Highlights

The multi-GPU Wetland DEM Ponding Model

Tonghe Liu,Sean J. Trim,Seok-Bum Ko,Raymond J. Spiteri

- `WDPM` simulates how runoff water is distributed across the Canadian Prairies
- `WDPM` has been generalized to take advantage of multiple GPUs
- A speedup of 2.39 has been observed using 4 GPUs with a real dataset
- By using leading computing paradigms, `WDPM` can perform difficult simulations

The multi-GPU Wetland DEM Ponding Model

Tonghe Liu[a], Sean J. Trim[a], Seok-Bum Ko[b] and Raymond J. Spiteri[a,*]

[a]Department of Computer Science, University of Saskatchewan, 110 Science Place, Saskatoon, S7N 5C9, SK, Canada
[b]Department of Electrical and Computer Engineering, University of Saskatchewan, 57 Campus Drive, Saskatoon, S7N 5A9, SK, Canada

ARTICLE INFO

Keywords:
Wetland DEM Ponding Model
scalability
multi-GPU

ABSTRACT

The Wetland DEM (Digital Elevation Model) Ponding Model (WDPM) is software to simulate how runoff water is distributed across the Canadian Prairies. Previous versions of the WDPM are able to run in parallel with a single CPU or GPU. Now that multi-device parallel computing has become an established method to increase computational throughput and efficiency, this study extends WDPM to a multi-GPU parallel algorithm with efficient data transmission methods via overlapping communication with computation. The new implementation is evaluated from several perspectives. First, the output summary and system are compared with the previous implementation to verify correctness and demonstrate convergence. Second, the multi-GPU code is profiled, showing that the algorithm carry out efficient data synchronization through optimized techniques. Finally, the new implementation was tested experimentally and showed improved performance and good scaling. Specifically, a 2.39 times speedup was achieved when using four GPUs compared to using one GPU.

1. Introduction

Graphic processing units (GPUs) are prevalent and effective platforms for executing general purpose parallel applications. The availability of various GPU programming systems, such as CUDA, OpenACC, and OpenCL, enables programmers to parallelize an application using thousands of threads. This parallelization can result in significant performance improvements across various fields, including physics (e.g., ANSYS, which simulates the interaction of liquids and gases with surfaces (Krawezik and Poole, 2009)), biological sciences (e.g., Basic Local Alignment Search Tool, which is one of the most widely used bioinformatics tools (Vouzis and Sahinidis, 2011)), chemistry (e.g., Vienna Ab-initio Simulation Package, which is used for quantum mechanics and molecular dynamics simulation (Hacene, Anciaux-Sedrakian, Rozanska, Klahr, Guignon and Fleurat-Lessard, 2012)), and weather forecasting (e.g., The Weather Research and Forecasting Model, a numerical weather forecast system (Michalakes and Vachharajani, 2008)). However, achieving peak GPU performance requires significant development because programmers must optimize the code to align with the characteristics of the underlying GPU hardware. Although hardware-specific algorithms can yield promising results, they can reduce the application's portability, and their implementation requires significant hardware knowledge. In this case, instead of designing an application that perfectly fits a particular GPU hardware configuration, increasing the degree of parallelism is a promising approach (Narasiman, Shebanow, Lee, Miftakhutdinov, Mutlu and Patt, 2011). This study demonstrates how to achieve optimal performance of the Wetland DEM (Digital Elevation Model) Ponding Model (WDPM) through the effective use of multiple GPUs.

1.1. Motivation

The WDPM is a tool developed by the Centre for Hydrology at the University of Saskatchewan to model the distribution of runoff water on the Canadian Prairies. This model plays a crucial role in determining the fraction of Prairie basins contributing flows to streams while dynamically changing with water storage in depressions. Furthermore, the model has been used to demonstrate the extent of flooding on Prairie landscapes (Kevin, Armstrong, Sharomi, Spiteri and Pomeroy, 2014). The program was initially developed in Fortran and later adapted to CPU parallel computing with OpenMP. Subsequently, the program was translated to C, and parallel computation was implemented using OpenCL, a cross-platform parallel programming language. With OpenCL, the WDPM achieved significant performance improvements and can now run efficiently on a GPU. For example, an exceptionally large simulation that originally took a

*Corresponding author
Email addresses: tonghe.liu@usask.ca (T. Liu); sean.trim@usask.ca (S.J. Trim); seokbum.ko@usask.ca (S. Ko); raymond.spiteri@usask.ca (R.J. Spiteri)
ORCID(s): 0000-0002-2714-3103 (T. Liu); 0000-0001-5076-0620 (S.J. Trim); 0000-0002-9287-317X (S. Ko); 0000-0002-3513-6237 (R.J. Spiteri)

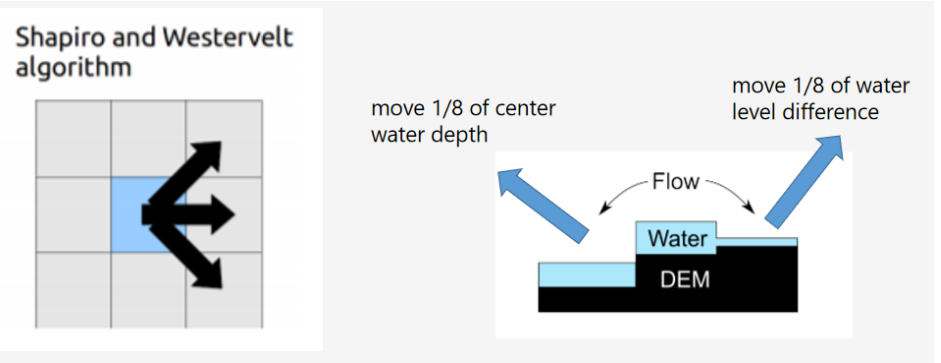

Figure 1: SW algorithm.

year to complete now takes only seven weeks. This significant performance improvement enables researchers to solve operations-scale problems in reasonable times.

This study investigates how the WDPM can be correctly and efficiently executed using multiple GPUs in parallel. It applies various general GPU programming optimization methods. However, as mentioned, hardware tuning is not the focus of this work. Instead, performance improvement is mainly embodied by the application of multi-GPU parallelism. The study aims to explore the degree of parallelism required to optimize performance in multi-GPU systems, investigate techniques to optimize communication between multiple GPUs, and evaluate the effectiveness of different parallelization strategies in multi-GPU computing systems.

1.2. Contributions

This study presents an efficient multi-GPU parallel algorithm for the WDPM by extending the existing codebase (WDPM version 1.0 (Kevin et al., 2014)) that previously allowed simulations to be run in parallel with one GPU or one CPU. A new main loop is introduced to enable the application to work with a host containing multiple GPUs, and an advanced data synchronization method is developed to minimize overhead and enhance performance.

The proposed multi-GPU parallel algorithm is based on CPU and single-GPU parallel methods that were previously applied to WDPM, and its correctness is confirmed by verifying the output summaries and errors of multi-GPU output maps. Additionally, an efficient data synchronization method is proposed to overlap data communication with computation. Experimental evaluations are performed to investigate the performance of the new implementation by solving problems of various sizes with different numbers of GPUs (up to four). For an approximate system size of 2500×3000, the maximum speedup is 1.61 when using 2 GPUs. For an approximate system size of 4500×4500, the maximum speedup is 1.92 when using 3 GPUs. Finally, for an approximate system size of 6000×6000, the maximum speedup is 2.39 when using 4 GPUs. These findings demonstrate the significant performance improvement of the new implementation and robust strong scaling characteristics. Therefore, it is expected that the performance improvement will be more pronounced when working with even larger systems.

2. Methods

2.1. The Shapiro–Westervelt algorithm

The WDPM is a fully distributed model of wetland storage and runoff that was described by Shook and Pomeroy (Shook and Pomeroy, 2011). The model finds the final spatial distribution of excess precipitation (water) evenly applied over a LiDAR-based DEM using the Shapiro–Westervelt (SW) iterative algorithm (Shapiro and Westervelt, 1992). Different from the D8 algorithm (**?**), the SW algorithm allows water in the center cell to flow to its eight neighbors. Figure 1 visually shows this water redistribution method, where "DEM" refers to the ground elevation and "WATER" refers to the water depth.

