Computers & Geosciences 89 (2016) 88–95

Contents lists available at ScienceDirect

Computers & Geosciences

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

Research paper

Parallel flow accumulation algorithms for graphical processing units with application to RUSLE model

Johan Sten [a], Harri Lilja [b,*], Jari Hyväluoma [b], Jan Westerholm [a], Mats Aspnäs [a]

[a] Åbo Akademi University, Faculty of Science and Engineering, Vattenborgsvägen 5, 20500 Turku, Finland
[b] Natural Resources Institute Finland, Tietotie 4, 31600 Jokioinen, Finland

ARTICLE INFO

Article history:
Received 22 January 2015
Received in revised form
13 January 2016
Accepted 14 January 2016
Available online 15 January 2016

Keywords:
Algorithms
DEM
GPGPU
Hydrology
Parallel
RUSLE

ABSTRACT

Digital elevation models (DEMs) are widely used in the modeling of surface hydrology, which typically includes the determination of flow directions and flow accumulation. The use of high-resolution DEMs increases the accuracy of flow accumulation computation, but as a drawback, the computational time may become excessively long if large areas are analyzed. In this paper we investigate the use of graphical processing units (GPUs) for efficient flow accumulation calculations. We present two new parallel flow accumulation algorithms based on dependency transfer and topological sorting and compare them to previously published flow transfer and indegree-based algorithms. We benchmark the GPU implementations against industry standards, ArcGIS and SAGA. With the flow-transfer D8 flow routing model and binary input data, a speed up of 19 is achieved compared to ArcGIS and 15 compared to SAGA. We show that on GPUs the topological sort-based flow accumulation algorithm leads on average to a speedup by a factor of 7 over the flow-transfer algorithm. Thus a total speed up of the order of 100 is achieved. We test the algorithms by applying them to the Revised Universal Soil Loss Equation (RUSLE) erosion model. For this purpose we present parallel versions of the slope, LS factor and RUSLE algorithms and show that the RUSLE erosion results for an area of 12 km x 24 km containing 72 million cells can be calculated in less than a second. Since flow accumulation is needed in many hydrological models, the developed algorithms may find use in many other applications than RUSLE modeling. The algorithm based on topological sorting is particularly promising for dynamic hydrological models where flow accumulations are repeatedly computed over an unchanged DEM.

1. Introduction

Soil erosion is a world-wide problem that has severe environmental impacts as well as economic consequences. Surface topography is a key factor in most erosion models (Toy et al., 2002) and thus all models use elevation information in some way. The light detection and ranging (Lidar) technique has made a breakthrough in recent years and has increased the accuracy of digital elevation models significantly. The revised universal soil loss equation (RUSLE) is used in the present work (Renard et al., 1991). The simple and robust model structure and minimal input data needed makes RUSLE capable of producing even global erosion risk maps (Yang et al., 2003). Although the model is simple, the continuous increase in the DEM resolution makes the computational cost of DEM-based RUSLE modeling a critical issue since finer DEM resolutions have been shown to improve the accuracy of RUSLE results (Zhao et al., 2010). Because long computation times are needed to produce detailed erosion maps for large areas it would be advantageous and highly important to be able to increase the speed of flow computations.

Using graphics processing units (GPU) for general applications has become known as general purpose computing on graphical processing units (GPGPU). GPUs are extremely well suited for data-parallel problems because they contain a large number of cores that concurrently execute the same instructions on independent data. This is known as single instruction multiple data (SIMD) parallelism (Trobec et al., 2009). Each of the cores runs a thread that receives a portion of the total work and the execution of the threads is in parallel.

Here we describe four approaches for parallel flow accumulation calculation. The Open Computing Language OpenCL (Munshi, 2011) has been chosen for implementation as it is suitable not only for any GPU but can also be run on a large range of other types of devices. In addition to two previously published GPGPU methods,

* Corresponding author.
E-mail addresses: harri.lilja@luke.fi (H. Lilja), jan.westerholm@abo.fi (J. Westerholm).

http://dx.doi.org/10.1016/j.cageo.2016.01.006
0098-3004/

the flow-transfer algorithm (Ortega and Rueda, 2010) and the in-degree-based algorithm (Zhan and Qin, 2011), we introduce two new methods called dependency transfer and topological sort based algorithms for GPGPUs. These algorithms are used as a part of the GPGPU implementation of RUSLE and their performance is measured and benchmarked against ArcGIS and SAGA, which serve as industry standards.

2. RUSLE model implementation for GPGPU

In this section we first give a brief description of the RUSLE model. Readers are referred to Kinell (2010) for more details. Then, the overall structure of the parallel OpenCL implementation is described.

2.1. RUSLE model

RUSLE is a computerized version of USLE with revised estimations of its equation factors. RUSLE estimates the long-term annual average soil loss per unit area A (in tons/hectare/year) as the product of five factors *R*, *K*, *LS*, *C* and *P*. The soil loss estimate is given by the equation

$$A = R \cdot K \cdot LS \cdot C \cdot P, \tag{1}$$

where *R* quantifies the impact of rainfall; *K* the soil erodibility; *LS* the effect of surface topography; *C* the cover management and *P*, human efforts to prevent erosion. Three calculations are needed to determine the *LS* factor from the DEM, namely slope, flow routing and flow accumulation (see Section 3.3 for details). *R* and *K* have units while the rest are relative and unitless quantities with values ranging from 0 to 1, the largest value corresponding to conditions identical to the unit reference plot used by RUSLE.

