Inter-Pixel Filtering of Digital Images With OpenCL on NVIDIA GPUs

José Valero
Francisco José de Caldas District University
Spatial Data Research Group (NIDE)
Engineering Faculty
110231, Bogotá, Colombia
jvalero@udistrital.edu.co

ABSTRACT

The process of filtering digital images represented by complex Cartesian allows to use the available one-dimensional (1D) elements (interpixel); however, having those additional 1D elements increases both the volume of data and the time for processing them. The time reduction strategy based on a parallel computing scheme on the number of available central processing units (CPUs) does not consider additional computing resources such as those offered by general purpose graphics processing units (GPUs) of NVIDIA. Parallel computing possibilities provided by the NVIDIA GPUs were explored and, based on them, a computational scheme for the digital image Cartesian complexes filtering task was proposed using the application program interface Open Computing Language (OpenCL) provided for NVIDIA corporation GPUs. The results assessment was established by comparing the response times of the proposed solution compared to those obtained using only CPU resources. The obtained implementation is an alternative to parallelization of the filtering task, which provides response times up to 14 times faster than those obtained with the implementation that uses only the CPU resource. The NVIDIA multicore GPU significantly improves the parallelism, which can be used in conjunction with the available multicore CPU computing capacity, balancing the workload between these two computing powers using both simultaneously.

Keywords

Cartesian complex, digital image, filtering, NVIDIA, OpenCL.

1 INTRODUCTION

The increasing availability of high geometric, radiometric, spectral and temporal resolution multispectral digital images of the earth surface increases the possibility of using them in space planning and management applications geographical. Such applications require object detection capabilities for monitoring based on edge detection procedures very frequently supported in the outcomes of the application of filters for edge detection, which, given the high volumes of data to be processed, demand huge computing resources. So, computational implementations, which are focused on reducing response times, are required.

In recent decades, specific purpose parallel computing capabilities for graphics processing have evolved, giving rise to graphical processing units (GPUs), which can be leveraged for other processing, when conveniently using application programming interfaces (APIs) they provide [1]. Efforts have been made to provide strategies and environments that facilitate programming to enhance GPU-based software development.

In [2] an introduction to modern PC architectures is given and discusses strategies and guidelines for developing GPU programs. [3] introduces a graphical user interface (GUI) tool called GPUBlocks whose purpose is to facilitate parallel programming in multicore computer systems. The GPU is designed for high-speed graphic processing that is inherently parallel. The Open Computing Language (OpenCL) takes a platform independent simple model of data parallelism and incorporates it into a programming model [4]. The OpenCL language makes the GPU look the same as another programmable device, using an execution model defined in terms of device kernels and a host program. In this way, the executable is in a virtual platform independent architecture, The OpenCL framework. This framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. Backwards compatibility allows

that a device consume earlier versions of the OpenCL C and other programming languages [5].

[6] manages to design and implement an alternative method for segmenting multispectral images based on axiomatic locally finite spaces (ALFS) provided by Cartesian complexes, and which takes into account topological and geometric properties [7]. This alternative representation model provides a geometric space that complies with the digital topology T_0 free of topological ambiguities, on which a new way of segmenting image data is built. The proposed model is developed and implemented in such a way that required topological and geometric characteristics present in the combinatorial semi-spaces are transformed into combinatorial structures which encode them using their associated oriented matroid [8, 9, 10]. Having additional inter-pixel elements means that the volume of data increases considerably. In particular, the task of texture analysis demands a significant amount of computing resources. The strategy adopted in [6], based on a parallel computing scheme from the execution in as many groups of 1-cells as the number of central processing units (CPUs) available, significantly reduces the time of processing required to process the entire Cartesian complex. However, the size of the digital image makes up such amount of data to be processed that the response times, even with dozens of CPUs, are prolonged.

[11] examines parallel algorithms for performing image boundary detection as part of the segmentation process and performs deployment on highly parallel NVIDIA processors, achieving a contour detector that provides accuracy with an F-measure of 0.70. The runtime is reduced from 4 minutes to 1.8 seconds. The efficiency gains that are made there allow the detection of contours in much larger images and the algorithms are applicable to various approaches to image segmentation. [12] starts from the fact that nowadays it is increasingly common to detect changes in land use and coverage using multispectral images and that a large part of the available change detection (CD) methods available focus on pixel-based operations. Since the use of spectral-spatial techniques helps to improve accuracy results, but also implies a significant increase in process time, [12] used a GPU framework to make object-based CD from multitemporal hyperspectral images and achieved real-time execution with accelerations of up to 46.5 times with respect to an open multiprocessing (OpenMP) implementation.

This research proposes the implementation of the multi-kernel multi-scale steerable filtering task of digital images represented as Cartesian complexes, on graphical processing units (GPUs), using a parallel computational scheme in terms of the OpenCL API. In this way, the processing that is currently assigned to each CPU can be performed in parallel including also the processing cores available on the GPU. The evaluation of the results was established by comparing the response times of the proposed solution compared to those obtained using only CPU resources. The produced implementation is an alternative parallelization of the filtering task, which provides response times up to 26 times faster than those obtained with the proposed implementation in [6].

