Heterogén számítási rendszerek
Házi feladat

2018. 12. 06.

Készítették: Papp Lilla és Stráhl Balázs

1. Feladatkiírás

A házi feladat egy 5x5 ablak méretű medián szűrő megvalósítása három technológia segítségével. A bemenet a gyakorlatokon is használt jpg fájl, a szűrést az egyszerűség kedvéért itt is színkomponensenként kell elvégezni.

A házi feladat elsődleges célja adott algoritmus minél hatékonyabb implementációja.

Az általunk választott algoritmus a Batcher odd-even merge sort algortimus.

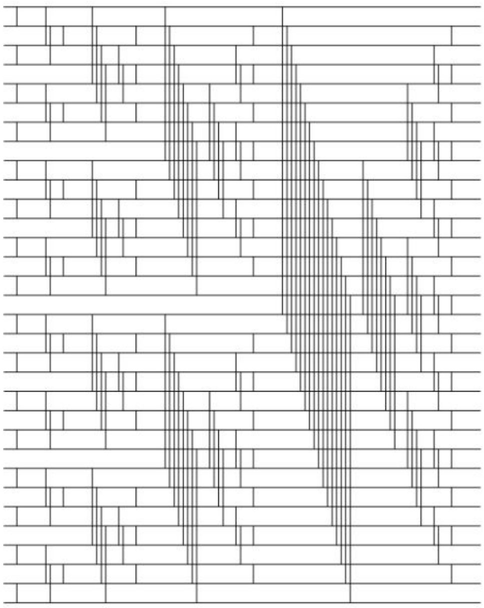
2. Batcher odd-even merge sort algortimus

Az algoritmust elsősorban két rendezett tömb „összefésülésére”, 1 tömbbé rendezésére találták ki. Mivel a kiinduló 2 tömbnek már rendezettnek kell lenni, így azokat is előállíthatjuk (rekurzívan) két résztömb összerendezésével.

A résztömbökre bontást, majd ezek összerendezését addig folytathatjuk, amíg ezek a résztömbök már csak 1-1 eleműek nem lesznek, így gyakorlatilag egy teljes rendezést meg tudunk valósítani ezzel azalgoritmussal.

Mivel nekünk a szűrőablak által lefedett sorbarendezett 25pixelből csupán a medián (12-es indexű) pixel értékének a megtalálása kell, így nem szükséges (és futási idő szempontjából nem is célszerű) a teljes szűrést megvalósítani.

Így azokat az összehasonlításokat és cseréket, amely esettében mindkét pixel indexe nagyobb, mint 12 ÉS egyik pixelt sem hasonlítjuk össze 12 kisebb vagy egyenlő indexű pixellel, nem végeztük el.



3. Medián szűrés

A medián szűrés lényege, hogy az egyes pixelekre helyezett adott méretű (jelenleg 5x5-ös) szűrőablak által lefedett pixeleket eltároljuk, érték alapján sorba rendezzük, majd a mediánt (a rendezett sorban a középső elemet) kiválasztjuk, és az adott pixelt vele helyettesítjük.

A medián szűrést mind a 3 színkomponensre (R, G, B) elvégezzük.

Megjegyzés: mivel a medián szűrést színkomponensenként (melyek értékei nem függenek egymástól) végezzük, így 1 szűrt pixel R, G és B komponensének az értéke nem biztos, hogy az 5x5-ös környezetéven lévő ugyanabból a pixelből származik.

4. Nem vektorizált CPU-s megoldás

Megvalósítás

A beolvasásnál úgy foglalunk helyet a kép pixeleinek a memóriában, mintha 4 színkomponensből állna, illetve float típusúként tároljuk, azért, hogy a vektorizációhoz a pixelek beolvasását ugyanezen kóddal végezhessük el, és a vektorizált megoldást egyszerűen implementálhassuk.