2.2. OpenCL implementation

OpenCL is an open-standard programming language platform that enables parallel high performance computation on heterogeneous architectures (Jones, 1998). Although OpenCL allows several or all available OpenCL devices to be used at the same time, in this work we use a single CPU or GPU as the device.

We implemented the kernels, i.e., the programs that will run on the device, using OpenCL (Munshi, 2011). As the device often has a highly parallel architecture the kernels are designed as a collection of preferably independent work-items, each executing a small part of the entire calculation. The kernel is scheduled to be executed in blocks of lock-stepped parallel work-items called work-groups. Each work-group can contain hundreds of work-items. At the program compile time we do not yet know which device the program will run on. Therefore, OpenCL has to compile the kernels on-the-fly for the specific device at hand before they can be run on the device. This is done automatically by the software driver for the intended device.

The overall structure of the OpenCL implementation of RUSLE is depicted in Fig. 1. Four factors, *R, K, C* and *P*, are typically local by nature, and they are given by similar rasters of cell values as the DEM. The fifth factor, LS, requires as input the DEM, a choice of flow routing model (D8, FD8, FDD8 or anisotropic), and a choice of flow accumulation method (transfer, dependency transfer, in-degree or topological). All steps are run by one invocation of the corresponding OpenCL kernel except the flow accumulation kernels which may require repeated invocations.

3. Algorithms

In this section we describe the flow routing, flow accumulation, slope, LS and RUSLE algorithms with OpenCL using pseudocode. As a sample case we have used the FD8 flow routing but all pseudocodes are easily adapted for the other flow routing methods of Section 3.1.

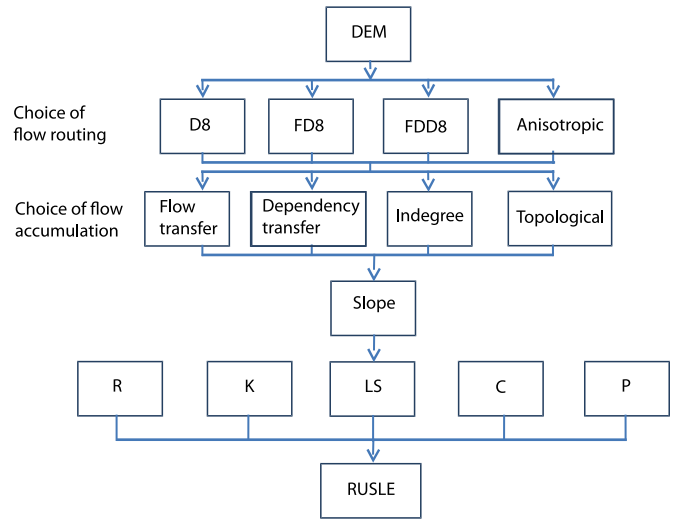

Fig. 1. Structure of the GPGPU RUSLE model and the program options.

3.1. Flow routing

The flow routing algorithms used in this paper are D8 (O'Callaghan and Mark, 1984), FD8 (Quinn et al., 1991), FDD8 (Schauble, 2005) and Anisotropic (Hyväluoma et al., 2013), explained in detail in their original articles. Here we only give a brief description of the basic idea behind the algorithms in Supplementary material. A review of the flow routing algorithms is presented in Wilson et al. (2008).

3.2. Flow directions and dependencies

For the flow accumulation computations it turns out to be advantageous to record into which DEM cells the flow will go from a given cell (flow directions) or from which cells a given cell will receive its flow (flow dependencies). Here we present algorithms to determine these two quantities.

Flow directions consist of those downslope directions into which a cell distributes its accumulated flow. Flow directions are determined by the used flow-routing algorithm. The number of recipient cells varies from at most a single recipient (D8) to possibly all downslope cells (FD8). As a sample case, the kernel to determine the FD8 flow directions is presented in Algorithm 1 as pseudocode in Supplementary material. The DEM serves as the input while the output is written to a byte array `FD8_Dirs`with the same dimensions as the original DEM. For each DEM cell, the result is stored as a byte in `FD8_Dirs` whose eight bits correspond to the eight possible directions towards the neighboring cells, e.g., starting from north and going clockwise. The pseudocode line 3, `parallel for each cell`, indicates that one OpenCL work-item is run for each DEM cell, independently and in an arbitrary order.

Flow dependencies give an inverted description of the flow directions. Instead of determining the directions into which a given cell distributes its flow, a downslope cell keeps track of those cells from which it receives flow. To compute the flow dependencies of a cell, the flow directions for all cells are first determined. Each cell is then treated as a potential downslope recipient cell, and the flow directions of the neighboring cells are checked to see if those cells will contribute flow to the cell under investigation. In Algorithm 2, Supplementary material, a kernel for

determining flow dependencies is presented. The input is a byte array `Dirs` of the same size as the DEM array, containing the flow directions for the appropriate flow model (the output from Algorithm 1) and the output is another DEM sized byte array `Deps` containing the dependencies, that is, for each cell the result is a byte whose eight bits correspond to the eight possible flow directions from the neighboring cells. Notice that even when the flow routing algorithm may limit the number of flow directions, the number of dependencies can be anything between zero and eight. For example, in D8 there is at most one flow direction from a cell, but even in D8 a pit cell may have eight dependencies. This Dependencies kernel works in the same way irrespective of the chosen flow model (D8, FD8, FDD8, anisotropic) but the result depends on the selected flow model as it affects the input flow directions.

