Hindawi
Modelling and Simulation in Engineering
Volume 2021, Article ID 9534495, 18 pages
https://doi.org/10.1155/2021/9534495

*Research Article*

# Parallel Algorithms of Well-Balanced and Weighted Average Flux for Shallow Water Model Using CUDA

**Nugool Sataporn,[1] Worasait Suwannik,[1] and Montri Maleewong [2,3]**

[1]*Department of Computer Science, Faculty of Science, Kasetsart University, Bangkok 10900, Thailand*
[2]*Department of Mathematics, Faculty of Science, Kasetsart University, Bangkok 10900, Thailand*
[3]*Excellent Center for Big Data Analytics on Food and Agriculture, Kasetsart University, Bangkok 10900, Thailand*

Correspondence should be addressed to Montri Maleewong; montri.m@ku.th

Received 23 July 2021; Revised 6 October 2021; Accepted 24 November 2021; Published 10 December 2021

Academic Editor: Javier Murillo



Compute Unified Device Architecture (CUDA) implementations are presented of a well-balanced finite volume method for solving a shallow water model. The CUDA platform allows programs to run parallel on GPU. Four versions of the CUDA algorithm are presented in addition to a CPU implementation. Each version is improved from the previous one. We present the following techniques for optimizing a CUDA program: limiting register usage, changing the global memory access pattern, and using loop unroll. The accuracy of all programs is investigated in 3 test cases: a circular dam break on a dry bed, a circular dam break on a wet bed, and a dam break flow over three humps. The last parallel version shows 3.84x speedup over the first CUDA implementation. We use our program to simulate a real-world problem based on an assumed partial breakage of the Srinakarin Dam located in Kanchanaburi province, Thailand. The simulation shows that the strong interaction between massive water flows and bottom elevations under wet and dry conditions is well captured by the well-balanced scheme, while the optimized parallel program produces a 57.32x speedup over the serial version.

## 1. Introduction

Shallow water equations are derived from the conservation of mass and momentum under hydrostatic approximation. This model can be used to describe fluid flow under various scenarios over time. One can apply the shallow water equations to simulate dam break problems, the flood plain over wet and dry areas, or even tsunami waves in some senses.

Shallow water equations can be solved numerically using the finite volume method. The computational domain is discretized by rectangular [1], triangular [2, 3], tetrahedral [4], or unstructured cells. The applications of shallow water equations using Roe-type solutions can be found in [5, 6]. The flux gradient at each cell interface can be approximated by the TVD-WAF (total variation diminishing-weighted average flux) or HLL (Harten-Lax-van Leer) methods [7]. A well-balanced scheme is introduced later for balancing the flux gradients and source term approximation at each cell interface; see [1] and compare the results in [8, 9]. Due to the simplicity of the rectangular cell, we implement a well-balanced scheme based on the weighted average flux (WAF) of the finite volume method extended from [8–11] to simulate various complex flow problems. The data elevation model (DEM) is available in rectangular format, so it is easy to utilize as an input in our program for simulating real-world problems. For a very large domain, the computational time is massive; thus, several approaches are proposed to reduce the computational time. One concept uses parallel computation. Our main study is to develop parallel programs on GPU where the parallel version is developed from the serial version proposed in [8, 9]. The most challenging is the implementation using unstructured meshes since the data structure and hence the memory access is not straightforward. Several types of research implement unstructured grid; see [2, 3], but the rectangular domain is simple and not much difficult to develop parallel programming.

Furthermore, this approach can be implemented on several platforms such as CPU [8, 9, 12, 13], GPU [6–9, 14–17], and a cluster of computers [3].

We implemented a program that runs on GPU. The program is written using the CUDA (Compute Unified Device Architecture) programming model. Several parallel implementations of the shallow water equations on CUDA can be found in the literature [17–22]. Most of them report a huge speedup in the computational time compared to serial implementation running on CPU. However, comparing the CUDA program with a serial program might not provide a meaningful result as there are several issues involved in the comparison. The first is the selection of the CPU and GPU models to be compared. It is arguable whether the selection should be based on factors such as the number of cores, number of transistors, clock speed, or price. The second is the effort required to optimize a program for both CPU and GPU implementation. For a balanced comparison, a baseline CPU implementation must be highly optimized (using an efficient data structure, taking advantage of special instructions, or taking advantage of a multicore architecture). Third, the size of the main memory is normally much larger than the size of the graphics card memory. As a result, the program that runs on the CPU can solve a larger problem than the one running on GPU. Thus, both the CPU and GPU implementations should solve the large problem.

From these reasons, our paper focuses on optimizing a CUDA implementation of the shallow water equation [8, 9]. Nsight, a CUDA profiler, is used to determine an area for computational improvement. We implement several programs based on Nsight reports.

## 2. Numerical Method

*2.1. Finite Volume Method.* The finite volume method is a popular method for solving shallow water equations, for examples, see [5, 8, 10, 23]. The two-dimensional shallow water equation can be written as

$$h_t + (hu)_x + (hv)_y = 0, \tag{1}$$

$$(hu)_t + \left(hu)^2 + \frac{1}{2}\, gh^2\right)_x + (huv)_y = -ghz_x + S_{fx}, \tag{2}$$

$$(hv)_t + \left(hv)^2 + \frac{1}{2}\, gh^2\right)_y + (huv)_x = -ghz_y + S_{fy}, \tag{3}$$

where $h$ is the water depth, $u$ and $v$ are the flow velocities in the $x$ and $y$ directions, respectively, $g$ is the acceleration due to gravity, and $z$ is the bottom elevation.

The shallow water equations (1)–(3) can be written in a conservation form as

$$U_t + F(U)_x + G(U)_y = S(U), \tag{4}$$

where $U = (h, hu, hv)^T$, $F = (hu, hu^2 + (1/2)gh^2, huv)^T$, $G = (hv, huv, hv^2 + (1/2)gh^2)^T$, and $S = S_b + S_f = (0, -ghz_x, -ghz_y)^T + (0, -ghS_{fx}, -ghS_{fy})^T$.

