Budapesti Műszaki és Gazdaságtudományi Egyetem
Heterogén számítási rendszerek (VIMIMA15)

Medián szűrő

HÁZI FELADAT

–
–

2017. december

Tartalomjegyzék

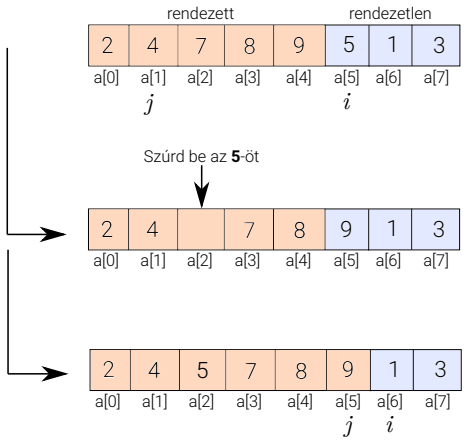
1. fejezet

A választott algoritmus

A házi feladat keretében megvalósított medián szűrőkhöz a **beszúrásos rendezést** (Insertion Sort) választottuk. Azért erre esett a választás, mivel a rendezni kívánt tömb (a szűrőablak által lefedett pixelek tartománya) mérete nem túl nagy (< 64) és kis elemszámú adathalmazra néhány esetben hatékonyabb lehet, mint pl. a gyorsrendezés (Quick Sort). Sajátossága, hogy majdnem rendezett listákra gyorsabb, mint a teljesen rendezetlenekre: teljesen rendezetten lineáris időben lefut. Bár a gyorsrendezés párhuzamosítható (az „oszd meg és uralkodj” elven, divide-and-conquer algorithm), addig a beszúrásos rendezés sajnos nem. Továbbá, a beszúrásos rendezés kevesebb összehasonlítást igényel, mint a kiválasztásos rendezés (Selection Sort), de cserébe több elemet kell mozgatnia a működése során.

Az algoritmus a működése során a tömböt (a rendezi kívánt adatokat) egy rendezett és egy rendezetlen részre osztja. Kezdetben a tömböt teljesen rendezetlennek tekinti, majd kiválasztja a legelső elemet ebből, amely a 0-ás indexű. Ez lesz az első elem, amely a rendezett részbe kerül, hiszen definíció szerint az egy elemű tömb rendezett. Aztán fogja az eggyel jobbra lévő elemet, amely az első elem a rendezetlen részben, és megpróbálja a megfelelő helyre beszúrni a rendezett részbe. Azaz folyamatosan lépeget visszafelé a rendezett részben (a nála kisebb indexű elemek már a rendezett részben vannak) és összehasonlítja a rendezett elemekkel a beszúrandó elemet. A visszafelé lépegetés során mindig eggyel jobbra (nagyobb indexű helyre) tolja az éppen összevetett, rendezett részbeli elemet, ha az nagyobb mint a beszúrni kívánt elem.

Az 1.1. ábrán egy egyszerű rajzzal vázoltuk az algoritmus működését. Az *i* ciklusváltozó mindig az első elemre mutat a rendezetlen részben, míg a *j* a legutolsó rendezett elemről indul és halad a tömb eleje felé és ahol megáll, ott a tömb eleme már kisebb, mint amire az *i* mutat. Így a $j + 1$ helyre beszúrjuk az $a[i]$-t.



1.1. Ábra – *Az algoritmus működésének szemléltetése*

Bár a medián elem megtalálásához **nem kell a teljes tömböt rendezni**, elegendő lenne csak a középső elemig, viszont ez az algoritmus amikor eljut odáig, akkor a nála kisebb elemek a rendezett listában nem biztos hogy véglegesek, így végig kell várni az összes elem sorra kerülését.

2. fejezet

Implementációk különböző platformokon

Ebben a fejezetben a különböző platformokra megvalósított szűrőket mutatjuk be. Az egyes platformokon létrehozott szűrők teljesítményeit (futási idejeiket) a 2.1. táblázatban szereplő számítógéppel vizsgáltuk meg.

Processzor	Videókártya
Intel Core i5-4200U @ 2,6 GHz, 2 mag 4 szál	NVIDIA GeForce 840M, Maxwell

2.1. Táblázat – *A mérésekhez használt számítógép fontosabb részletei*

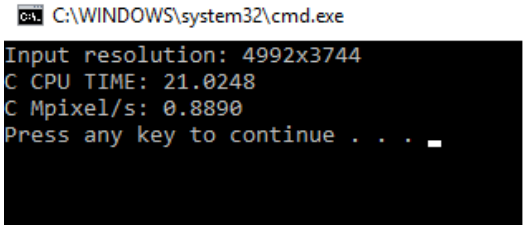
Az összes implementációhoz tartozó forráskód a dokumentáció legvégén, az Irodalomjegyzék után, megtalálható.

2.1. A C nyelvű, optimalizálatlan implementáció

Az optimalizálatlan megoldás során, a bemenetként kapott egyszeres pontosságú lebegőpontos memóriaterületen lineárisan végiglépkedünk, azaz **a kiterjesztett kép közepére helyezett valós kép** pixelein. Minden pixelhez 4 lebegőpontos komponens tartozik: R, G, B és A, ezek egymásután helyezkednek el a memóriában. Az A az alfa komponenst jelenti, amely a pixel átlátszóságáért felel. A kiterjesztett kép, a valós képhez képest, körbe ki van egészítve 2 sornyi és oszlopnyi segédpixellel (fekete pixelekkel), hogy a szűrőablak ne lógjon le és mindig tudjunk mediánt számolni az ablak alatt.

Tehát itt a feladat az, hogy az adott pixelhez tartozó szomszédos pixelek komponenseit felhozzuk 4 db 25 elemű tömbbe, majd ezen a tömbön elvégezzük a rendezést és az így kapott tömb 13. elemét visszaírjuk ugyanebbe a pixelbe, ami a szűrőablak közepén helyezkedik el.

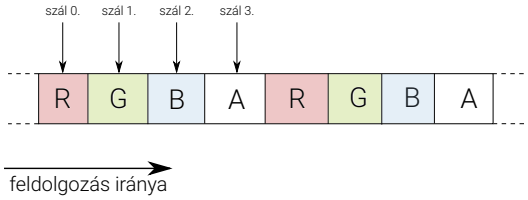
Optimalizálás gyanánt annyit tudtunk megtenni, hogy a szűrőablak feltöltését végző **két for-ciklus-t kézzel kiterítettük**, és így valamelyest megnövekedett a sebessége a programnak (a ciklusfeltétel vizsgálatának elhagyásával).
A 2.1. ábrán az így kapott mérési eredmények szerepelnek, 10 futásra átlagolva.



2.1. Ábra – *Az optimalizálatlan kód által szolgáltatott eredmények, 10 futásra vett átlag alapján*

2.2. A C nyelvű, OpenMP-vel optimalizált implementáció

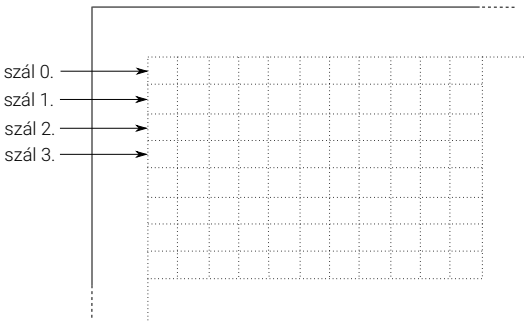
Itt az OpenMP által nyújtott párhuzamosítási lehetőséggel próbáltuk megnövelni a másodpercenként feldolgozott pixelek számát. Először jónak tűnt az az ötlet, hogy dolgozzuk fel mind a négy komponensét egy pixelnek a processzorunk 4 szálával, párhuzamosan, ahogy azt a 2.2. ábrán szemléltettük. De mint kiderült, sokkal lassabb lett,



2.2. Ábra – *Egy pixel feldolgozása négy szállal*

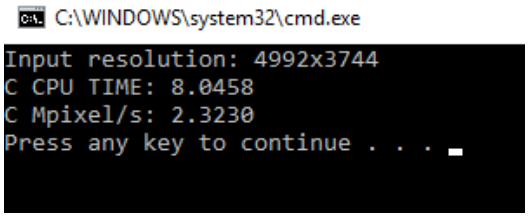
mint az optimalizálatlan megoldás, mivel elsőre nem gondoltunk arra, hogy a szálak pixelenkénti létrehozásának és lebontásának ekkora „ára” lesz. Így ezt a lehetőséget elvetettük.