A pixel értékeinek az összehasonlítására és cseréjére használt PIXEL_COMPARE_AND_SWAP – et makróként definiáltuk, hogy ne okozzanak overhead-et a függvényhívással járó stack műveletek.

```
#define PIXEL_COMPARE_AND_SWAP(x, y)      \
      if(arr[(x)] > arr[(y)]) {         \
            tmp = arr[(y)];             \
            arr[(y)] = arr[(x)];        \
            arr[(x)] = tmp;       }

void mergeSort(float * arr)
{
      float tmp;
      // 4x4
      PIXEL_COMPARE_AND_SWAP(0, 1);
      PIXEL_COMPARE_AND_SWAP(2, 3);
      PIXEL_COMPARE_AND_SWAP(0, 2);
      PIXEL_COMPARE_AND_SWAP(1, 3);
      PIXEL_COMPARE_AND_SWAP(1, 2);

      PIXEL_COMPARE_AND_SWAP(4, 5);
      PIXEL_COMPARE_AND_SWAP(6, 7);
      PIXEL_COMPARE_AND_SWAP(4, 6);
      PIXEL_COMPARE_AND_SWAP(5, 7);
      PIXEL_COMPARE_AND_SWAP(5, 6);

      PIXEL_COMPARE_AND_SWAP(0, 4);
      PIXEL_COMPARE_AND_SWAP(1, 5);
      PIXEL_COMPARE_AND_SWAP(2, 6);
      PIXEL_COMPARE_AND_SWAP(3, 7);

      PIXEL_COMPARE_AND_SWAP(2, 4);
      PIXEL_COMPARE_AND_SWAP(3, 5);

      PIXEL_COMPARE_AND_SWAP(1, 2);
      PIXEL_COMPARE_AND_SWAP(3, 4);
      PIXEL_COMPARE_AND_SWAP(5, 6);

      // 4x4
      PIXEL_COMPARE_AND_SWAP(8, 9);
      PIXEL_COMPARE_AND_SWAP(10, 11);
      PIXEL_COMPARE_AND_SWAP(8, 10);
      PIXEL_COMPARE_AND_SWAP(9, 11);
```

```
        PIXEL_COMPARE_AND_SWAP(9, 10);

        PIXEL_COMPARE_AND_SWAP(12, 13);
        PIXEL_COMPARE_AND_SWAP(14, 15);
        PIXEL_COMPARE_AND_SWAP(12, 14);
        PIXEL_COMPARE_AND_SWAP(13, 15);
        PIXEL_COMPARE_AND_SWAP(13, 14);

        PIXEL_COMPARE_AND_SWAP(8, 12);
        PIXEL_COMPARE_AND_SWAP(9, 13);
        PIXEL_COMPARE_AND_SWAP(10, 14);
        PIXEL_COMPARE_AND_SWAP(11, 15);

        PIXEL_COMPARE_AND_SWAP(10, 12);
        PIXEL_COMPARE_AND_SWAP(11, 13);

        PIXEL_COMPARE_AND_SWAP(9, 10);
        PIXEL_COMPARE_AND_SWAP(11, 12);
        PIXEL_COMPARE_AND_SWAP(13, 14);


        PIXEL_COMPARE_AND_SWAP(0, 8);
        PIXEL_COMPARE_AND_SWAP(1, 9);
        PIXEL_COMPARE_AND_SWAP(2, 10);
        PIXEL_COMPARE_AND_SWAP(3, 11);
        PIXEL_COMPARE_AND_SWAP(4, 12);
        PIXEL_COMPARE_AND_SWAP(5, 13);
        PIXEL_COMPARE_AND_SWAP(6, 14);
        PIXEL_COMPARE_AND_SWAP(7, 15);

        PIXEL_COMPARE_AND_SWAP(4, 8);
        PIXEL_COMPARE_AND_SWAP(5, 9);
        PIXEL_COMPARE_AND_SWAP(6, 10);
        PIXEL_COMPARE_AND_SWAP(7, 11);

        PIXEL_COMPARE_AND_SWAP(2, 4);
        PIXEL_COMPARE_AND_SWAP(3, 5);
        PIXEL_COMPARE_AND_SWAP(6, 8);
        PIXEL_COMPARE_AND_SWAP(7, 9);
        PIXEL_COMPARE_AND_SWAP(10, 12);
        PIXEL_COMPARE_AND_SWAP(11, 13);

        PIXEL_COMPARE_AND_SWAP(1, 2);
        PIXEL_COMPARE_AND_SWAP(3, 4);
        PIXEL_COMPARE_AND_SWAP(5, 6);
        PIXEL_COMPARE_AND_SWAP(7, 8);
        PIXEL_COMPARE_AND_SWAP(9, 10);
        PIXEL_COMPARE_AND_SWAP(11, 12);
        PIXEL_COMPARE_AND_SWAP(13, 14);

// Eddig 8x8-as (16 bemenet)

        //4x4
        PIXEL_COMPARE_AND_SWAP(16, 17);

        PIXEL_COMPARE_AND_SWAP(16, 18);
        PIXEL_COMPARE_AND_SWAP(17, 19);

        PIXEL_COMPARE_AND_SWAP(17, 18);
```

```
	PIXEL_COMPARE_AND_SWAP(16, 20);
	PIXEL_COMPARE_AND_SWAP(17, 21);

	PIXEL_COMPARE_AND_SWAP(17, 18);


	PIXEL_COMPARE_AND_SWAP(16, 24);

	PIXEL_COMPARE_AND_SWAP(17, 18);

// 16x16
	PIXEL_COMPARE_AND_SWAP(0, 16);
	PIXEL_COMPARE_AND_SWAP(1, 17);
	PIXEL_COMPARE_AND_SWAP(2, 18);
	PIXEL_COMPARE_AND_SWAP(3, 19);
	PIXEL_COMPARE_AND_SWAP(4, 20);
	PIXEL_COMPARE_AND_SWAP(5, 21);
	PIXEL_COMPARE_AND_SWAP(6, 22);
	PIXEL_COMPARE_AND_SWAP(7, 23);
	PIXEL_COMPARE_AND_SWAP(8, 24);


	PIXEL_COMPARE_AND_SWAP(8, 16);
	PIXEL_COMPARE_AND_SWAP(9, 17);
	PIXEL_COMPARE_AND_SWAP(10, 18);
	PIXEL_COMPARE_AND_SWAP(11, 19);
	PIXEL_COMPARE_AND_SWAP(12, 20);
	PIXEL_COMPARE_AND_SWAP(13, 21);


	PIXEL_COMPARE_AND_SWAP(4, 8);
	PIXEL_COMPARE_AND_SWAP(5, 9);
	PIXEL_COMPARE_AND_SWAP(6, 10);
	PIXEL_COMPARE_AND_SWAP(7, 11);

	PIXEL_COMPARE_AND_SWAP(12, 16);
	PIXEL_COMPARE_AND_SWAP(13, 17);

	PIXEL_COMPARE_AND_SWAP(20, 21);

	PIXEL_COMPARE_AND_SWAP(2, 4);
	PIXEL_COMPARE_AND_SWAP(3, 5);
	PIXEL_COMPARE_AND_SWAP(6, 8);
	PIXEL_COMPARE_AND_SWAP(7, 9);
	PIXEL_COMPARE_AND_SWAP(10, 12);
	PIXEL_COMPARE_AND_SWAP(11, 13);


	PIXEL_COMPARE_AND_SWAP(1, 2);
	PIXEL_COMPARE_AND_SWAP(3, 4);
	PIXEL_COMPARE_AND_SWAP(5, 6);
	PIXEL_COMPARE_AND_SWAP(7, 8);
	PIXEL_COMPARE_AND_SWAP(9, 10);
	PIXEL_COMPARE_AND_SWAP(11, 12);
}

void medianFilter(int imgHeight, int imgWidth, int imgWidthF, int imgFOffsetH,
int imgFOffsetW,  float *imgFloatSrc, float *imgFloatDst)
{
	// Kép sorai
for (int y=imgFOffsetH; y<(imgHeight + imgFOffsetH); y++)
```

```
// Kép oszlopai
for (int x=imgFOffsetW; x<(imgWidth + imgFOffsetW); x++)
// Szín komponensek
for (int rgb = 0; rgb < 4; rgb++)
{
	float medianArray[25];

	for (int medianY = 0; medianY < 5; medianY++)
	for (int medianX = 0; medianX < 5; medianX++)
	{
		medianArray[5*medianY  + medianX] =
		imgFloatSrc[((y+(medianY-2))*imgWidthF + x + (medianX-2))*4 + rgb];
	}

	mergeSort(medianArray);
	imgFloatDst[(y*imgWidth + x) * 4 + rgb] = medianArray[MEDIAN];
}
}
```

Futási idők

Az Odd-even algoritmusnak csak a szükséges részeinek az implementálásával:

C CPU TIME: 38.1550 C Mpixel/s: 0.4898

Megjegyzés

Teljes odd-even algoritmus implementálásával a futási idő:

C CPU TIME: 50.1040 C Mpixel/s: 0.3730 lenne.

A kapott eredmények - az algoritmus lassú működése miatt - a szűrés egyszeri futtatásából születtek.

5. Vektorizált CPU-s megoldás

Megvalósítás

A használt algritmus megegyezik a nem vektorizált megoldásnál használttal. A szűrendő adatokat _m128 típusú vektorba tesszük. Ebbe pont 4 db float érték fér el. Azaz egyszerre szűrjük a 3 színkomponenst. (A beolvasásnál úgy foglalunk helyet a kép pixeleinek a memóriában, mintha 4 színkomponensből állna, így könnyebben elvégezhető a számítás vektorizációja, - azaz, hogy egyszerre szűrjünk mindhárom színkomponens szerint.

