Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Wydział Matematyki, Fizyki i Informatyki

Bartłomiej Kotyra

Wydajne algorytmy równoległe w modelowaniu hydrologicznym

Rozprawa doktorska przygotowana pod opieką
dr. hab. Przemysława Stpiczyńskiego, prof. UMCS

Lublin, 2024

Składam serdeczne podziękowania
profesorowi dr. hab. Przemysławowi Stpiczyńskiemu
za opiekę naukową, wsparcie i nadanie kierunku
przeprowadzonym pracom badawczym.

Spis treści

Lista publikacji zawartych w rozprawie doktorskiej

1. Kotyra B., Chabudziński Ł., Stpiczyński P.
   High-performance parallel implementations of flow accumulation algorithms for multicore architectures
   Computers & Geosciences, vol. 151, 2021, 104741
   https://doi.org/10.1016/j.cageo.2021.104741

2. Kotyra B.
   High-performance watershed delineation algorithm for GPU using CUDA and OpenMP
   Environmental Modelling & Software, vol. 160, 2023, 105613
   https://doi.org/10.1016/j.envsoft.2022.105613

3. Kotyra B., Chabudziński Ł.
   Fast parallel algorithms for finding the longest flow paths in flow direction grids
   Environmental Modelling & Software, vol. 167, 2023, 105728
   https://doi.org/10.1016/j.envsoft.2023.105728

Streszczenie

Celem prac przeprowadzonych i opisanych w ramach niniejszej rozprawy było opracowanie nowych, wydajnych algorytmów równoległych, przeznaczonych do realizacji wybranych zadań z obszaru modelowania hydrologicznego i systemów informacji geograficznej (GIS). Rozważane zagadnienia obejmowały obliczanie akumulacji spływu powierzchniowego, wyznaczanie zlewni oraz identyfikację najdłuższych ścieżek spływu. Zaproponowane algorytmy zostały poddane szczegółowej ocenie i porównane z innymi istniejącymi rozwiązaniami, zarówno opisanymi w literaturze, jak i dostępnymi w powszechnie stosowanych pakietach oprogramowania. Uzyskane wyniki pozwalają wykazać istotną przewagę opracowanych algorytmów nad alternatywnymi rozwiązaniami przeznaczonymi do realizacji tych samych zadań. Co ważne, zaproponowane koncepcje pozwalają na znacznie wydajniejsze wykorzystanie własności współczesnych architektur wielordzeniowych.

Niniejszą rozprawę stanowi zbiór trzech powiązanych tematycznie publikacji naukowych. Każda porusza odrębny problem obliczeniowy i prezentuje opracowane przez autora rozwiązania, a także ocenia ich wartość w kontekście dotychczasowego stanu badań.

W rozdziale 1 nakreślono kontekst i tematykę zrealizowanych prac. Przedstawiono tutaj zakres i najważniejsze cele badań, a także zdefiniowano tezę niniejszej rozprawy.

Rozdział 2 stanowi krótkie wprowadzenie do obszaru obliczeń równoległych. Omówiono w nim zagadnienia stanowiące wspólne tło dla wszystkich trzech publikacji.

W rozdziale 3 przedstawiono wybrane tematy z zakresu modelowania hydrologicznego i systemów informacji geograficznej. Opisano tutaj zagadnienia obliczeniowe, będące głównym punktem zainteresowania przeprowadzonych badań.

Rozdział 4 omawia kolejno trzy publikacje zawarte w rozprawie. Przedstawiono w nim zakres zrealizowanych prac, uwzględniając przyjęte założenia, wybrane metody i zastosowane technologie. Zaprezentowano także najważniejsze z uzyskanych rezultatów.

W rozdziale 5 zawarto krótkie podsumowanie wykonanych prac badawczych, z uwzględnieniem najważniejszych wyników i kluczowych wniosków. Zaproponowano także potencjalne kierunki dalszych badań.

Załączniki A, B oraz C zawierają treść kolejnych publikacji i tym samym szczegółowo prezentują prace zrealizowane w ramach niniejszej rozprawy.

Abstract

The goal of the work presented in this dissertation was to develop new, efficient parallel algorithms for selected problems in hydrological modeling and geographic information systems (GIS). The issues addressed included calculating flow accumulation, delineating watersheds and identifying the longest flow paths. The proposed algorithms were closely evaluated and compared with other existing solutions, both from the literature and commonly used software packages. The results demonstrate significant advantages of the developed algorithms over alternative solutions designed to perform the same tasks. Importantly, the proposed concepts allow for much more efficient use of modern multi-core architectures.

The dissertation consists of three thematically related research publications. Each of them addresses a separate computational problem and presents the solutions developed by the author, assessing their value in relation to the current state of research.

Chapter 1 outlines the background and topics of the research conducted. It specifies the scope and most important objectives, as well as the thesis of this dissertation.

Chapter 2 provides a brief introduction to the area of parallel computing. It covers issues that constitute the common background for all three publications.

Chapter 3 presents selected topics related to hydrological modeling and geographic information systems. The computational issues that were the main focus of this research are outlined here.

Chapter 4 describes the three publications included in the dissertation. It presents the scope of research work, discussing the assumptions, methods and technologies used. The most important results were also highlighted.

Chapter 5 presents a short summary of all the research work carried out, including the most important results and key conclusions. Potential directions for further research were also proposed.

Appendices A, B and C contain the published research papers and therefore present in detail the work carried out within this dissertation.

Rozdział 1

Wstęp

Historia modelowania hydrologicznego sięga lat 50. XIX wieku, jednak okres najbardziej dynamicznego rozwoju tego obszaru rozpoczął się dopiero wraz z początkiem rewolucji cyfrowej. Rosnąca dostępność komputerów stworzyła możliwości obliczeniowe, jakie nigdy wcześniej nie były osiągalne. Podobnie jak w wielu innych dziedzinach, gwałtowny rozwój nowych technologii doprowadził do szybkich i znaczących postępów, fundamentalnie zmieniając charakter całego obszaru [44].

Jednym z czynników mających istotny wpływ na kierunek rozwoju modelowania hydrologicznego były narodziny, a następnie upowszechnienie się systemów informacji geograficznej (GIS) w ostatnich dekadach XX wieku. Choć z początku technologia GIS znajdowała zastosowanie głównie w innych obszarach, z biegiem czasu jej możliwości zaczęto dostrzegać i wykorzystywać także w kontekście zagadnień związanych z hydrologią. Obecnie systemy GIS stanowią nieodłączny element dużej części badań hydrologicznych [43].

W ciągu ostatnich kilku dekad przemysł sprzętu komputerowego doświadczył wielu istotnych przemian. Jedną z najważniejszych był zwrot w kierunku procesorów wielordzeniowych, których upowszechnienie w ogromnym stopniu zwiększyło możliwości obliczeniowe urządzeń dostępnych dla przeciętnego użytkownika. Należy jednak podkreślić, że efektywne korzystanie z tych zasobów możliwe jest jedynie za pośrednictwem oprogramowania, które zostało zaprojektowane i zaimplementowane z myślą o przetwarzaniu równoległym [17].

W tym kontekście na szczególną uwagę zasługują współczesne procesory graficzne (GPU). Choć pierwotnym przeznaczeniem urządzeń tej klasy było wydajne przetwarzanie grafiki, na przestrzeni ostatnich dwóch dekad ich możliwości zostały z powodzeniem zaadaptowane do szerokiej gamy innych zastosowań. Architektury nowoczesnych jednostek GPU, umożliwiające przeprowadzanie obliczeń równoległych z wykorzystaniem setek czy tysięcy rdzeni, pozwalają na osiągnięcie poziomów wydajności dalece wykraczających

poza ograniczenia konwencjonalnych procesorów. Nowe możliwości sprzętowe, wraz z upowszechnieniem standardów programowania GPU takich jak CUDA czy OpenCL, doprowadziły do głębokich przemian w obszarze wysokowydajnych obliczeń komputerowych [5, 20].

Należy podkreślić, że pomimo rosnącej dostępności nowoczesnych jednostek obliczeniowych i standardów ich programowania, projektowanie i implementacja wydajnych algorytmów równoległych wciąż pozostaje relatywnie trudnym zadaniem. W szczególności programowanie urządzeń GPU, ze względu na ich fundamentalnie odmienną architekturę sprzętową, jest często uważane za wyspecjalizowaną dziedzinę [20]. Tradycyjne algorytmy sekwencyjne nie pozwalają na pełne wykorzystanie współczesnych zasobów sprzętowych, a ich adaptacja do uwarunkowań przetwarzania równoległego często okazuje się złożonym zagadnieniem [36].

Ostatnie dekady to jednocześnie okres szybkiego rozwoju technik pozyskiwania i gromadzenia danych geoprzestrzennych, będących podstawą funkcjonowania systemów GIS. Rozdzielczość i precyzja dostępnych zbiorów danych znacząco wzrosły, dalece przewyższając dotychczasowe standardy. Z jednej strony obszerne i łatwo osiągalne zasoby tworzą nowe możliwości, pozwalając na przeprowadzanie bardziej złożonych i dokładniejszych obliczeń oraz symulacji. Z drugiej, ich praktyczne wykorzystanie wiąże się z nowymi wyzwaniami, wynikającymi z konieczności przetwarzania danych o niespotykanych dotąd rozmiarach [4, 49].

Istniejąca literatura sugeruje, że duża część opracowanych jeszcze w latach 80. i 90. algorytmów przeznaczonych do przetwarzania danych geoprzestrzennych wciąż jest powszechnie implementowana i wykorzystywana we współczesnym oprogramowaniu GIS. Obecnie w wielu sytuacjach wydajność tych narzędzi okazuje się dalece niewystarczająca. Rozmiary dostępnych zbiorów danych geoprzestrzennych różnią się o rzędy wielkości od tych, dla których pierwotnie przeznaczone były te rozwiązania. Choć dzisiejszy sprzęt komputerowy oferuje nieosiągalne wcześniej możliwości obliczeniowe, istniejące algorytmy często okazują się nieadekwatne do współczesnych realiów [49].

Celem prac przedstawionych w niniejszej rozprawie było opracowanie nowych, wydajnych algorytmów rozwiązujących wybrane problemy z obszaru modelowania hydrologicznego i systemów informacji geograficznej. Szczególną uwagę poświęcono możliwościom przetwarzania równoległego współczesnych jednostek obliczeniowych – zarówno wielordzeniowych procesorów CPU, jak i urządzeń GPU. Główną motywację stanowiła potrzeba opracowania rozwiązań, które pozwoliłyby przekroczyć ograniczenia związane z wydajnością istniejącego oprogramowania.

Przeprowadzone prace koncentrowały się na trzech powiązanych ze sobą zagadnieniach: obliczaniu akumulacji spływu powierzchniowego, wyznaczaniu zlewni i identyfikacji

najdłuższych ścieżek spływu. Realizacja tych zadań z wykorzystaniem technologii GIS i danych geoprzestrzennych nie jest nową koncepcją – pierwsze algorytmy przeznaczone do tego celu zaproponowano w literaturze już w latach 80. i 90. Nowsze publikacje podkreślają jednak czasochłonność lub wręcz niepraktyczność wykonywania tych operacji za pomocą istniejących narzędzi w kontekście współczesnych, obszernych zbiorów danych. We wszystkich trzech przypadkach za najważniejszy cel realizowanych prac przyjęto opracowanie oryginalnych algorytmów równoległych, których wydajność istotnie przewyższałaby dostępne dotąd rozwiązania.

Teza niniejszej rozprawy została zdefiniowana następująco:

Odpowiednio zaprojektowane i zaimplementowane algorytmy równoległe, przeznaczone do rozwiązywania zadań z obszaru modelowania hydrologicznego i systemów informacji geograficznej, mogą pozwolić na istotną poprawę wydajności obliczeń w stosunku do obecnie stosowanych rozwiązań.

Rozdział 2

Obliczenia równoległe

Historia rozwoju komputerów jest nierozerwalnie związana z zagadnieniami dotyczącymi prędkości wykonywania obliczeń. Od czasu wdrożenia pierwszych elektronicznych, cyfrowych komputerów w latach 40. XX wieku, rozwój tej technologii był w ogromnym stopniu kierowany potrzebą uzyskania wyższej mocy obliczeniowej. Rosnąca na przestrzeni dekad wydajność dostępnych urządzeń nie tylko drastycznie zredukowała czas potrzebny na wykonanie wielu złożonych zadań, ale również istotnie wpłynęła na to, jakie rodzaje problemów są dzisiaj uważane za możliwe do rozwiązania za pomocą komputerów [21, 36].

Jednym z przełomowych zdarzeń w historii rozwoju sprzętu komputerowego było upowszechnienie procesorów wielordzeniowych, rozpoczynające się w pierwszej dekadzie XXI wieku. Konstrukcja urządzeń wieloprocesorowych nie była już wówczas w żadnym stopniu nową ideą – komputery równoległe znajdowały praktyczne zastosowanie w wyspecjalizowanych obszarach już od kilkudziesięciu lat. Jednak dopiero zwrot w kierunku tych rozwiązań wśród najważniejszych producentów układów scalonych pozwolił na udostępnienie urządzeń tej klasy szerokiej grupie odbiorców [17, 21].

Wraz z nowymi możliwościami sprzętowymi pojawiły się także wyzwania o nowym charakterze. Obecny w latach 80. i 90. trend wzrostu mocy obliczeniowej tradycyjnych, jednordzeniowych procesorów gwarantował coraz bardziej wydajną realizację zadań zlecanych komputerowi z użyciem już istniejącego oprogramowania. Zwrot w kierunku procesorów wielordzeniowych zmienił tę sytuację – wprawdzie nowe jednostki obliczeniowe oferowały niespotykane dotąd możliwości, jednak ich efektywne wykorzystanie nie odbywało się automatycznie. Wzrost wydajności związany z równoległym wykonywaniem operacji na wielu rdzeniach procesora jest osiągalny jedynie za pośrednictwem odpowiednio zaprojektowanego i zaimplementowanego oprogramowania [38, 48].

Jak zaznaczono w [36]: „wyzwaniem związanym z przetwarzaniem równoległym nie jest sprzęt; jest nim fakt, że zbyt mało istotnych programów zostało przepisanych tak, aby umożliwić szybszą realizację zadań na komputerach wieloprocesorowych”.

2.1 Komputery sekwencyjne

W latach 60. XX wieku, Michael J. Flynn zaproponował relatywnie prosty sposób klasyfikacji architektur komputerowych w oparciu o liczbę równocześnie przetwarzanych strumieni instrukcji i strumieni danych [13, 14]. Stosowana do dzisiaj klasyfikacja, określana obecnie jako taksonomia Flynna, wyróżnia cztery główne klasy architektur:

- SISD – pojedynczy strumień rozkazów, pojedynczy strumień danych
- SIMD – pojedynczy strumień rozkazów, wiele strumieni danych
- MISD – wiele strumieni rozkazów, pojedynczy strumień danych
- MIMD – wiele strumieni rozkazów, wiele strumieni danych

Kategoria SISD obejmuje konwencjonalne komputery odpowiadające architekturze von Neumanna. Urządzenia tej klasy są wyposażone w pojedynczy procesor, realizujący strumień rozkazów jako sekwencję kolejnych kroków – dlatego zwykle określane są jako komputery sekwencyjne (ang. sequential computers). Zanim procesory wielordzeniowe stały się dostępne dla szerszej grupy użytkowników, architektury należące do kategorii SISD stanowiły powszechnie przyjęty standard [38].

Historycznie większość oprogramowania komputerowego powstawała z myślą o architekturach SISD. Programy przeznaczone do uruchamiania na urządzeniach tej klasy są interpretowane jako jednowątkowa sekwencja instrukcji. Tym samym w trakcie realizacji programu procesor generalnie wykonuje rozkazy jeden po drugim [1, 31].

Jednym z najistotniejszych czynników mających wpływ na wydajność pracy konwencjonalnego procesora jest częstotliwość taktowania, określająca długość cyklu, w trakcie którego procesor wykonuje pojedynczą, elementarną operację. Lata 80. i 90. stanowiły okres szczególnie intensywnego wzrostu częstotliwości taktowania dostępnych na rynku procesorów, jednak ten trend załamał się w pierwszej dekadzie XXI wieku [19]. Kluczowym powodem był fakt, iż wyższa częstotliwość pracy procesora pociąga za sobą także wyższe zużycie energii, co z kolei przekłada się na zwiększoną emisję ciepła. Problemy związane z chłodzeniem układów okazały się tym samym barierą ograniczającą możliwość dalszego wzrostu częstotliwości taktowania procesorów [31, 38].

Rosnąca wydajność procesorów jednordzeniowych naturalnie wiązała się z możliwością szybszego wykonywania dostępnych programów komputerowych przez nowsze jednostki. Utrzymujący się na przestrzeni dekad trend pozwalał oczekiwać, że urządzenia kolejnych generacji w naturalny sposób zagwarantują jeszcze bardziej wydajną realizację zadań bez konieczności wprowadzania istotnych zmian w już istniejącym oprogramowaniu. Rezygnacja przemysłu z dalszego dążenia do szybszej pracy pojedynczych rdzeni oznaczała, że ta sytuacja nie będzie miała już dłużej miejsca. Wprawdzie pojawiające się na rynku procesory wielordzeniowe oferowały znacznie większe możliwości obliczeniowe, ale ich

efektywne wykorzystanie za pomocą istniejących, sekwencyjnych programów komputerowych nie było możliwe. Charakter oprogramowania przeznaczonego dla architektur SISD ogranicza sposób jego wykonania do jednowątkowej pracy pojedynczego rdzenia procesora, bez względu na architekturę wykorzystywanego urządzenia [31, 39].

2.2 Procesory wielordzeniowe

Począwszy od pierwszej dekady XXI wieku, nowym standardem stały się procesory zawierające wiele fizycznych jednostek CPU (określanych jako rdzenie procesora). Obecnie niemal wszystkie komputery stacjonarne i serwery są wyposażone w procesory wielordzeniowe [36].

Architektury typowych, wielordzeniowych procesorów CPU należą do kategorii MIMD w taksonomii Flynna. Jednostki tej klasy pozwalają na jednoczesne wykonywanie wielu strumieni rozkazów na oddzielnych strumieniach danych. Z punktu widzenia programisty oznacza to możliwość wyrażenia zadania do wykonania w postaci zbioru niezależnych podzadań, które urządzenie może realizować równolegle, skracając tym samym czas potrzebny na uzyskanie końcowego wyniku [1].

Liczba dostępnych rdzeni jest zależna od architektury konkretnego modelu procesora. Jednostki dwurdzeniowe, które stały się powszechnie dostępne w pierwszej dekadzie XXI wieku, są od tego czasu stopniowo wypierane przez rozwiązania nowszych generacji. Obecnie normą na rynku konsumenckim jest dostępność procesorów cztero-, sześcio- czy ośmiordzeniowych. Jednocześnie liczby rdzeni dostępnych w jednostkach przeznaczonych do bardziej wymagających zastosowań nierzadko osiągają nawet trzycyfrowe wartości. Oczekuje się, że ten trend w rozwoju architektur CPU będzie trwał nadal [31, 36, 39].

Należy przy tym podkreślić, że wykorzystanie mocy obliczeniowej oferowanej przez dodatkowe rdzenie procesora nie odbywa się w sposób automatyczny. Kluczowym warunkiem jest skonstruowanie programu komputerowego w taki sposób, aby realizowane zadanie mogło zostać zinterpretowane jako wiele odrębnych sekwencji rozkazów, które mogą być przetwarzane równocześnie. Dopiero taka struktura oprogramowania pozwala na przydział pracy do więcej niż jednego rdzenia procesora i tym samym umożliwia wykonanie programu w sposób równoległy [31].

2.3 Procesory GPU

Odrębną kategorię urządzeń, mających istotne znaczenie w obszarze wysokowydajnych obliczeń równoległych, stanowią współczesne procesory graficzne (GPU). Historycznie jednostki należące do tej klasy wywodzą się z wyspecjalizowanych procesorów projekto-

wanych z myślą o szybkim przetwarzaniu grafiki 3D. Z uwagi na ten charakterystyczny, pierwotnie zakładany zakres zastosowań, architektury tych urządzeń znacząco różnią się od tych stosowanych w produkcji konwencjonalnych procesorów CPU [5, 20].

W anglojęzycznej literaturze architektury współczesnych jednostek GPU określane są terminem „manycore" (w odróżnieniu do wyrażenia „multi-core", używanego głównie wobec konwencjonalnych procesorów wielordzeniowych). Stosowanie odrębnego terminu jest konsekwencją wykorzystywania rozwiązań opartych na wyjątkowo wysokich liczbach rdzeni, często wyrażanych w setkach lub tysiącach. Należy jednak podkreślić, że rdzenie GPU mają fundamentalnie prostszą konstrukcję niż te spotykane w dzisiejszych procesorach CPU, tym samym oferując możliwości obliczeniowe o odmiennym charakterze [3, 8].

Architektura typowego, współczesnego procesora GPU może być w uproszczeniu interpretowana jako kolekcja jednostek należących do kategorii SIMD w taksonomii Flynna. Taka konstrukcja istotnie ogranicza charakter zadań obliczeniowych, które mogą być wykonywane na urządzeniach tej klasy przy efektywnym wykorzystaniu ich potencjału. Tym samym jednostki GPU nie są z natury procesorami ogólnego przeznaczenia – pomimo ich wysokiej mocy obliczeniowej, niektóre rodzaje zadań wciąż będą wykonywane wydajniej przez konwencjonalne procesory CPU [3, 39].

Urządzenia GPU wykazują swoją przewagę przede wszystkim w realizacji obliczeń o charakterze kompatybilnym z architekturą SIMD – czyli takich, gdzie pojedynczy ciąg instrukcji jest równolegle wykonywany na wielu strumieniach danych. Tym samym zadania, w których zastosowanie GPU może przynieść najwięcej korzyści, to te o strukturze przypominającej przetwarzanie grafiki, gdzie dla każdego punktu obrazu wykonywane są te same operacje [39].

Dopiero w ciągu ostatnich dwóch dekad procesory GPU znalazły szersze, praktyczne zastosowanie w zagadnieniach niezwiązanych bezpośrednio z grafiką komputerową. Obecnie urządzenia tej klasy są z powodzeniem używane w niemal wszystkich obszarach, w których wydajność obliczeń komputerowych ma istotne znaczenie. Należy jednak podkreślić, że warunkiem efektywnego wykorzystania możliwości oferowanych przez jednostki GPU jest formułowanie zadań obliczeniowych w sposób kompatybilny z ich specyficzną architekturą. Z tego względu programowanie GPU diametralnie różni się od tworzenia oprogramowania dla CPU i jest często postrzegane jako wyspecjalizowana dziedzina [5, 8, 20].

Podobnie jak w przypadku wielordzeniowych procesorów CPU, rozwój jednostek GPU prowadzi do stopniowego wzrostu liczby dostępnych rdzeni w urządzeniach kolejnych generacji. Oczekuje się, że ten trend będzie kontynuowany w kolejnych latach [31].

2.4 Standard OpenMP

Rozwój architektur wielordzeniowych naturalnie wiązał się z potrzebą opracowania nowych standardów programowania, pozwalających tworzyć aplikacje wykorzystujące możliwości przetwarzania równoległego. Obecnie jednym z najważniejszych i najszerzej stosowanych jest standard OpenMP [6]. Historia OpenMP sięga lat 90. – pierwsza wersja specyfikacji została opublikowana w roku 1997. Standard jest wciąż rozwijany, mając szerokie wsparcie ze strony przemysłu oprogramowania komputerowego [31, 39].

OpenMP nie jest samodzielnym językiem programowania. Standard ma formę rozszerzenia, którego składnia może być stosowana w ramach języków C, C++ oraz Fortran. Implementacja programów z wykorzystaniem OpenMP polega na umieszczaniu w kodzie źródłowym określonych instrukcji i dyrektyw, wyrażających oczekiwany sposób realizacji zadania na urządzeniu umożliwiającym przetwarzanie równoległe. Zadaniem kompilatorów wspierających standard OpenMP jest interpretacja tych konstrukcji, a następnie generowanie kodu, który będzie wykonywany przez maszynę w żądany sposób [3, 7, 31].

Standard OpenMP opiera się na modelu pamięci współdzielonej (ang. shared-memory). Tym samym zakłada się, że program zaimplementowany z użyciem OpenMP będzie wykonywany przez grupę procesorów mających dostęp do wspólnej przestrzeni adresowej. Taki model ogranicza kontekst, w którym standard OpenMP może być stosowany, wykluczając systemy rozproszone (składające się z niezależnych urządzeń). Z tego względu możliwości oferowane przez OpenMP są często wykorzystywane w połączeniu z innymi standardami, które pozwalają przekroczyć te ograniczenia [7, 39].

2.5 Platforma CUDA

Platforma CUDA została opracowana i udostępniona przez korporację NVidia jako środowisko umożliwiające programowanie i wykonywanie obliczeń równoległych na jednostkach GPU. Ze względu na własnościowy charakter projektu i jego ścisły związek z architekturami sprzętu dostarczanego przez NVidia, oprogramowanie tworzone z użyciem CUDA może być uruchamiane wyłącznie na urządzeniach GPU pochodzących od tego producenta. Mimo to, CUDA obecnie dominuje wśród technologii natywnego programowania GPU [3, 39].

Model programowania CUDA zakłada odrębność jednostki GPU od procesora CPU i jego pamięci (określanych terminem „host") – urządzenie GPU posiada własną, niezależną przestrzeń adresową. Typowy scenariusz realizacji zadań obliczeniowych obejmuje transfer danych wejściowych do pamięci GPU, wykonanie zleconych operacji, a następnie pobranie uzyskanych wyników z powrotem do pamięci hosta [8].

Ze względu na specyficzną architekturę jednostek GPU, programy implementowane z użyciem CUDA muszą mieć określoną formę. Kluczowy element stanowią funkcje określane terminem „kernel", których uruchomienie i wykonanie na GPU jest zlecane z poziomu hosta. Z punktu widzenia implementacji, kod źródłowy kernela ma strukturę programu sekwencyjnego, jednak jego realizacja może odbywać się w sposób równoległy (przez wiele wątków wykonujących niezależnie ten sam zestaw instrukcji) [8, 47].

Wątki wykonujące zadania na GPU są zorganizowane w dwupoziomową hierarchię: grupę wątków określa się terminem „blok" (ang. block), natomiast grupę bloków nazywa się „siatką" (ang. grid). Wywołanie kernela na GPU wiąże się z określeniem wymiarów siatki przeznaczonej do jego wykonania. Chociaż każdy wątek w siatce realizuje kod tego samego kernela, to dostępność informacji o własnym położeniu w strukturze bloków umożliwia wątkom odpowiedni podział pracy i tym samym wydajną, równoległą realizację zadania [3].

Podobnie jak w przypadku OpenMP, specyficzne możliwości oferowane przez technologię CUDA są często łączone z innymi standardami programowania równoległego [3, 31].

Rozdział 3

Wybrane zagadnienia z zakresu modelowania hydrologicznego i systemów informacji geograficznej

Hydrologia stanowi jeden z kluczowych działów geografii fizycznej, zajmujący się rolą i właściwościami wody występującej w środowisku ziemskim. Jako dyscyplina naukowa, hydrologia dąży przede wszystkim do pogłębienia wiedzy dotyczącej procesów związanych z obiegiem wody w przyrodzie. Badania hydrologiczne prowadzą do opracowywania praktycznych metod redukcji ryzyka związanego z powodziami, suszami, erozją gleby czy osuwiskami [26, 33].

Modelowanie hydrologiczne jest relatywnie szerokim pojęciem, obejmującym wiele zróżnicowanych metod i technik. Wspólnym celem stosowanych modeli jest szacowanie i przewidywanie wartości zmiennych, mających znaczenie dla badań i projektów hydrologicznych. W ogólności, modele hydrologiczne stanowią uproszczoną, zwykle opisaną matematycznie reprezentację rzeczywistego systemu wodnego wraz z prawami rządzącymi jego zachowaniem [33].

Badania wykorzystujące modelowanie hydrologiczne są zwykle przeprowadzane w skali obejmującej systemy wodne całych zlewni. Warunkiem koniecznym do ich przeprowadzania jest dostępność odpowiednich danych dotyczących badanego obszaru. Obecnie kluczową rolę w pozyskiwaniu i przetwarzaniu tych danych pełnią systemy informacji geograficznej (GIS) [33, 42].

Pojęciem GIS określa się zbiorczo technologię komputerową przeznaczoną do gromadzenia, przechowywania i przetwarzania danych geoprzestrzennych (powiązanych z położeniem na powierzchni Ziemi). Fundamentalnie systemy GIS są narzędziami ogólnego przeznaczenia o wielu zróżnicowanych zastosowaniach – ich przydatność w zagadnieniach hydrologicznych została dostrzeżona i doceniona dopiero z biegiem czasu. Ze względu na

przestrzenny charakter procesów leżących w centrum zainteresowania hydrologii, integracja możliwości oferowanych przez GIS jest dzisiaj traktowana jako naturalny, a często nieodłączny element badań w tym obszarze. Narzędzia GIS są obecnie powszechnie stosowane w procesie opracowywania danych dla modeli hydrologicznych [43, 44, 55].

Pozostałe sekcje tego rozdziału koncentrują się na omówieniu wybranych zagadnień, leżących na pograniczu modelowania hydrologicznego i systemów informacji geograficznej. Ze względu na obszerność obu działów, materiał został uproszczony i ograniczony do tematów mających bezpośredni związek z publikacjami zawartymi w niniejszej rozprawie.

3.1 Numeryczne modele terenu

Pojęciem numerycznego modelu terenu (ang. digital elevation model, DEM) określa się trójwymiarową, cyfrową reprezentację fragmentu powierzchni Ziemi. W ogólności modele DEM pozwalają na przybliżone określenie ukształtowania terenu, dostarczając informacji o wysokości wzniesienia w danych lokalizacjach [43].

Istniejące rodzaje numerycznych modeli terenu różnią się od siebie sposobem reprezentacji danych. W obszarze GIS powszechnie wykorzystywana jest forma rastrowa, w której DEM stanowi siatkę kwadratowych komórek, zawierających wartości określające wysokość powierzchni terenu w odpowiadających im lokalizacjach (względem przyjętego punktu odniesienia). Choć inne rodzaje modeli DEM również znajdują swoje zastosowania, w praktyce to właśnie forma rastrowa jest spotykana najczęściej [24, 33, 43]. Rycina 3.1 przedstawia uproszczony przykład tej reprezentacji.

7.8	8.0	7.7	7.8	7.9
7.7	7.8	7.6	7.7	7.8
7.6	7.4	7.7	7.8	7.9
7.3	7.4	7.5	7.7	7.8
7.3	7.5	7.6	7.8	7.7

Rysunek 3.1: Uproszczony przykład numerycznego modelu terenu w formie rastrowej

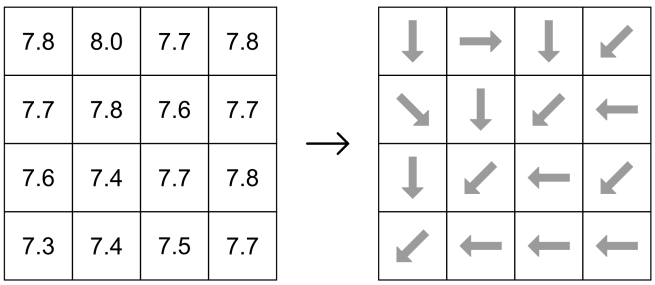
Obecnie numeryczne modele terenu są konstruowane głównie z wykorzystaniem technik teledetekcji i fotogrametrii. Z uwagi na intensywny rozwój technologii stosowanych w tych obszarach, w ciągu ostatnich dekad dostępność, rozdzielczość i precyzja modeli DEM znacząco wzrosły. Obecnie spotykane są już modele, w których pojedyncza komórka rastra reprezentuje obszar o powierzchni mniejszej niż jeden metr kwadratowy [4, 35, 52].

Nieprzetworzone modele DEM zawierają zwykle lokalne depresje – miejscowe obniżenia terenu otoczone komórkami o wyższych wartościach. Ich obecność może wynikać zarówno z rzeczywistego ukształtowania powierzchni, jak i nieuniknionych błędów pomiaru wysokości. Niezależnie od źródła pochodzenia, lokalne depresje stanowią przeszkodę dla poprawnego działania wielu algorytmów związanych z hydrologią. Z tego względu oryginalne modele DEM są często korygowane za pomocą procedur gwarantujących wyeliminowanie artefaktów tego rodzaju [50, 53].

Precyzja danych topograficznych okazuje się mieć wyjątkowo istotne znaczenie w zastosowaniach związanych z modelowaniem hydrologicznym [23, 27]. Tym samym rosnąca dokładność dostępnych modeli DEM jest naturalnie traktowana jako pożądane zjawisko. Należy jednak podkreślić, że wzrost rozdzielczości modeli przekłada się na coraz większe rozmiary zbiorów danych podlegających przetwarzaniu w ramach systemów GIS, co prowadzi do nowych wyzwań związanych z wydajnością wykorzystywanego oprogramowania [49].

3.2 Rastry kierunku spływu

Na podstawie modelu DEM (zwykle poddanemu odpowiedniej korekcji) możliwe jest wygenerowanie rastra kierunku spływu (ang. flow direction). Podstawowa koncepcja polega na przypisaniu każdej komórce modelu odpowiedniej wartości, pozwalającej określić w jaki sposób spływ jest kierowany z danej lokalizacji do sąsiednich komórek. Wynik tej operacji jest zwykle przechowywany w postaci odrębnego rastra (o wymiarach odpowiadających wejściowemu modelowi terenu) [54]. Rycina 3.2 przedstawia uproszczony przykład.



Rysunek 3.2: Przykład rastra kierunku spływu wygenerowanego na podstawie modelu DEM

Modelowanie spływu powierzchniowego w oparciu o raster „strzałek" wskazujących na sąsiednie komórki stało się szeroko stosowanym podejściem w latach 80. [22, 34]. Z biegiem czasu zaproponowano wiele modyfikacji, oryginalnych metod i nowych algorytmów

wyznaczania rastrów kierunku spływu [12, 15, 41, 51]. Fundamentalne koncepcje pozostają powszechnie stosowane do dzisiaj i są standardowo implementowane w pakietach oprogramowania GIS.

Metody generowania rastrów kierunku spływu można podzielić na dwie główne kategorie, określane w anglojęzycznej literaturze terminami „single-flow" oraz „multiple-flow". Podejście single-flow zakłada, że spływ jest kierowany z każdej komórki do tylko jednego z jej najbliższych sąsiadów. Z kolei metody należące do grupy multiple-flow dopuszczają kierowanie spływu do więcej niż jednego sąsiada (w określonych proporcjach). Oba podejścia znajdują praktyczne zastosowania [28, 40].

We wszystkich przypadkach podstawą do określenia kierunku spływu danej komórki jest lokalne nachylenie terenu, wyznaczane w oparciu o wartości modelu DEM. Prostsze metody koncentrują się jedynie na najbliższym sąsiedztwie każdej komórki, podczas gdy bardziej złożone rozwiązania uwzględniają również kontekst większego obszaru. Istnieją także algorytmy wprowadzające element losowy do podejmowanych decyzji w celu uzyskania rezultatów o bardziej naturalnej charakterystyce [12, 16].

Rastry kierunku spływu są z reguły wykorzystywane jako dane wejściowe dla kolejnych operacji związanych z modelowaniem hydrologicznym [54].

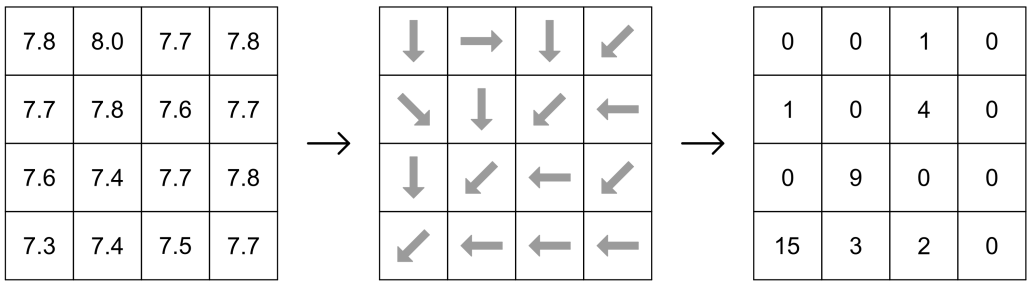
3.3 Akumulacja spływu powierzchniowego

Zagadnienie obliczania akumulacji spływu powierzchniowego zostało precyzyjnie zdefiniowane w literaturze już w latach 80. [22, 34]. Mimo istotnych przemian w obszarze GIS, zaproponowana wówczas koncepcja nadal funkcjonuje w praktycznie niezmienionej postaci. Powszechnie stosowane pakiety oprogramowania, przeznaczone do prac w zakresie hydrologii, z reguły oferują narzędzia umożliwiające wykonanie tej operacji.