Helyette azt tettük, hogy **a kép minden sorára külön szálat** indítottunk el OpenMP pragma-val és ezeket a szálakat nem várattuk meg a sorok végén. Ezt mutatja a 2.3. ábra. Gyakorlatilag azt tettük, hogy a külső for-ciklusunkat a forráskódban, ami a sorokon való végiglépkedésért felelős, szétdobtuk a szálak között, dinamikusan ütemezve.



2.3. Ábra – *Négy sor feldolgozása párhuzamosan, 4 szállal*

A futási eredményeket a 2.4. mutatja. Bár nem értük el az elméleti 4-szeres sebességnövekedést, viszont nagyjából 2,5-szer gyorsabb lett így a szűrő.



2.4. Ábra – *Az OpenMP-vel optimalizált kód által szolgáltatott eredmények, 10 futásra vett átlag alapján*

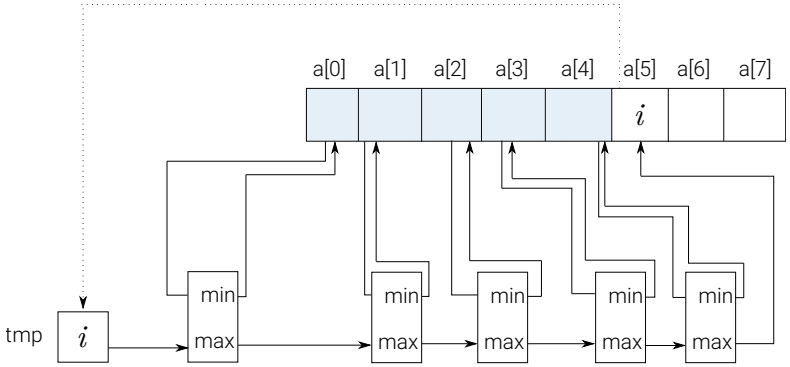
2.3. A C nyelvű, OpenMP-vel és SSE-vel optimalizált implementáció

Az OpenMP-vel történő párhuzamosítás után megpróbáltuk vektorizálni az algoritmusunkat. A vektorizációhoz az intel processzorok által támogatott SSE utasításkészletet használtuk. A beszúrásos rendezés algoritmusát egy az egyben nem sikerült vektorizálni, viszont miután átgondoltuk, hogy milyen SSE utasítások állnak a rendelkezésünkre és milyen is valójában a rendezés, képesek voltunk a rendezéshez egy nagyon hasonló működést vektorizálni.

Mivel az SSE regiszterek 128 bites értékeket képesek tárolni, így kézen fekvő volt az a gondolat, hogy vegyünk ismét egy 25-ös tömböt, egy 25-ös SSE tömböt, és abba töltsük bele a szűrőablakhoz tartozó pixelek összes komponenseit (egy komponens egy 32 bites lebegőpontos érték volt, így belefért négy egy SSE regiszterbe). Majd próbáljuk lefuttatni rá az eddig használt algoritmust. Ezt nem lehetett megtenni, hiszen mondjuk lehet, hogy pl. az R komponenshez tartozó 5. elem már jó helyen van, és nem lehet beszúrni a 0., 1., 2., 3. vagy a 4. elemek közé, viszont lehet, hogy a G komponens 5. elemét még be lehet rakni a saját, rendezett részébe, a tömb elejére, stb. Tehát látható, hogy nagyon körülményes lenne az elágazás és a vektorregiszterek értékeinek pakolása.

Ezért **a vektor komponenseitől független végrehajtású beszúrásos rendezést** kellett kitalálnunk. Onnét indultunk ki, hogy egy két elemű adathalmazt $\{a_1, a_2\}$ növekvő sorrendbe lehet rendezni a minimum és maximum műveletekkel: $\{min(a_1, a_2), max(a_1, a_2)\}$. Ebben az a jó, hogy ez igaz marad akkor is, ha a két elem történetesen vektor: a komponensekre kell ezeket a műveleteket végrehajtani.

Nem volt más hátra, mint ennek az elvnek az általánosítása a beszúrásos rendezésre. Ehhez egy szemléltető ábrát készítettünk a jobb érthetőség kedvéért, amely a 2.5. ábrán látható.

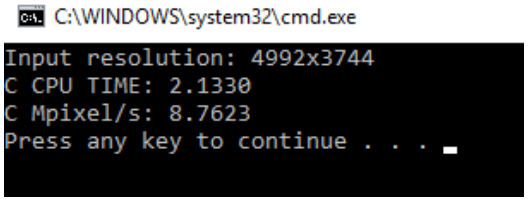


2.5. Ábra – *A minmax beszúrásos rendezés szemléltetése*

Ez az ábra már azt mutatja, hogy az 5. elemet (i) szeretnénk beszúrni a tömb elejére a kékkel színezett, rendezett helyek valamelyikére. Hogy ezt meglessen tenni, ügyelvén arra, hogy az ott már meglévő növekvő sorrendet nem rontjuk el, a tömb elejéről kezdve folyamatosan minimax-olni kell a beszúrni kívánt elemet a kék részben lévő elemekkel. Ennek végeredménye az i elemre a következő lehet: vagy valamelyik ottani elemnél kisebb és akkor helyet cserélnek, és a nagyobb elem terjed tovább az ábrán látható struktúrán. Vagy mindegyiknél nagyobb és akkor az alsó „max ágon” fog végigmenni és végül a saját helyére „pottyan” vissza.

A fentieket, az esetünkben, egészen a középső pozícióban szereplő elem (a[12]) beszúrásáig el kell végezni. Majd ezután, az ezt következő elemek (a[13], a[14], . . . a[24]) beszúrását már csak eddig kell továbbterjeszteni, hiszen a középső elem megtalálása a cél.

A minimax-os beszúrásos rendezés által szolgáltatott futási időket mutatja a 2.6. ábra. Gyorsabb ez, mint a simán csak párhuzamosított megoldás, mivel itt egyszerre négy komponens mediánszűrését tesszük meg, valamint a min/max műveletek az SSE-ben egy órajel ciklus alatt végbemennek.



2.6. Ábra – *A vektorizált kód által szolgáltatott eredmények, 10 futásra vett átlag alapján*

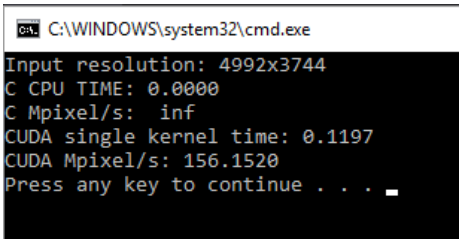
2.4. A CUDA implementáció

A CUDA implementáció során a képet 16×16-os szálblokkokra osztottuk. A szálak, egy blokkon belül, egy pixel teljes feldolgozásáért voltak felelősek: mindhárom komponensét a pixelnek ugyanaz a szál dolgozta fel. A blokkokhoz tartozó pixelek a valós kép pixeleit cserélik le a szűrőablak mediánjára, a beszúrásos rendezés után. Viszont a szűrőablak átlóg a kiterjesztett részre, a blokkok mindegyik szélén két sorral/oszloppal. Így a blokkokhoz tartozó megosztott memóriába ezeket az átlógó pixeleket is be kellet olvasni a globális memóriából. Tehát nem elég egy $[16][16][3]$-as megosztott memória, hanem néggyel több oszlopot és sort kellett lefoglalni az átlógó részekre ($[20][20][3]$).

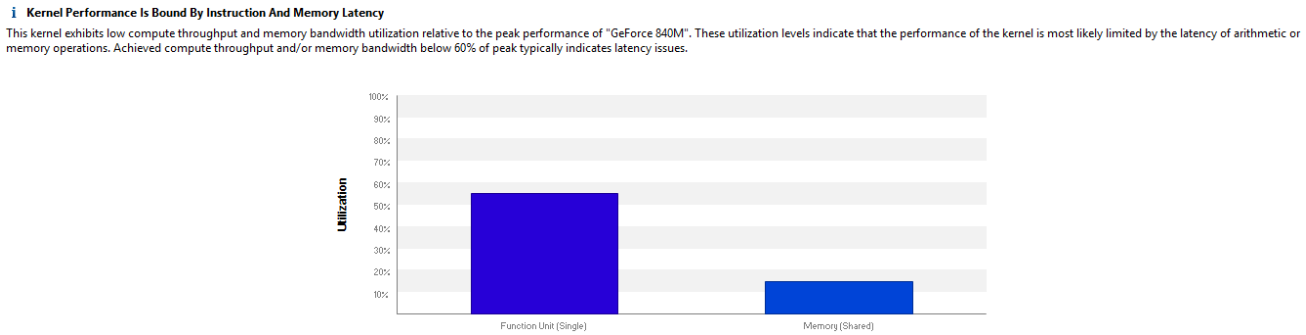
Mivel egy blokkban nincs annyi szál, hogy egyszerre fel tudtuk volna tölteni a megosztott memóriát, ezért a 20 sort 5 ütemben sikerült csak feltölteni. Egy ütemben 4 sort tudtunk csak bemásolni, az 5. sorhoz már nem volt elegendő szál (a maradék 16 szál nem csinált semmit a megosztott memória töltése során).

