20][20][3];
```



5. ábra. Banütközések 16 × 16-os blokkméretnél, half precision float típussal.

Az 5. ábrán a „pixel” sor jelenti az adott pixel lineáris sorszámát a thread block bemeneti régióján belül. Az „rgb” sorban az adott pixelek színkomponensei vannak felsorolva. A „bank” sorban az látszik, hogy a fölötte lévő színkomponens melyik bankba esik a shared memóriában. A „cím” sorban pedig a shared memóriában lévő tömbön belüli lineáris cím van, ahol az adott komponens található. Narancssárga jelöli azokat a címeket és a hozzájuk tartozó bankokat, amikről először olvasnak egy warp szálai (32 db). Pirossal pedig azokat a bankokat jelöltem, amiknek az elérésekor bankütközés történik.

A fenti alapértelmezett elrendezésnél kb. minden második olvasás jár bankütközéssel, ami nem tragikus, de a bankütközéseket el lehet kerülni teljesen. Az sem sokat javít a helyzeten, ha 32×8-asra változtatom a blokkméretet, ennek a hatása a 6. ábrán látható.

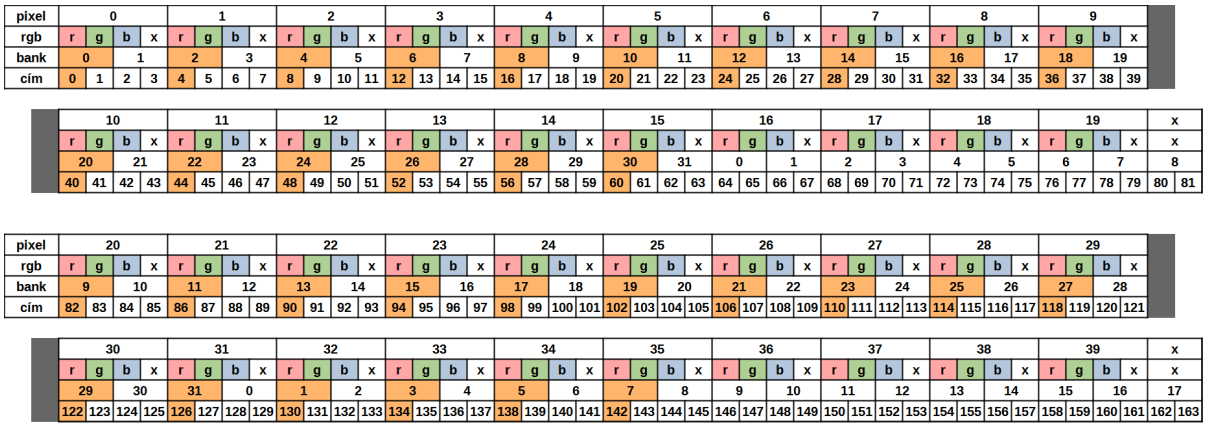


6. ábra. Banütközések 32×8-as blokkméretnél, half precision float típussal.

Végül az jelenti a megoldást, ha maradok a 16×16-os blokkméretnél és a shared memória deklarációját a következőre cserélem.

```
__local half shmem[20*4+2][20];
```

Ez azt jelenti, hogy bemeneti pixelenként nem három, hanem négy színcsatornányi hellyel számolok (a negyediket nem használom) és még a sorok végén két `half`-nyi, ezzel éppen egy banknyi üres helyet hagyok. Ezzel azt érem el, hogy a párhuzamos olvasásoknál minden második bank van használatban, tehát az első sorban a páros indexűek, a második sorban ugyanilyen séma szerint éppen a páratlan indexűek lesznek használatban, így nincs bankütközés. Ez a változat a 7. ábrán látható.



7. ábra. Nincs bankütközés 16×16-as blokkméretnél, half precision float típussal, negyedik virtuális színcsatornával és sorvégi offsettel.

A negyedik virtuális színcsatorna vagy a sorvégi offset magában nem elég a bankütközések elkerülésére, 16×16-os és 32×32-es blokkméretek mellett sem.

A fenti shared memória deklarációval a következőképpen módosul a shared memória indexelése az eredeti, háromdimenziósnak deklarált tömbhöz képest.

```
shmem[x][y][rgb] -> shmem[x*4+rgb][y]
```

Az OpenCL implementáció forráskódja a C.4. függelékben látható.

5. Futási idők

A teszteléshez használt hardver egy Intel i5-11400 processzor fix 2600 MHz-es órajellel, 3200 MHz-es DDR4 RAM-mal. A használt GPU ugyanennek a processzornak az integrált GPU-ja, ami ugyanezt a RAM-ot használja (Intel UHD 730 Graphics [1]).

Implementáció	Futások száma	Egy futás átlagos ideje	MP/s
C	10	90 430,6000 ms	0,2067
AVX2 + OpenMP referencia	1000	13,6100 ms	1373,2585
AVX2 + OpenMP	1000	13,2500 ms	1410,5697
OpenCL referencia	100	69,8705 ms	267,4954
OpenCL	500	42,8274 ms	436,4044

Hivatkozások

[1] Intel. Intel Processor Graphics Xe-LP API Developer and Optimization Guide. https://www.intel.com/content/www/us/en/developer/articles/guide/lp-api-developer-optimization-guide.html.

[2] Intel. Intrinsics guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[3] NVIDIA. Pascal gpu architecture whitepaper v1.2. https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/.

[4] Wikipedia. Batcher odd-even mergesort. https://en.wikipedia.org/wiki/Batcher_odd%E2%80%93even_mergesort.

A. Sztenderd C implementáció kódja

(Ez a változat ugyanazokat az egyéb fájlokat használja, mint az AVX2+OpenMP implementáció, csak másik függvény hívódik meg a main-ben.)

A.1. median_filter.cpp

```
#include "defs.h"
#include "stdio.h"

// after sort2(a,b,temp): a>=b
#define sort2(a, b, t) \
  (t) = ((a)<(b))?(a):(b); \
  (a)=((a)>(b))?(a):(b); \
  (b)=(t);

void median_filter(int imgHeight, int imgWidth, int imgWidthF,\
  unsigned char *imgSrcExt, unsigned char *imgDst)
{
  unsigned char read[75], temp;
  for (int yout = 0; yout < imgHeight; yout++)
  {
    for (int xout = 0; xout < imgWidth; xout++)
    {
      // filling up "read" with rgb bytes from 5*5 pixels
      for (int rgb=0; rgb < 3; rgb++)
        for (int dy = 0; dy < 5; dy++)
          for (int dx = 0; dx < 5; dx++)
            read[(dy * 5 + dx) * 3 + rgb] = imgSrcExt[((yout+dy)*imgWidthF+xout+dx)*3+rgb];

      // finding medians for r, g and b
      for (int rgb=0; rgb < 3; rgb++)
        for (int p = 1; p < 32; p=p*2)
          for (int k = p; k >=1 ; k/=2)
            for (int j = k % p; j <= 32 - 1 - k; j += 2 * k)
              for (int i = 0; i < k; i++)
                if ((i + j) / (p * 2) == (i + j + k) / (p * 2))
                  // not doing comparisons outside of 25 elements
                  if (i + j + k < 25)
                  {
                    sort2(read[(i + j + k) * 3 + rgb], read[(i + j) * 3 + rgb], temp)
                  }

      // writing 3 medians to output image
      for (int rgb = 0; rgb < 3; rgb++)
        imgDst[(yout * imgWidth + xout) * 3 + rgb] = read[12 * 3 + rgb];
    }
  }
}
```

B. AVX2 + OpenMP implementáció kódja

B.1. defs.h

```
#define FILTER_W 5
#define FILTER_H 5

#define RUNS 1000

#define USE_OMP 1
```

B.2. func.h

```
void median_filter(int imgHeight, int imgWidth, int imgWidthF, unsigned char* imgSrcExt,
    unsigned char* imgDst);

void median_filter_avx_omp(int imgHeight, int imgWidth, int imgWidthF, unsigned char*
    imgSrcExt, unsigned char* imgDst);
```

B.3. main.cpp

```
// lab1.cpp : Defines the entry point for the console application.
//

//#include "stdafx.h"
#include "memory.h"
#include "time.h"

#include "omp.h"

#include <IL/ilut.h>
#include <IL/ilu.h>

#include "emmintrin.h"
#include "nmmintrin.h"

#include "defs.h"
#include "func.h"


void main()
{
  ilInit(); iluInit();
  ILboolean ret;
  ILuint ilImg=0;
  ilGenImages(1, &ilImg);
  ilBindImage(ilImg);
    ret = ilLoadImage((const char*)("input.jpg"));
  ILubyte* imgData = ilGetData();

  int imgWidth = ilGetInteger(IL_IMAGE_WIDTH);
  int imgHeight = ilGetInteger(IL_IMAGE_HEIGHT);
  ILint imgOrigin = ilGetInteger(IL_ORIGIN_MODE);

  printf("Input resolution: %4dx%4d\n", imgWidth, imgHeight);

  unsigned char *imgSrcExt;
  int imgWidthF = imgWidth+FILTER_W-1;
  int imgHeightF = imgHeight+FILTER_H-1;
  int imgFOfssetW = (FILTER_W-1)/2;
  int imgFOfssetH = (FILTER_H-1)/2;
  imgSrcExt = (unsigned char *)(_aligned_malloc(3*imgWidthF*imgHeightF*sizeof(unsigned char),
    32));
    int row, col;

```

```
  for (row=0; row<imgHeightF; row++)
  {
    for (col=0; col<imgWidthF;col++)
    {
      int pixel = (row*imgWidthF + col)*3;
      *(imgSrcExt + pixel + 0) = 0;
      *(imgSrcExt + pixel + 1) = 0;
      *(imgSrcExt + pixel + 2) = 0;
    }
  }

  for (row=0; row<imgHeight; row++)
  {
    for (col=0; col<imgWidth;col++)
    {
      int pixel_dst = ((row+imgFOfssetH)*imgWidthF + (col+imgFOfssetW))*3;
      int pixel_src = (row*imgWidth + col)*3;
      *(imgSrcExt + pixel_dst + 0) = (unsigned char)(*(imgData + pixel_src + 0));
      *(imgSrcExt + pixel_dst + 1) = (unsigned char)(*(imgData + pixel_src + 1));
      *(imgSrcExt + pixel_dst + 2) = (unsigned char)(*(imgData + pixel_src + 2));
    }
  }

  unsigned char *imgRes;
  imgRes = (unsigned char *)(_aligned_malloc(3 * imgWidth*imgHeight * sizeof(unsigned char),
    32));

