

GPU-Accelerated Hydrology Algorithms for On-Prem Computation: Flow Accumulation, Drainage Lines, Watershed Delineation, Runoff Simulation

Rahul Kumar

Indian Institute of Technology Delhi
India
mcs232491@iitd.ac.in

Abhilash Jindal

Indian Institute of Technology Delhi
India
ajindal@cse.iitd.ac.in

Vatsal Jingar

Indian Institute of Technology Delhi
India
Vatsal.Jingar.cs520@cse.iitd.ac.in

Aaditeshwar Seth

Indian Institute of Technology Delhi
India
aseth@cse.iitd.ac.in

Abstract

Critical hydrology related algorithms to trace the path of surface water flows, including flow accumulation, stream order, watershed delineation, and runoff simulation, can be difficult to execute for large aerial extents at fine spatial and temporal resolutions. Libraries like GDAL that use multi-threaded CPU-based implementations running on a single host may be slow, and distributed infrastructures like Google Earth Engine may not support the kind of computational primitives required by these algorithms. We have developed a GPU-accelerated framework that re-engineers these four algorithms and is able to process areas as large as river basins of 250,000 km² on commodity GPU workstations. We express these algorithms in terms of easily parallelizable primitives of pixel independent (PI) and short-pixel (SP) operations, and iterative primitives of long-pixel (LP) operations. Each algorithm uses a different mix of the primitives which helps us ensure that the implementation is generic. We show that our implementation of these algorithms produces accurate outputs and is able to achieve significant performance benefits over alternative methods. Being able to execute the algorithms on a commodity GPU workstation paves the path to use on-prem infrastructure to produce national-scale outputs, and collaborate to pool multiple national-scale outputs together for global-scale analysis.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; • **Information systems** → *Geographic information systems*; • **Applied computing** → *Environmental sciences*.



This work is licensed under a Creative Commons Attribution 4.0 International License.

PROPL '25, Singapore, Singapore

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2161-8/25/10

<https://doi.org/10.1145/3759536.3763805>

Keywords: GPU, hydrology, parallel computing, flow accumulation, watershed delineation, runoff simulation

ACM Reference Format:

Rahul Kumar, Vatsal Jingar, Abhilash Jindal, and Aaditeshwar Seth. 2025. GPU-Accelerated Hydrology Algorithms for On-Prem Computation: Flow Accumulation, Drainage Lines, Watershed Delineation, Runoff Simulation. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming for the Planet (PROPL '25)*, October 12–18, 2025, Singapore, Singapore. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3759536.3763805>

1 Introduction

Hydrological modeling on geospatial data is a critical tool to understand and manage water resources, predict natural hazards, and support environmental planning. As the scale and complexity of geospatial datasets continue to grow, traditional computational approaches face significant performance bottlenecks that limit their practical application. A few such examples in the hydrology context are listed below:

Flow accumulation – calculate the upstream aerial extent from which rainfall runoff is expected to drain into a point. This algorithm is crucial to assess which areas might be prone to waterlogging or floods, and is an intermediate step to map river and stream networks [19].

Drainage line hierarchy – channels along with rainfall runoff are expected to flow can be identified using the flow accumulation algorithm. These drainage line segments are subsequently labeled based on their hierarchical position, called their stream order. This particularly helps assess stream flow and sediment transport volumes

Watershed delineation – considered as a basic hydrological unit for water-balance studies, a watershed is a closed aerial unit with one identified outlet point for rainfall runoff. Delineating boundaries of watersheds in a landscape is important to understand the hydrological connectivity in a

GitHub Repository: <https://github.com/RahulkumarRV/HydroGPU>

landscape and to plan infrastructure like dams and check-dams [6].

Runoff simulation – an hourly or daily simulation of the movement of surface water based on rainfall and the underlying terrain that determines flow direction and flow velocity is used to model inundation with heavy rainfall events and expected water storage in reservoirs and smaller waterbodies [10][3.1.2.2].

These algorithms typically analyze geospatial data organized into grids of pixels, one of the most significant sources being Digital Elevation Models (DEMs) that provide the elevation above mean sea level of each pixel. Raster layers of DEMs come in various resolutions, ranging from 1 meter to 90 meters or more, with finer resolutions providing greater detail at the cost of increased computational demand. The algorithms operating on such DEM layers can be expressed in terms of the following primitive operations:

Pixel independent operations (PI) computations are where the output value of each pixel depends solely on the input value at the same spatial location without considering neighboring pixels or spatial context. These operations are inherently parallelizable due to the absence of any inter-pixel dependencies. For example, to calculate the total runoff volume expected to arrive at a pixel based on the daily runoff estimates is simply a sum of the daily values recorded and stored at that pixel. This is illustrated in Figure A 1.

Short Pixel Operations (SP) (Figure B 1) are computations where the output value of each pixel is based on the values of pixels within a limited predefined neighborhood around the corresponding input pixel. This “bounded distance” means that only nearby pixels contribute to the calculation. For example, determining the dominant flow direction at each pixel can be done by analyzing the elevation values of its eight immediate neighbors and find the maximum to identify the direction of steepest descent.

Long pixel operations (LP) Figure 1 encompasses computations where an output pixel’s value can depend on pixels located at an unbounded distance across the entire raster. Examples include flow accumulation as shown in figure C 1, where the value of the last pixel is highlighted as a red cell with a flow accumulation value of 21 and was obtained by propagating values along the dominant flow direction from many upstream cells which may not be limited by any predefined distance.

Existing tools to run these operations present significant limitations. GDAL based libraries such as Grass which are widely used for geospatial processing rely primarily on CPU based computations with only thread level parallelization [2], and often result in processing times of hours or system crashes when the algorithms are run on large watersheds or river basins. Google Earth Engine (GEE) provides a distributed map-reduce framework to scale the aerial extent over which computations become feasible, but only provides support for SP and IP operations, with no support for LP

computations that might require iterative updates [3, 5]. GEE can also struggle with memory limitations when processing large datasets and may terminate tasks depending on the load.

In this paper, we address such computational bottlenecks by developing GPU-accelerated solutions for hydrological analysis. We show that it is feasible to leverage GPUs to achieve significant performance improvements while maintaining computational accuracy, particularly targeting the challenging LP operations that current tools are unable to handle efficiently. Importantly, our solutions are designed to run on commodity GPU workstations without requiring access to centralized high-performance computing infrastructures. All experiments and benchmarks were done on a workstation equipped with an Intel Core i9 CPU, 64 GB RAM, and an NVIDIA RTX GPU with 24 GB of VRAM. We found that this infrastructure is sufficient for river basins as large as 250,000 km² and can meet a service-level agreement (SLA) target of computing the above tasks within a few hours for such a basin. Researchers or modelers can therefore use local on-prem compute infrastructure for such analysis easily at a national scale; for instance, India has 25 major river basins of which only three have an aerial extent between 250,000 to 315,000 km² [7], and therefore all the basins can be serially processed on one or a couple of such workstations within a few days. National-scale outputs produced by different research groups can also be pooled together for larger global-scale analysis if needed.

Other studies also highlight the advantages of GPU accelerated geospatial processing in scientific workflows[14, 16], but generally focus on pixel-independent and short-pixel operations, which are easier to parallelize. Our main contribution is to demonstrate that long-pixel operations such as iterative propagation of runoff with path tracing over a heterogeneous terrain is also feasible and can be done on commodity workstations. When combined with a vision of planetary computing that makes dataset and algorithm sharing and combining easier [9], the feasibility of such local on-prem computation can pave the path for decentralized pooling of resources for planetary scale outputs.