The SW algorithm, as described in Algorithm 1 using the notation in Table 1, is applied in the wetland storage and runoff model WDPM. The input system contains a non-observation area outside the observation region that is excluded from the computation in WDPM. If a center or neighbor cell is a non-observation (or *missing*) point, no computation is required.

Despite the imperfection of the algorithm in each iteration due to potential inaccuracies in water depth transfer, thousands of iterations result in a realistic water surface (Kevin et al., 2014). Convergence of the simulation is determined by the user-specified elevation tolerance in WDPM. The algorithm runs for 1000 iterations and calculates the maximum water level difference between two consecutive iterations. If this value exceeds the user-set elevation tolerance, the algorithm continues for another 1000 iterations. The algorithm terminates when the maximum change of the water level is smaller than the elevation tolerance (Armstrong, Kayter, Shook and Hill, 2013). Although the SW algorithm is less computationally efficient than other hydrology applications due to its slow convergence, the WDPM offers the advantage of out-of-order calculations, making it suitable for parallel computing.

Table 1
Notation used in algorithm 1.

Notation	Meaning
N_0	The center cell
N_i	The eight neighbor cells
E_{wc}	The center cell water elevation
E_{wni}	The neighbor cell water elevation
E_{dc}	The center cell DEM elevation
D_{wc}	The center cell water depth

Algorithm 1 SW algorithm

```
if N_0 ≠ missing then
  for N_i from N_1 to N_8 do
    if (E_wc > E_wni) AND (N_i ≠ missing) then
      if E_dc > E_wni then
        move (1/8) D_wc to N_i
      else
        move (1/8)(E_wc − E_wn) to N_i
      end if
    end if
  end for
end if
```

2.2. Parallel programming on one GPU with OpenCL

The WDPM was originally developed in Fortran using OpenMP to enable parallel processing across multiple CPU cores. The parallel method involves dividing the domain into slices, where each slice is processed with one CPU core. Because a large proportion of data within each slice is independent and does not read or write to other slices, race conditions can be easily avoided by separately handling the boundary of each slice. However, GPUs have thousands of cores, and dividing the domain into thousands of slices is not feasible. Furthermore, because each cell is interdependent on its neighbor cells when using the SW algorithm, naively performing the algorithm on two adjacent cells simultaneously results in a race condition. To overcome this issue, the sliding window method is introduced. This method involves moving a window of a specified size over the data and performing computations over the data in the window. The output for each input sample is the computational result of the current sample and the previous samples. In the first time step, the window is filled with non-observational values to compute the first outputs when the window does not have enough data. In subsequent time steps, samples from the previous data frame are used to fill the window. The sliding window size is set to three in the WDPM, and Figure 2 and Figure 3 demonstrate how this method is used. The black cells shown in each figure are computed in parallel. By looping over the sliding window index i and j from 1 to 3, all cells in the input system can execute the SW algorithm without race conditions.

The kernel function that runs on the GPU is given in Algorithm 2. The host function performs the sliding window arrangement (as Figure 2 and Figure 3) and passes the sliding window indices, j and k, to the kernel function.

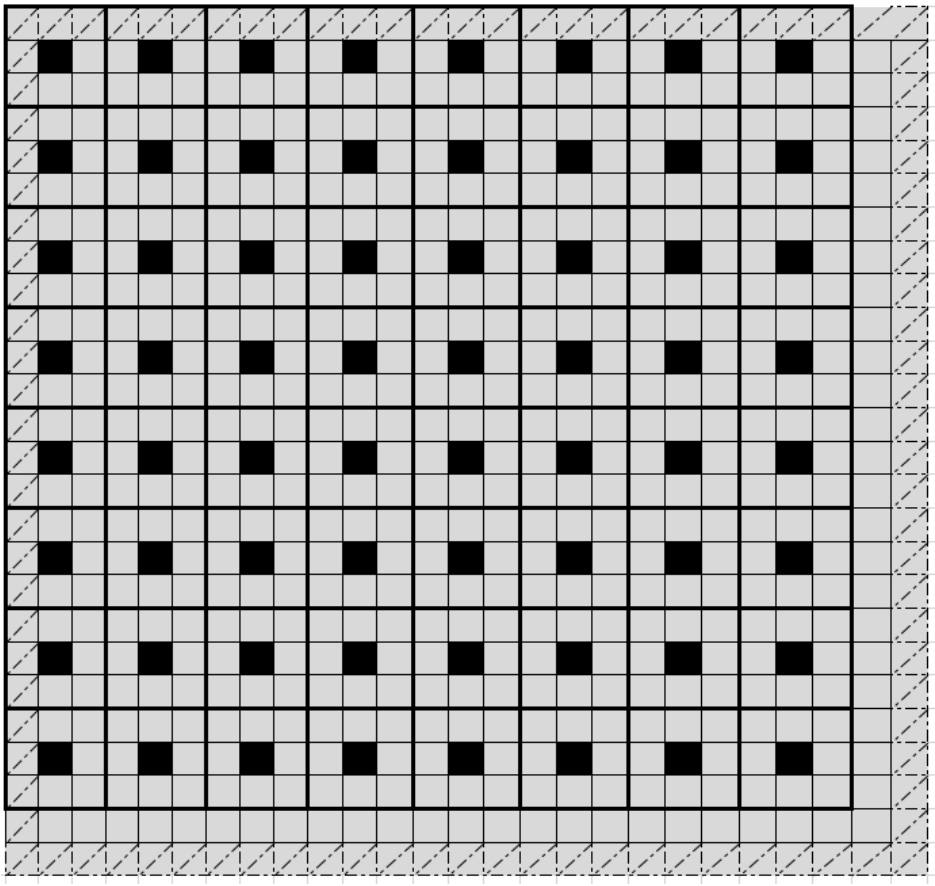

Figure 2: The sliding windows when j=1 and k=1

Algorithm 2 Kernel function

set $row^{'} = global\_id(0)$
set $col^{'} = global\_id(1)$
set $row = (j - S_{\text{window}}) + row^{'} \times S_{\text{window}}$
set $col = (k - S_{\text{window}}) + col^{'} \times S_{\text{window}}$
if $W_{rc} > 0.0$ **then**
  do SW algorithm (Algorithm 1)
end if

2.3. Domain decomposition

Taking inspiration from the CPU parallel computing method, the domain is partitioned into segments that are assigned to separate GPUs for processing. However, it is important to note that the CPU parallel method is based on a shared memory system, which makes data synchronization a relatively straightforward task. In contrast, the CPU-GPU architecture employs a distributed memory system, as illustrated in Figure 4 depicting the OpenCL memory hierarchy diagram. To synchronize data in such a system, the host CPU must retrieve data from the GPUs, and data communication occurs within the host. The updated data are then written back to the GPUs by the host CPU, and the algorithm proceeds to the next iteration.

Performing data transmission can be a time-intensive operation, particularly in the case of `WDPM`, which requires data synchronization in each iteration. With thousands of iterations to be performed in the simulation, it can be estimated that approximately half of the runtime is spent on data synchronization if all data are transferred between devices and the host. This results in longer running times when using two GPUs compared to using a single GPU.