// IMAGE PROCESSING
//-------------------------------------------------------------------------------------------
  clock_t s0, e0;
    double d0;

  double mpixel;

  short *imgDstConv;
  imgDstConv = (short*)(_aligned_malloc(3 * imgWidthF*imgHeightF * sizeof(short), 32));

  printf("Start median filtering, %d runs\n",RUNS);

#if 0
  s0 = clock();
for (int r=0; r<RUNS; r++)
{
  median_filter(imgHeight, imgWidth, imgWidthF, imgSrcExt, imgRes);
}

    e0 = clock();
    d0 = (double)(e0-s0)/(CLOCKS_PER_SEC);
  mpixel = (imgWidth*imgHeight/d0)/1000000*RUNS;
    printf("C CPU TIME: %4.4f s\n", d0);
  printf("C 1 RUN:    %4.4f ms\n", d0 * 1000 / RUNS);
  printf("C Mpixel/s: %4.4f\n", mpixel);
#endif

#if 1
      s0 = clock();
for (int r=0; r<RUNS; r++)
{
  median_filter_avx_omp(imgHeight, imgWidth, imgWidthF, imgSrcExt, imgRes);
}
    e0 = clock();
    d0 = (double)(e0-s0)/(CLOCKS_PER_SEC);
  mpixel = (imgWidth*imgHeight/d0)/1000000*RUNS;
  printf("AVX+OpenMP CPU TIME: %4.4f s\n", d0);
  printf("AVX+OpenMP 1 RUN:    %4.4f ms\n", d0 * 1000/RUNS);
  printf("AVX+OpenMP Mpixel/s: %4.4f\n", mpixel);
#endif

//-------------------------------------------------------------------------------------------
// IMAGE PROCESSING END

  for (row=0; row<imgHeight; row++)
  {
```

```
116     for (col=0; col<imgWidth;col++)
117     {
118       int pixel_src = (row*imgWidth + col)*3;
119       int pixel_dst = (row*imgWidth + col)*3;
120       *(imgData + pixel_dst + 0) = (ILubyte)(*(imgRes + pixel_src + 0));
121       *(imgData + pixel_dst + 1) = (ILubyte)(*(imgRes + pixel_src + 1));
122       *(imgData + pixel_dst + 2) = (ILubyte)(*(imgRes + pixel_src + 2));
123     }
124   }
125
126   _aligned_free(imgDstConv);
127   _aligned_free(imgSrcExt);
128     _aligned_free(imgRes);
129
130   ret = ilSetData(imgData);
131   ilEnable(IL_FILE_OVERWRITE);
132     ilSaveImage((const char*)("output.jpg"));
133   ilDeleteImages(1, &ilImg);
134 }
```

B.4. median_filter_avx_omp.cpp

```
1  #include "defs.h"
2  #include "stdio.h"
3  #include "omp.h"
4
5  #include <IL/ilut.h>
6  #include <IL/ilu.h>
7
8  #include "emmintrin.h"
9  #include "nmmintrin.h"
10 #include "immintrin.h"
11
12 // after sort2(a,b,temp): a>=b
13 #define sort2(a, b, temp) \
14   (temp) = _mm256_min_epu8 ((a), (b)); \
15   (a) = _mm256_max_epu8 ((a), (b)); \
16   (b) = (temp);
17
18 // after sort8(...): d0>=d1>=d2>=...>=d7
19 #define sort8(d0,d1,d2,d3,d4,d5,d6,d7,temp) \
20   /*sort 4 pairs*/\
21   sort2(d0, d1, temp) \
22   sort2(d2, d3, temp) \
23   sort2(d4, d5, temp) \
24   sort2(d6, d7, temp) \
25   /*merge 4 pairs into 2 groups of 4 sorted elements*/\
26   sort2(d0, d2, temp) \
27   sort2(d1, d3, temp) \
28   sort2(d4, d6, temp) \
29   sort2(d5, d7, temp) \
30   sort2(d1, d2, temp) \
31   sort2(d5, d6, temp) \
32   /*merge groups of 4 to sorted group of 8*/\
33   sort2(d0, d4, temp) \
34   sort2(d1, d5, temp) \
35   sort2(d2, d6, temp) \
36   sort2(d3, d7, temp) \
37   sort2(d2, d4, temp) \
38   sort2(d3, d5, temp) \
39   sort2(d1, d2, temp) \
40   sort2(d3, d4, temp) \
41   sort2(d5, d6, temp)
42
43
44 void median_filter_avx_omp(int imgHeight, int imgWidth, int imgWidthF, unsigned char *
       imgSrcExt, unsigned char *imgDst)
45 {
46   // r00...r14 are used to store image data, tmp is only used as temporary register during
       sorting
```

```
47    register __m256i r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14,
        tmp;
48    int y_out, x_rgb_out;
49
50  #if USE_OMP == 1
51    #pragma omp parallel private( y_out, x_rgb_out, \
52      r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14, tmp) \
53      shared(imgHeight, imgWidth, imgWidthF, imgSrcExt, imgDst)
54  #endif
55    {
56      // array to temporarily store register contents when registers need to be freed
57      __m256i* arr = (__m256i*)_mm_malloc(25 * 32, 32);
58  #if USE_OMP == 1
59      #pragma omp for schedule(dynamic)
60  #endif
61      for (y_out = 0; y_out < imgHeight; y_out++)
62      {
63        for (x_rgb_out = 0; x_rgb_out < imgWidth * 3; x_rgb_out += 32)
64        {
65          /*prefetch all of the necessary cache lines*/
66          _mm_prefetch((char const*)(imgSrcExt + imgWidthF * (y_out + 0) * 3 + x_rgb_out),
      _MM_HINT_NTA);
67          _mm_prefetch((char const*)(imgSrcExt + imgWidthF * (y_out + 0) * 3 + x_rgb_out + 32),
      _MM_HINT_NTA);
68          _mm_prefetch((char const*)(imgSrcExt + imgWidthF * (y_out + 1) * 3 + x_rgb_out),
      _MM_HINT_NTA);
69          _mm_prefetch((char const*)(imgSrcExt + imgWidthF * (y_out + 1) * 3 + x_rgb_out + 32),
      _MM_HINT_NTA);
70
71
72          /* load and sort the first and second 8 long group of values*/
73
74          // load the first 8 values (A00...A07 into regs r00...r07)
75          r00 = _mm256_load_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 0) * 3 + x_rgb_out
       + 0 * 3));
76          r01 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 0) * 3 +
      x_rgb_out + 1 * 3));
77          r02 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 0) * 3 +
      x_rgb_out + 2 * 3));
78          r03 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 0) * 3 +
      x_rgb_out + 3 * 3));
79          r04 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 0) * 3 +
      x_rgb_out + 4 * 3));
80          r05 = _mm256_load_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 1) * 3 + x_rgb_out
       + 0 * 3));
81          r06 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 1) * 3 +
      x_rgb_out + 1 * 3));
82          r07 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 1) * 3 +
      x_rgb_out + 2 * 3));
83        // values:    A00, A01, A02, A03, A04, A05, A06, A07, xxx, xxx, xxx, xxx, xxx, xxx, xxx
84        // registers: r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14
85
86          _mm_prefetch((char const*)(imgSrcExt + imgWidthF * (y_out + 2) * 3 + x_rgb_out),
      _MM_HINT_NTA);
87          _mm_prefetch((char const*)(imgSrcExt + imgWidthF * (y_out + 2) * 3 + x_rgb_out + 32),
      _MM_HINT_NTA);
88          _mm_prefetch((char const*)(imgSrcExt + imgWidthF * (y_out + 3) * 3 + x_rgb_out),
      _MM_HINT_NTA);
89          _mm_prefetch((char const*)(imgSrcExt + imgWidthF * (y_out + 3) * 3 + x_rgb_out + 32),
      _MM_HINT_NTA);
90          _mm_prefetch((char const*)(imgSrcExt + imgWidthF * (y_out + 4) * 3 + x_rgb_out),
      _MM_HINT_NTA);
91          _mm_prefetch((char const*)(imgSrcExt + imgWidthF * (y_out + 4) * 3 + x_rgb_out + 32),
      _MM_HINT_NTA);
92
93
94          // sort first 8 values
95          sort8(r00, r01, r02, r03, r04, r05, r06, r07, tmp)
96
97          // store A07 in arr and load next 8 values
98          // (load A08...A15 into regs r07...r14)
99          _mm256_store_si256(arr + 7, r07);
```

```
100         r07 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 1) * 3 +
    x_rgb_out + 3 * 3));
101         r08 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 1) * 3 +
    x_rgb_out + 4 * 3));
102         r09 = _mm256_load_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 2) * 3 + x_rgb_out
     + 0 * 3));
103         r10 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 2) * 3 +
    x_rgb_out + 1 * 3));
104         r11 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 2) * 3 +
    x_rgb_out + 2 * 3));
105         r12 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 2) * 3 +
    x_rgb_out + 3 * 3));
106         r13 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 2) * 3 +
    x_rgb_out + 4 * 3));
107         r14 = _mm256_load_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 3) * 3 + x_rgb_out
     + 0 * 3));
108       // values:    A00, A01, A02, A03, A04, A05, A06, A08, A09, A10, A11, A12, A13, A14, A15
109       // registers: r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14
110
111         // sort the next 8 values
112         sort8(r07, r08, r09, r10, r11, r12, r13, r14, tmp)
113
114
115         /* merge the first and secont sorted 8 long groups */
116
117         // sort A00 and A08
118         sort2(r00, r07, tmp)
119
120         // store A00 from r00 and load A07 to r00
121         _mm256_store_si256(arr + 0, r00);
122         r00 = _mm256_load_si256(arr + 7);
123       // values:    A07, A01, A02, A03, A04, A05, A06, A08, A09, A10, A11, A12, A13, A14, A15
124       // registers: r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14
125
126         // sort value pairs   (A01,A09), ... , (A06,A14), (A07,A15)
127         // sort register pairs  (r01,r08), ... , (r06,r13), (r00,r14)
128         sort2(r01, r08, tmp)
129         sort2(r02, r09, tmp)
130         sort2(r03, r10, tmp)
131         sort2(r04, r11, tmp)
132         sort2(r05, r12, tmp)
133         sort2(r06, r13, tmp)
134         sort2(r00, r14, tmp)
135
136         // sort value pairs   (A04,A08), (A05,A09), (A06,A10), (A07,A11)
137         // sort register pairs  (r04,r07), (A05,A08), (r06,r09), (r00,r10)
138         sort2(r04, r07, tmp)
139         sort2(r05, r08, tmp)
140         sort2(r06, r09, tmp)
141         sort2(r00, r10, tmp)
142
143         // sort value pairs   (A02,A04), (A03,A05), (A06,A08), (A07,A09), (A10,A12), (A11,A13)
144         // sort register pairs  (r02,r04), (r03,r05), (r06,r07), (r00,r08), (r09,r11), (r10,
    r12)
145         sort2(r02, r04, tmp)
146         sort2(r03, r05, tmp)
147         sort2(r06, r07, tmp)
148         sort2(r00, r08, tmp)
149         sort2(r09, r11, tmp)
150         sort2(r10, r12, tmp)
151
152         // sort value pairs   (A01,A02), (A03,A04), (A05,A06), (A07,A08), (A09,A10), (A11,A12)
    , (A13,A14)
153         // sort register pairs  (r01,r02), (r03,r04), (r05,r06), (r00,r07), (r08,r09), (r10,
    r11), (r12,r13)
154         sort2(r01, r02, tmp)
155         sort2(r03, r04, tmp)
156         sort2(r05, r06, tmp)
157         sort2(r00, r07, tmp)
158         sort2(r08, r09, tmp)
159         sort2(r10, r11, tmp)
160         sort2(r12, r13, tmp)
161
```