2 Methodology

Flow accumulation, drainage lines extraction and stream order numbering, watershed delineation, and runoff simulation are key hydrological algorithms that we chose to demonstrate implementation since they use all three (SP, PI, LP) types of operations.

We use CuPy [13] for GPU computing with vectorized operations and custom kernels. A vectorized approach means performing operations on entire arrays or matrices simultaneously, rather than processing elements one by one with explicit loops. A custom kernel approach involves writing specialized, low-level functions (kernels) that execute directly

Runoff 1			Runoff 2			Summed runoff		
1	2	3	0	2	3	1	4	6
1	3	1	1	3	0	2	6	1
2	1	2	0	1	2	2	2	4

Figure A - Pixel Independent (PI) Operation, value of each pixel represents the runoff volume at that location

DEM					Flow Direction		
78	72	69	71	58	2	2	4
74	67	56	49	46	1	2	4
69	53	44	37	38	128	1	2
64	58	55	22	31			
68	61	47	21	16			

Figure B. Short Pixel (SP) Operation, DEM, each pixel value represents the elevation at that location, and the flow direction shows the direction of the steepest neighbor according to direction coding

Flow direction Grid					Flow Accumulation				
2	2	2	4	4	1	1	1	1	1
2	2	2	4	4	1	2	2	3	2
1	1	2	4	8	1	4	8	6	3
128	128	1	2	4	1	1	1	19	1
2	2	1	4	2	1	1	1	2	21

Figure C. Long pixel (LP) Operation, flow direction of each pixel represents the steepest neighbor according to direction coding, and flow accumulation, where each pixel value represents. The number of upstream pixels draining into the pixel

Figure 1. Pixel Independent, Short Pixel, Long Pixel Operations

on the hardware's parallel processing units. This offers fine-grained control over execution for tailored computations when pre-built functions are not sufficient.

2.1 Flow Accumulation

The flow accumulation algorithm [1] computes for each cell the number of upstream pixels from which flow accumulates at the cell. It is computed after determining the flow direction for each cell, typically obtained from a Digital Elevation Model (DEM) [11] using methods such as the D8 [12] (deterministic eight-node) algorithm that identifies the direction of maximum downstream slope. Mathematically, for a cell c , the flow accumulation $A(c)$ is defined recursively as:

$$A(c) = 1 + \sum_{i \in P(c)} A(i)$$

Where $P(c)$ is the set of all immediate upstream cells that flow into cell c .

The CPU-based implementation as used in the GDAL grass library is illustrated in **Algorithm 1**. This approach relies on iterative computation and processes each pixel sequentially, resulting in long computation times, especially as the input size increases. To address this, we implemented an optimized GPU-based method shown in **Algorithm 2**. This works as follows.

Flow accumulation is itself an LP operation that uses multiple SP or PI operations internally. Initially, PI and SP operations are done using CuPy [13] arrays to allocate and set up the flow direction (flow_dir), accumulation raster (acc), indegree count (in_deg), and two working arrays — active and next_active — for tracking cells in successive iterations. Following this initialization, GPU kernels are invoked to compute the upstream contributor count shown in **Algorithm 3** to identify the initial set of active cells with no inflow (KERNEL_INITACTIVE) **Algorithm 4**. These are cells for which $\text{in_deg} = 0$ and represent the starting points of flow paths.

The main loop of the algorithm then applies a Long Pixel (LP) operation that iteratively calls KERNEL_PROPAGATEFLOW shown in **Algorithm 6**, itself an SP operation, to update downstream accumulation values and identify the next set of active cells. This strategy ensures that only necessary cells are processed in each iteration, enabling efficient wavefront-style propagation of flow.

This GPU-based method is highly efficient, as it exploits the parallel nature of CUDA by concurrently processing multiple independent flow paths.

The flow accumulation expands outward from leaf nodes (cells with no inflow), updating downstream values layer by layer. The total number of iterations required is proportional to the length of the longest flow path in the DEM rather than the total number of pixels. This results in substantial performance gains over traditional CPU-based methods that process pixels sequentially.

2.2 Drainage Line Hierarchy

Drainage lines represent the hierarchical structure of flow paths within a river or stream network. This hierarchy is computed after deriving flow direction and flow accumulation from the DEM. Once flow accumulation values are obtained for each pixel, a threshold is applied to identify potential drainage lines. A low threshold will help identify fine drainages while a higher threshold will only identify cells that are a part of larger streams. The segments in this network then need to be ranked using hierarchical rules, commonly following either the Strahler or Shreve method, as outlined next.

This process of ranking can be considered as a label propagation operation since each cell's label (or rank) depends on upstream contributors. This is also an LP operation since the distance of propagation is not fixed. Traditional tools like GDAL use a sequential breadth-first search (BFS)-like approach for this task which becomes slow on large inputs. Applying a GPU-based parallel method similar to the flow accumulation strategy can significantly speed this up. Two common strategies are followed to label the drainage lines as outlined below.

Strahler Method: In this approach, all headwater cells (drainage sources) are assigned a rank of 1. When two drainage

lines of the same rank meet, the downstream cell's rank is assigned as one more than this rank. If the merging drainage lines have different ranks, the downstream cell is assigned the value of the maximum of the two. As shown in **Algorithm 7**, the algorithm first identifies stream cells using a threshold on flow accumulation (`KERNEL_STREAM_CELL`, **Algorithm 5**), and then `KERNEL_COMPUTEINDEGREE` is used to compute the in-degree (number of upstream contributors) of the cells shown in **Algorithm 3**. Cells with zero in-degree are initialized with a rank of 1 and marked as active cells. The GPU then processes all active cells in parallel, updating downstream ranks based on upstream values. This efficiently models branching drainage networks.

Shreve Method: This method also starts with assigning rank 1 to headwater cells, but instead of choosing the max or incrementing by one when drainage lines meet, it sums the ranks of contributing upstream cells. This gives a sense of total upstream drainage volume. **Algorithm 8** shows the procedure. Using `SHREVE_KERNEL`, the GPU adds the value from each active cell to its downstream target based on the flow direction. The process repeats until all drainage lines are processed. Since it uses simple addition, it involves fewer conditional checks and runs faster on large datasets.

2.3 Watershed Delineation

The algorithm expects a threshold of the aerial extent in which to segment a landscape into watersheds. Typically, micro-watershed units of 500-1000 hectares are used for water security and flood planning purposes. The process of delineation begins with applying the *threshold* on flow accumulation values of cells belonging to the drainage network. Cells with values above this threshold are called **stream cells**.

A subset is then identified of **convergence points** with an in-degree greater than one where multiple stream segments are joining. Next, **outlet cells** adjacent to the convergence points are identified which flows directly into the convergence point. All steps so far are IP or SP operations and the GPU kernel (`KERNEL_FIND_OUTLETS`) shown in **Algorithm 10** is used to assign a *unique ID* to the outlet cells.

Through an LP operation, this unique ID then needs to be propagated upstream through the stream network using a parallel wavefront-style traversal until the aggregated stream segments have a maximum flow accumulation count below the aerial extent of the watershed delineation that is required. Next, the flow direction is used to identify all cells draining into these stream cells and are assigned the ID for the **watershed**. The process is then repeated for other outlet cells that have not been labeled as yet. **Algorithm 9** shows the pseudocode representation of this algorithmic process.