3.3. Flow accumulation

Flow accumulation is a key quantity in many surface hydrological simulations and in the calculation of the RUSLE LS factor. Flow accumulation kernels can be either distributing or gathering. A distributive algorithm sends the correct amount of water from a cell to its downslope cells. The number of recipient cells and the amount they receive are determined by the chosen flow routing model. Distributive algorithms can be implemented straightforwardly in sequential programs. However, when cells are processed by parallel threads, a data race may occur if the values from two (or more) upslope cells are added simultaneously to the same recipient cell. Parallel distributive algorithms need to utilize synchronization or atomic operations in order to prevent data races. A gathering kernel works in the opposite way. Here a cell gathers accumulation by reading the flow from its upslope neighbors. The advantage of the gathering variant is that it completely avoids the critical sections, i.e., sections of code that access shared resources that must not be updated concurrently. Ortega and Rueda (2010) have examined both variants and concluded that the synchronization overhead in distributing is greater than the cost of gathering the values from the upslope cells. For this reason, we only consider gathering kernels.

In the next subsections we describe four parallel flow accumulation algorithms: flow transfer, dependency transfer, indegree accumulation and accumulation using topological sorting.

3.3.1. Flow transfer

The conceptually simplest flow accumulation algorithm is based on flow transfer (O'Callaghan and Mark, 1984; Ortega and Rueda, 2010). The flow transfer algorithm iteratively transfers water from a cell to its neighbors according to the chosen flow routing model to implement flow transfer in parallel (see Algorithm 3), the dependency array is used to determine from which neighboring cells a given cell can receive flow. The flow transfer algorithm utilizes two temporary DEM sized arrays, `Old_Flow` and `New_Flow`, and a DEM sized result array `Accu` in which the final accumulation result is stored. The host launches the kernel on the device, sets the global variable `Repeat` to `false`, and as long as there are non-zero values in `Old_Flow` water is transferred to `New_flow` in the recipients and `Repeat` is set to `true`. Before the first iteration the `Old_Flow` array is initialized to a starting value, eg. 1, and `New_Flow` and `Accu` arrays are initialized to zero. After each iteration, `New_flow` is added to `Accu` and the host swaps the `Old_Flow` and `New_Flow` arrays, i.e., the output in `New_Flow` from the previous iteration will act as input `Old_Flow` in the next iteration. Iterations continue until `Repeat` is no longer set to `true` by any work-item.

When multiple flow direction algorithms are used, the flow from a cell is divided into smaller fractions that are transferred to all downslope cells. As cells usually have several downslope neighbors, pre-computation and storage of the flow fractions would require a lot of memory. Therefore we calculate and store only the normalization, i.e., the total sum of all flow fractions, in a grid of the same size as the original DEM, eg. `NormFD8`. Normalization values are computed in parallel, independently for each grid cell (see Supplementary material). The flow fractions are then computed by the accumulation kernel (function `computeFractionFD8` in Algorithm 3) as needed using the pre-computed normalization values. Actually the accumulation algorithms presented in this paper can be used for any multiple-direction flow model simply by changing the function `compute FractionFD8` to the appropriate one.

Algorithm 3. FlowTransfer kernel for flow accumulation computation.

```
Repeat = false
kernel FD8FlowTransfer(input Old_Flow, Deps, NormFD8, Accu
            output New_Flow, Accu, Repeat)
parallel for each cell in DEM
     if Old_Flow(cell) is 0 or cell is on DEM border then
     New_Flow(cell) = 0
  else
         accumulation = 0
         for each direction
                    if Deps(cell) contain direction then
                    neighbor = getNeighbor (cell,direction)
                    accumulation += Old_Flow (neighbor)*
                    computeFractionFD8 (neighbor,direction,NormFD8(neighbor))
                      end ifs
         end for each
     if accumulation > 0 then
          Accu(cell) += accumulation
                   Repeat = true
          end if
       New_Flow(cell) = accumulation
    end if
end parallel for each
end kernel
```

3.3.2. Dependency transfer

The flow transfer algorithm involves transferring the flow from a cell to its neighbors until the upslope parts of the DEM have run out of water. This approach results in a large number of repeated computations for some DEM cells. To minimize this repeated data transfer, we present a new algorithm, dependency transfer, where the accumulated water flows only from those cells which will not receive any more flow.