Wartości akumulacji są wyznaczane na podstawie rastra kierunku spływu. Operacja polega na obliczeniu i przypisaniu każdemu elementowi rastra liczby wszystkich komórek, które kierują spływ do danej lokalizacji (zarówno bezpośrednio, jak i za pośrednictwem innych komórek). Tym samym wartość akumulacji koreluje z powierzchnią obszaru, z którego spływ jest kierowany do określonego punktu [22, 30].

Rycina 3.3 przedstawia uproszczony przykład tej operacji. Komórki, które nie posiadają ani jednego sąsiada kierującego spływ w ich stronę, otrzymały wartość 0. Pozostałym elementom rastra przypisano sumaryczną liczbę komórek, z których spływ może być transportowany przez daną lokalizację.

Rastry z wartościami akumulacji spływu stanowią podstawę do wyznaczenia sieci hydrograficznej danego obszaru. Wykorzystuje się je także do precyzyjnego lokalizowania komórek stanowiących ujścia zlewni [25].



Rysunek 3.3: Przykład obliczania akumulacji spływu powierzchniowego

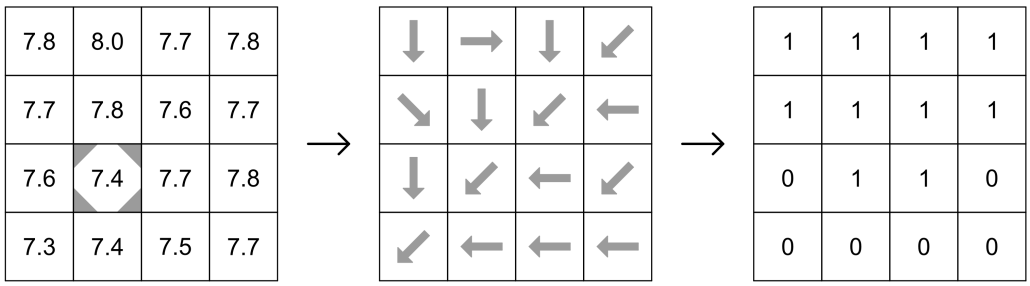
3.4 Wyznaczanie zlewni

Terminem „zlewnia" (ang. watershed, catchment, drainage basin) określa się obszar lądowy, z którego spływ trafia ostatecznie do pojedynczej lokalizacji, nazywanej ujściem (ang. watershed outlet). Tradycyjnie obszary zlewni były wyznaczane ręcznie za pomocą map topograficznych [10, 43].

Metody automatycznego wyznaczania zlewni z użyciem danych geoprzestrzennych były aktywnie badane od lat 80. [2, 22, 29]. Duża część koncepcji zaproponowanych w tym okresie znajduje zastosowanie do dzisiaj. Stosowane w praktyce operacje są w większości precyzyjnie zdefiniowane, jednoznacznie specyfikując oczekiwany wynik dla określonych danych wejściowych.

Najczęściej implementowane podejście opiera się na wykorzystaniu rastra kierunku spływu wraz z lokalizacją wybranego punktu ujścia. Operacja wyznaczania zlewni ma na celu zidentyfikowanie całego obszaru (ewentualnie jego granicy), z którego spływ powierzchniowy jest kierowany do wskazanej komórki (lub grupy komórek) [18, 25]. Wynik jest często generowany jako oddzielny raster, w którym komórki należące do obszaru zlewni zostają oznaczone odpowiednią wartością [22].

Rycina 3.4 przedstawia uproszczony przykład tej operacji. Jedna z lokalizacji została wybrana i wskazana jako punkt ujścia. Komórki zidentyfikowane jako należące do obszaru zlewni otrzymały wartość 1 (pozostałe oznaczono wartością 0).



Rysunek 3.4: Przykład wyznaczania zlewni w oparciu o raster kierunku spływu

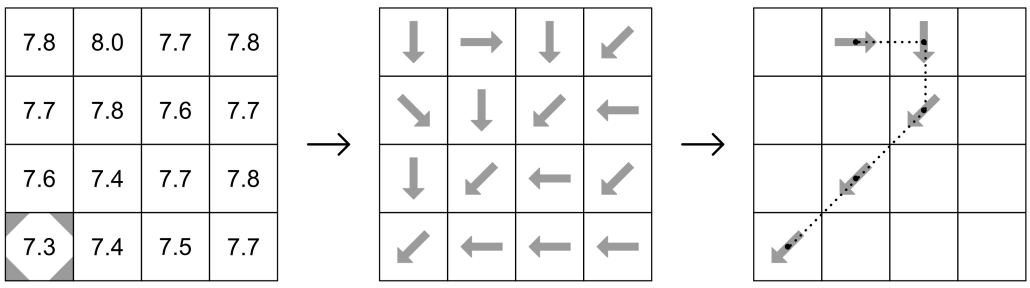
3.5 Identyfikacja najdłuższych ścieżek spływu

Najdłuższa ścieżka spływu (ang. longest flow path, longest drainage path) stanowi zagadnienie o szczególnie istotnym znaczeniu z perspektywy modelowania hydrologicznego. Operację identyfikacji najdłuższej ścieżki spływu przeprowadza się zwykle w obrębie obszaru wybranej zlewni. Następnie jej położenie oraz długość są wykorzystywane do określenia wartości parametrów modeli hydrologicznych, w szczególności czasu opóźnienia odpływu (ang. lag time) oraz czasu koncentracji odpływu (ang. time of concentration) [11, 32].

Pierwsze prace poświęcone automatycznej identyfikacji najdłuższych ścieżek spływu opublikowano w latach 90. [45, 46]. Podejście powszechnie stosowane w ramach systemów GIS opiera się na wykorzystaniu rastra kierunku spływu. Zwykle pojedyncza komórka rastra jest wskazywana jako punkt ujścia zlewni, a zadaniem algorytmu jest zidentyfikowanie najdłuższej ścieżki prowadzącej do tej lokalizacji.

Długość ścieżki spływu jest najczęściej szacowana jako suma odległości między centrami kolejnych komórek. Przyjmuje się, że dystans pomiędzy sąsiednimi elementami należącymi do tego samego wiersza lub tej samej kolumny jest równy długości boku pojedynczej komórki. Za odległość między elementami połączonymi diagonalnie przyjmuje się długość boku komórki przemnożoną przez $\sqrt{2}$. Pomimo względnej prostoty tego podejścia, uzyskiwane estymaty w przybliżeniu odpowiadają rzeczywistym długościom ścieżek spływu [12, 37].

Rycina 3.5 ilustruje uproszczony przykład tej operacji. Jedna z komórek została oznaczona jako ujście zlewni (i tym samym punkt końcowy poszukiwanej ścieżki spływu). Spośród wszystkich ścieżek prowadzących do wskazanej lokalizacji wybrana została ta o największej szacowanej długości.



Rysunek 3.5: Przykład identyfikacji najdłuższej ścieżki spływu

Rozdział 4

Omówienie uzyskanych wyników

W tym rozdziale opisano zawartość trzech powiązanych ze sobą publikacji naukowych, stanowiących kluczową część niniejszej rozprawy. Przedstawiono zakres prac zrealizowanych w ramach każdej z nich, a także omówiono zaproponowane rozwiązania i najważniejsze z uzyskanych wyników.

4.1 High-performance parallel implementations of flow accumulation algorithms for multicore architectures

Artykuł zatytułowany „High-performance parallel implementations of flow accumulation algorithms for multicore architectures" stanowi początek cyklu powiązanych tematycznie publikacji. Skoncentrowano się w nim na zagadnieniu akumulacji spływu powierzchniowego.

Celem przeprowadzonych badań było opracowanie możliwie szybkiego i skalowalnego algorytmu równoległego, przeznaczonego do obliczania macierzy akumulacji spływu. Bezpośrednią motywacją do podjęcia tego zagadnienia była niepraktyczna czasochłonność przeprowadzania tej operacji za pomocą powszechnie stosowanych narzędzi GIS, w szczególności w kontekście współczesnych, obszernych zbiorów danych. Prace były kierowane chęcią uzyskania możliwie jak największej redukcji czasu obliczeniowego potrzebnego do wygenerowania poprawnego wyniku. Szczególną uwagę poświęcono możliwościom przetwarzania równoległego oferowanym przez wielordzeniowe architektury CPU.

Definiując problem algorytmiczny założono, że dane wejściowe będzie stanowił raster kierunku spływu wygenerowany dowolną metodą typu single-flow. Oczekiwanym wynikiem działania algorytmu była macierz akumulacji spływu, w której wartość przypisana każdemu elementowi odpowiadała liczbie komórek kierujących spływ przez daną lokalizację. Przyjęto, że obliczenia będą przeprowadzane dla całego obszaru opisanego przez dane

wejściowe (każda komórka powinna otrzymać wartość akumulacji wynikającą z rastra kierunku spływu).

Na wczesnym etapie prac dostrzeżono, że spotykany w literaturze sposób definiowania obliczeń akumulacji spływu jako iteracyjnej symulacji przemieszczania się wody po powierzchni terenu jest nieadekwatny do faktycznej złożoności obliczeniowej zagadnienia. W rzeczywistości problem może zostać z powodzeniem rozwiązany w czasie liniowym. Odpowiednio zaprojektowane algorytmy wymagają jednorazowego przypisania wartości każdej komórce, unikając iteracyjnej aktualizacji stanu macierzy wynikowej. Ta obserwacja pozwoliła znacząco ograniczyć zakres rozpatrywanych koncepcji.

Implementacje, które wybrano do przedstawienia w ramach artykułu, bazowały na dwóch przeciwstawnych podejściach, oznaczonych jako „bottom-up" oraz „top-down". Koncepcja „bottom-up" opierała się na rozpoczynaniu obliczeń w lokalizacjach o najniższym położeniu i stopniowym przemieszczaniu się w stronę coraz wyższych komórek, przeciwnie do kierunku spływu. Algorytmy wykorzystujące to podejście zostały zaimplementowane w postaci rekurencyjnej, która w naturalny sposób odpowiadała tej formie przedstawienia problemu.

Podejście „top-down" zakładało rozpoczynanie obliczeń w komórkach źródłowych (nieposiadających sąsiadów kierujących spływ w ich stronę) i stopniowym przemieszczaniu się w kierunku coraz niższych lokalizacji. Algorytmy oparte na tym podejściu rozpoczynały pracę od zlokalizowania nieprzetworzonych jeszcze komórek źródłowych, a następnie poruszały się zgodnie z kierunkiem spływu, ustalając wartości elementów macierzy wynikowej.

Algorytmy oparte na obu podejściach zostały zaprojektowane z myślą o przetwarzaniu równoległym. W pierwszej kolejności zaimplementowano wersje sekwencyjne, kładąc przy tym nacisk na taką organizację zadań, która powinna umożliwić ich późniejszy podział między równolegle pracujące wątki. Następnie podjęto próby zredukowania czasu obliczeniowego algorytmów sekwencyjnych za pomocą technik programowania równoległego.

Implementacje opracowanych rozwiązań zostały przygotowane w języku C++. Równoległe wykonanie odpowiednich sekcji kodu osiągnięto za pomocą standardu OpenMP. W artykule opisano łącznie sześć implementacji – dwie sekwencyjne i cztery równoległe. Zrezygnowano z prezentacji innych rozwiązań, których wydajność okazała się znacznie odbiegać od najlepszych osiągniętych rezultatów.

Pomiary wydajności zostały przeprowadzone z wykorzystaniem 118 zróżnicowanych zestawów danych wejściowych. Numeryczny model terenu został skonstruowany w oparciu o zasoby udostępniane przez Główny Urząd Geodezji i Kartografii. Dane o jednometrowej rozdzielczości pokrywały fragment południowo-wschodniej Polski. W ramach tego obszaru wyselekcjonowano 59 zlewni o zróżnicowanych rozmiarach i charakterystykach, dla których następnie wygenerowano rastry kierunku spływu. W każdym przypadku dane

zostały przygotowane w dwóch formach: z uwzględnieniem tylko i wyłącznie komórek należących do danej zlewni, a także z uwzględnieniem wszystkich komórek leżących w obszarze prostokątnego rastra obejmującego zlewnię. Liczba niepustych elementów w tak przygotowanych rastrach rozciągała się w przybliżeniu od 46 milionów do 4,6 miliarda.

Jako środowisko do przeprowadzenia pomiarów wybrano maszynę wyposażoną w dwa procesory Intel Xeon E5-2670 v3 oraz koprocesor Intel Xeon Phi 7120P. Wybór był podyktowany chęcią zbadania skalowalności opracowanych algorytmów w kontekście możliwie dużej liczby równolegle pracujących wątków. Komputer był wyposażony w 128 GB pamięci RAM i działał pod kontrolą systemu Linux CentOS 7.0.

Uzyskane rezultaty wykazały znaczącą przewagę jednej z równoległych implementacji podejścia „top-down" nad wszystkimi pozostałymi rozpatrywanymi koncepcjami. Co ważne, wśród rozwiązań sekwencyjnych to algorytm oparty na idei „bottom-up" uzyskał najkrótsze czasy wykonania, jednak równoległe wersje tego podejścia okazały się oferować kilkukrotnie niższe przyspieszenie niż w przypadku „top-down". Wyjaśnieniem tych obserwacji mógłby być fakt, że implementacja „bottom-up" wymaga wykonania sumarycznie mniejszej liczby operacji obliczeniowych, podczas gdy struktura algorytmu „top-down" pozwala na bardziej zrównoważony przydział pracy do wątków.

Użycie koprocesora Intel Xeon Phi umożliwiło przeprowadzenie pomiarów z wykorzystaniem relatywnie dużej liczby rdzeni i tym samym ocenę skalowalności wybranych rozwiązań. Testy obejmowały zróżnicowane konfiguracje z liczbami równolegle pracujących wątków sięgającymi 240. Implementacja wyłoniona wcześniej jako najbardziej wydajna okazała się wciąż uzyskiwać coraz krótsze czasy wykonania wraz ze zwiększaniem liczby aktywnych rdzeni nawet w górnych granicach tego zakresu. Podczas eksperymentów z wykorzystaniem maksymalnej dostępnej liczby wątków, przyspieszenie uzyskiwane względem wersji sekwencyjnej osiągało wartości przekraczające 33.

Zakładanym punktem odniesienia dla uzyskanych wyników, wybranym we wczesnym stadium tych prac, była wydajność narzędzia Flow Accumulation dostępnego na platformie ArcGIS Desktop 10.6. Ostatecznie ze względu na niepraktyczną czasochłonność zrezygnowano z precyzyjnych pomiarów z wykorzystaniem tego oprogramowania. Zrealizowane, uproszczone porównania wykazały, że mając do dyspozycji identyczne środowisko sprzętowe, najbardziej wydajna z zaproponowanych implementacji realizuje te same zadania w czasie o dwa rzędy wielkości krótszym niż ArcGIS Desktop.

Kod źródłowy opracowanych algorytmów został udostępniony w publicznym repozytorium (`https://github.com/bkotyra/high_performance_flow_accumulation`).

Wkład autora rozprawy w tę publikację został oszacowany na 87%. Zakres wykonanych prac obejmował:

- przegląd literatury (część poświęcona algorytmom)
- zaprojektowanie i implementację wszystkich algorytmów omówionych w artykule
- wykonanie pomiarów wydajności
- analizę i wizualizację uzyskanych wyników
- przygotowanie większości treści manuskryptu (z wyłączeniem wstępu i opisu danych użytych do eksperymentów)

4.2 High-performance watershed delineation algorithm for GPU using CUDA and OpenMP

Artykuł zatytułowany „High-performance watershed delineation algorithm for GPU using CUDA and OpenMP" został poświęcony zagadnieniu wyznaczania zlewni. Skoncentrowano się w nim na wykorzystaniu możliwości jednostek GPU w celu poprawy wydajności tej operacji.

Punktem odniesienia dla przeprowadzonych badań były istniejące w literaturze algorytmy automatycznego wyznaczania zlewni. Na przestrzeni ostatnich kilkudziesięciu lat zaproponowano kilka fundamentalnie różnych podejść do tego samego zadania. Nowsze publikacje prezentują algorytmy zaprojektowane dla architektur GPU, opracowane z myślą o skróceniu czasu obliczeniowego potrzebnego na wykonanie tej operacji. Wydajność tych rozwiązań pozostawia jednak wiele do życzenia – w niektórych przypadkach implementacje wykorzystujące równoległość jednostek GPU okazywały się osiągać wyniki słabsze niż relatywnie proste algorytmy sekwencyjne.

Celem zrealizowanych prac było opracowanie algorytmu wyznaczania zlewni, oferującego wydajność wyższą niż inne, dostępne w literaturze rozwiązania. Z uwagi na dotychczasowy stan badań, szczególną uwagę poświęcono wykorzystaniu możliwości urządzeń GPU. Za bezpośredni punkt odniesienia do porównań przyjęto istniejące algorytmy opracowane dla tych architektur.

Określając problem algorytmiczny do rozwiązania w tej pracy przyjęto, że dane wejściowe będą składały się z rastra kierunku spływu wygenerowanego dowolną metodą single-flow oraz par współrzędnych określających położenie komórek ujściowych (wraz z przypisanymi do nich etykietami zlewni). Choć najczęściej spotykany w literaturze przypadek zakłada wyznaczanie pojedynczej zlewni i wybór tylko jednego punktu ujścia, w pracy dopuszczono możliwość wskazania wielu komórek ujściowych należących do tej samej lub różnych zlewni. Oczekiwanym wynikiem działania algorytmu był raster,

w którym każda komórka miała przypisaną etykietę zlewni, do której należała (lub wartość NONE, jeżeli znajdowała się poza obszarem zlewni branych pod uwagę).

Struktura opracowanego algorytmu została oparta na założeniu, że kluczowe operacje będą wykonywane przez kernele CUDA na jednostce GPU. Przyjęto, że każdej komórce rastra zostanie przypisany oddzielny wątek, który jako jedyny będzie mógł modyfikować jej zawartość. Instrukcje wykonywane po stronie hosta ograniczały się do odpowiedniego przygotowania danych wejściowych, przetransferowania ich do pamięci GPU, uruchomienia odpowiednich kerneli na urządzeniu i pobrania gotowych wyników.

Algorytm zaproponowany w ramach tej pracy ma charakter iteracyjny – zawartość komórek jest aktualizowana wielokrotnie, w każdym kroku wykorzystując stan rastra uzyskany w poprzedniej iteracji. Wartości indywidualnych komórek są ustalane i modyfikowane w sposób równoległy, efektywnie propagując wiedzę o przynależności poszczególnych lokalizacji do odpowiednich zlewni. Przygotowane zostały dwie implementacje, różniące się podejściem do synchronizacji pracy między wątkami. Pierwsza implementacja zakłada użycie dwóch buforów, z których jeden przechowuje stan rastra obliczony w poprzednim kroku (i jest wykorzystywany wyłącznie do odczytu), podczas gdy drugi stanowi miejsce zapisu zaktualizowanych wartości. Druga implementacja wykorzystuje pojedynczy bufor i operacje atomowe, dopuszczając odczyt i zapis komórek w ramach tej samej iteracji.

Do celów porównawczych zaimplementowano także dwa algorytmy wyznaczania zlewni dla GPU opisane w istniejącej literaturze. Według wiedzy autora, dotychczas opublikowano trzy algorytmy przeznaczone do realizacji tego zadania z wykorzystaniem architektur SIMD, jednak niezadowalająca wydajność jednego z nich została wykazana już w oryginalnej publikacji. Z tego względu uwagę poświęcono jedynie dwóm pozostałym.

Implementacje wszystkich algorytmów zostały przygotowane w języku C++ z wykorzystaniem standardu CUDA. Niektóre sekcje kodu wykonywane po stronie CPU zostały dodatkowo zrównoleglone z użyciem standardu OpenMP.

Do pomiarów wykorzystano trzydzieści zestawów danych o zróżnicowanych rozmiarach, obejmujących od 67,5 miliona do 2 miliardów komórek. Wszystkie zbiory zostały przygotowane w oparciu o ten sam obszar źródłowy, ale przedstawiony w różnych skalach. Takie podejście zostało wybrane w celu ujednolicenia wewnętrznej charakterystyki wszystkich zbiorów testowych. Eksperymenty zostały przeprowadzone na maszynie wyposażonej w dwa procesory Intel Xeon E5-2670 v3, 128 GB pamięci RAM oraz kartę GPU NVIDIA A100 Tensor Core.

Uzyskane wyniki pozwoliły jednoznacznie wykazać znaczącą przewagę opracowanego algorytmu nad istniejącymi alternatywami – czasy obliczeniowe potrzebne na wykonanie tych samych zadań różniły się średnio o dwa rzędy wielkości. Obie implementacje zaproponowanego rozwiązania osiągnęły względnie podobne rezultaty, wskazując na umiarkowaną

przewagę wersji opartej na pojedynczym buforze. Co ważne, wykorzystane algorytmy referencyjne dla GPU okazały się w rzeczywistości mniej wydajne niż relatywnie proste rozwiązanie sekwencyjne dla CPU, co podważa sens ich praktycznego stosowania. Ta obserwacja wydaje się pokrywać z wynikami publikowanymi wcześniej przez innych autorów.

Jedną z istotnych konsekwencji struktury zaproponowanego algorytmu jest możliwość jednoczesnego wyznaczania wielu zlewni w obrębie tego samego zbioru danych bez dodatkowego kosztu obliczeniowego. Z uwagi na równoległy, iteracyjny sposób organizacji pracy na jednostce GPU, czas potrzebny na wykonanie zadania nie jest zależny od liczby punktów ujściowych wskazanych przez użytkownika. Ta cecha nowego algorytmu stanowi istotną zaletę w kontekście typowych przypadków użycia, często wymagających wielokrotnego przeprowadzenia operacji wyznaczania zlewni dla różnych lokalizacji.

Implementacje opracowanych algorytmów zostały udostępnione w publicznym repozytorium pod adresem `https://github.com/bkotyra/watershed_delineation_gpu`.

Przedstawione prace badawcze zostały w całości zrealizowane przez autora niniejszej rozprawy. Tym samym publikacja ma charakter jednoautorski.

4.3 Fast parallel algorithms for finding the longest flow paths in flow direction grids

W artykule zatytułowanym „Fast parallel algorithms for finding the longest flow paths in flow direction grids" skoncentrowano się na zagadnieniu identyfikacji najdłuższych ścieżek spływu w zlewniach.

Pierwsze algorytmy przeznaczone do realizacji tego zadania zostały zaproponowane w literaturze już w latach 90. Nowsze prace podkreślają jednak problemy z wydajnością istniejących metod i wskazują na konieczność opracowania nowych rozwiązań. Według wiedzy autora, do momentu ukazania się tego artykułu istniejąca literatura nie proponowała algorytmów równoległych dla tego zagadnienia.

Najważniejszym celem prac było opracowanie nowego algorytmu, który pozwalałby w możliwie krótkim czasie identyfikować najdłuższe ścieżki spływu prowadzące do wybranych lokalizacji. Skoncentrowano się na wykorzystaniu możliwości przetwarzania równoległego na wielordzeniowych procesorach CPU. Dodatkowo zwrócono szczególną uwagę na problemy związane z precyzją uzyskiwanych wyników, występujące w istniejących algorytmach – jednym z założeń pracy było zaproponowanie rozwiązania, które pozwoliłoby ich uniknąć.

Powszechnie stosowane podejście do tego zagadnienia opiera się na wykorzystaniu

rastra kierunku spływu. W związku z tym, w ramach tej pracy również przyjęto, że dane wejściowe będzie stanowił raster kierunku spływu (wygenerowany dowolną metodą typu single-flow) wraz z lokalizacjami punktów końcowych poszukiwanych ścieżek. Dopuszczono możliwość wskazania więcej niż jednego punktu końcowego w tym samym zbiorze danych (najdłuższa ścieżka spływu powinna zostać znaleziona dla każdej z oznaczonych lokalizacji).

W ramach przeprowadzonych badań opracowano kilka odmiennych sposobów organizacji obliczeń. Ostatecznie zdecydowano się zaprezentować algorytmy oparte na dwóch odrębnych podejściach, oznaczonych w pracy jako „top-down" oraz „double drop". W artykule przedstawiono zarówno sekwencyjne, jak i równoległe implementacje tych koncepcji. Dodatkowo zaimplementowano rozwiązanie rekurencyjne, opisane w istniejącej literaturze, a także podjęto próbę poprawy jego wydajności z wykorzystaniem przetwarzania równoległego.

Podejście „top-down" zakładało organizację pracy w sposób pokrewny do idei opracowanej wcześniej dla zagadnienia akumulacji spływu. Algorytm rozpoczyna obliczenia w komórkach źródłowych, a następnie przemieszcza się w stronę coraz niższych lokalizacji, stopniowo gromadząc dane o istniejących ścieżkach spływu i propagując je do kolejnych elementów rastra. Przygotowano dwie sekwencyjne implementacje oparte na tym podejściu, z których jedna okazała się możliwa do efektywnego zrównoleglenia.

Koncepcja stojąca za podejściem „double drop" powstała z myślą o minimalizacji problemów związanych z próbami wykorzystania przetwarzania równoległego w innych algorytmach opracowanych dla tego zagadnienia. Nieintuicyjna organizacja pracy polega na rozpoczynaniu obliczeń od dowolnego punktu rastra, a następnie dwukrotnym trawersowaniu tej samej ścieżki spływu – za pierwszym razem mierząc odległość do punktu końcowego, a za drugim przypisując odpowiedni dystans każdej odwiedzanej komórce. Zakładano, że sekwencyjna wersja takiego rozwiązania prawdopodobnie okaże się niewydajna, jednak w wersji równoległej możliwe będzie uniknięcie czasochłonnej synchronizacji pracy między wątkami, co może pozwolić na znaczne skrócenie całkowitego czasu obliczeń.

Implementacje oparte na podejściu rekurencyjnym zostały potraktowane jako punkty odniesienia w ewaluacji nowych rozwiązań. Dodatkowo w porównaniach postanowiono uwzględnić narzędzie r.accumulate, dostępne na platformie GRASS GIS. Według wyników opublikowanych przez autora tego oprogramowania, dotychczas było to najbardziej wydajne z istniejących rozwiązań przeznaczonych do identyfikacji najdłuższych ścieżek spływu [9].

Implementacje algorytmów zostały przygotowane w języku C++. Sekcje kodu przeznaczone do wykonania w sposób równoległy zaimplementowano z użyciem standardu

OpenMP. Ze względu na problemy z precyzją wyników, rozpoznane w istniejących rozwiązaniach, we wszystkich implementacjach zastosowano podejście pozwalające wyeliminować akumulację błędów zaokrągleń.

Dane wykorzystane do eksperymentów obejmowały obszary zlewni o zróżnicowanych rozmiarach i charakterystykach, leżących na terenie Polski. Rastry zostały wygenerowane w kilku różnych rozdzielczościach, skutkując pokaźnym zestawem przypadków testowych. Największe wykorzystane zbiory danych zawierały w przybliżeniu 7 miliardów komórek.

Pomiary wydajności zostały przeprowadzone na dwóch maszynach o różnych specyfikacjach. Pierwszy komputer był wyposażony w dwa procesory Intel Xeon E5-2670 v3 oraz 128 GB pamięci RAM i działał pod kontrolą systemu AlmaLinux 8.4. Drugi posiadał dwa procesory Intel Xeon CPU E5-2620 v4 oraz 112 GB pamięci RAM, mając do dyspozycji dwa systemy operacyjne – Windows 10 Enterprise LTSC 64-bit oraz Ubuntu 22.04.1 LTS. Pomiary wydajności algorytmów zaimplementowanych w języku C++ zostały przeprowadzone na obu maszynach pod systemami z rodziny Linux. Eksperymenty dotyczące platformy GRASS GIS zostały zrealizowane zarówno pod Ubuntu jak i Windows, ze względu na zaobserwowane różnice w zachowaniu tego oprogramowania w zależności od systemu operacyjnego.

Uzyskane wyniki pozwoliły wykazać znaczną przewagę obu zaproponowanych algorytmów równoległych nad alternatywnymi rozwiązaniami. Biorąc pod uwagę średnie czasy wykonania, podejście „double drop" osiągnęło najlepsze rezultaty w zdecydowanej większości przypadków. Choć równoległa implementacja oparta na koncepcji „top-down" najczęściej uzyskiwała słabsze wyniki, przy odpowiednio wysokiej liczbie aktywnych wątków różnice między tymi algorytmami stawały się coraz mniej istotne. Porównanie opracowanych rozwiązań z narzędziem dostępnym na platformie GRASS GIS wykazało, że zaproponowane algorytmy pozwalają skrócić potrzebny czas obliczeniowy o rząd wielkości względem najbardziej wydajnej alternatywy, jaka była dotychczas dostępna.

Warto podkreślić, że zaproponowane algorytmy równoległe posiadają odmienne właściwości, które okazują się cenne w innych przypadkach użycia. Podejście „top-down" w naturalny sposób rozwiązuje wariant problemu, w którym konieczne jest znalezienie najdłuższych ścieżek spływu dla więcej niż jednego punktu końcowego. Z kolei podejście „double drop" pozwala na łatwe zidentyfikowanie alternatywnych ścieżek spływu o tej samej długości. W zależności od celu przeprowadzanej analizy, jedna z tych cech może mieć większą wartość dla użytkownika.

Kod źródłowy wszystkich opracowanych algorytmów (wraz z aplikacją umożliwiającą wykonywanie pomiarów) został udostępniony w publicznym repozytorium pod adresem `https://github.com/bkotyra/longest_flow_path`.

Wkład autora rozprawy w tę publikację został oszacowany na 97%. Zakres wykonanych prac obejmował:

- przegląd literatury
- zaprojektowanie i implementację wszystkich algorytmów omówionych w artykule
- wykonanie pomiarów wydajności (z wyłączeniem pomiarów dotyczących platformy GRASS GIS)
- analizę i wizualizację uzyskanych wyników
- przygotowanie większości treści manuskryptu

Rozdział 5

Podsumowanie

Prace zrealizowane i przedstawione w ramach niniejszej rozprawy koncentrowały się na wybranych, wzajemnie powiązanych tematach z obszaru modelowania hydrologicznego i systemów informacji geograficznej. Wspólnym problemem, rozpoznanym we wszystkich rozważanych zagadnieniach, była niska wydajność obliczeniowa istniejących algorytmów i narzędzi, dostrzegalna szczególnie w kontekście obszernych zbiorów danych geoprzestrzennych. Współczesna literatura podkreśla konieczność opracowania nowych, bardziej efektywnych rozwiązań. Przedstawione w rozprawie publikacje stanowią próbę odpowiedzi na to wyzwanie.

Wspólnym celem zaprezentowanych prac było opracowanie nowych algorytmów, które umożliwiłyby realizację wybranych zadań w sposób bardziej wydajny w stosunku do dostępnych dotychczas alternatyw. Szczególną uwagę poświęcono potencjałowi leżącemu w możliwościach przetwarzania równoległego, oferowanym przez współczesne jednostki obliczeniowe.

W ramach rozprawy przedstawiono trzy publikacje, z których każda odpowiada na jeden z wybranych problemów i szczegółowo omawia zaproponowane rozwiązania. Artykuły prezentują oryginalne, opracowane przez autora algorytmy równoległe, przeznaczone do obliczania akumulacji spływu powierzchniowego, wyznaczania zlewni i identyfikacji najdłuższych ścieżek spływu. W publikacjach zawarto wyniki pozwalające ocenić przydatność zaproponowanych algorytmów w kontekście innych, istniejących rozwiązań, przeznaczonych do realizacji tych samych zadań.

Głównym kryterium oceny nowych algorytmów, nadającym kierunek prowadzonym badaniom, był czas obliczeniowy wymagany do uzyskania prawidłowego wyniku. W trakcie prac zwrócono jednak uwagę także na inne aspekty, w których opracowywane rozwiązania mogłyby zaoferować przewagę w stosunku do istniejących alternatyw. W efekcie algorytmy przedstawione w publikacjach zostały zaprojektowane z myślą o wyeliminowaniu problemów charakterystycznych dla powszechnie stosowanych rozwiązań. Uwzględniono

również specyficzne przypadki użycia, które pomimo swojej praktyczności z reguły nie były rozważane w innych pracach.

Indywidualne cele, postawione w każdej z trzech publikacji, zostały pomyślnie zrealizowane. Zaproponowane algorytmy charakteryzują się wysoką wydajnością, stanowiąc odpowiedź na zidentyfikowane problemy związane z przetwarzaniem współczesnych, obszernych zbiorów danych geoprzestrzennych. Uzyskane wyniki pozwalają wykazać znaczącą przewagę opracowanych rozwiązań nad istniejącymi i stosowanymi obecnie alternatywami, wspierając tym samym tezę niniejszej rozprawy.

Obszar systemów informacji geograficznej wciąż jest aktywnie rozwijany. Opublikowane algorytmy mogą okazać się przydatne zarówno w opracowywaniu nowych, jak i rozbudowie istniejących pakietów oprogramowania. Skala różnic uzyskanych względem przyjętych punktów odniesienia wskazuje na potencjał wiążący się z praktycznym zastosowaniem zaproponowanych koncepcji.

Należy podkreślić, że przedstawione publikacje poruszają jedynie wybrane zagadnienia należące do obszaru, w którym podobnych problemów i wyzwań jest znacznie więcej. Duża część dostępnych w literaturze rozwiązań, które nadal są powszechnie stosowane, została opracowana w kontekście, który nie jest już dzisiaj aktualny. Dynamiczny rozwój technologii komputerowej, idący w parze z rosnącą dostępnością coraz większych zbiorów danych, wskazują na potrzebę kontynuacji prac w tym obszarze.

Na szczególną uwagę zasługują oryginalne metody proponowane w nowszych publikacjach z zakresu GIS, przy opracowywaniu których nie poświęcono wystarczającej uwagi kwestiom dotyczącym wydajności. Często prowadzi to do sytuacji, w których możliwość praktycznego zastosowania wartościowych koncepcji okazuje się ograniczona. Podjęcie prób rozwiązania tych problemów może stanowić obiecujący kierunek dalszych prac.

Bibliografia

[1] S. Akhter and J. Roberts. *Multi-core Programming: Increasing Performance Through Software Multi-threading.* Intel Press, 2006.

[2] Lawrence E. Band. Topographic partition of watersheds with digital elevation models. *Water Resources Research*, 22(1):15–24, 1986.

[3] G. Barlas. *Multicore and GPU Programming: An Integrated Approach, 2nd Edition.* Elsevier Science, 2022.

[4] Shashank Bhushan, David Shean, Oleg Alexandrov, and Scott Henderson. Automated digital elevation model (DEM) generation from very-high-resolution Planet SkySat triplet stereo and video imagery. *ISPRS Journal of Photogrammetry and Remote Sensing*, 173:151–165, 2021.

[5] André R. Brodtkorb, Trond R. Hagen, and Martin L. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[6] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP.* Morgan Kaufmann Publishers, San Francisco, 2001.

[7] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming.* The MIT Press, 2007.

[8] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming.* Wiley, 2014.

[9] Huidae Cho. A recursive algorithm for calculating the longest flow path and its iterative implementation. *Environmental Modelling & Software*, 131:104774, 2020.

[10] V.T. Chow, D.R. Maidment, and L.W. Mays. *Applied Hydrology.* McGraw-Hill, 1988.

[11] Isabel Kaufmann de Almeida, Aleska Kaufmann Almeida, Sandra Garcia Gabas, and Teodorico Alves Sobrinho. Performance of methods for estimating the time of concentration in a watershed of a tropical region. *Hydrological Sciences Journal*, 62(14):2406–2414, 2017.

[12] John Fairfield and Pierre Leymarie. Drainage networks from grid digital elevation models. *Water Resources Research*, 27(5):709–717, 1991.

[13] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[14] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[15] T.Graham Freeman. Calculating catchment area with divergent flow based on a regular grid. *Computers & Geosciences*, 17(3):413–422, 1991.

[16] Jurgen Garbrecht and Lawrence W Martz. The assignment of drainage direction over flat surfaces in raster digital elevation models. *Journal of Hydrology*, 193(1):204–213, 1997.

[17] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5), 2005.

[18] Scott Haag, Daniel Schwartz, Bahareh Shakibajahromi, Michael Campagna, and Ali Shokoufandeh. A fast algorithm to delineate watershed boundaries for simple geometries. *Environmental Modelling & Software*, 134:104842, 2020.

[19] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 5th Edition*. Morgan Kaufmann Publishers Inc., 2011.

[20] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. Optimization techniques for GPU programming. *ACM Comput. Surv.*, 55(11), mar 2023.

[21] W.W. Hwu, D.B. Kirk, and I.E. Hajj. *Programming Massively Parallel Processors: A Hands-on Approach, 4th Edition*. Morgan Kaufmann, 2023.

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

[23] Jing Li and David W.S. Wong. Effects of DEM sources on hydrologic applications. *Computers, Environment and Urban Systems*, 34(3):251–261, 2010.