2.4 Runoff Simulation

Runoff simulation is crucial to study the spatial and temporal behavior of surface water flow across a landscape [10] and

involves long pixel operations since the rainfall on a pixel can flow to an unbounded set of downstream pixels. The conventional method involves computing runoff for each time slice (E.g., hourly or 3-hourly or daily intervals) using models operating on underlying layers including DEM (to obtain the slope - higher slope leads to a higher runoff) [11], Land Use and Land Cover (LULC - more vegetated pixels lead to lower runoff) [4], and hydrological soil properties (E.g. clayey soils lead to higher runoff than sandy soils) [15], and rainfall data (both current and past rainfall data is used to model the wetness of the soil at that time - dry soils would lead to greater infiltration and less runoff than soils which have already been wetted from earlier rainfall). Once per-pixel runoff values have been computed for a given interval, the flow accumulation algorithm is used to simulate the downstream movement of this runoff, i.e. the algorithm instead of counting the number of cells draining into a given cell, now sums the runoff draining into a given cell.

While the per-pixel runoff values can be computed using mostly IP operations as mentioned in Figure 2, the runoff accumulation approach has redundant computation since the flow accumulation is recalculated at each timestep, even though the mapping of any source pixel to destination pixels remains the same. Simulating just one iteration, this flow accumulation of 3-hour runoff over a river basin using GDAL took over 20 hours, making it infeasible to run the computation for 3 months, which is the typical monsoon duration in India during which such simulations are required.

Our GPU-accelerated runoff simulation pipeline is outlined in Figure 2. The main idea is to precompute a jump matrix (an LP operation) by simulating how far a cell's runoff can travel within a fixed time window (E.g., 3 hours), considering the terrain. For each pixel (i, j) , the transfer time to its steepest downslope neighbor is calculated as $\Delta t = \frac{d}{v}$ where d is the pixel resolution and v is the velocity calculated using the Manning equation [18]. Over successive iterations, the runoff generated on a pixel thus "jumps" to its downstream destination until the cumulative travel time exceeds the pre-defined time window. The final output is a **Jump Matrix**, where each entry stores the row-major index of the destination pixel. Once generated, this matrix enables direct transfer (a short pixel operation) of runoff volume to the destination pixel across all timesteps. Figure 12 shows the **Jump Matrix** operation, followed by the **Route** function, which uses the **Jump Matrix** to transfer runoff from each cell to its destination.

Another slight difference in the GPU-accelerated version is that calculation of the per-pixel runoff incorporates the incoming runoff into the pixel (transported from upstream pixels via the **Jump Matrix**) in addition to the rainfall on that pixel. This is shown as the red colored lines in Figure 2. This would lead to more accurate runoff estimation by also modeling incoming runoff into a pixel from upstream areas

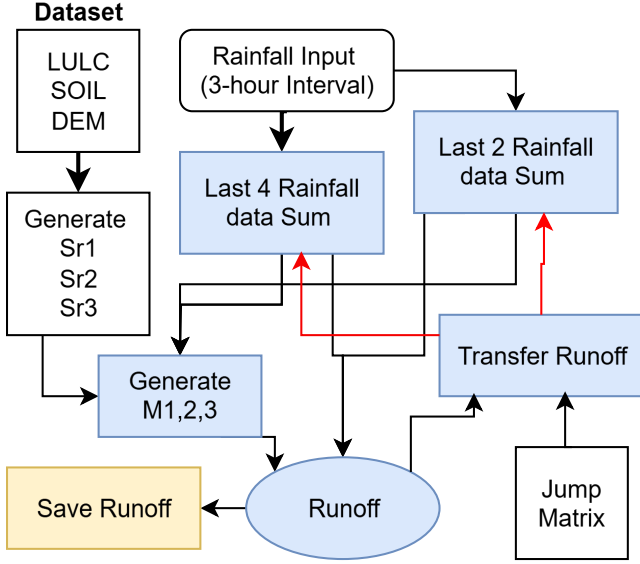


Figure 2. Runoff simulation pipeline showing the generation of abstraction layers (SR1–3), moisture modifiers (M1–3), and the use of daily and past rainfall (P, P5) for final runoff estimation. All SR and M layers are derived from slope-adjusted curve numbers (CN1a–3a) based on land use, soil type, and DEM-derived slope. Each iteration saves runoff and transfers using a jump matrix, used for the next runoff iteration. White Nodes indicate static operations that are computed once and can be saved for subsequent use.

in addition to the rain falling on that pixel, as compared to simpler approaches which only consider runoff resulting from rainfall on a pixel.

3 Results

We used an Intel Core i9 CPU with 64 GB RAM and an NVIDIA RTX GPU with 24 GB of VRAM. GPU-based algorithms were run using our CuPy implementation while CPU-based algorithms were run using GDAL configured with hyper-threading enabled.

We next present the performance and accuracy of our GPU accelerated framework for hydrological computations, comparing it with traditional CPU-based tools like GDAL and Google Earth Engine (GEE).

Experiments were conducted using the SRTM DEM [11], LULC [4], and soil datasets [15] reprojected to a common 30m resolution, and done at two scales: on the Lower Ganga Basin (LGB) which covers an aerial extent of approximately 249,661 km² [8] with a total DEM raster size of 2.6 GB, and on a smaller area of the Masalia sub-district in the Dumka district of Jharkhand, covering 464.81 km² [17] with a raster size of 5 MB. The smaller size dataset primarily served the role of validating our outputs against those produced by standard algorithm implementations in GDAL.

Flow accumulation: Table 1 highlights the substantial performance gains achieved by our GPU-accelerated flow accumulation implementation. Compared to GDAL (CPU-based processing), the GPU approach achieved a speedup of over 5× for the entire LGB and approximately 20× for the smaller Masalia region.

To validate our output, we extracted the Masalia sub-region (highlighted by the yellow square in the top-center map) from the GPU-computed flow accumulation for the entire LGB and also computed for this area using the GDAL flow accumulation implementation (Figure 3). Panel A shows the CPU-based (GDAL) flow accumulation visualized in QGIS while Panel B shows the corresponding GPU-based output clipped from the LGB output. Both are identical. Further, Panel C shows a pixel-wise scatter plot comparing flow accumulation values between the GPU and CPU outputs, confirming the correctness of the GPU implementation.

Drainage lines extraction: Our GPU implementation significantly accelerated both Strahler and Shreve stream order computations, as shown in Table 1. The Shreve method runs faster than Strahler largely due to its simpler algorithmic structure which involves less conditional branching, but in both cases the GPU implementation runs substantially faster than the CPU implementation. Figure 4 shows the comparison for validation purposes as well between the GPU based LGB output clipped to Masalia and the CPU output produced using GDAL for Masalia. Minor variations exist due to boundary effects arising from channels draining into Masalia which are not considered in the CPU version (see the straight edge dam structure on the East side, for example).

Micro-Watershed Delineation: As shown in Table 1, the GPU version consistently outperformed GDAL across both small (Masalia) and large-scale (LGB) regions. Visual comparisons for validation purposes shown in Figure 5 for the smaller region confirm the correctness of the implementation.

Runoff Simulation Time Series: The GPU-based runoff simulation pipeline demonstrates substantial performance improvements over traditional methods such as those using a combination of Google Earth Engine (GEE) and GDAL for computation. As summarized in Table 1, the initial iteration over the LGB to generate the velocity raster and jump matrix took approximately 30 minutes. These precomputed layers are reused in subsequent iterations, allowing each 3-hour simulation step to complete on average in 70 seconds, as shown in Figure 6C. Three key components make up this computation time: The per-pixel runoff computation and flow transfer time Figure 6D, time taken for disk to GPU data transfer Figure 6E, and time taken for GPU to disk data transfer Figure 6F.

