Computers & Geosciences 45 (2012) 212–220

Contents lists available at SciVerse ScienceDirect

Computers & Geosciences

journal homepage: www.elsevier.com/locate/cageo

Accelerating batch processing of spatial raster analysis using GPU

Mathias Steinbach*, Reinhard Hemmerling[1]

Chair for Computer Graphics and Ecological Informatics, University of Göttingen, Germany

A R T I C L E   I N F O

Article history:
Received 3 July 2011
Received in revised form
7 November 2011
Accepted 8 November 2011
Available online 17 November 2011

Keywords:
OpenCL
GPU
GIS
Raster analysis
Caching

A B S T R A C T

Batch processing of raster data performed by geographic information systems (GIS) is a time consuming procedure. Modern high performance GPUs are able to perform hundreds of arithmetical operations in parallel. These GPUs can help to reduce the computing time of such operations. In addition, most of the commonly used raster operations are I/O-bounded. Memory transfer between hard disk and RAM takes up more time than computations. The scope of this paper is to present an efficient two-level caching strategy for raster data and an acceleration of selected raster operations using the GPU, which were implemented as a plugin for the open source software GRASS. An example data flow based on a real world use-case will be presented and the obtainable and practically expectable speedup will be measured and discussed.

1. Introduction

Operations for spatial raster analysis are part of nearly every geographic information systems (GIS). Depending on the size of raster data, most of these operations are really fast. The user gets results within 2 s or less. However, in complex analyses, lots of raster operations are performed one after another via batch processing. This can lead to unacceptably long runtimes of some hours or more.

The reasons for this long runtime are twofold. First, a lot of GIS software only uses a single thread of execution to perform its computations and therefore does not make use of recent multi-core CPUs or even GPUs. Second, intermediate results are written to the hard disk and are reloaded for the next operation. Avoiding these redundant transfers will speed up the batch processing.

The contribution of this paper includes a way how to enable efficient caching of intermediate data as well as an acceleration of some existing algorithms by porting them to the GPU. The implementation is provided as a plugin for the open source GIS software *GRASS*. An example batch process is used to demonstrate how the extension can be used and to measure the acceleration one can expect from this approach.

2. Related work

GPUs have become a powerful tool for speeding up scientific computations. For instance, Viola et al. (2003), Yang et al. (2008), and Temizel et al. (2011) already used GPUs to accelerate image processing. Spatial raster data is also some kind of image. Thus, the GPU could also become a powerful tool for geoprocessing. Beutel et al. (2010) presented an GPU based algorithm to accelerate interpolations and Zhang et al. (2010) presented a data structure for handling huge raster files with GPU. Acceleration of geophysical simulations like Geofluids, Seismology and Magnetic force microscopy have been presented in Walsh et al. (2009) while in Ortega and Rueda (2010) GPUs were used to accelerate drainage networks determination. The authors could show that GPU-based simulations outperform CPU-based ones with a factor of eight and more. In Wu et al. (2007) a general framework for “Improving the performance of spatial raster analysis in GIS using GPU” was presented. They could show that GPU based algorithms are obviously faster than CPU based ones, but they did not implement their ideas in a real GIS. One of their main innovations was to reuse intermediate results without storing them on hard disk. Manifold presented the first GIS with GPU support. It accelerates some raster operations that are based on neighborhood analysis.

3. Our approach

Our goal is to analyze the possible speedup of batch processing of raster data. For this, we use the open source GIS *GRASS* (Neteler and Mitasova, 2008) as development platform for GPU based raster operations and as reference to compare computing times. We developed GRASS modules that use the computational power of a GPU. By accessing an OpenCL driver (OpenCL, 2010), GRASS modules are able to communicate with GPUs. OpenCL promises to be independent of computing device and operating system as long as there is an OpenCL driver available. In general, OpenCL

* Corresponding author. Tel.: +49 163 8269280.
E-mail addresses: mathias.steinbach@yahoo.com (M. Steinbach), rhemmer@gwdg.com (R. Hemmerling).
[1] Tel.: +49 551 393697.

0098-3004/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cageo.2011.11.012

algorithms are not bounded to GPUs. Also (multi-core) CPUs are able to execute them.

First, we parallelize some common used GRASS modules. Each of these GRASS modules is implemented as a new parallelized GRASS module that uses the GPU for computations. Every GRASS module runs as a single process. The only way to exchange data between modules is to store it on hard disk. To avoid disk access, a particular module was implemented that merges all parallelized algorithms and performs batch processing based on a provided BASIC script. This module allows to optimize the data exchange between GPU memory, RAM and hard disk. We then use the following example dataflow (shown in Fig. 3, its BASIC script in Listing 1) to measure the runtime of our implementation and to compare it with GRASS standard raster operations.

Listing 1–Dataflow as BASIC Script.

```
' declare variables
dim i as float
dim areas as raster, x as raster, y as raster
dim mask as raster, forest as raster, forest500
as raster, EDT as raster, avarage as Raster

' load areas of wood
areas = Load("areas_of_wood" )

' mask forest areas
mask = If(areas == empty, 0, 1 )

' store the results of neighbourhood analysis
Store(FocalSum(mask, 80, 100, 50), "N50" )
Store(FocalSum(mask, 80, 100, 25), "N25" )
Store(FocalSum(mask, 170, 190, 50), "W50" )
Store(FocalSum(mask, 170, 190, 25), "W25" )

' load maps with coordinates
x = Load("x-coordinate" )
y = Load("y-coordinate" )

i=0
repeat:
  ' set computational area to map areas
  SetRegion(areas, "rast" )

  ' select a forest
  forest = If(areas == i, 1, empty )

  ' resize forest
  EDT = EucDistance(forest )
  forest500=If(EDT<=500,1,empty)

  ' minimize computational region
  SetRegion(forest500, "zoom" )

  ' Mean forest500 coordinates
  mean = MaskMean(X, forest500 )
  Store(mean, "MeanX" + i )
  mean = MaskMean(Y, Wald500 )
  Store(mean, "MeanY" + i )

  i = i+1
ifi<884then
  goto repeat
endif
```