A másolás végén összeszinkronizáltuk a szálakat, annak érdekében, hogy az utána következő, megosztott memória olvasás során az összes elem már bent legyen a megfelelő helyen. A szálak először beírták az adott pixel komponenseit a megfelelő tömbökbe, amelyek, a fordításidőben ismert indexek miatt, regiszterbe kerültek.

Majd egyenként a regiszterek értékein elvégezték a három komponensre külön-külön a beszúrásos rendezést. Végül, a mediánt kiírták a globális memóriába, unsigned char-ra konvertálás után. A 200 futásra vett eredményt mutatja a 2.11. ábra.



2.7. Ábra – *Az NVIDIA videókártyán futó CUDA implementáció által szolgáltatott eredmények, 10 futásra vett átlag alapján. Ez tartalmazza a másolás idejét is.*



2.8. Ábra – *A Visual Profiler* ***Kernel Analysis*** *eredménye*

Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

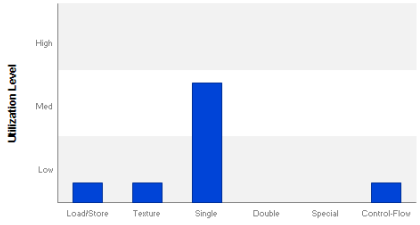
	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	43804800	46.442 GB/s	
Shared Stores	2920320	3.096 GB/s	
Shared Total	46725120	49.538 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	5293698	1.403 GB/s	
Writes	8760966	2.322 GB/s	
Total	14054664	3.725 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	12411360	1.403 GB/s	
Global Stores	8760960	2.322 GB/s	
Texture Reads	10951200	2.903 GB/s	
Unified Total	32123520	6.628 GB/s	Idle Low Medium High Max
Device Memory			
Reads	1756240	465.489 MB/s	
Writes	1757484	465.819 MB/s	
Total	3513724	931.308 MB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen2 x4, 5 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	1.325 kB/s	Idle Low Medium High Max

2.9. Ábra – *A Visual Profiler* ***Memory Bandwidth Analysis*** *eredménye*

Function Unit Utilization

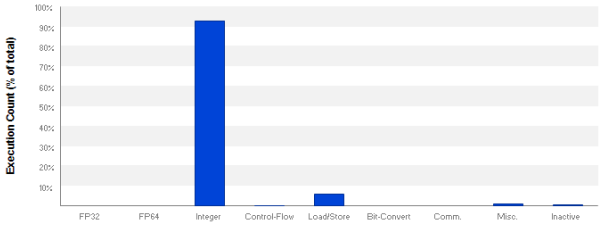
Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

Load/Store - Load and store instructions for shared and constant memory.
Texture - Load and store instructions for local, global, and texture memory.
Single - Single-precision integer and floating-point arithmetic instructions.
Double - Double-precision floating-point arithmetic instructions.
Special - Special arithmetic instructions such as sin, cos, popc, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.

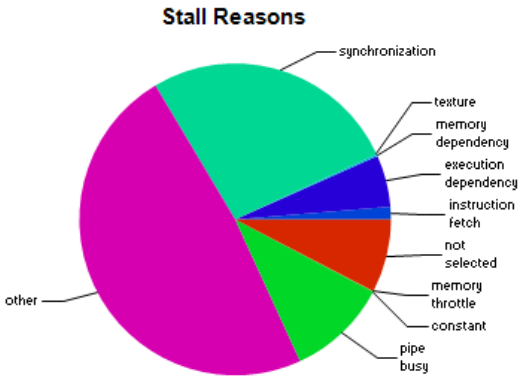


Instruction Execution Counts

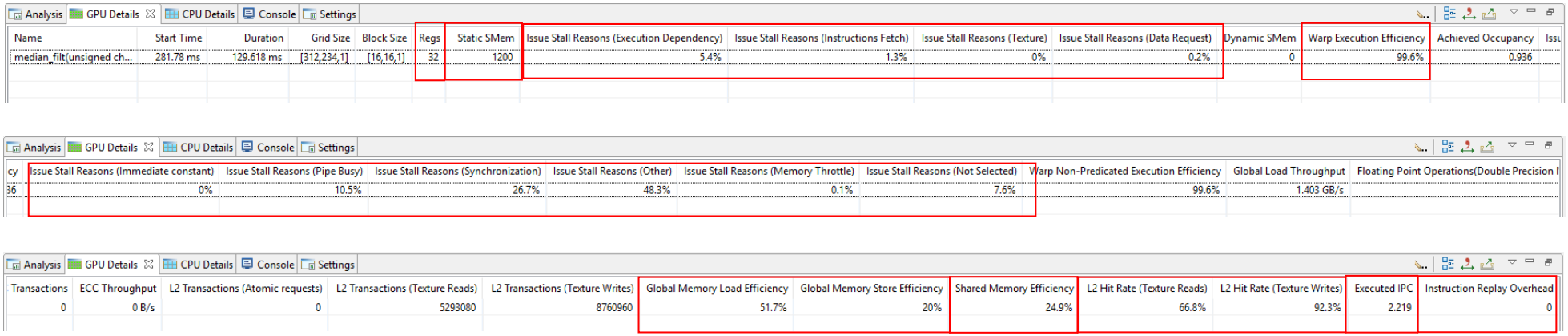
The following chart shows the mix of instructions executed by the kernel. The instructions are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing instructions in that class. The "Inactive" result shows the thread executions that did not execute any instruction because the thread was predicated or inactive due to divergence.



2.10. Ábra – *A Visual Profiler* ***Compute Analysis*** *eredménye*



2.11. Ábra – *A Visual Profiler* ***Latency Analysis*** *eredménye*



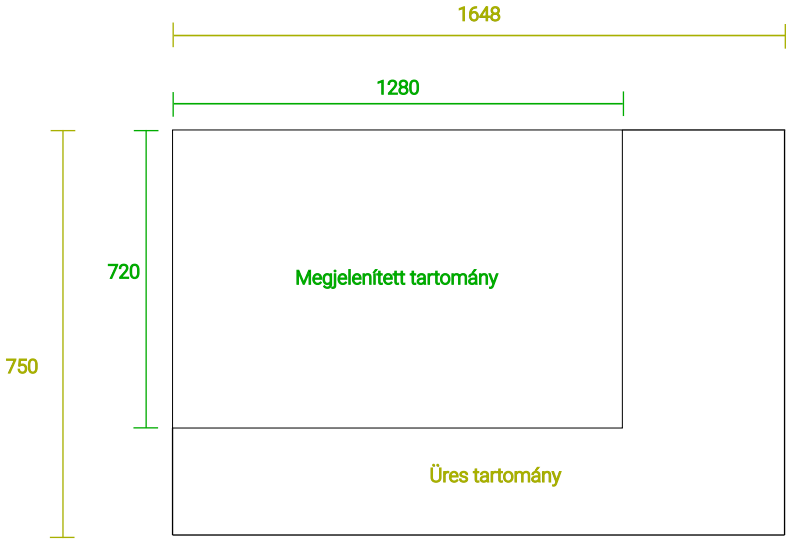
2.12. Ábra – *A Visual Profiler eredményei*

2.5. Az FPGA implementáció Vivado HLS segítségével

Utolsó feladatként az eddigiekben használt algoritmust kellett magas szintű szintézissel FPGA-ra implementálhatóvá tenni. A választott FPGA (Xilinx Artix-7, xc7a35tfgg484-2) ugyanaz volt, mint amire a HLS gyakorlatokon különböző felépítésű FIR szűrőket szintetizáltunk.