The dependencies calculated in Algorithm 2 can be used to decide which cells have received all their flows. In the dependency transfer algorithm a cell which has non-zero dependencies is assumed not to have received its final accumulation value whereas a cell with zero dependencies has. In Algorithm 4 each cell is tested whether any of its neighbors has reached its final accumulation value. If this is the case, the flow from that cell can be transferred and the dependency between these two cells is removed. Once all dependencies have been removed the flow accumulation for a cell is complete and that cell is no longer considered for accumulation

by future iterations. By keeping track of the dependency changes, the temporary flow arrays in the flow transfer algorithm can be replaced with temporary dependency arrays which require less memory. This parallelization strategy is demonstrated in Algorithm 4 for the FD8 flow routing model. The host controls the dependency transfers in the same way as in the flow transfer method by using the global variable `Repeat`. All input variables are DEM sized except the variable `Repeat`.

Algorithm 4. : `FD8DependencyTransfer` kernel for flow accumulation computation.

```
Repeat = false
kernel FD8DependencyTransfer(input Old_Deps, NormFD8, Accu
                             output New_Deps, Accu, Repeat)
parallel for each cell in DEM
            Dependencies = Old_Deps(cell)
            accumulation = 0
            for each direction
            if Dependencies contain direction then
              neighbor = getNeighbor(cell, direction)
              neighbor_dependencies = Old_Deps(neighbor)
              if neighbor_dependencies is 0 then
                    accumulation += Accu(neighbor)*
                    computeFractionFD8(neighbor,direction,NormFD8(neighbor))
                                removeBit(Dependencies,direction)
                Repeat = true
        end if
    end if
                  end for each
              Accu(cell) += accumulation
              New_Deps(cell) = Dependencies
end parallel for each
end kernel
```

3.3.3. Indegree-based algorithm

Zhan and Qin, (2011) proposed another improvement to the flow transfer algorithm by utilizing graph theory to improve the computational efficiency. In their method the DEM is treated as a graph where each cell is a node and the flow paths between the cells are edges. Each cell has a numerical property called indegree which is equal to the number of incoming flow paths (Weiss, 1995), and which is easily computed from the number of set bits in the `Deps` byte (Algorithm 2).

The indegree accumulation kernel for the FD8 flow routing model is given in Algorithm 5, and it is controlled by the host in the same way as in the previous iterative transfer methods. For each iteration only cells with indegree value zero are processed by letting downslope cells gather their accumulated flows. Once the cell has been fully processed we mark it by the value -1. The algorithm uses both distributive and gathering elements. A cell gathers its accumulation values by reading from neighboring cells while the `Indegrees` array is updated using atomic operations. This prevents data races as the other threads are forbidden to access the data during atomic operations. Here all input variables are again DEM sized except the variable `Repeat`.

Algorithm 5. : FD8IndegreeAccumulation kernel for flow accumulation computation.

```
Repeat = false
kernel FD8IndegreeAccumulation(
                    input Dirs, Deps, Indegrees, NormFD8, Accu
                    output Accu, Repeat)
parallel for each cell in DEM
    indegree = Indegrees(cell)
    if indegree is 0 then
         flow_to_cell = 0
         for each direction
                 if Deps(cell) contain direction then
                      neighbor = getNeighbor(cell,direction)
                      flow_to_cell += Accu(neighbor)
                      *computeFractionFD8(neighbor,direction,NormFD8(neighbor))
              end if
         end for each
              Accu(cell) += flow_to_cell
           for each direction
                if Dirs(cell) contains direction then
                      neighbor = getNeighbor(cell,direction)
                      atomicDecrement(Indegrees(neighbor))
                 end if
                 end for each
         Indegrees(cell) = -1
           Repeat = true
    end if
end parallel for each
end kernel
```

3.3.4. Topological sort-based algorithm

The accumulation algorithms presented so far rely on a brute force approach by examining all cells at each iteration. Sequential implementations of flow accumulation algorithms utilize sorting approaches as a pre-processing step to find an optimal computation path through the DEM and thus to speed up the accumulation process. Here we need to know which cells can be computed in parallel and when they can be computed. An ideal solution would be to group together all cells for which the flow accumulation can be computed in parallel.

The topological sort algorithm (Khan, 1962) can be adapted to place the DEM cells into a one-dimensional order that guarantees that a cell does not depend on any cell after it, only on cells before it, in the topologically sorted order. The topologically sorted cells can now be grouped using "Start" and "End" indices that tell us the following: Beginning from cell number "Start" in the topological order, we can process all cells in the interval [Start, End] *in parallel* because they do not depend on each other.

The topological sort and its application to DEMs were mentioned by Zhan and Qin (2011), but they did not use topological sort to preprocess the DEM to speed up the computation nor was the topological ordering considered from a parallelization point of view. However, our modified topological sort fulfills all the requirements of a preprocessing step. The DEM is two-dimensional but saved in computer memory as a continuous one-dimensional block. Each cell has a position within that block called the index,

from which both the row position and the column position can be inferred if the DEM row length is known. The optimal sorting solution should simply keep track of which set of indices are computed during which iteration of the flow accumulation. In the present work, the topological sort is accomplished as a sequential procedure to be run on the host and the algorithm is similar to the improved topological sort algorithm (Weiss, 1995). The topological sort-based algorithm has been previously used to speed up flow related computations by Huang and Lee (2013), Braun and Willett (2013), and Schwanghart and Scherler (2014) but only for sequential computations, not as groups of cells for parallel computation.