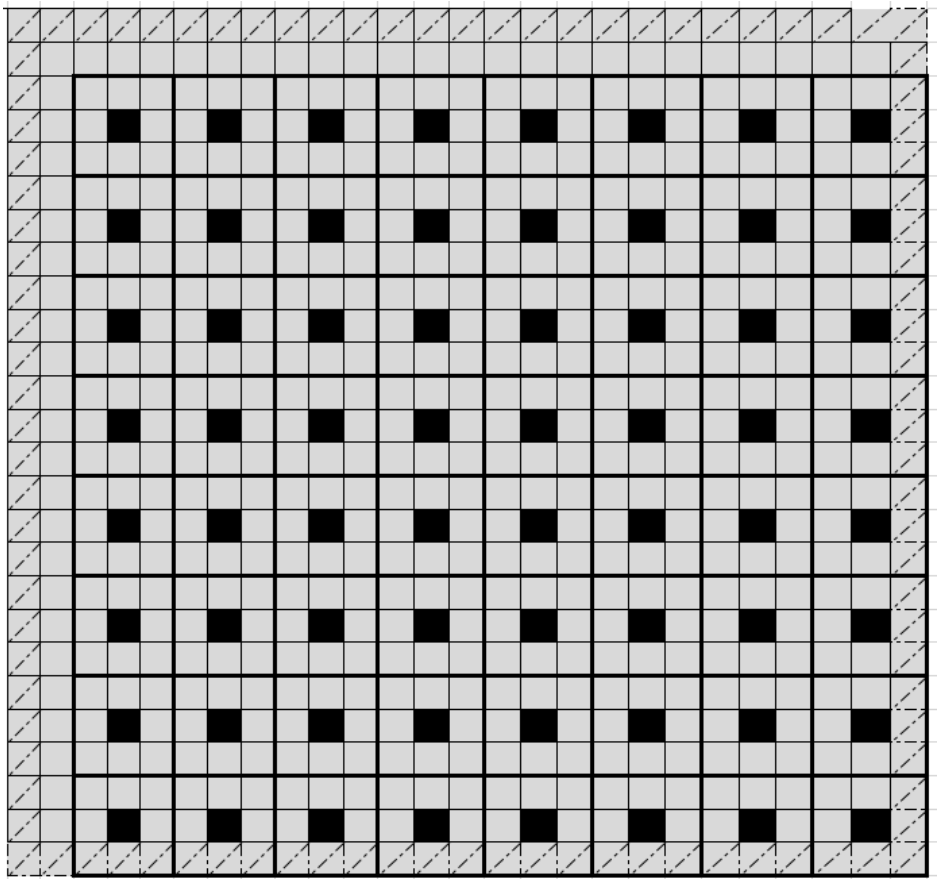

Figure 3: The sliding windows when j=3 and k=3

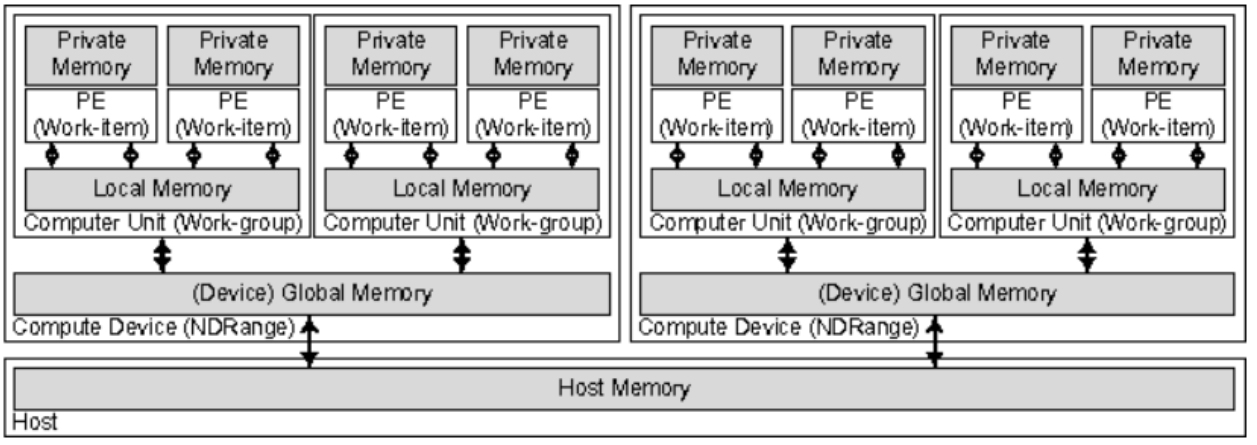

Figure 4: Memory hierarchy: In the image, the host at the bottom is the host CPU, and the two devices above refer to the two GPUs. An NDRange (N-Dimensional Range) is a multitude of kernel instances. arranged into one, two, or three dimensions. A single kernel instance in the index space is called a work-item. Work-group contains a set of work-items, which can synchronize internally using local memory

However, OpenCL provides users with the ability to manually manage data transmission by altering specific OpenCL

function parameters. Additionally, only boundary data require handling during data synchronization. Consequently, by transmitting only boundary data between the host and devices, communication costs can be significantly reduced.

OpenCL includes built-in support for processing image data, which are commonly two-dimensional. The use of image objects enables image data residing in host memory to be made available for processing by a kernel executing on an OpenCL device (Munshi, Gaster, Mattson and Ginsburg, 2011). Image objects offer native support for a wide variety of image formats, thereby simplifying the process of representing and accessing image data (Munshi et al., 2011). However, it is important to note that not all OpenCL devices support image features. To verify image support from the host, the `clGetDeviceInfo` function can be called with the "CL_DEVICE_IMAGE_SUPPORT" option. If the result is "CL_FALSE", then the device does not support images (Scarpino, 2011). Because we aim to ensure `WDPM` works for all OpenCL users, the two-dimensional input system is stored in a one-dimensional buffer.

2.4. Overlapping communication with computation

In data parallel applications, effectively utilizing accelerators is crucial for achieving higher system performance and energy efficiency. However, managing communication between multiple heterogeneous devices is a highly challenging issue because host CPUs and accelerators generally have disjoint memory spaces. This issue is particularly pronounced in large-scale supercomputing applications, where data may not fit into GPU memory, inter-node communication at the end of each time step requires data to be present at the host, or parts of the application must be executed on the CPU. Given the significant investment required to build and maintain supercomputing systems, even a 10-percent improvement in performance can result in substantial resource savings. Thus, it is imperative to minimize the extra time required for communication. To accomplish this, appropriate techniques must be employed to hide communication delays behind the computation performed on heterogeneous devices. In this context, we introduce two techniques that are particularly relevant to the `WDPM`.

1. *Loop indexing*
   In domain decomposition applications, the domain is usually decomposed into inner and outer regions (Hiranandani, Kennedy and Tseng, 1992). Boundary data to be communicated may be produced sooner by updating the outer regions first followed by the inner regions. This technique requires the rearrangement of the loop index. A data dependency analysis is required to ensure correctness.
2. *Loop distribution*
   This technique transfers a single loop into multiple loops by separating independent computation from dependent computation (Hiranandani et al., 1992).

As previously mentioned, the SW algorithm reads and writes to a 3×3 matrix while performing water redistribution. Therefore, each point in the system is dependent on the surrounding points. When performing domain decomposition on the input system, extracting boundary data directly is not feasible. To address this issue, we created an outer domain that includes the boundary data by applying loop distribution. Additionally, loop indexing can be utilized to ensure that the outer domain finishes computation before the inner domain.

In the multi-GPU implementation of the `WDPM`, boundary data are handled serially by the CPU after the GPUs complete their computation. This approach causes GPUs to be idle while the host CPU processes the boundaries. To reduce the communication cost, it is possible to perform CPU computation and GPU computation simultaneously. By doing so, the data communication cost can be reduced.

The implementation of loop indexing in the `WDPM` requires dividing the domain in each GPU into two distinct parts: the inner and the outer domains. The inner domain contains independent data and does not read or write to domains in other GPUs. In contrast, the outer domain contains the boundaries. By managing the loop order, the outer domain of the GPU is computed first, and the host CPU reads the boundary data back afterward. This enables simultaneous computation of the inner part of the GPU and the boundary part of the CPU. However, due to the event schedule mechanism of OpenCL, the data transfer between the CPU and GPUs cannot overlap with any computation.

2.5. Optimized communication

Without loss of generality, consider a system with three GPUs. The original system is sub-divided into three (essentially) equal sub-domains, and one sub-domain is sent respectively to each of the three GPUs.

