Massively parallel Landscape-Evolution Modelling using General Purpose Graphical Processing Units

A.S. McGough, S. Liang[1], M. Rapoportas[1], R. Grey[1] and G. Kumar Vinod[1]
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
stephen.mcgough@ncl.ac.uk

D. Maddy and A. Trueman
School of Geography, Politics and Sociology
Newcastle University
Newcastle upon Tyne, UK
{darrel.maddy, adam.trueman}@ncl.ac.uk

J. Wainwright
School of Geography
Durham University
Durham, UK
john.wainwright@durham.ac.uk

***Abstract*—As our expectations of what computer systems can do and our ability to capture data improves, the desire to perform ever more computationally intensive tasks increases. Often these tasks, comprising vast numbers of repeated computations, are highly interdependent on each other – a closely coupled problem. The process of Landscape-Evolution Modelling is an example of such a problem. In order to produce realistic models it is necessary to process landscapes containing millions of data points over time periods extending up to millions of years. This leads to non-tractable execution times, often in the order of years. Researchers therefore seek multiple orders of magnitude reduction in the execution time of these models. The massively parallel programming environment offered through General Purpose Graphical Processing Units offers the potential for multiple orders of magnitude speedup in code execution times. In this paper we demonstrate how the time dominant parts of a Landscape-Evolution Model can be recoded for a massively parallel architecture providing two orders of magnitude reduction in execution time.**

***Index Terms*—Parallel; LEM; GPU; CUDA;**

I. Introduction

Our ability to capture information about the world we live in and process this information has been growing at a phenomenal rate in recent years [1]. This has fuelled the desire to perform more realistic modelling of the world – far outstripping the ability for a single computer to keep pace. One such problem is that of modelling long-term landscape-evolution. Realistic models require high-resolution landscape representation (millions of sample points) and simulation of processes acting upon the landscape over thousands to millions of years in order to simulate the key landscape shaping processes such as erosion and deposition. A Landscape-Evolution Model (LEM), can be decomposed, for a single time step (normally an annual time interval), into the following stages:

- Determine the levels of precipitation on an area of land
- Calculate how much will contribute to runoff (i.e. accounting for losses due to interception, evapotranspiration and infiltration)
- Calculate which directions the runoff will flow across the land
- Accumulate the total runoff across each part of the landscape
- Compute the erosion (via water or mass-movement processes) and where this material will be deposited.

This process takes minutes to compute for a single time step and small area ($\sim$ 2 million cells), but becomes non-tractable when modelling over large temporal / spatial scales. To run the sequential model considered here for 1 million years over a data set of 2 million cells (far smaller than desired) is currently estimated to take around 390 days to complete.

Although a LEM can be readily formulated at a conceptual level, traditionally its implementation has required simplification of existing process knowledge in order to achieve manageable execution times. Reduced complexity models, such as CHILD [2], CEASAR [3] or LAPSUS [4] all make significant, but different, model process or scale compromises, to achieve run times of a few days or less. Inevitably these compromises lead to unrealistic model behaviour, which often requires parametization of physically meaningless tuning variables in order to maintain numerical stability. Clearly the ideal solution is to minimise the compromises by maximising computational efficiency. Nonetheless, modelling landscape-evolution at the scale of, for example, an entire river basin and over important landscape forming time-scale (i.e. millions of years), presents a significant computational challenge.

Past research into reducing the computational cost has focused on code optimisation, data input optimisation and alternative sequential algorithms (e.g. advances in algorithms for water flow routing [5]). However, these improvements fall short of the requirements needed for tractable run-times for the next generation of LEMs.

Parallel computing is one approach with the potential to reduce execution times by orders of magnitude. To parallelize the model between time steps would be difficult as flow-directions in one time step can be altered by erosion / deposition in the previous time step – leading to significant changes in the flow patterns which in turn lead to significant changes in erosion / deposition – though parallelization within each time step is comparatively easy. Conventional parallel computing facilities exploit coarse grained parallel programming techniques in which large sections of the original problem can

[1]Work carried out whilst studying at Newcastle University

be computed independently of each other reducing execution time by orders of magnitude. LEMs, however, exhibit a more close coupling.

Over the past decade the graphics card industry has been developing massively parallel processing cards for the gaming industry. Primarily developed for rendering 3D scenes onto a display, these cards have evolved to become arrays of hundreds of computational units. The Fermi architecture cards from NVIDIA, feature up to 512 General Purpose Graphical Processing Units (GPGPUs) – offering levels of parallelism previously unseen on a single card, or computer.

Researchers are now exploiting these cards to solve large scale, closely coupled, problems. Here we show how we have exploited this technology to achieve over two orders of magnitude execution-time reduction for the two most time-consuming elements, those of flow-direction and flow-accumulation, within an existing LEM.

The rest of this paper is structured as follows. In Section II we describe and analyse through profiling, an existing sequential LEM and identify the flow-routing and flow-accumulation methods as the major bottle-necks in computation which restricts the scope of grid-based LEMs. Section III discusses previous attempts to address the flow-routing and accumulation problems. Section IV discusses massively parallel processing including a description of the General Purpose Graphical Processing units and the Compute Unified Device language. This is followed, in Section V by a description of our parallel approaches to reducing the execution times for the flow direction and flow accumulation methods. Performance results for our work are presented in Section VI. Finally we provide conclusions in Section VII.

II. Description and Analysis of an Existing Sequential LEM

We aim to reduce the execution time of the LEM software developed by Wainwright [6] – referred to here as CybErosion. CybErosion models the evolution of a landscape over a period of time in response to erosion and sediment depositional processes (i.e. the sediment flux). Both fluvial and slope-system dynamics are modelled as a response to changing model inputs (temperature and precipitation) and as a consequence of system feedbacks. The model inputs vary through time and as the landscape is reshaped, the boundary conditions for each model run also vary. Thus the LEM is run iteratively with each iteration representing a defined time interval appropriately matched to the input data. Due to the long periods of time over which the simulations will run these are normally on the order of a year, though this need not be the case and ideally would be much less.