The main benefit of using a topologically sorted DEM to calculate flow accumulation compared to transfer based methods is that the algorithm scales better as the total number of work-items needed is size(DEM) instead of size(DEM)x(length of longest downslope). The utilization of the GPU improves as a larger number of work-items in the kernel will have work to do. Additionally, if a gathering kernel is used, no data race occurs which makes synchronization unnecessary. Neither is any communication needed between the host and the device until the process has completed, and thus the host can avoid checking the rerun flag `Repeat` from the device to see if the algorithm needs to be repeated and thus stalling the device in the process. Provided that the DEM is static, the topological sort only needs to be computed once. The sorted DEM and the corresponding start and end values for the parallel groups can be stored in memory and be reused by subsequent flow accumulation executions. Finally, because there are no data races and device-to-host communication is unnecessary, the kernel becomes simpler and faster. The primary disadvantage with the topological sort is the need to copy the sorted indices once into device memory, possibly straining already limited memory resources.

The end result of the topological sort is the array `Sort`, which contains the indices of the cells in the DEM and has the same number of elements as the DEM. An additional output is a list of sizes that tells us how large each topologically sorted group is. Pseudocode for the topological sort is presented in Supplementary material. The topologically sorted DEM can now be used to calculate flow accumulation efficiently on the GPU by invoking one kernel for each topologically sorted group, with the DEM cells found from the list `Sort` with indices ranging from `Start` to `End`. A sample case of topological sorting is presented in Fig. 2 where a 5×5 DEM with D8 flow routing has been topologically ordered into seven groups, numbered 1–7. The number of cells in each topologically ordered group is {17,2,2,1,1,1,1} indicating that we can first process 17 cells in parallel, then 2, and so on. The total number of threads is 25 which is clearly less than the number of threads in the flow transfer scheme, $7 \times 25 = 175$.

A flow accumulation algorithm utilizing the topologically sorted DEM is presented in Algorithm 6. Due to device hardware properties the number of work-items launched is often a multiple of 32. Therefore, we choose the number of work-items to be the smallest multiple of 32 equal to or exceeding the number of items in the topologically sorted group. To make sure we are not going beyond our present topologically sorted group, line 6 was inserted in Algorithm 6. The `thread_ID` is a numerical identifier for each thread, starting from 0. Thus, in line 5 in the algorithm, each thread chooses an individual work item among the topologically ordered values from the range [Start, End]. The cells in this range are independent of each other and can thus be processed in parallel. The arrays Sort, Deps, NormFD8 and Accu have the same size as the DEM while Start and End are arrays whose lengths depend on how many topologically sorted groups we have found. Roughly, the sizes of these two arrays are equal to the length of the longest slope.

Algorithm 6. `FD8TopologicalAccumulation` kernel for flow accumulation computation.

```
1.  kernel FD8TopologicalAccumulation(
2.                    input Sort, Deps, Start, End,
    NormFD8,
3.                    output Accu)
4.  parallel for each thread in invoked topologi-
    cally sorted group
5.      sort_position = Start + thread_ID
6.      if sort_position <= End then
7.          cell = Sort(sort_position)
8.          flow_to_cell = 0
9.          for each direction
10.             if Deps(cell) contain direction
    then
11.               neighbor = getNeighbor(cell,
    direction)
12.               flow_to_cell += Accu(neighbor)*
13.                 computeFractionFD8(neighbor,
    direction,NormFD8(neighbor))
14.             end if
15.         end for each
16.         Accu(cell) += flow_to_cell
17.     end if
18. end parallel for each
19. end kernel
```

3.4. Slope and LS-factor algorithms

To compute the *LS* factor in the RUSLE model the slope angle in the steepest direction has to be computed for each cell in the DEM. Slope refers to the absolute value of the inclination angle of the surface. The slope is computed by the discrete Horn differentiation operator (Horn, 1981). The slope angle kernel can be run in parallel over the DEM as each cell slope can be computed independently of other cells.

The *LS* factor combines the steepness factor *S* and slope length *L* into a single factor. To take the flow convergence into account, flow accumulation is often used in place of slope length. We use the equation for the *LS* factor presented by Mitasova et al. (1996) which relates the *LS* factor to flow accumulation per unit contour width and slope by the equation:

$$LS(\text{cell}) = (m+1)(\text{Accumulation(cell)}*\text{resolution(meters)}/A_0)^m \\ (sin(\text{SlopeAngle(cell)}))/b_0)^n \quad (2)$$

where $A_0 = 22.1$ m, $b_0 = \sin(5.14°) = 0.0896$, $m = 0.4$ and $n = 1.3$. In RUSLE the values for m and n are allowed to be terrain specific, providing the opportunity to fine tune the results for a particular type of DEM. Once the accumulation and the slope angle have been evaluated we can normalize these values for the unit plot values of the RUSLE model using Eq. (2). Again, this can be calculated in parallel, independently for each cell in the DEM.

3.5. Rusle soil erosion

The RUSLE long term soil erosion estimate is now given as the product of the five factors in Eq. (1). Four out of five factors, *R*, *K*, *C* and *P*, are given as independent input values on similarly-sized rasters as the DEM: `R_DEM`, `K_DEM`, `C_DEM` and `P_DEM`. The fifth factor, *LS*, depends on two properties of a cell, the slope and the accumulation, the latter of which can be computed in parallel on GPUs as presented above.