The black-marked areas in Figure 5 represent the outer parts of the sub-domains. In each iteration, GPUs first compute these outer parts. After this step, the boundary columns (8–11 and 18–21) are no longer being read or written to. The GPU computation is then paused, and the host CPU reads the boundary columns back. As shown in Figure 6,

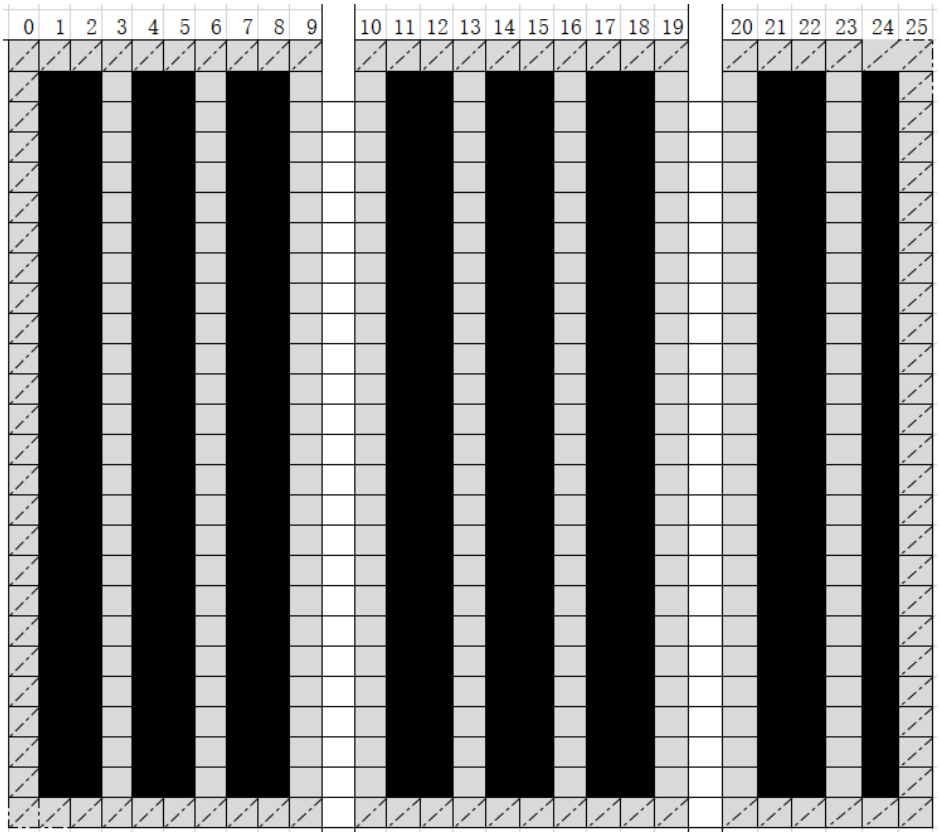

Figure 5: Outer portions of the sub-domains. Outer region comprises the black points. After those points complete their computation, the boundary points are no longer updated in the remaining computation in this iteration.

the boundary columns are merged into boundary matrices that are computed simultaneously with the GPU computation of the inner part. At the end of the iteration, the host separates the boundary matrices and writes the updated boundary columns back to each sub-domain. The complete multi-GPU parallel algorithm is described in Algorithm 3 using the notation defined in Table 2.

Table 2
Notation in the algorithms.

Notation	Meaning	Value in practice
N_{iter}	the number of iterations	1000
S_g	the size of sliding grid	3
N_d	the number of GPUs	-
i_d	the index of the GPU	-
N_r	the number of rows	-
N_{bc}	the number of boundary columns	-

3. Results

The simulations were conducted using the Smith Creek Research Basin (SCRB), located in southeastern Saskatchewan, Canada. The area is relatively flat, with slopes ranging from 2% to 5% and elevations from 490 m to 548 m (Pomeroy, Shook and Dumanski, 2014). The SCRB consists of five sub-basins, with names and system scales given in Table 3. The simulations mainly focus on three sub-basins of different sizes, namely “basin1”, “basin4”, and “basin5”. Addi-

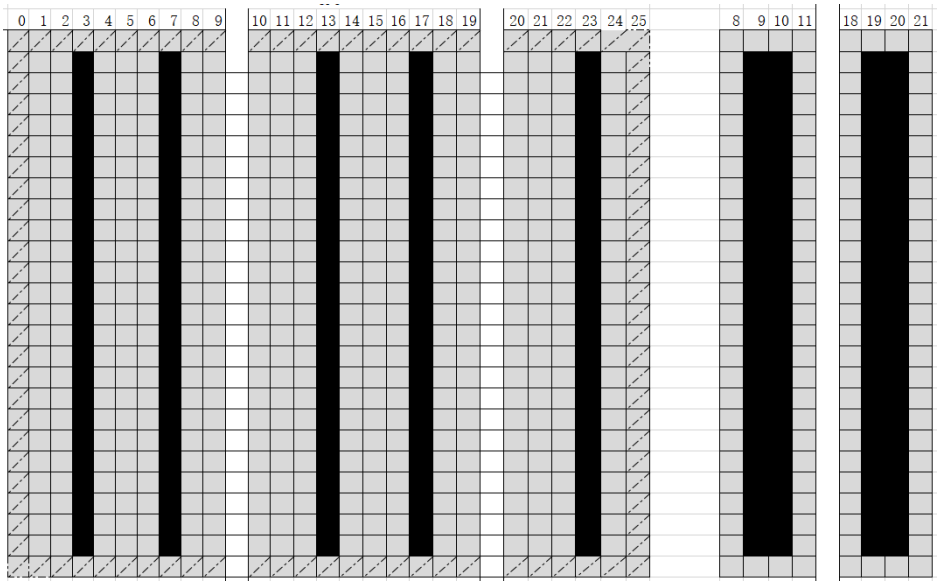

Figure 6: Inner and boundary portions of the sub-domains. Inner region and boundary region comprises black points. Those computations are independent.

tionally, the simulations introduce a DEM system for the area outside the SCRB that is larger than all the systems in the SCRB. The purpose of introducing this larger system is to test the performance improvement using multiple GPUs.

Table 3
Input systems.

System	Size
basin5.asc	482 × 471
basin4_5m.asc	2520 × 1833
culvert_basin1_5m.asc	3794 × 3986
smithcreek_dem1m_sb5.asc	4712 × 4826
patched.asc	5877 × 5519

3.1. Error analysis

We now present the sample systems utilized in the experimental analysis, as outlined in Table 3. The `WDPM` requires input data in the form of a DEM file and a water file, which respectively provide ground elevation and water depth information. The `WDPM` generates a final water distribution map as the output. Before discussing the parallel performance of the `WDPM` multi-GPU implementation, the accuracy of the new implementation is evaluated by comparing the final water distribution with that obtained using one GPU.

Upon completion of a simulation, the `WDPM` generates an output summary that includes information such as the maximum and mean water depth. Table 4 presents sample summaries of the "add" and "drain" modules. Although the multi-GPU implementation produces accurate final results, the intermediate outputs exhibit slight discrepancies.

These discrepancies can be attributed to the order of operations, which can be altered by rotating the input system. Preliminary experiments revealed that different rotation angles yield slightly varying output summaries for the same system. To investigate this phenomenon further, additional simulations were conducted to assess whether the simulations converge as they progress. Specifically, the differences in the final water distribution, resulting from running the simulation with different numbers of GPUs, are evaluated by computing the water depth difference in the output system. As described in Section 2.1, users must specify an elevation tolerance to determine the level of convergence achieved by the simulation. A small water elevation tolerance results in a nearly flat water surface, providing a means to verify the solution's convergence. The test system "basin5.asc" is used in simulations that involve adding 300 mm

Algorithm 3 Overlap communication and computation.

```
continue = true
while continue = true do
  for i from 1 to N_iter do
    if i ≠ 1 then
      Write the boundary data from CPU to GPUs
    end if
    for i_device from 0 to N_device − 1 do
      for j from 1 to S_g − 1 do
        for k from 1 to S_g do
          Write j and k to the kernel function
          Process kernel function (Algorithm 2)
        end for
      end for
    end for
    Block function
    if i ≠ N_iter then
      Read the boundary data from GPUs to CPU
    end if
    for i_d from 0 to N_d − 1 do
      Set k = S_g
      for j from 1 to S_g do
        Write j and k to the kernel function
        Process kernel function (Algorithm 2)
      end for
    end for
    for i from 1 to N_r do
      for j from 1 to N_bc do
        SW (Algorithm 1)
      end for
    end for
    Block function
  end for
  if maximum water elevation change is less than tolerance then
    continue = false
  end if
end while
```

of water to the system and then draining it. The simulations are performed using two GPUs and one GPU. The water depth errors of all pixels are computed, and the error map is plotted in Figures 3.1–3.1. The results reveal robust convergence of the solution when using a small water elevation tolerance. The maximum error is approximately 4×10^{-4} mm with a tolerance of 1.0 mm and about 5×10^{-5} mm with a tolerance of 0.1 mm, further validating the correctness of the multi-GPU implementation.