A PIXEL_COMPARE_AND_SWAP makróban itt már az egyszerű összehasonlítás helyett (a vektorizált, a színkomponensekre párhuzamosan végrehajtott összehasonlítás miatt) az _mm_min_ps és az _mm_max_ps függvényeket használtuk.

```
#define PIXEL_COMPARE_AND_SWAP(little, big) {           \
       tmpMin = arr[little];                            \
       arr[(little)] = _mm_min_ps(tmpMin, arr[(big)]);  \
       arr[(big)]    = _mm_max_ps(tmpMin, arr[(big)]);} \

void mergeSortAVX(__m128 * arr)
{

       __m128 tmpMin;

       // 4x4
       PIXEL_COMPARE_AND_SWAP(0, 1);
       PIXEL_COMPARE_AND_SWAP(2, 3);
       PIXEL_COMPARE_AND_SWAP(0, 2);
       PIXEL_COMPARE_AND_SWAP(1, 3);
       PIXEL_COMPARE_AND_SWAP(1, 2);

       PIXEL_COMPARE_AND_SWAP(4, 5);
       PIXEL_COMPARE_AND_SWAP(6, 7);
       PIXEL_COMPARE_AND_SWAP(4, 6);
       PIXEL_COMPARE_AND_SWAP(5, 7);
       PIXEL_COMPARE_AND_SWAP(5, 6);

       PIXEL_COMPARE_AND_SWAP(0, 4);
       PIXEL_COMPARE_AND_SWAP(1, 5);
       PIXEL_COMPARE_AND_SWAP(2, 6);
       PIXEL_COMPARE_AND_SWAP(3, 7);

       PIXEL_COMPARE_AND_SWAP(2, 4);
       PIXEL_COMPARE_AND_SWAP(3, 5);

       PIXEL_COMPARE_AND_SWAP(1, 2);
       PIXEL_COMPARE_AND_SWAP(3, 4);
       PIXEL_COMPARE_AND_SWAP(5, 6);
```

```
        // 4x4
        PIXEL_COMPARE_AND_SWAP(8, 9);
        PIXEL_COMPARE_AND_SWAP(10, 11);
        PIXEL_COMPARE_AND_SWAP(8, 10);
        PIXEL_COMPARE_AND_SWAP(9, 11);
        PIXEL_COMPARE_AND_SWAP(9, 10);

        PIXEL_COMPARE_AND_SWAP(12, 13);
        PIXEL_COMPARE_AND_SWAP(14, 15);
        PIXEL_COMPARE_AND_SWAP(12, 14);
        PIXEL_COMPARE_AND_SWAP(13, 15);
        PIXEL_COMPARE_AND_SWAP(13, 14);

        PIXEL_COMPARE_AND_SWAP(8, 12);
        PIXEL_COMPARE_AND_SWAP(9, 13);
        PIXEL_COMPARE_AND_SWAP(10, 14);
        PIXEL_COMPARE_AND_SWAP(11, 15);

        PIXEL_COMPARE_AND_SWAP(10, 12);
        PIXEL_COMPARE_AND_SWAP(11, 13);

        PIXEL_COMPARE_AND_SWAP(9, 10);
        PIXEL_COMPARE_AND_SWAP(11, 12);
        PIXEL_COMPARE_AND_SWAP(13, 14);


        PIXEL_COMPARE_AND_SWAP(0, 8);
        PIXEL_COMPARE_AND_SWAP(1, 9);
        PIXEL_COMPARE_AND_SWAP(2, 10);
        PIXEL_COMPARE_AND_SWAP(3, 11);
        PIXEL_COMPARE_AND_SWAP(4, 12);
        PIXEL_COMPARE_AND_SWAP(5, 13);
        PIXEL_COMPARE_AND_SWAP(6, 14);
        PIXEL_COMPARE_AND_SWAP(7, 15);


        PIXEL_COMPARE_AND_SWAP(4, 8);
        PIXEL_COMPARE_AND_SWAP(5, 9);
        PIXEL_COMPARE_AND_SWAP(6, 10);
        PIXEL_COMPARE_AND_SWAP(7, 11);

        PIXEL_COMPARE_AND_SWAP(2, 4);
        PIXEL_COMPARE_AND_SWAP(3, 5);
        PIXEL_COMPARE_AND_SWAP(6, 8);
        PIXEL_COMPARE_AND_SWAP(7, 9);
        PIXEL_COMPARE_AND_SWAP(10, 12);
        PIXEL_COMPARE_AND_SWAP(11, 13);

        PIXEL_COMPARE_AND_SWAP(1, 2);
        PIXEL_COMPARE_AND_SWAP(3, 4);
        PIXEL_COMPARE_AND_SWAP(5, 6);
        PIXEL_COMPARE_AND_SWAP(7, 8);
        PIXEL_COMPARE_AND_SWAP(9, 10);
        PIXEL_COMPARE_AND_SWAP(11, 12);
        PIXEL_COMPARE_AND_SWAP(13, 14);

// Eddig 8x8-as (16 bemenet)

        //4x4
        PIXEL_COMPARE_AND_SWAP(16, 17);
```

```
        PIXEL_COMPARE_AND_SWAP(16, 18);
        PIXEL_COMPARE_AND_SWAP(17, 19);

        PIXEL_COMPARE_AND_SWAP(17, 18);


        PIXEL_COMPARE_AND_SWAP(16, 20);
        PIXEL_COMPARE_AND_SWAP(17, 21);

        PIXEL_COMPARE_AND_SWAP(17, 18);

        PIXEL_COMPARE_AND_SWAP(16, 24);


        PIXEL_COMPARE_AND_SWAP(17, 18);
        //Eddig egy 8x8-as (De ez csak 9 bemenet)

// 16x16
        PIXEL_COMPARE_AND_SWAP(0, 16);
        PIXEL_COMPARE_AND_SWAP(1, 17);
        PIXEL_COMPARE_AND_SWAP(2, 18);
        PIXEL_COMPARE_AND_SWAP(3, 19);
        PIXEL_COMPARE_AND_SWAP(4, 20);
        PIXEL_COMPARE_AND_SWAP(5, 21);
        PIXEL_COMPARE_AND_SWAP(6, 22);
        PIXEL_COMPARE_AND_SWAP(7, 23);
        PIXEL_COMPARE_AND_SWAP(8, 24);


        PIXEL_COMPARE_AND_SWAP(8, 16);
        PIXEL_COMPARE_AND_SWAP(9, 17);
        PIXEL_COMPARE_AND_SWAP(10, 18);
        PIXEL_COMPARE_AND_SWAP(11, 19);
        PIXEL_COMPARE_AND_SWAP(12, 20);
        PIXEL_COMPARE_AND_SWAP(13, 21);


        PIXEL_COMPARE_AND_SWAP(4, 8);
        PIXEL_COMPARE_AND_SWAP(5, 9);
        PIXEL_COMPARE_AND_SWAP(6, 10);
        PIXEL_COMPARE_AND_SWAP(7, 11);

        PIXEL_COMPARE_AND_SWAP(12, 16);
        PIXEL_COMPARE_AND_SWAP(13, 17);

        PIXEL_COMPARE_AND_SWAP(20, 21);


        PIXEL_COMPARE_AND_SWAP(2, 4);
        PIXEL_COMPARE_AND_SWAP(3, 5);
        PIXEL_COMPARE_AND_SWAP(6, 8);
        PIXEL_COMPARE_AND_SWAP(7, 9);
        PIXEL_COMPARE_AND_SWAP(10, 12);
        PIXEL_COMPARE_AND_SWAP(11, 13);


        PIXEL_COMPARE_AND_SWAP(1, 2);
        PIXEL_COMPARE_AND_SWAP(3, 4);
        PIXEL_COMPARE_AND_SWAP(5, 6);
        PIXEL_COMPARE_AND_SWAP(7, 8);
        PIXEL_COMPARE_AND_SWAP(9, 10);
```

```
        PIXEL_COMPARE_AND_SWAP(11, 12);
}

void medianFilterAVX(int imgHeight, int imgWidth, int imgWidthF, int imgFOffsetH,
int imgFOffsetW, float *imgFloatSrc, float *imgFloatDst)
{
// Kép sorai
#pragma omp parallel for
for (int y = 0; y<imgHeight; y++)
// Az adott sor pixelei
for (int x = 0; x<imgWidth; x++)
{
        __m128 medianArray[25];
        for (int medianY = 0; medianY < 5; medianY++)
        for (int medianX = 0; medianX < 5; medianX++)
        {
                medianArray[5*medianY + medianX] = _mm_load_ps(imgFloatSrc +
                ((y+medianY)*imgWidthF + x + medianX) * 4);
        }

        mergeSortAVX(medianArray);
        _mm_stream_ps(imgFloatDst + (y*imgWidth + x) * 4, medianArray[MEDIAN]);
}
}
```

Futási idők

C CPU TIME: 0.6071   C Mpixel/s: 30.7858

Megjegyzések

A kódban található `#pragma omp parallel for` pragma nékül a futási idő:

C CPU TIME: 3.9234   C Mpixel/s: 4.7637 lenne.

Az eredményt 10-szeri futtatással (a gyakorlaton készített kód alapján az idők átlagolásával) kaptam.

6. GPU-val CUDA-ban történő megoldás

Megjegyzések

Kezdetben a kép pixeleit unsigned char típudú tömbben tároltuk.

Így (200 futást átlagolva) az alábbi futási időket kaptuk:

CUDA single kernel time: 0.1420 CUDA Mpixel/s: 131.6237

A teljesítmény javítása érdekében a kép pixeleit float típusú tömbben tároltuk el.

[S1] megjegyzést írt: Az jobban illeszkedik a GPU-hoz, vagy a 32 bit miatt, esetleg a bankütközések elkerülése miatt?

Így a képfeldolgozás teljesítménye körülbelül 2-szeresére nőtt.