3.1. Example dataflow

The dataflow is based on a map containing 844 woodlands (see Fig. 1) with a size of 2275×2263 pixels corresponding to an area of 22.7 km times 22.6 km. It contains two major steps. A neighborhood analysis of the whole map counts the cells in a specific neighborhood that contain wood. A mean value analysis of each single woodland then computes a kind of center for each woodland.

We would not discuss these results, rather we want to focus on the runtime of this simple task. We cannot analyze every possible raster operation. Because of this, we chose a dataflow that contains commonly used standard operations. We describe these operations in the following.

The first step (neighborhood analysis) consists of counting the cells in a specific neighborhood around each cell that contains wood. ArcGIS offers some predefined shapes for the neighborhood. We use the shape 'wedge', because it has an unexpected long runtime. Fig. 2 shows an example. The brown cells contain wood and the white ones are empty. We want to count these wood cells and store the result in the marked cell. This analysis has to be done for each cell. The counting is obtained by creating a mask with a value of one for cells that contain wood and a value of zero otherwise. In GRASS the map algebra (function `r.mapcalc`) is used to get this result. ArcInfo provides this functionality with the AML command `IF`. Fig. 3(**2**) shows the result.

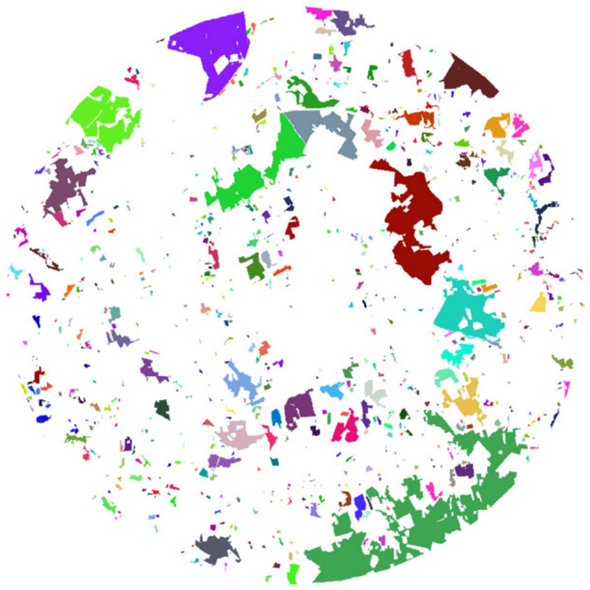

Fig. 1. Map with woodlands in Germany of 515 km^2, 5 million cells, every woodland has its own color.

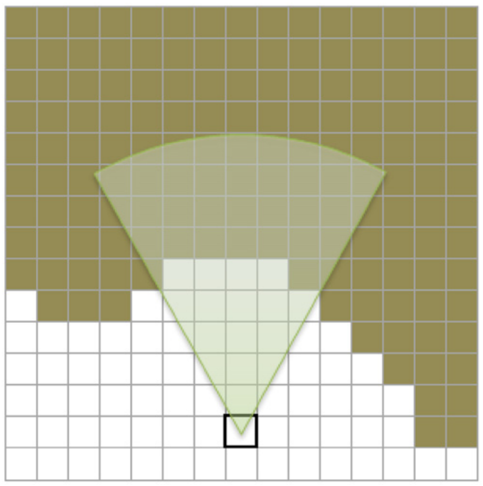

Fig. 2. Cells in a wedge neighborhood of one cell.

Now we can count the values of every cell in a wedge around this cell. We use the AML command `FocalSum` with different wedges as neighborhood. GRASS has not such a specific command, but it can be emulated with `r.mfilter` for small wedges. The four results are shown in Fig. 3(**2.1**–**2.4**). The first wedge is covered by the two angles 170 and 190° and a radius of 50 cells (500 m). It is directed to west.

The second step (mean value analysis) is performed for each woodland individually. Therefore, we have to pick a single woodland from the map by map algebra. Fig. 3(**1**) shows the selection of a particular woodland.

Then, for each cell, we compute the shortest distance from that cell to the current woodland. The resulting map is called *Euclidian distance transformation* (EDT) and shown in Fig. 3(**1.1**). The ArcInfo function `EucDistance` computes an exact EDT, while the GRASS function `r.grow.distances` computes only an approximation.

We use the EDT, to let the woodland grow by a radius of 500 m (50 cells). Therefore we choose all cells with values smaller then 500 and store them in a new result map using map algebra (see Fig. 3(**1.2**)).

To reduce the number of calculations, we shrink the computational area (ESRI names it *area of interest* (AOI)) to the grown woodland. Fig. 3(**1.3**) shows that the number of cells to compute decreases dramatically.

The AML command `ZonalMean` computes the mean of cells from a value map for each zone in a map with zones. We use the grown woodland as a sole zone and two maps, one with *x*- and the other with *y*-coordinates, as value maps. With these three maps, we perform `ZonalMean` twice. The results are shown in Fig. 3(**1.4** and **1.5**). The GRASS module `r.average` computes the same results. These two results describe a kind of center of the grown woodland. The centers of all grown woodlands are the results of the second step.