Applying the finite volume method, we obtain the discretized scheme as

$$\frac{dU_{ij}(t)}{dt} + \frac{\widehat{F}_{i+1/2,j} - \widehat{F}_{i-1/2,j}}{\triangle x} + \frac{\widehat{G}_{i,j+1/2} - \widehat{G}_{i,j-1/2}}{\triangle y} = S_{ij}, \tag{5}$$

where $\triangle x = x_{i+1/2} - x_{i-1/2}$, $\triangle y = y_{j+1/2} - y_{j-1/2}$, and $S_{ij}$ is the approximation of source term at cell $I_{ij}$. Here, $\widehat{F}_{i+1/2,j}$ and $\widehat{G}_{i+1/2,j}$ are numerical fluxes at the right and upper interfaces, respectively. The notations are similar to the other interfaces of a cell. In this work, we apply the weighted average flux method to approximate numerical fluxes. The reconstruction steps are implemented and the well-balanced scheme is also applied based on [8, 9].

*2.2. Weighted Average Flux (WAF).* Weighted average flux was first proposed by [11]. Intercell flux, $F\wedge^{\mathrm{WAF}}$ at the cell interface, $x_{i+1/2,j}$, is defined as an integral average of the flux function $F(U)$ at the half-time step as

$$\widehat{F}^{\mathrm{WAF}}_{i+1/2,j} = \frac{1}{\triangle x}\int_0^{\triangle y}\int_{-\triangle x/2}^{\triangle x/2} F\left(U_{i+(1/2),j}\left(x, y, \frac{\triangle t}{2}\right)\right)dxdy. \tag{6}$$

It can be written in the form of wave structure as

$$\widehat{F}^{\mathrm{WAF}}_{i+1/2,j} = \sum_{k=1}^{N_c+1} \omega_k F^{(k)}_{i+1/2,j}, \tag{7}$$

where $N_c$ is the number of waves in the solution of the Riemann problem and $F^{(k)}_{i+1/2,j}$ is the $k^{\mathrm{th}}$ flux in the solution of the Riemann problem. In this work, we will apply the numerical flux in equation (7) to equation (5).

*2.3. Linear Reconstruction.* Linear reconstruction is applied before calculating the numerical fluxes in equations (6) and (7). $U_{ij}$ is the approximation of $U$ which is defined by the cell averages over cell $I_{ij} = (x_{i-(1/2)}, x_{i+(1/2)}) \times (y_{j-(1/2)}, y_{j+(1/2)})$ as

$$U_{ij} = \frac{1}{\Delta x \Delta y}\int U(x, y)dxdy. \tag{8}$$

The approximate solution using the cell average in equation (8) yields just the first-order scheme; see [11]. We can obtain the second-order scheme by applying linear reconstruction to all conservative variables. For example, in the $x$ direction, we reconstruct variables in the right and the left limits at interfaces as

$$\begin{aligned} U^+_{i-1/2,j} &= U_{ij} - \sigma_{ij}\Delta x, \\ U^-_{i+1/2,j} &= U_{ij} + \sigma_{ij}\Delta x, \end{aligned} \tag{9}$$

where $\sigma_{ij}$ is a slope limiter that has various versions. Here, we apply the minmod slope limiter given by

$$\sigma_{ij} = \text{minmod}\left(\frac{U_{i-1,j} - U_{i,j}}{\Delta x}, \frac{U_{i+1,j} - U_{i,j}}{\Delta x}\right), \quad (10)$$

where

$$\text{minmod}(a,b) = \begin{cases} a \text{ if } |a| \le |b| \text{ and } ab > 0, \\ b \text{ if } |a| \ge |b| \text{ and } ab > 0, \\ 0 \; ab \le 0. \end{cases} \quad (11)$$

Similar calculations can be performed in the $y$ direction. Slope limiters can avoid spurious oscillations in the numerical solution for high-gradient free-surface flows; see [8, 11].

*2.4. Well-Balanced Finite Volume Method.* The hydrostatic well-balanced scheme proposed by Audusse et al. [24] is applied in this work. The concept is designed for balancing between the flux gradient and bottom slope approximations at a cell interface. The advantage of this scheme is the conservation properties of water at rest. Before calculating numerical flux, it is required to reconstruct water depth $h$ at interfaces in both the $x$ and $y$ directions. The well-balanced scheme can preserve the solution at a steady state and ensure nonnegativity of the water height using the following approximations

$$h^{\pm,*}_{i+(1/2),j} = \max\left(0, h^{\pm,*}_{i+(1/2),j} + z^{\pm}_{i+(1/2),j} - z_{i+(1/2),j}\right), \quad (12)$$

$$h^{\pm,*}_{i,j+(1/2)} = \max\left(0, h^{\pm,*}_{ij+(1/2),} + z^{\pm}_{i,j+(1/2),} - z_{i,j+(1/2)}\right), \quad (13)$$

where $z_{i+(1/2),j} = \max\left(z^{-}_{i+(1/2),j}, z^{+}_{i+(1/2),j}\right)$ and $z_{i,j+(1/2)} = \max\left(z^{-}_{i,j+(1/2)}, z^{+}_{i,j+(1/2)}\right)$.

The objective of this reconstruction is to balance the conservative fluxes and the bed slope source term with frictionless conditions.

Finally, all of the presented techniques are combined in the finite volume method. The numerical scheme is called the well-balanced finite volume with WAF scheme [8]. The discretization form is

$$\frac{dU_{ij}}{dt} + \frac{\widehat{F}^{\text{WAF}}_{i+1/2,j} - \widehat{F}^{\text{WAF}}_{i-1/2,j}}{\triangle x} + \frac{\widehat{G}^{\text{WAF}}_{i,j+1/2} - \widehat{G}^{\text{WAF}}_{i,j-1/2}}{\triangle y} = S_{ij}. \quad (14)$$

The updated solution can be obtained by the forward method. To improve the stability of the finite volume scheme, the source term including the roughness coefficient will be approximated by the splitting implicit method; see more details in [8]. Briefly, we solve the shallow water system by considering the ordinary differential equation

$$\frac{dU_{ij}}{dt} = \left(S_f\right)^{n+1}_{ij}, \quad (15)$$

where $n$ is the time step. For example, the implicit forward time in the $x$ direction is approximated by

$$\frac{d(hu)_{ij}}{dt} = \left(S_{fx}\right)^{n+1}_{ij}. \quad (16)$$

This can be rewritten in an explicit formula as

$$(hu)^{n+1}_{ij} = (hu)^{n}_{ij} + \Delta t \frac{\left(S_{fx}\right)^{n}_{ij}}{\left(D_x\right)^{n}_{ij}}, \quad (17)$$

where the implicit coefficient $D_x$ is defined by

$$\left(D_x\right)^{n}_{ij} = 1 + \Delta t C^{n}_{ij} \frac{2\left((hu)^{n}_{ij}\right)^2 + \left((hv)^{n}_{ij}\right)^2}{\left(h^{n}_{ij}\right)^2 \sqrt{\left((hu)^{n}_{ij}\right)^2 + \left((hv)^{n}_{ij}\right)^2}}, \quad C^{n}_{ij} = \frac{g {n_m}^2}{\left(h^{n}_{ij}\right)^{1/3}}, \quad (18)$$

where $n_m$ is the Manning coefficient. This technique provides a more stable finite volume method when dealing with a strong nonlinear friction term if we compare it with the standard explicit method.