CUDA single kernel time: 0.0692 CUDA Mpixel/s: 269.9464

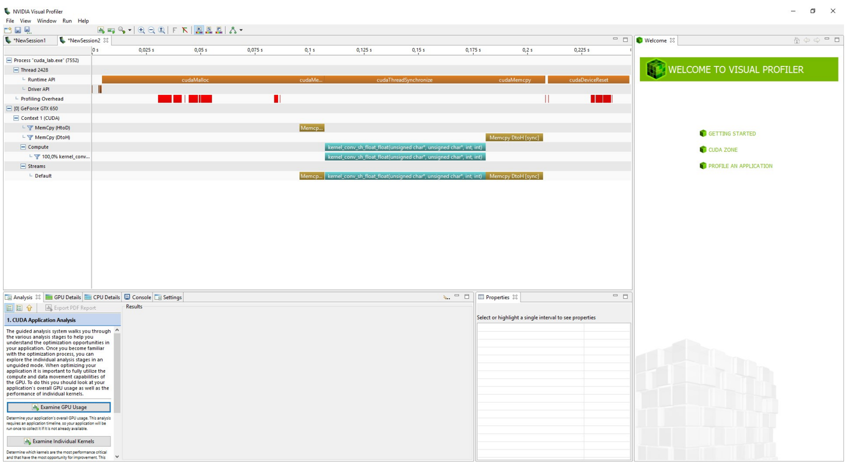
A további teljesítménynövekedés érdekében elhelyeztük az unroll pragmákat.

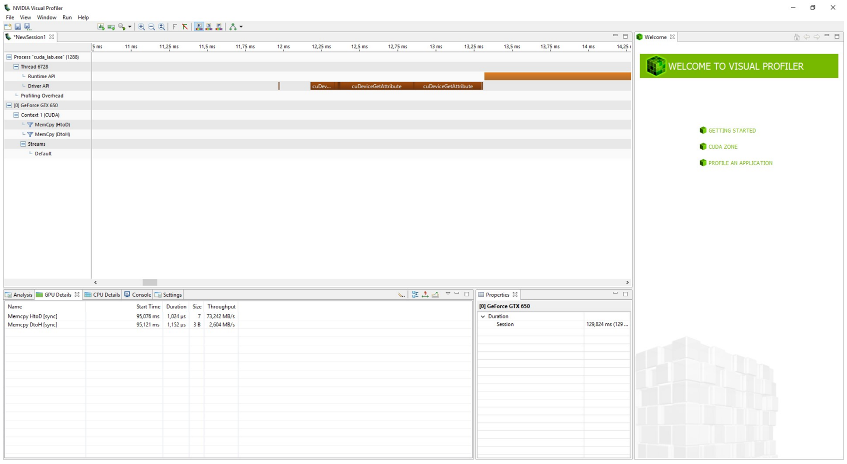
Így körülbelül (az unroll nélküli float adattípust használó verzióhoz viszonyítva) további 11% teljesítménynövekedést sikerült elérnünk.

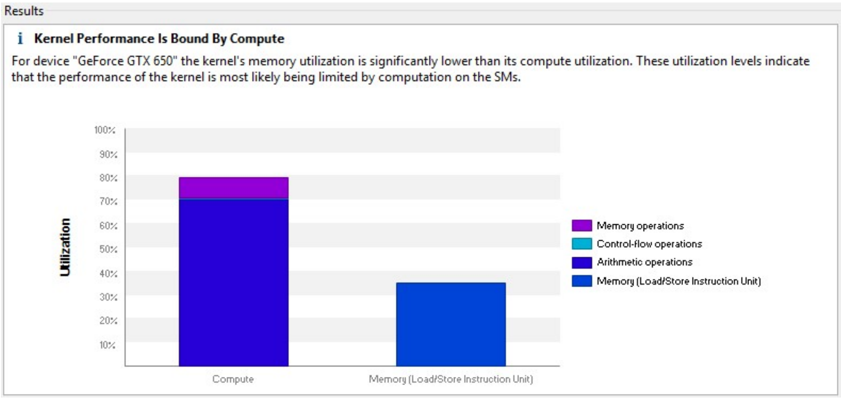
Így a végleges futási idők:

Eredmények

CUDA single kernel time: 0.0624 CUDA Mpixel/s: 299.3347



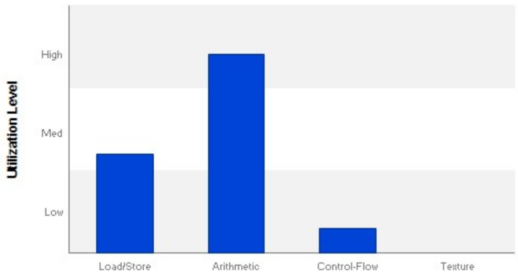

NVIDIA Visual Profiler
WELCOME TO VISUAL PROFILER


Results
Kernel Performance Is Bound By Compute
For device "GeForce GTX 650" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.
Utilization
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
Memory operations
Control-flow operations
Arithmetic operations
Memory (Load/Store Instruction Unit)
Compute
Memory (Load/Store Instruction Unit)

i Function Unit Utilization

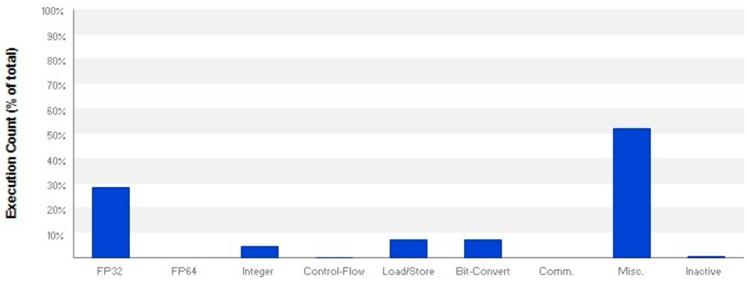
Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

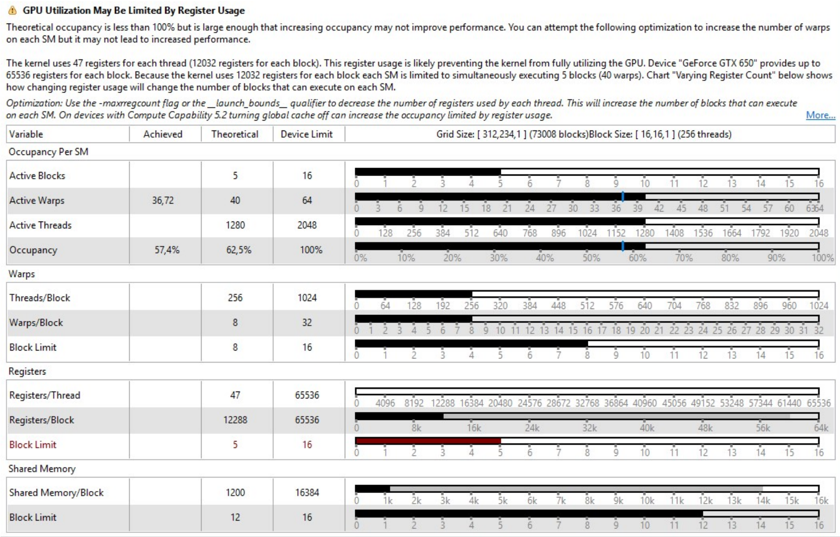
Load/Store - Load and store instructions for local, shared, global, constant, etc. memory.
Arithmetic - All arithmetic instructions including integer and floating-point add and multiply, logical and binary operations, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.
Texture - Texture operations.



i 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.