2 PARALLELIZATION SCHEMA

The possibilities of parallel computation provided by the GPUs were explored in the context of the multi-kernel multi-scale steerable filtering task proposed in [13] and reformulated in [6] in terms of Cartesian complex. Finally, a parallel computational scheme was proposed and the outcomes quality assessment was established by comparing the response times of the proposed solution against the response times obtained through the implementation proposed in [6].

2.1 RAM memory resource management scheme

Memory in OpenCL is divided into Host and Device Memories. The device memory is directly available to kernels executing on OpenCL devices. In fact, a OpenCL kernel through work-items can access data from private memory spaces, local memory that can be used to allocate variables that are shared by all work-items in a work-group. Additionally, all work-items in all work-groups running on any device within a context can access to any element of a memory object in both Global and Constant Memories [5]. Therefore, it is unavoidable to transfer data from the host memory space to the device memory for processing and from the device memory to the host memory to obtain the results.

In order to reduce the copying time, the OpenCL function call for non-blocking memory transfers between host-device are used, by submitting Memory commands to a command-queue. So, the waiting times of the data copying between device and host memories in non-blocking mode can concurrently be used to perform other tasks on the host side. Based on the above, the required global device memory must be fully allocated to copy to there the input data from the host memory and then passed to each device work-item the segments that it must copy to the shared local memory, process and , after getting the results, copy from the shared local memory to the global device memory. There are two sets of data that must be copied in this way: (i) the one-dimensional cells of the Cartesian complex space over which the filtering process is to be performed; and (ii) the set of multi-kernel and multi-scale steerable filters [6] that will be applied. The two data sets will remain in the device memory

shared by the threads within a block, which involves the intra-device copying between global and shared memories.

If GPU device provides support for concurrent copying, the transfer time between memory spaces becomes a load balancing parameter, being clear that whenever possible it should be performed simultaneously with data subset processing assigned to the threads available on the CPU. On the other hand, the use of the parallel computation processing available in the GPU can be incorporated into the processing of the functions of the multi-kernel and multi-scale oriented filtering task of Cartesian complexes based on at least two parallelization schemes: (i) considering each CORE of the GPU as an additional processing unit and extending to them the parallelism scheme proposed in [6], and (ii) creating as many GPU work-items as one-dimensional space cells present in the Cartesian complex corresponding to the image to be processed and leave parallelization management to the GPU. Regardless of the schema used, the code block that the kernel must execute must present as few branches as possible in order to increase the performance of the underlying work-items within a single work-group execute concurrently guaranteed to make independent progress in the presence of subgroups and device support [5]. Next, the algorithmic solution for each of the two parallelization schemes mentioned above will be presented.

2.2 Parallelism scheme considering each GPU core as a processing unit equivalent to a CPU core

One intended parallelism scheme using the highly massive capabilities provided by a GPU is to treat each GPU core as an equivalent CPU core. In parallelism scheme proposed in [6], the one-dimensional cells of the Cartesian complex space are distributed using the row prime path [14, pp. 233, 234], among the m available CPU cores (see Fig.1 left). What is proposed here is to make the distribution of the one-dimensional cells taking each one of the p GPU cores that are available, as an additional processing unit, for a total of $m+p$ processing units (see Fig.1 right). However, in the case of the GPU, the following should be taken into account: first, the necessary copies between the host and device memory resources must be made; and second, differentiated code bifurcations between work-items of the same work-group should be avoided [15].

In this parallelism scheme, there is a one-dimensional NDRange consisting of as many work-items as the number of cores available on the device (GPU). In order to avoid code divergence between work-items that are part of each work-group eventually released by the GPU, the number of one-dimensional cells assigned to each GPU core must be the same, so the expression for the size calculation of each group of one-dimensional cells proposed in [6, p. 98] should be rethought.

A balanced mapping is made between the CPU and GPU, with an initial distribution based on the proportion that the number of GPU cores represents of the total available processing units. So, the number pct_{GPU} of one-dimensional cells proportionally assigned to the GPU is set based on equation Equation (1), and number g_1 of one-dimensional cells assigned to each core based on Equation (2).

$$pct_{GPU} = ceil\left(r\frac{p}{m+p}\right)\times c\ , \tag{1}$$

where r and c are respectively the number of rows and columns in the image, while m and p are respectively the number of CPU cores and GPU cores. *ceil* means "the smallest integer larger than".

$$g_1 = \frac{pct_{GPU} - \left(pct_{GPU}\%p\right)}{p}\ , \tag{2}$$

where % represents the module operation. The number of cells assigned to the GPU n_{GPU} that guarantees the same number of cells for each of its threads is given by Equation (3).

$$n_{GPU} = g_1 \times p\ . \tag{3}$$

So, the number of one-dimensional cells assigned to the CPU n_{CPU} is set according to Equation (4).

$$n_{CPU} = (r\times c) - n_{GPU}\ . \tag{4}$$

Finally, the distribution scheme proposed in [6, p. 98] is applied to obtain the two group sizes applicable to the CPU cores according to Equations (5) and (6).

$$g_2 = \frac{n_{CPU} - \left(n_{CPU}\%m\right)}{m}\ . \tag{5}$$

$$g_3 = g_2 + 1\ . \tag{6}$$

Therefore, each of the p cores of GPU will get assigned g_1 one-dimensional cells, and there will be as many as $m-(n_{CPU}\%m)$ CPU cores with g_2 assigned one-dimensional cells and as many as $(n_{CPU}\%m)$ CPU cores with g_3 assigned one-dimensional cells.

