Computers & Geosciences 43 (2012) 7–16

Contents lists available at SciVerse ScienceDirect

Computers & Geosciences

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

Parallelizing flow-accumulation calculations on graphics processing units—From iterative DEM preprocessing algorithm to recursive multiple-flow-direction algorithm

Cheng-Zhi Qin [a,*], Lijun Zhan [a,b]

[a] State Key Laboratory of Resources and Environmental Information System, Institute of Geographical Sciences and Natural Resources Research, CAS, Beijing 100101, China
[b] Graduate School of the Chinese Academy of Sciences, Beijing 100049, China

ARTICLE INFO

Article history:
Received 24 November 2011
Received in revised form
20 February 2012
Accepted 24 February 2012
Available online 4 March 2012

Keywords:
Parallel computing
Graphics processing unit (GPU)
Digital terrain analysis
Flow accumulation
Multiple-flow-direction algorithm (MFD)
DEM preprocessing

ABSTRACT

As one of the important tasks in digital terrain analysis, the calculation of flow accumulations from gridded digital elevation models (DEMs) usually involves two steps in a real application: (1) using an iterative DEM preprocessing algorithm to remove the depressions and flat areas commonly contained in real DEMs, and (2) using a recursive flow-direction algorithm to calculate the flow accumulation for every cell in the DEM. Because both algorithms are computationally intensive, quick calculation of the flow accumulations from a DEM (especially for a large area) presents a practical challenge to personal computer (PC) users. In recent years, rapid increases in hardware capacity of the graphics processing units (GPUs) provided in modern PCs have made it possible to meet this challenge in a PC environment. Parallel computing on GPUs using a compute-unified-device-architecture (CUDA) programming model has been explored to speed up the execution of the single-flow-direction algorithm (SFD). However, the parallel implementation on a GPU of the multiple-flow-direction (MFD) algorithm, which generally performs better than the SFD algorithm, has not been reported. Moreover, GPU-based parallelization of the DEM preprocessing step in the flow-accumulation calculations has not been addressed. This paper proposes a parallel approach to calculate flow accumulations (including both iterative DEM preprocessing and a recursive MFD algorithm) on a CUDA-compatible GPU. For the parallelization of an MFD algorithm (*MFD-md*), two different parallelization strategies using a GPU are explored. The first parallelization strategy, which has been used in the existing parallel SFD algorithm on GPU, has the problem of computing redundancy. Therefore, we designed a parallelization strategy based on graph theory. The application results show that the proposed parallel approach to calculate flow accumulations on a GPU performs much faster than either sequential algorithms or other parallel GPU-based algorithms based on existing parallelization strategies.

1. Introduction

Calculation of flow accumulations or specific contributing areas (SCAs) from gridded digital elevation models (DEMs) is one of the key issues in digital terrain analysis (DTA) and has a marked influence on a wide range of applications such as hydrologic analysis, soil erosion, and geomorphology (Wilson and Gallant, 2000; Hengl and Reuter, 2008). The calculation of flow accumulations from a gridded DEM is normally performed using the flow-direction algorithm, which determines how to drain the flow from each given cell in the DEM into the neighboring cell(s) and then recursively calculates the flow accumulation for every cell (Freeman, 1991). Over the last twenty years, many flow-direction algorithms have been developed (Wilson et al., 2008). Based on whether it is assumed that the flow from a cell can drain into only one neighboring cell or into one or more downslope neighboring cells, existing flow-direction algorithms can be classified into two main types: single-flow-direction (SFD) algorithm (e.g., the *D8* algorithm proposed by O'Callaghan and Mark (1984)) and multiple-flow-direction (MFD) algorithms (e.g., the *FD8* algorithm proposed by Quinn et al. (1991), the *D-inf* algorithm proposed by Tarboton (1997), and the *MFD-md* algorithm proposed by Qin et al. (2007)). MFD has generally been recognized to perform better than SFD from the perspective of algorithm error, especially when the spatial pattern of SCA or SCA-based topographic attributes (e.g., topographic wetness index) at a fine scale is needed (Wolock and McCabe, 1995; Wilson et al., 2008; Qin et al., 2011).

During flow-accumulation calculations for real DEMs, a DEM preprocessing algorithm is generally used to fill in the depressions and remove the flat areas in the DEMs before the flow-direction

* Corresponding author. Tel.: +86 10 648 89777; fax: +86 10 6488 9630.
E-mail address: qincz@lreis.ac.cn (C.-Z. Qin).

0098-3004/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cageo.2012.02.022

algorithm is used (Hengl and Reuter, 2008). These depressions and flat areas commonly exist in real DEMs, both because of actual terrain conditions and because of errors introduced during the DEM production process. These real or spurious features in a DEM will cause the flow-direction algorithms to fail to determine the flow direction properly and to obtain hydrological correct results for flow accumulation. Many DEM preprocessing algorithms, often with an iterative process, have been proposed to assist in the application of flow-direction algorithms (e.g., Jenson and Domingue, 1988; Martz and de Jong, 1988; Planchon and Darboux, 2001).

In real applications, flow-accumulation calculations have high computational complexity and are often very time-consuming. The high computational complexity arises not only from the recursive MFD algorithm, but also from the iterative DEM preprocessing algorithm. The traditional algorithm for calculating flow accumulation is coded as a sequential program executed on a single computer processor. Therefore, the execution time is often very long, especially for DEMs of large area and finer scale.

To speed up the execution time for flow-accumulation calculations, some researchers have proposed to parallelize these calculations using parallel programs designed for specific hardware architectures. Currently, three main types of hardware are used to parallelize flow-accumulation calculations: clusters, multi-core CPUs in a single personal computer (PC), and graphics processing units (GPUs). The cluster, which in theory can use any number of processors, has a high theoretical scalability and therefore can process a huge DEM dataset. Based on computer clusters, a parallelization of the *D8* algorithm has been developed in a message-passing-interface (MPI) programming model and has achieved a significant improvement in processing times (e.g., Wallis et al., 2009; Do et al., 2011). However, the ownership and operational costs of a computer cluster are high, and cluster programming is difficult because of the difficulty of code debugging and the lack of performance tuning tools. The use of computer clusters is still limited for most users. Multi-core CPUs in a single PC, as a cheaper and easy-to-use hardware solution, have also been used as a way to parallelize the *D8* algorithm using an open-multi-processing (OpenMP) programming model (Xu et al., 2010). However, the multiple threads used in multi-cored CPUs in a PC cannot provide the large amount of speedup required because of the limited number of cores available in the CPU (Lee et al., 2010).

GPU devices are attracting attention because they can accelerate digital terrain analysis (Xia et al., 2010) in a more efficient and economical way than multi-core CPUs in single PCs or than clusters. Designed originally for view processing for computer displays, GPUs have become very powerful, with up to hundreds of cores, and are widely provided in modern PCs. With the emergence of general-purpose computing on GPU (GPGPU) technology, such as the compute-unified-device-architecture (CUDA) programming model, GPUs have been used to parallelize many high-computational-complexity tasks ranging from physical process simulation to geographical computations (Tukora and Szalay, 2008).

However, little research has yet been done on parallelizing flow-accumulation calculations on a GPU. Ortega and Rueda (2010) presented a method of parallelizing the SFD algorithm, *D8*, on a GPU having the theoretical peak speed of 360 GFLOPS (giga floating-point operations per second) with 16 multiprocessors (including 128 scalar processors in total). Compared with the sequential implementation of *D8* executed on a CPU having the theoretical peak speed of 32 GFLOPS, their CUDA-based parallel *D8* algorithm achieved a high running efficiency, with a speedup ratio of approximately eight times. To the best of the authors' knowledge, there are no reports in the literature on parallelizing an MFD algorithm on a GPU, although MFD is generally thought to be better than SFD. Furthermore, current research has not yet addressed GPU-based parallelization of the DEM preprocessing step in the flow-accumulation calculations.