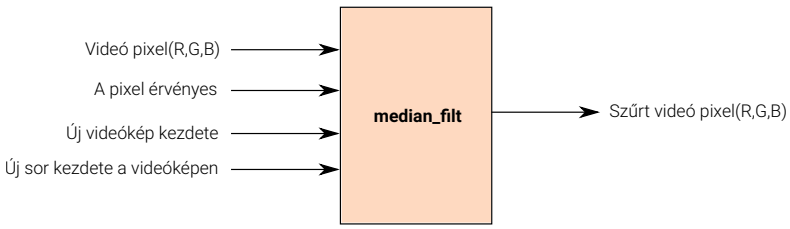
A medián szűrőnek alkalmasnak kellett lennie 1280×720 @ 60 Hz videóformátum fogadására és annak szűrésére (a pixel órajel 74,25 MHz).

A megvalósítás során feltételeztük, hogy nem csak a valós, megjelenítendő képtartomány kerül átküldésre, hanem a kiterjesztett, nem látható tartomány is (1648×750), ahogy az a a 2.13. ábrán látható. Ha ezt elfogadjuk, akkor nem kell, az eddigiekhez képest, a képet kiterjeszteni a széleken dummy sorokkal és oszlopokkal, hanem kihasználhatjuk az üres tartományokban szereplő üres (fekete) pixeleket erre a célra.



2.13. Ábra – *A teljes videókép*

Az általunk tervezett Cpp függvény nagyvonalú blokkdiagramja a 2.14. ábrán látható. Az elképzelésünk az volt, hogy a kiterjesztett tartomány pixelei sorosan (adatfolyam jellegűen) érkeznek be, időzítő jelekkel karöltve és a kimeneten csupán a szűrt videópixelek kerülnek továbbadásra. A kimenetre vezérlőjelek nem feltétlenül szükségesek, mivel HLS kiegészíti a kimenő pixeleket reprezentáló kimeneteket egy érvényes kimenettel, amivel (és plusz a bemeneti vezérlőjelekkel) a szűrőre csatlakozó következő fokozat tudni fogja, hogy hol tart a kép feldolgozása és mikor vehetők el a kész pixelek.



2.14. Ábra – *A szintetizálandó szűrő blokkdiagramja*

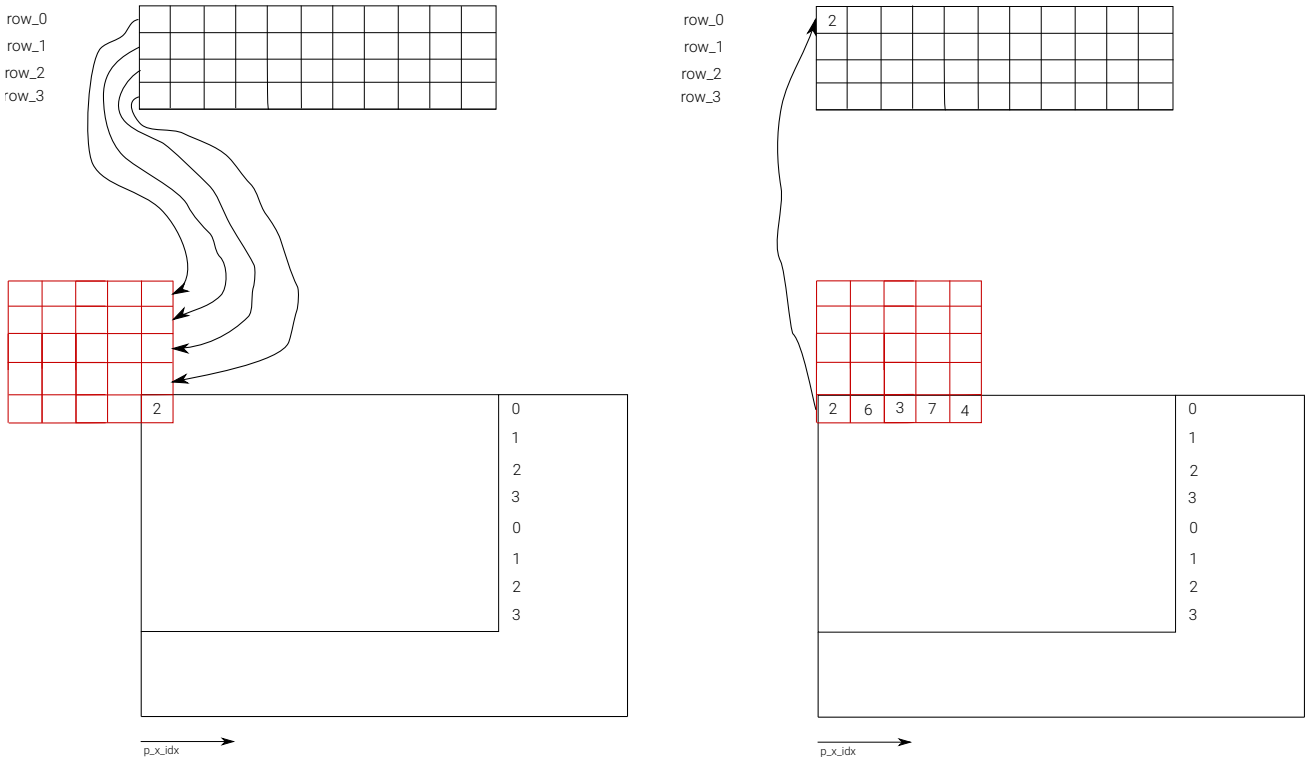
A függvény működése a következő volt: az FPGA felprogramozása után a sorbufferek (**_row_*[]*) és a szűrőablakok (**_fw[]*) regiszterei 0-ra vannak inicializálva. Ha a *new_pic* bemeneten 1-es jött, akkor a modul, a belsejében található (x,y) pixel-koordinátát számon tartó változókat kinullázta és ezután megnézte, hogy a bemenetén lévő adatok érvényesek-e, a *din_valid* bemenet segítségével. Ha ezt 1-esnek találta, akkor a bejövő RGB komponenseket beleírta a három külön szűrőablak jobb alsó sarkába, a **_fw[0]* változókba. Valamint a 4 darab sor bufferből, a x indexnek (*p_x_idx*) megfelelő helyről, kiolvasta a szűrőablak jobb első oszlopába kerülő maradék négy elemet (**_fw[5]*, **_fw[10]*, **_fw[15]* és **_fw[20]*). Ezután a szűrőablak tartalmát átmásolta egy lokális változóba (**_fw_tmp[]*), hiszen ha a szűrőablak tömbjén végeztük volna el a szűrést, akkor az elrontotta

volna a pixelek sorrendjét a következő pixel szűrése során. Mivel az ötlet az az volt, hogy ha lehet akkor először olvassunk be mindent (a bejövő pixelt és az aktuális x-index helyen lévő pixelt a sorbufferből) a szűrőablakba, és az új pixelek csak akkor kerüljenek vissza a sorbufferekbe, hogy ha már „kishiftelődnének" a szűrőablak legalsó sorából. Így elegendő volt 4 sorbuffer deklarálása. Ha végzett a szűréssel, akkor a lokális változó középső elemét kiírja kimenetre, majd a szűrőablakot eggyel balra shifteli, és növeli a x irányú indexet.

Annak érdekében, hogy az előbb elmondottak érthetőbbek legyenek, készítettünk néhány rajzot ezzel kapcsolatban, amelyek a 2.15. és a 2.16. ábrán láthatóak.