2.3 Parallelism scheme creating as many GPU threads as cells assigned to the device

The second tested parallelism scheme, using the highly massive capabilities provided by a GPU, is to create as many GPU work-items as the number of

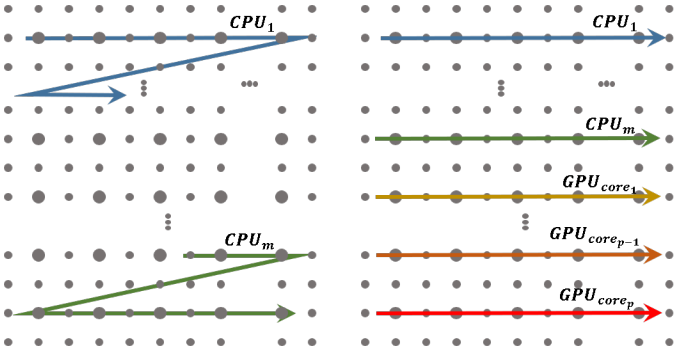

Figure 1: Only CPU versus GPU and CPU parallelism schemes.

two-dimensional cells assigned to it. This scheme uses the same group sizes used in section 2.2 by Equations (2), (5) and (6) but instead of creating p device work-items each with g_1 two-dimensional cells, $p \times g_1$ work-items are created in the device, each processing a cell of the Cartesian complex as shown in Figure 2 (bottom). As can be seen in the figure (above), the cells assigned to the CPU are distributed among the available cores using the scheme proposed in [6, p. 98].

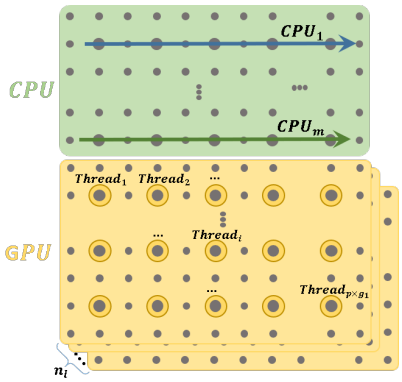

Figure 2: One GPU thread by two-dimensional cell parallelism scheme.

As in the scheme proposed in section 2.2, in the case of the GPU, the following should be considered: first, that the necessary copies between the host and device memory resources must be made; and second, that differentiated code bifurcations between work-items of the same work-group should be avoided. In this parallel scheme, a three-dimensional NDRange whose first and second dimension sizes correspond respectively with the number of rows and columns of the two-dimensional cells set assigned to the GPU, and the third dimension corresponds to n_l layers in the digital image to be processed. In this case the divergence of the code between the work-items that are part of each work-group, is counteracted by assigning to each work-item only one cell of the Cartesian complex in each layer of the digital image. Since the position of the two-dimensional cell assigned to a work-item is determined by the position of the work-item within the hierarchy of work-items, the number of cells must be a multiple of the number of columns in the image. By the above, in this case the number of two-dimensional cells assigned to the GPU is defined based on Equation 1 but using Equation 7 instead of Equation 3.

$$n_{GPU} = ceil\left(r \times c \frac{p}{m+p}\right) . \quad (7)$$

3 PARALLELISM MODEL IMPLEMENTATION

For performance reasons, the parallelization model of the multi-scale, multi-kernel steerable filtering process proposed in Section 2 using the OpenCL API, was adapted to Nvidia's CUDA programming framework and implemented using the integrated development environment (IDE) provided by Eclipse [16], for the Java and C/C++ programming languages. The Java language was used to include in the implementation proposed in [6] the copying of the host memory to that of the device (and vice versa) and for the configuration of the grid of threads blocks that are launched in the GPU using the respective support provided by the JCuda API version 10.1 [17]. The C / C ++ language was used to implement the kernel code that is executed by each thread. Next, a description of these components is made.

3.1 Host - device memory copying and thread block grid configuration

The implementation of the host - device memory copy scheme proposed in Section 2.1, included two phases: (i) the first phase includes copying the input data from the host memory to that of the device before launching the Thread block grid and the final copy of the results from the device memory to the host; and (ii) the second phase comprises the copying of the input data from the device memory to the memory shared by the threads of each block, as well as the final copying of the results from the memory shared to the device memory.

Copying between host and device memory was carried out concurrently for the four data sets: (i) the Cartesian input complex, (ii) the filter bank to be applied, (iii) the

arrangement with the number of 2-cells (pixels) in each filtering kernel, and (iv) the resulting Cartesian complex. This was implemented as shown in the code fragments of the filtering process shown in the Java program listing 1.