The `Rusle` kernel is presented in Supplementary material as

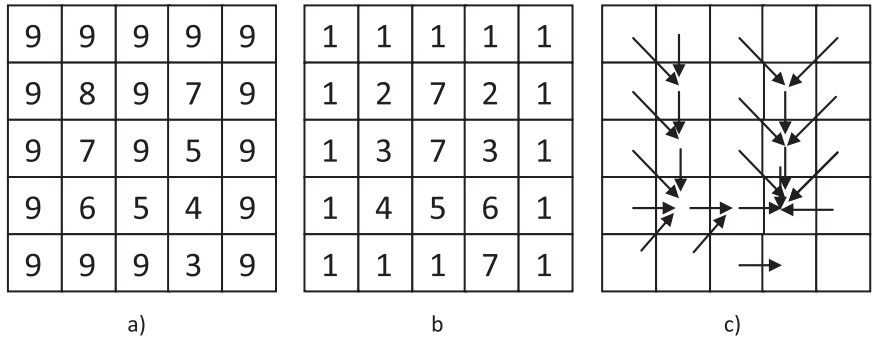

Fig. 2. Topological sort of a 5 × 5 DEM with D8 flow routing: (a) cell elevations, (b) index to topological sort group and (c) flow directions.

Algorithm 7, which computes the final RUSLE grid. This kernel is perfectly suited for parallel computing as the calculations for each cell are independent of each other.

4. Results

Three OpenCL capable GPUs were used to measure the performance of the algorithms presented above: NVIDIA Geforce 680 GTX, NVIDIA Geforce GTX Titan and AMD Radeon HD 7970. Since OpenCL can also run on a CPU, the program was executed on an Intel i7-3770K hyper-threaded quad-core processor. The properties of the devices are presented in Table 1. All devices were run using the OpenCL platforms provided by their respective device manufacturers.

The tests were performed using a DEM[1] with 2 m resolution. The size of the DEM is 12 km × 24 km, mean elevation 97 m and minimum/maximum values 76 m/142 m. In the tests we assume that the DEM is preprocessed, and that flat areas and depressions have already been handled in an appropriate way. Therefore, the performance times do not include file I/O or DEM preprocessing times (see footnote 1).

In the first test (Tables 2 and 3) we compared the performance of the various kernels of the GPGPU implementation by running the transfer based and the topological sort based accumulation methods for all flow routing models on the three different OpenCL platforms. In our second test (Table 4) we compared the speed of the GPGPU algorithm to industry references ArcGIS[2] and SAGA (see footnote 2), with an NVIDIA Titan card. Here topological sort was not used in order to maintain compatibility with ArcGIS and SAGA and their file formats.

The timing results show that the LS, Slope and RUSLE kernels perform well on GPUs and are the fastest individually measured kernels in the program (Table 3). The RUSLE kernel is currently the fastest as it is small and contains few instructions and branches. The kernels perform well because their data-parallel structure fits the single instruction multiple data execution model of the GPU architecture perfectly. The algorithms involve very little computation per cell and all cells can be computed at the same time. Only a minor amount of arithmetic operations is performed, and the outputs consist of single values per work-item. Our design was to arrange for the kernels to access memory in uniform strides and within the work groups almost all work items should take the same control paths. Optimization has been focused primarily on scheduling as many work-items as possible, keeping the device fully occupied. In the future if the number of cells in the DEM grows beyond what can fit into a single 32-bit word (corresponding to 16 GB for the DEM using single precision floating point values for the heights) the size of the topologically sorted indices will require 64-bit integers.

The second test was run with both ArcGIS and SAGA CPU based algorithms and our GPGPU code with flow transfer algorithm and D8 flow routing model. We used four file formats in the test. The ARC_ASCII,ARC_BINARY and SGRD are the industry standards while GPGPU binary refers to a binary file format created for our GPGPU implementation, essentially replacing the ASCII input file by an equivalent binary file.

ESRI's Arc_ASCII was used as a common import format. For ArcGIS the time is the sum of the run times for flow direction and flow accumulation. The flow accumulation is calculated from a binary file in ArcMap in both cases since the ASCII input file is converted to a binary file in ArcGIS before the flow accumulation is calculated. The GPGPU implementation does not require the flow direction as an external input file as the flow direction is formed internally. The execution time is expressed in seconds in Table 4.

5. Discussion and conclusions

In this paper we evaluate the possibility of using GPUs to accelerate the computation of flow routing and flow accumulation models by parallel calculation, and we present an application of these techniques using the RUSLE erosion model. To achieve this, every part of the RUSLE computation was implemented using OpenCL, with positive results. The algorithms were flow routing, flow direction and flow accumulation followed by Slope, LS and RUSLE. Most algorithms perform calculations on each cell independently of the calculations on other cells. This suits the GPU architecture perfectly as it enables the algorithms to make full use of the computational power of the GPU. The exceptions are the flow routing algorithms combined with the accumulation process which are the largest contributors to the overall time. Their slow computation is a result of the strict order of computing and it is an inherent property of the accumulation process that we presently are unable to bypass. The optimizations were targeted primarily on the way the work is scheduled and avoided smaller optimizations on the instruction level. For example in the topological sort based flow accumulation algorithm we schedule only such threads that complete the computations for a cell once it has been scheduled, instead of scheduling many threads for a cell as in the flow transfer algorithm.