A. Processing Pipeline for CybErosion

CybErosion consists of five main sequentially executed stages (Figure 1), described further below:

1) **Data input** – read digital landscape representation and climate parameters

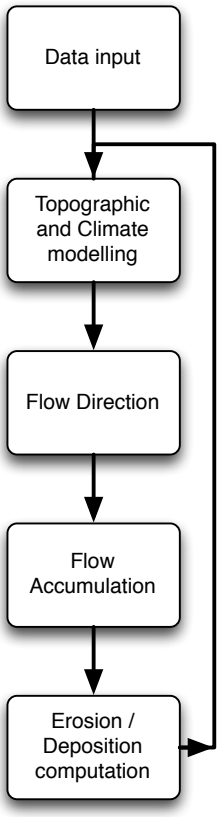


Fig. 1. Processes stages in performing a LEM

2) **Topographic and Climate modelling** – compute the volume of water which falls on the landscape
3) **Flow-direction calculation** – calculate water-flow-directions across the landscape
4) **Flow-accumulation** – calculate flow-accumulation across the landscape
5) **Sediment erosion, transport and deposition computation** – model erosion, sediment transport and depositional processes and recalculate the modified landscape.

Data Input: The availability of grid-based Digital Elevation Models (DEMs) has made regular two dimensional grids of cells the preferred landscape representation for LEMs. Alternative approaches discretize the landscape using a more irregular representation (CASCADE [7]), such as triangulated irregular networks (CHILD [2]). CybErosion adopts the more common format representing the landscape as a regular two dimensional grid of cells, each containing a single value representing the height of the land in that cell – we continue to use this approach. Ideally cells should be as small as possible to capture surface detail. However, the infeasibility of capturing accurate data at such resolution and the infeasibility to computationally model such landscapes has led to the use of large cells - on the order of 100×100 m for small areas ranging up beyond 1×1 km for larger areas.

Figure 2 illustrates the DEMs used in this paper (derived from OS Land-Form PROFILE DTM (2010) data[8]): region

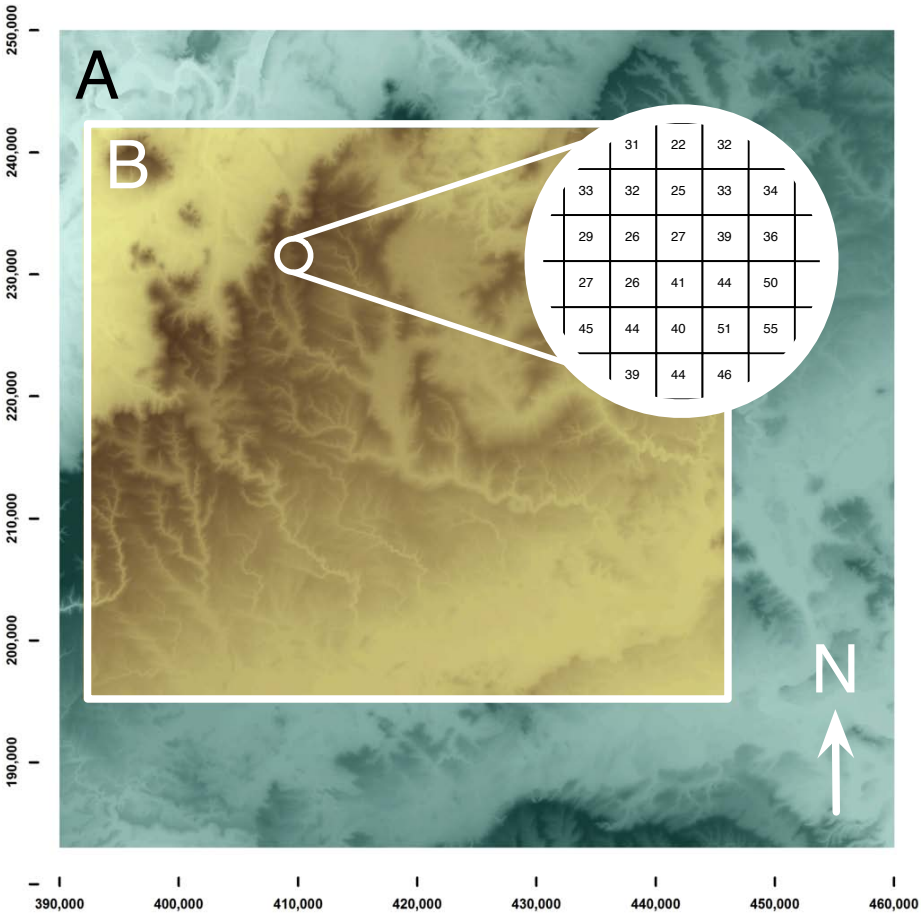


Fig. 2. A Digital Elevation Model for the Upper Thames and adjacent watersheds (scale in meters)

A represents the 46 million cell DEM at 10 m by 10 m with a height range of 9 to 331 m; and region B the 25 million cell DEM at 10 m by 10 m with height range of 12 to 331 m. Down-sampling of the 25 million cell DEM was used to produce the 11 million cell DEM at 15 m by 15 m; the 6 million cell DEM at 20 m by 20 m; and the 1 million cell DEM at 50 m by 50 m. For brevity the heights are shown as integers, the real values are double precision floating point values. It should be noted that past experiments have indicated that using single precision floating point numbers tends to lead to significant rounding errors which quickly propagate through the DEM leading to unnatural artefacts.

Model data such as the initial height of cells within the DEM, temperatures and the rainfall patterns are read at this stage. Originally CybErosion used flat ASCII files to represent this data. Alternatively the DEM can be read from a compressed TIFF file using the GDAL [9] library. This has advantages in both storage space requirements and time to read (store) data. We have added GDAL support to both the original CybErosion model and our parallel implementation.

Topographic and Climate modelling: Initially each cell is assigned values associated with its surface properties including sediment calibre, soil water content and biomass, with soil erodibility based on the k factor [10]. These constrain some of the physical processes modelled in *Sediment erosion, transport and deposition computation*, but also evolve through time in response to changing model inputs and as a dynamical feedback response to the geomorphological processes acting upon the cell. Hence, these properties are updated every time step. As the focus for this paper is parallel algorithm development the LEM temperature and precipitation inputs are considered constant at 15°C and 327 mm respectively and applied uniformly across the DEM.

Flow-direction calculation: Here the direction that water will flow from a cell is computed, known as the cell aspect. Two common approaches are single-flow-direction (SFD) (referred to as D_8) and multi-flow-direction (MFD) [11], [12], in both cases cells consider their eight neighbours when determining flow-direction. CybErosion uses SFD, where the entire flow goes to the neighbour with the steepest downwards slope (computed as fall / distance) – 'steepest descent' method – where distance is the scalar distance between the centres of two cells. For the cardinal directions this is the DEM 'cell size' whilest other directions are $\sqrt{2}\times$'cell size'. For MFD a proportion of the flow will go to each lower neighbour relative to the slope differences between all lower neighbours. Powers of 2 are used to indicate aspects, allowing for storage of MFD, starting with East as 2^0 incrementing clockwise (Figure 3).