3.2. Profiling results

OpenCL provides an event-based profiling mechanism by having "cl_event" objects hold timing information. This mechanism was used to capture information using an OpenCL function. We now describe details on the methodology used to obtain profiling results and their subsequent analysis. Computations were performed on the Cedar system of the Digital Alliance of Canada[1] that is equipped with four NVIDIA Tesla P100 GPUs.

[1] https://docs.alliancecan.ca/wiki/Cedar

Table 4
Output summary comparison.

Modules	Add		Drain	
	1 GPU	2 GPUs	1 GPU	2 GPUs
Initial vol (m^3)	0.0	0.0	110036.0	110036.0
Final vol (m^3)	56550681.0	56550681.0	97577.6	97577.6
Water coverage	7.4%	7.4%	10.1%	10.1%
Mean depth (mm)	40.6	40.6	87.4	87.4
Max depth (mm)	1791.4	1791.5	1038.3	1038.2

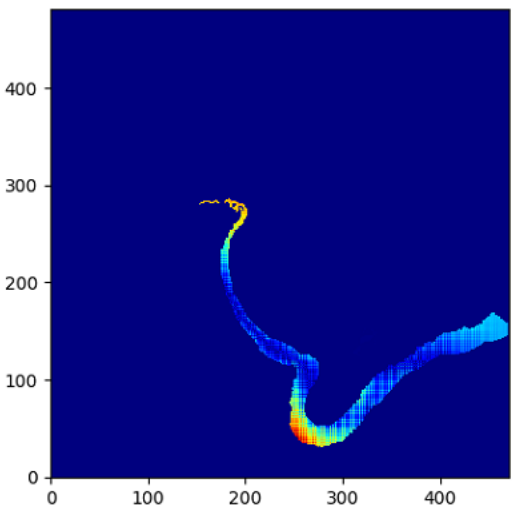

Figure 7: Error map for elevation tolerance 1 mm.

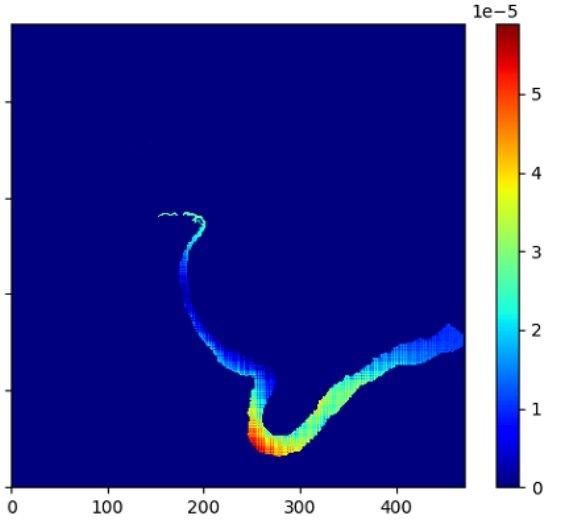

Figure 8: Error map for elevation tolerance 0.1 mm.

3.3. Timing acquisition

The OpenCL programming model involves using a "command queue" to instruct the devices within the specified "context" to execute a specific command. These commands typically include operations such as reading data from the device to the host, writing data from the host to the device, and executing the kernel function. In the mulit-GPU implementation, each device is controlled by its own "command queue". OpenCL also includes an event scheduler that allows programmers to target specific commands for processing. This event scheduler serves two purposes: recording the runtime of a command using the device time counters associated with the specified event, and blocking execution until a specified command finishes using blocking functions such as `clWaitForEvents`. This approach enables testing of the compute time of each segment in a precise manner.

The OpenCL function `clGetEventProfilingInfo` enables developers to measure the execution time of a single OpenCL command. As discussed in Section 2.5, the overlapping communication and computation technique is implemented to enhance the performance of the multi-GPU parallel algorithm. However, the OpenCL profiling function is only capable of timing an individual OpenCL command, necessitating a more comprehensive method to assess both GPU and CPU computations. In Section 2.5, it was explained that the sub-domain in each device is separated into two parts: the outer part, which contains the boundary data, and the inner part, which does not. The overlapping technique is applied to the GPU computation of the inner part and the CPU computation of boundary data. In practical implementation, the OpenCL blocking function is positioned after the CPU computation to initiate the boundary data computation without waiting for the GPU computation to finish. Figure 9 illustrates this modification in programming flow. To validate the effectiveness of the overlapping technique, the position of the blocking function must be altered to execute the inner part GPU computation and the boundary CPU computation sequentially. In this manner, the effi-

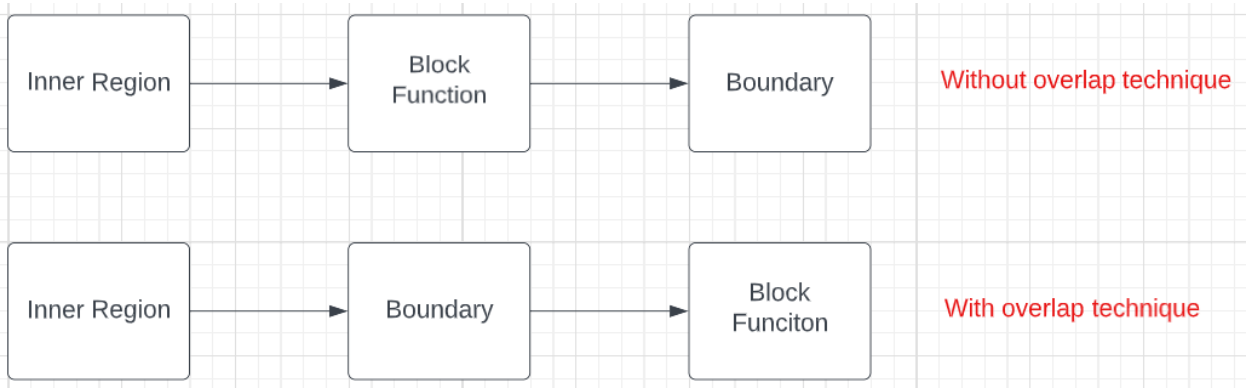

Figure 9: The programming flow of the experiment involves completing the computation of the inner region in the GPU and the boundary computation in the CPU. To evaluate the impact of overlapping CPU and GPU computations on the run-time, we compared the run-time of two flows. The first flow includes placing the block function in between, which forces the GPU and CPU to compute serially. The second flow overlaps the computations of the GPU and CPU. By comparing the run-time of these two flows, we can conclude the amount of run-time is shortened by overlapping the CPU computation with the GPU computation.

ciency of the overlapping technique can be verified if the runtime of the first programming flow in Figure 9 is longer than the second one.

3.4. Results

Table 5 presents the profiling results of different systems when running with four GPUs. As described in the previous section, each "event" is associated with a "command queue", and each "command queue" is linked to a specific GPU. Thus, the runtime of each GPU is tested separately. The experimental results in the table show the cumulative runtime of 1000 iterations of the "add" module. The table provides several timings. First, "read time" refers to the time required to read data from a GPU by the host CPU. Second, "write time" represents the time needed to write data from the host CPU to a GPU. Third, "outer time" refers to the time required to execute the outer region in a GPU. Fourth, "inner time" represents the time taken to execute the inner region on a GPU. Fifth, "boundary time" refers to the time needed to compute the boundary data in serial on the host CPU. Finally, "wall clock time" represents the actual runtime. Two wall clock times are reported, one associated with the outer time and one associated with the inner and boundary times. Because kernel functions in different GPUs are expected to execute in parallel, the "wall clock time" of the "outer time" refers to the time taken from the first kernel function that starts to run to the last kernel function that runs. The "wall clock time" reported in the rightmost column is the total runtime of the inner region computation plus the boundary serial computation. Based on the profiling results in Table 5, we note the following observations (where we have highlighted some table entries mentioned in the text for convenience).