Two new flow accumulation methods were devised, implemented and tested. The first is the dependency transfer method which is a modification of the flow transfer algorithm. It keeps track of the changes to the DEM dependency graph between two iterations of the algorithm instead of keeping track of the flow. This approach reduces the memory requirements by replacing

[1] The DEM (L3443) is downloadable at: https://tiedostopalvelu.maanmittauslaitos.fi/tp/kartta

[2] ESRI 2011. ArcGIS Desktop: Release 10.2.1, 32 Bit. Redlands, CA: Environmental Systems Research Institute.

Table 1
The test devices and their properties.

Device	Cuda cores/stream processors	Clock speed, Mhz	On-board memory, GB	OpenCL driver version
Nvidia Geforce GTX Titan, GPU	2688	836	6	355.60
Nvidia Geforce 680GTX, GPU	1536	1058	4	304.88
AMD Radeon HD 7970, GPU	2048	1000	3	1214.3
Intel i7-3770K, CPU		3500	32	1.2.0.76921

Table 2
Performance times in seconds for several flow routing kernels on three different OpenCL platforms. The times include the calculations of flow normalizations, flow directions and dependencies but they do not include input/output operations. The time needed to topologically sort the test DEM was 6.17 s and this is not included in the accumulation times.

Device	Geforce 680GTX, GPU	Radeon HD7970, GPU	Intel I7-3770K, CPU
Flow transfer (Ortega and Rueda, 2010)			
D8	1.54 s	1.33 s	16.64 s
Dependency transfer			
Anisotropic	16.37 s	9.63 s	113.28 s
D8	1.35 s	1.00 s	6.90 s
FD8	9.45 s	7.80 s	55.56 s
FDD8	9.94 s	6.54 s	59.65 s
Indegrees (Zhan and Qin, 2011)			
FD8	8.96 s	5.59 s	25.57 s
Topological sort			
Anisotropic	2.60 s	1.31 s	20.07 s
D8	0.32 s	0.24 s	0.84 s
FD8	0.96 s	0.64 s	2.37 s
FDD8	0.96 s	0.67 s	2.36 s

Table 3
Performance times in seconds for slope, LS and RUSLE kernels on three different OpenCL platforms. The times do not include input/output operations[a].

Device	Geforce 680GTX, GPU	Radeon HD7970, GPU	Intel I7-3770K, CPU
Slope	0.12 s	0.11 s	0.34 s
LS	0.13 s	0.11 s	0.46 s
RUSLE	0.12 s	0.10 s	0.23 s

[a] SAGA 2.2.3 System for Automated Geoscientific Analyses.

Table 4
Comparison between ArcGIS, SAGA and GPGPU algorithms performed with NVIDIA Geforce GTX Titan GPU and Intel Xeon CPU-E5-2630 v2 (2.60 GHz).

Implement	Data type	Time			Speed up
		Flow direction	Flow accumulation	Total	
ArcGIS 10.2.1	ARC_ASCII	95 s	75 s	170 s	1
ArcGIS 10.2.1	ARC_Binary	20 s	75 s	95 s	1
Saga 2.2.3	ARC_ASCII	78 s		78 s	1.95
Saga 2.2.3	SGRD_Binary	77 s		77 s	15.4
GPGPU	ASCII	40 s		40 s	4.25
GPGPU	Binary	5 s		5 s	19.0

temporary flow arrays by dependency arrays that require less memory, and thereby achieves a small speed up on the GPU (Table 2). The second new method is the topological sort algorithm, which utilizes a sorted DEM along with a list of group sizes to compute the accumulation very quickly. The advantages include less work-items to schedule, the GPU is filled with work with a minimum of idle threads, no synchronization or read-back is needed, no device stalling occurs, and kernel complexity is reduced. When comparing the topological sort against the corresponding transfer-based algorithms, a speed-up factor of minimum four and on average seven is obtained. This comes at the cost of additional memory requirements on the device and the time it takes for the host to sort the DEM before the accumulation can begin. The sort can be done once and saved to disk, allowing a reduced total time on subsequent executions. Thus the topological sort is particularly beneficial in applications where flow accumulations are computed several times for the same DEM. For all flow routing models except D8 we found that the topological sort based accumulation performed faster than the flow transfer even for a single simulation. The sorting time increases linearly with the number of cells. When the DEM size increases, at some point topological sorting becomes the fastest alternative even for D8 depending on a DEM's character and used hardware. On the other hand, for smaller DEM sizes the competitiveness of the algorithm is reduced.

The current implementation enables the calculation of RUSLE for an area of 12 kmx24 km (72 million cells, one UTM-25 mapsheet) in less than a second when using binary datasets and stored topological sorts. In dynamic hydrological models which repetitively compute flow accumulation over an unchanged DEM, the use of topological sorting will gain even greater benefits as sorting needs to be performed only once. On larger areas the GPGPU implementation of the RUSLE model would allow a faster updating of maps of erosion. Our implementation can be developed further to a platform that can be used as a system for erosion map production on large areas, e.g. a whole country

Acknowledgments

The research for this paper was financially supported by the Finnish Ministry of Agriculture and Forestry, Grant no: 1838/312/2013 and the Niemi Foundation.