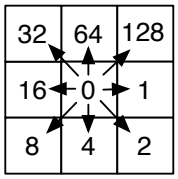


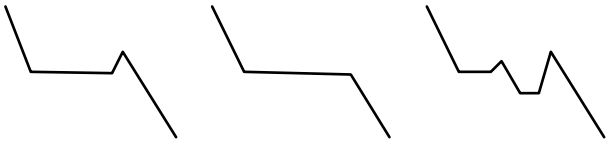
Fig. 3. Flow Numbering

Cells with no lower neighbours are part of a sink or plateau and are assigned an aspect of 0.

Sinks are areas in the DEM which entering water cannot leave (Figure 4(a)), plateaux are areas of equivalent height cells where water can leave (Figure 4(b)). Sinks and plateaux can be natural or artefacts of data acquisition and/or processing. Completely flat plateaux do not occur naturally, but occur in DEMs as a result of measurement precision and height averaging at comparatively low spatial resolution. To route water from a plateau we can assign arbitrary aspects allowing water flow to cells at the edge of the plateau with lower elevations. Sinks can be removed by 'flooding' an area with water until it becomes a plateaux, but as sinks may interact with each other (Figure 4(c)) this is a non-trivial task. CybErosion, originally designed for synthetic DEMs, does not support sink filling, however, we address this problem in our parallel version.

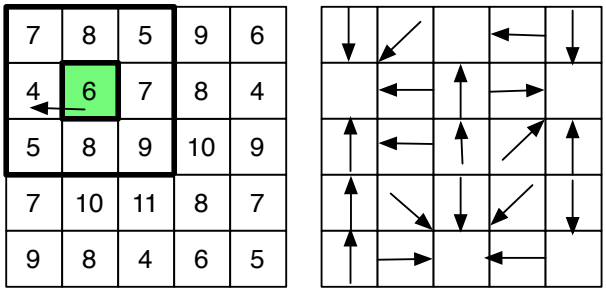
An example of SFD is given in Figure 5. Figure 5(a) indicates the height elevations for a 5 by 5 grid. The flow-direction for the highlighted cell (6) is computed by identifying the neighbour with steepest decent – the 4 immediately to the left ($decent = (6-4)/$*'cell size'*). Figure 5(b) indicates the flow-directions for all the cells in this grid. Sequential code for solving the flow-direction problem, both SFD and MFD, requires O(n) time to compute, where n is the number of cells within the DEM.

Flow-accumulation: We need to compute the amount of water flowing across each cell. In-coming precipitation is weighted for each cell using a run-off coefficient calculated from the surface properties. At the watersheds the weighted flow is the product of the water which lands on those cells and its weighting. Further down the valley this will be the cumulative sum of all grid cells for which their flow-direction path takes them through the given cell. Figure 6, continues our example, indicating how much water would flow through each cell in our grid assuming unit runoff (i.e. *runoff = precipitation*). The yellow cells feed into the orange cell giving a total flow across the orange cell of 8.



(a) Sink (b) Plateau (c) Interacting sinks

Fig. 4. 2D View



(a) (b)

Fig. 5. Flow-direction modelling

1	1	4	1	1
8	1	2	1	5
4	1	1	1	1
2	1	1	1	1
1	1	6	1	2

Fig. 6. Flow-accumulation

Sediment erosion, transport and deposition computation: The final phase uses well-established equations ([6]) to determine the amount of erosion (including diffuse hillslope and concentrated channel erosion) or deposition at a location based on the flow across the cell. Although we anticipate that this stage of the simulation would show significant improvement through parallelisation this is yet to be implemented.

B. CybErosion Runtimes and Profiling

Wainwright [6] reports run times of 72 hours for an 800,000-year simulation on a Pentium 4 processor using a synthetic DEM (smoothed numerical surface) comprising 51 by 100 cells of size 100×100 m using CybErosion. Optimisation of this code, referred to as CybErosion-slim, has reduced run time to $\sim$4.5 hours on an Intel 980X processor. However, this grid size is wholly inadequate for modelling real catchments at high spatial resolution.

Figure 7 illustrates the number of seconds spent in each major function within CybErosion-slim for various input DEMs. The DEMs were either synthetically generated, containing no sinks or plateaux, or by resampling the topographical map of the Upper Thames catchment area (see Figure 2) at progressively lower resolution. The Upper Thames DEMs contain plateaux but sinks were removed as CybErosion(-slim) does not process them. As the size of the synthetic DEM becomes larger, the flow-accumulation part dominates execution time. However, when using the real data from the Upper Thames the plateau routing code execution time becomes significant. CybErosion(-slim) was unable to correctly process the flow-accumulation for the Thames DEM, hence the almost zero

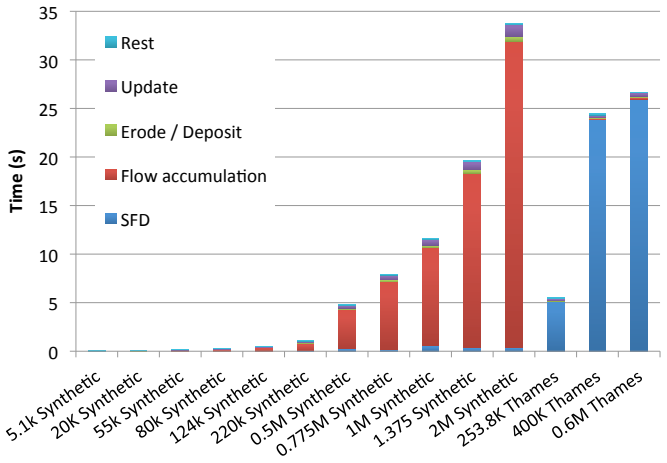


Fig. 7. Performance analysis of CybErosion

execution time in these cases, the results were included here only to illustrate that a real world DEM significantly increases the flow-direction computation of the model. We focused our attention on massively parallel solutions to both of these processes.

III. Addressing the Flow Routing and Accumulation Problem on Large Grids

Sequential algorithms for deriving flow routing and accumulation from DEMs have been available for over 20 years [13], [11] and strategies to address sink and plateau problems have been widely reported during this interval [14], [15], [16], [17]. Alternate routing methods, based upon MFD, have been proposed such as DEMON [18] and D_∞ [19]. The relative merits of these methods were discussed by Erskine [20] and although it is acknowledged that MFD produces more realistic drainage patterns in certain areas, the computational overhead introduced, especially for large DEMs, has seen it largely ignored in favour of the more efficient SFD. The influence of routing choice on LEM outcome is, however, unequivocal [21].