For the present step, we can calculate cell by cell sweeping along the $x$ direction until all numbers of cells are calculated. The computation by forwarding time and sweeping cell by cell allows us to modify the serial computation to a parallel computation. This is the main objective of our present work. We will design and give some concepts in the next section.

## 3. CUDA

GPU has a large number of cores compared with a multicore CPU, for example, NVIDIA QUADRO M6000 has 3072 cores. As a result, a GPU can process efficiently 3D graphics data such as texture shading. Therefore, GPU is suitable for massive parallel computations. Utilizing a GPU to run a general-purpose parallel program would greatly improve the program's running time. Compute Unified Device Architecture (CUDA) is a parallel model that allows a program to run on a GPU. CUDA was created by NVIDIA, a company that manufactures GPUs.

*3.1. Thread.* A thread is a sequence of programmed instructions that can be executed independently. A group of threads on the CUDA model is called a block. A group of blocks is called a grid. All threads in a grid can be launched from the same kernel, and each thread in the same block has a unique thread ID. Furthermore, every block in a grid has a unique block ID as shown in Figure 1.

We can create 1-, 2-, or 3-dimensional CUDA thread blocks. The information for 2-dimensional grid and thread blocks can be obtained using the variables of gridDim.x and gridDim.y. The variables of blockDim.x, blockDim.y, and blockDim.z are the numbers of threads in the $x$, $y$, and $z$ dimensions, respectively, on a block. The variables blockIdx.x, bloxkIdx.y, and bloxkIdx.z are the block IDs in the $x$ , $y$, and $z$ dimensions, respectively. Finally, the variables of

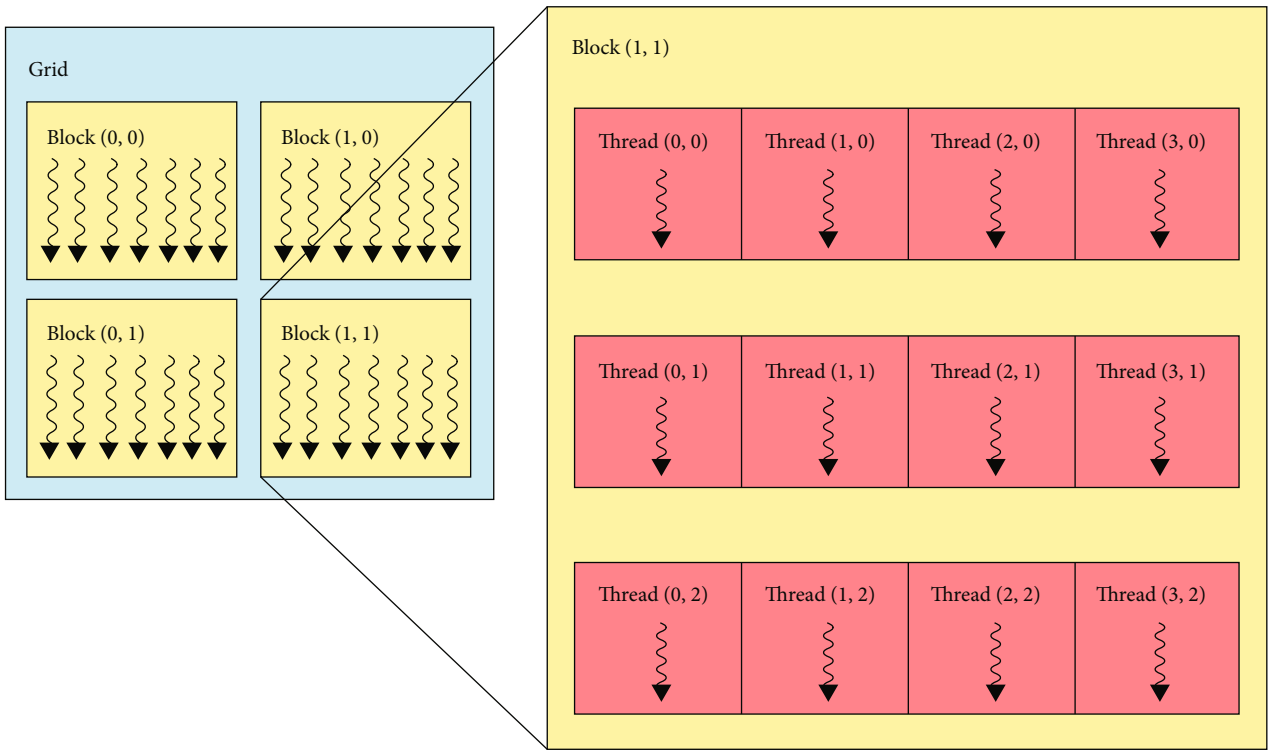



Figure 1: CUDA thread organization.

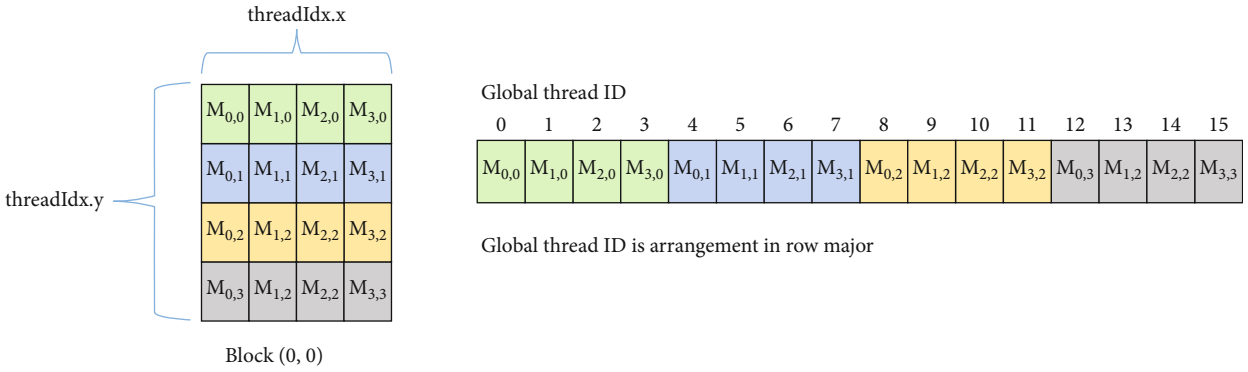



Figure 2: Global thread ID.

threadIdx.x, threadIdx.y, and threadIdx.z are the thread IDs in the $x$, $y$, and $z$ dimensions, respectively.