[24]	[23]	[22]	[21]	[20]
[19]	[18]	[17]	[16]	[15]
[14]	[13]	[12]	[11]	[10]
[9]	[8]	[7]	[6]	[5]
[4]	[3]	[2]	[1]	[0]

2.15. Ábra – *A szűrablakot reprezentáló tömb indexelése, a jobb alsó sarokba kerülnek bele a bejövő pixelek*



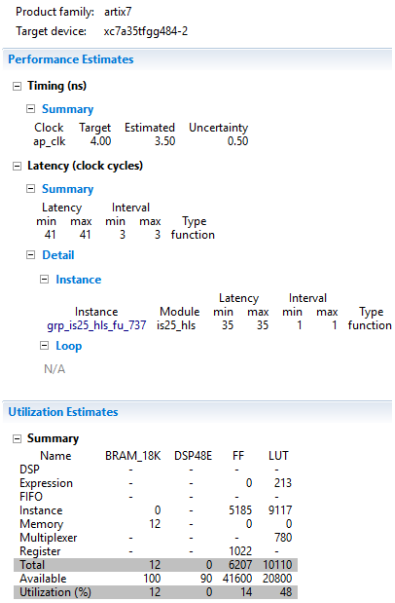
2.16. Ábra – *A bal oldalon a kezdeti pillanatok láthatóak: a szűrőablak jobb alsó sarkába bejön az új pixel, valamint a felette lévő helyekre bemásoljuk a sorbufferek tartalmát (ahol üresek a négyzetek ott az adat 0). A jobb oldali ábrán pedig az, hogy a szűrés után de még a shiftelés előtt a legalsó sor 5. elemét kiírjuk abba a sorbufferbe, amelyik az adott sorhoz tartozik. Ezt a p_y_idx % 4 kifejezéssel egyértelműen meghatározhatjuk.*

A célunk az volt a medián szűrővel kapcsolatban, hogy pixel órajelenként (f_{pixel}) tudjunk kimenetet szolgáltatni, persze miután feltöltődött a pipeline. Így azt tettük, hogy a medián szűrőt modellező függvényt (median_filt()) 3-szor nagyobb órajelről (f_{filt}) járattuk és a HLS PIPELINE II=3 pragma-val láttuk el. Ezzel azt mondtuk meg a szintézernek, hogy minden 3. órajelre vegyen adatot a bemenetről, és így mindig új szűrendő pixelt tudtunk átvenni, hiszen

$$f_{filt} = 3 \cdot f_{pixel}.$$

Mivel sejtettük, hogy a beszúrásos rendezéshez használt minmax-os megoldásunk FPGA-ra szintetizálva nagyon sok konfigurálható logikai elemet (LUT-ot) használna fel, így törekedtünk arra, hogy ebből csak egy kerüljön létrehozásra és ezt az egyet használjuk fel a komponensekhez tartozó tömbök (regiszterek) rendezésére. Ezt persze

csak pipeline-osítva lehet elérni, ezért alkalmaztuk a HLS PIPELINE II=1 pragma-t az is25_hls() függvényre, így a szintézer az egymásutáni függvényhívásokat 1 órajelnyi késleltetéssel indította. Végeredményképpen 3 órajel alatt „betolásra került” mind a három komponens a rendező struktúrába és ennek megfelelően minden pixel órajelre egy új szűrt pixel tudott megjelenni a kimeneten. A szintézis eredményeket a 2.17. és a **??** ábrák foglalják össze.



Product family: artix7
Target device: xc7a35tfgg484-2

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.50	0.50

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
41	41	3	3	function

Detail

Instance

Instance	Module	Latency min	Latency max	Interval min	Interval max	Type
grp_is25_hls_fu_737	is25_hls	35	35	1	1	function

Loop

N/A

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	213
FIFO	-	-	-	-
Instance	0	-	5185	9117
Memory	12	-	0	0
Multiplexer	-	-	-	780
Register	-	-	1022	-
Total	12	0	6207	10110
Available	100	90	41600	20800
Utilization (%)	12	0	14	48

2.17. Ábra – *A szintetizált modell eredményei*

A C nyelvű, optimalizálatlan implementáció

```
#include "defs.h"


static inline void insertion_sort_25(unsigned char arr[]){

    int ii,jj;                      // loop vars
    unsigned char key;              // separator between the sorted and unsorted part of the array

    for(ii = 1; ii < 25; ii++) {
        key = arr[ii];           // the first element (ii=0) is in the sorted part, but the second (ii=1) is not yet
        //jj  = ii - 1;            // let's start the other loop var(jj) from the beginning

        for(jj = ii - 1; jj >= 0 && arr[jj] > key; jj--)
            arr[jj + 1] = arr[jj];
        arr[jj + 1] = key;
    }
}


void median_filt(   int     imgHeight,      int     imgWidth,
                    int     imgHeightF,     int     imgWidthF,
                    int     imgFOffsetH,    int     imgFOffsetW,
                    float*  imgFloatSrc,    float*  imgFloatDst)
{
    int row;
    int col;

    // Stepping through the rows of the picture
    for(row = imgFOffsetH; row < (imgHeight + imgFOffsetH); row++)  {
        // For each pixel in the row
        for(col = imgFOffsetW; col < (imgWidth + imgFOffsetW); col++)   {
            unsigned char r_arr[25];
            unsigned char g_arr[25];
            unsigned char b_arr[25];
            unsigned char a_arr[25];

            int i_p_idx;
            int o_p_idx;

            i_p_idx = 4 * ((row - imgFOffsetH) * imgWidthF + (col - imgFOffsetW));
            o_p_idx = 4 * ((row)* imgWidthF + (col));

            // Uploading the array with the appropriate component values before sorting
            r_arr[0] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[0] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[0] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[0] = (unsigned char)imgFloatSrc[i_p_idx + 3];     i_p_idx += 4;

            r_arr[1] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[1] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[1] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[1] = (unsigned char)imgFloatSrc[i_p_idx + 3];     i_p_idx += 4;

            r_arr[2] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[2] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[2] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[2] = (unsigned char)imgFloatSrc[i_p_idx + 3];     i_p_idx += 4;

            r_arr[3] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[3] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[3] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[3] = (unsigned char)imgFloatSrc[i_p_idx + 3];     i_p_idx += 4;

            r_arr[4] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[4] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[4] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[4] = (unsigned char)imgFloatSrc[i_p_idx + 3];     i_p_idx += 4;

            i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);

            r_arr[5] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[5] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[5] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[5] = (unsigned char)imgFloatSrc[i_p_idx + 3];     i_p_idx += 4;

            r_arr[6] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[6] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[6] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[6] = (unsigned char)imgFloatSrc[i_p_idx + 3];     i_p_idx += 4;

            r_arr[7] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[7] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[7] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[7] = (unsigned char)imgFloatSrc[i_p_idx + 3];     i_p_idx += 4;

            r_arr[8] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[8] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[8] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[8] = (unsigned char)imgFloatSrc[i_p_idx + 3];     i_p_idx += 4;

            r_arr[9] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[9] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[9] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[9] = (unsigned char)imgFloatSrc[i_p_idx + 3];     i_p_idx += 4;

            i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);

            r_arr[10] = (unsigned char)imgFloatSrc[i_p_idx + 0];
```

```
            g_arr[10] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[10] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[10] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[11] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[11] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[11] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[11] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[12] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[12] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[12] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[12] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[13] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[13] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[13] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[13] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[14] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[14] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[14] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[14] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);

            r_arr[15] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[15] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[15] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[15] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[16] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[16] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[16] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[16] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[17] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[17] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[17] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[17] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[18] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[18] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[18] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[18] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[19] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[19] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[19] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[19] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);

            r_arr[20] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[20] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[20] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[20] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[21] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[21] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[21] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[21] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[22] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[22] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[22] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[22] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[23] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[23] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[23] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[23] = (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

            r_arr[24] = (unsigned char)imgFloatSrc[i_p_idx + 0];
            g_arr[24] = (unsigned char)imgFloatSrc[i_p_idx + 1];
            b_arr[24] = (unsigned char)imgFloatSrc[i_p_idx + 2];
            a_arr[24] = (unsigned char)imgFloatSrc[i_p_idx + 3];

            insertion_sort_25(r_arr);   //
            insertion_sort_25(g_arr);   // Apply the insertion sort
            insertion_sort_25(b_arr);   //
            insertion_sort_25(a_arr);   //

            // Swapping the original RGBA values with the RGBA medians: *_arr[12] is the median
            imgFloatDst[o_p_idx + 0] = r_arr[(25 - 1) / 2];
            imgFloatDst[o_p_idx + 1] = g_arr[(25 - 1) / 2];
            imgFloatDst[o_p_idx + 2] = b_arr[(25 - 1) / 2];
            imgFloatDst[o_p_idx + 3] = a_arr[(25 - 1) / 2];
        }
    }
}
```