Flow routing and flow-accumulation are common methods required for hydrological modelling and form a regular addition to the toolboxes of modern Geographical Information Systems such as ArcGIS and GRASS. Considerable effort has been undertaken to maximise these methods when applied to large grids. Perhaps the most influential work was undertaken by a team from Duke University, which concentrated on developing an efficient I/O system allowing large grids to be efficiently processed on machines with comparatively small amounts of RAM [5], [22]. This work powers one of the most optimised tools for this type of hydrological analysis, Terraflow [23], most commonly used through the r.terraflow tool implemented within GRASS [24]. However, this have come at the cost of coding complexity and thus far none of the available LEMs utilizes these methods.

Parallel processing has not gone unnoticed by the hydrological community. Recent papers have suggested appropriate strategies for parallel implementation [25], [26], [27]. Ortega and Reuda [28] described a flow-accumulation algorithm written to utilize CUDA on NVIDIA GPGPUs, reporting up to 8 times speedup for flow-accumulation in comparison to a sequential version of the same algorithm on grids up to 16 million cells.

IV. Massively Parallel Programming

The General Purpose Graphical Processing Unit (GPGPU) is a massively parallel programming architecture based around the Single Programme Multiple Data (SIMD) paradigm. Each thread within a kernel executes the same program and has access to a (relatively) small data cache. Execution is organised into 'Warps', within which up to 32 threads of execution perform the same instruction at the same time with code divergence causing warps to split. Thousands of concurrent threads are hardware managed allowing for massively parallel execution. This does, however, remove the ability to interact between threads, as different threads may be scheduled at different times. Synchronisation is only possible between threads in the same block – a collection of up to 1024 threads of execution.

In comparison to a conventional CPU, a GPGPU is a much simpler device consisting of a computation and a floating point unit (Figure 8). Although early GPUs had inaccurate floating point units, later generations (since Nvidia G80, 2006) support the IEEE floating point standard, increasing their applicability to scientific computations. GPGPUs also provide significantly higher memory bandwidth in comparison to CPU memory bandwidth ($\sim 9\times$ for the GTX580 GPGPU verses i7-3930K CPU).

The Compute Unified Device Architecture (CUDA) [29], [30] provides extensions to C / C++ for developing kernels, along with libraries for binding GPGPU and CPU code. As the GPGPU lacks support for global barrier synchronization and inter-process communication, kernels tend to be short to allow fallback to the main CPU for exchange of data.

V. Parallel Landscape-Evolution Modelling

Our current parallel implementation has focused on the two most time consuming parts of the existing LEM. Those of flow-direction assignment and flow-accumulation. Figure 9 illustrates the stages of the CyberErosion model that we have

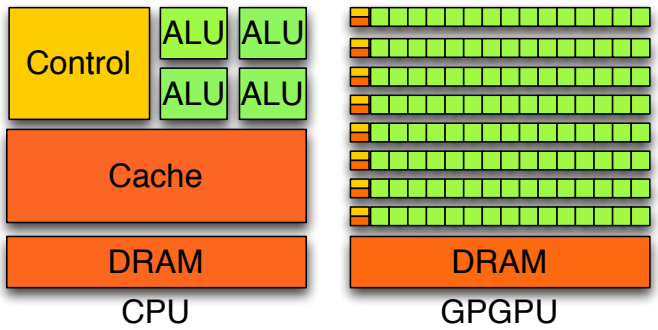


Fig. 8. Comparison of CPU and NVIDIA GPGPU Architecture

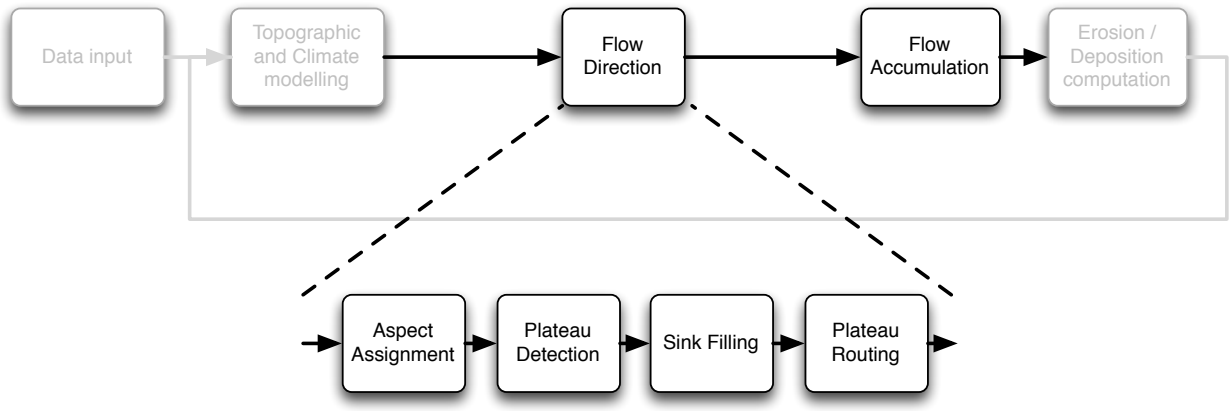


Fig. 9. Processes stages in performing the parallel LEM

parallelised (the highlighted stages). The second line of Figure 9 illustrates the extra sub-stages we have added into the flow-direction code in order to process sinks. We describe each of these processes in more detail below.

A. Flow-Direction

Flow-direction (c.f. Section II) consists of four sub-stages: aspect assignment, plateau detection, (where necessary) sink filling and plateau routing. Plateau detection and plateau routing are almost the same. To prevent plateaux being processed as sinks they must be first identified. However, once filled, sinks become plateaux. We therefore run a simplified version of the plateau-routing algorithm as the plateau detection algorithm and consider it no further here.

1) Aspect Assignment: The aspect of a cell can be determined independently of all other cells for both SFD (D_8) or MFD. Although the number of active threads on a GPGPU is much smaller than the size of a DEM, the number of virtual threads is far greater, allowing us to allocate one virtual thread per cell with the GPGPU taking responsibility for optimally scheduling these virtual threads. Algorithm 1 presents pseudo code for the kernel code to calculate SFD. Where c and r represent the column and row within the DEM, $aspect[c,r]$ is the cell aspect, $slope$ is the downslope of the cell, $cellSize$ is the size of a cell and $\delta_c[i]$ & $\delta_r[i]$ represent the change in c & r when moving in direction 2^i. Aspect assignment can be performed in $O(n/p)$ time where n is the number of cells within the DEM and p is the number of processing units. MFD is similar to Algorithm 1 with addition of proportioning slope and multiple aspects.