A unique ID can be made for each thread per kernel based on the block ID and thread ID and is called a global thread ID. Figure 2 shows the design of the global thread ID in the two-dimensional thread in a block.

*3.2. Kernel.* The CUDA program can be implemented in many languages including C, C++, or Fortran. The host and device refer to the CPU and GPU systems, respectively. A kernel is a function that can be parallel executed in a GPU. A kernel is declared by the __global__ keyword. For example, a kernel that receives a pointer can be written as

$$\_\_global\_\_void\ myKernel(int^{*}\ a)\ \{\cdots\}. \tag{19}$$

A kernel is launched using the <<<grid_size, block_

Table 1: Specification of NVIDIA GeForce GTX 1050 Ti.

| Specification | Properties |
|---|---|
| Max threads per block | 1024 |
| Max dimensions of a block | (1024, 1024, 64) |
| Max dimensions of a grid | (2147483647, 65535, 65535) |
| Max registers per block | 65536 |
| Warp size | 32 threads |
| CUDA core | 768 cores |
| Memory size | 4 GB |
| Memory bandwidth | 112 GB/sec |
| Memory clock | 1752 MHz |

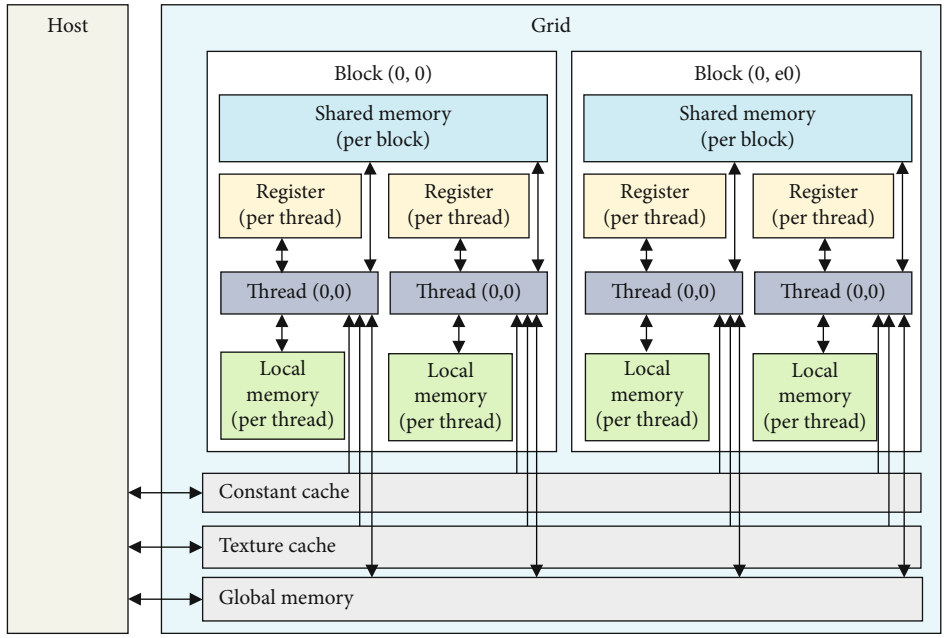



Figure 3: Memory hierarchy.

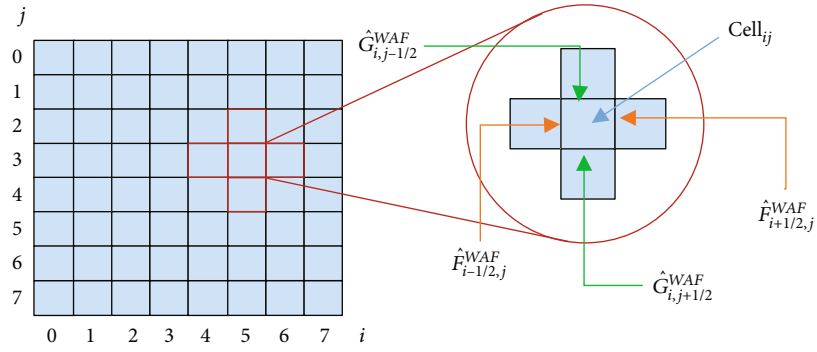



Figure 4: 2D domain and neighboring cells for executing single time step.

```
1: Initialize data
2:   for time = 0 to time_limit increment by Δt
3:     for i = 0 to row
4:       for j = 0 to column
5:         for q in [0, 1]
6:           Calculation flux i + q in x direction using equations (6) and (7)
7:           Calculation flux j + q in y direction using equations (6) and (7)
8:         end for
9:       Update solution (i, j) by equations (8)–(13)
10:    end for
11:  end for
9: end for
```

Algorithm 1: Serial: cell-based calculation.

size>>> execution configuration syntax, where the grid size is the number of blocks in a grid and the block_size is the number of threads in a block (these values can be specified in one or more dimensions). The maximum of grid_size and block_size are limited by device specific, for example, NVIDIA GeForce GTX 1050 Ti has the maximum block_size of 1024 and grid size is different in each dimension, 1024 for $x$ and $y$ dimensions and 64 for $z$ dimension (see

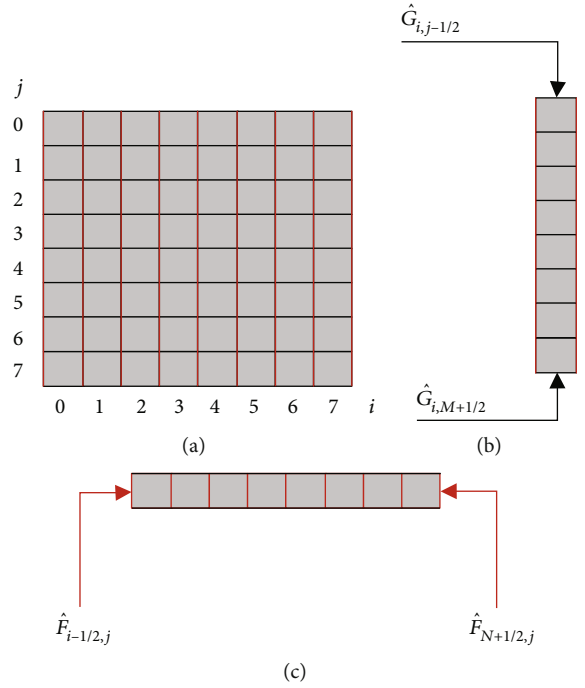



Figure 5: (a) 2D domain, (b) flux in $y$ direction, and (c) flux in $x$ direction.

```
1: Initialize data
2: while time < time_limit
3:    Calculate flux in direction x (call Algorithm 3)
4:    Calculate flux in direction y (call Algorithm 3)
5:    Update solution per cell
6:    time = time + Δt
7: end while
```

Algorithm 2: Serial: edge-based calculation.

```
1: for each row do
2:    for each edge do
3:       Calculate flux in equations (6) and (7)
4:       Store flux to memory
5:    end for
6: end for
```

Algorithm 3: Calculate flux in the $x$ or $y$ direction.