[24] Chang Liao, Tian Zhou, Donghui Xu, Richard Barnes, Gautam Bisht, Hong-Yi Li, Zeli Tan, Teklu Tesfa, Zhuoran Duan, Darren Engwirda, and L. Ruby Leung. Advances in hexagon mesh-based flow direction modeling. *Advances in Water Resources*, 160:104099, 2022.

[25] John B. Lindsay, James J. Rothwell, and Helen Davies. Mapping outlet points used for watershed delineation onto DEM-derived stream networks. *Water Resources Research*, 44(8), 2008.

[26] R.K. Linsley, M.A. Kohler, and J.L.H. Paulhus. *Hydrology for Engineers*. McGraw-Hill, 1982.

[27] Ralf Ludwig and Philipp Schneider. Validation of digital elevation models from SRTM X-SAR for applications in hydrologic modeling. *ISPRS Journal of Photogrammetry and Remote Sensing*, 60(5):339–358, 2006.

[28] M. López-Vicente, C. Pérez-Bielsa, T. López-Montero, L.J. Lambán, and A. Navas. Runoff simulation with eight different flow accumulation algorithms: Recommendations using a spatially distributed and open-source model. *Environmental Modelling & Software*, 62:11–21, 2014.

[29] Danny Marks, Jeff Dozier, and James Frew. Automated basin delineation from digital elevation data. *Geo-Processing*, 2:299–311, 10 1984.

[30] Lawrence W. Martz and Jurgen Garbrecht. Automated extraction of drainage network and watershed data from digital elevation models. *Journal of the American Water Resources Association*, 29(6):901–908, 1993.

[31] Wen mei W. Hwu, David B. Kirk, and Izzat El Hajj, editors. *Programming Massively Parallel Processors, 4th Edition*. Morgan Kaufmann, 2023.

[32] Eleni Maria Michailidi, Sylvia Antoniadi, Antonis Koukouvinos, Baldassare Bacchi, and Andreas Efstratiadis. Timing the time of concentration: shedding light on a paradox. *Hydrological Sciences Journal*, 63(5):721–740, 2018.

[33] A. Musy, B. Hingray, and C. Picouet. *Hydrology: A Science for Engineers*. Taylor & Francis, 2014.

[34] John F. O'Callaghan and David M. Mark. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics, and Image Processing*, 28(3):323–344, 1984.

[35] Chukwuma J. Okolie and Julian L. Smit. A systematic review and meta-analysis of digital elevation model (DEM) fusion: pre-processing, methods and applications. *ISPRS Journal of Photogrammetry and Remote Sensing*, 188:1–29, 2022.

[36] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. ISSN. Elsevier Science, 2020.

[37] Adriano Rolim da Paz, Walter Collischonn, Alfonso Risso, and Carlos André Bulhíes Mendes. Errors in river lengths derived from raster digital elevation models. *Computers & Geosciences*, 34(11):1584–1596, nov 2008.

[38] Thomas Rauber and Gudula Rünger. *Parallel Programming: for Multicore and Cluster Systems*. Springer Berlin Heidelberg, 2013.

[39] R. Robey and Y. Zamora. *Parallel and High Performance Computing*. Manning, 2021.

[40] Holger Schäuble, Oswald Marinoni, and Matthias Hinderer. A GIS-based method to calculate flow accumulation by considering dams and their specific operation time. *Computers & Geosciences*, 34(6):635–646, 2008.

[41] Jan Seibert and Brian L. McGlynn. A new triangular multiple flow direction algorithm for computing upslope areas from gridded digital elevation models. *Water Resources Research*, 43(4), 2007.

[42] V. P. Singh and M. Fiorentino. *Geographical Information Systems in Hydrology*. Springer Netherlands, 1996.

[43] Vijay Singh. *Handbook of Applied Hydrology, 2nd Edition*. McGraw-Hill Education, New York, N.Y, 2016.

[44] Vijay Singh. Hydrologic modeling: progress and future directions. *Geoscience Letters*, 5, 12 2018.

[45] Peter N. Smith. Hydrologic data development system. Master's thesis, University of Texas, Austin, 1995.

[46] Peter N. Smith. Hydrologic data development system. *Transportation Research Record*, 1599(1):118–127, 1997.

[47] Duane Storti and Mete Yurtoglu. *CUDA for Engineers: An Introduction to High-Performance Parallel Computing*. Addison-Wesley Professional, 2015.

[48] Herb Sutter and James Larus. Software and the concurrency revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry. *Queue*, 3(7):54–62, 2005.

[49] Wenwu Tang and Shaowen Wang. *High Performance Computing for Geospatial Applications*. Springer International Publishing, 01 2020.

[50] David Tarboton, Dan Watson, Robert Wallace, K. Schreuders, and T. Tesfa. Hydrologic terrain processing using parallel computing. *Civil and Environmental Engineering Faculty Publications. Paper 2715*, page 0867, 11 2009.

[51] David G. Tarboton. A new method for the determination of flow directions and upslope areas in grid digital elevation models. *Water Resources Research*, 33(2):309–319, 1997.

[52] M. Uysal, A.S. Toprak, and N. Polat. DEM generation with UAV photogrammetry and accuracy analysis in Sahitler hill. *Measurement*, 73:539–543, 2015.

[53] L. Wang and H. Liu. An efficient method for identifying and filling surface depressions in digital elevation models for hydrologic analysis and modelling. *International Journal of Geographical Information Science*, 20(2):193–213, 2006.

[54] John Wilson, Graeme Aggett, Yongxin Deng, and Christine Lam. Water in the landscape: A review of contemporary flow routing algorithms. *Advances in Digital Terrain Analysis*, pages 213–236, 01 2008.

[55] H. Zhang, C. T. Haan, and David L. Nofziger. Hydrologic modeling with GIS: An overview. *Applied Engineering in Agriculture*, 6:453–458, 1990.

Dodatek A

High-performance parallel implementations of flow accumulation algorithms for multicore architectures

Computers & Geosciences 151 (2021) 104741

Contents lists available at ScienceDirect

Computers and Geosciences

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

Research paper

High-performance parallel implementations of flow accumulation algorithms for multicore architectures

Bartłomiej Kotyra [a,*], Łukasz Chabudziński [b], Przemysław Stpiczyński [a]

[a] Maria Curie-Skłodowska University, Institute of Computer Science, ul. Akademicka 9, 20-031 Lublin, Poland
[b] Maria Curie-Skłodowska University, Institute of Earth and Environmental Sciences, al. Kraśnicka 2d, 20-718 Lublin, Poland

ARTICLE INFO

Keywords:
Flow accumulation
Parallel algorithms
OpenMP
Multicore processors
Manycore architectures
GIS

ABSTRACT

The calculation of flow accumulation is one of the tasks in digital terrain analysis that is not easy to parallelize. The aim of this work was to develop new, faster ways to calculate flow accumulation and achieve shorter execution times than popular software tools for this purpose. We prepared six implementations of algorithms based on both top-down and bottom-up approaches and compared their performance using 118 different data sets (including 59 subcatchments and 59 full frames) of various sizes but the same area and resolution. Our results clearly show that the parallel top-down algorithm (without the use of OpenMP tasks) is the most suitable implementation for flow accumulation calculations of all we have tested. The mean and median execution times of this algorithm are the shortest in all cases studied. The implementation is characterized by high speedups. The execution times of the parallel top-down implementation are two orders of magnitude shorter compared to the Flow Accumulation tool from ArcGIS Desktop. This is important, considering the performance of popular GIS platforms, where it takes hours to perform the same kind of operations with the use of similar equipment.

1. Introduction

For the last few decades, the size of the GIS data to process have been constantly growing together with the complexity of computations needed to perform spatial analyses (Tang and Wang, 2020). On the other hand, parallel processing has become ubiquitous. The market offers multicore and many-core processors, graphics processing units, and big computer clusters. Parallel computers with multicore and many-core processors have become popular because they achieve high performance execution together with energy efficiency measured by performance-per-watt (Gruber and Keller, 2010; Patterson and Hennessy, 2013). Modern processors offer performance previously achieved only by supercomputers and large clusters. Moreover, modern computer architectures equipped with multicore processors offer large shared memory, which is usually sufficient for GIS computations. It should be noted that the use of OpenMP, a standard for programming shared-memory parallel computers (Chapman et al., 2007), allows to obtain good performance with relatively little effort. Moreover, OpenMP 4.5 offers more sophisticated programming techniques like thread affinity, tasking and supports SIMD extensions of modern multicore processors (van der Pas et al., 2017). However, implementing parallel algorithms that effectively use underlying hardware is still a challenging task.

Flow accumulation (also referred to as contributing area, upslope contributing area, or upslope area) is one of the parameters that are challenging for efficient calculation in digital terrain analysis. This issue is highly important in the context of hydrological applications where the extent of the water aggregation area (from rainfall, snowfall, etc.) is relevant. In the natural environment, water flow on the terrain surface is caused by gravity. It is partially modified by the properties of the layer through or over which water flows. Depending on the area, this layer may comprise vegetation, anthropogenic objects (e.g. buildings, squares, roads), and soil. The gravity process is modeled with the use of regular-grid digital elevation models (DEMs). Assuming that water flows between the DEM cells, it becomes obvious that the obtained flow accumulation values should be well correlated with the actual flow values observed in the catchment either during a single episode or throughout the year (Gruber and Peckham, 2009). In this approach, the computed flow accumulation value in a given cell is the sum of the weights of all cells from which water flows into that cell. Therefore, high flow accumulation values indicate areas of runoff accumulation such as rivers or local wetlands. Consequently, low or zero flow accumulation values indicate local elevations, e.g. peaks or watersheds (Wang and Jin, 2001). The calculation of flow accumulation values in the DEM involves three steps: (1) DEM preprocessing

* Corresponding author.
E-mail addresses: bartlomiej.kotyra@poczta.umcs.lublin.pl (B. Kotyra), lchabudzinski@poczta.umcs.lublin.pl (Łu. Chabudziński), przem@hektor.umcs.lublin.pl (P. Stpiczyński).

https://doi.org/10.1016/j.cageo.2021.104741
Received 27 July 2020; Received in revised form 18 February 2021; Accepted 25 February 2021
Available online 3 March 2021
0098-3004/

to remove depressions and flat areas commonly contained in DEMs, (2) computing flow-direction, (3) calculation of flow accumulation. Various methods and algorithms are used in each stage, which affect the calculation time and the obtained values (Zhou et al., 2016; Lindsay, 2016; Wang et al., 2019).

The choice of the flow direction algorithm is definitely important (López-Vicente et al., 2014), whereas the choice of algorithms for depression filling and handling flat areas do not affect flow accumulation calculations directly (Zhou et al., 2016; Lindsay, 2016; Wang et al., 2019).

The results of this very common procedure in hydrological studies depend on the calculation algorithm used and input data — flow direction (Knight et al., 2014). The flow accumulation algorithm was one of the first developed by O'Callaghan and Mark (1984) in a group that can be referred to as hydrological algorithms. The other algorithms in this group were closely related to the flow accumulation algorithm and comprised the method of depression filling pretreatment, flow direction determination, upslope area accumulation, drainage channel extraction, geographic feature vectorization and topographic parameter calculations. It should be noted that flow accumulation is an essential element for drainage network extraction (Martz and Garbrecht, 1992; Soille and Gratin, 1994; Garbrecht and Martz, 1997b; Tarboton, 1997; Jones, 2002; Bai et al., 2015) and is used in other aspects of environmental modeling (Rengers et al., 2016; Almeida et al., 2019; Tillery and Rengers, 2020).

Flow accumulation is one of the key input elements in simple flood risk assessment models used for calculation of the flood level and volume of runoff water from the catchment (Choi, 2012; López-Vicente et al., 2014; Huang, 2020). It is especially used in the case of ungauged catchments, which facilitates real time flood forecasting or rainfall-runoff simulation (Mwakalila, 2003). It plays a fundamental role in more advanced hydrological models, although it is only one of the input elements for advanced computational algorithms modeling the water flow on and below the ground (Heathwaite et al., 2005; Irving et al., 2018). Soil moisture patterns, zones of saturation and the generation of runoff from saturated areas can be indicated by flow accumulation (Beven and Kirkby, 1979; Burt and Butcher, 1985; Güntner et al., 2004). Apart from the typical hydrological applications, flow accumulation is also one of the main components of the models for delineated landscape units (Rathjens et al., 2016) and new aggregated indexes (López-Vicente and Ben-Salem, 2019).

Efficient implementations of the time-consuming hydrological algorithms become a challenge for modern GIS software packages. As reported by Gruber and Peckham (2009), the processing of high-quality elevation data sets that are currently becoming widely available requires new computational algorithms, as well as further development and refinement of existing methods. It should be emphasized that there are many algorithms for the determination of flow accumulation. The required resources and computation efficiency may vary depending on the input data characteristics and the structure of the algorithm. This is important in the process of problem solving, as the knowledge of how an algorithm works helps to use it in the most efficient way (Tarboton, 1997; Arnold, 2010).

The main motivation of this work was to address performance issues of existing algorithms, related to the growing size of available spatial datasets. According to our knowledge and experience, the existing flow accumulation algorithms are not efficient or scalable enough, which can be seen especially when working with larger data sets. High computational complexity along with inefficient or non-existent parallelism make the existing algorithms unsuitable for modern data. Our aim was to implement, test and present new, more efficient algorithms that could improve this situation. Our approach was to develop the most efficient implementations we could, based on a variety of concepts with an emphasis on parallelism. The developed algorithms were then tested on multiple datasets to assess their performance.

2. Flow accumulation algorithms

2.1. Flow direction

Most algorithms for calculating the flow accumulation use a flow direction matrix as input data. It is one of the basic DEM derivatives determining the direction of water flow. It allows to identify the basic elements characterizing the catchment (drainage network, watershed), which are used to model the water flow (O'Callaghan and Mark, 1984; Jenson and Domingue, 1988; Tarboton, 1997; Martz and Garbrecht, 1999; Turcotte et al., 2001; Choi, 2012). Therefore, flow direction calculation algorithms are widely implemented in software packages such as ArcGIS (and their extensions, e.g. ArcHydro Tools) (Moore et al., 1993), GRASS (Metz et al., 2011) and TecDEM (Shahzad and Gloaguen, 2011). They can be used for the calculation of flow accumulation, although there are algorithms that allow determination of flow accumulation directly from DEM (Arge et al., 2003; Bai et al., 2015). However, they are employed relatively rarely.

The depression-filling pretreatment of a DEM allows the water from every cell to flow to the catchment outlet. It is the first step of the flow direction procedure (Grimaldi et al., 2007). Therefore, when flow direction is used, the values of cells in depressions and flat areas should be transformed with special algorithms that allow modeling the flow in the entire catchment (Garbrecht and Martz, 1997a; Barnes et al., 2014). There are also flow-routing algorithms that do not need a depression-filling preprocessing of the DEM. These solutions can process massive DEMs rapidly (Magalhães et al., 2012), generate wide watercourses, calculate divergent flows, and detect fluvial landforms (Rueda et al., 2013). At the same time, they are less affected by the accuracy or data acquisition errors that DEMs include.

There are two fundamentally different approaches to determining the flow direction: the single flow and the multiple flow. In the single flow approach, the whole content of a cell is always transferred to only one of its neighboring cells. In the multiple flow approach, the content is transferred proportionally to every lower neighboring cell (Schäuble et al., 2008; Arge et al., 2003). Both methods are often analyzed and compared regarding their advantages and disadvantages (López-Vicente et al., 2014). Each approach has its specific applications (Schäuble et al., 2008). It is worth noting that access to both methods is important and desirable for GIS software users (Barták, 2009).

These fundamental differences between the single flow and multiple flow approaches can substantially affect the structure of the flow accumulation algorithm. In this work, we decided to focus on the single flow approach.

One of the simplest and most frequently implemented single direction methods is the D8 method (Garbrecht and Martz, 1997a; Tarboton, 1997). Its main assumption is that the flow direction of a cell can be determined with the use of local topographic slope values considering its eight adjacent cells.

It is necessary to note the importance of issues related to the cycles in flow direction matrices. There are cases where some implementations of flow direction algorithms can generate recurring flow paths. This kind of anomalies can have an important impact on subsequent flow accumulation calculations. While some implementations are resistant to flow direction cycles, others can generate incorrect results or end up in an infinite loop, making it impossible to finish the calculations properly.

According to our experience, the D8 algorithm implementation available in ArcGIS Desktop 10.6 can generate flow direction matrices with two neighboring cells pointing at each other.

2.2. Algorithms

In one of the earliest papers dedicated to the problem of drainage network extraction (O'Callaghan and Mark, 1984), the iterative flow accumulation method was proposed. The algorithm is based on the single flow approach. In a single iteration, the algorithm scans consecutive cells in a row-by-row manner. The current state of a given cell is used to update the flow accumulation values in the neighboring cells. These iterations are repeated as long as any values are changing. It is important to note that the time complexity of implementations based on this method is relatively high — the number of required iterations is dependent on the number of cells included in the longest flow path in the matrix.

Since then, a lot of research has been dedicated to the design of better performing algorithms and reducing the computation time with the use of parallel programming techniques. Some works criticize the iterative approach to the flow accumulation problem and suggest methods based on ordering cells using a priority queue or sorting (Jones, 2002; Wallis et al., 2009; Sten et al., 2016). The main idea here is to eliminate repeated calculations and multiple updates of the same cells. This kind of approach makes it possible to reduce the required computational effort substantially, but it is important to note that the cell ordering procedure also affects the overall execution time of these algorithms.

Multiple works consider the idea of recursive flow accumulation algorithms (Freeman, 1991; Schäuble et al., 2008; Choi, 2012). These methods usually start processing DEM from the topographically lowest cell (bottom-up approach). Determining the accumulation value of a given cell requires recursive executions of the procedure for its closest inflow neighbors. In this way, the algorithm first moves contrarily to the flow direction (from the lowest to the highest cells) and then goes back, updating flow accumulation values in the matrix. One of the important advantages of this approach is the fact that every accumulation value is calculated only once — the processing order is naturally dictated by subsequent recursive executions.

However, critical views on the recursive approach can often be found in literature, too. Some authors point out the high memory requirements and the issues related to the parallelization of this kind of algorithms (Wallis et al., 2009; Qin and Zhan, 2012). Some papers suggest solving these problems by replacing recursion with adequate data structures, e.g. a stack-based approach (Do et al., 2011). Other works consider the possibility of reversing the order of calculations by starting at the topographically highest cells and gradually processing data in accordance with the flow direction (top-down approach) (Arge et al., 2003; Schäuble et al., 2008; Zhou et al., 2019). Methods from this family vary considerably in implementation details and computational complexity.

A lot of research was dedicated to parallelizing flow accumulation algorithms in order to shorten their execution time. These works differ in the problems addressed as well as the utilized hardware architectures and programming techniques. Some papers were dedicated to solutions based on computing clusters. In this kind of approach, input data is usually split and distributed between separate computing units. In addition to the obvious advantage of a shorter execution time, it enables processing of data sets that are too large for a single machine. Wallis et al. (2009) and Do et al. (2011) studied specific flow accumulation algorithms along with the methods of their parallelization for computational clusters. Barnes (2017) focused on the methods of distributing tasks between nodes and gathering partial results together, without considering any specific flow accumulation algorithm details.

Attempts to develop flow accumulation algorithms dedicated for GPU were another important research direction. Published implementations differ in terms of calculation methods and programming techniques used to parallelize them. Ortega and Rueda (2010) presented two relatively simple algorithms for GPU, implemented using the CUDA architecture (Cheng and et al, 2014). Their approach was based on the original flow accumulation method from O'Callaghan and Mark (1984). Qin and Zhan (2012) focused on using CUDA to parallelize calculations based on the multiple-flow direction. Sten et al. (2016) compares four different algorithmic concepts implemented with OpenCL (Kowalik and Puzniakowski, 2012), trying to achieve the highest possible efficiency in the flow accumulation phase. Rueda et al. (2016) compared implementations of the original algorithm (O'Callaghan and Mark, 1984) based on OpenACC (Chandrasekaran and Juckeland, 2018) and CUDA, while focusing on differences in the execution time and code complexity.

There are also works focused on modifying the standard approach to calculation of flow accumulation. Du et al. (2017) proposed a method that uses a small set of low resolution data as a first step. Based on the results obtained, high resolution data is then divided into smaller parts and processed in parallel.

Certain new procedures have been proposed that are more efficient in modeling the terrain surface, taking into account e.g. closed depressions. For example, a noteworthy algorithm has been presented by Arnold (2010). It provides a method for dealing with depressions in the context of flow accumulation calculations. It assumes that water can fill and then flow through topographic depressions.

3. Experimental data

3.1. Study area

The area selected for the investigations covers a part of southeastern Poland that varies in terms of its altitude, morphometry, and hydrology. According to the physical–geographical division of Poland (Solon et al., 2018), it is situated in two macro-regions: Wyżyna Lubelska Upland and Roztocze. In terms of the hydrographic division, the area is the watershed of the upper Wieprz river characterized by tectonic mobility; hence, its subcatchments have very different morphometric characteristics (Brzezińska-Wójcik, 2013; Chabudziński and Brzezińska-Wójcik, 2013; Margielewski et al., 2017) (Fig. 1).

3.2. Source data

The input data are derived from the Central Office of Geodesy and Cartography (GUGiK) in the ASCII XYZ GRID format with a 1-meter resolution. Individual files correspond to the range of map sheets in the flat rectangular PL-1992 coordinate system at a scale of 1:5000.

The 974 sheets combined into one file in the TIFF format with a resolution of 1 × 1 m were selected for the investigations. The Mosaic to new raster tool from the ArcGIS Desktop 10.6 was used for merging the data. Next, 118 test areas were extracted. They represented both subcatchments (59 objects) with various surfaces and shapes and their extents (59 objects) limited to rectangular frames (Fig. 2). These areas were extracted into separate TIFF format files using the Extract by mask (subcatchments) and Clip (areas in the rectangular frames) tools from the ArcGIS Desktop 10.6. DEM preprocessing was carried out in each file to remove the depressions and flat areas commonly contained in DEMs; next, flow-direction was computed using ArcGIS Desktop 10.6.

Subcatchment files contained cells with values (elevation, filled elevation, flow direction) only where the catchment was located and NoData cells outside of the catchment, while files limited to the rectangular frames were fully filled with data (Fig. 2). The number of non-empty cells ranged from 46 171 348 to 2 489 411 697 for the subcatchment files and from 119 366 531 to 4 600 417 303 for the rectangular frame files. The main reasons for dividing the data into two groups: subcatchments and their limiting ranges was the willingness to compare the execution times of our algorithms on various datasets and indicate whether filling cells outside the catchment area with NoData values has any impact.

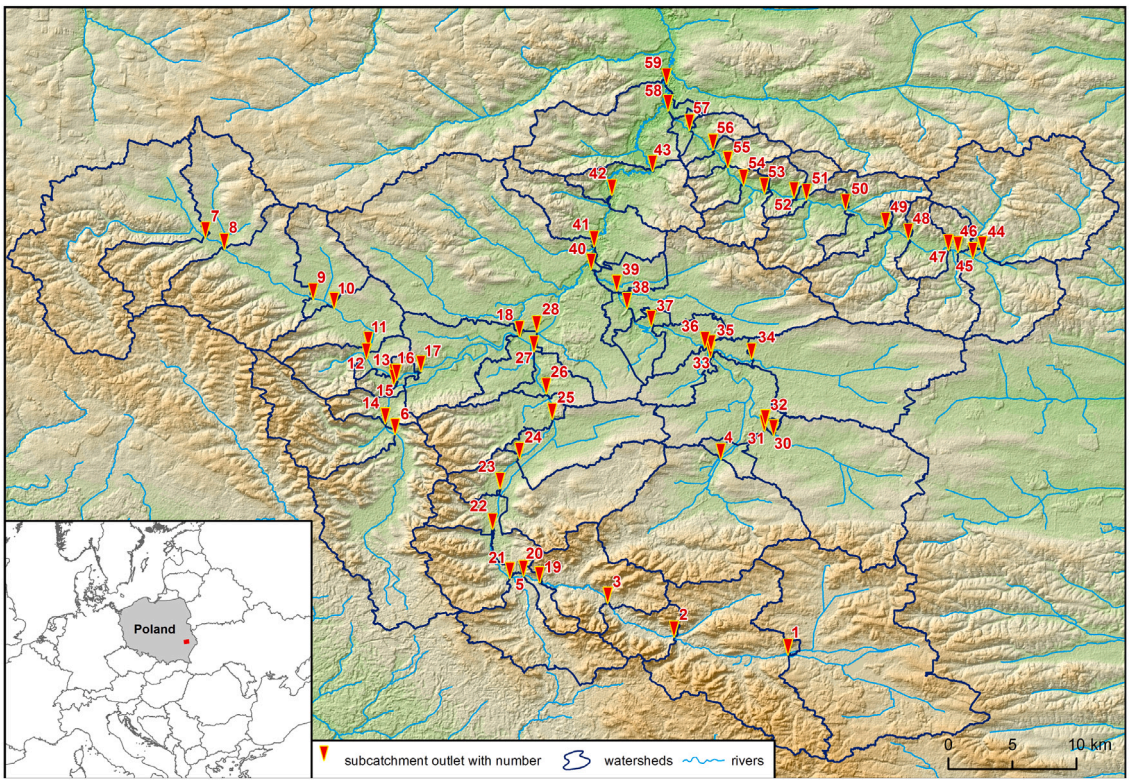


Fig. 1. Study area: subcatchments division of the upper Wieprz river watershed.

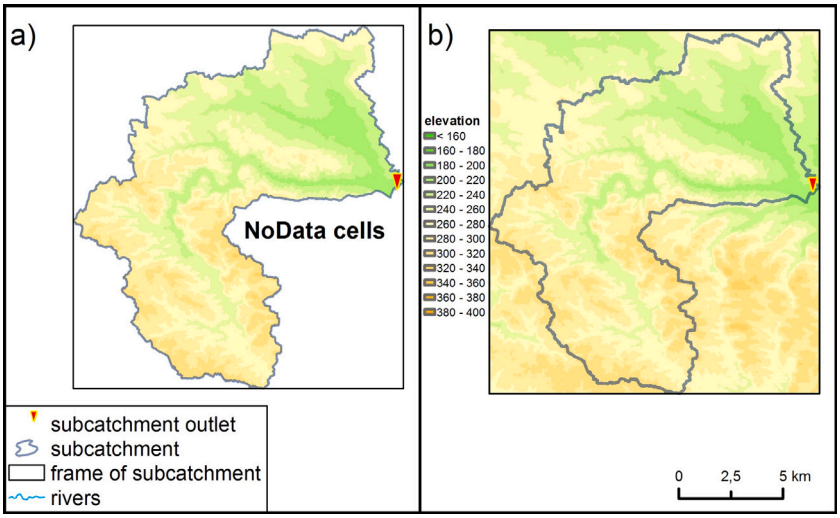


Fig. 2. Example of the data: subcatchment (a) and its frame (b).

4. Implemented algorithms

In this work, we focused on developing high-performance parallel flow accumulation algorithms for multicore architectures. We required that the basic versions of our methods should be characterized by linear computational complexity (thus, the flow accumulation value of each cell should be calculated and updated only once), to minimize the number of computational operations required. Moreover, to simplify the operations performed on each cell, we decided to introduce a frame of additional neutral cells around each matrix. For this reason, our algorithms process matrices using one-based indexing. We decided to consider two opposite approaches — the bottom-up method and the top-down method. This decision was based on the observation that these methods could be implemented with linear time complexity. First, we focused on the preparation of sequential algorithms that require as few computational operations as possible (deliberately avoiding queues or sorting). Next, we used these versions both as a basis for parallel versions and as benchmarks in performance measurements. We attempted to parallelize these algorithms using various methods, guided by our experience, understanding of the problem and the results of our early experiments.

All algorithms were implemented in C++ and parallelized using OpenMP. This standard introduces several directives that instruct the compiler that certain portions of the source code, i.e. loops and code sections, can be executed by threads working in parallel. Directives are also used to synchronize threads and specify data sharing. The

standard also defines functions and environment variables that affect the execution of parallel programs. Version 3.0 of the OpenMP standard introduced additional constructs for explicit support of task parallelism that can be used to parallelize recursive algorithms (van der Pas et al., 2017; Stpiczyński, 2018), thus it seems to be ideal for the simple and efficient implementation of the considered algorithms.

4.1. Bottom-up approach

The bottom-up approach starts calculations in the topographically lowest cells. Then, higher cells are processed step by step in the opposite order to the flow direction. Traditionally, algorithms in this category are implemented using recursion. By using the flow direction matrix to determine the order of cell processing, it is possible to achieve an implementation with linear time complexity, in which the value of each cell is calculated only once.

It should be noted that the bottom-up approach assumes no cycles in the flow direction matrix (Wallis et al., 2009). Such anomalies in the input data can result in a situation where the algorithm cannot finish the calculations, and subsequent recursive calls eventually lead to stack overflow.

As part of this work, we have prepared and tested three different implementations based on the bottom-up approach - a sequential version and two parallel ones.

4.2. Bottom-up - sequential implementation

The structure of the sequential version is based on a simple recursive function. In the first step, the algorithm prepares an inverse flow direction matrix, where the values indicate the neighbors from which water flows into the given cell. Then, the recursive function is started sequentially for all cells on DEM edges out of which water flows outside the processed area. Due to the possibility of using a DEM that is not hydrologically correct, a recursive function is also called for each DEM cell with no flow direction assigned.

The purpose of the recursive function is to calculate the flow accumulation value of a given cell and save the result in the output matrix. The obtained value is also returned as the result of the call. The function uses the inverse flow direction matrix prepared in the first step. The calculations are based on recursive function calls for the nearest neighboring cells from which water drains into a given cell. Returned values are added together. Then, the sum is increased by the number of these inflow neighbors, resulting in the final value of the cell's flow accumulation.

4.3. Bottom-up - parallel implementation

The first of the parallel implementations based on the bottom-up approach uses exactly the same recursive function as the sequential version. The difference lies in using OpenMP directives to start calculations from multiple cells in parallel. An independent parallel calculation procedure can be started in each cell from which water flows beyond the boundaries of the DEM. The same is true for cells without an assigned flow direction. The single-flow approach ensures that, for each cell, the recursive function will be called only once by a single thread. This guarantees that no data race will occur.

The first step of this algorithm is the parallel preparation of an inverse flow direction matrix. Cell values are calculated solely on the basis of the original flow direction matrix; therefore, parallelization of this stage is trivial. Next, threads begin an independent search for the starting points for a recursive procedure. Cells to be checked are allocated to threads using dynamic work distribution, which ensures that each cell is checked by only one thread. When a given thread finds a cell that directs flow outside the DEM boundaries or does not have a flow direction assigned, it calls the described recursive function for it. After returning from the function, the thread resumes searching for more starting points. The algorithm ends when all cells in the raster have been checked and all threads have completed their recursive calculations.

It should be noted that this approach assumes the existence of many starting points in the input data. The small number of such points limits the number of threads that can perform calculations simultaneously. Moreover, differences in catchment sizes can lead to unbalanced work allocation to threads. Thus, in this implementation, the possibility of using the advantages of parallelism is strongly dependent on the nature of the input data.

4.4. Bottom-up - task-based implementation

The second parallel implementation based on the bottom-up approach utilizes OpenMP tasks. This mechanism made it possible to use parallelism in a significantly different way than in the first implementation. Our goal here was to reduce the problems mentioned earlier, especially those related to the character of the input data.

This approach uses tasks to parallelize the recursive function. Subsequent calls are no longer made by the same thread but delegated in the form of tasks for later execution. The generated task can be performed by the same or a different thread. In this way, the execution of a recursive procedure started by a single thread can be parallelized by dividing it into tasks for different threads.

This method of calculation may prove valuable when the input data contains a small number of starting points for the algorithm. In this type of situation, although the calculations are initially started by only a small number of threads, the remaining threads receive their work allocation at higher levels of recursion nesting.

Our initial experiments have shown that creation of a large number of tasks can be associated with significant overhead of task creation and management. To reduce this problem, we decided to introduce a restriction that allows threads to create new tasks only to a certain level of recursion nesting. When the stack of dependent tasks reaches a certain maximum size, threads stop creating new tasks and perform all further calculations on their own. Owing to this approach, it was possible both to parallelize work effectively and to reduce the problems related to managing a large number of short tasks.

4.5. Top-down approach

The top-down approach starts the calculations from the locally highest cells (local peaks and ridges). Subsequent lower cells are processed in the order indicated by the flow direction. As in the bottom-up approach, the use of the flow direction matrix allows implementation with linear time complexity.

The change in the cell processing direction results in significant differences between the top-down and bottom-up approaches. First of all, the use of recursion is no longer a natural solution to the problem (it is no longer needed or useful). It is also important that the typical landscape of the catchment should contain many local peaks and ridges that can be used as starting points for the top-down calculation procedure. This creates new opportunities to introduce parallelism to algorithms.

We have prepared and tested three different implementations based on the top-down approach — the basic algorithm in the sequential version and its two parallel versions. The concept underlying the sequential version is similar to that presented in Zhou et al. (2019).

4.6. Top-down - sequential implementation

While working on the sequential version, we noticed that the ordering of all cells before starting the main calculations is essentially unnecessary and results in higher computational complexity and longer execution times. We decided to apply a simpler approach.

Listing 1: Bottom-up algorithm - recursive function

```
unsigned int updateValue(const FlowDirectionMatrix& reversalDirectionMatrix,
                         int row, int col, FlowAccumulationMatrix& accumulationMatrix)
{
  unsigned int sum = 0;

  if (reversalDirectionMatrix.value[row][col] & DIRECTION_RIGHT)
    sum += updateValue(reversalDirectionMatrix, row    , col + 1, accumulationMatrix) + 1;

  if (reversalDirectionMatrix.value[row][col] & DIRECTION_DOWN_RIGHT)
    sum += updateValue(reversalDirectionMatrix, row + 1, col + 1, accumulationMatrix) + 1;

  if (reversalDirectionMatrix.value[row][col] & DIRECTION_DOWN)
    sum += updateValue(reversalDirectionMatrix, row + 1, col    , accumulationMatrix) + 1;

  if (reversalDirectionMatrix.value[row][col] & DIRECTION_DOWN_LEFT)
    sum += updateValue(reversalDirectionMatrix, row + 1, col - 1, accumulationMatrix) + 1;

  if (reversalDirectionMatrix.value[row][col] & DIRECTION_LEFT)
    sum += updateValue(reversalDirectionMatrix, row    , col - 1, accumulationMatrix) + 1;

  if (reversalDirectionMatrix.value[row][col] & DIRECTION_UP_LEFT)
    sum += updateValue(reversalDirectionMatrix, row - 1, col - 1, accumulationMatrix) + 1;

  if (reversalDirectionMatrix.value[row][col] & DIRECTION_UP)
    sum += updateValue(reversalDirectionMatrix, row - 1, col    , accumulationMatrix) + 1;

  if (reversalDirectionMatrix.value[row][col] & DIRECTION_UP_RIGHT)
    sum += updateValue(reversalDirectionMatrix, row - 1, col + 1, accumulationMatrix) + 1;

  accumulationMatrix.value[row][col] = sum;

  return sum;
}
```

In the first step, using the flow direction raster, the algorithm prepares an inlet number matrix. Every value in this matrix corresponds to the number of inflows to a given cell (the number of adjacent cells from which water flows into the given cell). Then, the raster is linearly searched for cells where the number of inflows is zero, i.e. cells to which water does not flow from any of its neighbors. These cells are local peaks and ridges of the terrain and thus the starting points for the top-down calculation procedure.

When the cell forming the local peak or ridge is found, the algorithm temporarily stops the further screening of the raster and starts the main calculation procedure. The procedure is based on moving along the path determined by the flow direction. At each point, the algorithm calculates the total number of water units that have drained into the cell and updates the output flow accumulation matrix.

The inlet number matrix is used to recognize the junction cells (cells with more than one inflow). The values of such cells depend on several separate upstream channels, so their calculation requires prior processing of all cells in each of these channels.

While moving to subsequent cells, the calculation procedure decrements their values in the inlet number matrix. In this way, cells ready for further processing can be distinguished from those for which the necessary information has not yet been calculated. Only the cells whose inlet number is currently zero can be processed. When the procedure reaches the junction cell that is not yet ready to calculate its final flow accumulation value (its inlet number is greater than zero), the calculations for the current flow path are terminated and the algorithm returns to search for the next local peak or ridge.

This approach ensures that the flow accumulation value of each cell will be calculated and written to the output matrix only once. The correct order of cell processing is guaranteed despite the lack of an explicit cell ordering procedure.

4.7. Top-down - parallel implementation

The first parallel implementation of the top-down approach is based on a slightly modified sequential version. A few changes were necessary to facilitate seamless cooperation of multiple threads. OpenMP directives were used to parallelize the calculations.