2) Plateau Routing: We adapt the sequential algorithms proposed by Arge [22]. A parallel breadth-first search algorithm [31] is used to perform plateau routing. The kernel in Algorithm 2 is repeatedly called until no cell updates its value for $shortestPath$. Before execution, $shortestPath$ is set to 0 for cells with a non-zero aspect, ∞ otherwise. The height to which cell $[c,r]$ will flow when it leaves the plateau is stored in $lowHeight$. Figure 10 illustrates a routing towards two exit points (grey cells) for a plateau. Alternative routing can emerge from this approach – the flow pattern in Figures 11(a) and 11(b) are each of the same length. Sequential implementations of breadth first search would normally only show one of these solutions, however, due to the non-deterministic ordering of kernel invocations both solutions are observed. Either solution is valid as there is no way to determine which is more appropriate from a hydrological point of view, we are happy to accept this result. Each kernel invocation for plateau routing will require $O(n/p)$ time to perform. However, the number of calls to this kernel depends on the longest path within a plateau.

Algorithm 1 Calculate Aspect D_8 (c,r)

$aspect[c,r] \leftarrow 0$
$maxDiff \leftarrow 0$
for $i = 0 \rightarrow 7$ **do**
  $r' \leftarrow r + \delta_r[i]$
  $c' \leftarrow c + \delta_c[i]$
  $d \leftarrow (i\%2 = 1)\ ?\ 1 : \sqrt{2}$
  if $(z[c,r] - z[c',r'])/(d * cellSize) > maxDiff$ **then**
    $slope[c,r] \leftarrow (z[c,r] - z[c',r'])/(d * cellSize)$
    $aspect[c,r] \leftarrow 2^i$
    $maxDiff \leftarrow slope[c,r]$
  end if
end for

Algorithm 2 Calculate Plateau Routing (c,r)

for $i = 0 \rightarrow 7$ **do**
  $d \leftarrow (i\%2 = 1)\ ?\ 1 : \sqrt{2}$
  if $shortestPath[c + \delta_c[i], r + \delta_r[i]] + d < shortestPath[c,r]$ **then**
    $aspect[c,r] \leftarrow 2^i$
    $shortestPath[c,r] \leftarrow shortestPath[c + \delta_c[i], r + \delta_r[i]] + d$
    $lowHeight[c,r] \leftarrow lowHeight[c + \delta_c[i], r + \delta_r[i]]$
  end if
end for

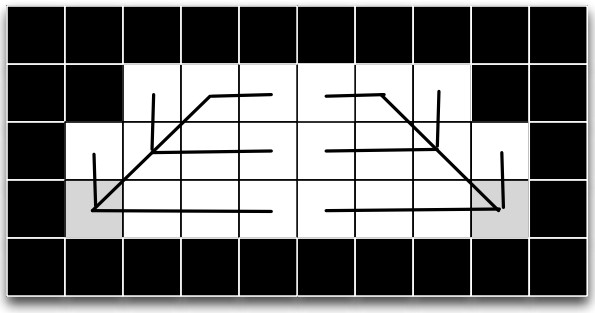
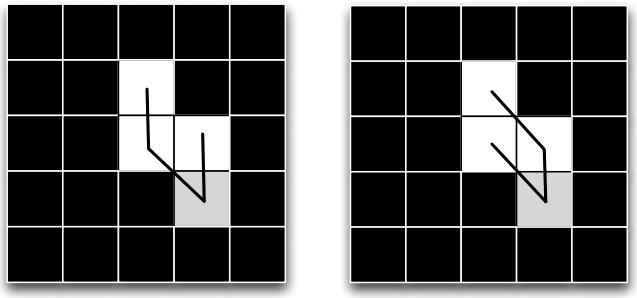


Fig. 10. Example: Breadth-first search plateau routing



(a) (b)

Fig. 11. Alternative Routing

3) Sink Filling: Our approach is based on the work by [5], [22] and comprises: (i) delineating all watersheds (the boundary of an area of land for which all water will drain to the same point), (ii) construct a undirected weighted graph to represent the relationships between these watersheds (the lowest point at which water can flow from one watershed to another), and (iii) raise the height of each watershed until all watersheds can flow out of the DEM.

The first step of delineating watersheds is achieved by assigning a unique $watershedID$ value to each sink cell and a $watershedID$ value of 0 to cells flowing out of the DEM. As most sinks consist of more than one cell we need to consolidate each sink into just one $watershedID$. This is achieved by repeatedly calling the kernel presented in Algorithm 3 until no $watershedID$'s are updated. Each kernel call will require $O(n/p)$ time, with the number of calls bounded by the longest dimension of the largest sink.

Algorithm 3 Delineate sinks (c,r)

for $i = 0 \rightarrow 7$ **do**
  $r' \leftarrow r + \delta_r[i]$
  $c' = c + \delta_c[i]$
  if $z[c', r'] = z[c, r]$ AND $watershedID[c, r] > watershedID[c', r']$ **then**
    $watershedID[c, r] \leftarrow watershedID[c', r']$
  end if
end for

Algorithm 4 Propagate watersheds (c,r)

if $watershedID[c, r]$ *is set*
  then return
end if
if $watershedPointer[c, r]$ *is unset* **then**
  $r' \leftarrow r + \delta_r[log_2(aspect[c, r])]$
  $c' = c + \delta_c[log_2(aspect[c, r])]$
  $watershedPointer[c, r] \leftarrow [c', r']$
end if
if $watershedID[watershedPointer]$ *is set* **then**
  $watershedID[c, r] \leftarrow$
    $watershedID[watershedPointer]$
else
  $watershedPointer[c, r] \leftarrow$
    $watershedPointer[watershedPointer[c, r]]$
end if

Watershed delineation is completed by propagating the $watershedID$ for a sink to all cells flowing to that sink. The kernel presented in Algorithm 4 uses index pointer jumping [32] to compute this in $O(log(j))$ kernel calls where j is the length of the longest path within a watershed and each kernel call taking $O(n/p)$ time.