Because the JNI API used by JCuda to pass the parameters in the components implemented in Java and C programming languages only allows the use of one-dimensional arrays, it was necessary to convert the input data sets to a representation of one-dimensional primitive data arrays and vice versa (lines 8 and 11). To carry out this copying process, the asynchronous memory copy support provided by the GPU was used. The asynchronous copying process began by creating a stream which was assigned in the creation of the page-locked memory segments of the host and device (lines 12, 14, 17, 19, 21, 24 and 31), in asynchronous copying from conventional memory to host page-locked memory (lines 13 and 15), from host memory to device memory (lines 18 and 20), as in those copied from the memory of the device to that of the host (line 32). To ensure that the asynchronous copying process was carried out consistently, before proceeding to a next phase, synchronization points were included (lines 16, 24, 30 and 33).

Copying to the shared memory segments was configured using the support provided by JCuda in launching the thread block grid (lines 26 to 29), defining the size of the memory segment shared by the threads within a block (line 28) and copying in it the input data for each GPU thread. The kernels that must be executed by each GPU thread, according to the massively concurrent processing schemes proposed in Sections 2.2 and 2.3, were implemented in C programming language in the way described below.

3.2 Kernel code implementation considering each GPU core as an additional processing unit

In the GPU Kernel listing 2, shown below, presents the GPU kernel that must be executed by each thread for the parallelism scheme defined in Section 2.2.

Each kernel starts by defining the memory segment shared by the threads that are members of a block (line 27), in which the input data that will be used in the filtering process is placed (line 30), seeking to use the memory management that guarantees the greatest efficiency. In contrast to the filtering process implemented in [6], it acts not on two-dimensional but on one-dimensional data structures, so that their indexing had to be adapted (lines 5, 6, 13, 23 and 34). On the other hand, in order to avoid the divergence of threads warps, the "if" sentences were completely replaced by logical expressions that evaluate to 1 or 0 used to adequately activate addends or factors in accumulation expressions (line 14). These logical expressions were also used to activate / deactivate displacements in indexing expressions (lines 14 and 23). In this way, the only flow control statement used is “for”, but assigning to each GPU thread the same number of one-dimensional cells, which guarantees that the number of cycles in each thread that is part of A warp is exactly the same.

3.3 Kernel code implementation for a GPU thread for each two-dimensional cell

In the GPU Kernel listing 3, shown below, presents the GPU kernel that must be executed by each thread for the parallelism scheme defined in Section 2.3.

Each kernel starts by defining the memory segment shared by the threads in a block (lines 19 and 21), in which the input data that will be used in the filtering process is placed (lines 25 and 26), looking for use memory management that guarantees the highest efficiency. As in the parallelism scheme implemented in Section 3.2, it acts not on two-dimensional data structures but on one-dimensional data, so, its indexing had to be adapted (lines 3, 4, 8, 9 and 10). On the other hand, with the purpose of avoiding the divergence of thread warps, the “if” sentences were completely replaced by logical expressions that evaluate values 1 or 0 used to adequately activate addends or factors in accumulation expressions (line 10); as well as displacements in indexing expressions (line 15). Additionally, the multidimensionality of the GPU threads hierarchy was used, using the thread X dimension (threadIdx.x) and thread blocks (blockIdx.x) to access the 2-cells of the input Cartesian complex, the Y dimension of the thread block grid (blockIdx.y) to access each filtering kernel, and the Z dimension of the thread block grid (blockIdx.z) to Access each input channel. This allowed the removal of the respective “for” cycles of the DirectionalFilterProcessor, computeFilteredCell and applyFilter functions, leaving only one “for” cycle in the applyFilter function to control the filtering kernel neighborhood.

Thus, unlike the parallelism scheme implemented in Section3.2, the only flow control statement used is a “for” cycle which is evidenced in the algorithmic simplicity of the implementation presented in the GPU Kernel listing 3 regarding the one presented in the listing 2. The “if” flow control statement used on line 39 does not introduce significant divergence of threads as it is used exclusively to avoid the process of threads that are not assigned 2-cells.

4 RESULTS AND DISCUSSION

The implementation described in Section 3, was executed on a machine with OpenSUSE linux Leap 15.1