```
162
163         /* load and sort the last 9 values */
164
165         // store values A07...A13 (in regs r0,r7,...,r12)
166         // values in r13 and r14 don't need to be stored, these regs can be reused
167         _mm256_store_si256(arr + 7, r00);
168         _mm256_store_si256(arr + 8, r07);
169         _mm256_store_si256(arr + 9, r08);
170         _mm256_store_si256(arr + 10, r09);
171         _mm256_store_si256(arr + 11, r10);
172         _mm256_store_si256(arr + 12, r11);
173         _mm256_store_si256(arr + 13, r12);
174       // values:    xxx, A01, A02, A03, A04, A05, A06, xxx, xxx, xxx, xxx, xxx, xxx, xxx, xxx
175       // registers: r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14
176
177         // load values A16...A24 to regs r07...r14,r00
178         r07 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 3) * 3 +
    x_rgb_out + 1 * 3));
179         r08 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 3) * 3 +
    x_rgb_out + 2 * 3));
180         r09 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 3) * 3 +
    x_rgb_out + 3 * 3));
181         r10 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 3) * 3 +
    x_rgb_out + 4 * 3));
182         r11 = _mm256_load_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 4) * 3 + x_rgb_out
     + 0 * 3));
183         r12 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 4) * 3 +
    x_rgb_out + 1 * 3));
184         r13 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 4) * 3 +
    x_rgb_out + 2 * 3));
185         r14 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 4) * 3 +
    x_rgb_out + 3 * 3));
186         r00 = _mm256_lddqu_si256((__m256i*)(imgSrcExt + imgWidthF * (y_out + 4) * 3 +
    x_rgb_out + 4 * 3));
187       // values:    A24, A01, A02, A03, A04, A05, A06, A16, A17, A18, A19, A20, A21, A22, A23
188       // registers: r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14
189
190         // sort the first 8 of the new 9 values
191         sort8(r07, r08, r09, r10, r11, r12, r13, r14, tmp)
192
193         // sort value pairs   (A16,A24), (A20,A24), (A18,A20), (A19,A21), (A22,A24)
194         // sort register pairs  (r07,r00), (r11,r00), (r09,r11), (r10,r12), (r13,r00)
195         sort2(r07, r00, tmp)
196         sort2(r11, r00, tmp)
197         sort2(r09, r11, tmp)
198         sort2(r10, r12, tmp)
199         sort2(r13, r00, tmp)
200
201         // sort value pairs   (A17,A18), (A19,A20), (A21,A22), (A23,A24)
202         // sort register pairs  (r08,r09), (r10,r11), (r12,r13), (r14,r00)
203         sort2(r08, r09, tmp)
204         sort2(r10, r11, tmp)
205         sort2(r12, r13, tmp)
206         sort2(r14, r00, tmp)
207
208
209         /* merge first 16 and last 9 sorted elements, find median */
210
211         // store A24 from r00 and load A00 to r00
212         _mm256_store_si256(arr + 24, r00);
213         r00 = _mm256_load_si256(arr + 0);
214       // values:    A00, A01, A02, A03, A04, A05, A06, A16, A17, A18, A19, A20, A21, A22, A23
215       // registers: r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14
216
217         // sort value pairs   (A00,A16), ... , (A06,A22)
218         // sort register pairs  (r00,r07), ... , (r06,r13)
219         sort2(r00, r07, tmp)
220         sort2(r01, r08, tmp)
221         sort2(r02, r09, tmp)
222         sort2(r03, r10, tmp)
223         sort2(r04, r11, tmp)
224         sort2(r05, r12, tmp)
225         sort2(r06, r13, tmp)
```

```
226
227         // store A06 from r06; load A07...A13 to regs r00...r06; load A24 to r13
228         // values in r00...r05,r13 don't need to be stored, these regs can be reused
229         _mm256_store_si256(arr + 6, r06);
230         r00 = _mm256_load_si256(arr + 7);
231         r01 = _mm256_load_si256(arr + 8);
232         r02 = _mm256_load_si256(arr + 9);
233         r03 = _mm256_load_si256(arr + 10);
234         r04 = _mm256_load_si256(arr + 11);
235         r05 = _mm256_load_si256(arr + 12);
236         r06 = _mm256_load_si256(arr + 13);
237         r13 = _mm256_load_si256(arr + 24);
238       // values:    A07, A08, A09, A10, A11, A12, A13, A16, A17, A18, A19, A20, A21, A24, A23
239       // registers: r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14
240
241         // sort value pairs   (A07,A23), (A08,A24)
242         // sort register pairs  (r00,r14), (r01,r13)
243         sort2(r00, r14, tmp)
244         sort2(r01, r13, tmp)
245
246         // load A06 into r14
247         // value in r14 don't need to be stored, this reg can be reused
248         r14 = _mm256_load_si256(arr + 6);
249       // values:    A07, A08, A09, A10, A11, A12, A13, A16, A17, A18, A19, A20, A21, A24, A06
250       // registers: r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14
251
252         // sort value pairs   (A08,A16), ..., (A13,A21)
253         // sort register pairs  (r01,r07), ..., (r06,r12)
254         sort2(r01, r07, tmp)
255         sort2(r02, r08, tmp)
256         sort2(r03, r09, tmp)
257         sort2(r04, r10, tmp)
258         sort2(r05, r11, tmp)
259         sort2(r06, r12, tmp)
260
261         // sort value pairs   (A12,A16), (A13,A17), (A06,A10), (A07,A11)
262         // sort register pairs  (r05,r07), (r06,r08), (r14,r03), (r00,r04)
263         sort2(r05, r07, tmp)
264         sort2(r06, r08, tmp)
265         sort2(r14, r03, tmp)
266         sort2(r00, r04, tmp)
267
268         // sort value pairs   (A10,A12), (A11,A13), (A11,A12)
269         // sort register pairs  (r03,r05), (r04,r06), (r04,r05)
270         sort2(r03, r05, tmp)
271         sort2(r04, r06, tmp)
272         sort2(r04, r05, tmp)
273
274
275         /* median is in r05 */
276         _mm256_store_si256((__m256i*)(imgDst + imgWidth * y_out * 3 + x_rgb_out), r05);
277       }
278     }
279     _mm_free(arr);
280   }
281 }
```

C. OpenCL implementáció kódja

C.1. defs.h

```
#define FILTER_W 5
#define FILTER_H 5

#define RUNS 1
#define KERNEL_RUNS 500

#define FIXED_OCL_DEVICE 0
#define FIXED_OCL_DEVICE_ID 0

//#define KERNEL_FILE_NAME ".\\opencl_kernels.cl"
#define KERNEL_FUNCTION "kernel_median_filter"
#define LOCAL_SIZE_X 16               // workgroup X size: 16 for all kernels
#define LOCAL_SIZE_Y 16               // workgroup X size: 16 for all kernels
```

C.2. func.h

```
double time_measure(int mode);

void median_filter_ocl(int imgHeight, int imgWidth, int imgHeightF, int imgWidthF,
  int imgFOfssetH, int imgFOfssetW,
  unsigned char *imgSrc, unsigned char *imgDst);
```

C.3. main.cpp

```
// lab1.cpp : Defines the entry point for the console application.
//

//#include "stdafx.h"
#include "memory.h"
#include "time.h"

#include "omp.h"

#include <IL/ilut.h>
#include <IL/ilu.h>

#include "emmintrin.h"
#include "nmmintrin.h"

#include "defs.h"
#include "func.h"


int main()
{
  ilInit(); iluInit();
  ILboolean ret;
  ILuint ilImg=0;
  ilGenImages(1, &ilImg);
  ilBindImage(ilImg);
    ret = ilLoadImage((const wchar_t*)("input.jpg"));
  ILubyte* imgData = ilGetData();

  int imgWidth = ilGetInteger(IL_IMAGE_WIDTH);
  int imgHeight = ilGetInteger(IL_IMAGE_HEIGHT);
  ILint imgOrigin = ilGetInteger(IL_ORIGIN_MODE);