We now generate the triples $\{w1, w2, h\}$, where $w1$ and $w2$ are $watershedID$'s and h is the lowest boundary between the two watersheds, for all touching watersheds. We use a hash table to store these triples as there may be thousands of watersheds at this stage. Algorithm 5 describes the kernel for identifying watershed boundaries and storing these within the hash table. The Fermi architecture does not support an atomic minimum operation on double precision floating point numbers, required for this kernel. However, it is possible to

Algorithm 5 Identify watershed boundaries (c,r)

for $i = 0 \rightarrow 7$ **do**
  $r' \leftarrow r + \delta_r[log_2(aspect[c, r])]$
  x$c' = c + \delta_c[log_2(aspect[c, r])]$
  if $watershedID[c, r]! = watershedID[c', r']$ **then**
    if $z[c, r] \geq z[c', r']$ **then**
      if $\{watershedID[c, r], watershedID[c', r'], h\}$
        or
        $\{watershedID[c', r'], watershedID[c, r], h\}$
        is already in the hash table
        then
          Store the minimum of $(z[c, r], h)$ at this location in the hash
      else
        Add $\{watershedID[c, r],$
        $watershedID[c', r'], z[c, r]\}$ to the hash table
      end if
    end if
  end if
end for

implement an atomic minimum operation using the atomic compute and exchange operation. As there is no global synchronisation primitive it is possible for two (or more) threads to request storage space for the same pair of $watershedID$s, thus duplicating storage. However, as the next stage is to sort the hash table, these duplicates can easily be removed.

The sequential algorithm presented by Arge [22] is used to compute, for each watershed i, the height to which all cells in watershed i must be raised ($raised[i]$) in order to allow all water to flow out of the DEM. The process of raising cells within the watersheds can now be performed in parallel in $O(n/p)$ time. This kernel sets $z[c,r]$ to the maximum of $z[c,r]$ and $raise[watershedID[c,r]]$. Thus creating plateaux in place of each of the sinks. Running the plateau routing algorithm over the DEM will now give us a fully conditioned (routed) DEM.

B. Flow-Accumulation

The Correct Flow kernel, Algorithm 6, is used to compute flow-accumulation. The available runoff from each cell is first placed into $resultGrid[c,r]$ and $correct[c,r]$ is set to false for all cells. The function $inv()$ inverts a direction (e.g. $inv(64) = 4$). The kernel works by only allowing a cell to compute its flow-accumulation when all flowing-in neighbours are marked as correct (true if no cells flow in). The kernel is repeatedly called until $correct[c,r]$ is true for all cells. Kernel calls requires $O(n/p)$ time with the longest path in the DEM determining the number of iterations.

VI. Results

A. Deployment environment

The test environment for this work consists of an Asus P6T7 WS Supercomputer motherboard populated with an Intel i7 980X Extreme 6 core processor along with 12GB corsair dominator DDR3 12800 RAM. The graphics processors are a 3GB Gainward Nvidia GTX580 (512 cores) along with a 448 core, 3GB C2050 Tesla card. The system ran Windows 7 running from a OCZ-AGILITY3 ATA SSD.

Algorithm 6 Correct Flow (c,r)

```
accumulation ← 0
for i = 0 → 7 do
  r' ← r + δ_r[i]
  c' ← c + δ_c[i]
  d ← (i%2 = 1) ? 1 : √2
  if aspect[c', r'] & inv(2^i) = inv(2^i) then
    if correct[c', r'] = false
      then return
    end if
    accumulation ← accumulation+
      resultGrid[c', r']
  end if
end for
resultGrid[c, r] ← accumulation
correct[c, r] ← true
```

Simulations were run for the 1, 6, 11, 25 and 46 million cell DEMs, for the parallel case along with equivalent direction and accumulation methods as implemented in the r.terraflow[23] code incorporated within GRASS 6.4. As our optimised sequential code (CybErosion-slim) was incapable of running such large 'real' DEMs, synthetic DEMs were produced at sizes of 5,100, 253,800, 397,488, 601,644 and 1 million cells. These synthetic DEMs were designed to contain no sinks as these cannot be handled by CybErision(-slim). Thus, the sequential results are an underestimate for the time to process a real DEM. To aid comparison, the 5,100 cell synthetic DEM was also run through the parallel implementation.

B. Overall execution time

Figure 12 illustrates the execution time for CybErosion-slim and our massively parallel model, showing a reduction of execution time of over two orders of magnitude. For extremely small grids (~5,100 cells), CybErosion-slim is able to outperform the parallel model – due to the overheads of the parallel implementation. However, this quickly reverses for 'real' sized DEMs. The average time for running the first 10 iterations of the parallel code is lower in all cases than the single iteration case, due to two main factors: the overheads for running the simulation are associated with the first and last iterations; and the original Thames watershed contains a large number of sinks and plateaux which are progressively removed in the parallel implementation through erosion and deposition during the first few iterations. This latter effect can be seen in Figure 13 in which the flow-accumulation remains constant across the ten iterations whilst the flow-direction stage is much higher for the first two iterations before becoming a more constant small value. This can also be observed from the number of watersheds (before flooding) whereby in the case of the 25 million cell DEM this decreases from 14,703 in iteration one to 2,834 by iteration four.

In all cases, the GTX580 outperforms the Tesla. Although the Tesla is capable of performing double precision floating point operations at full clock speed, as opposed to the quarter speed of the GTX580, the extra cores, higher core clock and memory speed seems to compensate for this limitation. Al-