This paper presents a design and implementation of parallelized flow-accumulation calculations (including both iterative DEM preprocessing and a recursive MFD algorithm) on the NVIDIA™ GPU using the CUDA programming model. A parallelization strategy based on graph theory is proposed to improve efficiency of parallelizing the MFD algorithm on GPU.

2. CUDA-compatible GPU

At the hardware level, the GPU is composed of a scalable array of multiprocessors (MPs). Each MP contains 8 scalar processors (SPs). During a given processing cycle, each SP in the MPs executes the same instruction on different data, which is similar to the SIMD (single instruction stream, multiple data stream) model of a computer.

To use the parallel computing capabilities of the GPU for general-purpose computations, a C-based programming model called CUDA has been developed by the popular graphics-card manufacturer NVIDIA™ and has become the most popular GPGPU technology (Danalis et al., 2010). Using CUDA, a sequential algorithm to be parallelized should be redesigned to be processed on two different hardware platforms concurrently, the CPU (the host) and the CUDA-compatible GPU (the device) (Halfhill, 2008). The general workflow of CUDA computation consists of three phases, *Initialization*, *GPU Execution*, and *Finalization* (Fig. 1). During the *GPU Execution* phase, a large number of threads are automatically created and

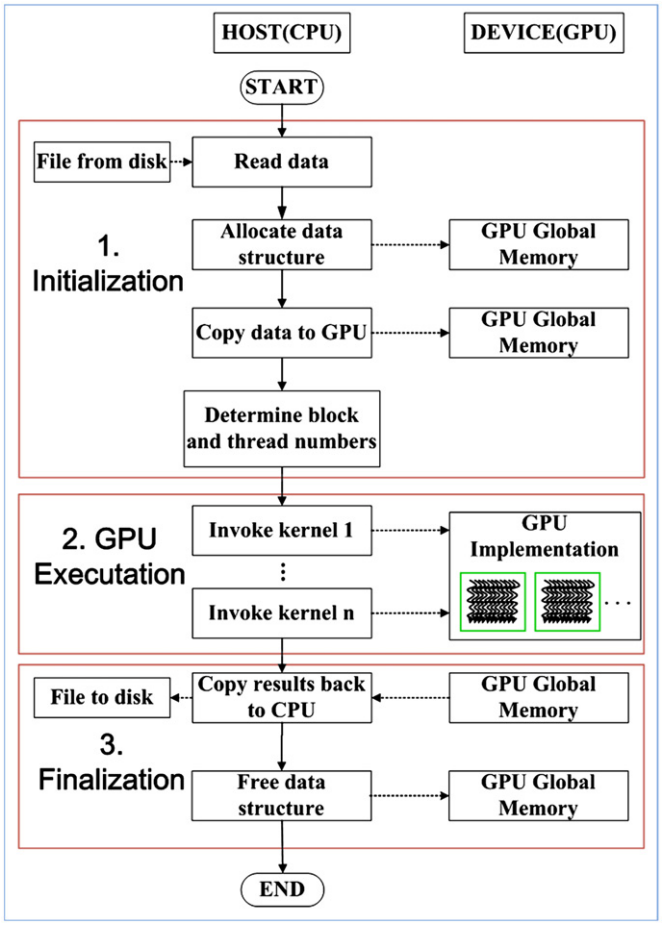

Fig. 1. Overall parallelization workflow on a GPU using a CUDA programming model.

then scheduled among MPs to keep processors computing and fully use the computing capability of a GPU.

3. Design for parallelization of a DEM preprocessing algorithm on a GPU

3.1. Original sequential DEM preprocessing algorithm proposed by Planchon and Darboux (2001)

The DEM preprocessing algorithm proposed by Planchon and Darboux (2001) (or P&D algorithm), which can iteratively revise the elevation of cells in depressions and flat areas in a DEM with a very small slope gradient, is thought to be the most suitable for MFD algorithms (Qin et al., 2007). In this study we select the P&D algorithm as a representative of DEM preprocessing algorithms for parallelization.

The P&D algorithm includes two steps: (1) the water-covering step, which involves adding a thick layer of water over the entire DEM except for the boundary, and (2) the water-removal step, which drains the excess water to ensure that for each cell, there is a path that leads to the boundary (Planchon and Darboux, 2001). According to the pseudocode of the P&D algorithm (Algorithm 1), the intensive computing part of the algorithm is the iterative process (lines 4–24 in Algorithm 1). The operation performed in each round of iterative processing is a neighborhood operation, which means that the computation for a cell will involve not only this cell, but also its neighboring cells. This iterative neighborhood operation has high potential for parallelization.

Algorithm 1. Pseudocode of sequential P&D DEM preprocessing algorithm (adapted from Planchon and Darboux (2001)). (zDEM and wDEM are the input DEM and the output DEM, respectively.)

```
1: void CPU_Preprocessing(zDEM[xSize][ySize], wDEM[xSize][ySize], gap)
2:     wDEM = Water_covering(zDEM)
3:     define bool stop = false
4:     repeat
5:         stop = true
6:         for i = 1 to xSize - 2 do      //except its boundary
7 :            for j = 1 to ySize -2 do
8:                if (wDEM[i][j] > zDEM[i][j]) then
9:                   for each existing neighbor [k][l] of [i][j] (in any order) do
10:                     if (zDEM[i][j] >= wDEM[k][l] + gap) then
11:                        wDEM[i][j] = zDEM[i][j]
12:                        stop = false
13:                     end if
14:                     else
15:                        if (wDEM[i][j] > wDEM[k][l] + gap) then
16:                           wDEM[i][j] = wDEM[k][l] + gap
17:                           stop = false
18:                        end if
19:                     end else
20:                  end for
21:               end if
22:            end for
23:        end for
24:    until stop == true
```

3.2. Strategy for parallelizing the DEM preprocessing algorithm on a GPU

Here the concern is how to parallelize the iterative part of the P&D algorithm. Unlike generic algorithms with iterative neighborhood operations, the sequential P&D algorithm permits the computation of the value of a given cell to use the values of its neighboring cells which have been updated during the current round of iteration (Planchon and Darboux, 2001). This means that the computation of each cell can be concurrent within a given round of the iterative process.

3.3. Parallel P&D algorithm

Based on the above strategy for parallelizing the P&D algorithm on a GPU, a CUDA-based parallel P&D algorithm was designed which consists of two parts: the host part and the device part. The host part, executed on the CPU, implements the water-covering step of the P&D algorithm (i.e., the *Initialization* phase in Fig. 1) and the output of the final result (i.e., the *Finalization* phase in Fig. 1). The iterative water-removal step of the P&D algorithm is parallelized on the device part and executed as a "kernel" on the GPU (i.e., the *GPU Execution* phase in Fig. 1).

The host part of the parallel P&D algorithm is shown in Algorithm 2. In line 2, the function *Water_covering()* serves to flood the whole surface except for the boundary with a thick layer of "water." In lines 4–6, all required data are copied from the host to global memory in the device, where all GPU threads can access it simultaneously for reading. Lines 8–9 assign the number of threads per block and the number of blocks based on the DEM size to ensure that each cell in the DEM, except the boundary, will be processed by a single thread. This thread-data mapping process requires a large number of threads and is feasible with a GPU in which the powerful arithmetic engine can run thousands of lightweight threads. The following loop code in the host part (lines 10–17) will iteratively call for the execution of each round in the water-removal step. This loop will terminate when the state variable *gpuStop* equals *true*, which means that none of the cells in the DEM has changed in altitude during the current round of the water-removal process on the GPU. Once the execution of the loop has terminated, the results of DEM preprocessing will be transferred from the GPU to the CPU (line 18).

The device part of the parallel P&D algorithm is shown in Algorithm 3. In lines 2–3, each thread obtains a thread ID and uses it as an index to obtain its corresponding data. The rest of Algorithm 3 implements the water-removal step of the P&D algorithm, which is similar to the corresponding code in the sequential Algorithm 1.