3.2. *Caching of intermediate files*

GIS applications produce intermediate data that are stored as images on hard disk. Because of reading/writing, packing/unpacking, converting (for instance coordinate systems) this takes up an nonnegligible amount of computing time. Effectively, before starting to accelerate individual operations by rewriting them for the GPU, one should first optimize the way such intermediate results are handled.

Of course, it is not possible to keep every raster of a whole batch script in main memory. Therefore, we developed a memory managing runtime environment. Based on a least recently used (LRU) strategy, the environment decides which variable (and therefore

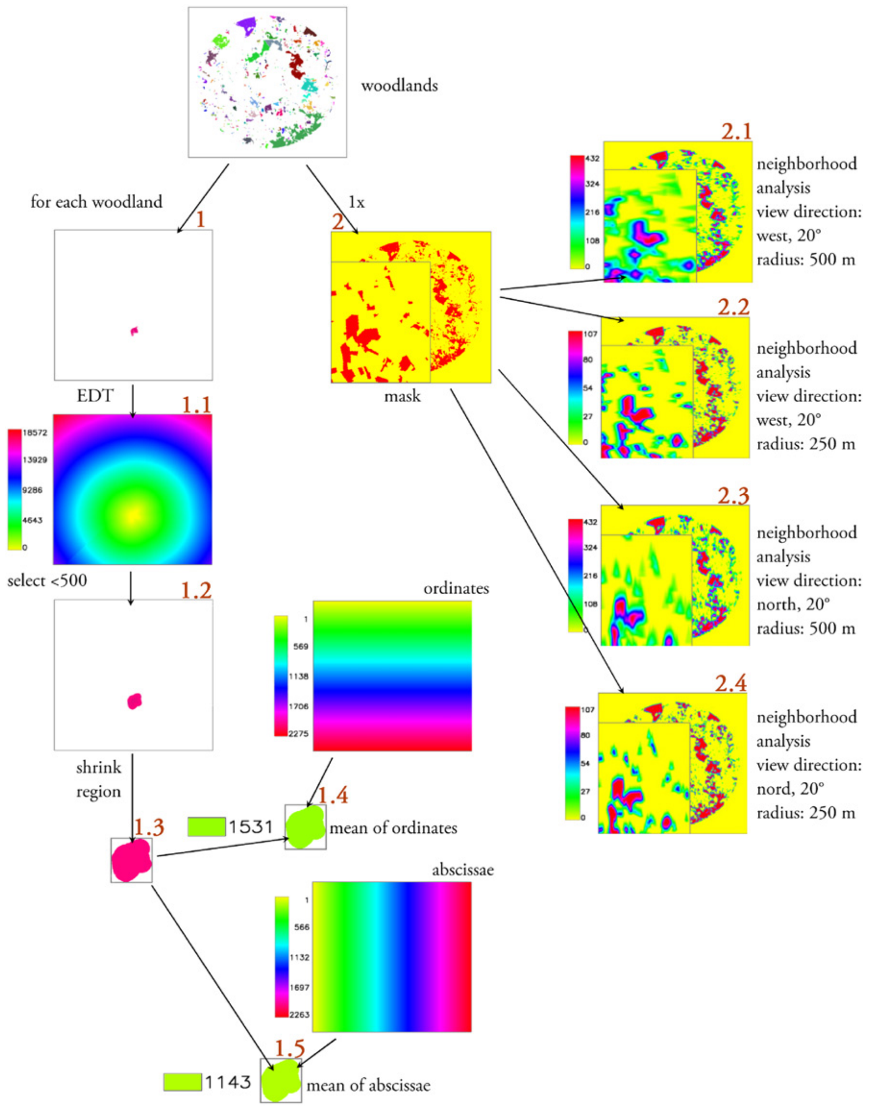

Fig. 3. Examplary dataflow.

raster) is hold in main memory and which is swapped out to hard disk. The variable that was not used for the longest period is swapped out. The LRU algorithm ensures that the result of the last operation is always in main memory and ready to use as input for the next operation. This is especially important for loops.

When using the GPU for computations, we have another stage in the memory hierarchy. The same LRU algorithm can be applied to keep as much data as possible inside the GPU memory and only swap to main memory on demand. Tables 1 and 2 list the transfer times for different raster sizes. The transfer speed between main memory and GPU memory is about 60 times faster than the transfer speed between hard disk and main memory. This comparison points out the need to avoid hard disk accesses.

Based on Aho et al. (2006), we developed a script compiler that generates code for the memory managing runtime environment. The compiler recognizes BASIC syntax. Listing 1 shows our example dataflow in that BASIC syntax.

Listing 2–Kernel code for adding two rasters.

```
// sum = left_addend + right_addend
__kernel void adding
(__write_only image2d_t sum,
__read_only image2d_t left_addend,
__read_only image2d_t right_addend,
   int2   left_addend_offset,
   int2   right_addend_offset
)
{
// select a raster cell by thread id
int2 current_cell = (int2)(get_global_id(0U),
                           get_global_id(1U));

// we use a block size of 16 × 16
// one multiprocessor computes 16 × 16 cells
// if output raster height and width are not
// an integer multiple of 16,
// the raster boundaries have to be checked
if(currentcell.x < getimagewidth(sum)&&
  currentcell.y < getimageheight(sum))
{

  // read a cell value from input rasters
  float left_cell_value =
    read_imagef(left_addend, sampler,
    current_cell + left_addend_offset).x;

  float right_cell_value =
    read_imagef(right_addend, sampler,
      current_cell + right_addend_offset).x;
  // compute result cell value
  float4 result = 0.0f;
  result.x = left_cell_value +
right_cell_value;

  // write result to output raster
  write_imagef(sum, current_cell, result);
}
}
```