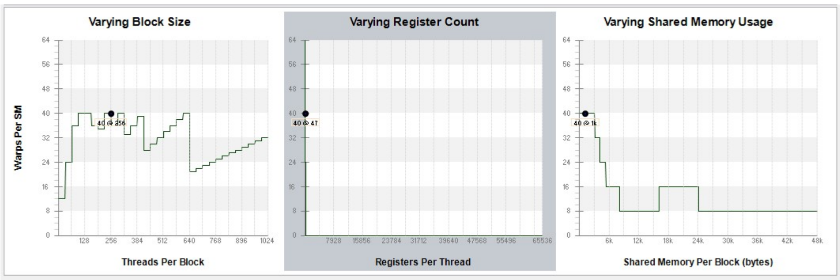
GPU Utilization May Be Limited By Register Usage

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.

The kernel uses 47 registers for each thread (12032 registers for each block). This register usage is likely preventing the kernel from fully utilizing the GPU. Device "GeForce GTX 650" provides up to 65536 registers for each block. Because the kernel uses 12032 registers for each block each SM is limited to simultaneously executing 5 blocks (40 warps). Chart "Varying Register Count" below shows how changing register usage will change the number of blocks that can execute on each SM.

Optimization: Use the -maxrregcount flag or the __launch_bounds__ qualifier to decrease the number of registers used by each thread. This will increase the number of blocks that can execute on each SM. On devices with Compute Capability 5.2 turning global cache off can increase the occupancy limited by register usage. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [312,234,1] (73008 blocks)Block Size: [16,16,1] (256 threads)
Occupancy Per SM				
Active Blocks		5	16	
Active Warps	36,72	40	64	
Active Threads		1280	2048	
Occupancy	57,4%	62,5%	100%	
Warps				
Threads/Block		256	1024	
Warps/Block		8	32	
Block Limit		8	16	
Registers				
Registers/Thread		47	65536	
Registers/Block		12288	65536	
Block Limit		5	16	
Shared Memory				
Shared Memory/Block		1200	16384	
Block Limit		12	16	



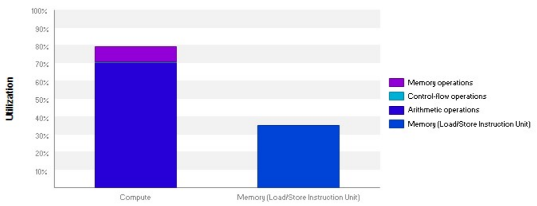
Memory Bandwidth And Utilization

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
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	43804800	152,498 GB/s	
Shared Stores	2920320	10,167 GB/s	
Global Loads	4334850	2,303 GB/s	
Global Stores	4380480	3,05 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	55440450	168,018 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	5293080	2,303 GB/s	
L1 Writes	7008768	3,05 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	12301848	5,353 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	3437399	1,496 GB/s	
Writes	3817078	1,661 GB/s	
Total	7254477	3,157 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	11	4,786 kB/s	Idle Low Medium High Max

Kernel Performance Is Bound By Compute

For device "GeForce GTX 650" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.



Duration: 73,82 ms

Registers/Threads: 47

Shared Memory Efficiency: 51%

Global Memory Load Efficiency: 51%

Global Memory Store Efficiency: 12%

Instructions Executed: 0,699

Static Shared Memory: 1200

A képfeldolgozáshoz 16x16 szálas thread blockot használtunk.