Algorithm 2. Pseudocode of the host part of the CUDA-based P&D algorithm. (cpuzDEM and cpuwDEM are the input DEM and the output DEM, respectively.)

```
1: void CUDA_Preprocessing (cpuzDEM[xSize][ySize],
cpuwDEM[xSize][ySize], gap)
2:     cpuwDEM = Water_covering (cpuzDEM)
3:     define Boolean cpuStop = false
4:     define gpu global memory Float gpuzDEM[xSize][ySize] ← cpuzDEM
5:     define gpu global memory Float gpuwDEM[xSize][ySize] ← cpuwDEM
6:     define gpu global memory Float gpugap ← gap
7:     define gpu global memory Boolean gpuStop
8:     numThreads(BLOCK_SIZE, BLOCK_SIZE)
9:     numBlocks(ROW_BLOCKS, COLUMN_BLOCKS)
10:    repeat
11:            cpuStop = true
12:            gpuStop ← cpuStop
13:            execute in each thread:
14:            Preprocessing_Thread (gpuzDEM, gpuwDEM, gpugap, gpuStop)
15:            cpuStop ← gpuStop
17:    until cpuStop == true
18:    cpuwDEM ← gpuwDEM
19:    freeData()
```

Algorithm 3. Pseudocode of the device part of the CUDA-based P&D algorithm.

```
1: void   Preprocessing_Thread (gpuzDEM, gpuwDEM, gpugap, gpuStop)
2:     define Integer j = blockIdx.y*blockDim.y + threadIdx.y
3:     define Integer i = blockIdx.x*blockDim.x + threadIdx.x
4:     if (i < xSize && j < ySize) then
5:       if(gpuwDEM[i][j] > gpuzDEM[i][j]) then
6:         for each existing neighbor [k][l] of [i][j] do
7:           define Float wdem = gpuwDEM[k][l]
8:           if(gpuzDEM[i][j] >= wdem + gpugap) then
9:             gpuwDEM[i][j] = gpuzDEM[i][j]
10:            gpuStop = false
11:          end if
12:          else
13:            if(gpuwDEM[i][j] > wdem + gpugap) then
14:              gpuwDEM[i][j] = wdem + gpugap
15:              gpuStop = false
16:            end if
17:          end else
18:        end for
19:      end if
20:    end if
```

4. Design for parallelization of a multiple-flow-direction algorithm on a GPU

4.1. Analysis of the parallelizability of the recursive MFD algorithm

Following the earliest MFD algorithm with a recursive design (Freeman, 1991; Quinn et al., 1991), later MFD algorithms have generally focused on how to model the allocation of flow among multiple neighboring cells of a given cell. Therefore, these algorithms can be formalized as a similar form as follows:

$$d_i = \frac{(\tan\beta_i)^p \times L_i}{\sum_{j=1}^{8} (\tan\beta_j)^p \times L_j} \quad (1)$$

where d_i is the fraction of flow into the i-th neighboring cell from a given cell, $\tan\beta_i$ is the slope gradient of the neighboring cell i, and L_i is the "effective contour length" of the neighboring cell i of the central cell. L_i equals 0.5 for downslope cells in cardinal directions, 0.354 for downslope cells in diagonal directions, and 0 for non-downslope neighboring pixels (Quinn et al., 1991). The flow-partition exponent p is set to a constant value in the classic MFD, e.g., $p=1$ in the *FD8* algorithm (Quinn et al., 1991). Other MFD algorithms generally suggest varying the flow-partition exponent p by a function related to the terrain conditions (e.g., Quinn et al., 1995; Kim and Lee, 2004; Qin et al., 2007).

This study uses *MFD-md*, an MFD algorithm proposed by Qin et al. (2007), as an example for parallelization of MFDs which are based on a form similar to Eq. (1). The *MFD-md* algorithm adapts to local terrain conditions by determining the flow-partition exponent based on the local maximum downslope gradient (Qin et al., 2007):

$$f(e) = 8.9 \times \min(e,1) + 1.1 \quad (2)$$

where e is the tangent value of the maximum downslope gradient, $\min(e,1)$ is the minimum of e and 1, and $f(e)$ is the function for determining the flow-partition exponent in Eq. (1). Experimental results have shown that *MFD-md* produces lower error on artificial surfaces and achieves a more reasonable result on real-world surfaces compared with classic MFD and SFD (Qin et al., 2007).

The sequential *MFD-md* algorithm consists of two steps: (1) data preparation and (2) flow-accumulation calculations. The tasks in the first step include the calculation and recording of both the multiple flow directions and the tangent value of the maximum downslope gradient for each cell in the DEM. Then the flow fractions among all neighboring cells of each cell are also determined based on Eqs. (1) and (2). The time consumed in this data-preparation step is relatively minor, and therefore it is not necessary to parallelize it on the GPU.

The flow-accumulation calculation step in the *MFD-md* algorithm is computationally intensive and time-consuming. Therefore, the parallelization of this step on a GPU has the potential to accelerate the execution of the *MFD-md* algorithm considerably. However, traditional flow-accumulation calculations often use a depth-first-search (DFS) process which recursively calculates from outlet to peak. The DFS process is thought to be an inherently sequential process and therefore has no parallel solution (Reif, 1985).

In this research, the recursive flow-accumulation calculation was first converted into an iterative algorithm which calculates the flow accumulation in a sequence from the peak to the outlet of each area. Then concurrency will be possible in each round of the iterative process. Two parallelization strategies will be explored in the remaining parts of this section.

4.2. Design 1: a flow-transfer-matrix-based parallel MFD-md algorithm on GPU

4.2.1. Parallelization strategy based on the flow-transfer matrix proposed by Ortega and Rueda (2010)

During the process of designing and parallelizing the MFD algorithm on a GPU, it was natural to consider whether or not the parallelization strategy used in the existing CUDA-based parallel SFD algorithm is still available in this case. This parallelization strategy proposed by Ortega and Rueda (2010) used a data structure called the "flow-transfer matrix" to parallelize the *D8* algorithm on a GPU. Based on the flow-transfer matrix, the flow-accumulation process which is recursively calculated in the traditional algorithm can be simulated on a GPU by an iterative flow-transfer process among the neighboring cells of every cell in the DEM. In each round of the iterative process, a flow-transfer matrix is used to record the flow accumulation transferred into every cell from its neighboring cells during the current round of the process. The flow transfer to every cell in an individual round of the iterative process can be parallelized on a GPU. The iterative process terminates when no cell has flow transferred to it from its neighboring cells in the current round of the process. The result of the flow-accumulation algorithm is the sum of all the flow-transfer matrices from each round of the process.

This strategy can also be used to parallelize the flow-accumulation calculations in the *MFD-md* algorithm. This parallelization process can be illustrated using a 3×3 DEM example (Fig. 2a). During the data-preparation step of the *MFD-md* algorithm (Eqs. (1) and (2)), the flow-transfer action for every cell of the DEM is determined both by its multiple flow directions and by their corresponding flow fractions (Fig. 2b). During the flow-accumulation calculation step of the *MFD-md* algorithm, the flow-transfer matrix, *flowTransfer*, is used to record the flow accumulation that is transferred in each round of the parallel process. The initial flow-transfer matrix, *flowTransfer0*, is set to have a value of one 1 for every cell (Fig. 2c) to simulate the amount of water that each *cell(i,j)* directly obtains from rainfall. In the first round of the process, a flow-transfer matrix, *flowTransfer1*, records the flow accumulation which every cell drains from its neighboring cells according to the flow-transfer action, together with the flow recorded in *flowTransfer0* (Fig. 2d). For example, the cell of interest (2,2) receives the flow drained from its upslope neighboring cells (i.e., *cell(1,1)*, *cell(1,2)*, *cell(1,3)*, and *cell(2,3)*) plus the amount recorded in these neighboring cells in *flowTransfer0* (Fig. 2c) multiplied by the corresponding flow fraction (Fig. 2b) from each of these neighboring cells to the cell