Listing 3–Kernel code for computing bounding box around nonempty cells.

```
// after initializing the profiles with zero,
// the profiles are computed this way:
__kernel void compProfiles
(__write_only image2d_t profile_x,
__write_only image2d_t profile_y,
__read_only image2d_t input_raster
)
}
// select a raster cell by thread id
int2 cell = (int2)(get_global_id(0U),
                   get_global_id(1U));

// we use a block size of 16 × 16
// one multiprocessor computes 16 × 16 cells
// if output raster height and width are not
// an integer multiple of 16,
// the raster boundaries have to be checked
if(cell.x < getimagewidth(inputraster)&&
  cell.y < getimageheight(inputraster))
{

  // read current value from input raster
  float4 current_value =
  read_imagef(input_raster, sampler, cell);

  // if current value is not an empty cell,
// the profiles are filled
  if (! isnan(current_value.s0) )
  {
    int4 one=1;

  pos_x = (int2)(cell.x, 0)
  pos_y = (int2)(cell.y, 0)
    write_imagei(profile_x, pos_x, one);
    write_imagei(profile_y, pox_y, one);
  }
}
}

// computing the bounding box
__kernel void bounding_box
(__read_only image2d_t profile_x,
__read_only image2d_t profile_y,
__global  int  top,
__global  int  left,
__global  int  right,
__global  int  bottom,
)
{
int thread_id = get_global_id(0U);

int i;
int4 value;

// the first thread searches the left boundary
if (thread_id == 0)
{
  i=0;
  value = read_imagei(profile_x,
  sampler, (int2)(i,0));

  while (value.s0 == 0 &&
  i < getimagewidth(profilex))
  {
    i++;
    value = read_imagei(profile_x,
  sampler, (int2)(i,0));
  }

  left=i;
}

// the second thread searches the right boundary
if (thread_id == 1)
{
  // ...
}

// the third and fourth threads search the top
// and bottom boundaries in profile_y
// ...
}
```

3.3. GPU processing

Modern high performance consumer GPUs like NVIDA GeForce 580 or ATI Radeon HD 6870 provide up to 500 computing units. That means, they are able compute up to 500 raster cells at the same time. In consideration of multimedia commands, modern CPUs are only able to compute about 30 cells at the same time. Every computing unit of these GPUs performs the same algorithm in parallel, called a kernel. The programmer just has to define how many instances of this algorithm should be computed. Those instances are called threads and each of them has an own ID. An algorithm can use this ID to identify the address space of input and output parameters. We use two-dimensional thread IDs to identify the input and output cells of raster operations.

3.3.1. Raster algebra

The raster algebra is a local operation that performs standard math operations like addition, subtraction, multiplication, division and comparisons on rasters. The value of an output raster cell depends only on the input raster cells at the same location. A simple example is to add a constant value to every cell of a raster. Another possibility is to compare two rasters.

Table 1
Data transfer time between hard disk and main memory.

Map size	Read from hard disk (s)	Write to hard disk (s)
1138 × 1132	0.15	0.33
2275 × 2263	0.32	0.80
4550 × 4526	0.95	2.50

Table 2
Data transfer time between GPU memory and main memory.

Map size	Read from GPU memory (ms)	Write to GPU memory (ms)
1138 × 1132	4.4	3
2275 × 2263	8.2	9
4550 × 4526	34.0	33

Let $M_{x,y}$ and $N_{x,y}$ denote the cells with column x and row y of two rasters M and N. The result $R = M < N$ is then computed by

$$R_{x,y} = \begin{cases} 1 & \text{if } M_{x,y} < N_{x,y}, \\ 0 & \text{otherwise.} \end{cases}$$

The GPU algorithm would look like this

$$Result_{ID_x,ID_y} = M_{ID_x,ID_y} < N_{ID_x,ID_y},$$

where ID is a two dimensional thread ID. The loops over the rows and columns are given implicitly by the number of threads and are performed by the GPU scheduler.

It is possible that there is an offset between input and output raster or also between several input rasters. In general, we define an offset $(\Delta x,\Delta y)$ for every input raster. The GPU algorithm then becomes

$$Result_{ID_x,ID_y} = M_{ID_x+\Delta x_M,ID_y+\Delta y_M} < N_{ID_x+\Delta x_N,ID_y+\Delta y_N}.$$

Listing 2 shows the GPU code for adding two rasters.

3.3.2. FocalSum by convolution

`FocalSum` analyzes the neighborhood of each cell. We have to sum up the values of the neighboring cells for each cell. Therefore, first the neighboring cells within a wedge have to be identified. We use a matrix to define the wedge. The matrix can be computed by the wedge parameters shown in Fig. 4. The GPU can compute each element of the convolution matrix in parallel by checking the following conditions:

$$x^2 + y^2 \le r^2,$$

$$\varphi_1 \le \arctan\frac{y}{x} \le \varphi_2,$$

where x and y are given by a two-dimensional thread ID. Every element of the convolution matrix can be computed in parallel.