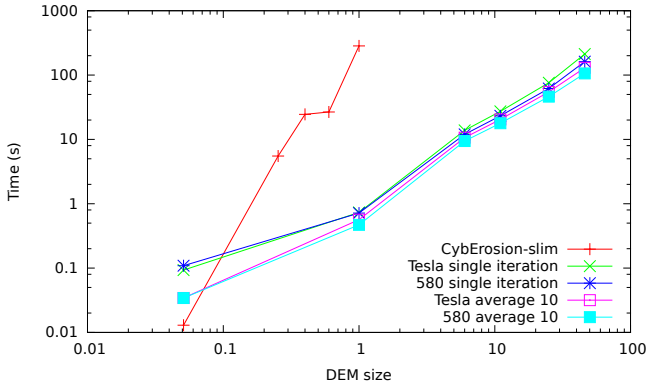


Fig. 12. Overall execution time

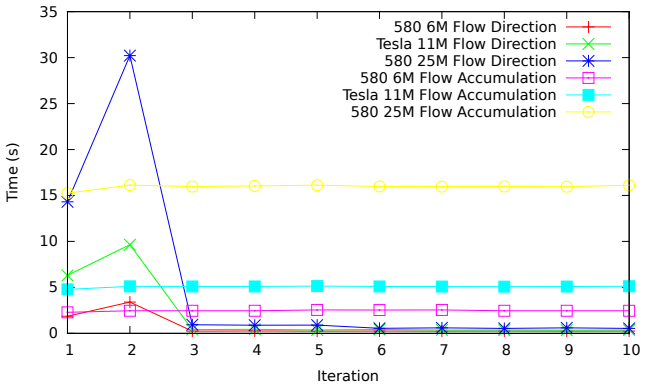


Fig. 13. Time per Iteration

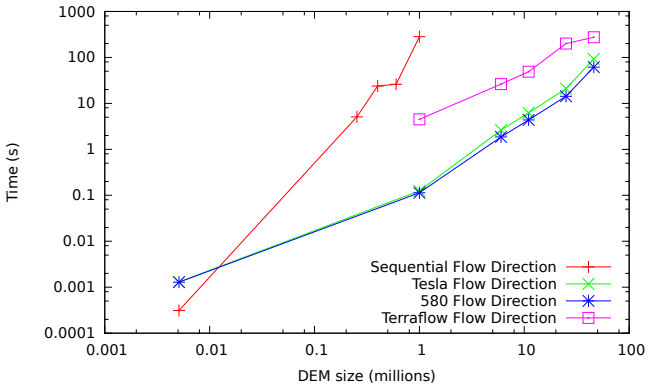


Fig. 14. Flow-Directions

though the code uses double precision floating point numbers, a substantial proportion of the code does not, hence the Tesla card provides no advantage in this case.

C. Flow-Direction Modelling

We investigate the performance of the Flow-direction modelling part of our parallel simulation in Figure 14. The parallel implementation shows much better performance than CybErosion-slim, exhibiting between two and three orders of magnitude difference. CybErosion(-slim) does not process sinks, and has a crude plateau-processing system, thus the parallel implementation is performing more work. The parallel code exhibits nearly two orders of magnitude reduction in execution time over r.terraflow for the one million cell DEM. However, this reduces as the size of the DEM increases. This warrants further investigation.

D. Flow-Accumulation

The performance of our parallel implementation of flow-accumulation is presented in Figure 15. For all but the smallest (5,100 cell) DEM, the parallel code outperforms CybErosion-slim. The parallel model outperforms r.terraflow, though by less than an order of magnitude. This lower than expected performance improvement can be attributed to the large number of iterations required by the Correct flow algorithm to complete.

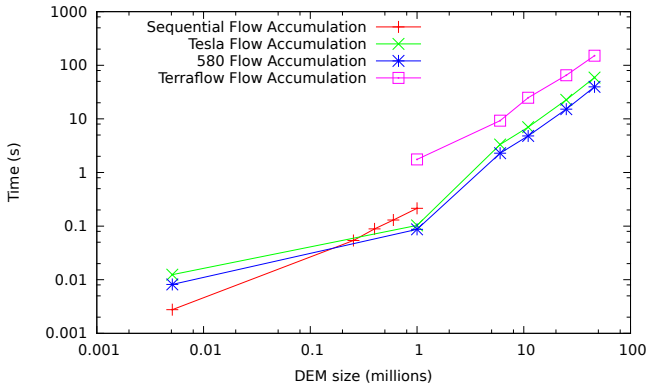


Fig. 15. Flow-Accumulations

Figure 16 exemplifies this problem for the 46 million cell DEM. It takes only 293 iterations to achieve 99% of cells correct, however, it then takes a further 17,118 iterations to mark the remaining cells correct. This is due to long runs of cells within the DEM, for example down the valley axes, in which the algorithm becomes effectively sequential, processing just one more cell down the axis per iteration.

Although the floating point arithmetic standard [33] is conformed to by both the computer CPU and the GPGPU the actual result values need not be identical due to alternative interpretations of the standard and the lee-way in implementations. Along with the uncertainty of ordering of kernel threads on the GPGPU it is not possible to generate exactly the same results. However, floating point differences appear to be insignificant when using double precision and as there is no way to determine if one ordering of the kernel threads – leading to one routing pattern over a flat – is more 'correct' than another these issues are tolerable.

VII. Conclusion

We have shown here that the majority of time in CybErosion-slim is spent in computing the flow-directions and flow-accumulations. We have developed a number of massively parallel algorithms to dramatically reduce the execution

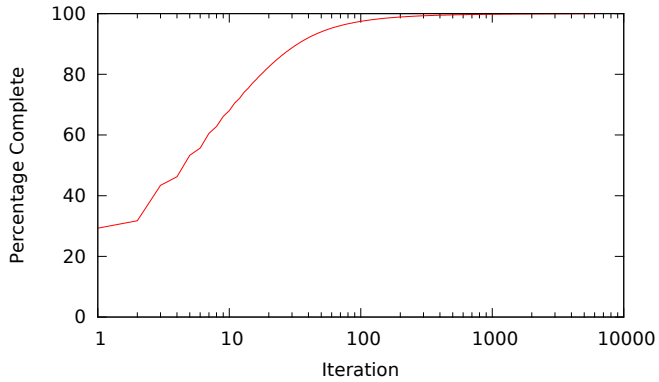


Fig. 16. Correct cells per iteration

times for these parts of the program. We have seen over two orders of magnitude reduction in execution time for the model, alongside a single order of magnitude reduction in comparison to the most optimal sequential model (r.terraflow). This will allow us to run our LEM at much greater temporal / spatial scale, without the need to reduce model complexity. Additionally we will be able to add further to the model thus producing a more realistic simulation.

We are aware of limitations in the current parallel code, such as the long tail problem of the Correct flow process, and are seeking alternative parallel algorithms to alleviate this.

So far we have only parallelised two of the four major components within the process pipeline. The two remaining components (topological and erosion modelling and deposition computation), although numerically complex, require little interaction between cells. Hence, parallelisation of these components should yield good performance improvements.

Differences between floating point units in the CPU and GPGPU along with the effects of the alternative ordering of threads warrant further investigations. Recent developments of the CUDA toolkit allow parallel code to be compiled for the CPU which could lead to similar speedups.

It is interesting to note that although the Tesla card is capable of full speed double precision floating point operations, in this modelling problem this additional performance is more than offset by the extra cores and faster clock speed of the cheaper (by a factor of $\frac{1}{5}$) GTX580 card.