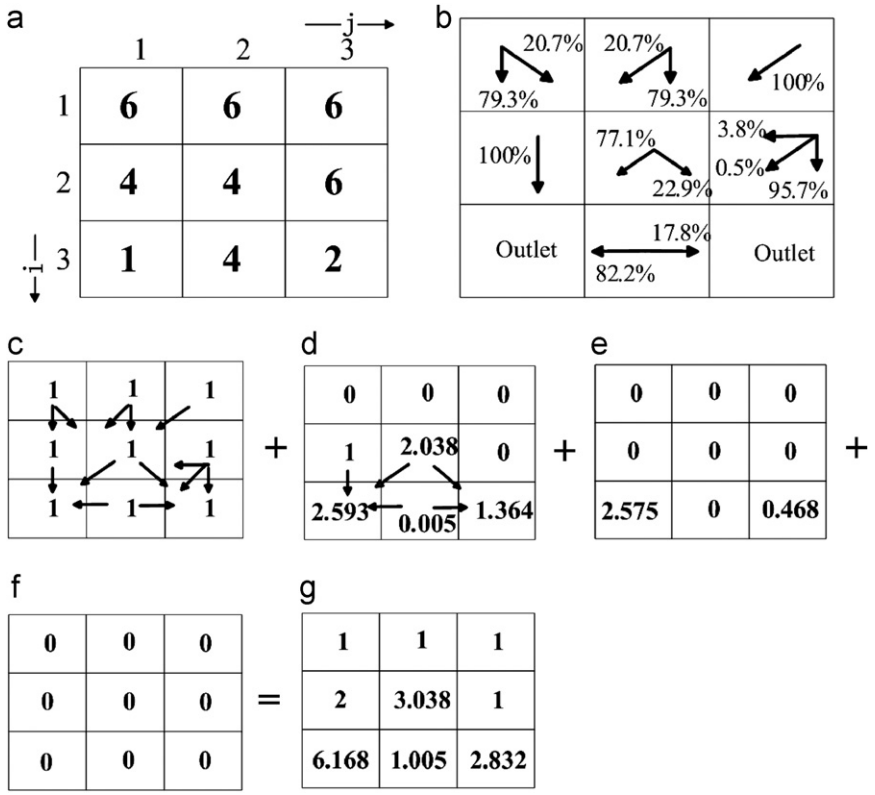

Fig. 2. Illustration of the parallel MFD-md algorithm using a flow-transfer matrix: (a) a 3 × 3 DEM; (b) multiple flow directions (marked as arrows) and corresponding flow fractions for the DEM in (a); (c) initial flow transfer matrix; (d–f) the flow-transfer matrices which record the flow accumulation transferred in the first to the last rounds of the parallel process using the MFD-md algorithm; (g) flow-accumulation result, which is the sum of the flow-transfer matrices (c–f).

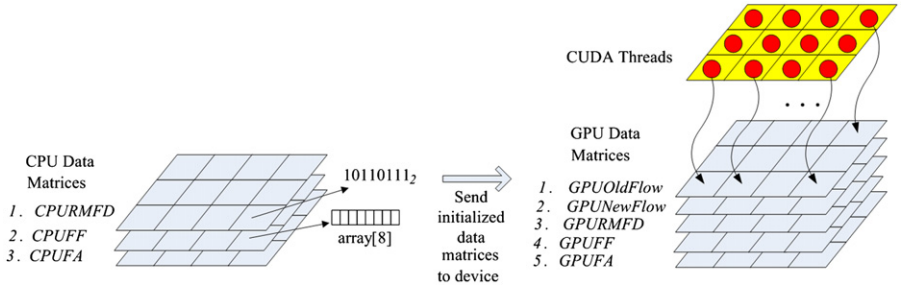

Fig. 3. Data matrices for flow-transfer-matrix-based parallel MFD-md algorithm and thread-data mapping in the GPU.

of interest (2,2) (i.e., 20.7% × 1, 79.3% × 1, 100% × 1, and 3.8% × 1 respectively, which sum to 2.038). Therefore, *flowTransfer1*(2,2) is set to 2.038, which means that *cell(2,2)* receives 2.038 units of flow in this round (Fig. 2d). This operation in one round of an iterative process is a neighborhood operator which is easy to parallelize on a GPU. The following rounds of the iterative process are similar (Fig. 2e). When no cell receives a nonzero flow transfer (Fig. 2f), the flow-accumulation result of the *MFD-md* algorithm can be summed up (Fig. 2g) using the equation:

$$flow\ Accumulation = \sum flow\ Transfer_i$$

4.2.2. Flow-transfer-matrix-based parallel MFD-md algorithm on a GPU

Using the parallelization strategy based on flow-transfer matrices, a CUDA-based parallel implementation of the *MFD-md* algorithm was designed. This flow-transfer-matrix-based parallel *MFD-md* algorithm consists of two parts, the host part and the device part.

The host part (Algorithm 4) first initializes three two-dimensional matrices allocated in PC internal memory and duplicates these matrices into the corresponding matrices allocated in GPU global memory (lines 2–6). As shown in Fig. 3, the first matrix in PC internal memory is the reversal multiple-flow-direction matrix (*CPURMFD* matrix), for which the corresponding matrix in GPU memory is *GPURMFD*. Each cell in *CPURMFD* and *GPURMFD* stores an eight-bit variable in which each bit records whether the neighboring cell in the corresponding direction will be drained into the current cell. The second matrix in PC internal memory is the flow-fraction matrix (*CPUFF*), which records the flow fractions used to partition flow among the downslope neighboring cells of each cell. The corresponding matrix in GPU global memory is *GPUFF*. Each cell in *CPUFF* and *GPUFF* stores an array with eight elements; each element records the fraction of flow going to a specific neighboring cell of the current cell. The third matrix in PC internal memory, the flow-accumulation matrix (*CPUFA*), records the flow accumulation. The corresponding matrix in GPU global memory is the *GPUFA* matrix. In GPU global memory, two more matrices, *GPUOldFlow* and *GPUNewFlow* (Fig. 3), are used to simulate the flow transfer in each round of the parallel process. The number of threads per block and the number of blocks are determined in lines 8–9 depending on the matrix size (i.e., the

size of the DEM) to ensure that each cell will be mapped to an individual thread (Fig. 3). Using the loop code (lines 10–17), the host part iteratively invokes the execution of a round of the flow-transfer process until no cell receives flow transferred from its neighboring cells in the current round of the process (or in other words, when the state variable *cpuStop* equals *true*).

The device part of the flow-transfer-matrix-based parallel *MFD-md* algorithm, *FlowAccu_Thread_FlowTranferMatrix()* (shown as Algorithm 5) was implemented to simulate the flow accumulation of each cell in one round of the flow-transfer process. It is invoked by the host part and is executed by each GPU thread which maps a cell in the DEM.