Listing 1: Java copying code snippet for FilteredCartesianComplex method

```
public FilteredCartesianComplex(...) {
  int cores = Runtime.getRuntime().availableProcessors();
  // Enable exceptions and omit all subsequent error checks
  JCudaDriver.setExceptionsEnabled(true);
  String ptxFileName = preparePtxFile("DirectionalFilterProcessorByCore.cu");
  cuInit(0); CUdevice device = new CUdevice();
  ...
  inputImage = aCartesianComplex.convert2PrimitiveArray(rows4GPU);
  hostInputImage = new Memory(MemoryType.PAGE_LOCKED,...,stream);
  hostInputImage.put(inputImage);
  float inputFilterBank[] = convert2PrimitiveArray(nFilterKernel, ...);
  Memory hostInputFilterBank = new Memory(MemoryType.PAGE_LOCKED, ...);
  hostInputFilterBank.put(inputFilterBank);
  Memory hostFilterKernelsCellCount = new Memory(MemoryType.PAGE_LOCKED,...);
  hostFilterKernelsCellCount.put(filterKernelsCellCount);
  cuStreamSynchronize(stream);
  deviceInputImage = new Memory(MemoryType.DEVICE, sizeOf2CellValues,stream);
  cuMemcpyHtoDAsync(deviceInputImage , hostInputImage, ...);
  deviceFilterBank = new Memory(MemoryType.DEVICE, ..., stream);
  cuMemcpyHtoDAsync(deviceFilterBank, ..., stream);
  devFilterKernelsCellCount = new Memory(MemoryType.DEVICE, ..., stream);
  cuMemcpyHtoDAsync(devFilterKernelsCellCount, ..., stream);
  deviceOutputImage = new Memory(MemoryType.DEVICE, ..., stream);
  cuStreamSynchronize(stream);
  ...
  cuLaunchKernel(function,1/gridSizeX/, 1, 1, // Grid dimension
   blockSizeX , 1, 1, // Block dimension
   inputFilterBank.length * Sizeof.FLOAT, null, // Shared memory size and stream
   kernelParameters, null); // Kernel- and extra parameters
  cuCtxSynchronize();
  Memory hostOutputImage = new Memory(MemoryType.PAGE_LOCKED, ..., stream);
  cuMemcpyDtoHAsync(hostOutputImage, deviceOutputImage, ..., stream);
  cuStreamSynchronize(stream);
        hostOutputImage.get(outputImage);
  cuStreamSynchronize(stream);
 ...
}
```

operating system, with *AMD FX*™ − 8320 processor with 8 cores, RAM memory (host memory) of 16 *GiB* and a GPU device with the configuration features shown in Table 1. With the purpose of dedicating the GPU to the filtering process only, the machine was equipped with a second ATI Radeon 3000 video card for the management of the Graphic interface.

For the multi-scale multi-kernel steerable filtering process it was used a 500 by 500 pixel image. The implementation execution was monitored using NVIDIA Nsight Systems software version 2019.5.2, taking the events of creation and destruction of the GPU context, respectively as the start and end of the filtering process. The monitoring the execution of the filtering process using only the 8 CPU cores showed that (Figure is not shown for reasons of space), the filtering process starts at 1.48594 seconds, after the end of the CUDA context creation event. The filtering process ends at 2.18768 seconds, before the start of the CUDA context destruc-

Nvidia Device	GeForce GTX 950
CUDA Driver Version	10.1
CUDA Capability	5.2
Total global memory	2002 MBytes
Multiprocessors (MP)	6
CUDA Cores / MP	128 (768)
Total shared memory	49152 bytes (per block)
Warp size	32
Max block size(x, y, z)	(1024, 1024, 64)
Max grid size (x, y, z)	(2147483647,65535,65535)
Concurrent copy and kernel execution	Yes, 2 copy engine (s)

Table 1: GPU hardware configuration of the test environment.

Listing 2: process that employs one GPU core per 2-cell group

```
extern "C" _device_ void applyFilter(int central1DCoord, ...) {
 int aCoord1D = 0;

 for (int channel = 0 ; channel < numberOfInputChannels ; channel++) {
  int inputChannelCentral1DCoord = channel * numberOfInputCellTypes * pixelsInX *
    pixelsInY + central1DCoord;
  int outputChannelCentral1DCoord = ((kernelIndex * numberOfInputChannels +
    channel) *
  numberOfOutputCellTypes + cellType) * pixelsInX * pixelsInY + central1DCoord;
  aFilteredCC[outputChannelCentral1DCoord] = 0;

  for (int cellInd = 0 ; cellInd < kernelCellCount ; cellInd++) {
   int cX = (aFilterKernel[cellInd * 3] - 1 + cellType) / 2,
    cY = (aFilterKernel[cellInd * 3 + 1] - cellType) / 2;
   aCoord1D = central1DCoord + (cY * pixelsInX) + cX;
   aFilteredCC[outputChannelCentral1DCoord] += ((1 && (aCoord1D >= 0) && (aCoord1D
    < (pixelsInY * pixelsInX))) * (aMonochromatic[inputChannelCentral1DCoord + (1
    && (aCoord1D >= 0) && (aCoord1D < (pixelsInY * pixelsInX))) * ((cY * pixelsInX)
    + cX)] * aFilterKernel[cellInd * 3 + 2]));}
 }
}
extern "C" _device_ void computeFilteredCell(int  central1DCoord, ...) {
 float * dF = nFilterKernel + cellType * filterKernelsCellCount[0] * 3;

 for(int iFK= cellType ; iFK < numberOfKernels ; iFK += 2){
  applyFilter(central1DCoord, ...);
  dF = dF + (filterKernelsCellCount[iFK] + filterKernelsCellCount[(iFK + 1) %
    numberOfKernels]) * 3;}
}
extern "C" _global_ void DirectionalFilterProcessor(...){
 extern _shared_ float interchangedData[];
 float * sharedFilterBank = interchangedData;
 int groupStart =  threadIdx.x * groupSize;
 if (threadIdx.x == 0) memcpy(sharedFilterBank, nFilterKernel, sizeof(float) *
    bankSize);
 __syncthreads();

 for(int i = 0; i < groupSize; i++){
  int central1DCoord = groupStart + i;
  computeFilteredCell(central1DCoord, HORIZONTAL_CRACK, ...);
  computeFilteredCell(central1DCoord, VERTICAL_CRACK, ...);}
}
```

tion event. That is, the filtering process performed concurrently only by the eight CPU cores, each processing 31250 or 31251 pixels according to Equations 5 and 6, took a time of 701.74 milliseconds.