The convolution of an input raster M and a convolution matrix W is computed by

$$Result_{s,t} = \sum_{y=1}^{W_{rows}} \sum_{x=1}^{W_{columns}} W_{x,y} * M_{s-x,t-y}.$$

The GPU can compute this equation in parallel. Every cell is computed by a sole thread. Because of overlapping neighborhoods, different threads access the same raster cells. The GPU is able to cache cells that are accessed twice or more. In our implementation we use the texture buffer for caching. The GPU manages this cache by itself. If the cache does not contain an expected value, the GPU loads this value from global memory. If

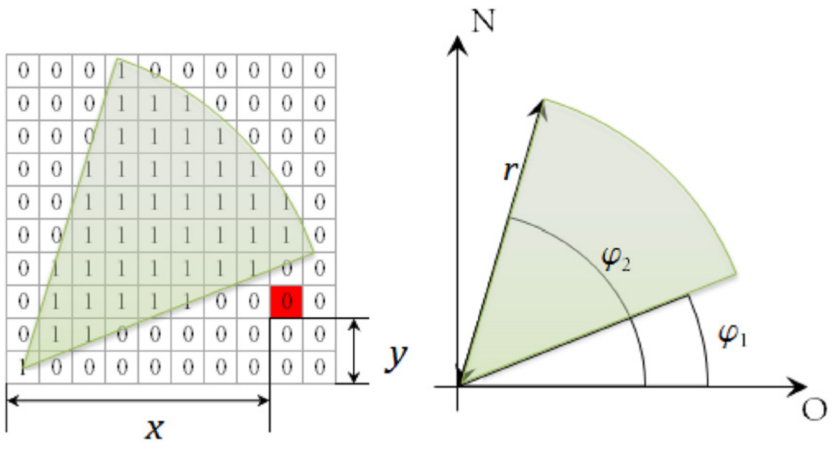

Fig. 4. Wedge described by matrix and by parameter.

the cache contains an expected value, the value is read from texture cache with speed comparable to access a register or local memory. The GPU fills up this cache when input data is read using OpenCL texture functions `read_imagei` and `write_imagef`. The use of texture functions is shown in Listing 2 and 3.

3.3.3. *Shrink computation region*

To reduce computational work, it is useful to shrink the computational region. Therefore, we need to find the first and last row respective column with a nonempty cell in a raster. The challenge is to compute only four values with hundreds of threads where each one performs the same algorithm.

We use a row and a column profile. The profiles are vectors. The row profile has R=*row count* elements and the column profile C=*column count*. $R+C$ threads initialize the vectors with zeros. We use $R*C$ threads to check if a cell is empty or not. If a cell (x,y) is not empty, we place a 1 at position x into the row profile and another 1 at position y into the column profile. Fig. 5 shows an example. Listing 3 shows the GPU code.

Now we can use four threads to search for the result in parallel. The first thread starts at the beginning of the row profile and searches for the first 1, while the second thread starts at the end of the row profile and searches backwards, also for the first 1. The result is the first and last row that contains nonempty cells. The other two threads do the same for the column profile.

3.3.4. *Euclidian distance transformation*

The Euclidian Distance Transformation (EDT) calculates the shortest Euclidian distance from each cell to the next nonempty cell. We use the parallel banding algorithm of Cao et al. (2010) to compute the EDT with a GPU.

Most EDT algorithms are based on row and column scans (see Cuisenaire, 1999, for more information). In general two row scans are used to get the next nonempty cell in a row. Fig. 6 shows the results of such a row scan. If an input row contains at least one nonempty cell, each cell of the output row contains the location of the nearest nonempty cell in that row.

Now, a column contains references to cells that could be potential nearest cells for that column. These cells are only nearest in a row. Cao et al. (2010) make use of Voronoi diagrams to remove all references from a column, that are not the nearest neighbor for at least one cell of that column. Therefore, three consecutive references are compared. According to Fig. 7, let a, b and c be nonempty cells that are referred to in column i. Let the intersection of the perpendicular bisector of a and b and column i be $p(i,u)$, and that of b and c be $q(i,v)$. If $u>v$ then b is not a nearest neighbor of any cell in column i and it can be removed as reference from this column. This comparison is done for each reference in a column by a column scan from top to bottom. Each column can be computed in parallel.

In a third step a last column scan is used to compute the distances. The columns of the final result is filled from top to bottom by comparing distances between sequently referred cells in that column. Let cell a be the first reference cell in column i, c the one after and let k be the actual computed cell of the EDT result. While the distance between a and k is shorter than the distance between c and k, a is the nearest neighbor of k and the distance between a and k is stored. When iterating down the column, the distance between c and k becomes shorter than distance between a and k. In this case, c will switch to the role a had previously and the next reference after c to the role c had. After the column scan, the result contains the Euclidian Distance Transformation.

To make use of the inherent parallelism of GPUs Cao et al. divide every row/column into a number of B bands. Every band is

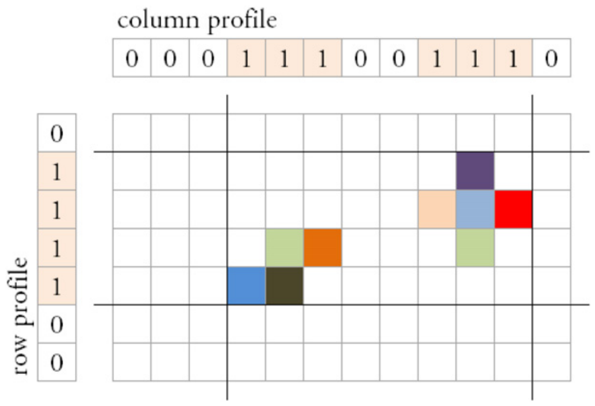

Fig. 5. Row and column profile of an example raster.