  printf("Input resolution: %4dx%4d\n", imgWidth, imgHeight);

```

```
38    unsigned char *imgSrc;
39    int imgWidthF = imgWidth+FILTER_W-1;
40    int imgHeightF = imgHeight+FILTER_H-1;
41    int imgFOfssetW = (FILTER_W-1)/2;
42    int imgFOfssetH = (FILTER_H-1)/2;
43    imgSrc   = (unsigned char *)(_aligned_malloc(3*imgWidthF*imgHeightF*sizeof(unsigned char),
        32));
44      int row, col;
45
46    for (row=0; row<imgHeightF; row++)
47    {
48      for (col=0; col<imgWidthF;col++)
49      {
50        int pixel = (row*imgWidthF + col)*3;
51        *(imgSrc + pixel + 0) = 0;
52        *(imgSrc + pixel + 1) = 0;
53        *(imgSrc + pixel + 2) = 0;
54      }
55    }
56
57    for (row=0; row<imgHeight; row++)
58    {
59      for (col=0; col<imgWidth;col++)
60      {
61        int pixel_dst = ((row+imgFOfssetH)*imgWidthF + (col+imgFOfssetW))*3;
62        int pixel_src = (row*imgWidth + col)*3;
63        *(imgSrc + pixel_dst + 0) = (unsigned char)(*(imgData + pixel_src + 0));
64        *(imgSrc + pixel_dst + 1) = (unsigned char)(*(imgData + pixel_src + 1));
65        *(imgSrc + pixel_dst + 2) = (unsigned char)(*(imgData + pixel_src + 2));
66      }
67    }
68
69
70  // IMAGE PROCESSING
71  //-------------------------------------------------------------------------------------------
72    clock_t s0, e0;
73      double d0;
74
75    unsigned char *imgRes;
76      imgRes = (unsigned char *)(_aligned_malloc(3*imgWidth*imgHeight*sizeof(unsigned char), 32)
        );
77
78    double mpixel;
79
80
81  #if 1
82    median_filter_ocl(imgHeight, imgWidth, imgHeightF, imgWidthF,
83      imgFOfssetH, imgFOfssetW,
84      imgSrc, imgRes);
85  #endif
86
87  //-------------------------------------------------------------------------------------------
88  // IMAGE PROCESSING END
89
90    for (row=0; row<imgHeight; row++)
91    {
92      for (col=0; col<imgWidth;col++)
93      {
94        int pixel_src = (row*imgWidth + col) * 3;
95        int pixel_dst = (row*imgWidth + col) * 3;
96        *(imgData + pixel_dst + 0) = (ILubyte)(*(imgRes + pixel_src + 0));
97        *(imgData + pixel_dst + 1) = (ILubyte)(*(imgRes + pixel_src + 1));
98        *(imgData + pixel_dst + 2) = (ILubyte)(*(imgRes + pixel_src + 2));
99      }
100   }
101   ret = ilSetData(imgData);
102   ilEnable(IL_FILE_OVERWRITE);
103   ilSaveImage((const wchar_t*)("output.jpg"));
104   ilDeleteImages(1, &ilImg);
105
106
107   _aligned_free(imgSrc);
108     _aligned_free(imgRes);
```

```
109
110   return 0;
111 }
```

C.4. opencl_kernels.cl

```
1  //
       ----------------------------------------------------------------------------------------------------------------
2  __kernel void kernel_median_filter(__global unsigned char* gInput,
3                         __global unsigned char* gOutput,
4                         int imgWidth,
5                         int imgWidthF)
6  {
7    // calculate index in global memory for copying (1 byte)
8    int BI = (get_local_size(1)*get_group_id(1)*imgWidthF + get_local_size(0)*get_group_id(0)) *
         3; // global base index
9    int L1DID = get_local_id(1)*get_local_size(0) + get_local_id(0); // local 1D index
10
11   // calculate index in local memory for copying (1 byte)
12   int CYOIP = L1DID/(20*3); // copy y offset in pixels from global base address
13   int CXOIP = (L1DID%(20*3))/3; // copy x offset in pixels from global base address
14   int CCO = L1DID%3; // copy channel offset
15   int rowstep = (get_local_size(0)*get_local_size(1))/(20*3); // next component to copy is
       this many rows down
16
17   // declare local memory, copy global -> shared (local) memory
18   __local half shmem[20*4+2][20];
19   if(L1DID<20*3*rowstep)
20   {
21     for(int row=0; row<get_local_size(1)+4; row+=rowstep)
22     {
23       shmem[CXOIP*4+CCO][CYOIP+row] = (half)(gInput[BI + (CYOIP*imgWidthF + CXOIP + row*
       imgWidthF)*3 + CCO]);
24     }
25   }
26
27   // wait for other threads to finish copy
28   barrier(CLK_LOCAL_MEM_FENCE);
29
30   // choose median for the 3 channels of the given pixel
31   half tmp;
32   half r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14, r15, r16,
       r17, r18, r19, r20, r21, r22, r23, r24;
33
34   // first result (byte) global index
35   BI = ((get_global_id(1))*imgWidth + get_global_id(0)) * 3;
36
37   for(int channel=0; channel<3; channel++)
38   {
39     // load the appropriate 25 values to be sorted
40     // for(int i=0; x<25; i++)
41     // {
42       // for(int y=0; y<5; y++)
43       // {
44         // values[x+y*5] = shmem[get_local_id(0)+x][get_local_id(1)+y][channel];
45       // }
46     // }
47     r00=shmem[(get_local_id(0)+0)*4+channel][get_local_id(1)+0];
48     r01=shmem[(get_local_id(0)+1)*4+channel][get_local_id(1)+0];
49     r02=shmem[(get_local_id(0)+2)*4+channel][get_local_id(1)+0];
50     r03=shmem[(get_local_id(0)+3)*4+channel][get_local_id(1)+0];
51     r04=shmem[(get_local_id(0)+4)*4+channel][get_local_id(1)+0];
52     r05=shmem[(get_local_id(0)+0)*4+channel][get_local_id(1)+1];
53     r06=shmem[(get_local_id(0)+1)*4+channel][get_local_id(1)+1];
54     r07=shmem[(get_local_id(0)+2)*4+channel][get_local_id(1)+1];
55     r08=shmem[(get_local_id(0)+3)*4+channel][get_local_id(1)+1];
56     r09=shmem[(get_local_id(0)+4)*4+channel][get_local_id(1)+1];
57     r10=shmem[(get_local_id(0)+0)*4+channel][get_local_id(1)+2];
58     r11=shmem[(get_local_id(0)+1)*4+channel][get_local_id(1)+2];
```

```
    r12=shmem[(get_local_id(0)+2)*4+channel][get_local_id(1)+2];
    r13=shmem[(get_local_id(0)+3)*4+channel][get_local_id(1)+2];
    r14=shmem[(get_local_id(0)+4)*4+channel][get_local_id(1)+2];
    r15=shmem[(get_local_id(0)+0)*4+channel][get_local_id(1)+3];
    r16=shmem[(get_local_id(0)+1)*4+channel][get_local_id(1)+3];
    r17=shmem[(get_local_id(0)+2)*4+channel][get_local_id(1)+3];
    r18=shmem[(get_local_id(0)+3)*4+channel][get_local_id(1)+3];
    r19=shmem[(get_local_id(0)+4)*4+channel][get_local_id(1)+3];
    r20=shmem[(get_local_id(0)+0)*4+channel][get_local_id(1)+4];
    r21=shmem[(get_local_id(0)+1)*4+channel][get_local_id(1)+4];
    r22=shmem[(get_local_id(0)+2)*4+channel][get_local_id(1)+4];
    r23=shmem[(get_local_id(0)+3)*4+channel][get_local_id(1)+4];
    r24=shmem[(get_local_id(0)+4)*4+channel][get_local_id(1)+4];

    // find the median, will be in r12
    tmp=fmax(r00,r01); r00=fmin(r00,r01); r01=tmp;
    tmp=fmax(r02,r03); r02=fmin(r02,r03); r03=tmp;
    tmp=fmax(r04,r05); r04=fmin(r04,r05); r05=tmp;
    tmp=fmax(r06,r07); r06=fmin(r06,r07); r07=tmp;
    tmp=fmax(r08,r09); r08=fmin(r08,r09); r09=tmp;
    tmp=fmax(r10,r11); r10=fmin(r10,r11); r11=tmp;
    tmp=fmax(r12,r13); r12=fmin(r12,r13); r13=tmp;
    tmp=fmax(r14,r15); r14=fmin(r14,r15); r15=tmp;
    tmp=fmax(r16,r17); r16=fmin(r16,r17); r17=tmp;
    tmp=fmax(r18,r19); r18=fmin(r18,r19); r19=tmp;
    tmp=fmax(r20,r21); r20=fmin(r20,r21); r21=tmp;
    tmp=fmax(r22,r23); r22=fmin(r22,r23); r23=tmp;
    tmp=fmax(r00,r02); r00=fmin(r00,r02); r02=tmp;
    tmp=fmax(r01,r03); r01=fmin(r01,r03); r03=tmp;
    tmp=fmax(r04,r06); r04=fmin(r04,r06); r06=tmp;
    tmp=fmax(r05,r07); r05=fmin(r05,r07); r07=tmp;
    tmp=fmax(r08,r10); r08=fmin(r08,r10); r10=tmp;
    tmp=fmax(r09,r11); r09=fmin(r09,r11); r11=tmp;
    tmp=fmax(r12,r14); r12=fmin(r12,r14); r14=tmp;
    tmp=fmax(r13,r15); r13=fmin(r13,r15); r15=tmp;
    tmp=fmax(r16,r18); r16=fmin(r16,r18); r18=tmp;
    tmp=fmax(r17,r19); r17=fmin(r17,r19); r19=tmp;
    tmp=fmax(r20,r22); r20=fmin(r20,r22); r22=tmp;
    tmp=fmax(r21,r23); r21=fmin(r21,r23); r23=tmp;
    tmp=fmax(r01,r02); r01=fmin(r01,r02); r02=tmp;
    tmp=fmax(r05,r06); r05=fmin(r05,r06); r06=tmp;
    tmp=fmax(r09,r10); r09=fmin(r09,r10); r10=tmp;
    tmp=fmax(r13,r14); r13=fmin(r13,r14); r14=tmp;
    tmp=fmax(r17,r18); r17=fmin(r17,r18); r18=tmp;
    tmp=fmax(r21,r22); r21=fmin(r21,r22); r22=tmp;
    tmp=fmax(r00,r04); r00=fmin(r00,r04); r04=tmp;
    tmp=fmax(r01,r05); r01=fmin(r01,r05); r05=tmp;
    tmp=fmax(r02,r06); r02=fmin(r02,r06); r06=tmp;
    tmp=fmax(r03,r07); r03=fmin(r03,r07); r07=tmp;
    tmp=fmax(r08,r12); r08=fmin(r08,r12); r12=tmp;
    tmp=fmax(r09,r13); r09=fmin(r09,r13); r13=tmp;
    tmp=fmax(r10,r14); r10=fmin(r10,r14); r14=tmp;
    tmp=fmax(r11,r15); r11=fmin(r11,r15); r15=tmp;
    tmp=fmax(r16,r20); r16=fmin(r16,r20); r20=tmp;
    tmp=fmax(r17,r21); r17=fmin(r17,r21); r21=tmp;
    tmp=fmax(r18,r22); r18=fmin(r18,r22); r22=tmp;
    tmp=fmax(r19,r23); r19=fmin(r19,r23); r23=tmp;
    tmp=fmax(r02,r04); r02=fmin(r02,r04); r04=tmp;
    tmp=fmax(r03,r05); r03=fmin(r03,r05); r05=tmp;
    tmp=fmax(r10,r12); r10=fmin(r10,r12); r12=tmp;
    tmp=fmax(r11,r13); r11=fmin(r11,r13); r13=tmp;
    tmp=fmax(r18,r20); r18=fmin(r18,r20); r20=tmp;
    tmp=fmax(r19,r21); r19=fmin(r19,r21); r21=tmp;
    tmp=fmax(r01,r02); r01=fmin(r01,r02); r02=tmp;
    tmp=fmax(r03,r04); r03=fmin(r03,r04); r04=tmp;
    tmp=fmax(r05,r06); r05=fmin(r05,r06); r06=tmp;
    tmp=fmax(r09,r10); r09=fmin(r09,r10); r10=tmp;
    tmp=fmax(r11,r12); r11=fmin(r11,r12); r12=tmp;
    tmp=fmax(r13,r14); r13=fmin(r13,r14); r14=tmp;
    tmp=fmax(r17,r18); r17=fmin(r17,r18); r18=tmp;
    tmp=fmax(r19,r20); r19=fmin(r19,r20); r20=tmp;
    tmp=fmax(r21,r22); r21=fmin(r21,r22); r22=tmp;
    tmp=fmax(r00,r08); r00=fmin(r00,r08); r08=tmp;
```