Algorithm 4. Pseudocode of the host part of the parallel *MFD-md* using the flow-transfer matrix. (cpuDEM and cpuFA are the input DEM and the output result, respectively.)

```
1: void CUDA_MFDmd_FlowTranferMatrix (cpuDEM[xSize][ySize])
2:       define gpu global memory Integer gpuRMFD [xSize][ySize] ← cpuRMFD
3:     define gpu global memory Float gpuFF[xSize][ySize] ← cpuFF
4:     define gpu global memory Float gpuOldFlow[xSize][ySize] = 1
5:     define gpu global memory Float gpuNewFlow[xSize][ySize] = 0
6:     define gpu global memory Float gpuFA[xSize][ySize] = 0
7:     define gpu global memory Boolean gpuStop
8:     numThreads(BLOCK_SIZE, BLOCK_SIZE)
9:     numBlocks(ROW_BLOCKS, COLUMN_BLOCKS)
10:     repeat
11:           define Boolean cpuStop = true
12:           gpuStop ← cpuStop
13:           execute in each thread:
14:           FlowAccu_Thread_FlowTranferMatrix (gpuRMFD, gpuFF, gpuFA, gpuOldFlow, gpuNewFlow, gpuStop)
15:           cpuStop ← gpuStop
16:           gpuOldFlow = gpuNewFlow
17:           gpuNewFlow = 0
18:     until cpuStop == true
19:     cpuFA ← gpuFA
20:     freeData()
```

Algorithm 5. Pseudocode of the device part of the parallel *MFD-md* using the flow-transfer matrix.

```
1: void FlowAccu_Thread_FlowTranferMatrix (gpuRMFD, gpuFF, gpuFA, gpuOldFlow, gpuNewFlow, gpuStop)
2:     define Integer j = blockIdx.y*blockDim.y + threadIdx.y
3:     define Integer i = blockIdx.x*blockDim.x + threadIdx.x
4:     if (i < xSize && j < ySize) then
5:         if(gpuOldFlow[i][j] == 0) then return
6:         define Float accu = 0
7:         define Integer revDir = gpuRMFD[i][j]
8:                 if(revDir & 1) then accu += gpuOldFlow[i][j+1] * gpuFF[i][j+1].w[1]
9:             if(revDir & 2) then accu += gpuOldFlow[i+1][j+1] * gpuFF[i+1][j+1].w[2]
10:             if(revDir & 4) then accu += gpuOldFlow[i+1][j] * gpuFF[i+1][j].w[3]
11:             if(revDir & 8) then accu += gpuOldFlow[i+1][j-1] * gpuFF[i+1][j-1].w[4]
12:             if(revDir & 16) then accu += gpuOldFlow[i][j-1] * gpuFF[i][j-1].w[5]
13:             if(revDir & 32) then accu += gpuOldFlow[i-1][j-1] * gpuFF[i-1][j-1].w[6]
14:             if(revDir & 64) then accu += gpuOldFlow[i-1][j] * gpuFF[i-1][j].w[7]
15:             if(revDir & 128) then accu += gpuOldFlow[i-1][j+1] * gpuFF[i-1][j+1].w[8]
16:         if(accu > 0) then
17:             gpuFA [i][j] += accu
18:             gpuNewFlow[i][j] = accu
19:             gpuStop = false
20:         end if
21:     end if
```

4.3. Computing redundancy problem in the flow-transfer-matrix-based parallelization strategy

All parallel flow-direction algorithms which use a flow-transfer matrix have the problem of computing redundancy, including the parallel *D8* algorithm and the parallel *MFD-md* algorithm. Taking a small 100×100 DEM as an example (Fig. 4), the elevation of this area drops continuously from the northeast to the southwest until the outlet, cell(100,1). In the first round of parallel processing, all cells are involved in the computations according to the iterative process in the parallel flow-direction algorithm using the flow-transfer matrix. However, the flow-accumulation computation is completed only for the cells in the first row and the 100th column. The flow-transfer computation among the other cells has no influence on the flow-accumulation computation for the cells in the first row and the 100th column. In other words, computing redundancy exists for 99×99 cells. In the second round of parallel processing, only the cells in the second row and the 99th column can have their flow accumulation calculated completely. The flow-transfer computation among the other 98×98 cells is redundant. The situation in subsequent rounds of parallel processing is similar. This computing redundancy can become enormous and limit the computational efficiency of the algorithm.

4.4. Design 2: a graph-theory-based parallel MFD-md algorithm on a GPU

This section presents an algorithm design based on graph theory instead of on the flow-transfer matrix to avoid the computing redundancy problem occurring in parallel flow-direction algorithms using the flow-transfer matrix.

4.4.1. Parallelization strategy based on graph theory

The basic idea of the algorithm is to view the calculation of flow accumulations from a graph-theory perspective (Arge et al., 2003; Wallis et al., 2009). A flow-direction graph can be naturally defined. Taking a 3×3 DEM as an example (Fig. 5a), an individual vertex in the graph can be associated with each cell in the DEM. There is an edge from vertex i to vertex j in the graph if the flow direction from cell i to cell j exists (Fig. 5c). Each edge in the graph is associated with an attribute value of the flow fraction determined by Eqs. (1) and (2) (Fig. 5b). A flow-direction graph defined in this way is a directed acyclic graph, as Arge et al. (2003) demonstrated.

The flow-direction graph can be used to specify the order of computation of the flow accumulations of all cells as a topological order in graph theory, which means that vertex i appears before vertex j in the ordering if there a path from vertex i to vertex j (Weiss, 1997). Under this ordering, the flow accumulation of a given cell cannot be calculated until the flow accumulation of

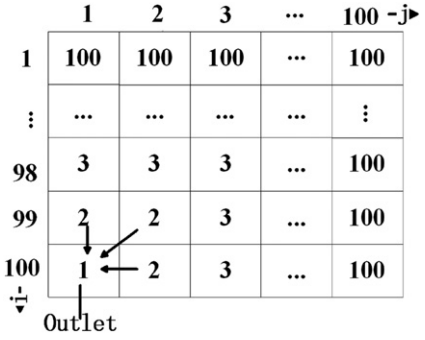

Fig. 4. 100×100 DEM example used for discussion of the computing redundancy problem in a parallel flow-direction algorithm using a flow-transfer matrix.

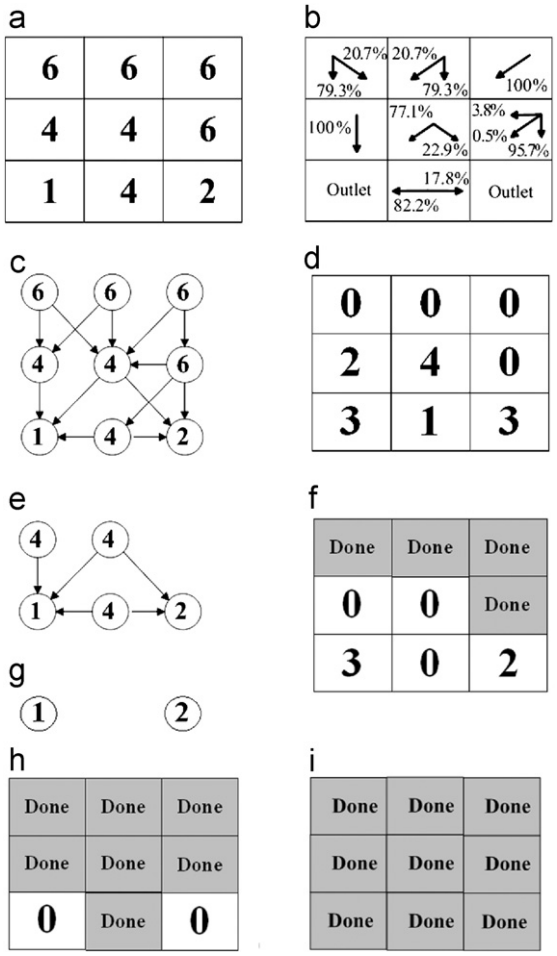

Fig. 5. Illustration of processing in the graph-theory-based parallel MFD-md algorithm: (a) 3×3 DEM example (the value in each cell is the elevation); (b) flow direction marked as an arrow with the flow-fraction value determined by MFD-md; (c) initial flow-direction graph (the value of each vertex in the flow-direction graph is the elevation), as in (e) and (g); (d) indegree matrix corresponding to (c); (e) flow-direction graph after the first round of parallel processing; (f) indegree matrix corresponding to (e) (a gray cell means that the flow-accumulation calculation for this cell is finished); (g) flow-direction graph after the second round of parallel processing; (h) indegree matrix corresponding to (g); (i) flow accumulation for all cells as calculated after the last round of processing.

every cell draining into it has been calculated previously. Thus, in one round of processing, the cells corresponding to vertices without incoming edges in the graph should have their flow accumulation calculated. Then, for those cells for which the flow-accumulation calculations are complete, the corresponding vertices and the edges between these vertices will be removed from the graph. During this round of processing, cells corresponding to vertices with at least one incoming edge require no processing. The remaining rounds of processing take place in a similar way. Therefore, the flow-direction algorithm designed on a graph-theory basis could avoid a large amount of computing redundancy compared with a flow-transfer-matrix-based parallel algorithm.