The time taken to complete a full 3-month duration (720 time-steps) was approximately 14 hours. Although this is beyond the ideal SLA we wanted to achieve, the time can be

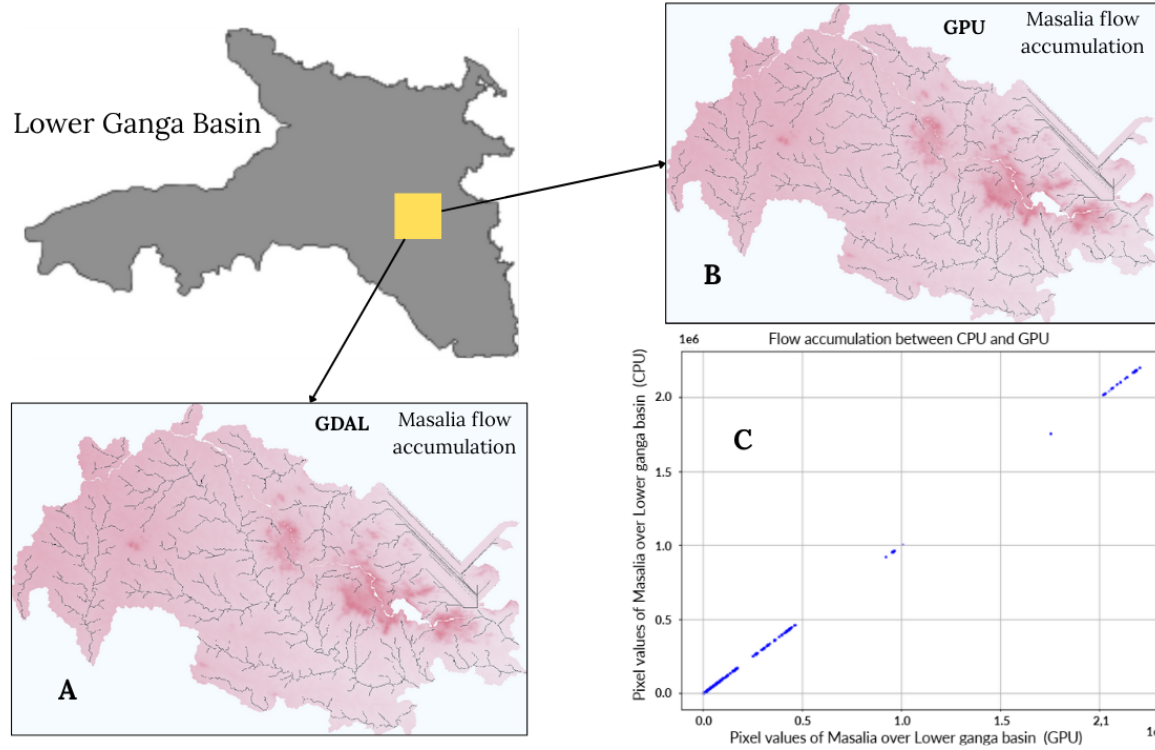


Figure 3. This figure shows the flow accumulation output for the Lower Ganga basin. A smaller region, the Masalia sub-district, was selected for comparative visualization.

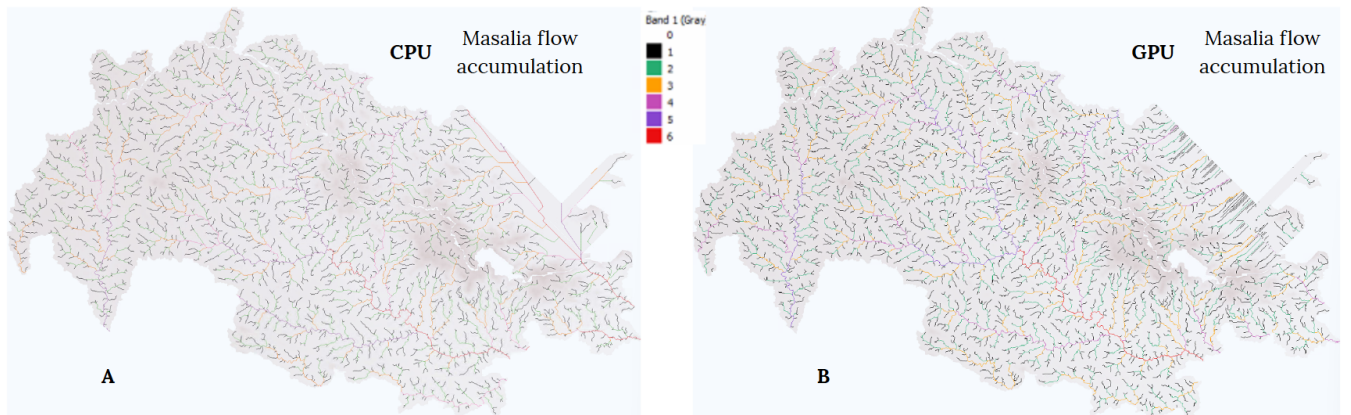


Figure 4. Figure A shows the GDAL-based drainage lines order. Figure B shows the GPU-based drainage lines order.

reduced by increasing the timestep of 3-hours at the cost of estimation fidelity. For example, with a 24-hour timestep, it takes only 2.5 hours for a 3-month simulation. In contrast, the GEE + GDAL pipeline failed to complete the full 3-month simulation at a 3-hour timestep, and required over 20 hours for a single flow accumulation timestep alone.

Figures 6A and 6B provide a comparative validation showing scatter plots of GPU vs GEE + GDAL per-pixel runoff values at two pixels within the LGB. Both show high agreement in runoff values across the frameworks, with minor

discrepancies due to the GPU version also accounting for incoming runoff in addition to rainfall for generation of the next iteration of runoff.

4 Future Work

First, while GPU implementation yields substantial speedups, some challenges remain. Most notably, memory constraints on a single GPU limit the spatial resolution that can be handled. We used the SRTM 30m DEM which is a popular data

Micro-Watershed

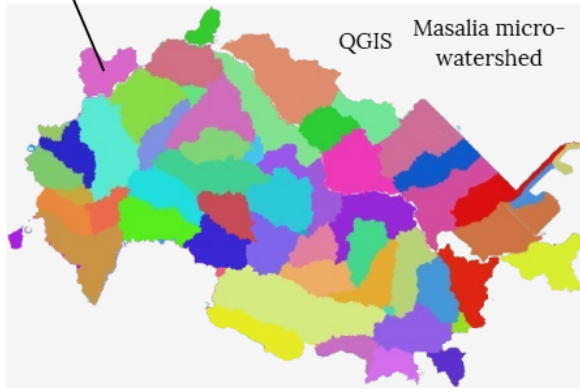


Figure A

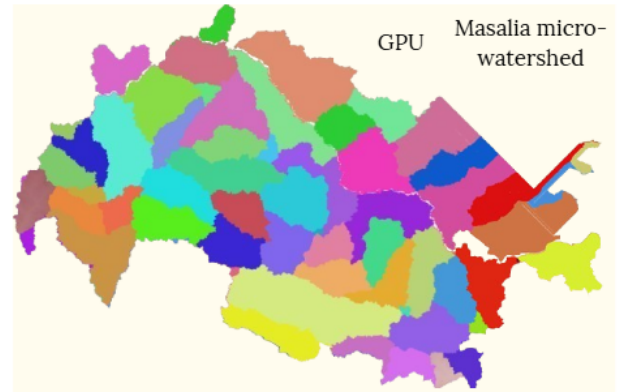


Figure B

Figure 5. Figure A shows GDAL micro-watersheds and Figure B shows GPU micro-watersheds.