On the other hand, Figure 3 presents the results of the monitoring of the filtering process execution using, in addition to the 8 CPU cores, also the 768 GPU cores. In this case, of the $500 \times 500 = 250,000$ pixels, $247,296$ were assigned to the GPU, that is, 768 groups (one for each GPU core), each of 322 pixels. As shown in the figure, the filtering process starts at 1.49453 seconds, after the end of the CUDA context creation event. As can also be seen, the filtering process ends at 1.64496 seconds, before the start of the CUDA context destruction event (labeled in the figure as call to cuCtxDestroy). Therefore, the filtering process performed concurrently by the eight CPU cores, each processing 338 or 339 pixels according to equations Equations 5 and 6, took a time of 150.43 milliseconds. In the case of GPU threads, the filtering process performed concurrently by processing each 322 pixels, started at 1.52946 seconds and finished 1.58076 (labeled in the figure as DirectionalFilterProcessor), that is, it took 51.3 milliseconds. This means that, using all available CPU and GPU computing resources, the process performed by the CPU

Listing 3: process that employs one GPU thread per 2-cell

```
extern "C" _device_ void applyFilter(int central1DCoord, ...) {
  int aCoord1D = 0;
  int inputChannelCentral1DCoord = channel * numberOfInputCellTypes * pixelsInX *
    pixelsInY + central1DCoord;
  int outputChannelCentral1DCoord = ((kernelIndex * numberOfInputChannels +
    channel) * numberOfOutputCellTypes + cellType) * pixelsInX * pixelsInY +
    central1DCoord;
  aFilteredCC[outputChannelCentral1DCoord] = 0;

  for (int cellInd = 0; cellInd < kernelCellCount; cellInd++) {
    int cX = (aFilterKernel[cellInd * 3] - 1 + cellType) / 2, cY =
    (aFilterKernel[cellInd * 3 + 1] - cellType) / 2;
    aCoord1D = central1DCoord + (cY * pixelsInX) + cX;
    aFilteredCC[outputChannelCentral1DCoord] += ((1 && (aCoord1D >= 0) && (aCoord1D
    < (pixelsInY * pixelsInX))) *  (aMonochromatic[inputChannelCentral1DCoord + (1
    && (aCoord1D >= 0) && (aCoord1D < (pixelsInY * pixelsInX))) * ((cY * pixelsInX)
    + cX)] * aFilterKernel[cellInd * 3 + 2]));}
}
extern "C" _device_ void computeFilteredCell(int  central1DCoord, ...) {
  applyFilter(central1DCoord, ...,
    nFilterKernel + filterKernelsCellCount[2 * (2 * iFK + cellType) + 1],
    filterKernelsCellCount[2 * (2 * iFK + cellType)], pixelsInX, pixelsInY,
    channel);
}
extern "C" _global_ void DirectionalFilterProcessor(...){
  extern _shared_ float interchangedData[];
  float * sharedFilterBank = interchangedData;
  extern _shared_ float sharedFilterKernelsCellCount[numberOfKernels * 4];
  int central1DCoord = blockIdx.x * blockDim.x + threadIdx.x;

  if (threadIdx.x == 0){
    memcpy(sharedFilterBank, nFilterKernel, sizeof(float) * bankSize);
    memcpy(sharedFilterKernelsCellCount, devFilterKernelsCellCount, sizeof(float) *
    numberOfKernels * 4);}

  __syncthreads();

  if (central1DCoord < (pixelsInX * pixelsInY)){
   computeFilteredCell(central1DCoord,HORIZONTAL_CRACK, blockIdx.z,blockIdx.y,...);
   computeFilteredCell(central1DCoord,VERTICAL_CRACK, blockIdx.z,blockIdx.y,...);}
}
```

cores was $\frac{701.74}{150.43} \cong 5$ times faster, while the process performed by the GPU cores was $\frac{701.74}{51.3} \cong 14$ times faster.

On the other hand, Figure 4 presents the results of the monitoring of the filtering process execution using all computing resources, the 8 CPU cores and the 768 GPU cores. In this case, the GPU was also assigned 247,296 2-cells (pixels), however, a GPU thread was created for every 2-cell of each channel in the Cartesian input complex and for each filtering kernel. This is evidenced in Figure 4 in which can be seen the hierarchy of threads composed of a grid of threads blocks of $242 \times 2 \times 1$ and each grid of threads within each block of $1024 \times 1 \times 1$. As shown in the figure, the filtering process starts at 1.37085 seconds and ends at 1.52964. Therefore, the filtering process performed concurrently by the eight CPU cores, each processing 338 or 339 pixels, took a time of 158.79 milliseconds. In the case of GPU threads, the filtering process carried out concurrently by processing each one an input channel and a kernel of the two 1-cells of the pixel assigned to it, started at 1.40355 seconds and ended 1.43087, that is, it took 27.32 milliseconds.