Table 1). An example of launching 3 blocks with 256 threads per block can be written as

$$\text{myKernel} < < < 3, 256 > > > (p). \tag{20}$$

*3.3. Memory Hierarchy.* GPUs have various memory types: global memory, texture memory, constant memory, shared memory, local memory, and register. The CUDA threads can access data from different memory spaces. Each thread has a private local memory and register. All threads in the same block can access the same shared memory. All threads in all grids can access global, constant, and texture memory.

Figure 3 shows that the host can access global, constant, and texture memory. Data can be transferred between host and device memories. Local memory can read and write per thread that can be declared in a kernel function. Shared memory can read and write per block which is allocated by the __shared__ qualifier. The performances of memory on GPUs differ because shared memory is based on chip memory. It will be able to read and write faster than the local or global memory. Shared memory has a latency that is lower than global memory and also reduces the blank conflicts between the thread in a block.

Our experiments are performed on GeForce GTX 1050 Ti GPU, which has CUDA Compute Capability 6.1 (see Table 1).

## 4. Serial and Parallel Implementations

*4.1. Serial: Cell-Based Calculation.* The first of our implementations is a serial program based on the well-balanced finite volume method for equation (5). The computational domain is divided into finite volume cells defined by indices of rows and columns. A solution $(h, hu, hv)$ at the present time step is stored for computing a solution for the next time step. We compute four sides of numerical fluxes which have two values in the $x$ direction and two values in the $y$ direction; see Figure 4 for these flux calculations.

To obtain an updated solution, we compute fluxes $\widehat{F}^{\text{WAF}}_{i+1/2,j}$ and $\widehat{F}^{\text{WAF}}_{i-1/2,j}$ as the $x$-axis and fluxes $\widehat{G}^{\text{WAF}}_{i,j+1/2}$ and $\widehat{G}^{\text{WAF}}_{i,j-1/2}$ as the $y$-axis using the weighted average flux (WAF) method. The

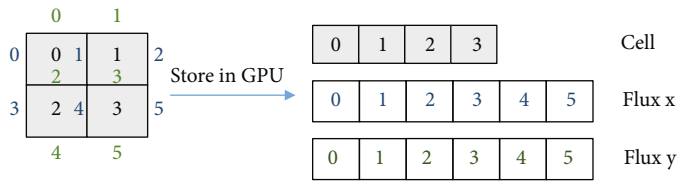



Figure 6: Data storage and data arrangement in the GPU.

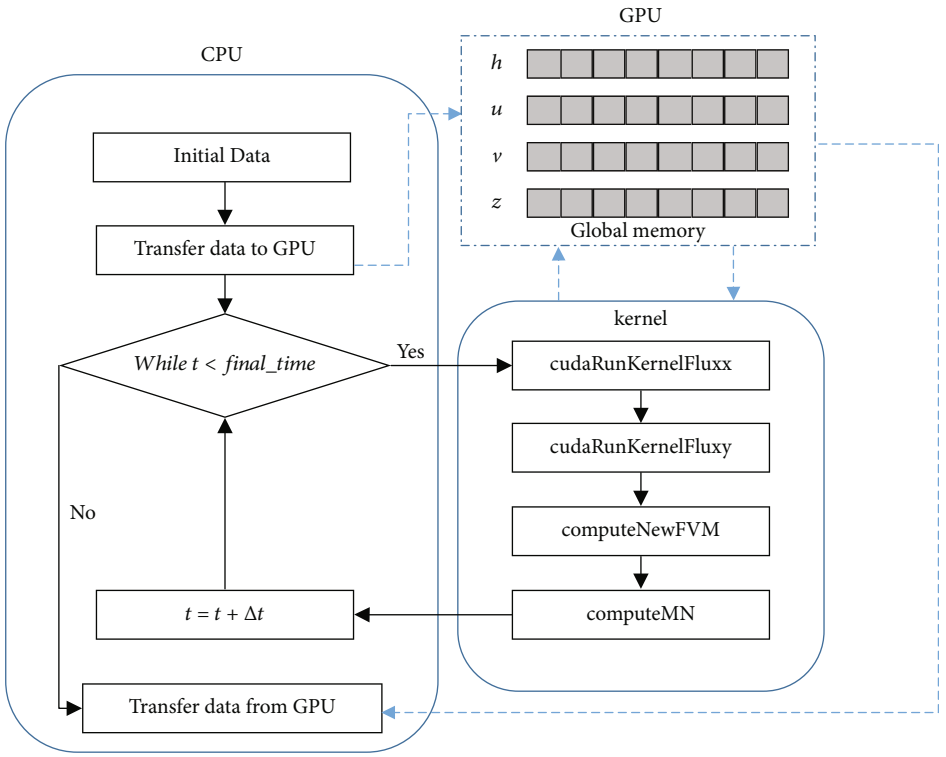



Figure 7: Parallel V1 on CPU and GPU.

| | Function Name | Grid Dimensions | Block Dimensions | Start Time (µs) | Duration (µs) | Occupancy | Registers per Thread |
|---|---|---|---|---|---|---|---|
| 1 | cudaRunKernelFluxy | {8, 8, 1} | {16, 16, 1} | 16,678,936.908 | 2,531.296 | 12.50% | 149 |
| 2 | cudaRunKernelFluxx | {8, 8, 1} | {16, 16, 1} | 32,861,032.396 | 2,232.832 | 12.50% | 149 |
| 3 | computeNewFVM | {8, 8, 1} | {16, 16, 1} | 40,518,661.804 | 603.264 | 25.00% | 88 |
| 4 | computeNonMN | {8, 8, 1} | {16, 16, 1} | 43,240,038.796 | 24.832 | 100.00% | 31 |

Figure 8: Registers per thread and occupancy on global kernel.

computations over cells will be executed again for the next time step. This is a serial version as shown in Algorithm 1.