First, it can be observed that the "read time" and "write time" are longer for GPUs located in the middle, such as GPU2 when using three GPUs, and GPU2 and GPU3 when using four GPUs. This is because the sub-domains in the middle have two additional boundary columns that increase the amount of data that needs to be transferred to the host CPU.

Second, it can be observed that the workload of each GPU is unbalanced. In any simulation, there are non-observation points on which the `WDPM` performs no computation. Therefore, all sub-domains may not have the same size. However, the runtime of the "outer time" indicates that computations on multiple GPUs can still be parallelized effectively.

Third, it can be observed that the overlapping communication and computation method performs well for large systems. In such cases, the runtime is significantly shorter when using overlapping compared to running the GPU computation and CPU computation sequentially. For instance, when using four GPUs on "patched.asc" (shown in Table 5), the boundary data computation (6.08359 s) is essentially completely hidden behind the inner region computation on the GPU (6.09307 s) — the total wall clock time of the inner and boundary time is 6.24318 s. However, using four GPUs on "basin5.asc" (also shown in Table 5) reveals that the boundary data computation (0.41139 s) takes

Table 5
the profiling result of using four GPUs

	read	write	outer	inner	boundary
system		basin5.asc (471×482)			
GPU 1	0.00177	0.00227	0.08768	0.04385	
GPU 2	0.00345	0.00450	0.09065	0.04491	0.41139
GPU 3	0.00352	0.00454	0.08777	0.04352	
GPU 4	0.00178	0.00232	0.08013	0.03996	
practical			0.11070	0.57250	
system		basin4_5m.asc (2520×1833)			
GPU 1	0.00352	0.00546	0.64573	0.32040	
GPU 2	0.00688	0.00943	1.07178	0.52759	1.35556
GPU 3	0.00710	0.01017	0.47532	0.23279	
GPU 4	0.00178	0.00585	0.37852	0.18788	
practical			1.25001	1.43666	
system		culvert_basin1_5m.asc (3794×3986)			
GPU 1	0.00620	0.01170	2.18561	1.09400	
GPU 2	0.01247	0.01849	5.91543	2.94556	4.37358
GPU 3	0.01274	0.01930	6.60300	3.28824	
GPU 4	0.00606	0.01211	4.44874	2.20622	
practical			7.83544	4.48975	
system		smithcreek_dem1m_sb5.asc (4712×4826)			
GPU 1	0.00754	0.01531	4.02102	2.00870	
GPU 2	0.01418	0.02311	8.48651	4.22610	4.64050
GPU 3	0.01408	0.02399	6.32994	3.15144	
GPU 4	0.00690	0.01572	2.39133	1.18867	
practical			8.56870	4.74995	
system		patched.asc (5877×5519)			
GPU 1	0.00872	0.01936	7.21244	3.59759	
GPU 2	0.01642	0.02812	11.53983	5.75371	6.08359
GPU 3	0.01642	0.02879	12.23909	6.09307	
GPU 4	0.00794	0.01975	5.82081	2.89596	
practical			12.47135	6.24318	

much longer than the inner GPU computation (0.04385 s), and hence the overlapping technique does not provide any computational advantage.

4. Discussion

When assessing performance, running simulations with multiple GPUs does not necessarily lead to decreased computation times. Figure 6 illustrates that when using multiple GPUs, four-column boundary data matrices are generated per GPU to synchronize data between adjacent sub-domains, and the boundary data are handled sequentially in the host CPU. This approach can cause significant overhead due to communication for small input systems because the boundary data can represent a sizeable proportion of the overall data. As a consequence, using four GPUs to run a small system (e.g., "basin5.asc") can result in extremely poor performance. However, the primary purpose of using multiple GPUs is to reduce the run time for solving large problems, and the `WDPM` has been well-developed for solving general problems.

As discussed previously, the input system is divided into several slices, and each slice is assigned to a separate GPU. Because different sub-domains may have different sizes and some regions may require no computations, the workload of each GPU can be unbalanced. To evaluate the ideal acceleration, synthetic systems were created using a random number generator, with sizes ranging from 1000×1000 to 6000×6000. The runtime of each module for every such system was recorded, and each simulation was executed three times to assess the robustness of the results.

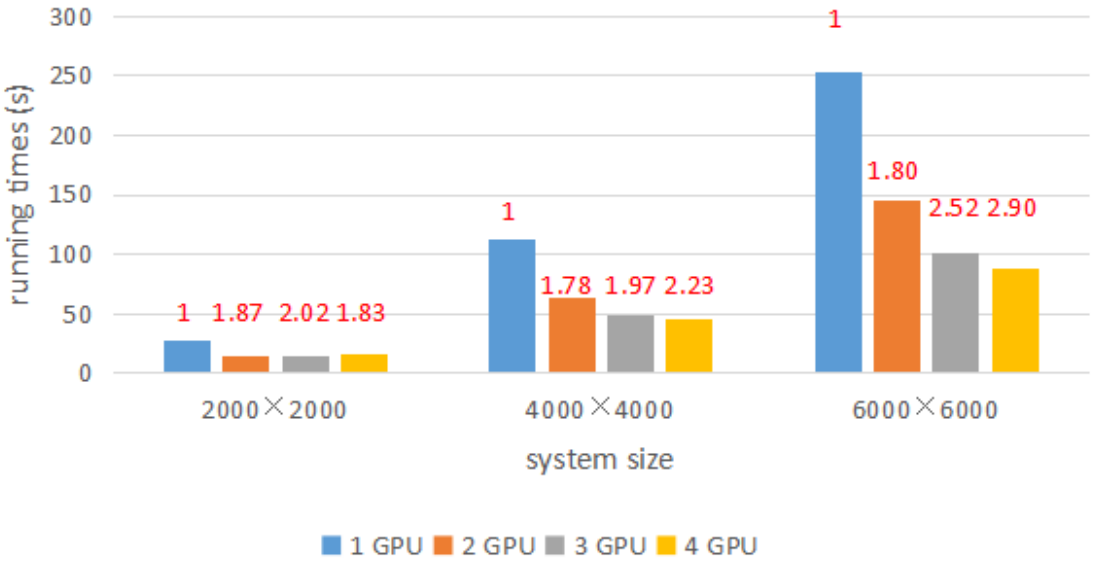

Figure 10: Test results of the "add" module with synthetic systems. The speedup compared with using one GPU is given at the top of each column.

Accordingly, the runtime of each task was calculated as

$$S = t^{(1)}/t^{(n)}_{\min}, \quad n = 2, 3, 4. \tag{1}$$

The performance improvements are evaluated using Equation 1, and the test results are presented in Figures 10–12. By examining the acceleration of the "add" module in Figure 10, it can be observed that the speedup is more significant when the input system is larger. Specifically, a speedup of approximately three times is achieved when working on a 6000×6000 system with four GPUs.

On the other hand, the efficiency improvements of the "subtract" and "drain" modules, as shown in Figures 11–12, are less significant than that of the "add" module. This is attributed to the varying computation amounts of different modules. For instance, when running the "add" module, the SW algorithm function is called 110,036,000 times per thousand iterations on average, whereas the "subtract" and "drain" modules call the SW function 5,992,000 times and 7,029,997 times, respectively. These numbers reveal a significant difference in calculation scales among the modules. Moreover, in `WDPM`, the SW algorithm only operates on pixels whose water depth is larger than 0, referred to as active points. For the "add" module, all observation points are active, whereas for the "subtract" and "drain" modules, only a small proportion of observation points are active. Therefore, working on the "subtract" and "drain" modules implies working with a relatively small system. As previously discussed, multi-GPU implementation is more efficient when working with a large system due to data synchronization overhead, hence explaining the higher speedup achieved with the "add" module when using multiple GPUs.