This means that, using all available CPU and GPU computing resources, the process performed by the CPU cores was $\frac{701.74}{158.79} \cong 4$ times faster, while the one performed by GPU cores were $\frac{701.74}{27.32} \cong 26$ times faster.

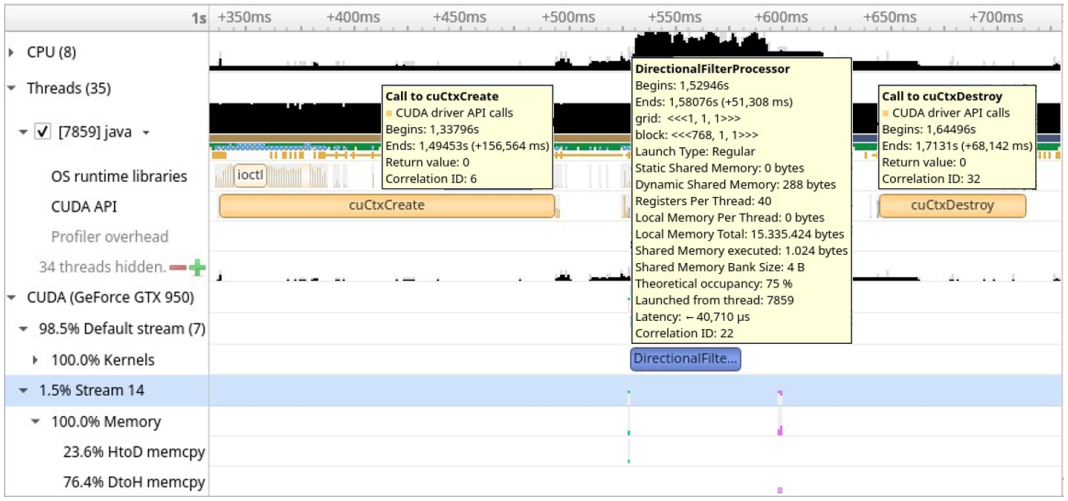

Figure 3: Monitoring of the filtering process using 8 CPU and 768 GPU cores.

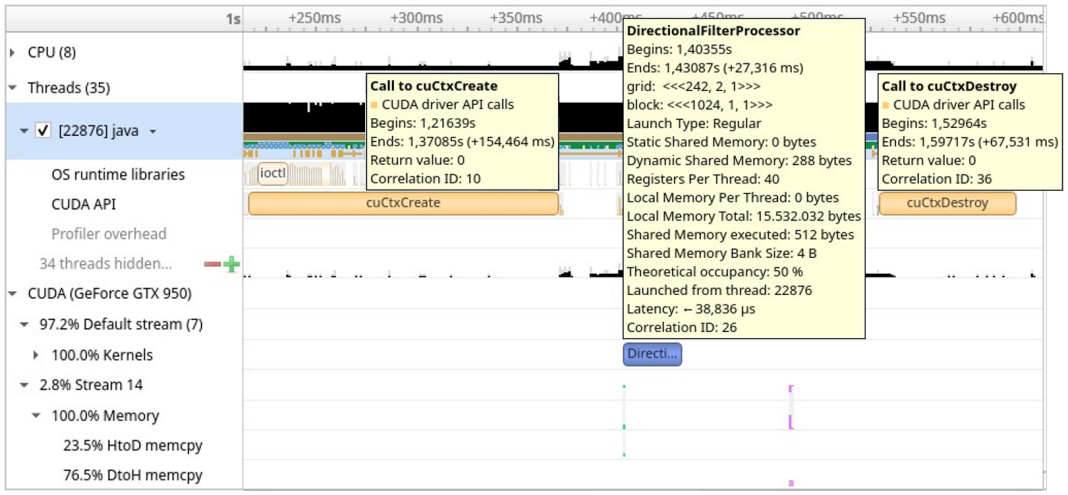

Figure 4: Monitoring of the filtering process using a GPU thread for every 2-cell.

The adopted hierarchy of memory scheme, which is based on the use of the device's global memory to copy there both the input image represented by the Cartesian complex, and the filter bank to be applied to each 1-Cell and then copy it to the memory shared by each block of threads, involves three phases of memory transfer, including the final copying of the results. The processing time is significantly reduced by performing it in parallel using the asynchronous copy scheme provided by the CUDA API, as well as assigning to each of the threads blocks the copying in the shared memory segments, the data to be processed. This means that, although it is true, additional time is devoted to copy processing, the overall performance is also increased when processing on the cells of the Cartesian complex, since they are placed in shared memory of the device, which has the largest bandwidth.