```
    tmp=fmax(r01,r09); r01=fmin(r01,r09); r09=tmp;
    tmp=fmax(r02,r10); r02=fmin(r02,r10); r10=tmp;
    tmp=fmax(r03,r11); r03=fmin(r03,r11); r11=tmp;
    tmp=fmax(r04,r12); r04=fmin(r04,r12); r12=tmp;
    tmp=fmax(r05,r13); r05=fmin(r05,r13); r13=tmp;
    tmp=fmax(r06,r14); r06=fmin(r06,r14); r14=tmp;
    tmp=fmax(r07,r15); r07=fmin(r07,r15); r15=tmp;
    tmp=fmax(r16,r24); r16=fmin(r16,r24); r24=tmp;
    tmp=fmax(r04,r08); r04=fmin(r04,r08); r08=tmp;
    tmp=fmax(r05,r09); r05=fmin(r05,r09); r09=tmp;
    tmp=fmax(r06,r10); r06=fmin(r06,r10); r10=tmp;
    tmp=fmax(r07,r11); r07=fmin(r07,r11); r11=tmp;
    tmp=fmax(r20,r24); r20=fmin(r20,r24); r24=tmp;
    tmp=fmax(r02,r04); r02=fmin(r02,r04); r04=tmp;
    tmp=fmax(r03,r05); r03=fmin(r03,r05); r05=tmp;
    tmp=fmax(r06,r08); r06=fmin(r06,r08); r08=tmp;
    tmp=fmax(r07,r09); r07=fmin(r07,r09); r09=tmp;
    tmp=fmax(r10,r12); r10=fmin(r10,r12); r12=tmp;
    tmp=fmax(r11,r13); r11=fmin(r11,r13); r13=tmp;
    tmp=fmax(r18,r20); r18=fmin(r18,r20); r20=tmp;
    tmp=fmax(r19,r21); r19=fmin(r19,r21); r21=tmp;
    tmp=fmax(r22,r24); r22=fmin(r22,r24); r24=tmp;
    tmp=fmax(r01,r02); r01=fmin(r01,r02); r02=tmp;
    tmp=fmax(r03,r04); r03=fmin(r03,r04); r04=tmp;
    tmp=fmax(r05,r06); r05=fmin(r05,r06); r06=tmp;
    tmp=fmax(r07,r08); r07=fmin(r07,r08); r08=tmp;
    tmp=fmax(r09,r10); r09=fmin(r09,r10); r10=tmp;
    tmp=fmax(r11,r12); r11=fmin(r11,r12); r12=tmp;
    tmp=fmax(r13,r14); r13=fmin(r13,r14); r14=tmp;
    tmp=fmax(r17,r18); r17=fmin(r17,r18); r18=tmp;
    tmp=fmax(r19,r20); r19=fmin(r19,r20); r20=tmp;
    tmp=fmax(r21,r22); r21=fmin(r21,r22); r22=tmp;
    tmp=fmax(r23,r24); r23=fmin(r23,r24); r24=tmp;
    tmp=fmax(r00,r16); r00=fmin(r00,r16); r16=tmp;
    tmp=fmax(r01,r17); r01=fmin(r01,r17); r17=tmp;
    tmp=fmax(r02,r18); r02=fmin(r02,r18); r18=tmp;
    tmp=fmax(r03,r19); r03=fmin(r03,r19); r19=tmp;
    tmp=fmax(r04,r20); r04=fmin(r04,r20); r20=tmp;
    tmp=fmax(r05,r21); r05=fmin(r05,r21); r21=tmp;
    tmp=fmax(r06,r22); r06=fmin(r06,r22); r22=tmp;
    tmp=fmax(r07,r23); r07=fmin(r07,r23); r23=tmp;
    tmp=fmax(r08,r24); r08=fmin(r08,r24); r24=tmp;
    tmp=fmax(r08,r16); r08=fmin(r08,r16); r16=tmp;
    tmp=fmax(r09,r17); r09=fmin(r09,r17); r17=tmp;
    tmp=fmax(r10,r18); r10=fmin(r10,r18); r18=tmp;
    tmp=fmax(r11,r19); r11=fmin(r11,r19); r19=tmp;
    tmp=fmax(r12,r20); r12=fmin(r12,r20); r20=tmp;
    tmp=fmax(r13,r21); r13=fmin(r13,r21); r21=tmp;
    // tmp=fmax(r14,r22); r14=fmin(r14,r22); r22=tmp;
    // tmp=fmax(r15,r23); r15=fmin(r15,r23); r23=tmp;
    // tmp=fmax(r04,r08); r04=fmin(r04,r08); r08=tmp;
    // tmp=fmax(r05,r09); r05=fmin(r05,r09); r09=tmp;
    tmp=fmax(r06,r10); r06=fmin(r06,r10); r10=tmp;
    tmp=fmax(r07,r11); r07=fmin(r07,r11); r11=tmp;
    tmp=fmax(r12,r16); r12=fmin(r12,r16); r16=tmp;
    tmp=fmax(r13,r17); r13=fmin(r13,r17); r17=tmp;
    // tmp=fmax(r14,r18); r14=fmin(r14,r18); r18=tmp;
    // tmp=fmax(r15,r19); r15=fmin(r15,r19); r19=tmp;
    // tmp=fmax(r20,r24); r20=fmin(r20,r24); r24=tmp;
    // tmp=fmax(r02,r04); r02=fmin(r02,r04); r04=tmp;
    // tmp=fmax(r03,r05); r03=fmin(r03,r05); r05=tmp;
    // tmp=fmax(r06,r08); r06=fmin(r06,r08); r08=tmp;
    // tmp=fmax(r07,r09); r07=fmin(r07,r09); r09=tmp;
    tmp=fmax(r10,r12); r10=fmin(r10,r12); r12=tmp;
    tmp=fmax(r11,r13); r11=fmin(r11,r13); r13=tmp;
    // tmp=fmax(r14,r16); r14=fmin(r14,r16); r16=tmp;
    // tmp=fmax(r15,r17); r15=fmin(r15,r17); r17=tmp;
    // tmp=fmax(r18,r20); r18=fmin(r18,r20); r20=tmp;
    // tmp=fmax(r19,r21); r19=fmin(r19,r21); r21=tmp;
    // tmp=fmax(r22,r24); r22=fmin(r22,r24); r24=tmp;
    // tmp=fmax(r01,r02); r01=fmin(r01,r02); r02=tmp;
    // tmp=fmax(r03,r04); r03=fmin(r03,r04); r04=tmp;
    // tmp=fmax(r05,r06); r05=fmin(r05,r06); r06=tmp;
```

```
205     // tmp=fmax(r07,r08); r07=fmin(r07,r08); r08=tmp;
206     // tmp=fmax(r09,r10); r09=fmin(r09,r10); r10=tmp;
207     tmp=fmax(r11,r12); r11=fmin(r11,r12); r12=tmp;
208     // tmp=fmax(r13,r14); r13=fmin(r13,r14); r14=tmp;
209     // tmp=fmax(r15,r16); r15=fmin(r15,r16); r16=tmp;
210     // tmp=fmax(r17,r18); r17=fmin(r17,r18); r18=tmp;
211     // tmp=fmax(r19,r20); r19=fmin(r19,r20); r20=tmp;
212     // tmp=fmax(r21,r22); r21=fmin(r21,r22); r22=tmp;
213     // tmp=fmax(r23,r24); r23=fmin(r23,r24); r24=tmp;
214
215     // copy medians to global memory
216     gOutput[BI+channel] = (unsigned char)(r12);
217   }
218 }
```

C.5. median_filter_ocl.cpp

(Ebben a fájlban lévő kernel string literál effektíve ugyanaz, mint a fönti .cl kernel kód.)