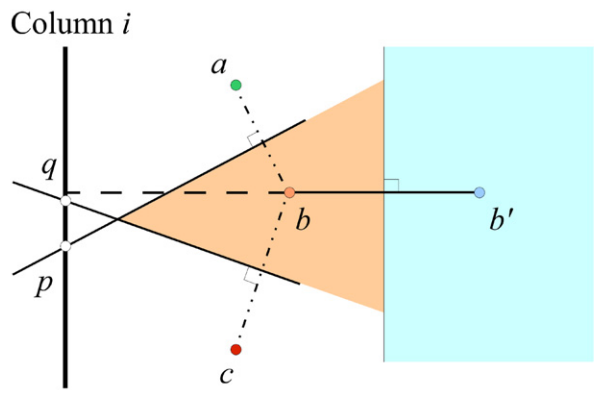

Fig. 7. Illustration of Voronai diagram properties that are used to compute EDT (Cao et al., 2010).

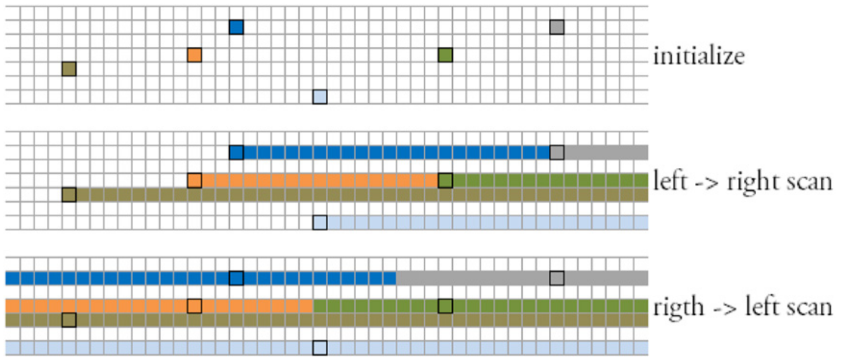

Fig. 6. The row scan process: the colors represent the nearest nonempty cell in a row.

analyzed by a scan. $B{*}R$ or $B{*}C$ threads are used to perform a complete scan of the whole raster, where R is the number of rows and C the number of columns. The results of these scans have to be propagated over the band corners. The complete paralellization is described in Cao et al. (2010).

3.3.5. Operations on zones

A zone is defined by the cells of a raster that contain the same value. Zonal Operations use a Raster with zones and analyze each zone. We concentrate on the ArcInfo operation `ZonalMean`. `ZonalMean` gets two input rasters, one with zones and one with values. `ZonalMean` then computes the mean of the values defined by a zone and stores the result in all output cells which belong to that zone. In general, the number of zones is unknown. Because of lack of dynamic memory allocation on the GPU during execution of a kernel, only a fixed number of zones is supported.

3.3.6. MaskMean

Within our example dataflow, we only need an operation that analyzes one single zone. Therefore we created an operation called `MaskMean`. A raster containing a mask defines one zone, every non-zero value marks cells belonging to that zone. `MaskMean` gets a raster with a mask and a raster with values and computes the same result as `ZonalMean` would compute for one zone.

We need to count the cells within the mask (*number*) and to sum up their values (*sum*). Again there is the problem to compute only few values (only *number* and *sum*) with many threads. We use iterations to solve the problem. Every thread in an iteration analyzes four cells. The result of each iteration is two rasters, one raster with summed up values and one with cell count. The size of these results is only a quarter of their input. Fig. 8 shows the iterations of an example. The output of the first iteration is the input for the next one. We repeat this algorithm until the resulting rasters have only a size of one cell.

The mean value is the quotient of *sum* and *number*. It has to be stored in every result cell, that lies within the mask. Therefore, we create *row count*$*$*column count* threads. Each thread fills one cell. If the cell is within the mask it is filled with the mean value, if not the cell is marked as empty.

3.4. Plugin for GRASS

Every raster operation of GRASS is implemented by modules. When a module is being applied, a new process is started executing the code of the module. A direct transfer of data between memory from one module invocation to the next one is not possible, except through files on the hard disk.

To circumvent this, a GPU module for GRASS was implemented that executes a BASIC script calling the accelerated GPU operations (see Listing 1 for an example). Using the LRU algorithm described earlier (Section 3.2), rasters can be swapped in and out on demand.

We also provide a function that allows the user to run standard GRASS modules from within the BASIC script. This makes the GPU accelerated module more versatile. Of course, the input and output rasters for these modules have to be stored on hard disk.

4. Results

We measured the performance of the individual raster operations as well as the time needed for the whole examplary dataflow, both for GPU and CPU. The test machine used for the measurements is equipped with an AMD Phenom II X4 940 CPU and a GeForce GTX 260 GPU with 896 MB of RAM.

4.1. Computations

In this subsection we focus only on the computing time of every particular raster operation. We do not care about the transfer time of data between hard disk and memory, as the caching strategy ensures that the rasters are generally already stored in GPU memory. All measurements are restricted to computations.

4.1.1. Raster algebra

Table 3 compares the computing time for adding two rasters. It is a simple operation. The CPU performs it in under half a second also for big rasters. Table 3 shows that our GPU is able to perform this operation 40 times faster than our CPU. The speedup of all

Table 3
Computing time of raster algebra ($R = A + B$).