*4.2. Serial: Edge-Based Calculation.* This version is a serial program similar to the cell-based calculation, but it solves the shallow water equations by recalculating flux quantities. In the cell-based calculation, fluxes of adjacent $c$ ells are calculated twice (see lines 5–8 in Algorithm 1). We can improve these calculations by approximating all fluxes in the $x$ and $y$ directions and then store these values in a global CPU memory. This method calculates fluxes in the directions of $x$ and $y$ separately; see Figures 5(a)–5(c). This concept is implemented using Algorithms 2 and 3, where Algorithm 2 calls Algorithm 3 in each time step.

Before updating the solutions in the next time step, all obtained numerical fluxes will be stored in a memory as a 2D array (line 4 of Algorithm 3). This design reduces the computational time to calculate fluxes in each direction while updating the solutions. We compare the speedup of the cell-based and edge-based algorithms for the test case


Figure 9: Register statistics for CudaRunKernelFluxy.

| | Function Name | Grid Dimensions | Block Dimensions | Start Time (µs) | Duration (µs) | Occupancy | Registers per Thread |
|---|---|---|---|---|---|---|---|
| 1 | cudaRunKernelFluxy | {8, 8, 1} | {16, 16, 1} | 16,837,511.318 | 996.864 | 75.00% | 40 |
| 2 | cudaRunKernelFluxx | {8, 8, 1} | {16, 16, 1} | 33,000,286.070 | 764.128 | 75.00% | 40 |
| 3 | computeNewFVM | {8, 8, 1} | {16, 16, 1} | 40,801,248.822 | 319.488 | 75.00% | 40 |
| 4 | computeNonMN | {8, 8, 1} | {16, 16, 1} | 43,528,207.446 | 25.056 | 100.00% | 31 |

Figure 10: Registers per thread and occupancy on a global kernel with __launch_bounds__.

of the rectangular dam break problem. The domain size is set as $128 \times 128$ grid cells. The serial edge-based program performs 1.8x speedup over the serial cell-based program on a unique CPU core and the same hardware. However, this concept can be improved further by parallel computations as presented in the next section.

*4.3. Parallel V1.* When CUDA threads are used to compute on GPU, we can identify the thread ID in the global thread ID. We need to transfer data between CPU and GPU at the start and the end of our program. Data can be transferred to GPU using the 1-dimensional array. We will apply this technique to store data on the GPU as shown in Figure 6. Numerical values at the cell interfaces and the cell center will be stored in the GPU by the 1-dimensional array of Cell, Flux $x$, and Flux $y$, respectively.

Figure 7 shows an outline of our first parallel implementation that uses edge-based calculation. The initial input data are water depth $h$, flow velocity in the $x$ direction, flow velocity in the $y$ direction, and bottom elevation. These inputs are initialized in a host using CPU and transferred to a device by a graphics adapter. Then, CUDA kernels or global functions are launched. We implement functions cudaRunKernelFluxy and cudaRunKernelFluxx for computing fluxes in both directions. Each thread will get data $(h, u, v, z)$ on cell edges from the device and then call a CUDA device function for solving equations (6) and (7). We use the straight array to store fluxes in the $x$ and $y$ directions on GPU. New solutions are updated using equations (8)–(11) based on computeNewFVM that executes in parallel on the CUDA device. Finally, the bottom slopes and the friction terms from equations (12) and (13) can be computed using the computeMN kernel.

The performance of this algorithm can be evaluated using the running time metric. Our first CUDA implementation is 6.5 times faster than the serial edge-based calculation for the rectangular dam break flow problem. All results will be summarized again in Results.

*4.4. Parallel Occupancy.* To optimize the performance of the parallel V1 program, we will identify the hotspot using NVIDIA Nsight which is a CUDA profiler. Since our CUDA implementations have 4 kernels, the profiler results of these kernels are shown in Figure 8. The kernels that take cudaRunKernelFluxy and cudaRunKernelFluxy are the hotspot. The occupancy of these two kernels is very low (12.50%). An occupancy is equal to the number of active warps per max warp per streaming multiprocessor (SM). This SM is a part of the GPU that runs the CUDA kernel. Each SM contains several streaming processors (SP) that can execute one instruction at a time. SM can be limited due to various reasons. Occupancy is a metric that is related to the number of active warps on a multiprocessor. Low occupancy always interferes with the ability to hide memory latency, resulting in performance; see [25, 26]. Figure 9 shows that our first parallel implementation has low performance for register usage.

We use the __launch_bounds__ directive to limit the usage of the register. The directive has two parameters: MAX_THREADS_PER_BLOCK and MIN_BLOCKS_PER_SM, and the result from varying those parameters is shown in Figure 10. The best result is not at 100% occupancy but at only 75% occupancy and is obtained when MAX_THREADS_PER_BLOCK is 512 and MIN_BLOCKS_PER_SM is 3. We find these results by testing different experiments as shown in Figure 11. Our second parallel implementation called parallel occupancy is 2.42x speedup compared with the parallel V1. This finding shows the computational benefits in that we can obtain better speedup without refactoring the code.