The massive parallelization schemes presented in Section 2, which were introduced using the Java programming language and the CUDA implementation for Java called Jcuda [17], as well as the C programming language for the case of the processing kernels to be executed by a GPU thread, show at least two ways of performing the multi-scale and multi-kernel filtering process of an image represented as a Cartesian complex, by distributing the cell processing among GPU and CPU resources highly massive available.

5 CONCLUSIONS AND FUTURE WORK

The proposed scheme for multi-scale and multi-kernel oriented filtering of an image represented as a Cartesian complex implies additional processing times dedicated to memory copying and conversion of the Cartesian complex to one-dimensional primitive arrays. However, since these processes are carried out in parallel, their impact on overall performance is negligible compared to the performance gain when performing the filtering process using the highly massive processing offered by GPU devices.

While it is true, the parallelism capability offered by a multi-core CPU is greatly exceeded by that offered by

a multi-core GPU, the proposed parallelism approach manages to perform a proportional balance between these two parallel computing capabilities using both in a concurrent way. Although in general the frequency of a CPU may be slightly higher than that of a GPU, the data orientation of the latter makes it more appropriate for the processing of high volumes of data such as digital images represented as Cartesian complexes.

The proposed approach constitutes a prototype and therefore presents several limitations that must be taken into account and that would warrant the continuation of this research. On the one hand, the proposed processing contemplates the copying of the input Cartesian complex, represented as a one-dimensional primitive arrangement, to the global and shared memories of the device. However, these resources are limited, so that the size of the image that can be processed is quite restricted for practical purposes. In order to be applicable to images regardless of their size, a research should be undertaken that extends the copying component to and from the global and shared memories of the device, by partitioning the image into fragments that can be housed in these memory resources.

The approach proposed in [6, pp. 94-97] supports the types of data that the input image presents through the use of the GDAL layer ([18]); however, the approach proposed here only supports the float data type, so a research should be undertaken that addresses the types supported by GDAL layer.

6 ACKNOWLEDGMENTS

Our thanks to the District University for allowing work time to carry out the research.

7 REFERENCES

[1] T. Soyata, *GPU Parallel Program Development Using CUDA*. CRC Press, Inc., 01 2018.

[2] A. R. Brodtkorb, T. R. Hagen, C. Schulz, and G. Hasle, "Gpu computing in discrete optimization. part i: Introduction to the gpu," *EURO Journal on Transportation and Logistics*, vol. 2, p. 129, May 2013.

[3] Y.-S. Hwang, H.-H. Lin, S.-H. Pai, and C.-H. Tu, "Gpublocks: Gui programming tool for cuda and opencl," *Journal of Signal Processing Systems*, p. 1, July 2018.

[4] T. Odaker, D. Kranzlmueller, and J. Volkert, "Gpu-accelerated triangle mesh simplification using parallel vertex removal," *International Journal of Computer and Information Engineering*, vol. 10, no. 1, pp. 160 – 166, 2016.

[5] Khronos OpenCL Working Group, *The OpenCL Specification, Version Version V2.2-11*, 2019.

[6] J. Valero, *Development of an Alternative Method for Multispectral Image Segmentation based on Cartesian Complexes and their associated Oriented Matroids*. phdthesis, Universidad Distrital Francisco Jose de Caldas, Feb. 2019.

[7] V. A. Kovalevsky, "Geometry of locally finite spaces," *International Journal of Shape Modeling*, vol. 14, no. 02, pp. 231–232, 2008.

[8] H. Whitney, "On the abstract properties of linear dependence," *American Journal of Mathematics*, vol. 57, pp. 509–533, 1935.

[9] J. G. Oxley, *Matroid Theory (Oxford Graduate Texts in Mathematics)*. New York, NY, USA: Oxford University Press, Inc., 2006.

[10] K. Fukuda, "Lecture notes on oriented matroids and geometric computation," tech. rep., 2004. RO-2004.0621, course of Doctoral school in Discrete System Optimization, EPFL 2004.

[11] B. Catanzaro, B. Su, N. Sundaram, Y. Lee, M. Murphy, and K. Keutzer, "Efficient, high-quality image contour detection," in *2009 IEEE 12th International Conference on Computer Vision*, pp. 2381–2388, Sept 2009.

[12] J. López-Fandiño, D. B. Heras, F. Argüello, and M. Dalla Mura, "Gpu framework for change detection in multitemporal hyperspectral images," *International Journal of Parallel Programming*, p. 1, Dec. 2017.

[13] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik, "Contour detection and hierarchical image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, pp. 898–916, May 2011.

[14] M. Worboys and M. Duckham, *GIS: A Computing Perspective, 2Nd Edition*. Boca Raton, FL, USA: CRC Press, Inc., 2004.

[15] NVIDIA, *NVIDIA CUDA C Programming Guide (Version 10.1)*, 2019.

[16] Eclipse, *The Platform for Open Innovation and Collaboration*. Eclipse foundation, version 2019-09 ed., Sept. 2109.

[17] M. Hutter, *Java bindings for CUDA*, 2019.

[18] GDAL Development Team, *GDAL - Geospatial Data Abstraction Library, Version 2.1.0*. Open Source Geospatial Foundation, 2016.