To determine the topological ordering of the flow-direction graph, all cells without incoming edges should first be identified. This process can be parallelized using an *indegree* matrix in which each cell records the count of neighboring cells that drain into this cell, i.e., the number of immediate incoming edges of the corresponding vertex of this cell in the flow-direction graph (Fig. 5d).

The parallel processing of *MFD-md* based on graph theory is illustrated in Fig. 5c through i. In the first round of processing, the flow accumulations of cells with zero *indegree* value are calculated. Then, the *indegree* values of these cells in the *indegree* matrix are marked as "Done." The *indegree* value of each cell which has immediate incoming edge(s) from these cells is updated by subtracting the count of immediate incoming edges from these cells (Fig. 5f). All these operations are conducted in parallel. Among the cells which have had their *indegree* values updated, those cells that currently have a zero *indegree* value are ready for flow-accumulation calculation in the next round of parallel processing. The process terminates when all cells are marked with an *indegree* value of "Done" (Fig. 5i).

4.4.2. *The graph-theory-based parallel MFD-md algorithm on GPU*

The host part and the device part of the graph-theory-based parallel *MFD-md* algorithm are shown as Algorithm 6 and Algorithm 7 respectively. The host part is similar to that of the flow-transfer-matrix-based parallel *MFD-md* algorithm (Algorithm 4). The difference is that Algorithm 6 uses new matrices, i.e., the *indegree* matrix and the multiple-flow-direction matrix according to the proposed parallelization strategy, instead of the *GPUOldFlow* and *GPUNewFlow* matrices used in Algorithm 4. The *indegree* matrix and the multiple-flow-direction matrix allocated in PC internal memory are *CPUIndegree* and *CPUMFD* respectively, whereas the corresponding matrices allocated in GPU global memory are *GPUIndegree* and *GPUMFD* respectively. The multiple-flow-direction matrix is used to determine which of the neighboring cells of a given cell should have its *indegree* value updated. Each cell in the multiple-flow-direction matrix stores an eight-bit variable in which each bit indicates whether the neighboring cell in the corresponding direction will be drained from the current cell. The thread-data mapping strategy is the same as that used in the flow-transfer-matrix-based parallel *MFD-md* algorithm (see Fig. 3).

On the device part, atomic functions are used to enforce atomic access to shared variables in *gpuIndegree*, as shown in lines 19–26 of Algorithm 7, *FlowAccu_Thread_Graph()*. This means that any other thread cannot access these variables until the current operation on these variables is complete.

Algorithm 6. Pseudocode of the host part of the graph-theory-based parallel *MFD-md* algorithm. (cpuDEM and cpuFA are the input DEM and the output result, respectively.)

```
1: void CUDA_MFDmd_Graph(cpuDEM[xSize][ySize])
2:     define gpu global memory Integer gpuIndegree[xSize][ySize] ← cpuIndegree
3:     define gpu global memory Integer gpuMFD [xSize][ySize] ← cpuMFD
4:     define gpu global memory Integer gpuRMFD [xSize][ySize] ← cpuRMFD
5:     define gpu global memory Float gpuFF[xSize][ySize] ← cpuFF
6:     define gpu global memory Float gpuFA[xSize][ySize] = 0
7:     define gpu global memory Boolean gpuStop
8:     numThreads(BLOCK_SIZE, BLOCK_SIZE)
9:     numBlocks(ROW_BLOCKS, COLUMN_BLOCKS)
10:    repeat
11:        define Boolean cpuStop = true
12:        gpuStop ← cpuStop
13:        execute in each thread:
14:        FlowAccu_Thread_Graph(gpuIndegree, gpuRMFD , gpuMFD, gpuFF, gpuFA, gpuStop)
15:        cpuStop ← gpuStop
16:    until cpuStop == true
17:    cpuFA ←gpuFA
18:    freeData()
```

Algorithm 7. Pseudocode of the device part of the graph-theory-based parallel *MFD-md* algorithm.

```
1: void MFDmd_Thread_Graph(gpuIndegree, gpuRMFD, gpuMFD, gpuFF, gpuFA, gpuStop)
2:     define Integer j = blockIdx.y*blockDim.y + threadIdx.y
3:     define Integer i = blockIdx.x*blockDim.x + threadIdx.x
4:     if (i < xSize && j < ySize) then
5:         if(gpuIndegree[i][j] == 0) then
6:             gpuIndegree[i][j] = -1
7:             define Float accu = 0
8:             define Integer revDir =  gpuRMFD[i][j]
9:             if(revDir & 1) then accu += gpuFA[i][j+1] * gpuFF[i][j+1].w[1]
10:                if(revDir & 2) then accu += gpuFA [i+1][j+1] * gpuFF[i+1][j+1].w[2]
11:            if(revDir & 4) then accu += gpuFA[i+1][j] * gpuFF[i+1][j].w[3]
12:                if(revDir & 8) then accu += gpuFA[i+1][j-1] * gpuFF[i+1][j-1].w[4]
13:            if(revDir & 16) then accu += gpuFA[i][j-1] * gpuFF[i][j-1].w[5]
14:                if(revDir & 32) then accu += gpuFA[i-1][j-1] * gpuFF[i-1][j-1].w[6]
15:            if(revDir & 64) then accu += gpuFA[i-1][j] * gpuFF[i-1][j].w[7]
16:                if(revDir & 128) then accu += gpuFA[i-1][j+1] * gpuFF[i-1][j+1].w[8]
17:            gpuFA[i][j] += accu
18:            define Integer dir = gpuMFD[i][j]
19:            if(dir & 1) then atomicSub(gpuIndegree[i][j+1], 1)
20:            if(dir & 2) then atomicSub(gpuIndegree[i+1][j+1], 1)
21:            if(dir & 4) then atomicSub(gpuIndegree[i+1][j], 1)
22:            if(dir & 8) then atomicSub(gpuIndegree[i+1][j-1], 1)
23:            if(dir & 16) then atomicSub(gpuIndegree[i][j-1], 1)
24:            if(dir & 32) then atomicSub(gpuIndegree[i-1][j-1], 1)
25:            if(dir & 64) then atomicSub(gpuIndegree[i-1][j], 1)
26:            if(dir & 128) then atomicSub(gpuIndegree[i-1][j+1], 1)
27:            gpuStop = false
28:        end if
29:    end if
```

5. Experiments and results

5.1. Experimental design

To assess the performance of the proposed parallel DEM preprocessing algorithm and various parallel *MFD-md* algorithms on a GPU, the run times of the parallel algorithms designed in this paper were measured, including the parallel P&D DEM preprocessing algorithm (called *Preprocessing_gpu*), the flow-transfer-matrix-based parallel *MFD-md* algorithm (*MFDmd_FTM_gpu*), and the graph-theory-based parallel *MFD-md* algorithm (*MFDmd_graph_gpu*), and compared with the run times of the corresponding sequential algorithms (*Preprocessing_cpu*, *MFDmd_FTM_cpu*, and *MFDmd_graph_cpu*). The performance of parallelization of flow-accumulation calculations (including both P&D DEM preprocessing and the *MFD-md* algorithm) was assessed on the GPU studied in this paper (called *Workflow_gpu*) by comparing the run time of *Workflow_gpu* (i.e., connecting *Preprocessing_gpu* to *MFDmd_graph_gpu*) with the run time of *Workflow_cpu* (i.e., connecting *Preprocessing_cpu* to *MFDmd_graph_cpu*). Here the run time of an algorithm is the execution time of each tested algorithm, including the time needed for matrix preparation (such as determining multiple flow directions and building the directed acyclic graph) and the time needed for transferring data between GPU and CPU. The time needed for loading the DEM data into PC internal memory and moving the results from PC internal memory to external memory is not counted in the run time.