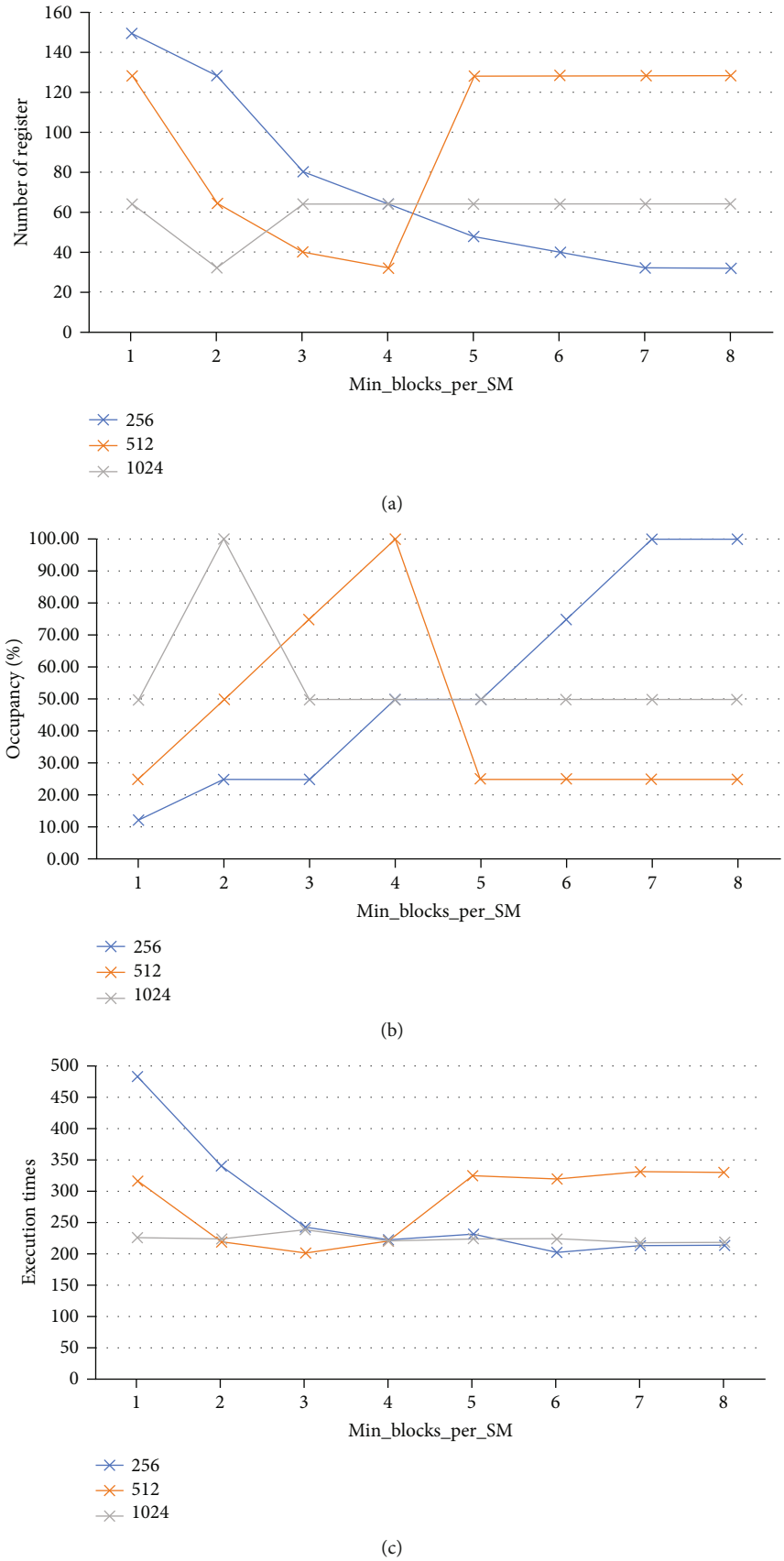



Figure 11: Different max threads per block (256, 512, and 1024) and min block per SM (1–8): (a) registers per thread, (b) occupancy, and (c) execution times.

| | Function Name | Grid Dimensions | Block Dimensions | Start Time (µs) | Duration (µs) | Occupancy | Registers per Thread |
|---|---|---|---|---|---|---|---|
| 1 | cudaRunKernelFluxy | {2, 128, 1} | {64, 1, 1} | 7,560,936.918 | 1,409.760 | 75.00% | 40 |
| 2 | cudaRunKernelFluxx | {2, 128, 1} | {64, 1, 1} | 14,789,814.806 | 587.712 | 75.00% | 40 |
| 3 | computeNewFVM | {2, 128, 1} | {64, 1, 1} | 18,205,253.142 | 300.320 | 75.00% | 40 |
| 4 | computeNonMN | {2, 128, 1} | {64, 1, 1} | 18,824,717.110 | 24.128 | 100.00% | 32 |

(a)

| | Function Name | Grid Dimensions | Block Dimensions | Start Time (µs) | Duration (µs) | Occupancy | Registers per Thread |
|---|---|---|---|---|---|---|---|
| 1 | cudaRunKernelFluxy | {128, 2, 1} | {1, 64, 1} | 17,742,422.067 | 618.497 | 75.00% | 40 |
| 2 | cudaRunKernelFluxx | {2, 128, 1} | {64, 1, 1} | 38,516,643.827 | 601.313 | 75.00% | 40 |
| 3 | computeNewFVM | {2, 128, 1} | {64, 1, 1} | 51,046,297.971 | 301.057 | 75.00% | 40 |
| 4 | computeNonMN | {2, 128, 1} | {64, 1, 1} | 59,524,225.075 | 23.937 | 100.00% | 32 |

(b)

Figure 12: Different execution times on both kernels: (a) same block dimension and (b) different block dimensions.

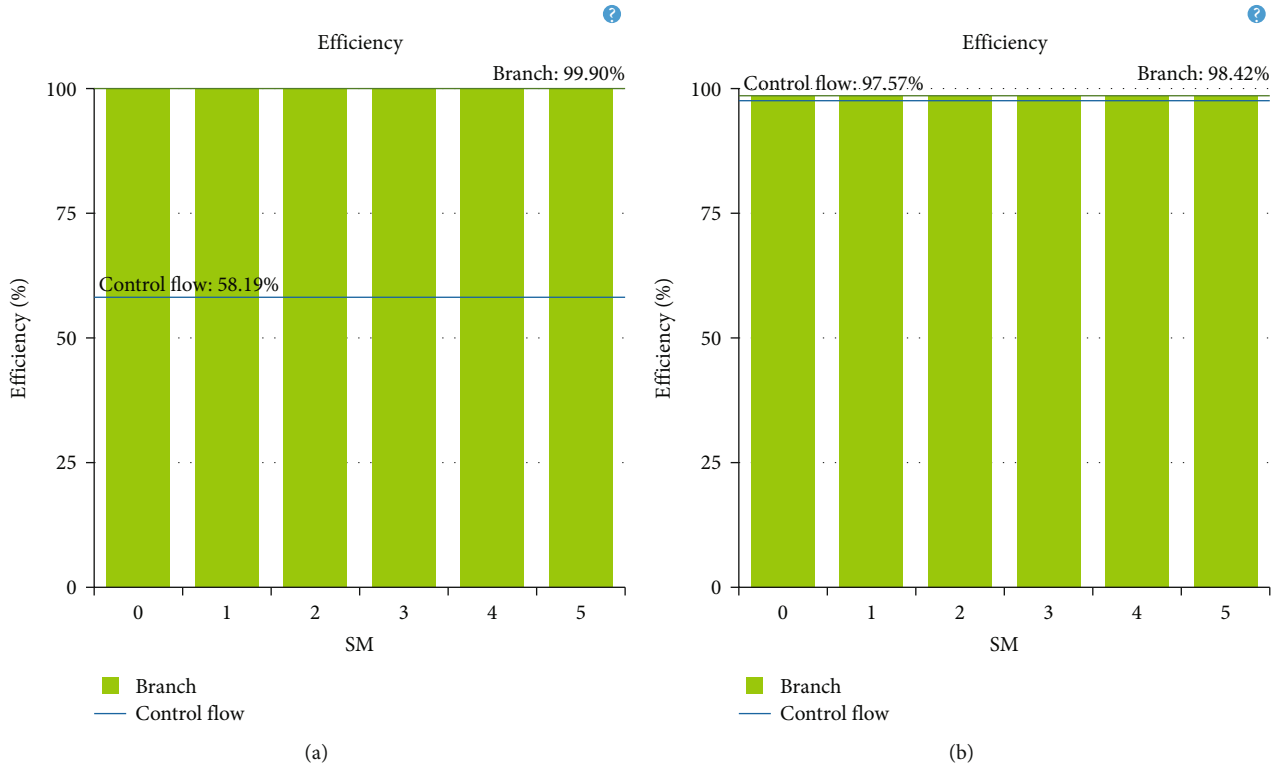
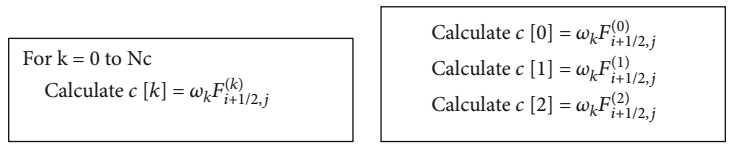