A C nyelvű, OpenMP-vel párhuzamosított implementáció

```
#include "omp.h"

#include "defs.h"


#define loop_and_insert(start_idx) {                        \
                                                            \
    tmp = arr[start_idx];                                   \
    for(j = start_idx; j >= 1 && tmp < arr[j-1]; j--)   \
        arr[j] = arr[j-1];                                  \
    arr[j] = tmp;                                           \
                                                            \
}


static inline void is25(unsigned char* arr) {
    int j;
    unsigned char tmp;

    loop_and_insert(1)
    loop_and_insert(2)
    loop_and_insert(3)
    loop_and_insert(4)
    loop_and_insert(5)
    loop_and_insert(6)
    loop_and_insert(7)
    loop_and_insert(8)
    loop_and_insert(9)
    loop_and_insert(10)
    loop_and_insert(11)
    loop_and_insert(12)
    loop_and_insert(13)
    loop_and_insert(14)
    loop_and_insert(15)
    loop_and_insert(16)
    loop_and_insert(17)
    loop_and_insert(18)
    loop_and_insert(19)
    loop_and_insert(20)
    loop_and_insert(21)
    loop_and_insert(22)
    loop_and_insert(23)
    loop_and_insert(24)
}


void median_filt(   int     imgHeight,      int     imgWidth,
                    int     imgHeightF,     int     imgWidthF,
                    int     imgFOffsetH,    int     imgFOffsetW,
                    float*  imgFloatSrc,    float*  imgFloatDst)
{
    int row;
    int col;

    #pragma omp parallel
    {
        // Every thread should work on one row at a time and when they finished,
        // they should not wait for each other
        #pragma omp for schedule(dynamic, 1) nowait
        for(row = imgFOffsetH; row < (imgHeight + imgFOffsetH); row++)  {
            // Stepping through the pixels in a row
            for(col = imgFOffsetW; col < (imgWidth + imgFOffsetW); col++)   {

                unsigned char r_arr[25];
                unsigned char g_arr[25];
                unsigned char b_arr[25];
                unsigned char a_arr[25];

                int i_p_idx;
                int o_p_idx;

                i_p_idx = 4 * ((row - imgFOffsetH) * imgWidthF + (col - imgFOffsetW));
                o_p_idx = 4 * ((row)* imgWidthF + (col));


                r_arr[0] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
                g_arr[0] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
                b_arr[0] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
                a_arr[0] =  (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

                r_arr[1] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
                g_arr[1] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
                b_arr[1] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
                a_arr[1] =  (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

                r_arr[2] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
                g_arr[2] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
                b_arr[2] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
                a_arr[2] =  (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

                r_arr[3] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
                g_arr[3] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
                b_arr[3] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
                a_arr[3] =  (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

                r_arr[4] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
                g_arr[4] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
                b_arr[4] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
```

```
a_arr[4] =  (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);

r_arr[5] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[5] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[5] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[5] =  (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

r_arr[6] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[6] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[6] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[6] =  (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

r_arr[7] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[7] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[7] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[7] =  (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

r_arr[8] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[8] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[8] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[8] =  (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

r_arr[9] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[9] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[9] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[9] =  (unsigned char)imgFloatSrc[i_p_idx + 3];    i_p_idx += 4;

i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);

r_arr[10] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[10] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[10] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[10] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

r_arr[11] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[11] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[11] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[11] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

r_arr[12] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[12] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[12] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[12] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

r_arr[13] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[13] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[13] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[13] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

r_arr[14] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[14] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[14] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[14] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);

r_arr[15] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[15] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[15] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[15] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

r_arr[16] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[16] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[16] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[16] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

r_arr[17] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[17] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[17] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[17] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

r_arr[18] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[18] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[18] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[18] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

r_arr[19] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[19] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[19] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[19] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);

r_arr[20] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[20] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[20] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[20] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

r_arr[21] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[21] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[21] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[21] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

r_arr[22] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
g_arr[22] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
b_arr[22] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
a_arr[22] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;
```

```
                r_arr[23] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
                g_arr[23] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
                b_arr[23] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
                a_arr[23] =  (unsigned char)imgFloatSrc[i_p_idx + 3];   i_p_idx += 4;

                r_arr[24] =  (unsigned char)imgFloatSrc[i_p_idx + 0];
                g_arr[24] =  (unsigned char)imgFloatSrc[i_p_idx + 1];
                b_arr[24] =  (unsigned char)imgFloatSrc[i_p_idx + 2];
                a_arr[24] =  (unsigned char)imgFloatSrc[i_p_idx + 3];

                is25(r_arr);    //
                is25(g_arr);    // Apply the insertion sort on the arrays
                is25(b_arr);    //
                is25(a_arr);    //

                // Swapping the original RGBA values with the RGBA medians: *_arr[12] is the median
                imgFloatDst[o_p_idx + 0] = r_arr[(25 - 1) / 2];
                imgFloatDst[o_p_idx + 1] = g_arr[(25 - 1) / 2];
                imgFloatDst[o_p_idx + 2] = b_arr[(25 - 1) / 2];
                imgFloatDst[o_p_idx + 3] = a_arr[(25 - 1) / 2];
            }
        }
    }
}
```

A C nyelvű, OpenMP-vel párhuzamosított és SSE-vel vektorizált megoldás

```
#include "omp.h"
#include "xmmintrin.h" //sse

#include "defs.h"


// i - the index of the element in the SSE array, which will grabbed and propagated through the SSE array.
// j - the "index" of the median element, and also the limit of the for-loop in the define.

#define insertion(i,j) {              \
    n = i;                            \
    a = arr[i-1];                     \
                                      \
    for(int m = 0; m < j - 1; m++)    \
    {                                 \
        b = arr[m];                   \
        c = _mm_min_ps(a,b);          \
        a = _mm_max_ps(a,b);          \
        arr[m] = c;                   \
    }                                 \
    arr[n - 1] = a;                   \
}


static inline void is25_sse(__m128* arr){
    int n;
    __declspec(align(16)) __m128 a ;
    __declspec(align(16)) __m128 b,c;

    insertion(2,2)
    insertion(3,3)
    insertion(4,4)
    insertion(5,5)
    insertion(6,6)
    insertion(7,7)
    insertion(8,8)
    insertion(9,9)
    insertion(10,10)
    insertion(11,11)
    insertion(12,12)
    insertion(13,13)
    insertion(14,14)    //  Inserting and sorting the array
    insertion(15,14)    //  only until the index of the median value is not reached
    insertion(16,14)    //
    insertion(17,14)    //
    insertion(18,14)    //
    insertion(19,14)    //
    insertion(20,14)    //
    insertion(21,14)    //
    insertion(22,14)    //
    insertion(23,14)    //
    insertion(24,14)    //
    insertion(25,14)    //
}


void median_filt(   int imgHeight,      int imgWidth,
                    int imgHeightF,     int imgWidthF,
                    int imgFOffsetH,    int imgFOffsetW,
                    float *imgFloatSrc, float *imgFloatDst)
{
    int row;
    int col;

    #pragma omp parallel
    {
        // Every thread should work on one row at a time and when they finished,
        // they should not wait for each other
        #pragma omp for schedule(dynamic, 1) nowait
        for(row = imgFOffsetH; row < (imgHeight + imgFOffsetH); row++)  {
            // Stepping through the pixels in a row
            for(col = imgFOffsetW; col < (imgWidth + imgFOffsetW); col++)   {

                __declspec(align(16)) __m128 rgba_arr[FILTER_W*FILTER_H];

                int i_p_idx;
                int o_p_idx;

                i_p_idx = 4 * ((row - imgFOffsetH) * imgWidthF + (col - imgFOffsetW));
                o_p_idx = 4 * ((row)* imgWidthF + (col));

                // Uploading the array with the appropriate component values before sorting
                rgba_arr[0] = _mm_load_ps(  (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[1] = _mm_load_ps(  (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[2] = _mm_load_ps(  (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[3] = _mm_load_ps(  (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[4] = _mm_load_ps(  (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;

                i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);

                rgba_arr[5] = _mm_load_ps(  (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[6] = _mm_load_ps(  (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[7] = _mm_load_ps(  (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[8] = _mm_load_ps(  (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[9] = _mm_load_ps(  (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;

                i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);
```

```
                rgba_arr[10] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[11] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[12] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[13] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[14] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;

                i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);

                rgba_arr[15] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[16] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[17] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[18] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[19] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;

                i_p_idx = i_p_idx + 4 * (imgWidthF - FILTER_W);

                rgba_arr[20] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[21] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[22] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[23] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;
                rgba_arr[24] = _mm_load_ps( (float* const)(imgFloatSrc + i_p_idx) );      i_p_idx += 4;

                // Run the vectorized insertion sort on the array
                is25_sse(rgba_arr);

                _mm_stream_ps((float* const)(imgFloatDst + o_p_idx),rgba_arr[12]); // The middle element is the median
            }
        }
    }
}
```