All algorithms were executed on a PC with an Intel Core 2 duo CPU (the theoretical peak speed is 32 GFLOPS, and only single core was used for the experiments) with 3 GB RAM and a GeForce GT 330 graphics card (the theoretical peak speed is 182 GFLOPS) with 1 GB global memory and 12 MPs (i.e. 96 SPs) running CUDA version 3.0. The operating system was 32-bit Windows XP Professional.

The test data are from a gridded DEM at 10-m resolution of a low-relief area (approximately 60 km^2, Fig. 6) in northeastern China and from another five gridded DEMs created by resampling the original DEM under different grid resolutions. The six DEMs have dimensions of 1225×855, 1633×1140, 2450×1710, 3063×2138, 3769×2631, and 4899×3420 respectively. They are used to assess the performance of the parallel algorithms designed in this paper on datasets of different dimensions.

5.2. Experimental results

The experiment results (Table 1, Fig. 7) show that in all cases the proposed parallel algorithms are more efficient than their corresponding sequential algorithms. This is because parallel algorithms take advantage of the GPU architecture which

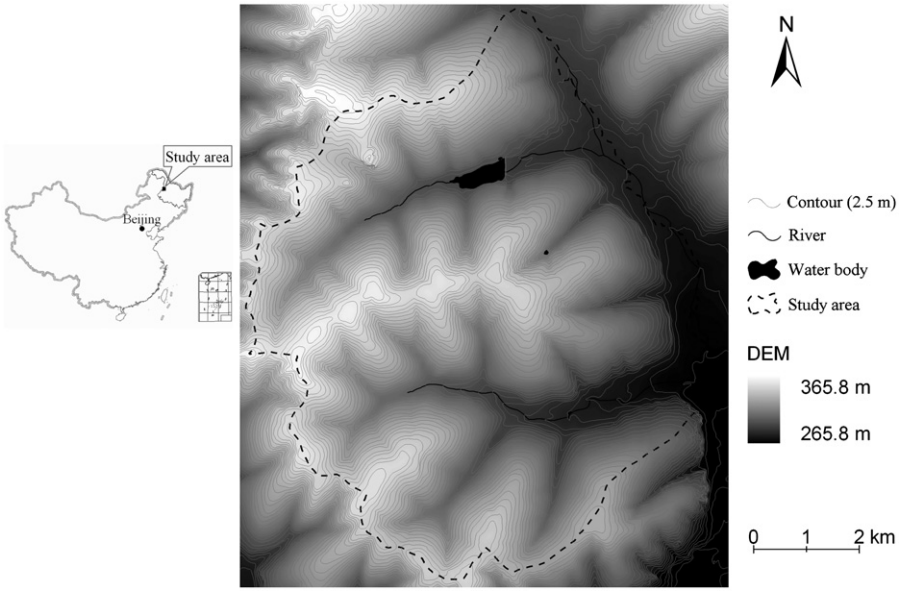

Fig. 6. Map of study area.

Table 1
Run times (ms) of algorithms with different dimensions of DEM (in cell).

Algorithm	Dimensions of DEM (in cell)					
	(1225 × 855)	(1633 × 1140)	(2450 × 1710)	(3063 × 2138)	(3769 × 2631)	(4899 × 3420)
Preprocessing_cpu	14,485	39,938	142,234	190,172	1,504,890	3,305,500
Preprocessing_gpu	906	2187	7078	11,784	67,562	148,296
MFDmd_FTM_cpu	22,943	41,639	94,084	315,443	308,260	651,250
MFDmd_FTM_gpu	6101	10,190	20,369	59,444	60,034	126,987
MFDmd_graph_cpu	14,829	32,724	67,868	162,423	163,425	510,716
MFDmd_graph_gpu	2547	5017	8491	16,607	21,216	46,803
Workflow_cpu	29,314	72,662	210,102	352,595	1,668,315	3,816,216
Workflow_gpu	3453	7204	15,569	28,391	88,778	195,099

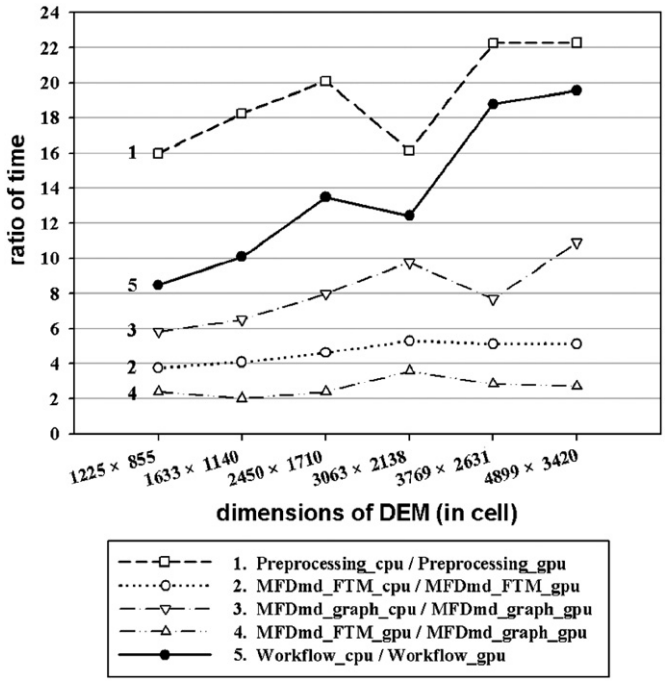

Fig. 7. Experimental result. Curve 1 shows the speedup of the parallel P&D DEM preprocessing algorithm; curves 2 and 3 represent the speedups of the flow-transfer-matrix-based and graph-based parallel MFD-md algorithm respectively; curve 4 compares the efficiency of the two parallel MFD-md algorithms; curve 5 compares the runtimes when the DEM preprocessing and MFD-md algorithms are executed as a whole workflow on a GPU or a CPU.

has a large number of low-clock-frequency processors working in parallel.

The parallel P&D DEM preprocessing algorithm (*Preprocessing_gpu*) shows the highest speedup, ranging from 15.9 to 22.3 times (curve 1 in Fig. 7). This can be attributed to the high degree of independence of the operations within an iteration of the P&D algorithm, as analyzed in Section 3.2.

The speedup of *MFDmd_graph_gpu* ranges from 5.8 to 10.9 times with different datasets in this test (curve 3 in Fig. 7). It is higher than *MFDmd_FTM_gpu*, which the speedup ranges from 3.8 to 5.3 times (curve 2 in Fig. 7). This shows the proposed parallelization strategy based on graph theory has higher parallelizability on the GPU than the existing parallelization strategy based on the flow-transfer matrix. Furthermore, due to avoiding the computing redundancy problem by using the graph-theory-based strategy instead of the flow-transfer-matrix strategy, *MFDmd_graph_gpu* is more than twice as fast as *MFDmd_FTM_gpu* (curve 4 in Fig. 7), even though the atomic functions used in the *MFDmd_graph_gpu* algorithm may cause some latency time.

Curve 5 in Fig. 7 shows that *Workflow_gpu* is also distinctly faster than the reference *Workflow_cpu*. The speedup of *Workflow_gpu* ranges from 8.5 to 19.5 times. This means that the proposed parallelization of the flow-accumulation calculations (including both iterative DEM preprocessing and the recursive MFD algorithm) on a CUDA-compatible GPU achieves the purpose of this study.