Map size	CPU (ms)	GPU (ms)	Speedup
1138 × 1132	11	0.3	39
2275 × 2263	47	1	46
4550 × 4526	175	4	44

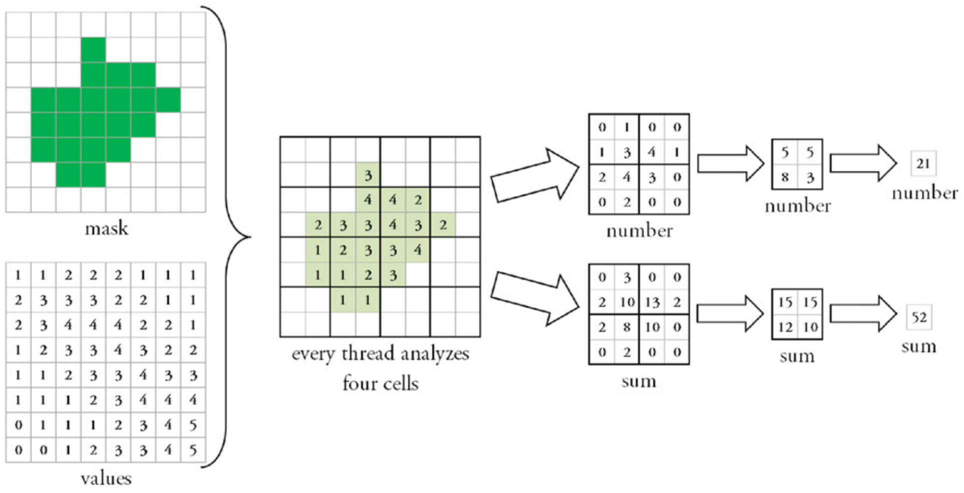

Fig. 8. Sum up and counting of masked value cells. Each arrow represents an iteration.

basic arithmetic operations and comparisons is similar to the presented one.

4.1.2. *Neighborhood analysis by convolution*

We chose the "wedge" neighborhood because of an unexpected long runtime of ArcInfo for this analysis. The ESRI GIS needs about 20 min for a wedge with $\varphi_1 = 80$, $\varphi_2 = 100$ and a radius of 50 cells (compare section 3.3.2) on a 2275×2263 raster. The most likely reason for that long runtime could be that ESRI computes the convolution matrix for each raster cell again, which is not necessary.

The GRASS module `r.mfilter` does not work with such a big convolution matrix. We developed a GRASS module that performs this operation using the CPU and use this module to compare it with our GPU implementation. Table 4 shows the computing time for a wedge with $\varphi_1 = 80$, $\varphi_2 = 125$ and a radius of 50 cells (size of convolution matrix: 38×50). Our GPU implementation is about 50 times faster than our CPU implementation.

4.1.3. *Shrink computational region*

To shrink the computational region, the smallest bounding box, that contains all nonempty cells, has to be found. CPU and GPU solve that task really fast. Table 5 shows the results. The GPU is about 10 times faster than the CPU.

4.1.4. *Euclidian distance transformation*

The PBA algorithm of Cao et al. computes an exact Euclidian distance transformation. The GRASS module `r.grow.distance` computes just an approximation. These algorithms are completely different. Thus the comparison in Table 6 is not only a comparison between CPU and GPU, but also one of the two algorithms. It is possible to implement the PBA also for CPU. Nevertheless the GPU performs the distance transformation about 40 times faster than the CPU.

4.1.5. *Mean in a particular zone*

Because of the iterations, the complexity of our GPU algorithm is higher then of the CPU one. This complexity affects the speedup. Table 7 shows computing time of CPU and GPU implementations. The GPU version is only about 15 times faster than the single threaded CPU version.

Table 4
Computing time for convolution with an 38×50 matrix.

Map size	CPU (s)	GPU (s)	Speedup
1138×1132	4.0	0.08	47
2275×2263	16.6	0.32	52
4550×4526	63.4	1.32	48

Table 5
Search time for finding smallest bounding box.

Map size	CPU (ms)	GPU (ms)	Speedup
1138×1132	3.1	0.6	5
2275×2263	11.7	1.2	10
4550×4526	44.7	3.2	14

Table 6
Computing time for Euclidian distance transformation.

Map size	CPU (ms)	GPU (ms)	Speedup
1138×1132	200	6	33
2275×2263	720	17	42
4550×4526	3080	58	52

4.2. *Batch processing of example dataflow*

In this subsection we want to focus on the complete runtime of our example dataflow. First, we analyze the effect of avoiding data transfer with the hard disk. We leave every intermediate result in memory, but we do not use the GPU for computations. Table 8 shows the result. If we only avoid redundant data transfers between memory and hard disk, the example data flow performed eight times faster when using the CPU.

In the second step we use the GPU for accelerating the raster operations. In general, it is not possible to leave every intermediate result in GPU memory. Thus, we measured different scenarios. Table 9 shows these scenarios. We need 6 rasters to perform the example dataflow. Three have to stay in GPU memory (two input and one output). If all six variables stay in GPU memory then the runtime is 150 times faster than the runtime of GRASS. If every intermediate result is stored on hard disk then the speedup drops to 1.2. The swap out of raster to main memory does not cost so much time. The runtime stays under 2 min, which equates to a speedup of about 40.

In general, it is not possible to compute any huge raster without tiling. The minimum amount of memory that is needed to compute a raster operation on the GPU is the memory for storing all input and result rasters. For instance, three rasters have to be stored for the operation `MaskMean`, two for input and one for output. We were able to compute the example dataflow with a resolution of 6100×6100 cells. Modern high performance GPUs offer up to 6 GB of memory (e.g., NVIDIA GeForce GTX 590 3 GB, ATI HD 6990 4 GB, NVIDIA TESLA C2070 6 GB). So it is possible to compute rasters with a size up to 1 GB. For most applications this size should be enough, but surely not for all. Zhang et al. (2010)

Table 7
Computing time of `MaskMean`.