[S2] megjegyzést írt: meg lehetne nézni 32x8 al is

[S3] megjegyzést írt:

```
#define PIXEL_COMPARE_AND_SWAP(x, y)       \
       if(arr[(x)] > arr[(y)]) {           \
       tmp = arr[(y)];                     \
       arr[(y)] = arr[(x)];                \
       arr[(x)] = tmp;      }


__inline__ __device__ void mergeSort(float * arr)
{
       float tmp;
       // 4x4
       PIXEL_COMPARE_AND_SWAP(0, 1);
       PIXEL_COMPARE_AND_SWAP(2, 3);
       PIXEL_COMPARE_AND_SWAP(0, 2);
       PIXEL_COMPARE_AND_SWAP(1, 3);
       PIXEL_COMPARE_AND_SWAP(1, 2);

       PIXEL_COMPARE_AND_SWAP(4, 5);
       PIXEL_COMPARE_AND_SWAP(6, 7);
       PIXEL_COMPARE_AND_SWAP(4, 6);
       PIXEL_COMPARE_AND_SWAP(5, 7);
       PIXEL_COMPARE_AND_SWAP(5, 6);

       PIXEL_COMPARE_AND_SWAP(0, 4);
       PIXEL_COMPARE_AND_SWAP(1, 5);
       PIXEL_COMPARE_AND_SWAP(2, 6);
       PIXEL_COMPARE_AND_SWAP(3, 7);

       PIXEL_COMPARE_AND_SWAP(2, 4);
       PIXEL_COMPARE_AND_SWAP(3, 5);

       PIXEL_COMPARE_AND_SWAP(1, 2);
       PIXEL_COMPARE_AND_SWAP(3, 4);
       PIXEL_COMPARE_AND_SWAP(5, 6);

       // 4x4
       PIXEL_COMPARE_AND_SWAP(8, 9);
       PIXEL_COMPARE_AND_SWAP(10, 11);
       PIXEL_COMPARE_AND_SWAP(8, 10);
       PIXEL_COMPARE_AND_SWAP(9, 11);
       PIXEL_COMPARE_AND_SWAP(9, 10);

       PIXEL_COMPARE_AND_SWAP(12, 13);
       PIXEL_COMPARE_AND_SWAP(14, 15);
       PIXEL_COMPARE_AND_SWAP(12, 14);
       PIXEL_COMPARE_AND_SWAP(13, 15);
       PIXEL_COMPARE_AND_SWAP(13, 14);

       PIXEL_COMPARE_AND_SWAP(8, 12);
       PIXEL_COMPARE_AND_SWAP(9, 13);
       PIXEL_COMPARE_AND_SWAP(10, 14);
       PIXEL_COMPARE_AND_SWAP(11, 15);

       PIXEL_COMPARE_AND_SWAP(10, 12);
       PIXEL_COMPARE_AND_SWAP(11, 13);

       PIXEL_COMPARE_AND_SWAP(9, 10);
       PIXEL_COMPARE_AND_SWAP(11, 12);
       PIXEL_COMPARE_AND_SWAP(13, 14);
```

```
PIXEL_COMPARE_AND_SWAP(0, 8);
PIXEL_COMPARE_AND_SWAP(1, 9);
PIXEL_COMPARE_AND_SWAP(2, 10);
PIXEL_COMPARE_AND_SWAP(3, 11);
PIXEL_COMPARE_AND_SWAP(4, 12);
PIXEL_COMPARE_AND_SWAP(5, 13);
PIXEL_COMPARE_AND_SWAP(6, 14);
PIXEL_COMPARE_AND_SWAP(7, 15);


PIXEL_COMPARE_AND_SWAP(4, 8);
PIXEL_COMPARE_AND_SWAP(5, 9);
PIXEL_COMPARE_AND_SWAP(6, 10);
PIXEL_COMPARE_AND_SWAP(7, 11);

PIXEL_COMPARE_AND_SWAP(2, 4);
PIXEL_COMPARE_AND_SWAP(3, 5);
PIXEL_COMPARE_AND_SWAP(6, 8);
PIXEL_COMPARE_AND_SWAP(7, 9);
PIXEL_COMPARE_AND_SWAP(10, 12);
PIXEL_COMPARE_AND_SWAP(11, 13);

PIXEL_COMPARE_AND_SWAP(1, 2);
PIXEL_COMPARE_AND_SWAP(3, 4);
PIXEL_COMPARE_AND_SWAP(5, 6);
PIXEL_COMPARE_AND_SWAP(7, 8);
PIXEL_COMPARE_AND_SWAP(9, 10);
PIXEL_COMPARE_AND_SWAP(11, 12);
PIXEL_COMPARE_AND_SWAP(13, 14);

// Eddig 8x8-as (16 bemenet)

//4x4
PIXEL_COMPARE_AND_SWAP(16, 17);

PIXEL_COMPARE_AND_SWAP(16, 18);
PIXEL_COMPARE_AND_SWAP(17, 19);

PIXEL_COMPARE_AND_SWAP(17, 18);

PIXEL_COMPARE_AND_SWAP(16, 20);
PIXEL_COMPARE_AND_SWAP(17, 21);

PIXEL_COMPARE_AND_SWAP(17, 18);

PIXEL_COMPARE_AND_SWAP(16, 24);

PIXEL_COMPARE_AND_SWAP(17, 18);
//Eddig egy 8x8-as (De ez csak 9 bemenet)

// 16x16
PIXEL_COMPARE_AND_SWAP(0, 16);
PIXEL_COMPARE_AND_SWAP(1, 17);
PIXEL_COMPARE_AND_SWAP(2, 18);
PIXEL_COMPARE_AND_SWAP(3, 19);
PIXEL_COMPARE_AND_SWAP(4, 20);
PIXEL_COMPARE_AND_SWAP(5, 21);
PIXEL_COMPARE_AND_SWAP(6, 22);
PIXEL_COMPARE_AND_SWAP(7, 23);
PIXEL_COMPARE_AND_SWAP(8, 24);
```

```
	PIXEL_COMPARE_AND_SWAP(8, 16);
	PIXEL_COMPARE_AND_SWAP(9, 17);
	PIXEL_COMPARE_AND_SWAP(10, 18);
	PIXEL_COMPARE_AND_SWAP(11, 19);
	PIXEL_COMPARE_AND_SWAP(12, 20);
	PIXEL_COMPARE_AND_SWAP(13, 21);


	PIXEL_COMPARE_AND_SWAP(4, 8);
	PIXEL_COMPARE_AND_SWAP(5, 9);
	PIXEL_COMPARE_AND_SWAP(6, 10);
	PIXEL_COMPARE_AND_SWAP(7, 11);

	PIXEL_COMPARE_AND_SWAP(12, 16);
	PIXEL_COMPARE_AND_SWAP(13, 17);

	PIXEL_COMPARE_AND_SWAP(20, 21);


	PIXEL_COMPARE_AND_SWAP(2, 4);
	PIXEL_COMPARE_AND_SWAP(3, 5);
	PIXEL_COMPARE_AND_SWAP(6, 8);
	PIXEL_COMPARE_AND_SWAP(7, 9);
	PIXEL_COMPARE_AND_SWAP(10, 12);
	PIXEL_COMPARE_AND_SWAP(11, 13);



	PIXEL_COMPARE_AND_SWAP(1, 2);
	PIXEL_COMPARE_AND_SWAP(3, 4);
	PIXEL_COMPARE_AND_SWAP(5, 6);
	PIXEL_COMPARE_AND_SWAP(7, 8);
	PIXEL_COMPARE_AND_SWAP(9, 10);
	PIXEL_COMPARE_AND_SWAP(11, 12);
}

__global__ void kernel_conv_sh_float_float(unsigned char* gInput, unsigned char*
gOutput, int imgWidth, int imgWidthF)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;
	int col = blockIdx.x * blockDim.x + threadIdx.x;

	//int out_pix = (row*imgWidth + col) * 3;
	//unsigned char arr[31];
	float arr[25];

	__shared__ unsigned char in_shmem[20][60];

	int th1D = threadIdx.y*blockDim.x + threadIdx.x;// thread ID 1D
	int rx = blockIdx.x * blockDim.x;
	int ry = blockIdx.y * blockDim.y;
	int base = (ry*imgWidthF + rx) * 3;

	int wr_x;
	int wr_y;


	if (th1D < 240)
	{
```

```
            #pragma unroll
            for (int i = 0; i<5; i++)
            {
                  wr_y = 4 * i + th1D / 60;
                  wr_x = th1D % 60;

                  in_shmem[wr_y][wr_x] = (float)gInput[base + wr_y * imgWidthF
                  * 3 + wr_x];
                  //
            }
      }
      __syncthreads();

      #pragma unroll 3
      for (int rgb = 0; rgb < 3; rgb++)
      {
            #pragma unroll 5
            for (int medianY = 0; medianY < 5; medianY++)
            {
                  #pragma unroll 5
                  for (int medianX = 0; medianX < 5; medianX++)
                        arr[medianY * 5 + medianX] = in_shmem[threadIdx.y +
                         medianY][(threadIdx.x + medianX) * 3 + rgb];
            }
            mergeSort(arr);
            *(gOutput + (row*imgWidth + col) * 3 + rgb) = (unsigned
             char)arr[MEDIAN];
      }

}
```

```
void cudaMain(int imgHeight, int imgWidth, int imgHeightF, int imgWidthF,
                 unsigned char *imgSrc, unsigned char *imgDst)

{
       double s0, e0;
       double d0;

       unsigned char *gInput, *gOutput;
       // GPU global memory foglalás a bemeneti (kiterjesztett) képnek
       int size_in = imgWidthF*imgHeightF*sizeof(unsigned char) * 3;
       cudaMalloc((void**)&gInput, size_in);
       // GPU global memory foglalás a kimeneti (nem kiterjesztett) képnek
       int size_out = imgWidth*imgHeight*sizeof(unsigned char) * 3;
       cudaMalloc((void**)&gOutput, size_out);

       // 16x16 szálas thread block
       dim3 thrBlock(16, 16);
       dim3 thrGrid(imgWidth/16, imgHeight/16);

       // bemeneti kép másolása host --> GPU
       cudaMemcpy(gInput, imgSrc, size_in, cudaMemcpyHostToDevice);

       // L1/Shared memory konfiguráció: sok cache
       cudaDeviceSetCacheConfig(cudaFuncCachePreferL1);

       // L1/Shared memory konfiguráció: sok cache
       //cudaDeviceSetCacheConfig(cudaFuncCachePreferShared);

       s0 = time_measure(1);
       for (int i = 0; i < KERNEL_RUNS; i++)
       {
              kernel_conv_sh_float_float << <thrGrid, thrBlock >> >(gInput,
              gOutput, imgWidth, imgWidthF);
       }
       cudaThreadSynchronize();
       e0 = time_measure(2);

       // Kimenet másolás: GPU --> host
        cudaMemcpy(imgDst, gOutput, size_out, cudaMemcpyDeviceToHost);

       // GPU memóriák felszabadítása
       cudaFree(gInput); cudaFree(gOutput);

       // Reset (profiler miatt)
       cudaDeviceReset();

       d0 = (double)(e0-s0)/(CLOCKS_PER_SEC*KERNEL_RUNS);
       double mpixel = (imgWidth*imgHeight / d0) / 1000000;
       printf("CUDA single kernel time: %4.4f\n", d0);
       printf("CUDA Mpixel/s: %4.4f\n", mpixel);
}
```

7. HLS megoldás

Megvalósítás

Először definiáltuk a megfelelő adattípusokat. Mindegyik ap_uint típusú, a bitszáma az adott típusú változóban tárolni kívánt maximális értéktől függ.

A módszer alapja, hogy 5 sornyi pixelt eltárolunk a `static pixel_t fiveLine[3][1280][5]` tömbben. A szűréshez színkomponensenként egy-egy 5x5-ös szűrőablakot (egy-egy 5x5-ös tömböt - bufferOrig_Red, bufferOrig_Green, bufferOrig_Blue) használunk. Ezeket minden szűrés esetén balra shifteljük, majd a legutolsó oszlopba tesszük az új pixel értékeket a fiveLine tömbből.

Mivel a 3 bufferOrig értékét a függvény következő lefutásakor újra használjuk (ez megtehető, mivel static tömb), ezért a szűrést nem ezeken a tömbökön, hanem a buffer_Red, buffer_Green és a buffer_Blue tömbökön végezzük.

Ezért minden szűrés előtt a bufferOrig tömböket átmásoljuk a buffer tömbökbe (természetesen színhelyesen – azaz a Redbe, a Redet, stb.)