Finally, the performance of the proposed approach was evaluated on real systems listed in Table 3. Table 6 presents the runtime of the different systems when using various numbers of GPUs based on Equation 1. For comparison, the runtime using Intel(R) Xeon(R) CPU E5-2650 (2.20 GHz) in serial and parallel CPU (32 threads) is also recorded. The results show that using GPUs significantly improves the simulation speed, allowing the simulation to be completed in less time. The speedup achieved using different numbers of GPUs compared to using only one GPU is illustrated in Figure 13. However, due to the imbalance in workloads across devices resulting from differences in the proportions of non-observation areas in each sub-domain, the performance improvement is not as significant as in the synthetic system. Notably, the largest real system used in the experiments is "patched.asc" with a size of 5877×5519. Our experiments show that the best speedup achieved for real systems using four GPUs is 2.39 times.

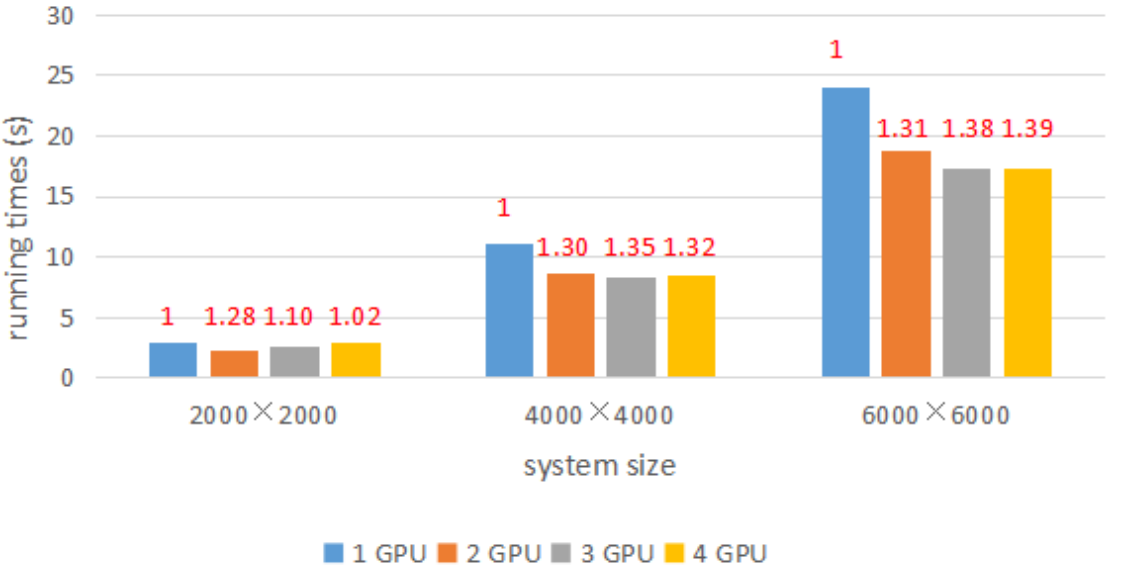

Figure 11: Test results of the "subtract" module with synthetic systems. The speedup compared with using one GPU is given at the top of each column.

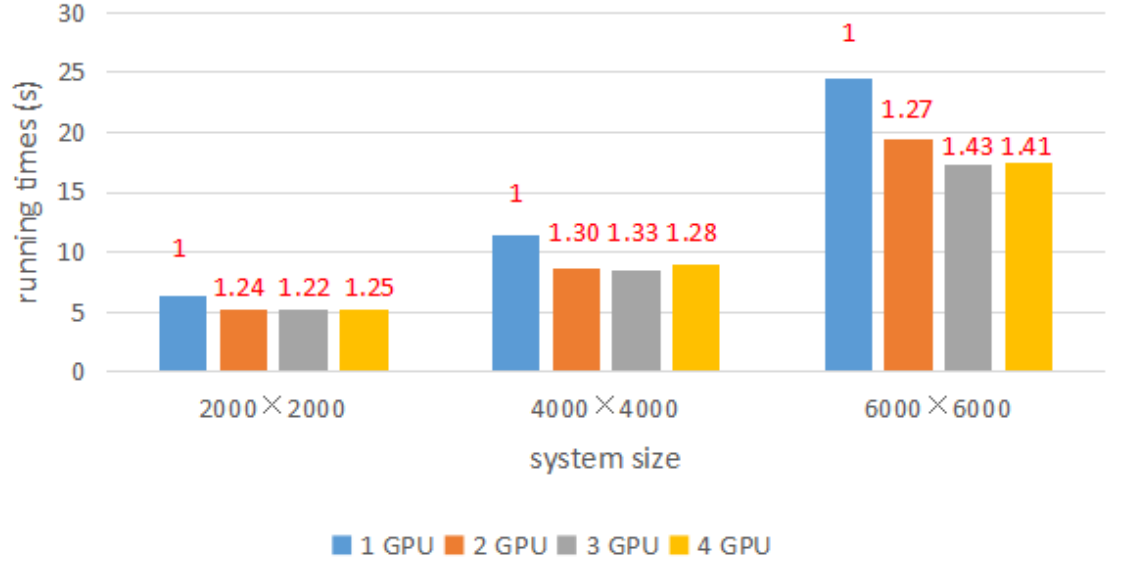

Figure 12: Test results of the "drain" module with synthetic systems. The speedup compared with using one GPU is given at the top of each column.

5. Conclusions

The Wetland DEM Ponding Model is a valuable component of Canadian Prairie hydrology simulations of runoff water distribution based on an input DEM system. The model has been used in numerous case studies and adapted to several applications, including the Government of Canada Land and Infrastructure Resiliency Assessment Project (**?**). However, working with large systems requires more efficient simulations for water redistribution and routing so that

Table 6
The test results of the "add" module with the real system.

system		basin5.asc (+10 mm)	culvert_basin1_5m.asc (+3 mm)	patched.asc (+3 mm)
size		471×482	3794×3986	5877×5519
CPU (s)		1132.47	360950.98	≈115604.75
CPUs (32) (s)		77.86	13758.98	44624.19
GPUs	1 (s)	8.23	3203.29	11149.74
	2 (s)	14.77	1889.50	6079.70
	3 (s)	24.04	1680.17	4976.20
	4 (s)	32.12	1683.01	4673.24

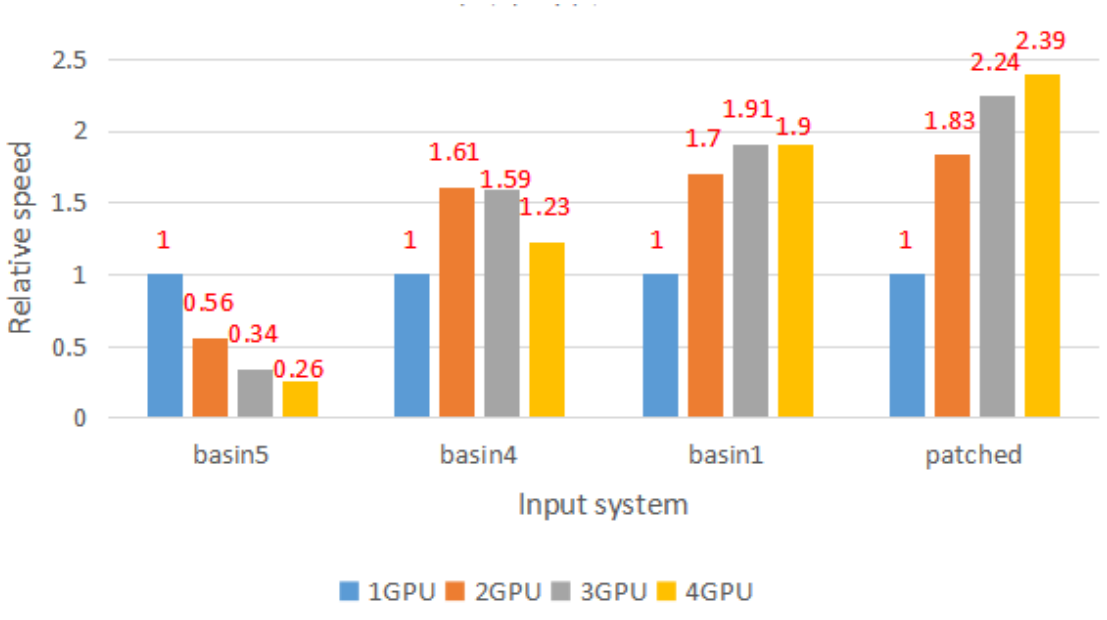

Figure 13: Test results with running on GPUs.