Map size	CPU (ms)	GPU (ms)	Speedup
1138×1132	15	1.5	10
2275×2263	58	3.4	17
4550×4526	229	13.5	17

Table 8
Runtime of examplary dataflow with GRASS. The "optimized" runtime avoids redundant data transfer from and to the hard disk.

Map size	GRASS	Optimized (min)	Speedup
1138×1132	21 min	2.6	8.1
2275×2263	79 min	10.3	7.7
4550×4526	5 h 23 min	41.1	7.9

Table 9
Runtime of GPU on example dataflow with different storage places for intermediate results (raster size).

Number of rasters on			Runtime of example dataflow
GPU	RAM	Hard disk	
6	0	0	32.5 s
5	1	0	57 s
4	2	0	1 min 36 s
3	2	1	2 min 20 s
3	1	2	4 min 9 s
Every intermediate result on hard disk			51 min 19 s

present an approach to handle large-scale raster data with GPUs. By adapting this approach, the maximum raster size would increase. On the other hand, some computing algorithms have to be adapted to this data management.

5. Conclusion

Based on an example dataflow, we analyzed the possibilities of accelerating batch processing of spatial raster data. We use the computational power of a GPU to accelerate local operations, neighborhood analysis and a distance measurement operation. We also tried to accelerate operations on zones, but we did not find a general way.

We use GRASS as development platform for the parallelized operations and as reference for the runtime of an example dataflow. GRASS needs 79 min to perform the whole dataflow.

We changed the way a GIS handles intermediate results of batch processing. Such results are hold in RAM as long as possible (a similar mechanism offered by ArcGIS are in-memory workspaces). On Linux systems, an alternative would be to let GRASS store intermediate files on a RAM-disk. Without these caching tools, using the GPU to parallelize computations makes not much sense. If we only accelerate the computations, our example dataflow is performed 1.2 times faster in total (51 min) than GRASS would perform it without any parallelization. The change of memory management gives a big benefit to the user. If we keep every intermediate result of our example dataflow in memory, this dataflow is performed eight times faster. GRASS needs only 10 min without any parallelization. Therefore the user has to differentiate between results that should be stored on hard disk and results that are only intermediate results. Intermediate results are no longer available after a dataflow was performed.

If we also use a GPU to accelerate computations, a speedup of 150 could be reached. The runtime of the whole dataflow drops from 79 min to 33 s.

Acknowledgments

We would like to thank the reviewers for their interest in our work and their helpful feedback. We thank Dr. Rainer Schulz from the Department Ecoinformatics, Biometrics and Forest Growth of the Büsgen Institute of the Georg-August-University Göttingen for providing us with the example dataflow and raster maps. We also thank Max Daenner for some initial inspiring discussions.

References

Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D., 2006. Compilers: Principles, Techniques, and Tools, 2nd ed. Addison Wesley August.

Beutel, A., Mølhave, T., Agarwal, P.K., 2010. Natural neighbor interpolation based grid DEM construction using a GPU. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems. GIS'10. , ACM, New York, NY, USA, pp. 172–181.

Cao, T.-T., Tang, K., Mohamed, A., Tan, T.-S., 2010. Parallel banding algorithm to compute exact distance transform with the GPU. In: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games. I3D'10. ACM, New York, NY, USA, pp. 83–90 http://doi.acm.org/10.1145/1730804.1730818.

Cuisenaire, O., 1999. Distance transformations: fast algorithms and applications to medical image processing.

Neteler, M., Mitasova, H., 2008. Open Source GIS: A GRASS GIS Approach, 3rd ed. Springer.

OpenCL, 2010. The OpenCL Specification: Version 1.1. Khronos Group, document Revision: 36.

Ortega, L., Rueda, A., 2010. Parallel drainage network computation on CUDA. Computers & Geosciences 36 (2), 171–178 http://www.sciencedirect.com/science/article/pii/S0098300409002970.

Temizel, A., Halici, T., Logoglu, B., Temizel, T.T., Omruuzun, F., Karaman, E., 2011. Experiences on image and video processing with CUDA and OpenCL. In: GPU Computing Gems. Morgan Kaufmann, Boston, pp. 547–567. (Chapter 34) 〈http://www.sciencedirect.com/science/article/pii/B9780123849885000346〉.

Viola, I., Kanitsar, A., Gröller, M.E., October 2003. Hardware-based nonlinear filtering and segmentation using high-level shading languages. In: Turk, G., van Wijk, J., Moorhead, K. (Eds.), Proceedings of IEEE Visualization 2003. IEEE, pp. 309–316.

Walsh, S.D., Saar, M.O., Bailey, P., Lilja, D.J., 2009. Accelerating geoscience and engineering system simulations on graphics hardware. Computers & Geosciences 35 (12), 2353–2364. 〈http://www.sciencedirect.com/science/article/pii/S0098300409001903〉.

Wu, Y., Ge, Y., Yan, W., Li, X., 2007. Improving the performance of spatial raster analysis in GIS using GPU. In: Gong, P., Liu, Y. (Eds.), Proceedings of SPIE, vol. 6754. , SPIE, pp. 67540P1–67540P11.

Yang, Z., Zhu, Y., Pu, Y., 2008. Parallel image processing based on CUDA. Proceedings of the 2008 International Conference on Computer Science and Software Engineering. CSSE'08, vol. 3. IEEE Computer Society, Washington, DC, USA, pp. 198–201.

Zhang, J., You, S., Gruenwald, L., 2010. Indexing large-scale raster geospatial data using massively parallel GPGPU computing. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems. GIS'10. ACM, New York, NY, USA, pp. 450–453.