A CUDA implementáció

```
#include <stdio.h>
#include "time.h"
#include "defs.h"
#include "func.h"


#define insertion(i,j) {              \
    n = i;                            \
    a = arr[i-1];                     \
                                      \
    for(int m = 0; m < j - 1; m++)    \
    {                                 \
        b = arr[m];                   \
        c = min(a,b);                 \
        a = max(a,b);                  \
        arr[m] = c;                   \
    }                                 \
    arr[n - 1] = a;                   \
}


__device__ inline void is25_cuda(int* arr){
    int n;
    int a ;
    int b,c;

    insertion(2,2)
        insertion(3,3)
        insertion(4,4)
        insertion(5,5)
        insertion(6,6)
        insertion(7,7)
        insertion(8,8)
        insertion(9,9)
        insertion(10,10)
        insertion(11,11)
        insertion(12,12)
        insertion(13,13)
        insertion(14,14)    //  Inserting and sorting the array
        insertion(15,14)    //  only until the index of the median value is not reached
        insertion(16,14)    //
        insertion(17,14)    //
        insertion(18,14)    //
        insertion(19,14)    //
        insertion(20,14)    //
        insertion(21,14)    //
        insertion(22,14)    //
        insertion(23,14)    //
        insertion(24,14)    //
        insertion(25,14)    //
}

__global__ void median_filt(unsigned char* gInput,unsigned char* gOutput,int imgWidth,int imgWidthF)
{
    // The row and column indices will be generated
    // from the blockID and its dimension,
    // and from the threadID.

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;


    // The output pixel index: it is used when the median
    // will be written back to the global memory.

    int o_p_idx = (row*imgWidth + col) * 3;

    // Local 32 bit integer resources for the temporary calculations.
    // Will be registers.

    int r_arr[25];
    int g_arr[25];
    int b_arr[25];

    // Declaring Shared memory for fast reading and writing in the Block.
    // For each component, a 20 x 20 block will be allocated.
    // The not extented picture will be divided into 16x16 regions, and therefore
    // a [16][16][3] SHMem would be enough, but in our case, the input is the extended picture
    // and we also need to load the two dummy rows and columns. So it is better
    // to allocate a [16 + 4][16 + 4][3], because the filter window will eventually overlap with
    // the other blocks.
    __shared__ unsigned char sh_mem[16 + 4][16 + 4][3];

    // 1D linear thread ID in the block.
    // It is like we would be in blockDim wide picture.
    int thid_1d = threadIdx.y * blockDim.x + threadIdx.x;

    // Creating the indices for the pixel components
    // from the linear thread ID.
    int rgb  = thid_1d % 3;

    // The column index of the Shared memory.
    // The threadID is generated with respective to the pixel components.
    // So first, we need to divide it by 3, to get the linear pixel index.
    // Then apply the modulus on it.
    int shm_col  = (thid_1d / 3) % 20;

    // The row index of the Shared memory.
```

```
    // One row in the Shared memory, there are 60 elements (bytes).
    // If we divide the linear threadID by the length of the row and
    // floor the value, then we end up with the row index.
    int shm_row  = thid_1d / 60;


    // Calculating the base address of the rows, from the threads will load in values.
    // This is giving back the baseaddress of the upper left corner of the blocks.
    int row_base_idx = ( (blockIdx.y * blockDim.y) * 3 * imgWidthF )  + (blockIdx.x * blockDim.x) * 3 + (shm_row * 3 * imgWidthF);


    // Updating the shared memory with the threads.
    // Using 240 threads from the 256 to load up the Shared memory in 5 runs.
    if(thid_1d <(3 * 20 * 4)) {

        //Uploading the 20 rows of the Shared mem in 5 runs
        // First (i=0) just the 0, 1, 2, 3, 4 rows, and then
        // (i=1) the 5, 6, 7, 8, 9, and so on.
        #pragma unroll
        for(int i=0; i<5; i++){

            // Loading in the rows of the Shared memory from the Global memory.
            sh_mem[shm_row + (i * 4)][shm_col][rgb] = gInput[row_base_idx + (thid_1d % 60)];

            // And we jump down by 4 rows.
            row_base_idx += (imgWidthF * 3) * 4;
        }
    }

    //=============================================================================================
    // After updating the shared memory, the threads will be sync.-ed, to make sure everything
    // is loaded into the Shared memory
    __syncthreads();



    // Loading the pixels under the filter window from shared memory to registers.
    // Because the indices are constant during compile time, registers will be inferred from them.
    #pragma unroll 5
    for(int fy=0; fy<5; fy++) {
        #pragma unroll 5
        for(int fx=0; fx<5; fx++) {
            // Uploading the registers with the pixel components.
            r_arr[(fy * 5) + fx] = (int) sh_mem[threadIdx.y + fy][threadIdx.x + fx][0];
            g_arr[(fy * 5) + fx] = (int) sh_mem[threadIdx.y + fy][threadIdx.x + fx][1];
            b_arr[(fy * 5) + fx] = (int) sh_mem[threadIdx.y + fy][threadIdx.x + fx][2];
        }
    }

    is25_cuda(r_arr); //
    is25_cuda(g_arr); // Applying the insertion sort.
    is25_cuda(b_arr); //


    gOutput[o_p_idx + 0] = (unsigned char)(r_arr[12]); //
    gOutput[o_p_idx + 1] = (unsigned char)(g_arr[12]); // Writing the medians to the global memory.
    gOutput[o_p_idx + 2] = (unsigned char)(b_arr[12]); //
}
```

```
#include<ap_int.h>

//#include"types.h"


// Parameters for 1280x720 @ 60Hz,
// blank areas are included.
#define EXTND_H_PIXELS  1648
#define EXTND_V_PIXELS  750

// Parameters of the visible area.
#define TRUE_H_PIXELS   1280
#define TRUE_V_PIXELS   720

// Filter properties.
#define FILTER_W        5
#define FILTER_H        5



#define max(a,b) ((a) > (b) ? a : b)
#define min(a,b) ((a) < (b) ? a : b)



typedef ap_int<1>       signal_1;
typedef ap_int<6>       signal_6;
typedef ap_int<11>      signal_11;
typedef ap_int<10>      signal_10;

#define insertion(i,j) {               \
    n = i;                             \
    a = arr[i-1];                      \
                                       \
    for(int m = 0; m < j - 1; m++)     \
    {                                  \
        b = arr[m];                    \
        c = min(a,b);                  \
        a = max(a,b);                  \
        arr[m] = c;                    \
    }                                  \
    arr[n - 1] = a;                    \
}


// The insertion sort's branchless version.
void is25_hls(unsigned char* arr){

#pragma HLS PIPELINE II=1


    unsigned char n;
    unsigned char a ;
    unsigned char b,c;

    insertion(2,2)
        insertion(3,3)
        insertion(4,4)
        insertion(5,5)
        insertion(6,6)
        insertion(7,7)
        insertion(8,8)
        insertion(9,9)
        insertion(10,10)
        insertion(11,11)
        insertion(12,12)
        insertion(13,13)
        insertion(14,14)    //  Inserting and sorting the array
        insertion(15,14)    //  only until the index of the median value is not reached
        insertion(16,14)    //
        insertion(17,14)    //
        insertion(18,14)    //
        insertion(19,14)    //
        insertion(20,14)    //
        insertion(21,14)    //
        insertion(22,14)    //
        insertion(23,14)    //