6. Conclusions and discussion

This paper presents a parallelization of flow-accumulation calculations (including both an iterative DEM preprocessing step and a recursive MFD algorithm) on a CUDA-compatible GPU. The parallel algorithms designed in this paper include a parallel P&D DEM preprocessing algorithm and two parallel *MFD-md* algorithms based on different parallelization strategies. The existing parallelization strategy, based on the flow-transfer matrix used in a parallel *D8* algorithm (Ortega and Rueda, 2010), has the problem of computing redundancy. Therefore, a parallelization strategy based on graph theory has been proposed, and a graph-theory-based parallel *MFD-md* algorithm has been designed. The experimental results show that the proposed parallelization of flow-accumulation calculations on a GPU performs much faster than either of the sequential algorithms or the parallel algorithm on a GPU, based on the existing parallelization strategy.

In this study, the *MFD-md* algorithm was used as an example of MFD for parallelization on a CUDA-compatible GPU. In fact, the proposed parallelization strategy is also available for other MFD algorithms (e.g. *D-inf* algorithm). Furthermore, the methodology used in the design of the proposed parallel *MFD-md* algorithm, which is to change the recursive algorithm into an iterative process for better parallelizability, is also potentially useful for the parallelization of other recursive algorithms in DTA (e.g., Tarboton et al., 2009; Tesfa et al., 2011).

This study uses a GPU as the hardware for parallelization, instead of a PC cluster or multi-core CPUs in a single PC. It should be noted that the size of GPU global memory limits the dimension of DEM which can be processed by parallel GPU-based algorithms. For example, the maximum dimension of DEM which can be processed by *MFDmd_graph_gpu* algorithm with a GPU having 1 GB global memory (the case in this test) is about 5000 × 5000 cells, because *MFDmd_graph_gpu* algorithm needs 3 integer arrays and 9 float arrays to be allocated in GPU global memory. The maximum dimension of DEM which can be processed by *MFDmd_FTM_gpu* algorithm is smaller than that. To handle larger DEMs by the parallel algorithms designed in this paper, more advanced GPU should be used. Although this limitation in parallelizing DTA algorithms on a GPU, high speedup for computing-intensive and time-consuming DTA algorithms is achieved in an efficient and economical way. The parallel algorithms designed in this paper can work on any PC with a GPU graphic card running

CUDA version 1.1 or higher. This study shows the potential of the GPU for parallelization for both algorithms and applications in the DTA domain.

Acknowledgments

This study was funded by the National High-Tech Research and Development Program of China (2011AA120302) and the Knowledge Innovation Program of the Chinese Academy of Sciences (KZCX2-YW-Q10-1-5). This study was also partly funded by the National Natural Science Foundation of China (40971235) and the Institute of Geographical Sciences and Natural Resources Research (2011RC203). We thank Prof. David Tarboton and an anonymous reviewer for their constructive comments on the earlier version of this paper.

Appendix A. Supporting information

Supplementary data associated with this article can be found in the online version at doi:10.1016/j.cageo.2012.02.022.

References

Arge, L., Chase, J.S., Halpin, P., Toma, L., Vitter, J., Urban, S.D., Wickremesinghe, R., 2003. Efficient flow computation on massive grid terrain datasets. Geoinformatica 7 (4), 283–313.

Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S., 2010. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In: Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3), New York, pp. 63–74.

Do, H.T., Limet, S., Melin, E., 2011. Parallel computing flow accumulation in large Digital Elevation Models. Procedia Computer Science 4, 2277–2286.

Freeman, T.G., 1991. Calculating catchment area with divergent flow based on a regular grid. Computers & Geosciences 17 (3), 413–422.

Halfhill, T.R., 2008. Parallel processing with CUDA: Nvidia's high-performance computing platform uses massive multithreading. Microprocessor Report 22, 1–8.

Hengl, T., Reuter, H.I., 2008. Developments in soil science, In: Geomorphometry: Concepts, Software, Application, vol. 33. Elsevier, Amsterdam, Netherlands, 707 pp.

Jenson, S.K., Domingue, J.O., 1988. Extracting topographic structure from digital elevation data for geographical information system analysis. Photogrammetric Engineering and Remote Sensing 54 (11), 1593–1600.

Kim, S., Lee, H., 2004. A digital elevation analysis: a spatially distributed flow apportioning algorithm. Hydrological Processes 18 (10), 1777–1794.

Lee, V.W., Kim, C., Chhuqani, J., Deisher, M., Kim, D., Nquyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Sinqhal, R., Dubey, P., 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, New York, pp. 451–460.

Martz, L.W., de Jong, E., 1988. Catch: a fortran program for measuring catchment area from digital elevation models. Computers & Geosciences 14 (5), 627–640.

O'Callaghan, J.F., Mark, D.M., 1984. The extraction of drainage networks from digital elevation data. Computer Vision, Graphics, and Image Processing 28 (3), 323–344.

Ortega, L., Rueda, A., 2010. Parallel drainage network computation on CUDA. Computers & Geosciences 36 (2), 171–178.

Planchon, O., Darboux, F., 2001. A fast, simple, and versatile algorithm to fill the depressions of digital elevation models. Catena 46 (2–3), 159–176.

Quinn, P.F., Beven, K.J., Chevallier, P., Planchon, O., 1991. The prediction of hillslope flow paths for distributed hydrological modeling using digital terrain models. Hydrological Processes 5 (1), 59–79.

Quinn, P.F., Beven, K.J., Lamb, R., 1995. The $\ln(\alpha/\tan\beta)$ index: how to calculate it and how to use it within the Topmodel framework. Hydrological Processes 9 (2), 161–182.

Qin, C.Z., Zhu, A.-X., Pei, T., Li, B.L., Zhou, C., Yang, L., 2007. An adaptive approach to selecting a flow-partition exponent for a multiple-flow-direction algorithm. International Journal of Geographical Information Science 21 (4), 443–458.

Qin, C.-Z., Zhu, A.-X., Pei, T., Li, B.-L., Scholten, T., Behrens, T., Zhou, C.-H., 2011. An approach to computing topographic wetness index based on maximum downslope gradient. Precision Agriculture 12 (1), 32–43.

Reif, J.H., 1985. Depth-first search is inherently sequential. Information Processing Letters 20 (5), 229–234.

Tarboton, D.G., 1997. A new method for the determination of flow directions and contributing areas in grid digital elevation models. Water Resources Research 33 (2), 309–319.

Tarboton, D.G., Schreuders, K.A.T., Watson, D.W., Baker, M.E., 2009. Generalized terrain-based flow analysis of digital elevation models. In: Proceedings of the 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation, Cairns, Australia, pp. 2000–2006.

Tesfa, T.K., Tarboton, D.G., Waston, D.W., Schreuders, K.A.T., Baker, M.E., Wallace, R.M., 2011. Extraction of hydrological proximity measures from DEMs using parallel processing. Environmental Modelling & Software 26 (12), 1696–1709.

Tukora, B., Szalay, T., 2008. High-performance computing on graphics processing units. International Journal for Engineering and Information Sciences 3 (2), 27–34.

Wallis, C., Watson, D., Tarboton, D., Wallace, R., 2009. Parallel flow-direction and contributing area calculation for hydrology analysis in digital elevation models. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, pp. 467–472.

Weiss, M.A. (Ed.), 2nd ed. Addison-Wesley, Menlo Park, California. (600 pp.).

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.), Advances in Terrain Analysis. Springer, New York, pp. 213–236.

Wilson, J.P., Gallant, J.C. (Eds.), 2000. Terrain Analysis: Principles and Applications. Wiley, New York, NY, 479 pp.

Wolock, D.M., McCabe, G.J., 1995. Comparison of single and multiple flow direction algorithms for computing topographic parameters in Topmodel. Water Resources Research 31 (5), 1315–1324.

Xia, Y., Li, Y., Shi, X., 2010. Parallel viewshed analysis on GPU using CUDA. In: Proceedings of the Third International Joint Conference on Computational Sciences and Optimization, Huangshan, China. Piscataway, NJ, pp. 373–374.

Xu, R., Huang, X.X., Luo, L., Li, S.C., 2010. A new grid-associated algorithm in distributed hydrological model simulations. Science in China (Technological Sciences) 53 (1), 235–241.