```
1  // OCLTest1.cpp : Defines the entry point for the console application.
2  //
3
4  #define CL_TARGET_OPENCL_VERSION 120
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <cstring>
9
10 #include "time.h"
11
12 #include "CL\cl.h"
13
14 #include "defs.h"
15 #include "func.h"
16
17 ////////////////////////////////////////////////////
18 // SEE defs.h for kernel selection!!!
19
20
21 const char *getErrorString(cl_int error)
22 {
23   switch (error){
24     // run-time and JIT compiler errors
25   case 0: return "CL_SUCCESS";
26   case -1: return "CL_DEVICE_NOT_FOUND";
27   case -2: return "CL_DEVICE_NOT_AVAILABLE";
28   case -3: return "CL_COMPILER_NOT_AVAILABLE";
29   case -4: return "CL_MEM_OBJECT_ALLOCATION_FAILURE";
30   case -5: return "CL_OUT_OF_RESOURCES";
31   case -6: return "CL_OUT_OF_HOST_MEMORY";
32   case -7: return "CL_PROFILING_INFO_NOT_AVAILABLE";
33   case -8: return "CL_MEM_COPY_OVERLAP";
34   case -9: return "CL_IMAGE_FORMAT_MISMATCH";
35   case -10: return "CL_IMAGE_FORMAT_NOT_SUPPORTED";
36   case -11: return "CL_BUILD_PROGRAM_FAILURE";
37   case -12: return "CL_MAP_FAILURE";
38   case -13: return "CL_MISALIGNED_SUB_BUFFER_OFFSET";
39   case -14: return "CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST";
40   case -15: return "CL_COMPILE_PROGRAM_FAILURE";
41   case -16: return "CL_LINKER_NOT_AVAILABLE";
42   case -17: return "CL_LINK_PROGRAM_FAILURE";
43   case -18: return "CL_DEVICE_PARTITION_FAILED";
44   case -19: return "CL_KERNEL_ARG_INFO_NOT_AVAILABLE";
45
46     // compile-time errors
47   case -30: return "CL_INVALID_VALUE";
48   case -31: return "CL_INVALID_DEVICE_TYPE";
49   case -32: return "CL_INVALID_PLATFORM";
50   case -33: return "CL_INVALID_DEVICE";
51   case -34: return "CL_INVALID_CONTEXT";
52   case -35: return "CL_INVALID_QUEUE_PROPERTIES";
```

```
53    case -36: return "CL_INVALID_COMMAND_QUEUE";
54    case -37: return "CL_INVALID_HOST_PTR";
55    case -38: return "CL_INVALID_MEM_OBJECT";
56    case -39: return "CL_INVALID_IMAGE_FORMAT_DESCRIPTOR";
57    case -40: return "CL_INVALID_IMAGE_SIZE";
58    case -41: return "CL_INVALID_SAMPLER";
59    case -42: return "CL_INVALID_BINARY";
60    case -43: return "CL_INVALID_BUILD_OPTIONS";
61    case -44: return "CL_INVALID_PROGRAM";
62    case -45: return "CL_INVALID_PROGRAM_EXECUTABLE";
63    case -46: return "CL_INVALID_KERNEL_NAME";
64    case -47: return "CL_INVALID_KERNEL_DEFINITION";
65    case -48: return "CL_INVALID_KERNEL";
66    case -49: return "CL_INVALID_ARG_INDEX";
67    case -50: return "CL_INVALID_ARG_VALUE";
68    case -51: return "CL_INVALID_ARG_SIZE";
69    case -52: return "CL_INVALID_KERNEL_ARGS";
70    case -53: return "CL_INVALID_WORK_DIMENSION";
71    case -54: return "CL_INVALID_WORK_GROUP_SIZE";
72    case -55: return "CL_INVALID_WORK_ITEM_SIZE";
73    case -56: return "CL_INVALID_GLOBAL_OFFSET";
74    case -57: return "CL_INVALID_EVENT_WAIT_LIST";
75    case -58: return "CL_INVALID_EVENT";
76    case -59: return "CL_INVALID_OPERATION";
77    case -60: return "CL_INVALID_GL_OBJECT";
78    case -61: return "CL_INVALID_BUFFER_SIZE";
79    case -62: return "CL_INVALID_MIP_LEVEL";
80    case -63: return "CL_INVALID_GLOBAL_WORK_SIZE";
81    case -64: return "CL_INVALID_PROPERTY";
82    case -65: return "CL_INVALID_IMAGE_DESCRIPTOR";
83    case -66: return "CL_INVALID_COMPILER_OPTIONS";
84    case -67: return "CL_INVALID_LINKER_OPTIONS";
85    case -68: return "CL_INVALID_DEVICE_PARTITION_COUNT";
86
87      // extension errors
88    case -1000: return "CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR";
89    case -1001: return "CL_PLATFORM_NOT_FOUND_KHR";
90    case -1002: return "CL_INVALID_D3D10_DEVICE_KHR";
91    case -1003: return "CL_INVALID_D3D10_RESOURCE_KHR";
92    case -1004: return "CL_D3D10_RESOURCE_ALREADY_ACQUIRED_KHR";
93    case -1005: return "CL_D3D10_RESOURCE_NOT_ACQUIRED_KHR";
94    default: return "Unknown OpenCL error";
95    }
96  }
97
98  #define MAX_PROG_SIZE 65536
99
100 void median_filter_ocl(int imgHeight, int imgWidth, int imgHeightF, int imgWidthF,
101   int imgFOfssetH, int imgFOfssetW,
102   unsigned char *imgSrc, unsigned char *imgDst)
103 {
104   cl_device_id device_id = NULL;
105   cl_context context = NULL;
106   cl_command_queue command_queue = NULL;
107   cl_program program = NULL;
108   cl_kernel kernel = NULL;
109   cl_platform_id platform_id = NULL;
110   cl_uint ret_num_devices;
111   cl_uint ret_num_platforms;
112   cl_int ret;
113
114   clock_t s0, e0;
115   double d0;
116
117   int size_in;
118   size_in = imgHeightF*imgWidthF * 3;
119   int size_out;
120   size_out = imgHeight*imgWidth * 3;
121
122   //
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
123   // Init OpenCL
```

```
124
125   /* Get Platform and Device Info */
126   ret = clGetPlatformIDs(0, NULL, &ret_num_platforms);
127   cl_platform_id *platforms;
128   platforms = (cl_platform_id*)malloc(sizeof(cl_platform_id)* ret_num_platforms);
129   ret = clGetPlatformIDs(ret_num_platforms, platforms, &ret_num_platforms);
130
131   int num_devices_all = 0;
132   for (int platform_id = 0; platform_id < ret_num_platforms; platform_id++)
133   {
134     ret = clGetDeviceIDs(platforms[platform_id], CL_DEVICE_TYPE_ALL, 0, NULL, &ret_num_devices
      );
135     num_devices_all = num_devices_all + ret_num_devices;
136   }
137   cl_device_id *devices;
138   int device_offset = 0;
139   devices = (cl_device_id*)malloc(sizeof(cl_device_id)* num_devices_all);
140   for (int platform_id = 0; platform_id < ret_num_platforms; platform_id++)
141   {
142     ret = clGetDeviceIDs(platforms[platform_id], CL_DEVICE_TYPE_ALL, 0, NULL, &ret_num_devices
      );
143     ret = clGetDeviceIDs(platforms[platform_id], CL_DEVICE_TYPE_ALL, ret_num_devices, &devices
      [device_offset], &ret_num_devices);
144     device_offset = device_offset + ret_num_devices;
145   }
146
147   char cBuffer[1024];
148   for (int device_num = 0; device_num < num_devices_all; device_num++)
149   {
150     printf("Device id: %d,  ", device_num);
151
152     ret = clGetDeviceInfo(devices[device_num], CL_DEVICE_VENDOR, sizeof(cBuffer), &cBuffer,
      NULL);
153     printf("%s ", cBuffer);
154
155     ret = clGetDeviceInfo(devices[device_num], CL_DEVICE_NAME, sizeof(cBuffer), &cBuffer, NULL
      );
156     printf("%s\r\n", cBuffer);
157   }
158
159   //
      ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
160   // Select device to be used
161 #if FIXED_OCL_DEVICE == 0
162   printf("\n\nSelect OpenCL device and press enter:");
163   int device_sel = getchar()-0x30;
164   device_id = devices[device_sel];
165 #else
166   device_id = devices[FIXED_OCL_DEVICE_ID];
167   free(devices);
168 #endif
169
170   // Load the source code containing the kernel
171   const char *kernel_source="__kernel void kernel_median_filter(__global unsigned char* gInput
      ,\n\
172                       __global unsigned char* gOutput,\n\
173                       int imgWidth,\n\
174                       int imgWidthF)\n\
175 {\n\
176   // calculate index in global memory for copying (1 byte)\n\
177   int BI = (get_local_size(1)*get_group_id(1)*imgWidthF + get_local_size(0)*get_group_id(0)) *
       3; // global base index\n\
178   int L1DID = get_local_id(1)*get_local_size(0) + get_local_id(0); // local 1D index\n\
179   \n\
180   // calculate index in local memory for copying (1 byte)\n\
181   int CYOIP = L1DID/(20*3); // copy y offset in pixels from global base address\n\
182   int CXOIP = (L1DID%(20*3))/3; // copy x offset in pixels from global base address\n\
183   int CCO = L1DID%3; // copy channel offset\n\
184   int rowstep = (get_local_size(0)*get_local_size(1))/(20*3); // next component to copy is
      this many rows down\n\
185   \n\
186   // declare local memory, copy global -> shared (local) memory\n\
```