In the first step, the algorithm prepares the inlet number matrix in parallel (the values of individual cells can be calculated independently of each other; therefore, the parallelization of this task is trivial). Next, threads begin the process of searching the raster for local peaks and ridges. When such a cell is identified, the thread stops searching and begins the calculation procedure, moving along the flow path and updating the values in the output flow accumulation matrix.

Therefore, the task of each thread is to search a part of the raster and perform the calculation procedure, starting with all the peak and ridge cells identified by it. Due to the varying lengths of flow paths, we decided to use dynamic work allocation to threads.

Many threads can simultaneously perform their own calculation procedures, processing subsequent cells along the flow direction. Therefore, value modifications related to junction cells (with more than one inflow) involve potential data races. Our implementation eliminates this problem by using two types of atomic operations. The first case is related to the updates of flow accumulation values in the output matrix, which are implemented with the use of the atomic update directive. The second case is related to decrementations of values in the inlet number matrix, where it is necessary not only to modify the cell value atomically, but also to capture the result of this operation immediately to avoid data races in the loop condition. For this reason, the atomic capture operation was used here.

This implementation also required the distinction of cells with no inflows (local peaks and ridges) from those whose number of inflows

was reduced to zero during processing. Therefore, these two states (peak/ridge cell, processed cell) are distinguished by two different negative values in the inlet number matrix. This eliminates a scenario where a thread recognizes a cell with zero inflows in place already processed by another thread.

4.8. Top-down - task-based implementation

The second parallel implementation based on the top-down approach uses OpenMP tasks. The idea of the algorithm is mostly the same as in the previous implementation. The difference lies in placing the main calculation procedure inside the task directive.

In this implementation, when the peak or ridge cell is recognized, the thread does not start the calculation procedure itself, but delegates its execution in the form of a task. This task can be performed later by any thread. The idea was to separate the raster search and the main calculation procedure from each other, which in some situations could lead to a better balance of workloads.

It is worth noting that the presented implementations based on the top-down approach are naturally resistant to the issue of flow direction cycles. Algorithms mark the processed cells with a certain value in order to distinguish cells with no inflows from those with the number of inflows reduced to zero during calculations. This mechanism also ensures that the algorithm will never fall into an infinite loop and the calculations will always be completed. Thus, the presented implementations do not require additional memory or instructions to address the issue of flow direction cycles.

5. Testing procedure

The testing procedure consisted of a series of steps, including preprocessing of input data, verification of output results, and repeated execution of a given algorithm with time measurements. The input for our algorithms was files containing flow direction rasters based on filled (hydrologically correct) DEMs. We made sure that the generated flow direction rasters did not contain any cycles. Anomalies of this kind were removed.

We agreed that one of the methods to verify the correctness of our implementations could be to compare their results with the files generated by the flow accumulation tool on the ArcGIS Desktop 10.6. We used this tool to prepare our control data. Next, each of our implementations was repeatedly run on different data sets, and the results obtained were compared with the control data.

Each test consisted of starting the application, loading the flow direction raster from the file, starting the time measurement, executing the given algorithm, and ending the time measurement. All additional calculations (including the preparation of an inverse flow direction matrix and an inlet number matrix) were treated as the internal part of the tested algorithm and were included in the measurements. An application reset in each test was necessary to eliminate the impact of cache warming.

Each of the six algorithms was tested on all prepared data sets (59 subcatchments and 59 rectangular frames). The task-based bottom-up implementation was tested for three different task creation limits (maximum recursion levels at which new tasks can be created): 250, 500, and 1000. Parallel algorithms were tested on three different thread affinity configurations (none, scatter, and compact) and four different threads per core settings (1, 2, 3, and 4). When measuring execution times, calculations were performed on all available cores. Each implementation was executed ten times for all data sets in each configuration.

6. Results of experiments

All experiments were performed on a computer with two Intel Xeon E5-2670 v3 processors (in total 24 cores, 2.3 GHz), and one Intel Xeon Phi Coprocessor 7120P (60 cores with multithreading, 1.238 GHz, 16 GB RAM) (Jeffers and Reinders, 2013). The computer equipped with 128 GB of RAM was running under Linux CentOS version 7.0. Our implementations of flow accumulation algorithms were compiled using the C++ Compiler which is a part of Intel Parallel Studio 2017. We used the `-O3` flag to optimize the code.

6.1. Performance comparison

The results generated by each of our implementations turned out to be identical to those obtained with the use of the ArcGIS Desktop 10.6. All output raster cells contained exactly the same values as the control data. This confirms that our algorithms are correct.

According to our measurements, the first parallel version of the top-down algorithm (without tasks) proved to be the fastest in all cases. For each dataset and in each multithreading configuration, the average and median execution time of this algorithm was the shortest among all tested implementations. In all cases (including the largest datasets and suboptimal configurations), it took less than 30 s to generate results (the longest recorded execution time was 26.2 s).

The task-based top-down implementation turned out to be less efficient than the version without tasks (on average, the processing time for the same dataset was 21.1% longer for subcatchments and 32.7% longer for rectangular frames). Our analysis showed that this approach leads to the creation of a huge number of tasks, many of which process only a few cells. It seems that the overhead of task creation and management has a significant impact on the performance of this algorithm.

Among the parallel implementations, the bottom-up approach proved to be much less efficient than the top-down approach. On average, the execution times of both parallel bottom-up implementations were more than three times longer than those achieved by the fastest top-down version (for both subcatchments and rectangular frames).

The size of the stack turned out to be an important issue in the parallel bottom-up implementations. The default size of 4 MB for each thread turned out to be insufficient for larger datasets. In some cases, recursive calls overflowed the stack, leading to program termination. To facilitate calculations for all data sets, it was necessary to increase the stack size of each thread to 16 MB using OMP_STACKSIZE. The other implementations did not cause such problems.

The comparison of the two bottom-up implementations (with and without tasks) leads to interesting observations. In most cases, both versions performed calculations at a similar time. However, for some input data sets and configurations, the task-based version turned out to be up to almost two times faster. In other cases, the average execution time was several dozen percent longer (up to 38.4% in one case). It seems that both input data characteristics and the multithreading configuration have a significant impact on the performance of the task-based version.

It is worth noting that the comparison of the execution times for the different multithreading configurations revealed certain differences between the top-down and bottom-up approaches. Parallel top-down implementations seem to have the shortest execution times for configurations with multiple threads per core. The results obtained for the parallel bottom-up implementations were significantly different. In most cases, they achieved the best results for configurations with a single thread per core.

The comparison of the sequential top-down and bottom-up versions leads to substantially different conclusions. The average execution times of both sequential implementations were relatively similar. For the vast majority of the subcatchments and all rectangular frames, the average and median bottom-up execution times were slightly shorter

Listing 2: Top-down algorithm - parallel implementation (excerpt)

```
#pragma omp parallel
{
  #pragma omp for schedule(dynamic)
  for (int row = 1; row <= matrixHeight; ++row)
  {
    for (int col = 1; col <= matrixWidth; ++col)
    {
      if ((inletNumberMatrix.value[row][col] == cellRidge) &&
          (directionMatrix.value[row][col] != DIRECTION_NONE))
      {
        int pathRow = row;
        int pathCol = col;
        char pathInletNumber;

        do
        {
          const unsigned int pathLoad = accumulationMatrix.value[pathRow][pathCol] + 1;
          inletNumberMatrix.value[pathRow][pathCol] = cellCompleted;

          switch (directionMatrix.value[pathRow][pathCol])
          {
            case DIRECTION_RIGHT:                ++pathCol; break;
            case DIRECTION_DOWN_RIGHT: ++pathRow; ++pathCol; break;
            case DIRECTION_DOWN:       ++pathRow;            break;
            case DIRECTION_DOWN_LEFT:  ++pathRow; --pathCol; break;
            case DIRECTION_LEFT:                 --pathCol; break;
            case DIRECTION_UP_LEFT:    --pathRow; --pathCol; break;
            case DIRECTION_UP:         --pathRow;            break;
            case DIRECTION_UP_RIGHT:   --pathRow; ++pathCol; break;
          }

          #pragma omp atomic update
          accumulationMatrix.value[pathRow][pathCol] += pathLoad;

          #pragma omp atomic capture
          pathInletNumber = --inletNumberMatrix.value[pathRow][pathCol];
        }
        while ((pathInletNumber == 0) &&
               (directionMatrix.value[pathRow][pathCol] != DIRECTION_NONE));
      }
    }
  }
}
```

Table 1
Average execution times (in milliseconds) for chosen subcatchments (best results selected across all multithreading configurations).

Dataset	Cells (non-empty)	Top-down sequential	Top-down parallel	Top-down tasks	Bottom-up sequential	Bottom-up parallel	Bottom-up tasks (250)	Bottom-up tasks (500)	Bottom-up tasks (1000)
0	169 586 945	8227	1331	1635	8228	4807	2971	2970	2973
8	276 862 675	12 468	1909	2326	11 944	7440	7444	7435	7421
12	368 445 621	17 778	2806	3326	17 628	10 065	10 215	10 118	10 163
22	460 794 411	25 854	3972	4683	24 858	12 506	12 320	12 311	12 362
24	575 105 179	30 737	4926	5893	31 092	15 758	15 826	15 820	15 855
26	630 348 921	34 672	5592	6709	35 231	17 450	17 162	17 281	17 312
42	2 049 527 215	101 788	14 708	18 563	99 790	55 333	55 086	55 714	55 594
58	2 489 411 697	122 402	17 570	22 179	119 057	67 421	66 850	66 992	66 817

than those achieved by the top-down implementation. Still, the average speedup of the parallel top-down versions was much higher compared to the bottom-up implementations (Table 3). Thus, after parallelization, the top-down algorithms achieved much more attractive execution times.

Tables 1 and 2 present the average execution times for selected datasets. Detailed results of all measurements can be found in our public repository.

6.2. Scalability analysis

Based on the results obtained, we considered the top-down algorithm in the parallel version (without tasks) to be the most suitable implementation for flow accumulation calculations. We performed additional tests to further examine the properties of this algorithm. In particular, we focused on analyzing the strong scalability of this implementation. We conducted two studies of the relationship between the number of computing cores used and the average execution time.

Listing 3: Top-down algorithm - task-based implementation (excerpt)

```
#pragma omp for schedule(dynamic)
for (int row = 1; row <= matrixHeight; ++row)
{
  for (int col = 1; col <= matrixWidth; ++col)
  {
    if ((inletNumberMatrix.value[row][col] == cellRidge) &&
        (directionMatrix.value[row][col] != DIRECTION_NONE))
    {
      #pragma omp task
      {
        int pathRow = row;
        int pathCol = col;
        char pathInletNumber;

        do
        {
          // the same as in the version without tasks
        }
        while ((pathInletNumber == 0) &&
               (directionMatrix.value[pathRow][pathCol] != DIRECTION_NONE));
      }
    }
  }
}
```

Table 2
Average execution times (in milliseconds) for chosen frames (best results selected across all multithreading configurations).

Dataset	Cells (non-empty)	Top-down sequential	Top-down parallel	Top-down tasks	Bottom-up sequential	Bottom-up parallel	Bottom-up tasks (250)	Bottom-up tasks (500)	Bottom-up tasks (1000)
0	328 524 987	12 427	1652	2190	11 228	5025	5038	5121	5227
8	456 220 275	17 384	2279	2978	15 386	7403	7454	7496	7644
12	697 485 292	26 978	3485	4558	24 024	10 215	10 167	10 350	10 457
22	999 276 228	42 903	5081	6665	35 943	12 563	12 460	12 456	12 587
24	1 290 109 002	51 153	6368	8536	45 877	16 288	16 316	16 077	16 443
26	1 492 618 476	59 392	7352	9885	53 238	17 372	17 470	17 394	17 306
42	3 879 244 800	153 488	18 453	25 226	138 017	56 686	56 899	56 132	56 851
58	4 600 417 303	181 914	21 956	29 634	162 793	67 224	66 674	66 800	66 827

Table 3
Average parallel algorithm speedups across all datasets and multithreading configurations (with all 24 cores used).

Algorithm	Average speedup (subcatchments)	Average speedup (frames)
Top-down parallel	6.07	7.33
Top down tasks	5.01	5.52
Bottom-up parallel	1.74	2.21
Bottom-up tasks (250)	1.65	1.95
Bottom-up tasks (500)	1.67	1.95
Bottom-up tasks (1000)	1.69	1.94

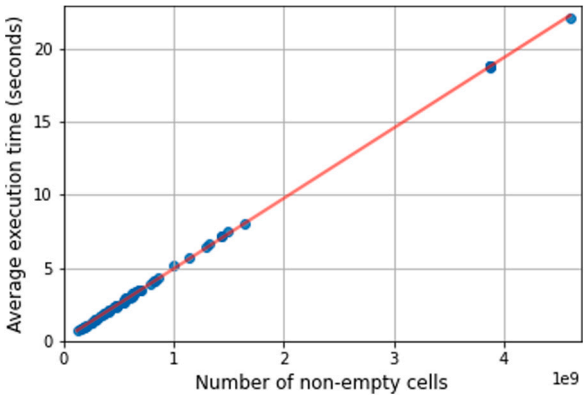


Fig. 3. Top-down (parallel) - time complexity.

For this purpose, we used both the host architecture and the MIC coprocessor.

Fig. 3 shows the relationship between the number of non-empty cells in a frame and the average algorithm execution time. As expected, the shape of the graph illustrates the linear time complexity of the algorithm.

On the host, the algorithm scalability was measured in the scatter configuration, using both sockets and assigning two threads per core. The algorithm performance was tested for twelve different settings (from 1 to 12 cores per socket). All tests were performed for frame 58, which was our largest dataset. Ten measurements of the execution time were made in each setting.

The MIC-based test was conducted in the compact configuration with four threads per core. The number of cores used was gradually increased from 1 to 60, allowing us to test the performance of this implementation for settings from 4 to 240 threads. Due to the smaller amount of RAM available on this architecture, the calculations were made for frame 17 (containing 1 643 792 696 cells). Similarly to the procedure described earlier, ten measurements of the execution time were made for each setting.

The results obtained show the high strong scalability of this implementation. Figs. 4 and 5 show a strong relationship between the number of cores used and the speedup compared to the sequential

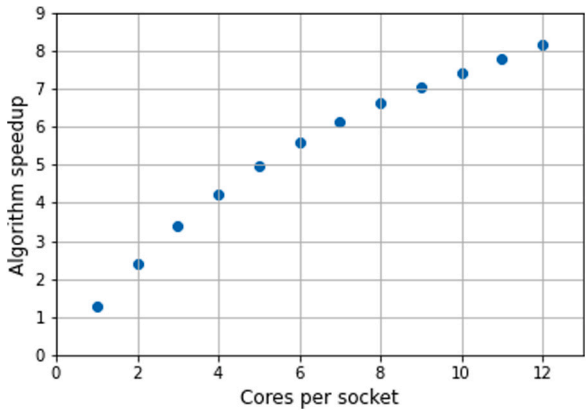


Fig. 4. Top-down (parallel) - speedup (host, scatter affinity, two sockets, two threads per core).

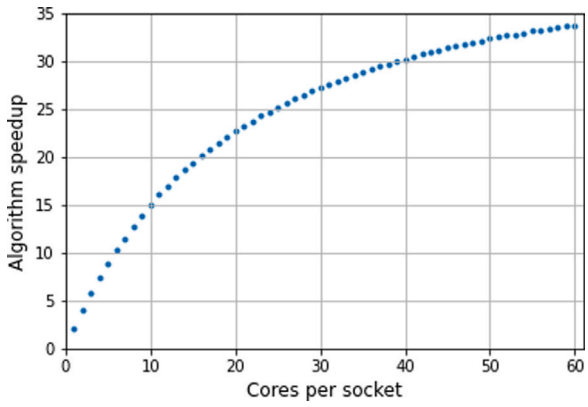


Fig. 5. Top-down (parallel) - speedup (MIC, compact affinity, single socket, four threads per core).

version. Increasing the number of computing cores up to 60 still reduces the average computation time.

7. Conclusions

In this paper, we have presented six implementations of flow accumulation algorithms, including two sequential and four parallel versions. We based our algorithms on both top-down and bottom-up approaches. We compared their performance using 118 different data sets (including 59 subcatchments and 59 full frames) of various sizes.

Our results clearly show that the parallel top-down algorithm (without the use of tasks) is the most suitable implementation for flow accumulation calculations of all we have tested. The mean and median execution times of this algorithm were the shortest in all cases studied (on all input data sets and in all multithreading configurations). The implementation is characterized by high speedups. It should also be noted that this approach does not require a large stack size for each thread, which proved necessary in the parallel bottom-up implementations.

A simple comparison with the Flow Accumulation tool from ArcGIS Desktop 10.6 showed that our parallel top-down implementation achieves execution times shorter by two orders of magnitude on a machine with the same specification. This is important considering the performance of popular GIS platforms, where it can easily take hours to perform the same kind of operations with the use of similar equipment.

It is worth noting that in the vast majority of cases the computation times for subcatchments were shorter than the corresponding times for full frames. Considering repeated tests performed for all the algorithms, data sets and configurations, the mean execution times for subcatchments were shorter in over 95% of cases. On average (across all the algorithms, data sets and configurations), the time needed to process the subcatchment was 81.72% of the time needed to process the corresponding frame (median 80.34%). However, this ratio seems to differ significantly from case to case, depending on the dataset and the algorithm used. It can be concluded that the use of cut-out subcatchments (filled with NoData values outside the catchment area) is an approach worth considering as it allows to reduce the necessary calculation time in most cases. Further analysis could discover how this advantage can be extended.

There are still many challenges related to GIS algorithms. Performance issues, especially in the context of large datasets, occur in many existing tools. In future work, we would like to focus on solving these problems.

Computer Code Availability

The source code of all implementations discussed is available in the public repository at https://github.com/bkotyra/high_performance_flow_accumulation/.

CRediT authorship contribution statement

Bartłomiej Kotyra: Conceptualization, Methodology, Software, Visualization, Writing - original draft. **Łukasz Chabudziński:** Data Curation, Validation, Visualization, Writing - review & editing. **Przemysław Stpiczyński:** Conceptualization, Supervision, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Almeida, R., Griebeler, N., Oliveira, M., Botelho, T., Moreira, A., 2019. Flow accumulation based method for the identification of erosion risk points in unpaved roads. Environ. Monit. Assess. 191. http://dx.doi.org/10.1007/s10661-019-7949-3.

Arge, L., Chase, J., Halpin, P., Toma, L., Vitter, J., Urban, D., Wickremesinghe, R., 2003. Efficient flow computation on massive grid terrain datasets. GeoInformatica 7, 283–313. http://dx.doi.org/10.1023/A:1025526421410.

Arnold, N., 2010. A new approach for dealing with depressions in digital elevation models when calculating flow accumulation values. Prog. Phys. Geogr. Earth Environ. 34, 781–809. http://dx.doi.org/10.1177/0309133310384542.

Bai, R., Li, T., Huang, Y., Li, J., Wang, G., 2015. An efficient and comprehensive method for drainage network extraction from DEM with billions of pixels using a size-balanced binary search tree. Geomorphology 238, 56–67. http://dx.doi.org/10.1016/j.geomorph.2015.02.028.

Barnes, R., 2017. Parallel non-divergent flow accumulation for trillion cell digital elevation models on desktops or clusters. Environ. Model. Softw. 92, 202–212. http://dx.doi.org/10.1016/j.envsoft.2017.02.022.

Barnes, R., Lehman, C., Mulla, D., 2014. An efficient assignment of drainage direction over flat surfaces in raster digital elevation models. Comput. Geosci. 62, 128–135. http://dx.doi.org/10.1016/j.cageo.2013.01.009.

Barták, V., 2009. How to extract river networks and catchment boundaries from DEM: a review of digital terrain analysis techniques. J. Land. Stud. 2, 57–68.

Beven, K.J., Kirkby, M.J., 1979. A physically based, variable contributing area model of basin hydrology. Hydrol. Sci. Bull. 24, 43–69. http://dx.doi.org/10.1080/02626667909491834.

Brzezińska-Wójcik, T., 2013. Morfotektonika Annopolsko-Lwowskiego Segmentu Pasa Wyzynnego W Świetle Analizy Cyfrowego Modelu WysokoŚciowego Oraz WskaŹnikÓw Morfometrycznych. UMCS, Lublin.

Burt, T.P., Butcher, D.P., 1985. Topographic controls of soil moisture distributions. J. Soil Sci. 36, 469–486. http://dx.doi.org/10.1111/j.1365-2389.1985.tb00351.x.

Chabudziński, L., Brzezińska-Wójcik, T., 2013. Zastosowanie ArcNEO do oceny przejawów neotektoniki na przykładzie zlewni górnego Wieprza (Roztocze, środkowowschodnia Polska). Land. Anal. 24, 11–22. http://dx.doi.org/10.12657/landfana.024.002.

Chandrasekaran, S., Juckeland, G. (Eds.), 2018. OpenACC for Programmers: Concepts and Strategies. Addison-Wesley.

Chapman, B., Jost, G., Pas, R.v.d., 2007. Using OpenMP: Portable Shared Memory Parallel Programming. the MIT Press.

Cheng, J., Grossman, M., McKercher, T. (Eds.), 2014. Professional CUDA C Programming. Wiley and Sons..

Choi, Y., 2012. A new algorithm to calculate weighted flow-accumulation from a DEM by considering surface and underground stormwater infrastructure. Environ. Model. Softw. 30, 81–91. http://dx.doi.org/10.1016/j.envsoft.2011.10.013.

Do, H.T., Limet, S., Melin, E., 2011. Parallel computing flow accumulation in large digital elevation models. Procedia Comput. Sci. 4, 2277–2286. http://dx.doi.org/10.1016/j.procs.2011.04.248, proceedings of the International Conference on Computational Science, ICCS 2011.

Du, C., Ye, A., Gan, Y., You, J., Duan, Q., Ma, F., Hou, J., 2017. Drainage network extraction from a high-resolution DEM using parallel programming in the .NET Framework. J. Hydrol. 555, 506–517. http://dx.doi.org/10.1016/j.jhydrol.2017.10.034.

Freeman, T., 1991. Calculating catchment area with divergent flow based on a regular grid. Comput. Geosci. 17, 413–422. http://dx.doi.org/10.1016/0098-3004(91)90048-I.

Garbrecht, J., Martz, L.W., 1997a. The assignment of drainage direction over flat surfaces in raster digital elevation models. J. Hydrol. 193, 204–213. http://dx.doi.org/10.1016/S0022-1694(96)03138-1.

Garbrecht, J., Martz, L.W., 1997b. Automated channel ordering and node indexing for raster channel networks. Comput. Geosci. 23, 961–966. http://dx.doi.org/10.1016/S0098-3004(97)00055-1.

Grimaldi, S., Nardi, F., Benedetto, F.D., Istanbulluoglu, E., Bras, R.L., 2007. A physically-based method for removing pits in digital elevation models. Adv. Water Resour. 30, 2151–2158. http://dx.doi.org/10.1016/j.advwatres.2006.11.016, recent Developments in Hydrologic Analysis.

Gruber, R., Keller, V., 2010. HPC@Green IT: Green High Performance Computing Methods, 1st Ed. Springer Publishing Company, Incorporated.

Gruber, S., Peckham, S., 2009. Land-surface parameters and objects in hydrology 33, pp. 171–194 http://dx.doi.org/10.1016/S0166-2481(08)00007-X.

Güntner, A., Seibert, J., Uhlenbrook, S., 2004. Modeling spatial patterns of saturated areas: an evaluation of different terrain indices. Water Resour. Res. 0, 40. http://dx.doi.org/10.1029/2003WR002864.

Heathwaite, A., Quinn, P., Hewett, C., 2005. Modelling and managing critical source areas of diffuse pollution from agricultural land using flow connectivity simulation. J. Hydrol. 304, 446–461. http://dx.doi.org/10.1016/j.jhydrol.2004.07.043.

Huang, P.C., 2020. Analysis of hydrograph shape affected by flow-direction assumptions in rainfall-runoff models. Water 0 (12), http://dx.doi.org/10.3390/w12020452.

Irving, K., Kuemmerlen, M., Kiesel, J., Kakouei, K., Domisch, S., Jähnig, S.C., 2018. A high-resolution streamflow and hydrological metrics dataset for ecological modeling using a regression model. Sci. Data 0 (5), http://dx.doi.org/10.1038/sdata.2018.224.

Jeffers, J., Reinders, J., 2013. Intel Xeon Phi Coprocessor High-Performance Programming. Morgan Kaufman, Waltham, MA, USA.

Jenson, S.K., Domingue, J.O., 1988. Extracting topographic structure from digital elevation data for geographic information system analysis. Photogramm. Eng. Remote Sens. 54, 1593–1600.

Jones, R., 2002. Algorithms for using a DEM for mapping catchment areas of stream sediment samples. Comput. Geosci. 28, 1051–1060. http://dx.doi.org/10.1016/S0098-3004(02)00022-5.

Knight, J., Rampi, L., Lenhart, C., 2014. Comparison of flow direction algorithms in the application of the CTI for mapping wetlands in Minnesota. Wetlands 0, 34. http://dx.doi.org/10.1007/s13157-014-0517-2.

Kowalik, J.S., Puzniakowski, T., 2012. Using opencl - programming massively parallel computers. In: Volume 21 of Advances in Parallel Computing. IOS Press, http://dx.doi.org/10.3233/978-1-61499-030-7-S1.

Lindsay, J.B., 2016. Efficient hybrid breaching-filling sink removal methods for flow path enforcement in digital elevation models. Hydrol. Process. 30, 846–857. http://dx.doi.org/10.1002/hyp.10648.

López-Vicente, M., Ben-Salem, N., 2019. Computing structural and functional flow and sediment connectivity with a new aggregated index: a case study in a large mediterranean catchment. Sci. Total Environ. 651, 179–191. http://dx.doi.org/10.1016/j.scitotenv.2018.09.170.

López-Vicente, M., Pérez-Bielsa, C., López-Montero, T., Lambán, L., Navas, A., 2014. Runoff simulation with eight different flow accumulation algorithms: recommendations using a spatially distributed and open-source model. Environ. Model. Softw. 62, 11–21. http://dx.doi.org/10.1016/j.envsoft.2014.08.025.

Magalhães, S.V.G., Andrade, M.V.A., Randolph Franklin, W., Pena, G.C., 2012. A New Method for Computing the Drainage Network Based on Raising the Level of an Ocean Surrounding the Terrain. Springer Berlin Heidelberg, pp. 391–407. http://dx.doi.org/10.1007/978-3-642-29063-3_21.

Margielewski, W., Jankowski, L., Krąpiec, M., Garecka, M., Hałas, S., Urban, J., 2017. Analysis of reworked sediments as a basis of the palaeogene-neogene palaeogeography reinterpretation: case study of the roztocze region (se poland). Sedim. Geol. 352, 14–29. http://dx.doi.org/10.1016/j.sedgeo.2017.02.009.

Martz, L., Garbrecht, J., 1992. Numerical definition of drainage network and subcatchment areas from digital elevation models. Comput. Geosci. 18, 747–761. http://dx.doi.org/10.1016/0098-3004(92)90007-E.

Martz, L.W., Garbrecht, J., 1999. An outlet breaching algorithm for the treatment of closed depressions in a raster DEM. Comput. Geosci. 25, 835–844. http://dx.doi.org/10.1016/S0098-3004(99)00018-7.

Metz, M., Mitasova, H., Harmon, R.S., 2011. Efficient extraction of drainage networks from massive, radar-based elevation models with least cost path search. Hydrol. Earth Syst. Sci. 15, 667–678. http://dx.doi.org/10.5194/hess-15-667-2011.

Moore, I., Turner, A., Wilson, J., Jenson, S., Band, L., 1993. Gis and land-surface-subsurface process modeling. Environ. Model. GIS 19, 6–230.

Mwakalila, S., 2003. Estimation of stream flows of ungauged catchments for river basin management. Phys. Chem. Earth, Parts A/B/C 28, 935–942. http://dx.doi.org/10.1016/j.pce.2003.08.039.

O'Callaghan, J.F., Mark, D.M., 1984. The extraction of drainage networks from digital elevation data. Comput. Vis. Graph. Image Process. 28, 323–344. http://dx.doi.org/10.1016/S0734-189X(84)80011-0.

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

van der Pas, R., Stotzer, E., Terboven, C., 2017. Using OpenMP – The Next Step. Affinity, Accelerators, Tasking, and SIMD. MIT Press, Cambridge MA.

Patterson, D.A., Hennessy, J.L., 2013. Computer Organization and Design, Fifth Edition: The Hardware/Software Interface, 5th ed. Morgan Kaufmann Publishers Inc.

Qin, C.Z., Zhan, L., 2012. Parallelizing flow-accumulation calculations on graphics processing units—from iterative DEM preprocessing algorithm to recursive multiple-flow-direction algorithm. Comput. Geosci. 43, 7–16. http://dx.doi.org/10.1016/j.cageo.2012.02.022.

Rathjens, H., Bieger, K., Chaubey, I., Arnold, J.G., Allen, P.M., Srinivasan, R., Bosch, D.D., Volk, M., 2016. Delineating floodplain and upland areas for hydrologic models: a comparison of methods. Hydrol. Process. 30, 4367–4383. http://dx.doi.org/10.1002/hyp.10918.

Rengers, F.K., McGuire, L.A., Coe, J.A., Kean, J.W., Baum, R.L., Staley, D.M., Godt, J.W., 2016. The influence of vegetation on debris-flow initiation during extreme rainfall in the northern colorado front range. Geology 44, 823–826. http://dx.doi.org/10.1130/G38096.1.

Rueda, A.J., Noguera, J.M., Luque, A., 2016. A comparison of native GPU computing versus OpenACC for implementing flow-routing algorithms in hydrological applications. Comput. Geosci. 87, 91–100. http://dx.doi.org/10.1016/j.cageo.2015.12.004.

Rueda, A., Noguera, J.M., Martínez-Cruz, C., 2013. A flooding algorithm for extracting drainage networks from unprocessed digital elevation models. Comput. Geosci. 59, 116–123. http://dx.doi.org/10.1016/j.cageo.2013.06.001.

Schäuble, H., Marinoni, O., Hinderer, M., 2008. A GIS-based method to calculate flow accumulation by considering dams and their specific operation time. Comput. Geosci. 34, 635–646. http://dx.doi.org/10.1016/j.cageo.2007.05.023.

Shahzad, F., Gloaguen, R., 2011. TecDEM: A MATLAB based toolbox for tectonic geomorphology, part 1: Drainage network preprocessing and stream profile analysis. Comput. Geosci. 37, 250–260. http://dx.doi.org/10.1016/j.cageo.2010.06.008.

Soille, P., Gratin, C., 1994. An efficient algorithm for drainage network extraction on DEMs. J. Vis. Commun. Image Represent. 5, 181–189. http://dx.doi.org/10.1006/jvci.1994.1017.

Solon, J., Borzyszkowski, J., Bidłasik, M., Richling, A., Badora, K., Balon, J., Brzezińska-Wójcik, T., Chabudziński, L., Dobrowolski, R., Grzegorczyk, I., Jodłowski, M., Kistowski, M., Kot, R., Krąż, P., Lechnio, J., Macias, A., Majchrowska, A., Malinowska, E., Migoń, P., Myga-Piątek, U., Nita, J., Papińska, E., Rodzik, J., Strzyż, M., Terpiłowski, S., Ziaja, W., 2018. Physico-geographical mesoregions of Poland: Verification and adjustment of boundaries on the basis of contemporary spatial data. Geogr. Polon. 0 (91), http://dx.doi.org/10.7163/GPol.0115.

Sten, J., Lilja, H., Hyväluoma, J., Aspnäs, M., 2016. Parallel flow accumulation algorithms for graphical processing units with application to RUSLE model. Comput. Geosci. 89, 88–95. http://dx.doi.org/10.1016/j.cageo.2016.01.006.

Stpiczyński, P., 2018. Language-based vectorization and parallelization using intrinsics, OpenMP, TBB and Cilk Plus. J. Supercomput. 74, 1461–1472. http://dx.doi.org/10.1007/s11227-017-2231-3.

Tang, W., Wang, S., 2020. High performance computing for geospatial applications. http://dx.doi.org/10.1007/978-3-030-47998-5.

Tarboton, D.G., 1997. A new method for the determination of flow directions and upslope areas in grid digital elevation models. Water Resour. Res. 33, 309–319. http://dx.doi.org/10.1029/96WR03137.

Tillery, A.C., Rengers, F.K., 2020. Controls on debris-flow initiation on burned and unburned hillslopes during an exceptional rainstorm in southern new mexico, usa. Earth Surf. Proces. Land. 45, 1051–1066. http://dx.doi.org/10.1002/esp.4761.

Turcotte, R., Fortin, J.P., Rousseau, A., Massicotte, S., Villeneuve, J.P., 2001. Determination of the drainage structure of a watershed using a digital elevation model and a digital river and lake network. J. Hydrol. 240, 225–242. http://dx.doi.org/10.1016/S0022-1694(00)00342-5.

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, pp. 467–472.

Wang, X., Jin, J., 2001. Assessing the impact of urban growth on flooding with an integrated curve number-flow accumulation approach. Water Int. 26, 215–222. http://dx.doi.org/10.1080/02508060108686907.

Wang, Y.J., Qin, C.Z., Zhu, A.X., 2019. Review on algorithms of dealing with depressions in grid DEM. Annals GIS 25, 83–97. http://dx.doi.org/10.1080/19475683.2019.1604571.

Zhou, G., Sun, Z., Fu, S., 2016. An efficient variant of the priority-flood algorithm for filling depressions in raster digital elevation models. Comput. Geosci. 90, 87–96. http://dx.doi.org/10.1016/j.cageo.2016.02.021.

Zhou, G., Wei, H., Fu, S., 2019. A fast and simple algorithm for calculating flow accumulation matrices from raster digital elevation. Front. Earth Sci. 13, 317–326. http://dx.doi.org/10.1007/s11707-018-0725-9.

Dodatek B

High-performance watershed delineation algorithm for GPU using CUDA and OpenMP

Environmental Modelling and Software 160 (2023) 105613

Contents lists available at ScienceDirect

Environmental Modelling and Software

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

High-performance watershed delineation algorithm for GPU using CUDA and OpenMP

Bartłomiej Kotyra

Maria Curie-Skłodowska University, Institute of Computer Science, ul. Akademicka 9, 20-033 Lublin, Poland

ARTICLE INFO

Keywords:
Watershed delineation
GIS
Parallel algorithms
GPU
CUDA
OpenMP

ABSTRACT

Watershed delineation is one of the fundamental tasks in hydrological studies. Tools for extracting watersheds from digital elevation models and flow direction rasters are commonly implemented in GIS software packages. However, the performance of available techniques and algorithms often turns out to be far from sufficient, especially when working with large datasets. While modern hardware offers high computing performance through massive parallelism, there is still a need for algorithms that can effectively use these capabilities. This paper proposes an algorithm for rapid watershed delineation directly from flow direction rasters, using the possibilities offered by modern GPU devices. Performance measurements show a significant reduction in execution time compared to other parallel solutions proposed for this task in the literature. Moreover, this implementation makes it possible to delineate multiple watersheds from the same dataset simultaneously, each having one or more outlet cells, with virtually no additional computational cost.

1. Introduction

Watershed (also referred to as drainage area, basin or catchment) is considered one of the basic concepts in hydrological studies (Tesfa et al., 2011). It is defined as the area of land whose drainage eventually concentrates in a single location, called the watershed outlet (Chow et al., 1988). Delineating watersheds and their boundaries is one of the fundamental tasks in this field and is used in many different contexts within and beyond the area of hydrology (Daniel, 2011; Singh, 2018).

While watersheds can be delineated manually using topographic data, automatic techniques are widely accepted as being much faster and more precise (Karimipour et al., 2013). Procedures using digital elevation models (DEMs) as basic input data are well known and widely implemented in GIS software (Barták, 2009). In recent decades, the availability and precision of this type of data has increased significantly, which considerably facilitates accurate calculations and simulations, but at the same time creates new challenges related to the processing of large datasets (Tang and Wang, 2020).

Currently, the market offers hardware architectures that enable achieving previously unavailable high computing performance. Wide access to multicore and many-core processors and graphics processing units (GPUs) creates new possibilities for parallel processing, making it possible to solve complex tasks on large datasets in a much shorter time. Parallel programming standards like OpenMP and CUDA allow access to these hardware capabilities with relatively little effort (Chapman et al., 2007; Cheng et al., 2014). Still, the effective use of the possibilities offered by modern devices requires creating suitable parallel algorithms, which remains a challenging task. There is a significant time gap between the available technological solutions and their practical applications (Tang and Wang, 2020). Existing software often turns out to be insufficiently scalable or simply unsuitable for working with modern datasets.