A szűrést az eddigiekhez hasonlóan a Batche odd-even mergeshort algoritmussal végeztük el.

A legvégén a medián elemet küldtük ki a kimenetre.

A vezérléshez a newLine jelet használtuk.

Felhasznált pragmák

```
#pragma HLS PIPELINE II=1
#pragma HLS ARRAY_PARTITION variable=fiveLine complete dim=3:
A fiveLine tömböt partícionálja az 3. dimenziója mentén.

#pragma HLS ARRAY_PARTITION variable=fiveLine complete dim=1:
A fiveLine tömböt partícionálja az 1. dimenziója mentén.


#pragma HLS ARRAY_PARTITION variable=bufferOrig_Red complete dim=0
A bufferOrig_Red tömböt partícionálja minden dimenziója mentén.

#pragma HLS ARRAY_PARTITION variable=bufferOrig_Green complete dim=0
A bufferOrig_Green tömböt partícionálja minden dimenziója mentén.
```

```
#pragma HLS ARRAY_PARTITION variable=bufferOrig_Blue complete dim=0
A bufferOrig_Blue tömböt partícionálja minden dimenziója mentén.


#pragma HLS ARRAY_PARTITION variable=buffer_Red complete dim=0
A buffer_Red tömböt partícionálja minden dimenziója mentén.

#pragma HLS ARRAY_PARTITION variable=buffer_Green complete dim=0
A buffer_Green tömböt partícionálja minden dimenziója mentén.

#pragma HLS ARRAY_PARTITION variable=buffer_Blue complete dim=0
A buffer_Blue tömböt partícionálja minden dimenziója mentén.

#pragma HLS UNROLL:
Párhuzamosan hajtja végre a ciklus iterációit

#pragma HLS INLINE
Inlineként definiálja az adott függvény

#pragma HLS PIPELINE II=1
Pipeline utasítás végrehajtás
```