```
  // shared memory padded with one dummy channel per pixel plus two dummy channels (1 bank) per row at the ends\n\
  __local half shmem[20*4+2][20];\n\
  if(L1DID<20*3*rowstep)\n\
  {\n\
    for(int row=0; row<get_local_size(1)+4; row+=rowstep)\n\
    {\n\
      shmem[CXOIP*4+CCO][CYOIP+row] = (half)(gInput[BI + (CYOIP*imgWidthF + CXOIP + row*imgWidthF)*3 + CCO]);\n\
    }\n\
  }\n\
\n\
  // wait for other threads to finish copy\n\
  barrier(CLK_LOCAL_MEM_FENCE);\n\
\n\
  // choose median for the 3 channels of the given pixel\n\
  half tmp;\n\
  half r00, r01, r02, r03, r04, r05, r06, r07, r08, r09, r10, r11, r12, r13, r14, r15, r16, r17, r18, r19, r20, r21, r22, r23, r24;\n\
  \n\
  // first result (byte) global index\n\
  BI = ((get_global_id(1))*imgWidth + get_global_id(0)) * 3;\n\
\n\
  for(int channel=0; channel<3; channel++)\n\
  {\n\
    // load the appropriate 25 values to be sorted\n\
    r00=shmem[(get_local_id(0)+0)*4+channel][get_local_id(1)+0];\n\
    r01=shmem[(get_local_id(0)+1)*4+channel][get_local_id(1)+0];\n\
    r02=shmem[(get_local_id(0)+2)*4+channel][get_local_id(1)+0];\n\
    r03=shmem[(get_local_id(0)+3)*4+channel][get_local_id(1)+0];\n\
    r04=shmem[(get_local_id(0)+4)*4+channel][get_local_id(1)+0];\n\
    r05=shmem[(get_local_id(0)+0)*4+channel][get_local_id(1)+1];\n\
    r06=shmem[(get_local_id(0)+1)*4+channel][get_local_id(1)+1];\n\
    r07=shmem[(get_local_id(0)+2)*4+channel][get_local_id(1)+1];\n\
    r08=shmem[(get_local_id(0)+3)*4+channel][get_local_id(1)+1];\n\
    r09=shmem[(get_local_id(0)+4)*4+channel][get_local_id(1)+1];\n\
    r10=shmem[(get_local_id(0)+0)*4+channel][get_local_id(1)+2];\n\
    r11=shmem[(get_local_id(0)+1)*4+channel][get_local_id(1)+2];\n\
    r12=shmem[(get_local_id(0)+2)*4+channel][get_local_id(1)+2];\n\
    r13=shmem[(get_local_id(0)+3)*4+channel][get_local_id(1)+2];\n\
    r14=shmem[(get_local_id(0)+4)*4+channel][get_local_id(1)+2];\n\
    r15=shmem[(get_local_id(0)+0)*4+channel][get_local_id(1)+3];\n\
    r16=shmem[(get_local_id(0)+1)*4+channel][get_local_id(1)+3];\n\
    r17=shmem[(get_local_id(0)+2)*4+channel][get_local_id(1)+3];\n\
    r18=shmem[(get_local_id(0)+3)*4+channel][get_local_id(1)+3];\n\
    r19=shmem[(get_local_id(0)+4)*4+channel][get_local_id(1)+3];\n\
    r20=shmem[(get_local_id(0)+0)*4+channel][get_local_id(1)+4];\n\
    r21=shmem[(get_local_id(0)+1)*4+channel][get_local_id(1)+4];\n\
    r22=shmem[(get_local_id(0)+2)*4+channel][get_local_id(1)+4];\n\
    r23=shmem[(get_local_id(0)+3)*4+channel][get_local_id(1)+4];\n\
    r24=shmem[(get_local_id(0)+4)*4+channel][get_local_id(1)+4];\n\
\n\
    // find the median, will be in r12\n\
    tmp=fmax(r00,r01); r00=fmin(r00,r01); r01=tmp;\n\
    tmp=fmax(r02,r03); r02=fmin(r02,r03); r03=tmp;\n\
    tmp=fmax(r04,r05); r04=fmin(r04,r05); r05=tmp;\n\
    tmp=fmax(r06,r07); r06=fmin(r06,r07); r07=tmp;\n\
    tmp=fmax(r08,r09); r08=fmin(r08,r09); r09=tmp;\n\
    tmp=fmax(r10,r11); r10=fmin(r10,r11); r11=tmp;\n\
    tmp=fmax(r12,r13); r12=fmin(r12,r13); r13=tmp;\n\
    tmp=fmax(r14,r15); r14=fmin(r14,r15); r15=tmp;\n\
    tmp=fmax(r16,r17); r16=fmin(r16,r17); r17=tmp;\n\
    tmp=fmax(r18,r19); r18=fmin(r18,r19); r19=tmp;\n\
    tmp=fmax(r20,r21); r20=fmin(r20,r21); r21=tmp;\n\
    tmp=fmax(r22,r23); r22=fmin(r22,r23); r23=tmp;\n\
    tmp=fmax(r00,r02); r00=fmin(r00,r02); r02=tmp;\n\
    tmp=fmax(r01,r03); r01=fmin(r01,r03); r03=tmp;\n\
    tmp=fmax(r04,r06); r04=fmin(r04,r06); r06=tmp;\n\
    tmp=fmax(r05,r07); r05=fmin(r05,r07); r07=tmp;\n\
    tmp=fmax(r08,r10); r08=fmin(r08,r10); r10=tmp;\n\
    tmp=fmax(r09,r11); r09=fmin(r09,r11); r11=tmp;\n\
    tmp=fmax(r12,r14); r12=fmin(r12,r14); r14=tmp;\n\
    tmp=fmax(r13,r15); r13=fmin(r13,r15); r15=tmp;\n\
```

```
257     tmp=fmax(r16,r18); r16=fmin(r16,r18); r18=tmp;\n\
258     tmp=fmax(r17,r19); r17=fmin(r17,r19); r19=tmp;\n\
259     tmp=fmax(r20,r22); r20=fmin(r20,r22); r22=tmp;\n\
260     tmp=fmax(r21,r23); r21=fmin(r21,r23); r23=tmp;\n\
261     tmp=fmax(r01,r02); r01=fmin(r01,r02); r02=tmp;\n\
262     tmp=fmax(r05,r06); r05=fmin(r05,r06); r06=tmp;\n\
263     tmp=fmax(r09,r10); r09=fmin(r09,r10); r10=tmp;\n\
264     tmp=fmax(r13,r14); r13=fmin(r13,r14); r14=tmp;\n\
265     tmp=fmax(r17,r18); r17=fmin(r17,r18); r18=tmp;\n\
266     tmp=fmax(r21,r22); r21=fmin(r21,r22); r22=tmp;\n\
267     tmp=fmax(r00,r04); r00=fmin(r00,r04); r04=tmp;\n\
268     tmp=fmax(r01,r05); r01=fmin(r01,r05); r05=tmp;\n\
269     tmp=fmax(r02,r06); r02=fmin(r02,r06); r06=tmp;\n\
270     tmp=fmax(r03,r07); r03=fmin(r03,r07); r07=tmp;\n\
271     tmp=fmax(r08,r12); r08=fmin(r08,r12); r12=tmp;\n\
272     tmp=fmax(r09,r13); r09=fmin(r09,r13); r13=tmp;\n\
273     tmp=fmax(r10,r14); r10=fmin(r10,r14); r14=tmp;\n\
274     tmp=fmax(r11,r15); r11=fmin(r11,r15); r15=tmp;\n\
275     tmp=fmax(r16,r20); r16=fmin(r16,r20); r20=tmp;\n\
276     tmp=fmax(r17,r21); r17=fmin(r17,r21); r21=tmp;\n\
277     tmp=fmax(r18,r22); r18=fmin(r18,r22); r22=tmp;\n\
278     tmp=fmax(r19,r23); r19=fmin(r19,r23); r23=tmp;\n\
279     tmp=fmax(r02,r04); r02=fmin(r02,r04); r04=tmp;\n\
280     tmp=fmax(r03,r05); r03=fmin(r03,r05); r05=tmp;\n\
281     tmp=fmax(r10,r12); r10=fmin(r10,r12); r12=tmp;\n\
282     tmp=fmax(r11,r13); r11=fmin(r11,r13); r13=tmp;\n\
283     tmp=fmax(r18,r20); r18=fmin(r18,r20); r20=tmp;\n\
284     tmp=fmax(r19,r21); r19=fmin(r19,r21); r21=tmp;\n\
285     tmp=fmax(r01,r02); r01=fmin(r01,r02); r02=tmp;\n\
286     tmp=fmax(r03,r04); r03=fmin(r03,r04); r04=tmp;\n\
287     tmp=fmax(r05,r06); r05=fmin(r05,r06); r06=tmp;\n\
288     tmp=fmax(r09,r10); r09=fmin(r09,r10); r10=tmp;\n\
289     tmp=fmax(r11,r12); r11=fmin(r11,r12); r12=tmp;\n\
290     tmp=fmax(r13,r14); r13=fmin(r13,r14); r14=tmp;\n\
291     tmp=fmax(r17,r18); r17=fmin(r17,r18); r18=tmp;\n\
292     tmp=fmax(r19,r20); r19=fmin(r19,r20); r20=tmp;\n\
293     tmp=fmax(r21,r22); r21=fmin(r21,r22); r22=tmp;\n\
294     tmp=fmax(r00,r08); r00=fmin(r00,r08); r08=tmp;\n\
295     tmp=fmax(r01,r09); r01=fmin(r01,r09); r09=tmp;\n\
296     tmp=fmax(r02,r10); r02=fmin(r02,r10); r10=tmp;\n\
297     tmp=fmax(r03,r11); r03=fmin(r03,r11); r11=tmp;\n\
298     tmp=fmax(r04,r12); r04=fmin(r04,r12); r12=tmp;\n\
299     tmp=fmax(r05,r13); r05=fmin(r05,r13); r13=tmp;\n\
300     tmp=fmax(r06,r14); r06=fmin(r06,r14); r14=tmp;\n\
301     tmp=fmax(r07,r15); r07=fmin(r07,r15); r15=tmp;\n\
302     tmp=fmax(r16,r24); r16=fmin(r16,r24); r24=tmp;\n\
303     tmp=fmax(r04,r08); r04=fmin(r04,r08); r08=tmp;\n\
304     tmp=fmax(r05,r09); r05=fmin(r05,r09); r09=tmp;\n\
305     tmp=fmax(r06,r10); r06=fmin(r06,r10); r10=tmp;\n\
306     tmp=fmax(r07,r11); r07=fmin(r07,r11); r11=tmp;\n\
307     tmp=fmax(r20,r24); r20=fmin(r20,r24); r24=tmp;\n\
308     tmp=fmax(r02,r04); r02=fmin(r02,r04); r04=tmp;\n\
309     tmp=fmax(r03,r05); r03=fmin(r03,r05); r05=tmp;\n\
310     tmp=fmax(r06,r08); r06=fmin(r06,r08); r08=tmp;\n\
311     tmp=fmax(r07,r09); r07=fmin(r07,r09); r09=tmp;\n\
312     tmp=fmax(r10,r12); r10=fmin(r10,r12); r12=tmp;\n\
313     tmp=fmax(r11,r13); r11=fmin(r11,r13); r13=tmp;\n\
314     tmp=fmax(r18,r20); r18=fmin(r18,r20); r20=tmp;\n\
315     tmp=fmax(r19,r21); r19=fmin(r19,r21); r21=tmp;\n\
316     tmp=fmax(r22,r24); r22=fmin(r22,r24); r24=tmp;\n\
317     tmp=fmax(r01,r02); r01=fmin(r01,r02); r02=tmp;\n\
318     tmp=fmax(r03,r04); r03=fmin(r03,r04); r04=tmp;\n\
319     tmp=fmax(r05,r06); r05=fmin(r05,r06); r06=tmp;\n\
320     tmp=fmax(r07,r08); r07=fmin(r07,r08); r08=tmp;\n\
321     tmp=fmax(r09,r10); r09=fmin(r09,r10); r10=tmp;\n\
322     tmp=fmax(r11,r12); r11=fmin(r11,r12); r12=tmp;\n\
323     tmp=fmax(r13,r14); r13=fmin(r13,r14); r14=tmp;\n\
324     tmp=fmax(r17,r18); r17=fmin(r17,r18); r18=tmp;\n\
325     tmp=fmax(r19,r20); r19=fmin(r19,r20); r20=tmp;\n\
326     tmp=fmax(r21,r22); r21=fmin(r21,r22); r22=tmp;\n\
327     tmp=fmax(r23,r24); r23=fmin(r23,r24); r24=tmp;\n\
328     tmp=fmax(r00,r16); r00=fmin(r00,r16); r16=tmp;\n\
329     tmp=fmax(r01,r17); r01=fmin(r01,r17); r17=tmp;\n\
```