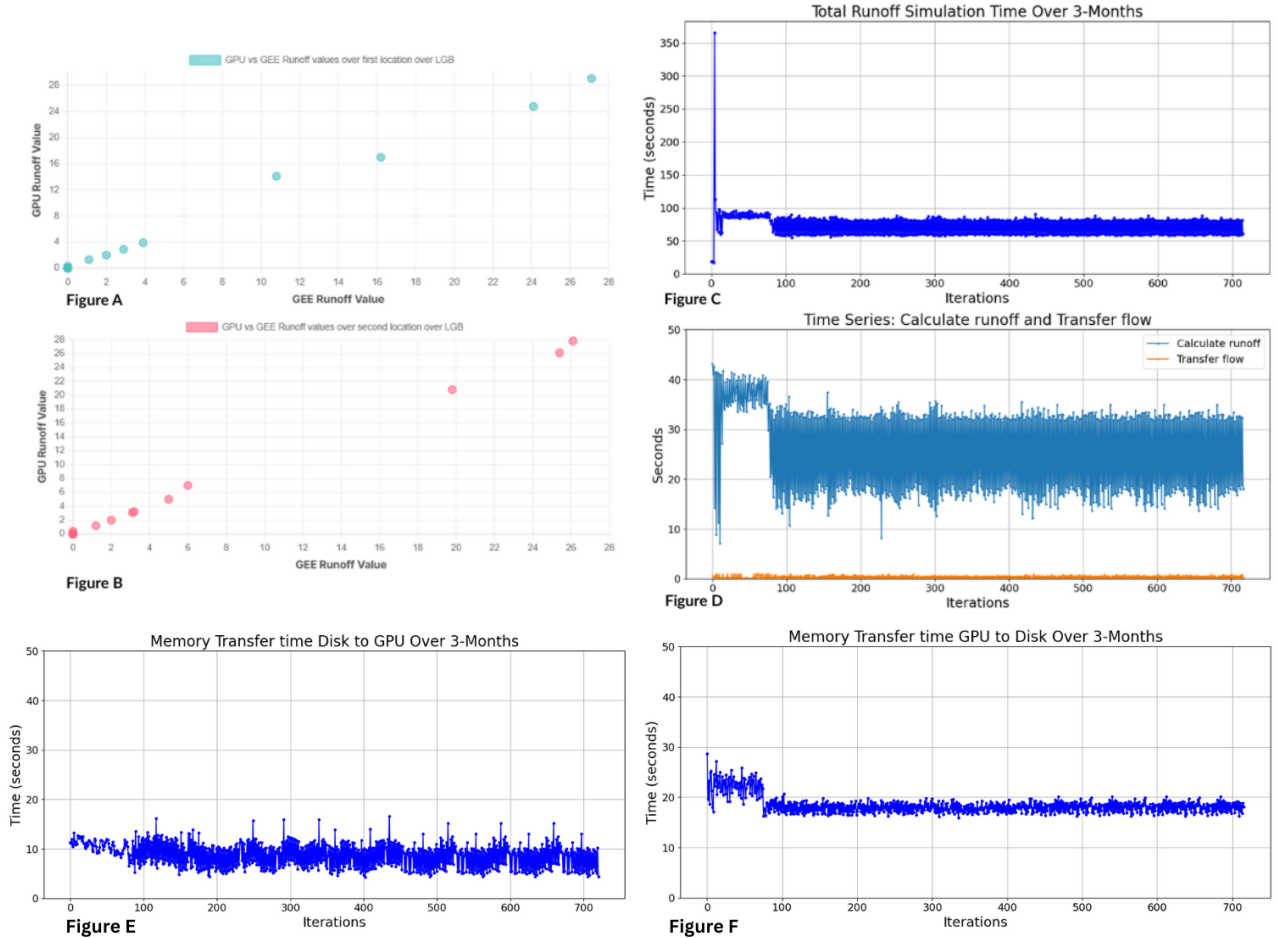


Figure 6. Runoff simulation performance and accuracy comparison. (A) GPU vs GEE+GDAL runoff values for the first location in the LGB; (B) GPU vs GEE+GDAL runoff values for the second location; (C) GPU-based 3-month total runoff simulation time over 720 iterations; (D) Per-pixel runoff computation and flow transfer time. (E) Memory transfer from disk to GPU, which includes a one-time transfer of the Soil, LULC, DEM, and FLOW direction layers, and in each iteration a transfer of rainfall data over the basin size. (F) Memory transfer from GPU to disk, which includes the final runoff output of each iteration. Intermediate results not needed for the next iteration are deallocated.

Table 1. Computation Time and Memory Comparison for Hydrological Tasks: CuPy GPU vs GDAL/GEE

Common Input Size: Masalia = 5 MB, Lower Ganga Basin (LGB) = 2.6 GB, H = height of input image, W = width of input image, C and K are constants

Task	Metric	CuPy GPU (Masalia)	CuPy GPU (LGB)	GDAL/GEE (Masalia)	GDAL/GEE (LGB)
Flow Accumulation	Computation Time	~2 sec	~4.5 hrs	~1 min	>20 hrs
	Memory Efficiency	$O(4HW)$	$O(4HW)$	$O(KHW)$	-
Strahler Stream Order	Computation Time	~10–14 sec	~4 hrs	~40 sec	>15 hrs
	Memory Efficiency	$O(4HW)$	$O(4HW)$	$O(KHW)$	-
Micro-Watershed Delineation	Threshold	10000			
	Computation Time	~2–2.5 sec	~2 hrs	~50 sec	>6 hrs
	Memory Efficiency	$O(CHW)$	-	$O(KHW)$	-
Runoff Simulation	First Iteration Time	~8 sec	~30 min	~2 min	>20 hrs
	3-Month Session Time	~3 min	~14 hrs	~25 min	Not computable
	Memory per Run	$O(CHW)$	-	$O(KHW)$	-

product for this purpose but with improvements in remote sensing coverage and modeling, higher resolution global DEMs are likely to become available too. The algorithms will need to be enhanced in that case to handle loading/unloading of multiple tiles in case the whole region is too large to fit into the GPU memory in one go. LP operations will become more complex to handle in that case.

Second, we see that as larger datasets are used, GPU-based methods give lesser speed-ups over CPU-based methods (the 2.6GB LGB dataset gave a 5x speed-up whereas the 5MB Masalia region gave a 20x speed-up). The amount of GPU speedup depends on memory access patterns and workload irregularity. The smaller Masalia dataset fits in the GPU's fast L2 cache, enabling high speedup. The larger LGB dataset exceeds the cache size, forcing reliance on slower VRAM. Further, the topography governs the number of iterations required in several LP-based operations, E.g. flow accumulation requires iterations equal to the length of the longest path. Larger datasets would typically have longer and more diverse flow paths. Finally, temporary arrays need to be carefully handled by freeing or reusing memory when not in use. More detailed benchmarking across datasets of different sizes and complexity can help improve the implementation.