REFERENCES

[1] A. J. G. Hey and A. E. Trefethen, "The data deluge: An e-science perspective," in *Grid Computing - Making the Global Infrastructure a Reality*, F. Berman, G. C. Fox, and A. J. G. Hey, Eds. Wiley and Sons, 2003, pp. 809–824, chapter: 36. [Online]. Available: http://eprints.soton.ac.uk/257648/

[2] G. Tucker, S. Lancaster, N. Gasparini, and R. Bras, "The channel-hillslope intergrated landscape development model (child)," in *Landscape erosion and evolution modelling*, H. S. and D. W, Eds. Kluwer, 2001, pp. 349–388.

[3] T. J. Coulthard, M. J. Kirkby, and M. G. Macklin, "Non-linearity and spatial resolution in a cellular automaton model of a small upland basin," *Hydrology and Earth System Sciences*, vol. 2, no. 2/3, pp. 257–264, 1998. [Online]. Available: http://www.hydrol-earth-syst-sci.net/2/257/1998/

[4] J. Schoorl, M. Sonneveld, and A. Veldkamp, "3d landscape process modelling: the effect of dem resolution," *Earth Surface Processes and Landforms*, vol. 25, pp. 1025–1034, 2000.

[5] L. Arge, L. Toma, and J. S. Vitter, "I/o-efficient algorithms for problems on grid-based terrains," *J. Exp. Algorithmics*, vol. 6, Dec. 2001. [Online]. Available: http://doi.acm.org/10.1145/945394.945395

[6] J. Wainwright, "Degrees of separation: Hillslope-channel coupling and the limits of palaeohydrological reconstruction," *CATENA*, vol. 66, no. 1-2, pp. 93 – 106, 2006.

[7] J. Braun and M. Sambridge, "Modelling landscape evolution on geological time scales: a new method based on irregular spatial discretization," *Basin Research*, vol. 9, no. 1, pp. 27–52, 1997. [Online]. Available: http://dx.doi.org/10.1046/j.1365-2117.1997.00030.x

[8] Ordnance Survey, GB. Using EDINA Digimap Ordnance Survey Service, "OS Land-Form PROFILE DTM [ASC geospatial data], Scale 1:10000, NE(514005,269005) SW(389995, 159995), Updated August 2003," http://edina.ac.uk/digimap, June 2010.

[9] "GDAL - Geospatial Data Abstraction Library," http://www.gdal.org/.

[10] X. Zhang, N. Drake, and J. Wainwright, "Scaling land surface parameters for global-scale soil erosion estimation," *Water Resour. Res.*, vol. 38, no. 9, 09 2002. [Online]. Available: http://dx.doi.org/10.1029/2001WR000356

[11] S. K. Jenson and J. O. Dominque, "Extracting topographic structure from digital elevation data foar geographic information system analysis," *Photogrammetric Engineering and Remote Sensing*, vol. 54, no. 11, 1988.

[12] T. Freeman, "Calculating catchment area with divergent flow based on a regular grid," *Computers & Geosciences*, vol. 17, no. 3, pp. 413 – 422, 1991.

[13] J. F. O'Callaghan and D. M. Mark, "The extraction of drainage networks from digital elevation data," *Computer Vision Graphics and Image Processing*, vol. 28, no. 3, pp. 323–344, 1984.

[14] L. W. Martz and J. Garbrecht, "The treatment of flat areas and depressions in automated drainage analysis of raster digital elevation models," *Hydrological Processes*, vol. 12, no. 6, pp. 843–855, 1998.

[15] L. W. Martz and J. Gabrecht, "An outlet breaching algorithm for the treatment of closed depressions in a raster dem," *Computers & Geosciences*, vol. 25, no. 7, pp. 835 – 844, 1999.

[16] P. Soille, J. Vogt, and R. Colombo, "Carving and adaptive drainage enforcement of grid digital elevation models," *Water Resources Research*, vol. 39, 2003.

[17] A. Temme, J. Schoorl, and A. Veldkamp, "Algorithm for dealing with depressions in dynamic landscape evolution models," *Computers & Geosciences*, vol. 32, no. 4, pp. 452 – 461, 2006.

[18] M. C. Costa-Cabral and S. J. Burges, "Digital elevation model networks (demon): A model of flow over hillslopes for computation of contributing and dispersal areas," *Water Resour. Res.*, vol. 30, no. 6, pp. 1681–1692, 1994. [Online]. Available: http://dx.doi.org/10.1029/93WR03512

[19] D. Tarboton, "A new method for the determination of flow directions and uplose areas in grid digital elevation models," *Water Resources Research*, vol. 33, no. 2, pp. 309–319, 1997.

[20] R. H. Erskine, T. R. Green, J. A. Ramirez, and L. H. MacDonald, "Comparison of grid-based algorithms for computing upslope contributing area," *Water Resour. Res.*, vol. 42, no. 9, 09 2006. [Online]. Available: http://dx.doi.org/10.1029/2005WR004648

[21] J. D. Pelletier, "Persistent drainage migration in a numerical landscape evolution model," *Geophys. Res. Lett.*, vol. 31, no. 20, 10 2004. [Online]. Available: http://dx.doi.org/10.1029/2004GL020802

[22] L. Arge, J. S. Chase, P. Halpin, L. Toma, J. S. Vitter, D. Urban, and R. Wickremesinghe, "Flow computation on massive grid terrains," *GEOINFORMATICA*, vol. 7, p. 2003, 2001.

[23] Terraflow project team, "Terraflow project," http://www.cs.duke.edu/geo*/terraflow/, 1999.

[24] M. Neteler and H. Mitasova, *Open source GIS: a GRASS GIS approach*, M. Neteler and H. Mitasova, Eds. Kluwer Academic Pub, 2002, vol. 689.

[25] C. Wallis, D. Watson, D. Tarboton, and R. Wallace, "Parallel flow-direction and contributing area calculation for hydrology analysis in digital elevation models." in *PDPTA*, H. R. Arabnia, Ed. CSREA Press, 2009, pp. 467–472.

[26] R. Wallace, D. Tarboton, D. Watson, K. Schreuders, and T. Tesfa, "Parallel algorithms for processing hydrologic properties from digital terrain," in *GIScience*, 2010.

[27] L. Zhan and C. Qin, "A graph-theory-based method for parallelizing the multiple-flow-direction algorithm on cuda compatible graphics processing units," in *ICSDM*. IEEE, 2011, pp. 137–141.

[28] L. Ortega and A. Rueda, "Parallel drainage network computation on cuda," *Computers & Geosciences*, vol. 36, no. 2, pp. 171 – 178, 2010.

[29] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[30] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, 2007.

[31] D. E. Knuth, *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[32] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, Dec. 1986. [Online]. Available: http://doi.acm.org/10.1145/7902.7903

[33] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1 –58, 29 2008.