The motivation behind this research was to investigate and address issues related to the performance of existing watershed delineation algorithms. Concluding from the available literature, this problem has been discussed in multiple studies, but there is still room and need for significant improvement. The goal of this work was to develop and present a new raster-based algorithm for delineating watersheds (operating on DEMs and derivative data), allowing this task to be performed more efficiently compared to existing alternatives. Particular attention was paid to the possibilities offered by modern GPU devices, along with parallel processing capabilities on the host side.

1.1. Existing techniques

Many different methods and approaches for automatic watershed delineation have been described in the literature. They differ significantly in terms of technology and architecture applied, the type of input and data structures used, as well as the computational complexity of implemented algorithms.

Most of the existing research and practical implementations in this area use square-grid DEMs and their derivatives (such as flow direction

E-mail address: bartlomiej.kotyra@mail.umcs.pl.

https://doi.org/10.1016/j.envsoft.2022.105613
Received 4 October 2022; Received in revised form 18 December 2022; Accepted 20 December 2022
Available online 22 December 2022

rasters) as the basis for further operations. The foundations of this approach are well known (O'Callaghan and Mark, 1984; Jenson and Domingue, 1988). However, it is worth noting that this is not the only way to address the task. Some research works are focused on delineating watershed areas from triangle-based terrain models (Jones et al., 1990; Nelson et al., 1994; de Azeredo Freitas et al., 2016) or hexagonal grids (Liao et al., 2020). There are also alternative methods that may be useful when accurate DEMs are not available (Karimipour et al., 2013).

The most common, well-established workflow for delineating watersheds from square-grid DEMs consists of several separate processing stages (Baker et al., 2006; Eränen et al., 2014). Before a proper hydrological analysis can be carried out, the elevation data must be corrected by removing spurious sinks and local depressions. This is due to the fact that many existing models and algorithms require each DEM cell to have a downslope path leading to the edge of the raster (Wang and Liu, 2006). Depression filling is the most common method used to satisfy this condition (Tarboton et al., 2009). However, many recent papers suggest using other approaches, as they can produce the desired result with much less modification to the original elevation values (Lindsay, 2016; Chen et al., 2021).

Once the depressions are removed, the hydrologically corrected DEM can be used to calculate the flow direction data (Lindsay et al., 2008). At this stage, each cell is assigned a value corresponding to the expected direction of its further downstream flow. A variety of approaches and specific algorithms for determining flow directions exist in the literature (O'Callaghan and Mark, 1984; Fairfield and Leymarie, 1991; Freeman, 1991; Tarboton, 1997; Seibert and McGlynn, 2007). In essence, the purpose of this step is to determine how the outflow from each cell is distributed to its immediate neighbors (Wilson et al., 2008).

The next step is usually to calculate the flow accumulation data, where each cell is assigned the total number of cells that eventually flow to it (Jenson and Domingue, 1988; Martz and Garbrecht, 1993). These values can be used to delineate the drainage network, as well as to precisely locate the watershed outlet cells (Lindsay et al., 2008).

Once the flow direction is determined and the outlet cells are located, it is possible to perform the delineation of selected watersheds. In general, this stage is usually carried out using algorithms that identify all upslope cells connected to chosen outlet points by overland flow paths (Lindsay et al., 2008).

The remaining sections of this paper mainly deal with the last stage of this workflow, assuming that flow direction data and outlet cell locations are already available.

1.1.1. Recursive algorithms

Among the algorithms that utilize square-grid DEMs, one of the first and best known is the recursive approach. The earliest presentation of this concept that the author is aware of comes from Marks et al. (1984). In this approach, calculations start from a designated cell of the watershed outlet. The algorithm analyzes the immediate neighbors of this location, identifying the upstream cells (flowing to a given point) and classifying them as part of the watershed area. The same procedure is then performed recursively for each cell classified this way. The algorithm moves in a bottom-up manner, starting from the outlet point and following the flow paths upwards (opposite to the flow direction).

The main advantages of the recursive approach are its simplicity and relative efficiency. Only cells belonging to the watershed area and their direct neighborhood are analyzed (other cells are not entered through the procedure). However, the nature of recursive algorithms in general often leads to memory issues (especially on larger datasets) and proves difficult to effectively parallelize (Wallis et al., 2009; Qin and Zhan, 2012).

1.1.2. Iterative algorithms

There are multiple references to iterative implementations of watershed delineation algorithms in the literature. Unfortunately, not all papers discuss their approach in detail.

Jenson and Domingue (1988) described a procedure that uses a flow direction dataset and a "starter" raster where outlet points are marked with numerical values. The algorithm uses flow direction data to iteratively fill all cells with the "start" values of the outlet to which they flow.

Martz and Garbrecht (1993) mentioned a watershed boundary delineation procedure based on flow vectors, determined with the use of the D8 method. The algorithm identifies all cells that eventually flow into the user-specified outlet cell. The details of this stage are not extensively discussed, but it can be inferred from the context that the procedure is based on following the steepest descent path starting from each cell individually (this method was used there for accumulated drainage area calculations).

A similar concept was used for TIN data in Nelson et al. (1994). The algorithm starts at the centroid of each triangle and follows the flow path until it hits one of the terminus points.

Another approach is described in Choi and Engel (2003). Here, an iterative method using flow direction was used, which avoids scanning the entire raster. The procedure starts with a single outlet cell selected by the user. In each iteration, the algorithm considers only the closest neighborhood of cells classified as part of the watershed in the previous step. Using the flow direction data, successive upstream cells are identified. This procedure is repeated until no more matching cells can be found.

1.1.3. Sorting and priority queues

Another type of approach is based on sorting DEM cells and visiting them in a specific order. Arge et al. (2003) described an algorithm that processes cells in a bottom-up manner (reverse topological order), gradually propagating watershed labels from lower to higher cells. However, it should be noted that "computing watersheds" is understood here as part of a complex flooding procedure, rather than the actual delineation of watershed areas. Unique labels are assigned to each local sink and then propagated to the remaining cells. The idea is to identify areas concentrating their drainage in common sinks and to construct a graph of relationships between these areas. It is then used to raise the elevations to ensure that for each cell there is a non-ascending path to the edge of the terrain.

Barnes et al. (2014) integrated and improved on multiple related works by several authors and presented a unified algorithm based on a priority queue, primarily intended for filling depressions in DEMs. It was pointed out that this concept can be adapted and used for labeling watersheds as well. In this variant of the algorithm, unlabeled cells at the edges of the available data are assumed to be watershed outlets and are assigned unique labels. These values are then propagated to the remaining cells by "flooding" the DEM inwards.

It is worth noting that this group of algorithms processes cells in a relatively straightforward manner, but this comes at the expense of additional computational time needed to perform the ordering. It is also important to note that this type of approach is based on distinct assumptions, placing it outside the typical watershed delineation workflow.

1.1.4. Other data models

Yet another approach to the problem can be found in Haag et al. (2018, 2020). The algorithms presented here are intended to significantly reduce the computational time by marching around the watershed boundary, without entering or leaving its area. However, it is necessary to note that these techniques require converting the flow direction to other data models, specifically designed for this task, which entails additional computational and storage costs. Performing such an operation directly on the flow direction is not possible.

Castronova and Goodall (2014) highlighted the issues related to processing large datasets and demonstrated an alternative approach to watershed delineation. However, this technique relies on the availability of additional datasets, without which the DEM-based approach still appears to be a valid choice.

1.1.5. Parallel algorithms

Parallel algorithms are a separate category. In general, this approach aims to make better use of hardware capabilities and reduce computational time.

The earliest application of SIMD (Single Instruction, Multiple Data) computers to watershed delineation that the author is aware of can be found in Mower (1994). The data-parallel approach was used in two different iterative algorithms developed for Thinking Machines CM-5. Both ideas are based on copying watershed labels from neighboring cells. Conceptually, in the first algorithm, each thread "pushes" its own label to uphill cells, and in the second one, it "pulls" the label up from lower neighbors. In both cases, each iteration of the algorithm propagates the watershed label by only one cell along the flow path, but for multiple paths in parallel. According to the results presented, the label pulling approach turned out to be significantly more effective.

Some of the more recent works use the capabilities of SIMD architectures available on the graphics processing units. One of the steps described in Eränen et al. (2014) is the watershed delineation algorithm that performs all computations on the GPU. In this implementation, each thread starts at the assigned cell and follows the entire flow path until a defined stream section or DEM boundary is reached. Cells whose flow path ends in the defined stream section are marked as belonging to the watershed area. The work presented in Makinen et al. (2016) adapts a similar concept, but extends its use to a multi-GPU environment.

Another work addressing the problem of fast delineation of the watershed area was presented in Sit et al. (2019). Both sequential and parallel approaches were described here. The basic iterative sequential algorithm finds cells belonging to the watershed in a bottom-up manner, starting from the outlet. In subsequent iterations, the procedure analyzes the flow direction of the nearest neighborhood of cells classified in the previous step. The parallel approach takes advantage of the GPU capabilities using WebGL shaders. The algorithm runs for each cell independently. Its main idea is to use flow direction to identify the downstream neighbor and repeatedly read its label in each cycle. If the downstream cell is identified as belonging to the catchment area, the current cell is also marked as such. This procedure is repeated iteratively until there are no more updates.

1.2. Other related studies

Many published papers focus on developing efficient algorithms related to other aspects of hydrological modeling. Aside from delineating watershed areas, issues such as filling depressions in DEMs or calculating flow accumulation are often considered (Planchon and Darboux, 2002; Wang and Liu, 2006; Barnes et al., 2014; Zhou et al., 2016, 2019). Particularly noteworthy are works considering the parallelization of computations (Wallis et al., 2009; Do et al., 2011; Barnes, 2017; Zhu et al., 2019), including the use of graphics processing units (Ortega and Rueda, 2010; Qin and Zhan, 2012; Rueda et al., 2016; Sten et al., 2016; Wu et al., 2019).

1.2.1. SIMD pointer processing

Although this research focuses on hydrological modeling topics, some related work outside of this area should also be mentioned. In particular, a specific, decades-old SIMD technique for processing pointers is relevant to the rest of this work.

Hillis and Steele (1986) described a set of data-parallel algorithms, designed to be performed on machines with a large number of processors. One of the concepts presented here was dedicated to locating the last element in a linear linked list. It was pointed out that although this task appears to be inherently sequential, the work can be organized in a parallel manner, allowing it to be completed in less time. The core idea starts with expressing the order of elements as a series of pointers, each referring to the one immediately behind it. These are then repetitively reassigned by acquiring the addresses stored in the pointers to which they currently refer. This operation is performed for all elements in parallel, quickly leading to a state where all pointers refer to the last element of the list (except for the last one, which is marked with a special null value). Hillis and Steele (1986) described this technique as surprising and counterintuitive, but also pointed out that it had been discovered in other contexts before.

While coming from different fields, some of the problems considered in mathematical morphology, digital image processing and computer vision share certain similarities with GIS-related issues. Traces of the same SIMD technique can also be found in these areas.

One of the best known and most important problems in digital image processing is called segmentation (Beucher and Meyer, 1993). It can be broadly defined as the task of separating objects present in the image from their background (Roerdink and Meijster, 2000). Over the years, many techniques and approaches to this task have been developed, ranging from the simplest grayscale threshold methods to solutions based on neural networks and deep learning (Yuheng and Hao, 2017; Dmitruk et al., 2021; Minaee et al., 2022).

One of the classic approaches to image segmentation, first introduced in Digabel and Lantuéjoul (1978), is called the watershed transformation. Both the name and the intuitive idea come from the area of hydrology, metaphorically referring to a landscape being flooded by water or immersed in a lake (Roerdink and Meijster, 2000). The method operates on grayscale images, aiming at segmenting them into regions representing separate objects. The key idea is to interpret gray value discontinuity points as object boundaries. Despite significant advancements in the development of modern techniques, the watershed transformation can still be used for some specific issues (Kornilov et al., 2022).

Many noteworthy publications focus on developing efficient implementations of the watershed transformation, often taking advantage of parallel processing and GPU capabilities. The amount of research in this area is substantial and has been reviewed and summarized multiple times over the years (Roerdink and Meijster, 2000; Kornilov and Safonov, 2018; Kornilov et al., 2022). Some of these works implement variants of the pointer processing technique described previously. The key idea here is to treat the pixels of a two-dimensional image as pointers to one another and reduce the dependencies between them in a way similar to the one presented in Hillis and Steele (1986). Vitor et al. (2010) referred to this concept as path compressing and representative propagation, while Yeghiazaryan and Voiculescu (2018) called it path reduction and label propagation.

The only adaptation of this technique to a GIS-related context that the author was able to find comes from McGough et al. (2012). The paper focuses on applying parallel processing to the landscape evolution model in order to reduce the required computational time. One of the repetitively performed steps of the model is the sink filling procedure, which was implemented here using the general workflow presented in Arge et al. (2003) (described in earlier sections). The procedure consists of assigning unique identifiers to each local sink, propagating them to all connected cells, building a relationship graph between the identified areas and raising the elevation values so that each cell has a non-ascending path to the edge of the DEM. Similarly to Arge et al. (2003), watersheds are understood here as collections of cells flowing into common local sinks in a pre-filled DEM, rather than actual hydrologic units (the authors point out that thousands of watersheds can be identified during this procedure). The propagation of identifiers was implemented using a variant of the discussed SIMD technique. Here, two rasters are processed simultaneously, one with interrelated pointers and the other containing progressively propagated identifiers. The authors refer to this concept as index pointer jumping.

Since the flow direction rasters can be interpreted as matrices of pointers referring to their neighboring cells, the discussed technique can be seamlessly adapted to the watershed delineation workflow used in hydrological modeling.

2. Watershed delineation algorithms

2.1. Problem specification

Considering the numerous references to watershed delineation algorithms appearing in the literature, as well as many attempts to develop more efficient methods, it can be concluded that there is a need for a highly efficient algorithm allowing this operation to be performed on large datasets in a relatively short time.

When specifying the problem to be solved by the algorithm, it was assumed that the input data consists of a two-dimensional flow direction raster and a set of labeled outlet cell locations. The possibility of delineating multiple watersheds in a single algorithm run, as well as marking more than one cell as the outlet of the same watershed, was taken into account. Many techniques existing in the literature consider delineating only a single watershed at a time, and as noted in Haag et al. (2020), most of them use only a single outlet point, which is not always suitable. The author of this paper believes that it is worth extending the problem specification to include the possibility of using many such points belonging to the same or different watersheds. This will not only address specific practical use cases, but also allow for a more detailed analysis of the differences between algorithms, especially in the context of parallel data processing.

There are two main ways to define flow direction in the literature. In the single-flow approach, each raster cell points to only one downstream neighbor. Thus, all drainage is directed to a single neighboring cell. In the multiple-flow approach, drainage can be transferred proportionally to more than one neighbor. There are papers comparing the two approaches and pointing to the advantages and specific applications of both (Barták, 2009; López-Vicente et al., 2014).

The algorithms presented in this work use single-flow direction rasters as the main input data. As noted in Barták (2009), this approach seems more appropriate for watershed delineation as it avoids flow dispersion and catchment overlap. Specific flow direction algorithms are beyond the scope of this work and are not discussed here. In fact, any single-flow direction algorithm could be used to prepare input rasters (which is one of the main reasons why this form of data was chosen as the main input over raw DEMs). The only requirement is that the input dataset is hydrologically correct and does not contain recurring flow paths.

It is assumed that no additional input datasets (other than the flow direction raster and the set of outlet point markers) are needed. In this work, any possible transformation of this basic data or conversion to alternative models is considered as part of the watershed delineation procedure. Any processing of raw flow direction data is considered an internal part of the algorithm and taken into account when measuring its performance.

The output of the algorithm will be a raster of numerical values. In a correct solution, each cell will contain the index of the watershed it belongs to, or the NONE value if its flow path does not end at any of the specified outlet points or goes beyond the DEM boundary.

Since it should be possible to delineate multiple watersheds in a single algorithm run, a scenario of simultaneously delineating a larger watershed and its smaller sub-watersheds must be considered. In such cases, the cells in the output raster will be labeled as part of the innermost sub-watershed to which they belong.

It is assumed that the algorithm is executed on a single GPU-equipped machine, and the size of available memory is sufficient for the data to be processed.

2.2. Implementation details

All implementations developed as part of this work were written in C++11. The code dedicated to the GPU was implemented using CUDA. Additionally, some parts of the code executed on the host were parallelized using OpenMP for faster performance. Only the fundamental constructs introduced in the early versions of these two standards were used.

The implementations store the watershed labels in 8-bit unsigned char cells. This makes it possible to delineate up to 255 watersheds in a single algorithm run (one of the 256 possible values is reserved for the NONE value). In case more unique labels were needed, implementations could be easily extended by changing the variable type.

Wherever it was necessary to index raster cells with single numbers, unsigned 32-bit ints were used to store the indices. This imposes a technical constraint on these implementations, limiting the maximum number of cells to 2^{32}. Again, in case larger data sizes are needed, it is possible to increase this limit by changing the variable type.

2.3. Flow path reduction algorithm

The algorithm developed as a key part of this work is designed for GPU devices and takes advantage of the specific properties associated with their massive parallelism. In essence, each raster cell is processed in parallel by a separate thread.

The core idea behind the algorithm can be divided into three main stages. First, an array of cell indices is prepared using the flow direction data — initially, each index points to its nearest downstream neighbor. The flow paths are then reduced so that each cell ultimately contains the index of the outlet cell to which it eventually flows. Finally, these indices are converted to watershed labels.

2.3.1. Data preparation (host side)

Before executing the key parts of the algorithm, the host reshapes and adjusts the data to the required form.

In this work, the data structure developed in Kotyra et al. (2021) was used to store flow direction rasters on the host. In essence, the data is stored in a two-dimensional array with an additional frame of cells containing neutral values (a single row or column of neutral cells on each of the four edges of the raster). As a result, rows and columns containing relevant data are indexed from 1 (one-based indexing). This design was developed due to the large number of raster algorithms using the values of neighboring cells (the frame of neutral values often makes it possible to simplify the code and reduce execution time). While this feature has no significant application in this paper, this structure was chosen to maintain work continuity and consistency in the published code.

The first step in preparing the data for the GPU is to flatten a two-dimensional flow direction raster into a one-dimensional array. This approach aims to simplify and speed up the transfer and data processing on the device. Consecutive rows are rewritten without gaps, skipping the frame of neutral cells. While this operation is trivial, it can take a noticeable amount of time for large datasets. To accelerate this step, the code was parallelized using OpenMP clauses.

The next step is to remove the direction from all cells marked as outlets (setting their direction values to NONE). Due to the design of some parts of the algorithm, it is required to ensure that the outlet cells are the endpoints of the flow paths and do not point to subsequent cells. As the number of the outlet cells was assumed to be rather small (a single cell being the most common scenario), this operation did not seem worth parallelizing.

The prepared one-dimensional flow direction array is then transferred to the GPU, where it can be further processed.

2.3.2. Calculation of downstream cell indices

The first kernel to run on the device performs a relatively simple transformation of the flow direction data into cell indices. A new one-dimensional array is prepared, where each cell contains the index of its downstream neighbor (the cell immediately next in the flow path). For this purpose, the index of each cell is shifted by an appropriate offset according to the corresponding flow direction.

The outlet cells (marked with the NONE direction) receive their own index and thus point to themselves. The cells directing the flow outside the raster boundaries are treated in the same way. Consequently, the endpoint cell of each flow path is marked with its own index. The hydrological correctness of the input data is assumed, therefore all remaining cells should receive the index of one of their neighbors.

This stage requires a single kernel run. Each thread processes a single cell by reading its flow direction value, calculating the index, and storing it in a new array. As each cell can be processed independently, parallelizing this operation was straightforward.

2.3.3. Reduction of flow paths

When the indices of all downstream neighbors are calculated, the data is ready to be processed by the main loop of the algorithm. Conceptually, it is now possible to traverse a flow path from any cell to its endpoint by reading the indices of successive cells (each pointing to the next downstream neighbor), until reaching a cell that points to itself. The aim of this stage is to reduce these flow paths to just a single step. The index array is transformed so that each cell points to the final endpoint of its flow path. When this stage is done, each cell should either point to itself (being an endpoint) or directly to another cell pointing to itself. After this transformation, by reading the index stored in any cell, it will be possible to immediately identify the watershed outlet to which it flows.

The transformation of the array is done by executing the path reduction kernel multiple times (this is considered the main loop of the algorithm). In a single kernel run, the index of each cell is established by reaching its target cell, reading the index stored there and updating its own if necessary. Conceptually, the following procedure is performed on each cell:

- read your target index
- reach the target cell and read its index
- if the target cell does not point to itself, save its target index as your own

The kernel is iteratively executed as long as any changes to the array are made.

It is important to emphasize why this approach is highly efficient. The GPU architecture allows this operation to be performed in a massively parallel manner (each cell is processed in parallel by a separate thread). Conceptually, the moment a cell is updated with a new target index, that target cell is also updated in a similar way. Although the first iteration of the kernel shifts the target indices only a single step in the flow paths, each subsequent execution allows for a jump approximately twice as long, using the knowledge accumulated in the previous steps. As a result, the processing of even large datasets is possible in a relatively small number of iterations.

It is worth noting that the number of iterations needed is directly dependent on the longest flow path (as measured by the number of cells) present in the input data. It can be shown that the time complexity of this stage of the algorithm is sublinear, as a flow path containing twice as many cells requires just one additional iteration. Conceptually, the number of cells identified as flowing into a given endpoint grows exponentially with each step. It is also worth emphasizing that the number of outlet cells (belonging to the same or different watersheds) does not affect the execution time.

The first implementation of this idea (labeled “back buffer”) uses two data buffers. In each iteration, the current state of the cells is read from one buffer, while new values are written into the other. The roles of the buffers are swapped before the next step. This approach easily eliminates the scenario where some cells might be read by one thread and modified by another in a single iteration. This guarantees completely predictable results at every step. However, the obvious disadvantage of this implementation is the additional memory requirement, as data is continuously transferred between two buffers of the same size.

The second implementation uses a single buffer, allowing both reading and writing of the same cells within the same iteration. Although this makes the result of a single step less predictable (it is not obvious whether a given cell will be read before or after being modified by another thread), it does not have a negative impact on the final result of the entire procedure. In fact, this implementation allows not only lower memory consumption, but in some scenarios also fewer iterations required and consequently faster execution time. A similar approach was discussed in Golub and Ortega (1993), where bypassing synchronization in iterative methods was used to reduce the overhead. The authors referred to this solution as the asynchronous method.

In assessing this solution, some concern may be related to the atomicity of the cell modification. While it is not possible to predict whether a cell will be read before or after being modified by another thread, it is necessary to guarantee that the value will be correct in both cases (being either the value from the previous or the current step). The Device Memory Accesses section of the CUDA C++ Programming Guide (NVIDIA, 2022) states that an access to data in global memory is compiled into a single instruction under certain conditions, namely when the data size is 1, 2, 4, 8, or 16 bytes, and when the data is naturally aligned (meaning the memory address is a multiple of that size). If these conditions are not met, the data access can be compiled into multiple instructions, creating potential problems in a multithreaded context. As the presented implementation meets the above conditions, any additional synchronization of data access would be redundant.

2.3.4. Assignment of watershed labels

The last step performed on the GPU is converting the target indices to watershed labels. At this stage, each cell points to its flow path endpoint. Conceptually, this operation assigns each cell a label associated with the watershed outlet it points to (or NONE if its flow path does not end at any of the specified outlets).

In the simplest case, assuming a single watershed with only one outlet point, the operation would be to compare the target index of each cell with the outlet location. Cells pointing to it would be labeled as belonging to the watershed area, while the remaining ones as being located outside of it. However, it was decided that the algorithm should be able to delineate multiple watersheds at once, each with one or more outlet cells, so a more general solution had to be implemented.

First, the algorithm prepares an array where the outlet cells contain the labels of the corresponding watersheds, and all other cells are marked as NONE. In order to reduce the memory consumption on the GPU, the same array that originally stored the flow direction values is used. The array is cleared (filled with NONE values) in parallel by a simple kernel, then the watershed labels are copied into the outlet cells.

The conversion is performed in a single kernel execution (all cells are processed independently of each other in a parallel manner). For each cell, a thread reads its target index, and then assigns it a label located at that index in the label array. Consequently, cells with flow paths ending at specified outlet points receive valid labels. All other cells are marked with the NONE value. The result of this operation is saved in the label array. It is worth noting that the outlet cells point to themselves, so their labels essentially do not change.

Next, the array prepared in this way is transferred back to the host memory. It is then unpacked into a two-dimensional raster containing a

Listing 1: CUDA kernel used for determining downstream cell indices

```
__global__ void directionToTargetKernel(unsigned char* directionArray,
                                        unsigned int* targetArray,
                                        int height,
                                        int width)
{
    const unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < height * width)
    {
        int row = index / width;
        int col = index % width;

        switch (directionArray[index])
        {
            case DIRECTION_RIGHT:              ++col; break;
            case DIRECTION_DOWN_RIGHT: ++row; ++col; break;
            case DIRECTION_DOWN:       ++row;        break;
            case DIRECTION_DOWN_LEFT:  ++row; --col; break;
            case DIRECTION_LEFT:               --col; break;
            case DIRECTION_UP_LEFT:    --row; --col; break;
            case DIRECTION_UP:         --row;        break;
            case DIRECTION_UP_RIGHT:   --row; ++col; break;
        }

        targetArray[index] = ((row >= 0) && (row < height) &&
                              (col >= 0) && (col < width))
                              ? row * width + col : index;
    }
}
```

Listing 2: Main CUDA kernel used by the flow path reduction algorithm (single-buffer implementation)

```
__global__ void pathReductionKernel(unsigned int* targetArray,
                                    unsigned int size,
                                    bool* changes)
{
    const unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;

    if ((index < size) && (targetArray[index] != targetArray[targetArray[index]]))
    {
        targetArray[index] = targetArray[targetArray[index]];
        *changes = true;
    }
}
```

frame of neutral cells (similar to the structure used to store the original flow direction data). Early measurements showed that parallelizing this operation can noticeably reduce its execution time, therefore OpenMP clauses were used for this section of the code as well.

It is worth noting that the data arrays transferred to the GPU (flow direction) and from it (watershed labels) are both of 8-bit values. The index array, which requires considerably more memory, is created and processed solely on the GPU, never being transferred. This design aims to minimize the time needed for data transfers.

2.4. Reference algorithms

In order to be able to compare the performance of the proposed algorithm to the existing solutions, several reference algorithms were implemented. These implementations were prepared by the author of this paper, but are based on concepts from the existing literature. Special attention was paid to related studies involving parallel processing and GPU devices — the main techniques described there were adapted and used in performance comparisons.

2.4.1. Sequential recursive algorithm

As one of the simplest and most intuitive solutions to the problem, the recursive algorithm was implemented in a straightforward, single-threaded version. In the first step, the inverse flow direction matrix is prepared with each cell pointing to its inflow neighbors. Then, starting with the outlet cells, a recursive procedure is performed. The procedure "climbs" in a bottom-up manner, recursively invoking itself for each inflow neighbor. Each cell traversed in this way is marked as part of the watershed area, as it eventually flows down to the outlet point where the procedure started.

Since the reference algorithms should be able to work with multiple outlet points as well, it was necessary to handle a scenario where one of the outlet cells belongs to another, larger watershed area, marked with some other outlet point. A simple mechanism was implemented to limit recursive bottom-up climbing when one of these cells is reached. This approach eliminates scenarios where any sub-area is traversed more than once.

It is worth noting that the execution time of this algorithm depends on the total size of the watershed areas marked in the input data. The

recursive procedure only moves within these regions, ignoring the cells outside.

It is also worth emphasizing that in practical applications, this class of algorithms can lead to memory problems such as stack overflows. In this paper, these issues are not discussed in detail, as this implementation is treated purely as a simple benchmark for performance measurements.

2.4.2. Flow path tracing algorithm (GPU)

This implementation is based on the concept described in Eränen et al. (2014). In this algorithm, each raster cell is assigned its own individual thread on the GPU device. The main idea is to follow the flow path of each cell individually, until a specified stream section or the DEM boundary is reached. Depending on where the cell's flow path ends, a corresponding watershed label is assigned to it.

The algorithm begins with the preparation and transfer of input data (flow direction and outlet cell markers) to the GPU memory. There, a working array is prepared where the outlet cells are initialized with their watershed labels. Initially, all other cells contain the NONE value. The goal of the algorithm is to assign each cell to the correct watershed and update its label.

The main part of this implementation is the path tracing kernel, based on an inner loop. The procedure is performed for each cell that initially does not have a watershed label assigned to it. Each thread starts by reading the flow direction value of its cell and then follows it to its downstream neighbor. This step is repeated until a given thread reaches the outlet point (marked with no flow direction value) or crosses the DEM boundary. If the flow path ends at one of the outlet points, the watershed label is copied from there to the thread's starting cell. When this stage is completed, the array can be transferred back to the host memory.

It should be highlighted that this idea involves different threads doing the same work multiple times. A single cell in the flow path will be repeatedly traversed by different threads starting from its upstream cells. While the parallel nature of GPU devices accelerates processing, this way of organizing work is a significant drawback of this algorithm.

As in the flow path reduction algorithm, data reshaping and manipulation steps on the host side (both at the first and the last stage) were parallelized using OpenMP.

2.4.3. Label pulling algorithm (GPU)

Another reference implementation is inspired by the concepts presented in Mower (1994) and Sit et al. (2019). Again, each cell in the raster is assigned its own individual thread on the GPU device. The main idea is to use the flow direction of a cell to locate its downstream neighbor, and then iteratively reach for the neighbor's watershed label. When a valid label becomes available, it is copied ("pulled up") to the thread's cell (and consequently becomes available to other threads).

The algorithm starts by adjusting the input data format and transferring it to the GPU (again, data manipulation on the host side is performed in parallel). There, the flow direction is used to calculate and store the downstream neighbor index for each cell.

Next, a working array is prepared, where the outlet cells are initialized with the corresponding labels and all other cells are initially set to NONE. The goal of the algorithm is to propagate the labels from these outlet points, gradually updating successive upstream cells.

The main loop of this algorithm iteratively calls the GPU kernel. Each thread reads its own cell first. If the cell does not have a valid (other than NONE) label yet, the label of its downstream neighbor is read. If this label is found to be valid, it is copied into the current cell. The kernel is iteratively invoked as long as at least one thread reports that a modification has been made.

It should be emphasized that although many threads work in parallel here, only a small fraction of all operations are meaningful. In each iteration, a given thread reads the label from its cell and reaches its downstream neighbor if needed. Of all these cycles, only one results in an actual modification of that thread's cell. In all other iterations, the thread either does not yet have a valid label from its neighbor or has already copied it. The number of kernel calls needed to produce a complete result is directly, linearly related to the length of the flow path with the highest cell count.

3. Performance measurements

3.1. Data

The largest dataset used in this work was a DEM with over two billion cells, originating from publicly available resources of the Head Office of Geodesy and Cartography (GUGiK). The data covered a part of south-eastern Poland (one-meter resolution, PL-1992 coordinate system). The DEM underwent a depression-filling procedure to ensure its hydrological correctness. More than 90% of all cells in this dataset belong to a single watershed area.

This DEM was used to prepare flow direction rasters, later used as input data for the algorithms. The largest raster was generated directly from the entire DEM. Next, the dataset was scaled down to prepare smaller test cases with approximately the same internal characteristics (covering the same area of land, but at different scales). A total of 30 input datasets were prepared (one with original size and 29 scaled down), with the number of cells increasing linearly between approximately 67.5 million and 2 billion. This approach was adopted to make it easier to examine the relationship between data size and algorithm performance, while minimizing other differences in data characteristics.

3.2. Testing procedure

The direct input for all tested algorithms were flow direction rasters, loaded from previously prepared files. Before starting the actual tests, their correctness was verified.

Before the measurements, each algorithm passed through multiple series of automated tests to confirm that its implementation was correct. Various scenarios, including specific corner cases, different data sizes and multithreading configurations were taken into account.

The performance of the algorithms was measured by repeatedly running each one on every dataset and recording execution times. In each test, the measurement application was restarted to reflect a realistic use case and eliminate the potential distortion of execution times related to the cache memory. Every test included loading a flow direction file, starting time measurement, executing a given algorithm and stopping the measurement. All operations needed to produce the final result (performed after successfully loading the flow direction file into memory) were treated as part of the algorithm and included in the time measurements. Additionally, the generated result was verified after each execution.

All tests were performed on a machine equipped with two Intel Xeon E5-2670 v3 processors (24 cores in total), 128 GB RAM and NVIDIA A100 Tensor Core GPU (in 40 GB memory version). The computer was running under AlmaLinux 8.4. The source code was compiled with the NVCC compiler from the CUDA Toolkit v11.2. Code optimization (-O3 flag) and OpenMP support were enabled.

For GPU computation, the maximum available thread block size (1024) was used. It was assumed that the device is initialized and synchronized. The code sections executed in parallel on the host had all 24 cores available.

As part of the performance comparison, each tested algorithm was executed ten times on every input dataset. The task was to delineate the single largest watershed area in the dataset (a single outlet cell was marked).

Table 1
Average execution times (in milliseconds) for selected datasets.

Dataset scale	Cells	Flow path reduction (single buffer)	Flow path reduction (back buffer)	Recursive sequential	Flow path tracing	Label pulling
0.1	202,492,900	159	200	4280	8733	39,596
0.2	404,995,500	292	372	8755	23,449	102,122
0.3	607,499,256	419	488	13,680	41,727	177,975
0.4	810,000,060	537	618	17,886	62,076	257,470
0.5	1,012,480,580	664	778	22,194	84,676	341,582
0.6	1,214,975,592	780	962	26,640	107,384	426,723
0.7	1,417,484,850	927	1127	31,091	130,595	514,730
0.8	1,620,022,250	998	1234	35,478	153,706	599,035
0.9	1,822,478,790	1078	1315	39,286	175,091	680,408
1.0	2,025,045,000	1286	1541	41,628	191,538	761,085

4. Results

4.1. Performance comparison

All results generated in each test passed the automated validation.

The time measurements clearly show that the flow path reduction algorithm performed by far the best in all cases. Both implementations of this approach achieved significantly shorter execution times than all the other tested algorithms.

The single-buffer version of the algorithm achieved execution times shorter by over 18% on average, compared to the back buffer implementation. This result is not surprising, considering the memory access patterns and differences in the number of iterations needed, as described earlier.

Averaging across all datasets, the recursive implementation took approximately 33 times longer to generate the result than the single-buffer flow path reduction algorithm. Taking the sequential recursive approach as a straightforward reference solution, this can be considered a significant improvement.

The flow path reduction algorithm also proved to be significantly more efficient than the GPU-based reference solutions. In general, the measured execution times varied by two orders of magnitude. Across all datasets, the flow path tracing algorithm took, on average, as much as 119 times longer to execute than the single-buffer implementation. For the label pulling algorithm, the average ratio was even greater, reaching 483 times. In both cases, the difference becomes more significant as the data size increases.

It is worth emphasizing that the two GPU-based reference algorithms turned out to be actually less efficient than simple sequential recursion. As mentioned earlier, much of the computation performed by parallel versions of these techniques is in fact redundant, which may help explain these results. The label pulling algorithm achieved by far the longest execution times of all tested implementations. Although these measurements are not directly comparable to those presented in Sit et al. (2019), the results appear to be somewhat in line with the fact that the GPU-based label pulling technique tested there achieved longer execution times than some sequential implementations.

Table 1 shows the average execution times for selected datasets. Fig. 1 presents a visual comparison of the average execution times for all datasets (results of the back buffer version are not present here as they were indistinguishable from the single-buffer version at this scale). Detailed results of all measurements are available in the public repository.

4.2. Further analysis

After the performance comparison, additional tests and measurements were performed to further investigate the properties of the flow path reduction algorithm (in the single-buffer version). In particular, it was examined which stages of the algorithm require the most computational time. A series of tests was performed using the largest dataset to measure the execution time of each stage individually.

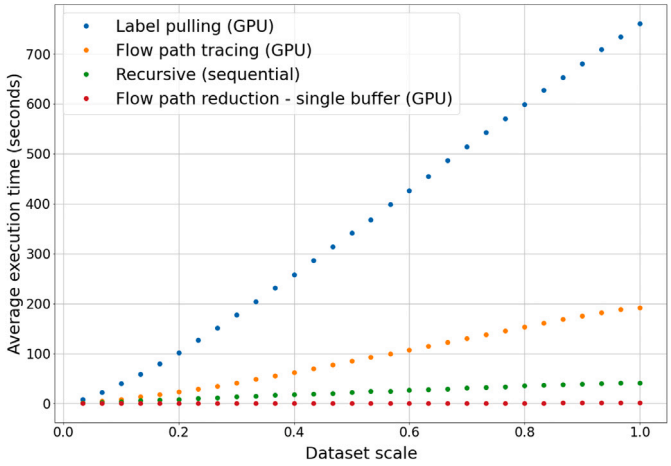


Fig. 1. Average execution times of tested algorithms across all datasets.