Third, our current implementation is based on the CUDA architecture. Being able to convert LP operations into matrix operations can help us leverage Tensorflow for platform independence. We did make some initial attempts but the memory demand of Tensorflow was higher than that of Cupy. However, we believe that further investigation and memory optimization can solve this problem.

Fourth, being able to standardize algorithm implementations and dataset formats can enable sharing and collaboration at a global scale while using entirely local on-prem compute infrastructure. This can open up new pathways for planetary computing to be done in a decentralized manner, potentially anchored at academic institutions, with little or no dependence on commercial cloud infrastructure. The

computation can be further customized to suit local requirements such as to adjust time-steps according to the desired accuracy, based on whether real-time or offline results are required, and thereby provide greater flexibility closer to the point of service.

Acknowledgments

We sincerely thank Tower Research for providing funding to support this project, and the alumni fund of the Department of Computer Science and Engineering for providing travel support to present the paper.

Algorithm 1 Flow Accumulation - CPU Approach

```

1: Input: DEM: Digital Elevation Model raster
2: Output: ACC: Flow accumulation raster
3: procedure FlowAccum_CPU(DEM)
4:   for all cell  $(i, j)$  in DEM do
5:     Find neighbor with steepest downhill slope
6:     Assign flow direction value to  $(i, j)$  according to that neighbor direction.
7:   end for
8:   for all cell  $(i, j)$  do
9:      $ACC[i][j] \leftarrow 1$ 
10:  end for
11:  for all cell  $(i, j)$  do
12:    Compute  $IN\_DEGREE[i][j]$ : number of upstream cells flowing into each cell
13:  end for
14:   $QUEUE \leftarrow$  list of cells with  $IN\_DEGREE = 0$ 
15:  while  $QUEUE$  is not empty do
16:     $cell \leftarrow QUEUE.pop()$ 
17:     $downstream \leftarrow FD[cell]$ 
18:    if  $downstream$  exists then
19:       $ACC[downstream] \leftarrow ACC[downstream] + ACC[cell]$ 
20:       $IN\_DEGREE[downstream] \leftarrow IN\_DEGREE[downstream] - 1$ 
21:      if  $IN\_DEGREE[downstream] = 0$  then
22:         $QUEUE.push(downstream)$ 
23:      end if
24:    end if
25:  end while
26:  return  $ACC$ 
27: end procedure

```

► Compute Flow Direction (FD)

► Initialize Accumulation Grid

► Each cell counts itself initially for accumulation.

► Initialize Topological Sort

► Propagate Accumulation

Algorithm 2 Flow Accumulation - GPU

```

1: Input: DEM: Digital Elevation Model raster
2: Output: ACC: Flow accumulation raster
3: procedure FLOWACCUM(DEM)
    ▶ here i, j means, same operation will perform in Parallel on every (i, j) cell
4:   Initialize  $FLOW\_DIR[i][j]$ 
5:   Initialize  $in\_deg[i][j] \leftarrow 0$ 
6:   Initialize  $acc[i][j] \leftarrow 1$ 
7:   Initialize  $active[i][j] \leftarrow 0$ 
8:   Initialize  $next\_active[i][j] \leftarrow 0$ 
    ▶ Assign the direction value according to slope of DEM
    ▶ Stores number of upstream contributors
    ▶ Each cell starts with 1 unit of flow
    ▶ All these assignment operations are (PI)

9:   KERNEL_COMPUTEINDEGREE( $flow\_dir, in\_deg$ )
    ▶ SP operation

10:  KERNEL_INITACTIVE( $in\_deg, active$ )
    ▶ SP operation

11:  while any cell in  $active$  is 1 do
12:    Reset  $next\_active \leftarrow 0$ 
13:    KERNEL_PROPAGATEFLOW( $flow\_dir, acc, in\_deg, active, next\_active$ )
14:    Swap  $active \leftrightarrow next\_active$ 
15:  end while
16:  return  $acc$ 
17: end procedure

```

Algorithm 3 GPU Kernel: ComputeInDegree

```

1: procedure KERNEL_COMPUTEINDEGREE( $flow\_dir, in\_deg$ )
2:   Parallel do for all  $(i, j) \in ACTIVE$ :
3:      $dir \leftarrow flow\_dir[x][y]$ 
4:     If  $dir$  is valid (1 to 8)
5:       Compute target cell  $(tx, ty)$  from  $dir$ 
6:       Atomically increment  $in\_deg[tx][ty]$ 
7:   end procedure

```

Algorithm 4 GPU Kernel: InitActive

```

1: procedure KERNEL_INITACTIVE( $in\_deg, active$ )
2:   Parallel do for all  $(i, j) \in ACTIVE$ :
3:     If  $in\_deg[i] = 0$ 
4:        $active[i] \leftarrow 1$ 
5:   end procedure

```

Algorithm 5 GPU Kernel: STREAM CELLS

```

1: procedure KERNEL_STREAM_CELL( $flow\_accum, threshold, stream\_cells$ )
2:   Parallel do for all  $(i, j) \in ACTIVE$ :
3:     If  $flow\_accum[i] \geq threshold$ 
4:        $stream\_cells[i] \leftarrow TRUE$ 
5:   end procedure

```

Algorithm 6 GPU Kernel: PropagateFlow

```

1: procedure KERNEL_PROPAGATEFLOW(flow_dir, acc, in_deg, active, next_active)
2:   Parallel do for all  $(i, j) \in \text{ACTIVE}$  where  $\text{active}[i][j] = 1$ :
3:      $\text{dir} \leftarrow \text{flow\_dir}[i][j]$ 
4:     If dir is valid
5:       Compute  $(tx, ty)$  downstream target
6:       Atomic add  $\text{acc}[i][j]$  to  $\text{acc}[tx][ty]$ 
7:        $\text{prev} \leftarrow$  Atomic decrement  $\text{in\_deg}[tx][ty]$ 
8:       If  $\text{prev} = 1$ :
9:          $\text{next\_active}[tx][ty] \leftarrow 1$ 
10:  end procedure

```

Algorithm 7 Strahler Stream Order Computation

```

Input: FLOW_DIR, FLOW_ACCUM, THRESHOLD
Output: STREAM_ORDER raster
1: procedure COMPUTE_STREAM_ORDER(FLOW_DIR, FLOW_ACCUM, THRESHOLD)
2:   Initialize  $\text{IN\_DEGREE}[i][j] \leftarrow 0$  for all cells
3:   Initialize  $\text{STREAM\_ORDER}[i][j] \leftarrow 0$  for all cells
4:   Initialize  $\text{STREAM\_CELLS}[i][j] \leftarrow \text{FALSE}$ 

5:   KERNEL_STREAM_CELL(FLOW_ACCUM, THRESHOLD, STREAM_CELLS)

6:   KERNEL_COMPUTEINDEGREE(FLOW_DIR, IN_DEGREE)

7:   Initialize  $\text{STREAM\_ORDER}[i][j] \leftarrow 1$  where  $\text{IN\_DEGREE}[i][j] = 0$  and  $\text{STREAM\_CELLS}[i][j] = \text{TRUE}$ 
8:   ACTIVE  $\leftarrow$  cells where  $\text{STREAM\_ORDER} = 1$ 

9:   while any ACTIVE cells exist do
10:    for all D8 directions do
11:      Parallel do for all  $(i, j) \in \text{ACTIVE}$ :
12:         $\text{target} \leftarrow$  downstream neighbor of  $(i, j)$  using FLOW_DIR
13:         $\text{incoming\_order} \leftarrow \text{STREAM\_ORDER}[i][j]$ 
14:         $\text{MAX\_UPSTREAM}[\text{target}] \leftarrow \max\{\text{MAX\_UPSTREAM}[\text{target}], \text{incoming\_order}\}$ 
15:        Increment  $\text{MAX\_COUNT}[\text{target}]$  if  $\text{incoming\_order} = \text{MAX\_UPSTREAM}[\text{target}]$ 
16:        Mark target as reached
17:      end for
18:    for all  $(i, j) \in \text{ACTIVE}$  (in parallel) do
19:      if all upstreams have been reached then
20:        if multiple upstreams with same max stream order then
21:           $\text{STREAM\_ORDER}[i][j] \leftarrow \text{MAX\_UPSTREAM} + 1$ 
22:        else
23:           $\text{STREAM\_ORDER}[i][j] \leftarrow \text{MAX\_UPSTREAM}$ 
24:        end if
25:        Mark  $(i, j)$  as next ACTIVE cell
26:      end if
27:    end for
28:  end while
29:  return STREAM_ORDER
30: end procedure