results are computed in a reasonable timeframe. To address this, a multi-GPU implementation has been developed to increase performance. This study contributes to the `WDPM` code in the following ways:

1. A multi-GPU parallel algorithm has been developed to extend the previous CPU parallel algorithm and the single-GPU parallel algorithm. The correctness of the implementation is verified by evaluating the output summaries and errors of the multi-GPU output maps. The simulations produce error maps with one GPU output system and multi-GPU output systems, and by increasing the number of iterations, the solutions converge significantly. This result confirms that the multi-GPU implementation produces the correct results.
2. An event-based technique is used to profile the OpenCL program in the simulations. Although the workload is not the same on each GPU, multiple GPUs can be parallelized well. The new implementation uses overlapping communication and computation techniques to utilize CPU computation and GPU computation simultaneously. The overlapping technique is demonstrated to be efficient when working with large systems because the CPU computation can be mostly hidden behind the GPU computation.
3. The compute time of working with different numbers of GPUs (up to four) is tested when solving problems of different sizes. Synthetic systems of different sizes are also used to calculate an ideal performance improvement when working with multiple GPUs. Using multiple GPUs worsens the performance for small systems, such as "basin5.asc" (471×482), where it is around 1.59 times slower when using 2 GPUs compared to using 1 GPU. However, as the problem size increases, using multiple GPUs exhibits increased performance.

For the "basin4_5m.asc" (2520×1833), the maximum speedup is about 1.61 when using 2 GPUs. For the "smithcreek_dem1m_s.asc" (4712×4826), the maximum speedup is about 1.92 when using 3 GPUs. For the "patched.asc" (5877×5519), the maximum speedup is 2.39 when using 4 GPUs. Furthermore, a synthetic 6000×6000 DEM system is produced to experiment with an ideal (load-balanced) system, where the maximum speedup is about 3.1 when using 4 GPUs. These results demonstrate the significant performance improvement of running the `WDPM` on multiple GPUs, and they further demonstrate the effective scaling of the new implementation.

5.1. Future Work

There are several potential directions for future research on the `WDPM` and related models:

1. Field Programmable Gate Arrays (FPGAs) are integrated circuits that provide customers the ability to reconfigure the hardware to meet specific use case requirements after the manufacturing process. The parallel programming strategy of FPGAs exploits pipeline parallelism to execute different stages of instructions simultaneously on multiple work items. Compared to GPUs, FPGAs exhibit superior energy efficiency and bandwidth, making them a compelling option for reducing compute times and improving adaptability to diverse hardware and platforms for the `WDPM`. Therefore, future research aims to decompose the SW algorithm to implement the water redistribution model on FPGA for enhanced performance.
2. Recent studies have focused on the use of data-driven models and machine learning (ML) techniques to predict runoff with high accuracy. Notably, studies such as (Mohammadi, 2021) have shown that ML models, including the Adaptive Neuro-Fuzzy Inference System (ANFIS), Artificial Neural Network (ANN), and Support Vector Machine (SVM), are powerful tools for runoff modeling in different regions. In particular, Dibike, Velickov, Solomatine and Abbott (2001) utilized the SVM method for rainfall-runoff simulation using daily rainfall, evapotranspiration, and streamflow data from different catchments with varying precipitation rates to obtain appropriate data formats for SVM. This approach is similar to the `WDPM` "add", "subtract", and "drain" modules. Moreover, the SVM method demonstrated higher accuracy in runoff estimation compared to other methods, as supported by the findings in (Dibike et al., 2001). Most recently, deep learning methods like Long Short-Term Memory (LSTM) networks (**?**) and Transformers (**??**) have become popular in rainfall-runoff modeling due to advancements in computer technology and the size and quality of datasets (**??**). According to recent surveys, the best option for rainfall-runoff modeling depends on the specific problem and data characteristics. However, LSTM models at present seem to be the most popular choice (**??**). Accordingly, the use of `WDPM` in the training and validation of ML models is a promising direction for future research.

Acknowledgments

The authors are grateful for funding from the Natural Sciences and Engineering Research Council of Canada under its Discovery Grant program [RGPN-2018-06317 (SBK) and RGPN-2020-04467 (RJS)].

CRediT authorship contribution statement

Tonghe Liu: Conceptualization; Data curation; Formal analysis; Investigation; Methodology; Software; Validation; Visualization; Roles/Writing - original draft; and Writing - review & editing. **Sean J. Trim:** Writing - review & editing. **Seok-Bum Ko:** Conceptualization; Funding acquisition; Investigation; Methodology; Project administration; Resources; Supervision; Writing - review & editing. **Raymond J. Spiteri:** Conceptualization; Funding acquisition; Investigation; Methodology; Project administration; Resources; Supervision; Writing - review & editing.

Software and data availability

The software and data used for the research presented in this paper are free and open source and available at ??.

References

Armstrong, R., Kayter, C., Shook, K., Hill, H., 2013. Using the wetland DEM ponding model as a diagnostic tool for prairie flood hazard assessment. Putting Prediction in Ungauged Basins into Practice , 255–270.

Dibike, Y.B., Velickov, S., Solomatine, D., Abbott, M.B., 2001. Model induction with support vector machines: Introduction and applications. Journal of Computing in Civil Engineering 15, 208–216. doi:10.1061/(ASCE)0887-3801(2001)15:3(208).

Hacene, M., Anciaux-Sedrakian, A., Rozanska, X., Klahr, D., Guignon, T., Fleurat-Lessard, P., 2012. Accelerating VASP electronic structure calculations using graphic processing units. Journal of Computational Chemistry 33, 2581–2589.

Hiranandani, S., Kennedy, K., Tseng, C.W., 1992. Compiling Fortran D for MIMD distributed-memory machines. Communications of the ACM 35, 66–80.

Kevin, S.R., Armstrong, R., Sharomi, O., Spiteri, R., Pomeroy, J.W., 2014. The WDPM user's guide.

Krawezik, G.P., Poole, G., 2009. Accelerating the ANSYS direct sparse solver with GPUs, in: Symposium on Application Accelerators in High Performance Computing, SAAHPC.

Michalakes, J., Vachharajani, M., 2008. GPU acceleration of numerical weather prediction. Parallel Processing Letters 18, 531–548.

Mohammadi, B., 2021. A review on the applications of machine learning for runoff modeling URL: NVIDIA.

Munshi, A., Gaster, B., Mattson, T.G., Ginsburg, D., 2011. OpenCL programming guide. Pearson Education.

Narasiman, V., Shebanow, M., Lee, C.J., Miftakhutdinov, R., Mutlu, O., Patt, Y.N., 2011. Improving GPU performance via large warps and two-level warp scheduling, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 308–317.

Pomeroy, J.W., Shook, K.R., Dumanski, S., 2014. Improving and testing the prairie hydrological model at Smith Creek Research Basin. Centre for Hydrology, University of Saskatchewan.

Scarpino, M., 2011. OpenCL in action: how to accelerate graphics and computations .

Shapiro, M., Westervelt, J., 1992. R. MAPCALC, in: An algebra for GIS and image processing. United States Army Construction Engineering Research Laboratory Champaign.

Shook, K.R., Pomeroy, J.W., 2011. Memory effects of depressional storage in Northern Prairie hydrology. Hydrological Processes 25, 3890–3898.

Vouzis, P.D., Sahinidis, N.V., 2011. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. Bioinformatics 27, 182–188.