Although the main loop of the algorithm (repeatedly invoking the GPU kernel) is the most time-consuming step, it turned out to account for only 28.8% of the total execution time on average. This may seem surprising, but it can be explained by the fact that this key operation is effectively parallelized on the GPU and is performed relatively quickly compared to the other stages.

Data transfers accounted for a large part of the execution time — transferring input data to the GPU and retrieving the results back to the host added up to 34.5% of the total time on average. Also data reshaping operations, despite being parallelized on the host side, added up to a significant 33.1%. Table 2 presents the averaged measurements of all individual stages.

Another series of tests was carried out to verify whether the algorithm is able to delineate multiple watersheds simultaneously without the need for additional computational time. Measurements were performed using the largest dataset. Up to eight watersheds were delineated simultaneously. Table 3 shows the average execution times. There were no significant deviations from the time needed to delineate a single watershed using the same dataset.

5. Conclusions

As part of this work, the watershed delineation algorithms available in the literature were reviewed. Issues related to their performance were identified, indicating a need for improvement. A new algorithm was developed and presented, proposing an approach that effectively uses the massive parallelism of GPU devices to rapidly delineate watershed areas. Its performance was measured and compared with reference algorithms based on concepts existing in the literature.

The results show that the proposed approach allows for a significant reduction in the computational time needed to perform the watershed delineation compared to other tested algorithms. As noted in Haag

Table 2
Flow path reduction algorithm (single buffer) — computational time of individual stages.

Algorithm stage	Average execution time (in milliseconds)	Percentage share of total execution time
GPU allocation	11	0.8%
Input reshaping	158	12.2%
Transfer to GPU	231	17.9%
Direction to index	23	1.8%
Main loop	373	28.8%
Index to label	13	1.0%
Transfer from GPU	216	16.6%
Result reshaping	271	20.9%

Table 3
Flow path reduction algorithm (single buffer) — delineating multiple watersheds simultaneously.

Number of watersheds	Average execution time (in milliseconds)
2	1283
3	1304
4	1275
5	1258
6	1263
7	1259
8	1293

et al. (2018), the existing techniques are insufficient to delineate the watershed boundaries 'on the fly'. The author believes that this work can help fill this gap, allowing for a fast delineation of watersheds directly from basic flow direction data, even on larger datasets.

The author decided not to use existing GIS software packages as references in performance comparisons, instead focusing on parallel algorithms available in the literature. Although accurate measurements were not carried out, some early tests showed that it is not unusual for popular GIS platforms to take several orders of magnitude longer to perform similar operations on a machine with the same specification. It seems clear that there is still room for significant improvement.

It is worth emphasizing that the solution presented in this paper makes it possible to delineate multiple watersheds simultaneously (as well as use multiple outlet cells for the same watershed) with practically zero additional computational costs. This can be considered a significant advantage of this method, especially in the context of the cited literature.

In fact, delineating a single watershed area could be considered a special case of the more general task discussed here. An implementation focusing specifically on this scenario could possibly reduce the computation time of some stages even further.

While the techniques and ideas presented in this paper were designed for square-grid DEMs, they seem to be easily applicable to other types of data as well. This may prove important, as recent studies focused on hexagonal grids have shown promising results (Liao et al., 2020). In addition, although the presented concepts were developed specifically for watershed delineation, perhaps they could also be adapted to other similarly structured modeling issues.

Software availability

- Software: Watershed delineation algorithms
- Description: Source code of all algorithms developed, tested and presented as part of this work (including a simple measurement application)
- Developer: Bartłomiej Kotyra
- Contact address: bartlomiej.kotyra@mail.umcs.pl
- Language: C++11, CUDA, OpenMP
- Libraries required: GDAL (Geospatial Data Abstraction Library)
- Availability: Freely available at https://github.com/bkotyra/watershed_delineation_gpu/

Declaration of competing interest

The author declares that he has no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper. This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Data availability

The source code is available in a public repository. The data used in performance measurement originate from publicly available resources (referenced in the manuscript).

References

Arge, L., Chase, J., Halpin, P., Toma, L., Vitter, J., Urban, D., Wickremesinghe, R., 2003. Efficient flow computation on massive grid terrain datasets. GeoInformatica 7, 283–313. http://dx.doi.org/10.1023/A:1025526421410.

Baker, M.E., Weller, D.E., Jordan, T.E., 2006. Comparison of automated watershed delineations: Effects on Land Cover Areas, percentages, and relationships to nutrient discharge. Photogramm. Eng. Remote Sens. 72 (2), 159–168. http://dx.doi.org/10.14358/PERS.72.2.159.

Barnes, R., 2017. Parallel non-divergent flow accumulation for trillion cell digital elevation models on desktops or clusters. Environ. Model. Softw. 92, 202–212. http://dx.doi.org/10.1016/j.envsoft.2017.02.022.

Barnes, R., Lehman, C., Mulla, D., 2014. Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models. Comput. Geosci. 62, 117–127. http://dx.doi.org/10.1016/j.cageo.2013.04.024.

Barták, V., 2009. How to extract river networks and catchment boundaries from DEM: a review of digital terrain analysis techniques. J. Landsc. Stud. 2, 57–68.

Beucher, S., Meyer, F., 1993. The morphological approach to segmentation: The watershed transformation. 34, pp. 433–481. http://dx.doi.org/10.1201/9781482277234-12,

Castronova, A.M., Goodall, J.L., 2014. A hierarchical network-based algorithm for multi-scale watershed delineation. Comput. Geosci. 72, 156–166. http://dx.doi.org/10.1016/j.cageo.2014.07.014.

Chapman, B., Jost, G., Pas, R.v.d., 2007. Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press.

Chen, B., Ma, C., Xiao, Y., Gao, H., Shi, P., Zheng, J., 2021. Retaining relative height information: An enhanced technique for depression treatment in digital elevation models. Water 13 (23), http://dx.doi.org/10.3390/w13233347.

Cheng, J., Grossman, M., McKercher, T. (Eds.), 2014. Professional CUDA C Programming. Wiley and Sons.

Choi, J.-Y., Engel, B., 2003. Real-time watershed delineation system using Web-GIS. J. Comput. Civ. Eng. 17, http://dx.doi.org/10.1061/(ASCE)0887-3801(2003)17:3(189).

Chow, V., Maidment, D., Mays, L., 1988. Applied Hydrology. McGraw-Hill.

Daniel, E., 2011. Watershed modeling and its applications: A state-of-the-art review. Open Hydrol. J. 5, 26–50. http://dx.doi.org/10.2174/1874378101105010026.

de Azeredo Freitas, H.R., da Costa Freitas, C., Rosim, S., de Freitas Oliveira, J.R., 2016. Drainage networks and watersheds delineation derived from TIN-based digital elevation models. Comput. Geosci. 92, 21–37. http://dx.doi.org/10.1016/j.cageo.2016.04.003.

Digabel, H., Lantuéjoul, C., 1978. Iterative algorithms. In: Verlag, R. (Ed.), Proceedings of the 2nd European Symposium Quantitative Analysis of Microstructures in Material Science, Biology and Medicine. pp. 85–99.

Dmitruk, K., Denkowski, M., Mikołajczak, P., Benedykciuk, E., 2021. The method for adaptive material classification and pseudo-coloring of the baggage X-Ray images. In: Computer Analysis of Images and Patterns. Springer International Publishing, Cham, pp. 75–87. http://dx.doi.org/10.1007/978-3-030-89131-2_7.

Do, H.-T., Limet, S., Melin, E., 2011. Parallel computing flow accumulation in large digital elevation models. Procedia Comput. Sci. 4, 2277–2286. http://dx.doi.org/10.1016/j.procs.2011.04.248, Proceedings of the International Conference on Computational Science, ICCS 2011.

Eränen, D., Oksanen, J., Westerholm, J., Sarjakoski, T., 2014. A full graphics processing unit implementation of uncertainty-aware drainage basin delineation. Comput. Geosci. 73, 48–60. http://dx.doi.org/10.1016/j.cageo.2014.08.012.

Fairfield, J., Leymarie, P., 1991. Drainage networks from grid digital elevation models. Water Resour. Res. 27 (5), 709–717. http://dx.doi.org/10.1029/90WR02658.

Freeman, T., 1991. Calculating catchment area with divergent flow based on a regular grid. Comput. Geosci. 17 (3), 413–422. http://dx.doi.org/10.1016/0098-3004(91)90048-I.

Golub, G., Ortega, J.M., 1993. Chapter 8 - iterative methods. In: Golub, G., Ortega, J.M. (Eds.), Scientific Computing. Academic Press, San Diego, pp. 321–369. http://dx.doi.org/10.1016/B978-0-12-289253-0.50012-9.

Haag, S., Schwartz, D., Shakibajahromi, B., Campagna, M., Shokoufandeh, A., 2020. A fast algorithm to delineate watershed boundaries for simple geometries. Environ. Model. Softw. 134, 104842. http://dx.doi.org/10.1016/j.envsoft.2020.104842.

Haag, S., Shakibajahromi, B., Shokoufandeh, A., 2018. A new rapid watershed delineation algorithm for 2D flow direction grids. Environ. Model. Softw. 109, 420–428. http://dx.doi.org/10.1016/j.envsoft.2018.08.017.

Hillis, W.D., Steele, G.L., 1986. Data parallel algorithms. Commun. ACM 29 (12), 1170–1183. http://dx.doi.org/10.1145/7902.7903.

Jenson, S.K., Domingue, J.O., 1988. Extracting topographic structure from digital elevation data for geographic information system analysis. Photogramm. Eng. Remote Sens. 54 (11), 1593–1600.

Jones, N., Wright, S., Maidment, D., 1990. Watershed delineation with triangle-based terrain models. J. Hydraul. Eng. 116, http://dx.doi.org/10.1061/(ASCE)0733-9429(1990)116:10(1232).

Karimipour, F., Ghandehari, M., Ledoux, H., 2013. Watershed delineation from the medial axis of river networks. Comput. Geosci. 59, 132–147. http://dx.doi.org/10.1016/j.cageo.2013.06.004.

Kornilov, A.S., Safonov, I.V., 2018. An overview of watershed algorithm implementations in open source libraries. J. Imaging 4 (10), http://dx.doi.org/10.3390/jimaging4100123.

Kornilov, A., Safonov, I., Yakimchuk, I., 2022. A review of watershed implementations for segmentation of volumetric images. J. Imaging 8 (5), http://dx.doi.org/10.3390/jimaging8050127.

Kotyra, B., Chabudziński, L., Stpiczyński, P., 2021. High-performance parallel implementations of flow accumulation algorithms for multicore architectures. Comput. Geosci. 151, 104741. http://dx.doi.org/10.1016/j.cageo.2021.104741.

Liao, C., Tesfa, T., Duan, Z., Leung, L.R., 2020. Watershed delineation on a hexagonal mesh grid. Environ. Model. Softw. 128, 104702. http://dx.doi.org/10.1016/j.envsoft.2020.104702.

Lindsay, J.B., 2016. Efficient hybrid breaching-filling sink removal methods for flow path enforcement in digital elevation models. Hydrol. Process. 30 (6), 846–857. http://dx.doi.org/10.1002/hyp.10648.

Lindsay, J.B., Rothwell, J.J., Davies, H., 2008. Mapping outlet points used for watershed delineation onto DEM-derived stream networks. Water Resour. Res. 44 (8), http://dx.doi.org/10.1029/2007WR006507.

López-Vicente, M., Pérez-Bielsa, C., López-Montero, T., Lambán, L., Navas, A., 2014. Runoff simulation with eight different flow accumulation algorithms: Recommendations using a spatially distributed and open-source model. Environ. Model. Softw. 62, 11–21. http://dx.doi.org/10.1016/j.envsoft.2014.08.025.

Makinen, V., Sarjakoski, T., Oksanen, J., Westerholm, J., 2016. A multi-GPU program for uncertainty-aware drainage basin delineation: Scalability benchmarking with country-wide data sets. IEEE Geosci. Remote Sens. Mag. 4 (3), 59–68. http://dx.doi.org/10.1109/MGRS.2016.2561405.

Marks, D., Dozier, J., Frew, J., 1984. Automated basin delineation from digital elevation data. Geo-Processing 2, 299–311.

Martz, L.W., Garbrecht, J., 1993. Automated extraction of drainage network and watershed data from digital elevation models. J. Am. Water Resour. Assoc. 29 (6), 901–908. http://dx.doi.org/10.1111/j.1752-1688.1993.tb03250.x.

McGough, A.S., Liang, S., Rapoportas, M., Grey, R., Vinod, G.K., Maddy, D., Trueman, A., Wainwright, J., 2012. Massively parallel landscape-evolution modelling using general purpose graphical processing units. In: 2012 19th International Conference on High Performance Computing. pp. 1–10. http://dx.doi.org/10.1109/HiPC.2012.6507488.

Minaee, S., Boykov, Y., Porikli, F., Plaza, A., Kehtarnavaz, N., Terzopoulos, D., 2022. Image segmentation using deep learning: A survey. IEEE Trans. Pattern Anal. Mach. Intell. 44 (7), 3523–3542. http://dx.doi.org/10.1109/TPAMI.2021.3059968.

Mower, J.E., 1994. Data-parallel procedures for drainage basin analysis. Comput. Geosci. 20 (9), 1365–1378. http://dx.doi.org/10.1016/0098-3004(94)90060-4.

Nelson, E., Jones, N., Miller, A., 1994. Algorithm for precise drainage-basin delineation. J. Hydraul. Eng. 120, http://dx.doi.org/10.1061/(ASCE)0733-9429(1994)120:3(298).

NVIDIA, 2022. CUDA C++ Programming Guide, version 11.7.1. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

O'Callaghan, J.F., Mark, D.M., 1984. The extraction of drainage networks from digital elevation data. Comput. Vis. Graph. Image Process. 28 (3), 323–344. http://dx.doi.org/10.1016/S0734-189X(84)80011-0.

Ortega, L., Rueda, A., 2010. Parallel drainage network computation on CUDA. Comput. Geosci. 36 (2), 171–178. http://dx.doi.org/10.1016/j.cageo.2009.07.005.

Planchon, O., Darboux, F., 2002. A fast, simple and versatile algorithm to fill the depressions of digital elevation models. CATENA 46 (2), 159–176. http://dx.doi.org/10.1016/S0341-8162(01)00164-3.

Qin, C.-Z., Zhan, L., 2012. Parallelizing flow-accumulation calculations on graphics processing units—From iterative DEM preprocessing algorithm to recursive multiple-flow-direction algorithm. Comput. Geosci. 43, 7–16. http://dx.doi.org/10.1016/j.cageo.2012.02.022.

Roerdink, J., Meijster, A., 2000. The watershed transform: Definitions, algorithms and parallelization strategies. Fund. Inform. 41, http://dx.doi.org/10.3233/FI-2000-411207.

Rueda, A.J., Noguera, J.M., Luque, A., 2016. A comparison of native GPU computing versus OpenACC for implementing flow-routing algorithms in hydrological applications. Comput. Geosci. 87, 91–100. http://dx.doi.org/10.1016/j.cageo.2015.12.004.

Seibert, J., McGlynn, B.L., 2007. A new triangular multiple flow direction algorithm for computing upslope areas from gridded digital elevation models. Water Resour. Res. 43 (4), http://dx.doi.org/10.1029/2006WR005128.

Singh, V., 2018. Hydrologic modeling: progress and future directions. Geosci. Lett. 5, http://dx.doi.org/10.1186/s40562-018-0113-z.

Sit, M., Sermet, Y., Demir, I., 2019. Optimized watershed delineation library for server-side and client-side web applications. Open Geospat. Data Softw. Stand. 4, 8. http://dx.doi.org/10.1186/s40965-019-0068-9.

Sten, J., Lilja, H., Hyväluoma, J., Westerholm, J., Aspnäs, M., 2016. Parallel flow accumulation algorithms for graphical processing units with application to RUSLE model. Comput. Geosci. 89, 88–95. http://dx.doi.org/10.1016/j.cageo.2016.01.006.

Tang, W., Wang, S., 2020. High performance computing for geospatial applications. http://dx.doi.org/10.1007/978-3-030-47998-5.

Tarboton, D.G., 1997. A new method for the determination of flow directions and upslope areas in grid digital elevation models. Water Resour. Res. 33 (2), 309–319. http://dx.doi.org/10.1029/96WR03137.

Tarboton, D., Watson, D., Wallace, R., Schreuders, K., Tesfa, T., 2009. Hydrologic Terrain Processing Using Parallel Computing. Civil and Environmental Engineering Faculty Publications. Paper 2715, p. 0867.

Tesfa, T.K., Tarboton, D.G., Watson, D.W., Schreuders, K.A., Baker, M.E., Wallace, R.M., 2011. Extraction of hydrological proximity measures from DEMs using parallel processing. Environ. Model. Softw. 26 (12), 1696–1709. http://dx.doi.org/10.1016/j.envsoft.2011.07.018.

Vitor, G., Körbes, A., Lotufo, R., Ferreira, J., 2010. Analysis of a step-based watershed algorithm using CUDA. Int. J. Nat. Comput. Res. 1, 16–28. http://dx.doi.org/10.4018/jncr.2010100102.

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. pp. 467–472.

Wang, L., Liu, H., 2006. An efficient method for identifying and filling surface depressions in digital elevation models for hydrologic analysis and modelling. Int. J. Geogr. Inf. Sci. 20 (2), 193–213. http://dx.doi.org/10.1080/13658810500433453.

Wilson, J., Aggett, G., Deng, Y., Lam, C., 2008. Water in the landscape: A review of contemporary flow routing algorithms. Adv. Digit. Terrain Anal. 213–236. http://dx.doi.org/10.1007/978-3-540-77800-4_12.

Wu, Q., Chen, Y., Wilson, J.P., Liu, X., Li, H., 2019. An effective parallelization algorithm for DEM generalization based on CUDA. Environ. Model. Softw. 114, 64–74. http://dx.doi.org/10.1016/j.envsoft.2019.01.002.

Yeghiazaryan, V., Voiculescu, I., 2018. Path reducing watershed for the GPU. In: 2018 IEEE Winter Conference on Applications of Computer Vision. WACV, pp. 577–585. http://dx.doi.org/10.1109/WACV.2018.00069.

Yuheng, S., Hao, Y., 2017. Image segmentation algorithms overview. http://dx.doi.org/10.48550/ARXIV.1707.02051.

Zhou, G., Sun, Z., Fu, S., 2016. An efficient variant of the Priority-Flood algorithm for filling depressions in raster digital elevation models. Comput. Geosci. 90, 87–96. http://dx.doi.org/10.1016/j.cageo.2016.02.021.

Zhou, G., Wei, H., Fu, S., 2019. A fast and simple algorithm for calculating flow accumulation matrices from raster digital elevation. Front. Earth Sci. 13 (2), 317–326. http://dx.doi.org/10.1007/s11707-018-0725-9.

Zhu, L.-J., Liu, J., Qin, C.-Z., Zhu, A.-X., 2019. A modular and parallelized watershed modeling framework. Environ. Model. Softw. 122, 104526. http://dx.doi.org/10.1016/j.envsoft.2019.104526.

Dodatek C

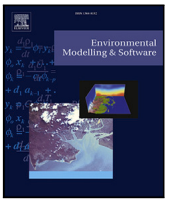
Fast parallel algorithms for finding the longest flow paths in flow direction grids

Environmental Modelling and Software 167 (2023) 105728

Contents lists available at ScienceDirect

Environmental Modelling and Software

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



Position Paper

Fast parallel algorithms for finding the longest flow paths in flow direction grids

Bartłomiej Kotyra [a,*], Łukasz Chabudziński [b]

[a] Maria Curie-Skłodowska University, Institute of Computer Science, ul. Akademicka 9, 20-033 Lublin, Poland
[b] Maria Curie-Skłodowska University, Institute of Earth and Environmental Sciences, al. Kraśnicka 2d, 20-718 Lublin, Poland

ARTICLE INFO

Keywords:
Longest flow path
GIS
Hydrology
Parallel algorithms
High-performance computing
OpenMP

ABSTRACT

In hydrological modeling, the longest flow path is an important feature used to characterize a catchment. Many existing GIS platforms offer dedicated software tools for its identification and delineation, generally implementing methods based on searching through the flow direction data. Unfortunately, currently available algorithms for this task often turn out to be inefficient, especially when working with modern large datasets. Moreover, existing methods often rely on incorrect assumptions or perform calculations in a way that can lead to precision issues. In this work, new parallel algorithms were developed, tested and presented. Measurements show that two of the newly proposed implementations are able to identify the longest flow paths in significantly less time compared with other existing methods.

Software availability

Software: Algorithms for finding the longest flow paths
Description: Source code of all algorithms developed, tested and presented as part of this work (including a simple measurement application)
Developer: Bartłomiej Kotyra
Contact address: bartlomiej.kotyra@mail.umcs.pl
Language: C++11, OpenMP
Libraries required: GDAL (Geospatial Data Abstraction Library)
Availability: Freely available at https://github.com/bkotyra/longest_flow_path/

1. Introduction

In the area of hydrological modeling, catchments are characterized and described using multiple features and various parameters. One of the important ones is the longest flow path, also referred to as the longest watercourse or the longest drainage path (Dawson et al., 2006; Huang and Lee, 2016; Michailidi et al., 2018).

In the hydrological context, a flow path is understood as a route followed by water draining from one point to another. There are multiple flow paths of varying lengths in any given catchment. The longest one leads to the watershed outlet, usually starting at some location on the watershed boundary, although this is not always the case (Cho, 2020).

The longest flow path is typically used to determine the attributes for hydrological model parameterization, mainly the time of concentration and the lag time of a watershed (Michailidi et al., 2018; Sultan et al., 2022). It can also be used as a basis for calculating some of the properties and characteristics of a catchment (Maathuis and Wang, 2006; Jaffrés et al., 2021). The longest flow path, as well as other features derived from it, may be useful in estimating certain flood-related indices (Karalis et al., 2014; Latt et al., 2015).

Both early and modern GIS software packages consider the longest flow path as an essential element in hydrological modeling and provide tools for its delineation and analysis (Gallant and Wilson, 1996; Maidment and Morehouse, 2002; Merkel et al., 2008; Ramly and Tahir, 2016). Recent literature draws attention to the time-consuming nature of this operation and the low processing speed of the commonly used tools (Castro and Maidment, 2020). Attempts are still being made to develop new, more efficient algorithms for this task (Cho, 2020).

The geospatial data available today offers high resolution and unprecedented precision, but this comes at the cost of significant dataset sizes, creating new computational challenges (Sten et al., 2016). At the same time, modern hardware architectures make the access to multicore and many-core processors more and more common. Programming standards like OpenMP make it relatively easy to achieve high computing performance on these devices due to parallelism (Chapman et al., 2007). However, designing and implementing parallel algorithms still remains a non-trivial task, leading to a significant gap between

* Corresponding author.
E-mail addresses: bartlomiej.kotyra@mail.umcs.pl (B. Kotyra), lukasz.chabudzinski@mail.umcs.pl (Ł. Chabudziński).

https://doi.org/10.1016/j.envsoft.2023.105728
Received 25 November 2022; Accepted 16 May 2023
Available online 24 May 2023

the existing technology and its practical applications (Tang and Wang, 2020). The available software often fails to efficiently use the underlying hardware and proves to be impractical in the context of modern, large datasets.

The main motivation behind this research was the need for more efficient tools and techniques for finding the longest flow paths in geospatial data. As can be concluded from a review of the available software and literature, there is still room for significant improvement. The goal of this study was to develop and present new, fast raster-based algorithms for solving this task, with particular attention to the possibilities offered by modern multicore architectures. The authors believe that this work improves upon existing solutions and takes a significant step towards higher performance.

The remainder of this chapter outlines the foundations of existing techniques for finding the longest flow paths in raster data, and describes the available algorithms and software tools for this task. Section 2 specifies the constraints of the problem to be solved (along with its specific variants) and describes the approach used in this work to design and implement new, more efficient solutions. In Section 3, all the developed algorithms (a total of seven implementations, including two reference ones) are presented and discussed in detail. Section 4 describes the data and procedures used to measure and evaluate the performance of the proposed algorithms, including comparisons with existing GIS software. The results of these measurements are presented and discussed in Section 5. Finally, the conclusions of this study, its limitations and possible future work are briefly summarized in Section 6.

1.1. Existing techniques

The existing literature related to finding the longest flow path, as well as the available software tools, generally refer to and build upon the approach based on square-grid digital elevation models (DEMs) and their derivative data (specifically flow direction rasters). The basic concepts behind these methods are well known and have been widely used for decades (O'Callaghan and Mark, 1984; Jenson and Domingue, 1988).

In the literature, there are two significantly different approaches to the concept of flow direction. In the single-flow approach, each raster cell points to one of its immediate neighbors, directing all flow to a single downstream cell. In the multiple-flow approach, the flow may be proportionally divided among more than one neighbor (Schäuble et al., 2008). Both approaches have multiple practical applications, as some problems are naturally better expressed in terms of one or the other (Barták, 2009). However, different algorithms for determining the flow direction can lead to significantly varying results (Huang and Lee, 2016). The single-flow approach is most often considered more appropriate for determining and analyzing flow paths, as multiple-flow algorithms lead to excessive dispersion and overestimated path lengths (Orlandini and Moretti, 2009; Li et al., 2020).

The length of a given flow path is typically estimated as the sum of the distances between successive raster cells belonging to that path (Paz et al., 2008). Fig. 1 illustrates this fundamental idea. The flow path can be conceptually expressed as a line connecting the centers of consecutive downstream cells. The center-to-center distance between orthogonal (horizontal and vertical) neighbors is considered equal to the dimension of a single cell, and the distance between diagonal neighbors is calculated as the dimension of the cell multiplied by the square root of 2 (≈1.414). Although this measure is essentially a simple approximation, it is generally regarded as being relatively consistent with the actual length of the flow path (Fairfield and Leymarie, 1991; Paz et al., 2008). The flow direction raster is used as fundamental data in these calculations, as it allows any flow path to be traced cell by cell (Smith, 1997; Cho, 2020; Lindsay, 2022). It should be noted that while this kind of approach may be considered unsuitable for some other hydrological modeling tasks (i.e. requiring consideration of a wider range of factors), it is broadly accepted and implemented as a standard method for quickly comparing flow paths when searching for the longest one.

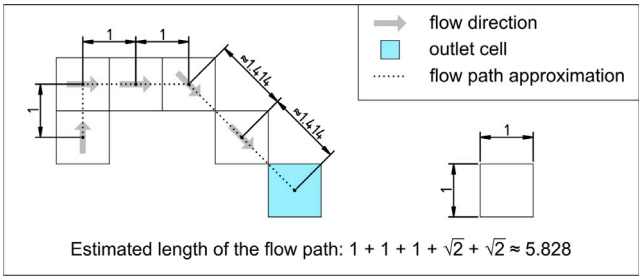


Fig. 1. Estimating the length of a flow path by accumulating distances between its successive cells.

The remaining sections of this paper deal with techniques, algorithms and tools based on these foundations.

1.1.1. Algorithms described in the literature

The earliest published works discussing the algorithmic determination of the longest flow path, of which the authors are aware, were presented in Smith (1995, 1997). The papers are focused on the architecture of a software system called the Hydrologic Data Development System (HDDS). It uses spatial-analysis techniques to determine multiple hydrologic parameters, including the length and location of the longest flow path.

The approach used in the HDDS is based on square-grid DEMs. The procedure of determining the longest flow path begins with the preparation of two additional rasters, which are calculated using the flow direction data. In the first one, each cell contains its downstream flow length (which is defined as the downslope distance to a pour point). The second one consists of upstream flow length values (defined as the distance to the drainage divide). The values of the corresponding cells from both rasters are then added together. The highest values obtained in this way are equal to the length of the longest flow path. Importantly, only cells belonging to the longest path can obtain the maximum value, which allows them to be easily identified (all other cells obtain lower values).

Olivera and Maidment (1998) presented another grid-based geographic information system for hydrologic modeling. The method for determining the longest flow path described here is generally the same as in Smith (1997). The system requires both the upstream and downstream flow length rasters as input data. These rasters are added together, and then the set of cells with the highest sum is identified as the longest flow path. This approach was later described in more detail by the same author in Olivera (2001).

Cho (2020) proposed using a recursive approach to improve performance over existing techniques and reduce computational time needed. The basic idea behind this method is to search the flow direction raster by starting at the outlet point and moving upstream (recursively invoking the calculation of the longest flow path for topographically higher cells). This approach makes it possible to avoid calculating downstream and upstream flow length rasters. Due to memory issues typical of recursion (i.e. stack overflows), an iterative version of the algorithm was also presented and recommended for use.

Developing this concept further, Cho (2020) proposed a strategy based on Hack's law (Hack, 1957) to filter out some of the flow direction branches as early as possible and stop the recursive procedure from traversing them. This method uses flow accumulation values (calculated using flow direction data) to estimate potential longest flow lengths. The cells that cannot lead to the longest flow path are rejected early. While this approach does avoid traversing some of the suboptimal areas, it requires flow accumulation to be calculated, which is a time-consuming step in itself.

Apart from the mentioned works, little attention has been paid to the problem of algorithmic determination of the longest flow paths. Huang and Lee (2016) investigated the impact of various flow direction algorithms on selected geomorphological properties, including the longest watercourse, but specific methods for determining the longest path were not discussed here. Related studies discussing the accuracy of flow paths derived from DEMs were presented in Paz et al. (2008), Orlandini and Moretti (2009) and Li et al. (2020).

The authors of this work are unaware of any previous studies focused on developing parallel algorithms for identifying the longest flow paths.

1.1.2. Existing software tools

The widespread use of the longest flow path in hydrological analysis has led to the development of dedicated tools for its delineation in various GIS applications (Maidment and Morehouse, 2002; Cho, 2020; Lindsay, 2022) and hydrological modeling software (Arnold et al., 1998; Feldman, 2000). Depending on the tool, different methods may be used to calculate the result.

On many GIS platforms, it is possible to delineate the longest flow path by following a procedure similar to the one proposed in Smith (1997). Even if a dedicated tool for this task is not available, it may still be possible to identify the longest flow path, as long as the platform provides the ability to compute flow-length rasters. For this kind of approach, the flow direction raster is used as the main input data.

Using dedicated tools to find the longest flow path is usually straightforward. Unfortunately, the underlying algorithms and procedures are not always openly described by their developers.

The Longest Flow Path tool from the Arc Hydro package requires a catchment (or possibly multiple ones) as a vector layer, along with a flow direction raster (Djokic et al., 2011). The vector layer attributes must include a HydroID field with unique values identifying each subcatchment. It is important to note that this tool does not take outlet points as input, and the algorithm can in fact delineate the longest path beyond the catchment boundary (stopping at the edge of the available data). Therefore, in order to correctly determine the path within a given catchment, it is necessary to first remove the flow direction values outside its boundaries (by setting them to NoData) or to properly clip the raster to the area.

Regardless of the characteristics of the drainage network, Arc Hydro delineates only a single path. By the underlying assumption, the path can only begin in the watershed boundary zone (which, in fact, is not always the case). The result is returned as a vector layer.

It is worth noting that due to the widespread popularity of Arc Hydro, some of its features (including the Longest Flow Path tool) are often directly integrated into other hydrological modeling applications (Olivera et al., 2003; Merkel et al., 2008).

Multiple modules for determining the longest flow path were developed and made available for the open-source GRASS GIS platform (Neteler et al., 2012). The most recent one, named r.accumulate, was described in detail by the developer in Cho (2020). As input, it takes outlet points (marking the endpoints of the paths searched) and a drainage direction raster (which can be generated in the GRASS environment using the r.watershed tool). Depending on the drainage network, the end result may be a single path or multiple alternative ones. The tool returns them as a vector layer where each path is a separate object.

WhiteboxTools is another advanced, although less known platform with a dedicated tool for determining the longest flow path (Lindsay, 2016, 2022). The LongestFlowpath tool operates on a depressionless DEM (calculating the flow direction internally), along with a raster that defines one or more areas to be analyzed. The design assumes that the longest flow paths must start at the watershed boundaries (which, again, may not be true in some cases). The results are returned as vector objects.

In recent years, an increasing number of software solutions, including GIS and hydrological modeling platforms, base their architecture on web services and cloud computing (Goodall et al., 2011; Ames et al., 2012; Vitolo et al., 2015; Gichamo et al., 2020). Just a single example of such a tool is SCALGO Live (SCALGO, 2022). The platform, designed and built on a web-based architecture, is intended for broadly understood water management. Implementing a wide set of raster algorithms, the tool is able to automatically perform a variety of hydrological modeling operations, including determining the longest flow path in a given catchment.

To the knowledge of the authors of this work, there are currently no tools available that determine the longest flow paths using parallel algorithms.

2. Methods

Considering the numerous applications of the longest flow path in hydrological modeling, as well as the existence of various software tools for its determination and the attempts to improve their performance, it can be concluded that there is a need to develop new, more efficient algorithms for solving this task.

2.1. Adopted terminology

For referring to specific cell types in raster data, this paper adopts a vocabulary based on that proposed in Tarboton et al. (1991):

- source cell — a cell with no inflow neighbors (the most upstream cell in a given flow path, and thus its starting point)
- link cell — a cell with a single inflow neighbor
- junction cell — a cell with more than one inflow neighbor (where two or more drainage channels merge into one)
- outlet cell — the most downstream cell in a given catchment (where all drainage is eventually concentrated)

2.2. Problem specification

The key problem to be solved by each of the developed algorithms is to identify the longest flow path among all those ending at a given outlet point.

It is assumed that the underlying spatial data can be expressed in the form of square-grid rasters. The input dataset for the algorithms consists of a hydrologically correct flow direction raster (generated by any single-flow algorithm) and the location (possibly multiple locations) of the outlet point for which the longest flow path is to be found. As there are many single-flow direction algorithms with different properties (any of which could be preferred by the user), this form of data was chosen as the basic input rather than the raw digital elevation model. The location of the outlet cell (possibly multiple outlet cells) is determined by the user and considered as part of the input data.

While some existing tools require auxiliary input (e.g. a watershed mask raster), it is assumed that the algorithms considered in this work do not have access to any additional datasets. Any processing of the basic input data is considered part of the algorithm (and therefore reflected in performance measurements).

It is important to note that, in this work, the resolution of the input data (and thus the cell size) is generally treated as irrelevant. Assuming that all cells in the raster have the same dimensions, the process of identifying the longest flow path does not depend on their particular sizes. In fact, it is possible to find the longest path in a flow direction raster without knowing the resolution of the dataset. For this reason, all algorithms considered in this work were designed and implemented as resolution-independent. As a unit of measurement for comparing the lengths of different flow paths, the dimension of a single raster cell was adopted. An orthogonal step between cells is simply counted as 1, while a diagonal step is counted as the square root of 2 (e.g. a flow

path consisting of three vertical steps is considered to be 3 units long, regardless of raster resolution).

In terms of the expected result, it is important to allow the identified flow path to be precisely delineated, usually by marking all raster cells belonging to it or generating a corresponding vector shape. Therefore, the correct result of the algorithm will be the location of the starting point (possibly multiple locations of such points) of the longest flow path existing for the given input data. This approach allows for high flexibility in the presentation and use of the result obtained. Assuming the availability of flow direction data, delineating the path in any chosen form (e.g. vector object or raster with labeled cells) can be quickly performed by traversing successive cells from the starting point downwards.

It is important to emphasize that the longest flow path could potentially start from any cell within the catchment. Some existing tools implicitly assume that the longest path always starts at the watershed boundary, but this is simply not correct and may lead to inaccurate results (Cho, 2020).

2.2.1. Different variants of the problem

Considering the possible use cases, the distinctive needs of the end users, and the expected characteristics of the input data, it is possible to define several variants of the task to be solved.

In some cases, it is possible to find more than one longest flow path within the same catchment (each being equally long). Fig. 2 illustrates how different paths can be estimated to have the same length. It is necessary to specify the expected behavior of the algorithm in such scenarios, as this kind of decision may substantially reshape the task to be solved, and consequently lead to a different algorithm design. Finding any of the longest paths might be sufficient in some use cases (e.g. when only its length is relevant), while identifying all of the alternatives may be required in others (e.g. when the location of the selected path may affect subsequent stages of the analysis, and it would be reasonable to leave this choice to the user).

In this paper, both use cases are considered relevant. For datasets where more than one longest flow path could be found, the minimum requirement for all algorithms under development was the ability to correctly identify at least one of them (each being considered a valid result). However, the possibility of identifying all alternative longest paths for the same outlet point was also included in some designs. While not required of all algorithms, this capability was considered valuable and implemented where the underlying concept was compatible.

Another use case to consider is identifying the longest flow paths for multiple outlet points (e.g. analyzing more than one catchment within the same dataset simultaneously). As a practical scenario, this capability is offered by some of the existing tools. While it is always possible to sequentially execute the search for individual outlet points, a design that allows them to be processed together in a single algorithm run could have significant performance advantages. Depending on how the calculations are organized, it may be possible to reduce the time required to identify all the paths individually.