```

Algorithm 8 Shreve Stream Order (GPU-based)

```

1: Input: Flow direction, flow accumulation, threshold
2: Output: Stream order raster
3: procedure SHREVEORDER(FLOW_DIR, FLOW_ACCUM, THRESHOLD)
4:    $H, W \leftarrow \text{shape of } FLOW\_DIR$                                 ▶ Get raster dimensions
5:   Initialize  $ACC[i][j] \leftarrow 0$                                 ▶ Initialize accumulation grid
6:   Initialize  $CURRENT[i][j] \leftarrow 0$                             ▶ Current active cell buffer
7:   Initialize  $NEXT[i][j] \leftarrow 0$                                 ▶ Next iteration buffer
8:   Initialize  $ACTIVE[i][j] \leftarrow \text{False}$                     ▶ Mask for active stream cells

9:   KERNEL_STREAM_CELL(FLOW_ACCUM, THRESHOLD, ACTIVE)            ▶ Identify stream cells above threshold

10:   $CURRENT[ACTIVE] \leftarrow 1$ ,  $STREAM\_ORDER[ACTIVE] \leftarrow 1$     ▶ Mark initial active cells with stream order 1

11:  while any cell in CURRENT is active do                        ▶ Loop until no active cells remain
12:    SHREVE_KERNEL(FLOW_DIR, CURRENT, STREAM_ORDER, NEXT, H, W)
13:    Swap  $CURRENT \leftrightarrow NEXT$                                     ▶ Move to next set of active cells
14:  end while
15:  return STREAM_ORDER                                            ▶ Final stream order result
16: end procedure

17: function SHREVE_KERNEL(FLOW_DIR, CURRENT, STREAM_ORDER, NEXT, H, W)
18:  for all cell (i, j) where  $CURRENT[i][j] = 1$  in parallel do    ▶ GPU-parallel per active cell
19:    (di, dj)  $\leftarrow$  downstream cell from  $FLOW\_DIR[i][j]$         ▶ Find flow destination
20:     $STREAM\_ORDER[di][dj] \leftarrow STREAM\_ORDER[di][dj] + STREAM\_ORDER[i][j]$ 
21:     $NEXT[di][dj] \leftarrow 1$                                        ▶ Activate downstream cell for next step
22:  end for
23: end function

```

Algorithm 9 Micro-Watershed Delineation (GPU)

```

1: Input: FLOW_DIRECTION, FLOW_ACCUMULATION, THRESHOLD
2: Output: WATERSHEDS raster
3: procedure MICRO_WATERSHED(FLOW_DIRECTION, THRESHOLD, FLOW_ACCUMULATION)
4:   Initialize WATERSHEDS  $\leftarrow$  0
5:   Initialize ASSIGNED_MASK  $\leftarrow$  False
6:   Initialize INCOMING_COUNT  $\leftarrow$  0
7:   Initialize STREAM_MASK  $\leftarrow$  False

8:   KERNEL_STREAM_CELL(FLOW_ACCUMULATION, THRESHOLD, STREAM_MASK)
9:   KERNEL_COMPUTE_INDEGREE(FLOW_DIRECTION, INCOMING_COUNT)
10:  For cells where STREAM_MASK is false,  $INCOMING\_COUNT \leftarrow 0$ 

11:  Define CONVERGENCE_POINTS  $\leftarrow$  cells where  $INCOMING\_COUNT > 1$  and STREAM_MASK is true
12:  Initialize POTENTIAL_OUTLETS  $\leftarrow$  False

13:  KERNEL_IDENTIFYOUTLETS(FLOW_DIRECTION, CONVERGENCE_POINTS, WATERSHEDS, ASSIGNED_MASK, POTENTIAL_OUTLETS)

14:  Initialize PROPAGATING_MASK  $\leftarrow$  POTENTIAL_OUTLETS

15:  while any cell in PROPAGATING_MASK is true do
16:    Initialize NEXT_PROPAGATING_MASK  $\leftarrow$  False
17:    for each direction  $(dr, dc)$  in D8 do
18:       $d \leftarrow$  direction code for offset  $(dr, dc)$ 
19:      Identify upstream cells  $(r, c)$  where  $FLOW\_DIRECTION[r][c] = d$ 
20:      Compute downstream cell  $(r + dr, c + dc)$ 
21:      if downstream cell is in bounds and in PROPAGATING_MASK then
22:        if ASSIGNED_MASK $[r][c]$  is false then
23:          WATERSHEDS $[r][c] \leftarrow$  WATERSHEDS $[r + dr][c + dc]$ 
24:          ASSIGNED_MASK $[r][c] \leftarrow$  true
25:          NEXT_PROPAGATING_MASK $[r][c] \leftarrow$  true
26:        end if
27:      end if
28:    end for
29:    PROPAGATING_MASK  $\leftarrow$  NEXT_PROPAGATING_MASK
30:  end while
31:  return WATERSHEDS
32: end procedure

```

Algorithm 10 Kernel to Identify Potential Outlets from Stream Convergence

```

1: procedure KERNEL_IDENTIFYOUTLETS(FLOW_DIRECTION, CONVERGENCE_POINTS, WATERSHEDS, AS-
   SIGNED_MASK, POTENTIAL_OUTLETS)
2:   for each cell  $(r, c)$  do
3:      $d \leftarrow \text{FLOW\_DIRECTION}[r][c]$ 
4:      $(dr, dc) \leftarrow$  offset corresponding to direction  $d$ 
5:      $(r', c') \leftarrow (r + dr, c + dc)$ 
6:     if  $(r', c')$  is within bounds then
7:       if CONVERGENCE_POINTS[ $r'$ ][ $c'$ ] is true then
8:         POTENTIAL_OUTLETS[ $r$ ][ $c$ ]  $\leftarrow$  true
9:         WATERSHEDS[ $r$ ][ $c$ ]  $\leftarrow r \times W + c + 1$ 
10:        ASSIGNED_MASK[ $r$ ][ $c$ ]  $\leftarrow$  true
11:      end if
12:    end if
13:  end for
14: end procedure