Appendix A. Supplementary material

Supplementary data associated with this article can be found in the online version at http://dx.doi.org/10.1016/j.cageo.2016.01.006.

References

Braun, J., Willett, S.D., 2013. A very efficient O(n), implicit and parallel method to solve the stream power equation governing fluvial incision and landscape evolution. Geomorphology 180–181, 170–179. http://dx.doi.org/10.1016/j.geomorph.2012.10.008.
Horn, B., 1981. Hill shading and the reflectance map. Proc. IEEE 69, 14–47.
Huang, P.-C., Lee, K.T., 2013. An efficient method for DEM-based overland flow routing. J. Hydrol. 489, 238–245. http://dx.doi.org/10.1016/j.jhydrol.2013.03.014.
Hyväluoma, J., Lilja, H., Turtola, E., 2013. An anisotropic algorithm for digital elevation models. Comput. Geosci. 60, 81–87. http://dx.doi.org/10.1016/j.cageo.2013.07.012.
Jones, K.H., 1998. A Comparison of algorithms used to compute hill slope as a property of the DEM. Comput. Geosci. 24, 315–323.
Khan, A.B., 1962. Topological sorting of large networks. Commun. ACM 5, 558–562.
Kinell, P., 2010. Event soil loss, runoff and the universal soil loss equation family of models: a review. J. Hydrol. 385, 384–397. http://dx.doi.org/10.1016/j.jhydrol.2010.01.024.

Mitasova, H., Hofierka, J., Zlocha, M., Iverson, L.R., 1996. Modelling topographic potential for erosion and deposition using GIS. Int. J. Geogr. Inf. Syst. 10, 629–641.

Munshi, A., The OpenCL Specification. < https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf >, 2011 (accessed 1.12.14).

O'Callaghan, J.F., Mark, M., 1984. The extraction of drainage networks from digital elevation. Data. Comput. Vis., Graph., Image Process. 28, 323–344.

Ortega, L., Rueda, A., 2010. Parallel drainage network computation on Cuda. Comput. Geosci. 36, 171–178. http://dx.doi.org/10.1016/j.cageo.2009.07.005.

Quinn, P., Beven, K., Chevalier, P., Planchon, O., 1991. The prediction of hillslope flow paths for distributed hydrological modelling using digital terrain models. Hydrol. Process. 5, 59–79.

Renard, K.G., Foster, G.R., Wessies, G.A., Porter, J.P., 1991. Revised universal soil loss equation. J. Soil Water Conserv. 46, 30–33.

Schauble H., Sedimentfrachtprognosen mit GIS—Neue Strategien für globale Modellgleichungen unter besonderer Berucksichtigung von Staudammen und des zeitlichen Wandels.Dissertation.Institut für Angewandte Geowissenschaften, Technische Universität Darmstadt, Darmstadt,Germany. < http://tuprints.ulb.tu-darmstadt.de/id/eprint/653 >, 2005 (accessed 10.09.15).

Schwanghart, W., Scherler, D., 2014. TopoToolbox 2 - MATLAB-based software for topographic analysis and modeling in Earth surface sciences. Earth Surf. Dyn. 2, 1–7. http://dx.doi.org/10.5194/esurf-2-1-2014.

Toy, T.J., Foster, G.R., Renard, K.G., 2002. Soil Erosion: Processes, Prediction, Measurement, and Control. John Wiley & Sons, New York.

Trobec, R., Vajtersic, M., Zinterhof, P., 2009. Parallel computing - Numerics, Applications and Trends. Springer, Londonhttp://dx.doi.org/10.1007/978-1-84882-409-6_1.

Weiss, M.A., 1995. Data Structures and Algorithmic Analysis, 2nd Edition. The Benjamin/Cummins Publishing Company, Redwood City, USA.

Wilson, J.P., Aggett, G., Deng, Y., Lam, C.S., 2008. Water in the Landscape: A Review of Contemporary Flow Routing Algorithms. In: Zhou, Q., Lees, B., Tang, G. (Eds.), Lecture Notes in Geoinformation and Cartography: Advances in Digital Terrain Analysis, pp. 213–236. http://dx.doi.org/10.1007/978-3-540-77800-4_12.

Yang, D., Kanae, S., Oki, T., Koike, T., Musiake, K., 2003. Global potential soil erosion with reference to land use and climate changes. Hydrol. Process. 17, 2913–2928. http://dx.doi.org/10.1002/hyp.1441.

Zhan, L., Qin, C. (2011). A graph-theory-based method for parallelizing the multiple-flow-direction algorithm, In: Proceedings of the IEEE International Conference on Spatial Data Mining and Geographical Knowledge Services (ICSDM), DOI:10.1109/ICSDM.2011.5969020, pp. 137–141.

Zhao, Z., Benoy, G., Chow, T.L., Rees, H.W., Daigle, J.L., Meng, F.R., 2010. Impacts of accuracy and resolution of conventional and LiDAR based DEMs on parameters used in hydrologic modelling. Water Resour. Manag. 24 (7), 1363–1380. http://dx.doi.org/10.1007/s11269-009-9503-5.