```
    tmp=fmax(r02,r18); r02=fmin(r02,r18); r18=tmp;\n\
    tmp=fmax(r03,r19); r03=fmin(r03,r19); r19=tmp;\n\
    tmp=fmax(r04,r20); r04=fmin(r04,r20); r20=tmp;\n\
    tmp=fmax(r05,r21); r05=fmin(r05,r21); r21=tmp;\n\
    tmp=fmax(r06,r22); r06=fmin(r06,r22); r22=tmp;\n\
    tmp=fmax(r07,r23); r07=fmin(r07,r23); r23=tmp;\n\
    tmp=fmax(r08,r24); r08=fmin(r08,r24); r24=tmp;\n\
    tmp=fmax(r08,r16); r08=fmin(r08,r16); r16=tmp;\n\
    tmp=fmax(r09,r17); r09=fmin(r09,r17); r17=tmp;\n\
    tmp=fmax(r10,r18); r10=fmin(r10,r18); r18=tmp;\n\
    tmp=fmax(r11,r19); r11=fmin(r11,r19); r19=tmp;\n\
    tmp=fmax(r12,r20); r12=fmin(r12,r20); r20=tmp;\n\
    tmp=fmax(r13,r21); r13=fmin(r13,r21); r21=tmp;\n\
    // tmp=fmax(r14,r22); r14=fmin(r14,r22); r22=tmp;\n\
    // tmp=fmax(r15,r23); r15=fmin(r15,r23); r23=tmp;\n\
    // tmp=fmax(r04,r08); r04=fmin(r04,r08); r08=tmp;\n\
    // tmp=fmax(r05,r09); r05=fmin(r05,r09); r09=tmp;\n\
    tmp=fmax(r06,r10); r06=fmin(r06,r10); r10=tmp;\n\
    tmp=fmax(r07,r11); r07=fmin(r07,r11); r11=tmp;\n\
    tmp=fmax(r12,r16); r12=fmin(r12,r16); r16=tmp;\n\
    tmp=fmax(r13,r17); r13=fmin(r13,r17); r17=tmp;\n\
    // tmp=fmax(r14,r18); r14=fmin(r14,r18); r18=tmp;\n\
    // tmp=fmax(r15,r19); r15=fmin(r15,r19); r19=tmp;\n\
    // tmp=fmax(r20,r24); r20=fmin(r20,r24); r24=tmp;\n\
    // tmp=fmax(r02,r04); r02=fmin(r02,r04); r04=tmp;\n\
    // tmp=fmax(r03,r05); r03=fmin(r03,r05); r05=tmp;\n\
    // tmp=fmax(r06,r08); r06=fmin(r06,r08); r08=tmp;\n\
    // tmp=fmax(r07,r09); r07=fmin(r07,r09); r09=tmp;\n\
    tmp=fmax(r10,r12); r10=fmin(r10,r12); r12=tmp;\n\
    tmp=fmax(r11,r13); r11=fmin(r11,r13); r13=tmp;\n\
    // tmp=fmax(r14,r16); r14=fmin(r14,r16); r16=tmp;\n\
    // tmp=fmax(r15,r17); r15=fmin(r15,r17); r17=tmp;\n\
    // tmp=fmax(r18,r20); r18=fmin(r18,r20); r20=tmp;\n\
    // tmp=fmax(r19,r21); r19=fmin(r19,r21); r21=tmp;\n\
    // tmp=fmax(r22,r24); r22=fmin(r22,r24); r24=tmp;\n\
    // tmp=fmax(r01,r02); r01=fmin(r01,r02); r02=tmp;\n\
    // tmp=fmax(r03,r04); r03=fmin(r03,r04); r04=tmp;\n\
    // tmp=fmax(r05,r06); r05=fmin(r05,r06); r06=tmp;\n\
    // tmp=fmax(r07,r08); r07=fmin(r07,r08); r08=tmp;\n\
    // tmp=fmax(r09,r10); r09=fmin(r09,r10); r10=tmp;\n\
    tmp=fmax(r11,r12); r11=fmin(r11,r12); r12=tmp;\n\
    // tmp=fmax(r13,r14); r13=fmin(r13,r14); r14=tmp;\n\
    // tmp=fmax(r15,r16); r15=fmin(r15,r16); r16=tmp;\n\
    // tmp=fmax(r17,r18); r17=fmin(r17,r18); r18=tmp;\n\
    // tmp=fmax(r19,r20); r19=fmin(r19,r20); r20=tmp;\n\
    // tmp=fmax(r21,r22); r21=fmin(r21,r22); r22=tmp;\n\
    // tmp=fmax(r23,r24); r23=fmin(r23,r24); r24=tmp;\n\
\n\
    // copy medians to global memory\n\
    gOutput[BI+channel] = (unsigned char)(r12);\n\
  }\n\
}\n\
";
  size_t kernel_size=strlen(kernel_source);
  //
  //FILE *kernel_file;
  //char fileName[] = KERNEL_FILE_NAME;

  //fopen_s(&kernel_file, fileName, "r");
  //if (kernel_file == NULL) {
  //  fprintf(stderr, "Failed to read kernel from file.\n");
  //  exit(1);
  //}
  //fseek(kernel_file, 0, SEEK_END);
  //kernel_size = ftell(kernel_file);
  //rewind(kernel_file);
  //kernel_source = (char *)malloc(kernel_size + 1);
  //kernel_source[kernel_size] = '\0';
  //int read = fread(kernel_source, sizeof(char), kernel_size, kernel_file);
  //if (read != kernel_size) {
  //  fprintf(stderr, "Error while reading the kernel.\n");
  //  exit(1);
  //}
```

```
403   //fclose(kernel_file);
404
405   /* Create OpenCL context */
406   context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
407
408   /* Create Command Queue */
409   command_queue = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE, &ret);
410
411   /* Create Memory Buffer on device*/
412   cl_mem device_imgSrc, device_imgDst, device_coeffs;
413   device_imgSrc = clCreateBuffer(context, CL_MEM_READ_ONLY, size_in, NULL, &ret);
414   device_imgDst = clCreateBuffer(context, CL_MEM_WRITE_ONLY, size_out, NULL, &ret);
415
416
417   /* Create Kernel Program from the source */
418   program = clCreateProgramWithSource(context, 1, (const char **)&kernel_source,
419     (const size_t *)&kernel_size, &ret);
420
421   /* Build Kernel Program */
422   ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
423   size_t param_value_size, param_value_size_ret;
424
425   if (ret != CL_SUCCESS)
426   {
427     size_t len;
428     char buffer[2048];
429     cl_build_status bldstatus;
430     printf("\nError %d: Failed to build program executable [ %s ]\n", ret, getErrorString(ret)
      );
431     ret = clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_STATUS, sizeof(bldstatus)
      , (void *)&bldstatus, &len);
432     printf("Build Status %d: %s\n", ret, getErrorString(ret));
433     printf("INFO: %s\n", getErrorString(bldstatus));
434     ret = clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_OPTIONS, sizeof(buffer),
      buffer, &len);
435     printf("Build Options %d: %s\n", ret, getErrorString(ret));
436     printf("INFO: %s\n", buffer);
437     ret = clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer),
      buffer, &len);
438     printf("Build Log %d: %s\n", ret, getErrorString(ret));
439     printf("%s\n", buffer);
440     exit(1);
441   }
442
443
444   /* Create OpenCL Kernel */
445   kernel = clCreateKernel(program, KERNEL_FUNCTION, &ret);
446
447   /* Set OpenCL Kernel Parameters */
448   ret = clSetKernelArg(kernel, 0, sizeof(device_imgSrc), (void *)&device_imgSrc);
449   ret = clSetKernelArg(kernel, 1, sizeof(device_imgDst), (void *)&device_imgDst);
450   ret = clSetKernelArg(kernel, 2, sizeof(int), &imgWidth);
451   ret = clSetKernelArg(kernel, 3, sizeof(int), &imgWidthF);
452
453
454   // Copy input data to device memory
455   ret = clEnqueueWriteBuffer(command_queue, device_imgSrc, CL_TRUE, 0,
456     size_in, imgSrc, 0, NULL, NULL);
457
458
459   clFinish(command_queue);
460
461   /* Execute OpenCL Kernel */
462   size_t local_size[] = { LOCAL_SIZE_X, LOCAL_SIZE_Y };
463   size_t global_size[] = { imgWidth, imgHeight };
464
465   time_measure(1);
466
467   cl_event event[1024];
468   for (int runs = 0; runs < KERNEL_RUNS; runs++)
469     ret = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL, global_size, local_size, 0,
      NULL, &event[runs]);
470
```

```
471   if (ret != CL_SUCCESS)
472   {
473     printf("\nError %d: Failed to build program executable [ %s ]\n", ret, getErrorString(ret)
      );
474     exit(1);
475   }
476
477   clWaitForEvents(1, &event[KERNEL_RUNS - 1]);
478
479   double runtime = time_measure(2);
480
481   cl_ulong time_start, time_end;
482   double total_time;
483   clGetEventProfilingInfo(event[0], CL_PROFILING_COMMAND_START, sizeof(time_start), &
      time_start, NULL);
484   clGetEventProfilingInfo(event[KERNEL_RUNS-1], CL_PROFILING_COMMAND_END, sizeof(time_end), &
      time_end, NULL);
485   total_time = time_end - time_start;
486   printf("Total kernel time = %6.4f ms, # of runs: %d\r\n", (total_time / (1000000.0)),
      KERNEL_RUNS);
487   printf("Average single kernel time = %6.4f ms\r\n", (total_time / (1000000.0*KERNEL_RUNS)));
488   double mpixel = (KERNEL_RUNS * 1000.0 * double(imgWidth*imgHeight) / (total_time /
      (1000000.0))) / 1000000;
489   printf("Single run MPixel/s: %4.4f\r\n", mpixel);
490   printf("Meas time: %6.4f ms\r\n", (total_time/1000000.0));
491
492
493
494   /* Copy results from the memory buffer */
495   ret = clEnqueueReadBuffer(command_queue, device_imgDst, CL_TRUE, 0,
496     size_out, imgDst, 0, NULL, NULL);
497
498
499   /* Finalization */
500   ret = clFlush(command_queue);
501   ret = clFinish(command_queue);
502   ret = clReleaseKernel(kernel);
503   ret = clReleaseProgram(program);
504   ret = clReleaseCommandQueue(command_queue);
505   ret = clReleaseContext(context);
506
507   //free(kernel_source);
508
509 }
```