```

Algorithm 11 Runoff Simulation Using Jump Matrix - GPU

```

1: Input: FLOW_DIR, DEM, SOIL, LULC, RAINFALL SERIES raters
2: Output: Runoff rasters of time series will save to disk
3: procedure RUNOFFSIMULATION(FLOW_DIR, DEM, SOIL, LULC, RAINFALL_SERIES)
4:   Define pixel resolution RES  $\leftarrow 30m$ 
5:   Define per-pixel velocity map using slope of digital elevation model ▷ SP
6:   Define time threshold  $T_{th} \leftarrow 3$  Hour
7:   JUMP_MATRIX  $\leftarrow \text{JumpMatrix}(\text{FLOW\_DIR}, \text{RES}, T_{th}, \text{velocity})$  ▷ LP
8:   Save JUMP_MATRIX to disk for reuse

9:   Initialize ROUTED_RO  $\leftarrow \text{NULL}$ 
10:  for all  $t$  in RAINFALL_SERIES do
11:    Load rainfall images  $R_{t-4}$  to  $R_t$  and convert to CuPy ▷ SP
12:     $P \leftarrow R_t + R_{t-1}$  ▷ SP
13:     $P5 \leftarrow \sum_{i=t-4}^t R_i$  ▷ SP
14:    if previous ROUTED_RO exists then
15:       $P \leftarrow P + \text{ROUTED\_RO}$  ▷ SP
16:       $P5 \leftarrow P5 + \text{ROUTED\_RO}$  ▷ SP
17:    end if
18:    // Runoff generation using input parameters ▷ SP-heavy
19:    RUNOFF  $\leftarrow \text{RunoffGeneration}(P, P5, \text{SOIL}, \text{LULC})$ 
20:    Save current RUNOFF to disk
21:    // Route runoff downstream using Jump Matrix ▷ SP
22:    ROUTED_RO  $\leftarrow \text{Route}(\text{RUNOFF}, \text{JUMP\_MATRIX})$ 
23:    Deallocate intermediate rainfall arrays to save memory
24:  end for
25: end procedure

```

Algorithm 12 Jump Matrix Computation - GPU

```

1: Input: FLOW_DIR, VELOCITY, RESOLUTION, THRESHOLD rasters
2: Output: JUMP_MATRIX raster
3: procedure JUMP_MATRIX(FLOW_DIR, VELOCITY, RESOLUTION, THRESHOLD)

4:   Initialize TIME  $\leftarrow$  0 for all cells
5:   Initialize JUMP_MATRIX with row-major indices of each cell
6:   VALID_CELLS  $\leftarrow$  cells where VELOCITY is not null

7:   while any VALID_CELLS have TIME < THRESHOLD do
8:     ACTIVE_CELLS  $\leftarrow$  VALID_CELLS where TIME < THRESHOLD
9:     for all direction  $d$  in D8 do
10:      NG_CELLS  $\leftarrow$  downstream neighbors of ACTIVE_CELLS with FLOW_DIR =  $d$ 
11:      Update TIME[ACTIVE_CELLS with dir =  $d$ ]  $\leftarrow$  TIME + RESOLUTION/VELOCITY
12:      Update JUMP_MATRIX[ACTIVE_CELLS with dir =  $d$ ]  $\leftarrow$  NG_CELLS
13:    end for
14:  end while
15:  return JUMP_MATRIX
16: end procedure

```

References

- [1] Esri. 2025. How Flow Accumulation works ArcGIS Pro | Documentation. <https://pro.arcgis.com/en/pro-app/latest/tool-reference/spatial-analyst/how-flow-accumulation-works.htm>. Accessed: 2025-06-18.
- [2] GDAL Developers. 2025. Multi-threading — GDAL documentation. <https://gdal.org/en/stable/user/multithreading.html>. Accessed: 2025-08-19.
- [3] GIS StackExchange. 2019. Calculating flow accumulation using Google Earth Engine. <https://gis.stackexchange.com/questions/279793/calculating-flow-accumulation-using-google-earth-engine>. Accessed: 2025-06-18.
- [4] Google Earth Engine and World Resources Institute. 2021. Dynamic World V1: Near Real-Time Land Use/Land Cover Dataset. https://developers.google.com/earth-engine/datasets/catalog/GOOGLE_DYNAMICWORLD_V1. Accessed: 2025-08-19.
- [5] Noel Gorelick, Matt Hancher, Mike Dixon, Simon Ilyushchenko, David Thau, and Rebecca Moore. 2017. Google Earth Engine: Planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment* 202 (2017). doi:10.1016/j.rse.2017.06.031
- [6] Andrew Goudie and David Brunsden. 2019. *Fundamentals of Hydrology* (2 ed.). Routledge. https://students.aiu.edu/submissions/profiles/resources/onlineBook/B9P2U2_Fundamentals%20of%20Hydrology-2019.pdf Accessed: 2025-08-17.
- [7] India-WRIS. 2021. INDIA WRIS WIKI. <https://indiawriss.gov.in/wiki/doku.php?id=india-wris>. Accessed: 2025-08-19.
- [8] India WRIS. 2025. River Basins of India – India-WRIS Wiki. <https://indiawriss.gov.in/wiki/doku.php?id=basins>. Accessed: 2025-08-19.
- [9] Vikrant Maurya and Manika Gupta. 2024. Advancements and Challenges in Hydrological Modelling. *International Journal for Research in Applied Science & Engineering Technology (IJRASET)* 12, 11 (November 2024). doi:10.22214/ijraaset.2024.65161
- [10] Shivani A Mehta et al. 2025. CoRE stack: Commoning for Resilience and Equality - Geospatial data layers. <https://www.cse.iitd.ernet.in/~aseth/core-stack-layers-mar-2025.pdf>.
- [11] NASA/CGIAR. 2000. SRTM Digital Elevation Data Version 4. https://developers.google.com/earth-engine/datasets/catalog/CGIAR_SRTM90_V4. Accessed: 2025-08-19.
- [12] J. F. O'Callaghan and D. M. Mark. 1984. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics, and Image Processing* 28, 3 (1984). doi:10.1016/S0734-189X(84)80011-0
- [13] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. http://learningsys.org/nips17/assets/papers/paper_16.pdf
- [14] Sushil K. Prasad, Michael McDermott, Satish Puri, Dhara Shah, Danial Aghajarian, Shashi Shekhar, and Xun Zhou. 2015. A vision for GPU-accelerated parallel computation on geo-spatial datasets. *SIGSPATIAL Special* (2015). doi:10.1145/2766196.2766200
- [15] C.W. Ross, L. Prihodko, J.Y. Anchang, S.S. Kumar, W. Ji, and N.P. Hanan. 2018. Global Hydrologic Soil Groups (HYSOGs250m) for Curve Number-Based Runoff Modeling. *Scientific Data* 5 (2018), 180091. doi:10.1038/sdata.2018.91
- [16] S. Saupi Teri, I. A. Musliman, and A. Abdul Rahman. 2022. GPU Utilization in Geoprocessing Big Geodata: A Review. In *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*. doi:10.5194/isprs-archives-XLVI-4-W3-2021-295-2022
- [17] VillageInfo.org. 2025. Masalia Subdivision of Dumka, Jharkhand | Population, Area, Villages, List of Villages. <https://villageinfo.org/sub-district/masalia>. Accessed: 2025-08-19.
- [18] Wikipedia contributors. 2025. Manning formula. https://en.wikipedia.org/wiki/Manning_formula. Accessed: 2025-08-19.
- [19] Guiyun Zhou, Hongqiang Wei, and Suhua Fu. 2019. A fast and simple algorithm for calculating flow accumulation matrices from raster digital elevation. *Frontiers of Earth Science* 13 (2019). doi:10.1007/s11707-018-0725-9

Received 2025-07-02; accepted 2025-08-11