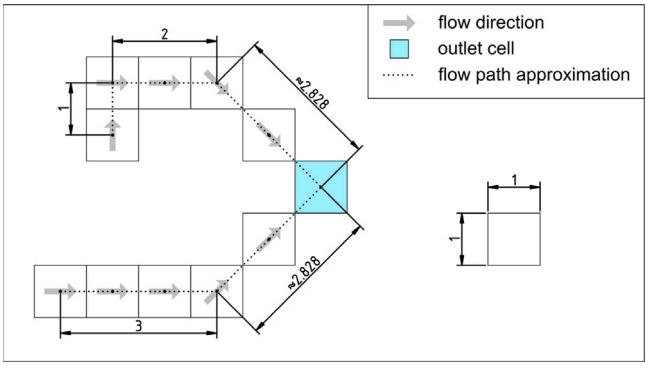


Fig. 2. Two flow paths of equal estimated lengths leading to the same outlet cell.

In this work, the minimum requirement for all considered algorithms was to be able to work with a single outlet cell. However, for some designs, the simultaneous identification of the longest flow paths for different outlet points would require little or virtually no additional computational cost. As a valuable property, this mode of operation was included in compatible implementations. It should be noted that this capability is only considered for performance reasons and does not affect the results generated by the algorithm.

2.3. Precision issues and proposed solution

Although the main focus of this study is related to algorithmic efficiency and performance, it is important to also highlight some common issues related to the accuracy of the calculations and their results. In particular, the typical way of implementing the path length estimation method can lead to unnecessary, gradual accumulation of round-off errors, which in turn may prevent the correct identification of the longest flow path. It should be emphasized that the issues discussed in this section are not related to imperfections in the input data, but rather to the way algorithms and software tools perform their computations on them.

Precision issues with floating point calculations are well known and have been considered since the early decades of computers (Wilkinson, 1963). One of the classic problems is sequence summation, where round-off errors accumulate with each step and can quickly become significant (Linz, 1970). It is well known that summing a set of floating point numbers by simply adding consecutive values to the total sum is not a reliable approach (Caprani, 1975).

As noted in Paz et al. (2008), the length of the flow path is usually calculated step by step, accumulating the distances between successive cells. Across all the literature and available source code known to the authors of this work, this operation is implemented using floating point calculations, by simply adding consecutive distances to the total sum. In practice, this approach can lead to a noticeable degree of inaccuracy in path length estimates.

It is relatively easy to find a case where the accumulation of round-off errors in existing GIS software results in an inaccurate length measurement, leading to the incorrect flow path being identified as the longest. To clearly describe and demonstrate the issue, two simple, hypothetical flow paths of similar lengths were chosen. Path A consisted of 14 143 vertically connected cells (including the outlet cell), having a total center-to-center length of 14 142 units. Path B consisted of 10 001 diagonally connected cells (also including the outlet cell), thus containing 10000 diagonal center-to-center distances and having a total length of $10000 \times \sqrt{2} \approx 14142.136$ units. Although the difference between these estimates is relatively small, the values indicate that path B is slightly longer and should be considered as the expected, correct result of an algorithm searching for the longest flow path.

Fig. 3 illustrates the results obtained by following the identification method from Smith (1997) under ArcGIS Pro 3.0.2. A flow direction raster with both flow paths (leading to the same outlet cell) was prepared and used as input dataset. It can be seen that although path A was measured as might be expected, the length of path B was underestimated. Due to the accumulation of round-off errors, the value obtained in the flow-length raster is lower than the correct sum of the distances between successive cells. As a result, path A was incorrectly selected as the longer one. Although the discrepancy between the calculated length and the expected, mathematically correct value is not large, the accumulated error is sufficient to reach the incorrect conclusion.

In this work, an alternative approach is proposed. Instead of a floating point value, a pair of integers can be used to precisely express the path length, avoiding the accumulation of round-off errors entirely. The main idea is to count and store the numbers of orthogonal and diagonal

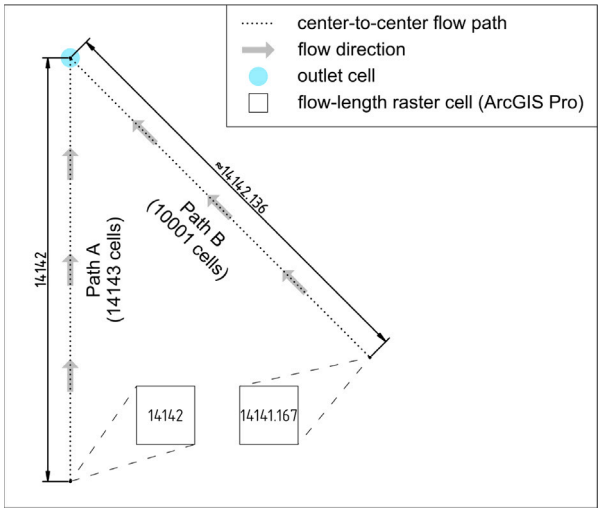


Fig. 3. Inaccurate path length obtained under ArcGIS Pro 3.0.2 (flow-length raster method).

steps between cells separately. Although eight distinct directions are considered in single-flow rasters, only these two kinds of distance are relevant for path length measurements in square-grid data.

The length of a flow path can still be measured by traversing it step by step, but instead of accumulating the distances between successive cells, one of the two counters is incremented with each step. Only when the path length needs to be expressed as a single number, the values are converted by multiplying the counter of diagonal steps by the square root of 2 and adding the number of orthogonal steps to it. This method effectively minimizes the use of floating point calculations by relying primarily on error-free integer incrementations. Fig. 4 illustrates the concept.

It should be noted that this approach is mathematically equivalent to the method of accumulating distances between successive cells. However, the key difference lies in how the calculations are organized, and therefore how they are performed by the machine. In practical implementation, the proposed approach avoids the accumulation of round-off errors and leads to more precise results. The tests carried out in the early stages of this work confirmed that this method correctly solves the case presented in Fig. 3.

It should be emphasized that the concepts presented in this section are not directly related to algorithmic performance, but to the accuracy of path length measurements. All algorithms developed and presented in this work are based on the proposed approach.

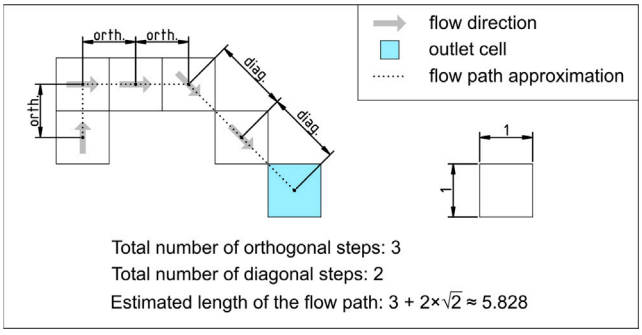


Fig. 4. Estimating the length of a flow path by counting orthogonal and diagonal steps separately.

2.4. Design and implementation

All algorithms developed as part of this work were implemented in C++. Code sections intended to be executed by multiple threads simultaneously were parallelized using OpenMP directives.

The data structure developed in Kotyra et al. (2021) was used to store the rasters in memory. The cells are stored in a two-dimensional array with an additional (external) single-cell frame of neutral values. Due to the large number of GIS algorithms accessing the values of the nearest neighboring cells, this design often makes it possible to simplify the code and increase its efficiency.

The aim during the design and implementation phase was to achieve the shortest possible execution times while maintaining the maximum possible precision of the results. It was assumed that the task of identifying the longest flow paths is solvable in linear time. Thus, only algorithms with linear time complexity were considered in this work.

3. Developed algorithms

3.1. Reference implementations

As the first step in the development and evaluation of new algorithms, the basic recursive approach was adapted and expanded upon. Two relatively straightforward implementations were prepared, one sequential and the other attempting to parallelize computations using OpenMP tasks. Although both of them were treated mainly as reference solutions, they were also the starting point for the development of entirely new algorithms, and therefore they are presented here first.

Only solving the fundamental version of the problem (with a single outlet point and identifying any of the longest paths) was required of the reference implementations. It is worth noting that the developed adaptations of the recursive approach still follow the requirements and incorporate the concepts introduced in this work (e.g. their methods of comparing path lengths are based on counting orthogonal and diagonal steps separately).

3.1.1. Recursive approach (sequential)

The underlying idea of the recursive approach is to search through all flow paths within a catchment by starting at the outlet cell and "climbing" upstream (contrary to the flow direction). The recursive procedure calls itself for successive higher cells, effectively traversing the entire catchment area in a depth-first manner. Fig. 5 illustrates the general concept.

The algorithm starts the search by calling the recursive procedure for the designated outlet cell. The procedure uses the flow direction data to identify the inflow neighbors of the cell, and recursively invokes itself for each of them. The same process is repeated for successive

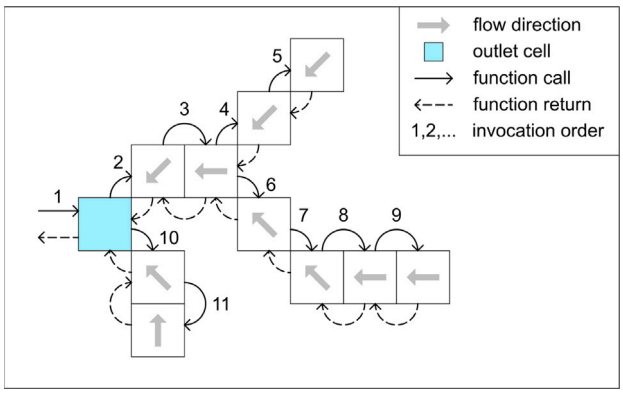


Fig. 5. Recursive procedure traversing successive upstream cells in a depth-first manner.

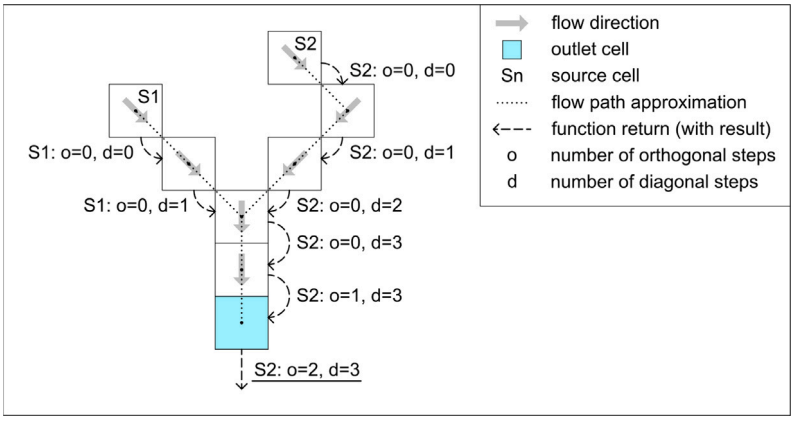


Fig. 6. Recursive procedure identifying the longest flow path leading to the outlet cell.

cells, effectively climbing up from the outlet towards topographically higher locations. The earlier, pending calls (for lower cells) wait for the subsequent invocations to complete and return their results.

Conceptually, the aim of the recursive procedure, invoked for any given cell, is to identify the longest flow path leading to that particular location. Thus, the initial call (for the outlet cell) carries out the task of finding the longest flow path for the entire catchment. The procedure accomplishes this by delegating subtasks (partial searches covering smaller areas) to its successive, recursive calls. Similarly, each partial search is broken down into even smaller subtasks, which are then delegated to subsequent invocations. The results obtained and returned for these partial searches are captured and used by earlier, pending calls.

The result produced for any given cell contains the length of the longest flow path leading to this particular point, along with the location of the source cell where that path begins. Since the source cells have no inflow neighbors (and are thus the starting points of the flow paths), the result returned for any of them contains a zero path length and the location of the cell itself. For all other types of cells, the procedure captures the results returned by its subsequent invocations, increments the appropriate step counter in each of them, selects the one with the longest path length, and returns it as its own result. Ultimately, the initial call (for the outlet cell) returns a catchment-wide search result containing the length and starting location of the longest flow path leading to the outlet. Fig. 6 illustrates the process.

The algorithm was implemented with the intention of minimizing its execution time. The implementation has a linear time complexity, and the time needed to perform the search depends directly on the number of cells belonging to the catchment (the recursive climbing procedure does not cross the watershed boundary, thus ignoring any cells beyond it). It is worth noting that this solution does not require raster storage for partial results during calculations.

Multiple well-known problems are related to recursive implementations in general. Although this approach generally leads to relatively simple and efficient code, it is also inherently related to memory issues (i.e. stack overflows), often making it impractical for larger datasets (Wallis et al., 2009). Since this approach was implemented mainly as a reference solution, these issues are not discussed in detail here.

3.1.2. Recursive approach (parallel)

Recursive implementations frequently prove to be difficult, counter-intuitive, or even impossible to efficiently parallelize (Qin and Zhan, 2012; Stpiczyński, 2018). As an attempt to introduce parallelism to the presented recursive algorithm, an implementation based on OpenMP tasks was developed.

Assuming a single outlet point, only one thread starts the recursive procedure at the beginning. However, in this implementation, OpenMP tasks are created for each identified inflow neighbor, allowing other threads to join the processing and start performing partial computations. This makes it possible for separate flow direction branches to be searched through in parallel.

Unfortunately, this approach introduces the additional overhead of creating and managing a large number of tasks. The implementation prepared as part of this work aims to limit this cost by allowing the creation of new tasks only to a certain level of recursion. Above the limit, a thread stops generating tasks and performs the rest of the computations in the branch on its own. This idea optimistically assumes that it may be possible to reasonably distribute the work between threads at lower levels of recursion (close to the outlet point). However, given the realistic underlying landform, this may prove to be rare or even impossible. Considering that a large fraction of flow direction branches may contain only one or a few cells, the benefits of parallelization may turn out to be moderate and easily outweighed by the overhead.

3.2. New algorithms

During the design and implementation phase, two distinct, general concepts for the new algorithms were developed. In the following sections, they are referred to as “top-down” and “double drop” approaches. Based on these two fundamental ideas, a total of five new algorithms were implemented, including three sequential and two parallel versions.

3.2.1. Top-down: maximum length (sequential)

The fundamental concept underlying the approach referred to here as “top-down” is to traverse the raster along the flow direction, starting from source cells (“tops”, the most upstream locations) and moving downstream. In a way, this is the opposite of the recursive approach. In a typical flow direction raster, a large number of source cells are the starting points of flow paths that gradually merge and eventually end at the same outlet point. Approaching the problem from this perspective creates new possibilities for parallelization.

Conceptually, the algorithm can start from any source cell in the raster. The flow path is traversed cell by cell along the flow direction, while its length is measured by counting successive orthogonal and diagonal steps. A naive implementation of this idea would be to fully traverse and measure each existing flow path separately (sequentially starting at each source cell in the raster and moving downstream until the endpoint is reached) and compare their lengths (e.g. by recording the longest one found so far). However, this would not be a well-performing approach, since a large fraction of the cells in the raster would be unnecessarily traversed multiple times (common parts of converging flow paths would be needlessly re-measured for each of them). A much more efficient solution would be to compare the partial

flow paths as soon as they merge in junction cells and only proceed downstream with the longest one, discarding the other candidates (since it is already clear that they cannot be the longest path in the catchment).

Incorporating this concept, the algorithm starts traversing and measuring a flow path from its source cell, and moves downstream until it encounters a junction cell. Here, the traversing procedure is temporarily stopped, and the algorithm starts another, separate measurement from a different source cell. Only after measuring all partial flow paths leading to a given junction cell, the algorithm proceeds downstream to further traverse and measure the longest of them. With this approach, each cell in the raster can only be traversed downstream once (a junction cell cannot be passed until the longest upstream path leading to it is identified). This property is directly related to the linear time complexity of the algorithm.

The first implementation of this approach (called "top-down max" for short) uses a specific data structure to compute and store partial results, namely a raster of two-integer cells (each cell containing two numbers). A single cell can be used to store either a distance (as separate numbers of orthogonal and diagonal steps) or a location of another cell in the raster (as row and column coordinates). This concept was implemented as a union of two structures (representing distance and location data separately), allowing for two possible interpretations of the same cell in the code. Ultimately, every source cell is used to store distance, while every non-source cell stores location.

Initially, all cells of the working raster are set to a special UNDETERMINED state (being marked as not containing any valid value yet). The source cells are then used as working memory to calculate, store and compare the partial lengths of the analyzed flow paths. The main part of the algorithm aims to fill the raster so that every other (non-source) cell points to the source location where the longest path leading to that particular cell begins. This then makes it possible to immediately locate the beginning of the longest flow path leading to any given point (most importantly, to the outlet cells specified in the input data). Extracting the source location for any completed cell can be done by checking the cell type (as source cells are a special case, being their own source locations) and reading the coordinates stored in it.

Fig. 7 shows a conceptual example of how the search for the longest flow path is performed. The algorithm identifies the source cells and starts the downstream traversing procedure for each of them. The integers stored in each source cell are treated as orthogonal and diagonal step counters, first set to zero, and then incremented accordingly with each step along the path. At the same time, each traversed non-source cell is updated with the coordinates of a source location that begins the longest currently known path to this particular cell. Eventually, when this process is complete, the outlet cell contains the final search result, pointing to the source location of the longest flow path in the entire catchment.

For each non-source cell encountered, a path-comparison operation is performed to select the longer of the two partial paths leading to that point — the one currently being traversed and the one starting from a location previously stored in the cell (if the cell still contains the UNDETERMINED value, the current path is immediately selected). The lengths of both partial paths (distances traversed so far — from their starting points to the junction) are extracted from their source cells and compared, and the junction cell is updated with the new source location if necessary (if the path currently being traversed turns out to be shorter than the one already found for that cell, the current path is discarded and the cell does not change its state). Thus, comparisons of partial paths effectively take place in junction cells, and only the selected, longer path is further considered. The lengths of the discarded paths (stored in their source cells) are no longer incremented. Link cells (having a single inflow neighbor) are only entered and updated once, changing their state from UNDETERMINED to valid source coordinates.

It is necessary to establish the final value of the junction cell (and thus identify the longest path leading to it) before proceeding further to its downstream cells. As noted earlier, this requires traversing all partial paths leading to the junction cell and comparing their lengths (using the path-comparison operation in the junction). For this reason, the traversal procedure has to be temporarily stopped in cells for which the lengths of all inflow paths are not yet known. Only when the final value of the cell is determined, the data needed to continue the downstream traverse is available. This property was implemented using an inlet number matrix, where each cell is first initialized with the number of its inflow neighbors (except for the source cells, which receive a predefined value for easy identification). Each time the traversing procedure reaches a given cell, its inlet number in the matrix is decremented. Only when this value reaches zero (meaning that all paths leading to this point are completely processed), the algorithm is allowed to proceed

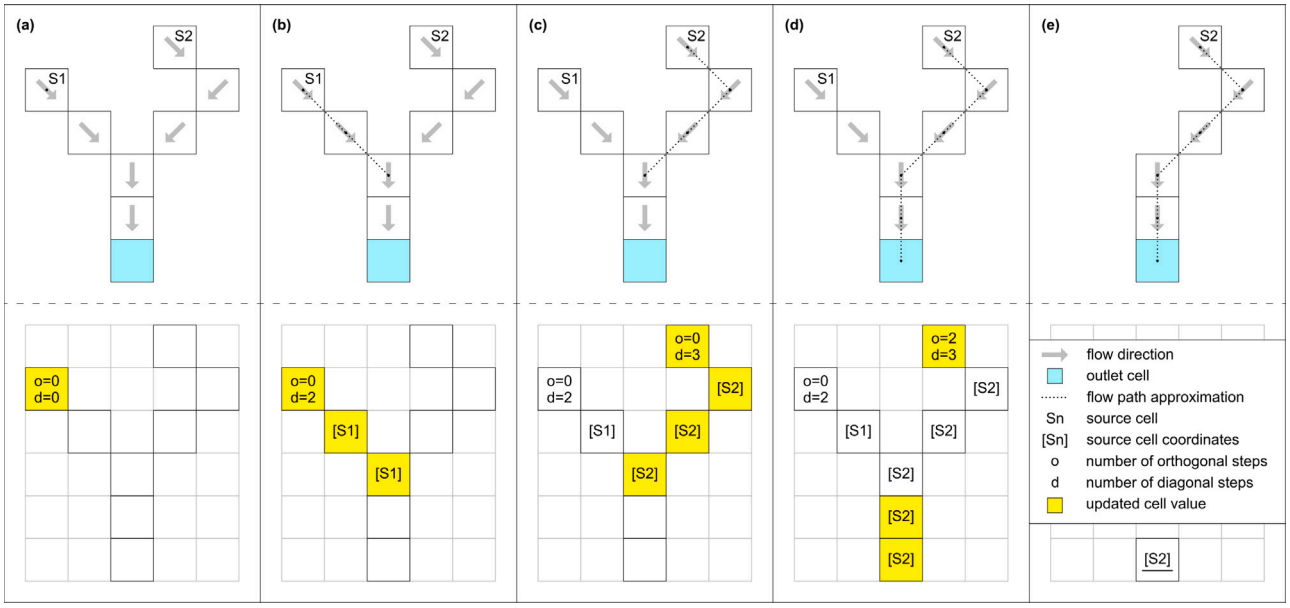


Fig. 7. Top-down: maximum length algorithm — conceptual example: (a) identifying the first source cell; (b) measuring the first partial flow path (stopping at the junction cell); (c) measuring the second partial flow path and selecting the longer candidate (updating the junction cell); (d) measuring the further section of the selected candidate path; (e) identifying the longest flow path leading to the outlet cell (by extracting the coordinates stored in it).

to the subsequent, downstream cells. This technique of temporarily stopping computations in junction cells was previously described for flow accumulation algorithms (Zhou et al., 2019; Kotyra et al., 2021). The processing order of the partial flow paths is irrelevant to the final result.

Although this algorithm was developed with parallelization in mind, ultimately only a sequential version was implemented. The non-atomic nature of the path-comparison operation would require the heavy use of synchronization mechanisms to prevent data races between threads, resulting in impractical overhead levels. To address this issue, another version of the top-down approach was implemented with the intention of eliminating potential conflicts between threads through a different workflow.

3.2.2. Top-down: single update (sequential)

The next algorithm (called "top-down single" for short) is based on the top-down max implementation, but with some important modifications. The key idea was to reorganize the workflow so that the value of each non-source cell in the raster is determined and set only once. Fig. 8 presents a conceptual example.

As in the top-down max implementation, the algorithm aims to fill each non-source cell with the coordinates of the source point where the longest flow path leading to that particular cell begins. However, instead of storing and repeatedly updating the partial result (the source location of the longest path to that point identified so far) in the junction cell, the algorithm leaves it uninitialized until the lengths of all its inflow paths are determined. Only when the values of all inflow neighbors are established, the algorithm compares all paths leading to that point and determines the cell value. This implementation also uses the inlet number matrix, but here the decrementation to zero is required to enter and update the cell, rather than to proceed downstream from it. Algorithm 1 presents a simplified pseudocode covering this concept.

It is worth noting that only junction cells are processed and updated in this specific way, based on reaching out to their inflow neighbors once all their values are determined. Single-inflow cells are processed in a straightforward manner and simply filled with the starting point of the path that is currently being advanced.

An important advantage of this design is that it no longer requires raster cells to be initialized. As each cell is updated only once, its previous state is irrelevant and does not affect the final result. The algorithm has a linear time complexity (each cell is traversed downstream only once).

3.2.3. Top-down: single update (parallel)

The idea of a single cell update eliminates problematic conflicts between threads and makes it possible to parallelize this algorithm in a relatively simple manner.

In this implementation, multiple threads traverse partial flow paths simultaneously, starting from different source cells. Junction cells are still locations of potential conflicts, but the single update workflow reduces the issue significantly. In fact, the only synchronization mechanism required is the atomicity of the decrementation in the inlet number matrix (with direct capture of the result). Since different threads may reach the same junction cell and simultaneously attempt to decrement its number of unprocessed inflow paths, it has to be ensured that this will not result in data races.

The last thread that decreases the value of the junction cell in the inlet number matrix (reaching zero) compares all paths leading to the cell and selects the longest one. At this point, it is guaranteed that all inflow neighbors contain correct values. The thread updates the junction cell and proceeds to traverse its successive downstream cells.

The inlet number counters (and their atomic decrementation) are only needed for junction cells. All other cells are traversed only once, therefore there is no possibility of any conflict between threads. As synchronization is not needed there, it is not used, and, as a result, non-junction cells are updated in an unsynchronized, straightforward manner.

In all the top-down implementations developed in this work, the final result is obtained by extracting the coordinates stored in the outlet cell specified in the input data. Each non-source cell in the raster indicates where the longest path leading to this cell begins. These coordinates are treated as the end result of the algorithm. The only special (though impractical) case is a scenario where the specified outlet point is actually a source cell (which does not store location but distance). To ensure the correctness of the algorithm for each case, this scenario has to be recognized and handled properly.

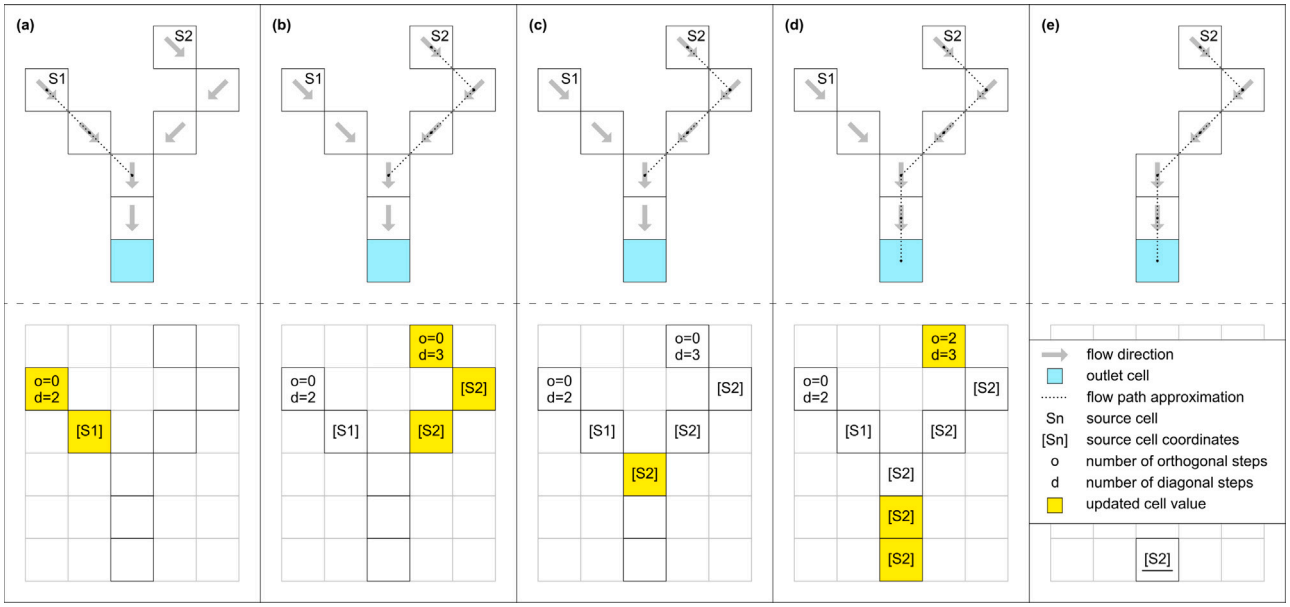


Fig. 8. Top-down: single update algorithm — conceptual example: (a) measuring the first partial flow path (without modifying the junction cell); (b) measuring the second partial flow path; (c) selecting the longer candidate path and setting the value of the junction cell; (d) measuring the further section of the selected candidate path; (e) identifying the longest flow path leading to the outlet cell (by extracting the coordinates stored in it).

```
input : flowDirectionRaster, outletLocation
output: source location of the longest flow path leading to the outletLocation
calculate inletNumberMatrix, mark SOURCE and LINK cells with special values;
allocate pathMatrix without initializing it;
for each location start in the raster do
    if the start cell is a SOURCE and has a valid flow direction then
        pathSource = start;
        pathMatrix[pathSource] = (0, 0);
        incompleteInflows = 0;
        current = start;
        do
            move current location to the next cell (by flow direction);
            update pathMatrix[pathSource] by incrementing the appropriate step counter;
            if the current cell is a LINK then
                pathMatrix[current] = pathSource;
            else
                incompleteInflows = −−inletNumberMatrix[current];
                if incompleteInflows == 0 then
                    maxLength = pathMatrix[pathSource];
                    for each inflow neighbor of the current cell do
                        neighborSource = extractSourceLocation(neighbor);
                        if pathMatrix[neighborSource] > maxLength then
                            maxLength = pathMatrix[neighborSource];
                            pathSource = neighborSource;
                        end
                    end
                    pathMatrix[current] = pathSource;
                end
            end
        while incompleteInflows == 0 and the current cell has a valid flow direction;
    end
end
return extractSourceLocation(outletLocation);
```

Algorithm 1. Top-down: single update — sequential version (simplified pseudocode).

It should be emphasized that all the top-down implementations have a natural ability to determine the longest flow paths for multiple outlet points in the same run. As the main part of the algorithm fills all non-source cells with the correct source coordinates, determining the longest flow paths for other outlet points only requires extracting these additional starting locations. Thus, determining the longest paths for multiple outlet points in the same run involves only a small (negligible in practice) additional cost. Each of these algorithms has been implemented in a way that allows both single and multiple outlet points to be taken as input and processed.

3.2.4. Double drop (sequential)

Another approach to the problem, developed as part of this research, is referred to in this paper as the "double drop". Although it is also based on traversing flow paths along the flow direction, the concept differs significantly from the top-down approach.

The main idea behind the algorithm is relatively straightforward. The most characteristic property is that it passes the same sections of the flow paths twice. Conceptually, the traversing procedure can be started from any cell (it does not have to be a source cell). The algorithm first moves downstream (along the flow direction), evaluating whether a given flow path leads to the specified outlet point and measuring the distance traveled from the starting cell. Then the same path is traversed again, this time recording the remaining distance to the outlet in each passed cell (or marking it as not leading to the outlet point).

The algorithm also uses a working raster of two-integer cells, but here the content of each cell is always interpreted as a distance (expressed as separate numbers of orthogonal and diagonal steps). The main part of the algorithm aims to fill the raster so that each cell contains the path length from itself to the watershed outlet specified in the input (or the predefined OUT_OF_BASIN value if the path does not lead to the outlet). Fig. 9 shows a conceptual example. Algorithm 2 presents a simplified pseudocode of this approach.

As the first step, the working raster is initialized with the UNDETERMINED values (the external frame is filled with OUT_OF_BASIN values for easy handling of boundary crossings). Only the outlet cell is set to a valid distance value of zero (being the correct distance to itself). Next, the algorithm starts traversing and measuring subsequent flow paths, gradually filling the raster with the calculated distances to the outlet point.

As mentioned, the most distinctive property of this approach is that each partial flow path is traversed twice. On the first pass of a given path, no modifications are made to the raster, only the distance traveled from the starting cell is measured using the orthogonal and diagonal step counters (by incrementing them accordingly with each step). As soon as the traverse procedure enters a cell with any value other than UNDETERMINED or crosses the raster boundary, the second pass begins, starting again from the same location and following the

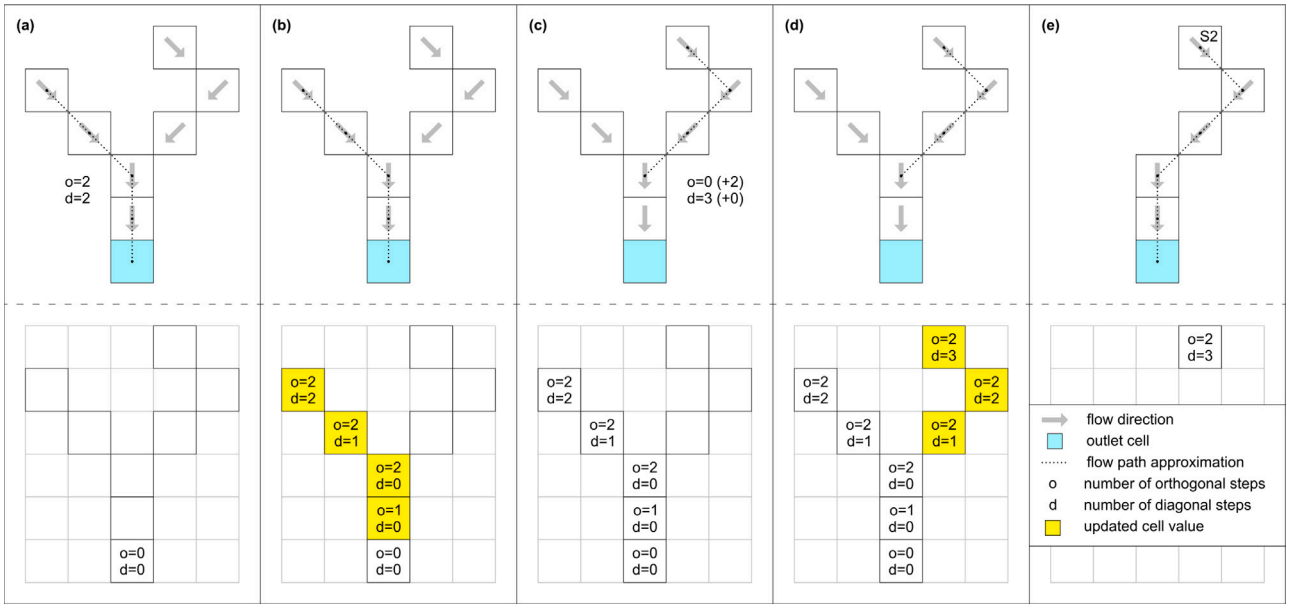


Fig. 9. Double drop algorithm — conceptual example: (a) measuring the first flow path (first pass); (b) recording the remaining distances to the outlet cell (second pass); (c) measuring another (partial) flow path (first pass); (d) recording the calculated distances to the outlet cell (second pass); (e) identifying the longest flow path leading to the outlet cell (by selecting the longest recorded distance).

same path. This time, each cell passed through is modified (and thus set to its final value).

If the first pass ends with crossing the raster boundary or entering a cell with the OUT_OF_BASIN value, the second pass sets all cells along the entire path to OUT_OF_BASIN (a special case of reaching a cell without a valid flow direction value is handled in the same way). Any cell marked with this value is already known not to lead to the outlet point specified in the input. Entering such a cell while traversing downstream means that the entire path is located outside the catchment.

Cells containing a valid distance are known to lead to the outlet point (and their value describes the remaining length of that path). When the algorithm enters such a cell during the first pass, the distance traversed so far (downstream from the starting point) is summed up with the value stored in that cell. In this way, the total distance from the starting point to the outlet cell is obtained. The second pass fills all cells along the path with the remaining distance to the outlet point. This time, the orthogonal and diagonal step counters (expressing the remaining distance) are appropriately decremented with each step.

The algorithm records the length and starting location of the longest path identified so far. Comparisons between potential candidates are made when a cell with a valid distance is entered during the first pass, and the total distance from the starting point to the outlet is calculated. Alternatively, a simple one-time raster scan to find the maximum length could be implemented at the end of the algorithm, but this would require significantly more comparisons.

As with the top-down approach, the order in which the partial flow paths are traversed is irrelevant. However, the double drop algorithm does not require the traversing procedure to be started from a source cell (in fact, it can be started from any unprocessed cell, regardless of its type). This makes the detection of source cells unnecessary, significantly simplifying the workflow. Once all cells are processed, the algorithm returns the starting location of the longest identified flow path as the final result.

It should be emphasized that although this algorithm traverses each location twice, it treats all types of cells in the same way and does not require detection or special handling of either source or junction cells. This simple workflow could be considered a significant advantage of this approach. The implementation has a linear time complexity.

3.2.5. *Double drop (parallel)*

The parallel implementation of the double drop algorithm allows multiple threads to process partial flow paths simultaneously. The fundamental idea remains the same as in the sequential version, although some minor changes were introduced.

As long as there are unprocessed cells in the raster, the threads select them as starting points and proceed to the downstream traversing procedure. The same raster is being filled in multiple locations simultaneously, allowing the threads to share their results with each other.

Each thread individually stores the length and starting location of the longest path it has identified so far. These local results are compared with the global maximum only at the end of the algorithm run. This approach minimizes the need for synchronization between threads comparing large numbers of paths.

It is possible for multiple threads, starting from different locations, to traverse the same common part of their flow paths in the first pass simultaneously. The design of the algorithm allows for such a scenario, considering the cost of possible synchronization mechanisms as inviable. Apart from potential additional, partially redundant work, this property has no negative effects on the operation of the algorithm. In the second pass, such cases are generally eliminated early, as the thread stops the filling procedure as soon as it reaches the first cell with a value other than UNDETERMINED.