A forráskód:

```
//#include <stdint.h>

#include "ap_int.h"
#include "ap_fixed.h"

typedef ap_uint<8> pixel_t;       // pixelek
typedef ap_uint<3> row_t;         // 5 sor
typedef ap_uint<11> col_t;        // 1280 oszlop
typedef ap_uint<3> median_t;      // 5 széles, 5 hosszú

#define PIXEL_COMPARE_AND_SWAP(x, y)    \
      if(arr[(x)] > arr[(y)]) {         \
            tmp = arr[(y)];             \
            arr[(y)] = arr[(x)];        \
            arr[(x)] = tmp;     }

void mergeSort(pixel_t* arr)
{
      #pragma HLS INLINE
      #pragma HLS PIPELINE II=1
      pixel_t tmp;
      // 4x4
      PIXEL_COMPARE_AND_SWAP(0, 1);
      PIXEL_COMPARE_AND_SWAP(2, 3);
      PIXEL_COMPARE_AND_SWAP(0, 2);
      PIXEL_COMPARE_AND_SWAP(1, 3);
      PIXEL_COMPARE_AND_SWAP(1, 2);

      PIXEL_COMPARE_AND_SWAP(4, 5);
      PIXEL_COMPARE_AND_SWAP(6, 7);
      PIXEL_COMPARE_AND_SWAP(4, 6);
```

```
PIXEL_COMPARE_AND_SWAP(5, 7);
PIXEL_COMPARE_AND_SWAP(5, 6);

PIXEL_COMPARE_AND_SWAP(0, 4);
PIXEL_COMPARE_AND_SWAP(1, 5);
PIXEL_COMPARE_AND_SWAP(2, 6);
PIXEL_COMPARE_AND_SWAP(3, 7);

PIXEL_COMPARE_AND_SWAP(2, 4);
PIXEL_COMPARE_AND_SWAP(3, 5);

PIXEL_COMPARE_AND_SWAP(1, 2);
PIXEL_COMPARE_AND_SWAP(3, 4);
PIXEL_COMPARE_AND_SWAP(5, 6);

// 4x4
PIXEL_COMPARE_AND_SWAP(8, 9);
PIXEL_COMPARE_AND_SWAP(10, 11);
PIXEL_COMPARE_AND_SWAP(8, 10);
PIXEL_COMPARE_AND_SWAP(9, 11);
PIXEL_COMPARE_AND_SWAP(9, 10);

PIXEL_COMPARE_AND_SWAP(12, 13);
PIXEL_COMPARE_AND_SWAP(14, 15);
PIXEL_COMPARE_AND_SWAP(12, 14);
PIXEL_COMPARE_AND_SWAP(13, 15);
PIXEL_COMPARE_AND_SWAP(13, 14);

PIXEL_COMPARE_AND_SWAP(8, 12);
PIXEL_COMPARE_AND_SWAP(9, 13);
PIXEL_COMPARE_AND_SWAP(10, 14);
PIXEL_COMPARE_AND_SWAP(11, 15);

PIXEL_COMPARE_AND_SWAP(10, 12);
PIXEL_COMPARE_AND_SWAP(11, 13);

PIXEL_COMPARE_AND_SWAP(9, 10);
PIXEL_COMPARE_AND_SWAP(11, 12);
PIXEL_COMPARE_AND_SWAP(13, 14);


PIXEL_COMPARE_AND_SWAP(0, 8);
PIXEL_COMPARE_AND_SWAP(1, 9);
PIXEL_COMPARE_AND_SWAP(2, 10);
PIXEL_COMPARE_AND_SWAP(3, 11);
PIXEL_COMPARE_AND_SWAP(4, 12);
PIXEL_COMPARE_AND_SWAP(5, 13);
PIXEL_COMPARE_AND_SWAP(6, 14);
PIXEL_COMPARE_AND_SWAP(7, 15);


PIXEL_COMPARE_AND_SWAP(4, 8);
PIXEL_COMPARE_AND_SWAP(5, 9);
PIXEL_COMPARE_AND_SWAP(6, 10);
PIXEL_COMPARE_AND_SWAP(7, 11);

PIXEL_COMPARE_AND_SWAP(2, 4);
PIXEL_COMPARE_AND_SWAP(3, 5);
```

```
PIXEL_COMPARE_AND_SWAP(6, 8);
PIXEL_COMPARE_AND_SWAP(7, 9);
PIXEL_COMPARE_AND_SWAP(10, 12);
PIXEL_COMPARE_AND_SWAP(11, 13);

PIXEL_COMPARE_AND_SWAP(1, 2);
PIXEL_COMPARE_AND_SWAP(3, 4);
PIXEL_COMPARE_AND_SWAP(5, 6);
PIXEL_COMPARE_AND_SWAP(7, 8);
PIXEL_COMPARE_AND_SWAP(9, 10);
PIXEL_COMPARE_AND_SWAP(11, 12);
PIXEL_COMPARE_AND_SWAP(13, 14);
// Eddig 8x8-as (16 bemenet)

//4x4
PIXEL_COMPARE_AND_SWAP(16, 17);
PIXEL_COMPARE_AND_SWAP(18, 19);
PIXEL_COMPARE_AND_SWAP(20, 21);
PIXEL_COMPARE_AND_SWAP(22, 23);

PIXEL_COMPARE_AND_SWAP(16, 18);
PIXEL_COMPARE_AND_SWAP(20, 22);
PIXEL_COMPARE_AND_SWAP(17, 19);
PIXEL_COMPARE_AND_SWAP(21, 23);

PIXEL_COMPARE_AND_SWAP(17, 18);
PIXEL_COMPARE_AND_SWAP(21, 22);


PIXEL_COMPARE_AND_SWAP(16, 20);
PIXEL_COMPARE_AND_SWAP(17, 21);
PIXEL_COMPARE_AND_SWAP(18, 22);
PIXEL_COMPARE_AND_SWAP(19, 23);

PIXEL_COMPARE_AND_SWAP(18, 20);
PIXEL_COMPARE_AND_SWAP(19, 21);

PIXEL_COMPARE_AND_SWAP(17, 18);
PIXEL_COMPARE_AND_SWAP(19, 20);
PIXEL_COMPARE_AND_SWAP(21, 22);

PIXEL_COMPARE_AND_SWAP(16, 24);

PIXEL_COMPARE_AND_SWAP(20, 24);

PIXEL_COMPARE_AND_SWAP(18, 20);
PIXEL_COMPARE_AND_SWAP(19, 21);
PIXEL_COMPARE_AND_SWAP(22, 24);

PIXEL_COMPARE_AND_SWAP(17, 18);
PIXEL_COMPARE_AND_SWAP(19, 20);
PIXEL_COMPARE_AND_SWAP(21, 22);
PIXEL_COMPARE_AND_SWAP(23, 24);
//Eddig egy 8x8-as (De ez csak 9 bemenet)

// 16x16
PIXEL_COMPARE_AND_SWAP(0, 16);
PIXEL_COMPARE_AND_SWAP(1, 17);
```

```
	PIXEL_COMPARE_AND_SWAP(2, 18);
	PIXEL_COMPARE_AND_SWAP(3, 19);
	PIXEL_COMPARE_AND_SWAP(4, 20);
	PIXEL_COMPARE_AND_SWAP(5, 21);
	PIXEL_COMPARE_AND_SWAP(6, 22);
	PIXEL_COMPARE_AND_SWAP(7, 23);
	PIXEL_COMPARE_AND_SWAP(8, 24);


	PIXEL_COMPARE_AND_SWAP(8, 16);
	PIXEL_COMPARE_AND_SWAP(9, 17);
	PIXEL_COMPARE_AND_SWAP(10, 18);
	PIXEL_COMPARE_AND_SWAP(11, 19);
	PIXEL_COMPARE_AND_SWAP(12, 20);
	PIXEL_COMPARE_AND_SWAP(13, 21);
	PIXEL_COMPARE_AND_SWAP(14, 22);
	PIXEL_COMPARE_AND_SWAP(15, 23);


	PIXEL_COMPARE_AND_SWAP(4, 8);
	PIXEL_COMPARE_AND_SWAP(5, 9);
	PIXEL_COMPARE_AND_SWAP(6, 10);
	PIXEL_COMPARE_AND_SWAP(7, 11);

	PIXEL_COMPARE_AND_SWAP(12, 16);
	PIXEL_COMPARE_AND_SWAP(13, 17);


	PIXEL_COMPARE_AND_SWAP(2, 4);
	PIXEL_COMPARE_AND_SWAP(3, 5);

	PIXEL_COMPARE_AND_SWAP(6, 8);
	PIXEL_COMPARE_AND_SWAP(7, 9);

	PIXEL_COMPARE_AND_SWAP(10, 12);
	PIXEL_COMPARE_AND_SWAP(11, 13);


	PIXEL_COMPARE_AND_SWAP(1, 2);
	PIXEL_COMPARE_AND_SWAP(3, 4);
	PIXEL_COMPARE_AND_SWAP(5, 6);
	PIXEL_COMPARE_AND_SWAP(7, 8);
	PIXEL_COMPARE_AND_SWAP(9, 10);
	PIXEL_COMPARE_AND_SWAP(11, 12);
}

typedef enum {RED, GREEN, BLUE}Colors;
#define MEDIAN 2

void medianFilter(pixel_t pixelIn[3], int newLine, pixel_t pixelOut[3])
{
#pragma HLS PIPELINE II=1

	static pixel_t fiveLine[3][1280][5];
	#pragma HLS ARRAY_PARTITION variable=fiveLine complete dim=3
	#pragma HLS ARRAY_PARTITION variable=fiveLine complete dim=1
```

```
	static pixel_t bufferOrig_Red[5][5];
	static pixel_t bufferOrig_Green[5][5];
	static pixel_t bufferOrig_Blue[5][5];
	#pragma HLS ARRAY_PARTITION variable=bufferOrig_Red complete dim=0
	#pragma HLS ARRAY_PARTITION variable=bufferOrig_Green complete dim=0
	#pragma HLS ARRAY_PARTITION variable=bufferOrig_Blue complete dim=0
	// Minden dimenzió mentén partícionáljon

	static pixel_t buffer_Red[5][5];
	static pixel_t buffer_Green[5][5];
	static pixel_t buffer_Blue[5][5];
	#pragma HLS ARRAY_PARTITION variable=buffer_Red complete dim=0
	#pragma HLS ARRAY_PARTITION variable=buffer_Green complete dim=0
	#pragma HLS ARRAY_PARTITION variable=buffer_Blue complete dim=0
	// Minden dimenzió mentén partícionáljon

	static col_t fiveLineX = 0;
	static row_t fiveLineY = 0;

	// Új sor jel
	if (newLine)
	{
		fiveLineX = 0;
		if (fiveLineY == 4)
			fiveLineY = 0;
		else
			fiveLineY++;
	}
	else
		fiveLineX++;

	fiveLine[RED][fiveLineX][fiveLineY] = pixelIn[RED];
	fiveLine[GREEN][fiveLineX][fiveLineY] = pixelIn[GREEN];
	fiveLine[BLUE][fiveLineX][fiveLineY] = pixelIn[BLUE];

// A szűrőablaka tartalmának balra shiftelése
shiftY: for (median_t medianY = 0; medianY < 5; medianY++)
		{
			#pragma HLS UNROLL
			buffer_shift_x:
			for (median_t medianX = 1; medianX < 5; medianX++)
			{
				#pragma HLS PIPELINE II=1
				#pragma HLS UNROLL
				bufferOrig_Red[medianX-1][medianY] =
				bufferOrig_Red[medianX][medianY] ;

				bufferOrig_Green[medianX-1][medianY] =
				bufferOrig_Green[medianX][medianY] ;

				bufferOrig_Blue[medianX-1][medianY] =
				bufferOrig_Blue[medianX][medianY] ;
			}
		}

// Új elem behelyezése
newPixel:	for (median_t medianY = 0; medianY < 5; medianY++)
			{
```

```
                        #pragma HLS UNROLL
                        bufferOrig_Red[4][medianY] =
                        fiveLine[RED][fiveLineX][medianY];

                        bufferOrig_Green[4][medianY] =
                        fiveLine[GREEN][fiveLineX][medianY];

                        bufferOrig_Blue[4][medianY] =
                        fiveLine[BLUE][fiveLineX][medianY];
                  }

// Buffer tartalmának a lemásolása
bufferCopyY: for (median_t medianY = 0; medianY < 5; medianY++)
            {
                  #pragma HLS UNROLL
                  bufferCopyX:
                  for (median_t medianX = 0; medianX < 5; medianX++)
                  {
                        #pragma HLS UNROLL
                        buffer_Red[medianX][medianY] =
                        bufferOrig_Red[medianX][medianY];

                        buffer_Green[medianX][medianY] =
                        bufferOrig_Green[medianX][medianY];

                        buffer_Blue[medianX][medianY] =
                        bufferOrig_Blue[medianX][medianY];
                  }
            }

            mergeSort((pixel_t*)buffer_Red);
            mergeSort((pixel_t*)buffer_Green);
            mergeSort((pixel_t*)buffer_Blue);

      // Kimeneti pixel beállítása
      pixelOut[RED] = buffer_Red[MEDIAN][MEDIAN];
      pixelOut[GREEN] = buffer_Green[MEDIAN][MEDIAN];
      pixelOut[BLUE] = buffer_Blue[MEDIAN][MEDIAN];
}
```

Késleltetések, igénybe vett erőforrások

Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	6.60	5.152	0.82

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
10	10	2	2	function

Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	8655
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	15	-	0	0
Multiplexer	-	-	-	255
Register	-	-	3126	-
Total	15	0	3126	8910
Available	270	240	84400	42200
Utilization (%)	5	0	3	21