        insertion(24,14)    //
        insertion(25,14)    //
}



void median_filt(
    unsigned char*      din_r,
    unsigned char*      din_g,
    unsigned char*      din_b,
    unsigned char*              din_valid,
    unsigned char*              new_pic,
    unsigned char*              new_line,
    unsigned char*      dout_r,
    unsigned char*      dout_g,
    unsigned char*      dout_b
)
{
#pragma HLS PIPELINE II=3
```

```
// These arrays will represent memories for the pixels in a row.
// Four of them are sufficient, even though a 5x5 filter window is used.
// Every time, when the filter window moves to the right by one,
// 5 elements are shifted out, namely in the leftmost column.
// These are the [4] [9] [14] [19] [24] elements.
// The [9], [14], [19] and [24] elements are coming from one of the
// row memories. But the [4] is coming from the first row (at the bottom)
// of the filter. Before it gets shifted out, it should be written back to
// the appropriate row memory, because later, when the filter gets lower
// in the picture, this value will contribute to a median value.

static unsigned char r_row_0[TRUE_H_PIXELS] ={0};
static unsigned char r_row_1[TRUE_H_PIXELS] ={0};
static unsigned char r_row_2[TRUE_H_PIXELS] ={0};
static unsigned char r_row_3[TRUE_H_PIXELS] ={0};

static unsigned char g_row_0[TRUE_H_PIXELS] ={0};
static unsigned char g_row_1[TRUE_H_PIXELS] ={0};
static unsigned char g_row_2[TRUE_H_PIXELS] ={0};
static unsigned char g_row_3[TRUE_H_PIXELS] ={0};

static unsigned char b_row_0[TRUE_H_PIXELS] ={0};
static unsigned char b_row_1[TRUE_H_PIXELS] ={0};
static unsigned char b_row_2[TRUE_H_PIXELS] ={0};
static unsigned char b_row_3[TRUE_H_PIXELS] ={0};


// Pragma used here to infer registers from the Cpp arrays below.
// The 'fw' stands for filter window.
// For each component, there is a separate filter window.

static unsigned char r_fw[FILTER_W * FILTER_H] ={0};
#pragma HLS ARRAY_PARTITION variable=r_fw complete dim=0
static unsigned char g_fw[FILTER_W * FILTER_H] ={0};
#pragma HLS ARRAY_PARTITION variable=g_fw complete dim=0
static unsigned char b_fw[FILTER_W * FILTER_H] ={0};
#pragma HLS ARRAY_PARTITION variable=b_fw complete dim=0

// The median filtering cannot be executed on the filter windows,
// because it would corrupt the order of the values in it.
// And eventually the generated output and filtered video frame.
// For this reason, local resources are used, into which the contents of the
// filter window are passed and then the median filtering are performed
// on these.

unsigned char r_fw_tmp[FILTER_W * FILTER_H];
#pragma HLS ARRAY_PARTITION variable=r_fw_tmp complete dim=0
unsigned char g_fw_tmp[FILTER_W * FILTER_H];
#pragma HLS ARRAY_PARTITION variable=g_fw_tmp complete dim=0
unsigned char b_fw_tmp[FILTER_W * FILTER_H];
#pragma HLS ARRAY_PARTITION variable=b_fw_tmp complete dim=0

// Counters for the pixel (x,y) indices.
static signal_11 p_x_idx = 0;
static signal_10 p_y_idx = 0;
//static int p_x_idx;
//static int p_y_idx;

// If a new picture is going to be started,
// then the pixel's coordinate should be reverted to zero.

if(*new_pic == 1) {
    p_x_idx = 0;
    p_y_idx = 0;
}

// If a new line of an ongoing picture will be transmitted,
// then the x-index should be moved back to zero,
// and the y-index should be incremented by one.

if(*new_line == 1 && p_y_idx < EXTND_V_PIXELS - 1){
    p_x_idx = 0;
    p_y_idx++;
}

// If the new pixel is arrived and can be sampled on the inputs.
if(*din_valid == 1) {

    // Then it (the RGB components of it) is stored down
    // to the right bottom of the filter (*_fw[0])

    r_fw[0] = *din_r;
    g_fw[0] = *din_g;
    b_fw[0] = *din_b;

    // And also the remaining elements ( [5] [10] [15] [20] )
    // in the rightmost column of the filter window
    // are read from the *_row_*  memories.
    // The order is not strict and not relevant, because
    // it does not affect the outcome of the median filter process.
    // If, let's say, in the 3rd will be the values from row_1 and
    // in the 2nd will be from row_3, etc.
    // And also the inherent decoding logics can be omitted and spared.

    r_fw[20] = r_row_0[p_x_idx];
    g_fw[20] = g_row_0[p_x_idx];
    b_fw[20] = b_row_0[p_x_idx];

    r_fw[15] = r_row_1[p_x_idx];
```

```
    g_fw[15] = g_row_1[p_x_idx];
    b_fw[15] = b_row_1[p_x_idx];

    r_fw[10] = r_row_2[p_x_idx];
    g_fw[10] = g_row_2[p_x_idx];
    b_fw[10] = b_row_2[p_x_idx];

    r_fw[5] = r_row_3[p_x_idx];
    g_fw[5] = g_row_3[p_x_idx];
    b_fw[5] = b_row_3[p_x_idx];

}


// Passing the values in the filter window to local
// resources (registers).
for_cp: for(int i=0; i<FILTER_W*FILTER_H; i++) {
#pragma HLS UNROLL
    r_fw_tmp[i] = r_fw[i];
    g_fw_tmp[i] = g_fw[i];
    b_fw_tmp[i] = b_fw[i];
}


// The values are now ready to be filtered.
is25_hls(r_fw_tmp);
is25_hls(g_fw_tmp);
is25_hls(b_fw_tmp);




// The valid values from *_fw[4]
// should be stored down into the appropriate row memory.
if(p_x_idx - (FILTER_W - 1) >= 0) {
    if(p_y_idx % 4 == 0) {
        r_row_0[p_x_idx - (FILTER_W - 1)] = r_fw[4];
        g_row_0[p_x_idx - (FILTER_W - 1)] = g_fw[4];
        b_row_0[p_x_idx - (FILTER_W - 1)] = b_fw[4];
    }
    if(p_y_idx % 4 == 1) {
        r_row_1[p_x_idx - (FILTER_W - 1)] = r_fw[4];
        g_row_1[p_x_idx - (FILTER_W - 1)] = g_fw[4];
        b_row_1[p_x_idx - (FILTER_W - 1)] = b_fw[4];
    }
    if(p_y_idx % 4 == 2) {
        r_row_2[p_x_idx - (FILTER_W - 1)] = r_fw[4];
        g_row_2[p_x_idx - (FILTER_W - 1)] = g_fw[4];
        b_row_2[p_x_idx - (FILTER_W - 1)] = b_fw[4];
    }
    if(p_y_idx % 4 == 3) {
        r_row_3[p_x_idx - (FILTER_W - 1)] = r_fw[4];
        g_row_3[p_x_idx - (FILTER_W - 1)] = g_fw[4];
        b_row_3[p_x_idx - (FILTER_W - 1)] = b_fw[4];
    }
}


// Shifting the filter windows.
// Actually only the leftmost column gets shifted out
// of the window.
r_fw[24] = r_fw[23];
r_fw[23] = r_fw[22];
r_fw[22] = r_fw[21];
r_fw[21] = r_fw[20];

r_fw[19] = r_fw[18];
r_fw[18] = r_fw[17];
r_fw[17] = r_fw[16];
r_fw[16] = r_fw[15];

r_fw[14] = r_fw[13];
r_fw[13] = r_fw[12];
r_fw[12] = r_fw[11];
r_fw[11] = r_fw[10];

r_fw[9] = r_fw[8];
r_fw[8] = r_fw[7];
r_fw[7] = r_fw[6];
r_fw[6] = r_fw[5];

r_fw[4] = r_fw[3];
r_fw[3] = r_fw[2];
r_fw[2] = r_fw[1];
r_fw[1] = r_fw[0];

g_fw[24] = g_fw[23];
g_fw[23] = g_fw[22];
g_fw[22] = g_fw[21];
g_fw[21] = g_fw[20];

g_fw[19] = g_fw[18];
g_fw[18] = g_fw[17];
g_fw[17] = g_fw[16];
g_fw[16] = g_fw[15];

g_fw[14] = g_fw[13];
g_fw[13] = g_fw[12];
```

```
    g_fw[12] = g_fw[11];
    g_fw[11] = g_fw[10];

    g_fw[9] = g_fw[8];
    g_fw[8] = g_fw[7];
    g_fw[7] = g_fw[6];
    g_fw[6] = g_fw[5];

    g_fw[4] = g_fw[3];
    g_fw[3] = g_fw[2];
    g_fw[2] = g_fw[1];
    g_fw[1] = g_fw[0];

    b_fw[24] = b_fw[23];
    b_fw[23] = b_fw[22];
    b_fw[22] = b_fw[21];
    b_fw[21] = b_fw[20];

    b_fw[19] = b_fw[18];
    b_fw[18] = b_fw[17];
    b_fw[17] = b_fw[16];
    b_fw[16] = b_fw[15];

    b_fw[14] = b_fw[13];
    b_fw[13] = b_fw[12];
    b_fw[12] = b_fw[11];
    b_fw[11] = b_fw[10];

    b_fw[9] = b_fw[8];
    b_fw[8] = b_fw[7];
    b_fw[7] = b_fw[6];
    b_fw[6] = b_fw[5];

    b_fw[4] = b_fw[3];
    b_fw[3] = b_fw[2];
    b_fw[2] = b_fw[1];
    b_fw[1] = b_fw[0];

    // Incrementing the x-index of the pixel.
    if(p_x_idx < EXTND_H_PIXELS - 1){
        p_x_idx++;
    }


    // Writing the medians to the outputs
    *dout_r = r_fw_tmp[12];
    *dout_g = g_fw_tmp[12];
    *dout_b = b_fw_tmp[12];
}
```