It is worth noting that the double drop approach allows for straightforward determination of all alternative longest flow paths leading to the same outlet point. As the length of the longest path is known at the end of the algorithm, it is possible to simply search the entire raster for all cells containing a distance equal to the maximum length. In this way, the algorithm can return not just one, but all alternative longest paths with little additional cost. Both implementations of this approach, developed as part of this work, include this mode of operation.

Although this algorithm was designed to work with a single outlet cell, it is possible to extend these implementations to handle multiple outlets in the same run. This would require computing and storing additional data for each cell (perhaps a single integer index), indicating to which outlet point its flow path leads. Instead of being marked with OUT_OF_BASIN, cells flowing to other outlet points would store valid distances and still be easily identifiable. While requiring more memory, it would likely reduce the time needed for individual algorithm runs when working with multiple outlets.

```
input : flowDirectionRaster, outletLocation
output: source location of the longest flow path leading to the outletLocation
initialize lengthMatrix: all cells to UNDETERMINED, external frame to OUT_OF_BASIN;
lengthMatrix[outletLocation] = (0, 0);
longestPathSource = outletLocation;
longestPathLength = (0, 0);
for each location start in the raster do
    if the start cell is UNDETERMINED and has a valid flow direction then
        pathLength = (0, 0);
        current = start;
        do
            move current location to the next cell (by flow direction);
            update pathLength by incrementing the appropriate step counter;
        while the current cell is UNDETERMINED and has a valid flow direction;
        if the reached current cell contains a valid distance then
            pathLength += lengthMatrix[current];
            if pathLength > longestPathLength then
                longestPathSource = start;
                longestPathLength = pathLength;
            end
            current = start;
            do
                lengthMatrix[current] = pathLength;
                move current location to the next cell (by flow direction);
                update pathLength by decrementing the appropriate step counter;
            while the current cell is UNDETERMINED;
        else
            lengthMatrix[current] = OUT_OF_BASIN;
            current = start;
            do
                lengthMatrix[current] = OUT_OF_BASIN;
                move current location to the next cell (by flow direction);
            while the current cell is UNDETERMINED;
        end
    end
end
return longestPathSource;
```

Algorithm 2. Double drop — sequential version (simplified pseudocode).

4. Performance measurements

4.1. Data

The Bystrzyca catchment was selected as the first source area for measurements and analyses. It is a third-order watershed with a total area of 94 km^2, located in the south-eastern part of Poland. Square-grid DEM data with one-meter resolution was obtained from the publicly available resources of the Head Office of Geodesy and Cartography (GUGiK). The source dataset was referenced in the PL-1992 system (National Geodetic Coordinate System 1992 for Poland).

The original data was processed to generate a wide variety of input datasets for performance measurements. In the first stage, the outlet point was determined and the entire catchment, along with the longest flow path belonging to it, was delineated. Subsequently, the division into sub-catchments was carried out, adopting two different approaches. As for the first one, the main catchment was divided along the longest flow path every two kilometers, starting from the watershed boundary. This resulted in 47 sub-catchments with varying areas. The same procedure was carried out on the source data scaled down to 2 × 2 m, 4 × 4 m, 6 × 6 m, 8 × 8 m and 10 × 10 m resolutions. The generated datasets were intended mainly for comparing the execution times of various algorithms and tools, as well as examining the relationship between their performance and the size of the input data.

As for the second approach, the ten-meter resolution data was used. Based on the drainage network, a set of sub-catchments was delineated, with 3.0 km^2 chosen as a surface threshold. In this way, 118 relatively small sub-catchments were generated, with areas ranging from 3.0 to 3.4 km^2. These datasets were mainly intended for controlling and analyzing the results of the developed algorithms, as well as comparing them with those generated by selected GIS software.

Additional source areas were selected in order to obtain larger, more computationally demanding datasets. Four other watersheds located in Poland were chosen (the Kamienna, Tanew, Barycz and Wieprz) with areas ranging from approximately 1,300 to 4,440 km^2. The acquired DEMs (with one-meter resolution, originating from the GUGiK) were used to generate datasets ranging in size from approximately two to seven billion cells (Table 1 shows the details). These datasets were intended for examining the performance and characteristics of the developed algorithms, as well as comparing them with each other.

Each flow direction raster used in performance measurements was prepared in two variants. In the first kind (referred to as “full frames”),

Table 1
Dimensions of the largest datasets used for algorithm comparisons.

Dataset	Area in km^2	Raster dimensions	Raster cells
Kamienna	1300.37	29,954 × 64,080	1,919,452,320
Tanew	1411.16	45,553 × 47,023	2,142,038,719
Wieprz	4026.48	84,463 × 71,540	6,042,483,020
Barycz	4440.21	104,326 × 66,747	6,963,447,522

each cell contained a valid direction value. In the second one (simply referred to as "basins"), all cells outside the catchment boundaries had a direction value set to NONE (only cells belonging to the catchment contained a valid value). Previous research by the authors has shown that using data filtered in this way can have a noticeable impact on the computational times of some algorithms (Kotyra et al., 2021).

Fig. 10 shows both the selected source areas and sub-catchments obtained with the methods described. All datasets were precisely clipped to the selected catchments (each row and column of every raster contained at least one cell belonging to the catchment).

4.2. Testing procedure for C++ algorithms

Before running any measurements, all implementations were validated against a suite of automated tests. The algorithms were tested for multiple scenarios, including both standard and special cases, as well as a variety of runtime configurations. All files with input data intended for measurements were also verified for correctness.

Performance measurements were based on repeatedly executing each algorithm on multiple datasets while measuring and recording computational times. Each test, performed by an automated bash script, consisted of restarting the measurement application, loading input data

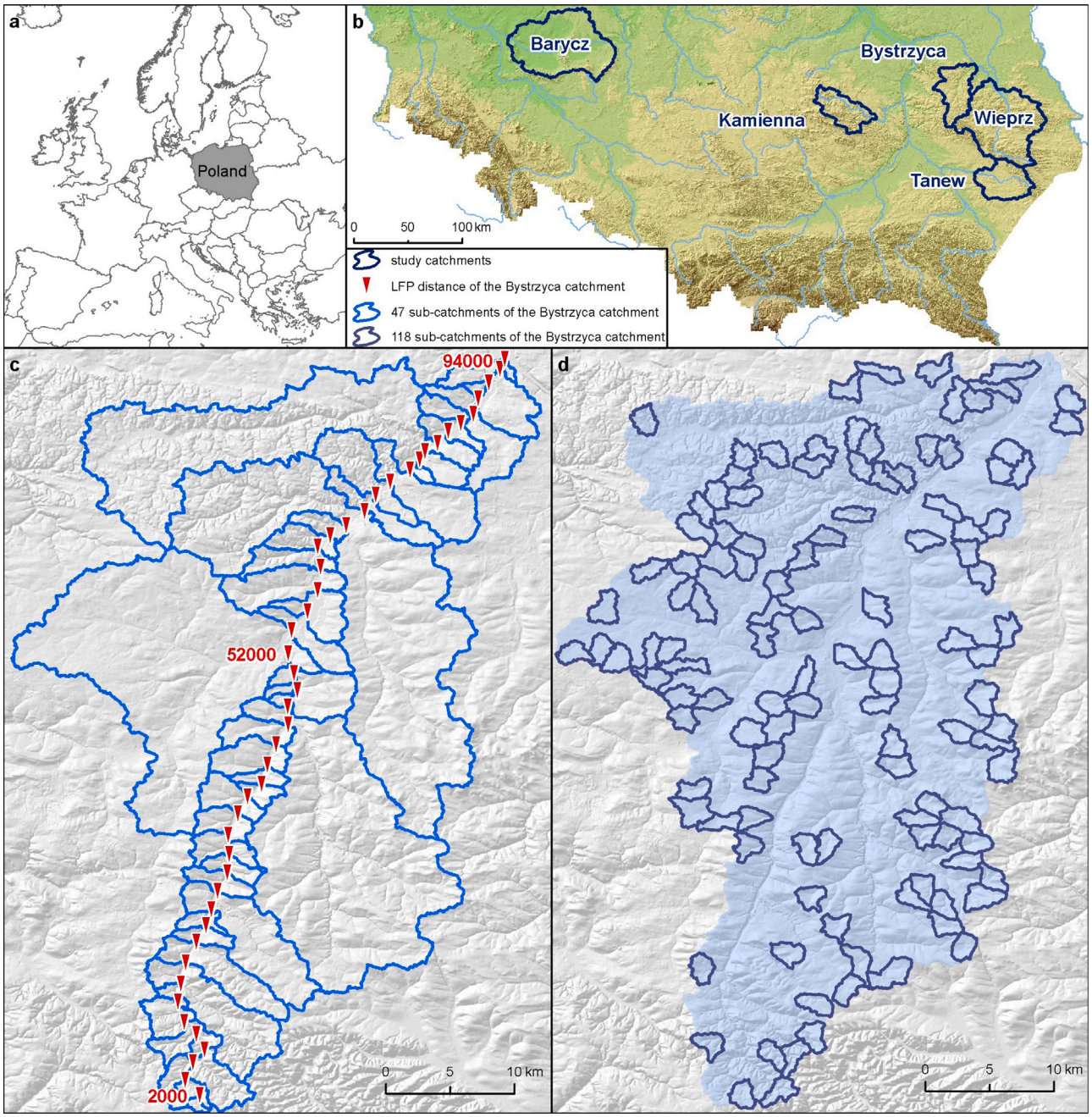


Fig. 10. Study area with catchment locations.

files, executing the selected algorithm and verifying the correctness of its output. Time measurements were started immediately after the input dataset was loaded into memory and stopped after the algorithm generated the final result.

As the recursive implementations were included in the tests, stack overflows were expected. In such cases, the thread stack size was doubled in the runtime configuration (more than once if needed), and the test was repeated.

Measurements were performed in two separate test environments. Machine A, running under AlmaLinux 8.4, was equipped with a dual Intel Xeon E5-2670 v3 processor (24 cores in total) and 128 GB RAM. Machine B, with two operating systems (Windows 10 Enterprise LTSC 64-bit and Ubuntu 22.04.1 LTS), was equipped with a dual Intel Xeon CPU E5-2620 v4 processor (16 cores in total) and 112 GB RAM. On machine B, algorithm performance measurements were carried out under Ubuntu.

All source code was compiled using the GNU C++ compiler (version 8.4.1 on machine A, version 11.2.0 on machine B) with O3 level optimization and OpenMP support enabled. Parallel implementations were allowed to use all available cores.

In order to precisely compare the performance of the developed algorithms, each of them was executed and measured 30 times on each of the eight large datasets (Kamienna, Tanew, Barycz and Wieprz; both full frame and basin variants). For a fair comparison, only the fundamental variant of the problem (assuming one outlet point and requiring a single path to be identified) was taken into account. This procedure was performed on both machines.

Two parallel implementations of the newly presented algorithms were also tested in a similar manner on a set of 564 sub-catchments of the Bystrzyca (47 areas, each in six resolutions and two variants). These measurements provided the basis for a more detailed analysis, including comparisons with other existing software.

In order to examine the efficiency of the two new parallel algorithms in various multithreading configurations, additional measurements were performed on machine A. Using the largest frame dataset (Barycz), the algorithms were tested with thread limits ranging from 1 to 48. Performance was measured 30 times for each configuration.

The task-based recursive implementation requires a parameter that specifies the highest level of recursion at which new tasks can still be created. Based on the foundations of this concept, a relatively low limit of 100 was selected and then doubled several times, reaching 200, 400, 800 and 1600. Separate tests were performed for each of these values. It was assumed that this variety should allow for an overall evaluation of this approach.

4.3. Testing procedure for existing GIS software

To obtain reference computational times for the algorithms developed in this work, the performance of the r.accumulate tool was measured. It was selected as the most recent module for determining the longest flow paths available on the GRASS GIS platform. The measurements presented in Cho (2020) showed significant advantages of this tool over existing alternatives.

Tests were performed using GRASS GIS version 8.0.2. Unfortunately, due to the limitations inherent in this platform, larger datasets (greater than approximately 2 billion cells) could not be used. Measurements were carried out on machine B, under both Windows 10 and Ubuntu. The r.accumulate tool was executed 30 times on each of the selected datasets, under both operating systems. An automated script monitored the calculation runs and retrieved the times needed to generate the results.

5. Results and discussion

5.1. Performance comparison of the developed algorithms

Tables 2 and 3 present the average execution times of the developed algorithms, measured using the eight large datasets on both machines. The two approaches to data preparation were denoted by FF for full frames and B for basins.

The parallel implementation of the double drop approach turned out to be the most efficient in the vast majority of cases (Table 3). However, while the parallel version of the top-down single algorithm performed slightly worse on average, it turned out to be the fastest in some instances. The performance differences between these two implementations are more apparent for the measurements made on machine B (in favor of the double drop algorithm). It appears that both the data characteristics (basins versus full frames) and the underlying hardware (particularly the different number of CPU cores) contribute to the relative performance of these algorithms.

Among the sequential implementations, the recursive approach turned out to achieve the shortest execution times in all cases, on both machines (Table 2). The sequential double drop algorithm achieved times ranging from approximately 54% to 126% longer, and the sequential top-down implementations from 151% to 291% longer than the recursive algorithm. As both the double drop and top-down approaches are relatively more complex and require processing of all cells in the raster (as opposed to the recursive algorithm, which only visits cells belonging to the catchment), these results are not surprising.

However, the results turned out to be completely different for parallel implementations. While the newly developed algorithms achieved significant speedups over their respective sequential versions, the task-based implementation did not improve the performance of the recursive approach. The execution times of the parallel recursive implementation were on average several percent longer than those achieved by the sequential version. This observation holds for all tested task creation limits. Moreover, while the differences in computational times between the sequential and parallel implementations are relatively small, there seems to be a strong positive correlation between the task creation limit and the average time achieved by the task-based version. This may suggest that increasing the limit further will only degrade the performance more.

The parallel double drop implementation achieved an average speedup of 11.2 on machine A and 8.5 on machine B, across all the datasets. For the top-down single algorithm, the average speedup was significantly higher: 18.3 on machine A and 11.7 on machine B. It is worth emphasizing that the sequential implementation of the double drop approach achieved on average 41% shorter times than the sequential top-down single algorithm. While the top-down single approach achieves higher speedups, the double drop algorithm seems to excel when the number of threads is lower. This could help explain why, while the double drop algorithm achieved the shortest execution times in the vast majority of cases, the top-down approach turned out to be slightly faster on a few occasions.

It is worth noting the differences in computational times achieved on both types of datasets. For the recursive approach (both sequential and parallel versions), there is no particular difference between processing full frames and basins from the same source area. However, for all the other implementations, the differences are significant. Depending on the algorithm, calculations on the basin variant of the data (with cells outside the catchment boundary set to NONE) turned out to be approximately 20% to 31% faster on average.

As expected, the recursive implementations (both sequential and parallel) repeatedly caused stack overflows and consequently application crashes. In some cases, it was necessary to double the stack size several times before the test could be successfully completed. This can be considered as a confirmation that the recursive approach is in fact impractical, especially for larger datasets.

Table 2
Average execution times (in seconds) on the largest datasets: sequential implementations.

Machine A (single thread):				
Dataset	Recursive (sequential)	Top-down max (sequential)	Top-down single (sequential)	Double drop (sequential)
Kamienna FF	25.9	87.1	93.1	54.6
Tanew FF	28.9	98.4	106.3	59.9
Wieprz FF	80.5	280.0	305.0	166.2
Barycz FF	91.1	326.4	356.6	206.0
Kamienna B	25.9	64.9	66.5	40.6
Tanew B	28.8	72.6	74.6	44.3
Wieprz B	80.0	207.1	211.6	125.1
Barycz B	91.3	232.9	238.6	155.0
Machine B (single thread):				
Dataset	Recursive (sequential)	Top-down max (sequential)	Top-down single (sequential)	Double drop (sequential)
Kamienna FF	27.7	90.8	95.1	57.9
Tanew FF	30.6	102.0	108.4	64.1
Wieprz FF	84.9	299.3	326.5	182.0
Barycz FF	96.7	336.9	378.2	208.4
Kamienna B	27.9	68.2	68.7	44.3
Tanew B	30.7	75.3	76.9	48.6
Wieprz B	84.7	221.6	226.7	139.3
Barycz B	96.9	241.5	248.4	153.2

Table 3
Average execution times (in seconds) on the largest datasets: parallel implementations.

Machine A (24 cores, 48 threads):							
Dataset	Recursive (tasks, 100)	Recursive (tasks, 200)	Recursive (tasks, 400)	Recursive (tasks, 800)	Recursive (tasks, 1600)	Top-down single (parallel)	Double drop (parallel)
Kamienna FF	25.9	26.0	26.2	26.1	26.2	5.2	4.6
Tanew FF	29.3	29.1	29.4	29.9	30.2	5.7	5.2
Wieprz FF	81.7	81.3	81.6	81.8	82.1	16.4	14.7
Barycz FF	91.5	92.6	92.1	93.1	93.5	18.8	17.2
Kamienna B	25.9	26.0	26.1	26.2	26.1	3.8	3.7
Tanew B	29.1	29.3	29.3	29.6	30.2	4.0	4.2
Wieprz B	80.8	81.0	80.7	81.2	81.6	11.7	12.2
Barycz B	91.8	92.9	92.9	92.4	94.0	12.9	13.5
Machine B (16 cores, 32 threads):							
Dataset	Recursive (tasks, 100)	Recursive (tasks, 200)	Recursive (tasks, 400)	Recursive (tasks, 800)	Recursive (tasks, 1600)	Top-down single (parallel)	Double drop (parallel)
Kamienna FF	28.1	28.4	28.6	28.7	28.7	8.1	6.3
Tanew FF	31.3	31.2	31.3	31.2	31.6	9.3	7.2
Wieprz FF	85.9	86.8	86.8	87.1	86.5	27.8	21.7
Barycz FF	97.6	98.2	97.4	97.7	99.1	31.1	25.5
Kamienna B	28.2	28.4	28.7	29.0	28.8	5.9	5.0
Tanew B	31.0	31.0	31.2	31.4	31.6	6.6	5.8
Wieprz B	85.9	86.7	85.7	86.4	86.3	19.7	17.4
Barycz B	97.4	98.3	97.6	98.1	98.3	21.0	19.7

5.2. Comparison with existing GIS software

Across all the datasets used, the newly proposed parallel algorithms achieved significantly better performance compared with the reference times generated using the r.accumulate tool. Fig. 11 shows a comparison of the average execution times obtained on machine B, using the full frame datasets.

For the top-down single algorithm, the reference times were on average 10.3 times longer on Ubuntu and 12.9 times longer on Windows, across all the datasets used. It is worth noting that in some cases this ratio reached almost 17 on Ubuntu and exceeded 30 on Windows.

For the double drop algorithm, the reference times obtained on Ubuntu were on average 14.6 times longer, exceeding 26 times in some cases. On Windows, the average ratio was 18.1, reaching almost 46 for some datasets.

It is worth noting that the times obtained by the r.accumulate tool are relatively similar on the two operating systems for smaller datasets, but differ significantly for the larger ones. The exact reason is unknown to the authors. It is also necessary to emphasize that the r.accumulate module operates in a specific environment of the GRASS GIS platform, not as a standalone tool. For this reason, these measurements were intended mainly to provide some external context rather than to serve as a direct comparison.

5.3. Scalability analysis

Fig. 12 shows the average computational times of the two newly proposed parallel algorithms for different multithreading configurations.

The double drop algorithm clearly outperforms the top-down single implementation for smaller numbers of threads. However, as can be seen in Table 4, the top-down single algorithm benefits more from parallel processing when the number of active threads is higher. Ultimately, as the thread limit increases, the differences between the execution times of these two algorithms seem to become more and more insignificant. It is worth noting that both algorithms achieve high speedups even for relatively small numbers of threads.

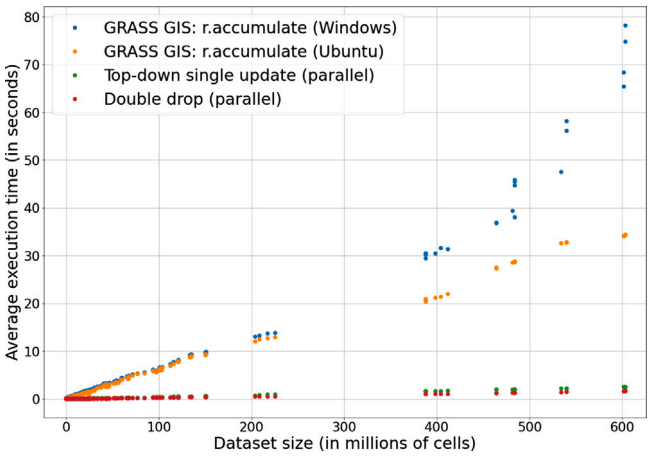


Fig. 11. Average execution times — r.accumulate tool and newly proposed algorithms, machine B.

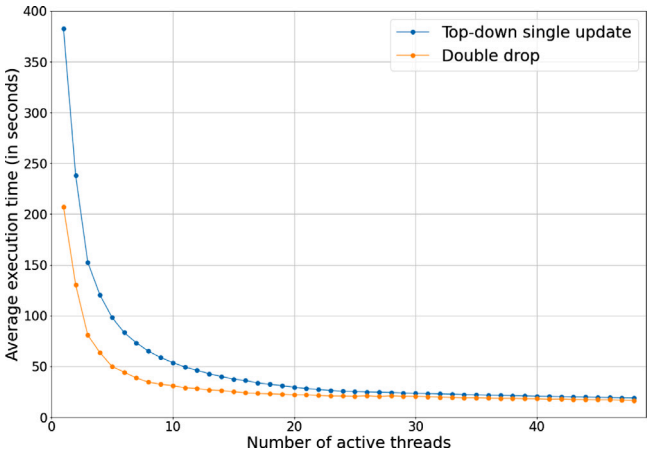


Fig. 12. Average execution times for different thread limits — Barycz FF dataset, machine A.

Table 4
Average speedups of parallel algorithms over their respective sequential versions — Barycz FF dataset, machine A.

Number of active threads	8	16	24	32	40	48
Top-down: single update	5.8	10.5	14.8	16.5	18.3	19.9
Double drop	5.7	8.2	9.4	10.0	10.8	11.9

5.4. Analysis of the generated results

Using the 118 relatively small test datasets (covering the Bystrzyca sub-catchments in ten-meter resolution), the paths identified by the developed algorithms were analyzed and evaluated. In addition to detailed validation, the results were compared to the longest flow paths delineated under ArcGIS Pro 3.0.2 using the flow-length raster approach from Smith (1997). Locations, courses and numbers of cells belonging to the paths confirmed the correctness of the generated results.

Out of the 118 analyzed datasets, the vast majority (104 cases) were found to contain only a single longest flow path. For the remaining 14, it was possible to identify more than one valid outcome. In ten cases, two different paths with the same maximum length were found. Four datasets contained as many as three equally valid alternatives.

However, for all 14 datasets containing more than one longest flow path, the differences between the alternative outcomes were rather small. The number of differently located cells ranged from one to nine, with a median of just two. Alternative source cells were located relatively close to each other.

In five out of 118 cases, the longest flow path did not start at the watershed boundary, but inside the catchment.

6. Conclusions

In this work, the existing algorithms and software tools for finding the longest flow paths were reviewed. Issues were identified with respect to both their performance and precision. Addressing the need for more efficient solutions, new algorithms were developed, tested and presented. Their performance was measured and compared with both other implementations and selected GIS software.

Measurements show that the two new parallel algorithms are able to identify the longest flow paths much more efficiently compared with existing alternatives. These two implementations have distinct properties, and therefore both of them could be considered useful and noteworthy. Depending on the context of use and the needs of a particular user (e.g. requiring all alternative longest paths to be identified, or only estimating maximum path lengths for multiple outlet points), one may be more suitable than the other.

Scalability analysis shows that both new algorithms achieve high speedups even for small numbers of threads. This makes them suitable not only for high performance hardware, but also for widely available low-cost multicore devices.

It is worth noting the differences in the ability of these algorithms to solve specific variants of the problem. The top-down approach is naturally capable of working with multiple outlet points simultaneously, but its data structure allows it to identify only a single longest path leading to each location. On the other hand, the double drop approach is designed to work with a single outlet point, but is capable of recognizing all alternative longest paths at a low additional cost. Moreover, the double drop implementations can be extended relatively easily to process multiple outlet points simultaneously. Thus, the choice of a suitable implementation should depend on the needs of the end user.

Both new algorithms achieve shorter execution times on the flow direction data with cells outside the studied catchment set to NONE. This observation is consistent with the conclusions presented in Kotyra et al. (2021) regarding flow accumulation algorithms. Although this gain may not be large enough to intentionally prepare data in this way for just this operation, it seems worthwhile to use such datasets when available.

It should be noted that the algorithms, techniques and ideas presented in this work are based on the use of flow direction data. Although this kind of approach is generally accepted and used in virtually every available tool for finding the longest flow paths, modeling the underlying terrain and drainage processes in this way may be considered a simplification and therefore a limitation of this study. Perhaps the development of algorithms and tools based on more complex models could be a valuable direction for future research.

CRediT authorship contribution statement

Bartłomiej Kotyra: Conceptualization, Methodology, Software, Investigation, Formal analysis, Validation, Visualization, Writing. **Łukasz Chabudziński:** Data curation, Validation, Visualization, Writing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The source code is freely available in a public GitHub repository. The data used for performance measurements originated from publicly available resources.

References

Ames, D.P., Horsburgh, J.S., Cao, Y., Kadlec, J., Whiteaker, T., Valentine, D., 2012. HydroDesktop: Web services-based software for hydrologic data discovery, download, visualization, and analysis. Environ. Model. Softw. 37, 146–156. http://dx.doi.org/10.1016/j.envsoft.2012.03.013.

Arnold, J.G., Srinivasan, R., Muttiah, R.S., Williams, J.R., 1998. Large area hydrologic modeling and assessment part I: Model development. J. Am. Water Resour. Assoc. 34 (1), 73–89. http://dx.doi.org/10.1111/j.1752-1688.1998.tb05961.x.

Barták, V., 2009. How to extract river networks and catchment boundaries from DEM: a review of digital terrain analysis techniques. J. Landscape Stud. 2, 57–68.

Caprani, O., 1975. Roundoff errors in floating-point summation. BIT Numer. Math. 15 (1), 5–9. http://dx.doi.org/10.1007/BF01932993.

Castro, C.V., Maidment, D.R., 2020. GIS preprocessing for rapid initialization of HEC-HMS hydrological basin models using web-based data services. Environ. Model. Softw. 130, 104732. http://dx.doi.org/10.1016/j.envsoft.2020.104732.

Chapman, B., Jost, G., Pas, R.v.d., 2007. Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press.

Cho, H., 2020. A recursive algorithm for calculating the longest flow path and its iterative implementation. Environ. Model. Softw. 131, 104774. http://dx.doi.org/10.1016/j.envsoft.2020.104774.

Dawson, C., Abrahart, R., Shamseldin, A., Wilby, R., 2006. Flood estimation at ungauged sites using artificial neural networks. J. Hydrol. 319 (1), 391–409. http://dx.doi.org/10.1016/j.jhydrol.2005.07.032.

Djokic, D., Ye, Z., Dartiguenave, C., 2011. Arc Hydro Tools overview. Redland, Canada, ESRI 5.

Fairfield, J., Leymarie, P., 1991. Drainage networks from grid digital elevation models. Water Resour. Res. 27 (5), 709–717. http://dx.doi.org/10.1029/90WR02658.

Feldman, A.D., 2000. Hydrologic Modeling System HEC-HMS: Technical Reference Manual. US Army Corps of Engineers, Hydrologic Engineering Center, Davis, CA.

Gallant, J.C., Wilson, J.P., 1996. TAPES-G: A grid-based terrain analysis program for the environmental sciences. Comput. Geosci. 22 (7), 713–722. http://dx.doi.org/10.1016/0098-3004(96)00002-7.

Gichamo, T.Z., Sazib, N.S., Tarboton, D.G., Dash, P., 2020. HydroDS: Data services in support of physically based, distributed hydrological models. Environ. Model. Softw. 125, 104623. http://dx.doi.org/10.1016/j.envsoft.2020.104623.

Goodall, J.L., Robinson, B.F., Castronova, A.M., 2011. Modeling water resource systems using a service-oriented computing paradigm. Environ. Model. Softw. 26 (5), 573–582. http://dx.doi.org/10.1016/j.envsoft.2010.11.013.

Hack, J.T., 1957. Studies of longitudinal stream profiles in Virginia and Maryland. Geological Survey Professional Paper 294-B, http://dx.doi.org/10.3133/pp294B.

Huang, P.-C., Lee, K.T., 2016. Distinctions of geomorphological properties caused by different flow-direction predictions from digital elevation models. Int. J. Geogr. Inf. Sci. 30 (2), 168–185. http://dx.doi.org/10.1080/13658816.2015.1079913.

Jaffrés, J.B., Cuff, B., Cuff, C., Faichney, I., Knott, M., Rasmussen, C., 2021. Hydrological characteristics of Australia: relationship between surface flow, climate and intrinsic catchment properties. J. Hydrol. 603, 126911. http://dx.doi.org/10.1016/j.jhydrol.2021.126911.

Jenson, S.K., Domingue, J.O., 1988. Extracting topographic structure from digital elevation data for geographic information system analysis. Photogramm. Eng. Remote Sens. 54 (11), 1593–1600.

Karalis, S., Karymbalis, E., Valkanou, K., Chalkias, C., Katsafados, P., Kalogeropoulos, K., Batzakis, V., Bofilios, A., 2014. Assessment of the relationships among catchments' morphometric parameters and hydrologic indices. Int. J. Geosci. 05 (13), 1571–1583. http://dx.doi.org/10.4236/ijg.2014.513128.

Kotyra, B., Chabudziński, L., Stpiczyński, P., 2021. High-performance parallel implementations of flow accumulation algorithms for multicore architectures. Comput. Geosci. 151, 104741. http://dx.doi.org/10.1016/j.cageo.2021.104741.

Latt, Z., Wittenberg, H., Urban, B., 2015. Clustering Hydrological Homogeneous Regions and neural network based index flood estimation for ungauged catchments: an example of the Chindwin River in Myanmar. Water Resour. Manag. 29, http://dx.doi.org/10.1007/s11269-014-0851-4.

Li, Z., Yang, T., Xu, C.-Y., Shi, P., Yong, B., Huang, C.-S., Wang, C., 2020. Evaluating the area and position accuracy of surface water paths obtained by flow direction algorithms. J. Hydrol. 583, 124619. http://dx.doi.org/10.1016/j.jhydrol.2020.124619.

Lindsay, J.B., 2016. Whitebox GAT: A case study in geomorphometric analysis. Comput. Geosci. 95, 75–84. http://dx.doi.org/10.1016/j.cageo.2016.07.003.

Lindsay, J., 2022. WhiteboxTools user manual, version 2.2.0. URL https://www.whiteboxgeo.com/manual/wbt_book/.

Linz, P., 1970. Accurate floating-point summation. Commun. ACM 13 (6), 361–362. http://dx.doi.org/10.1145/362384.362498.

Maathuis, B.H.P., Wang, L., 2006. Digital elevation model based hydro-processing. Geocarto Int. 21 (1), 21–26. http://dx.doi.org/10.1080/10106040608542370.

Maidment, D., Morehouse, S., 2002. Arc Hydro: GIS for Water Resources (3rd Edition). ESRI Press.

Merkel, W.H., Kaushika, R.M., Gorman, E., 2008. NRCS GeoHydro—A GIS interface for hydrologic modeling. Comput. Geosci. 34 (8), 918–930. http://dx.doi.org/10.1016/j.cageo.2007.05.020.

Michailidi, E.M., Antoniadi, S., Koukouvinos, A., Bacchi, B., Efstratiadis, A., 2018. Timing the time of concentration: shedding light on a paradox. Hydrol. Sci. J. 63 (5), 721–740. http://dx.doi.org/10.1080/02626667.2018.1450985.

Neteler, M., Bowman, M.H., Landa, M., Metz, M., 2012. GRASS GIS: A multi-purpose open source GIS. Environ. Model. Softw. 31, 124–130. http://dx.doi.org/10.1016/j.envsoft.2011.11.014.

O'Callaghan, J.F., Mark, D.M., 1984. The extraction of drainage networks from digital elevation data. Comput. Vis. Graph. Image Process. 28 (3), 323–344. http://dx.doi.org/10.1016/S0734-189X(84)80011-0.

Olivera, F., 2001. Extracting hydrologic information from spatial data for HMS modeling. J. Hydrol. Eng. 6 (6), 524–530. http://dx.doi.org/10.1061/(ASCE)1084-0699(2001)6:6(524).

Olivera, F., Dodson, R., Djokic, D., 2003. Use of Arc Hydro for integration of hydrologic applications. In: World Water &Amp; Environmental Resources Congress 2003. pp. 1–9. http://dx.doi.org/10.1061/40685(2003)249.

Olivera, F., Maidment, D., 1998. Geographic information system use for hydrologic data development for design of highway drainage facilities. Transp. Res. Rec. 1625 (1), 131–138. http://dx.doi.org/10.3141/1625-17.

Orlandini, S., Moretti, G., 2009. Determination of surface flow paths from gridded elevation data. Water Resour. Res. 45 (3), http://dx.doi.org/10.1029/2008WR007099.

Paz, A.R.d., Collischonn, W., Risso, A., Mendes, C.A.B., 2008. Errors in river lengths derived from raster digital elevation models. Comput. Geosci. 34 (11), 1584–1596. http://dx.doi.org/10.1016/j.cageo.2007.10.009.

Qin, C.-Z., Zhan, L., 2012. Parallelizing flow-accumulation calculations on graphics processing units—From iterative DEM preprocessing algorithm to recursive multiple-flow-direction algorithm. Comput. Geosci. 43, 7–16. http://dx.doi.org/10.1016/j.cageo.2012.02.022.

Ramly, S., Tahir, W., 2016. Application of HEC-GeoHMS and HEC-HMS as Rainfall–Runoff model for flood simulation. In: Tahir, W., Abu Bakar, P.I.D.S.H., Wahid, M.A., Mohd Nasir, S.R., Lee, W.K. (Eds.), ISFRAM 2015. Springer Singapore, Singapore, pp. 181–192. http://dx.doi.org/10.1007/978-981-10-0500-8_15.

SCALGO, 2022. SCALGO Live documentation. URL https://scalgo.com/en-US/scalgo-live-documentation.

Schäuble, H., Marinoni, O., Hinderer, M., 2008. A GIS-based method to calculate flow accumulation by considering dams and their specific operation time. Comput. Geosci. 34 (6), 635–646. http://dx.doi.org/10.1016/j.cageo.2007.05.023.

Smith, P.N., 1995. Hydrologic data development system. (Master's thesis). University of Texas, Austin, URL http://hdl.handle.net/2152/6732.

Smith, P.N., 1997. Hydrologic data development system. Transp. Res. Rec. 1599 (1), 118–127. http://dx.doi.org/10.3141/1599-15.

Sten, J., Lilja, H., Hyväluoma, J., Westerholm, J., Aspnäs, M., 2016. Parallel flow accumulation algorithms for graphical processing units with application to RUSLE model. Comput. Geosci. 89, 88–95. http://dx.doi.org/10.1016/j.cageo.2016.01.006.

Stpiczyński, P., 2018. Language-based vectorization and parallelization using intrinsics, OpenMP, TBB and Cilk Plus. J. Supercomput. 74 (4), 1461–1472. http://dx.doi.org/10.1007/s11227-017-2231-3.

Sultan, D., Tsunekawa, A., Tsubo, M., Haregeweyn, N., Adgo, E., Meshesha, D.T., Fenta, A.A., Ebabu, K., Berihun, M.L., Setargie, T.A., 2022. Evaluation of lag time and time of concentration estimation methods in small tropical watersheds in Ethiopia. J. Hydrol.: Reg. Stud. 40, 101025. http://dx.doi.org/10.1016/j.ejrh.2022.101025.

Tang, W., Wang, S., 2020. High Performance Computing for Geospatial Applications. http://dx.doi.org/10.1007/978-3-030-47998-5.

Tarboton, D.G., Bras, R.L., Rodriguez-Iturbe, I., 1991. On the extraction of channel networks from digital elevation data. Hydrol. Process. 5 (1), 81–100. http://dx.doi.org/10.1002/hyp.3360050107.

Vitolo, C., Elkhatib, Y., Reusser, D., Macleod, C.J., Buytaert, W., 2015. Web technologies for environmental big data. Environ. Model. Softw. 63, 185–198. http://dx.doi.org/10.1016/j.envsoft.2014.10.007.

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. pp. 467–472.

Wilkinson, J.H., 1963. Rounding Errors in Algebraic Processes. H'M's Stationery Office, London.

Zhou, G., Wei, H., Fu, S., 2019. A fast and simple algorithm for calculating flow accumulation matrices from raster digital elevation. Front. Earth Sci. 13 (2), 317–326. http://dx.doi.org/10.1007/s11707-018-0725-9.