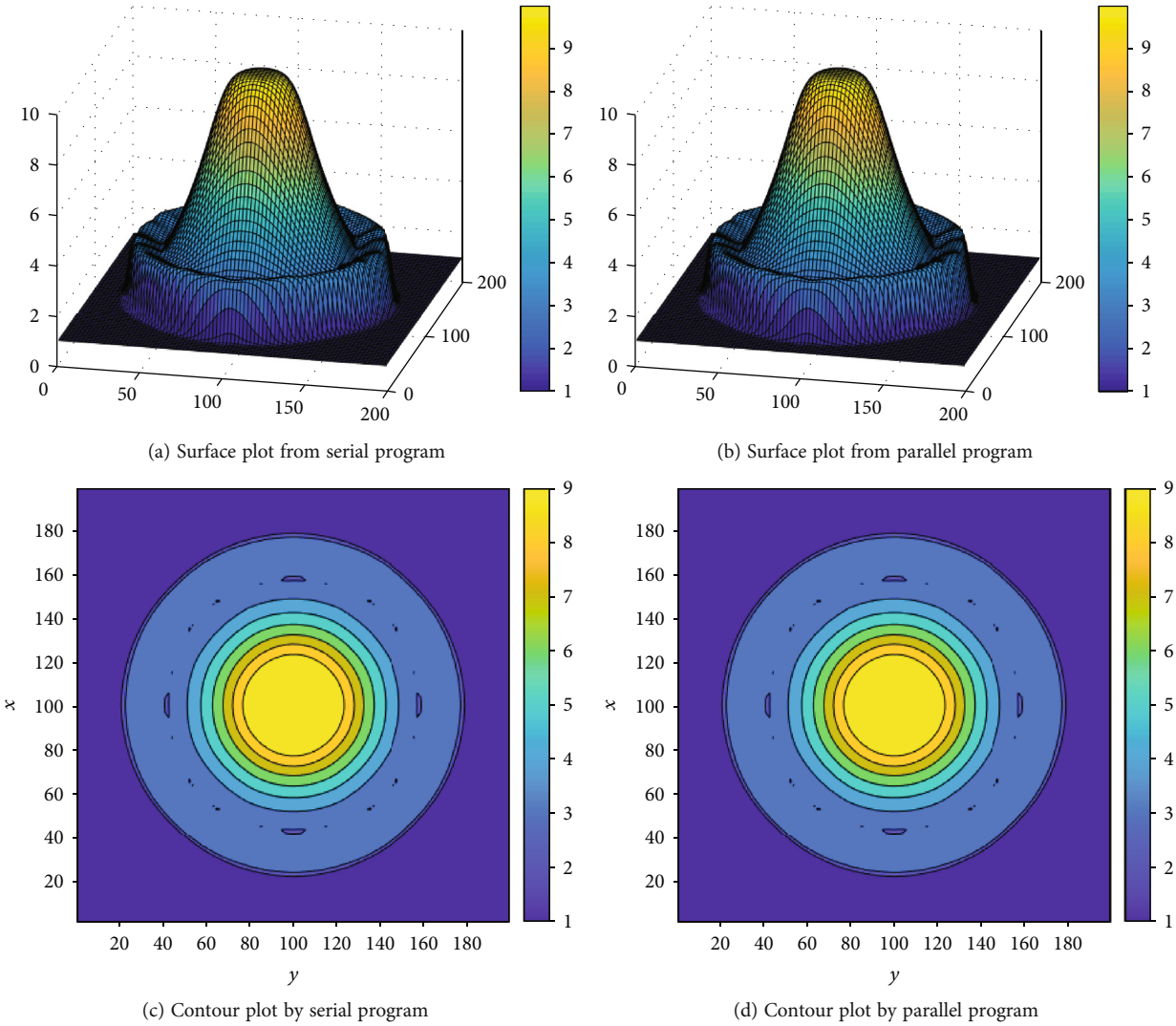

Figure 13: Control flow efficiency: (a) same launch configuration for computing flux in both directions and (b) different launch configuration for computing flux in each direction.

*4.5. Parallel Memory Pattern.* The previous implementation of parallel occupancy uses a 2D thread block which is natural for the 2D domain. In this proposed version, we will transfer the 2D data to the device using a 1D array. Therefore, we refactor the code to launch both hotspot kernels using a 1D thread block. We assume that threads can access memory consecutively and coalesce. Unexpectedly, this implementation is slower than the previous one as shown by the Nsight report in Figure 12(a). The computation of flux is slower only in the $y$ directions. We launch both kernels using 64 threads in one dimension of the thread block. When we launch the cudaRunKernelFluxy kernel that




(a) (b)

Figure 14: Optimization of flux computation by loop unroll: (a) normal loop and (b) loop unroll.




(a) Surface plot from serial program

(b) Surface plot from parallel program

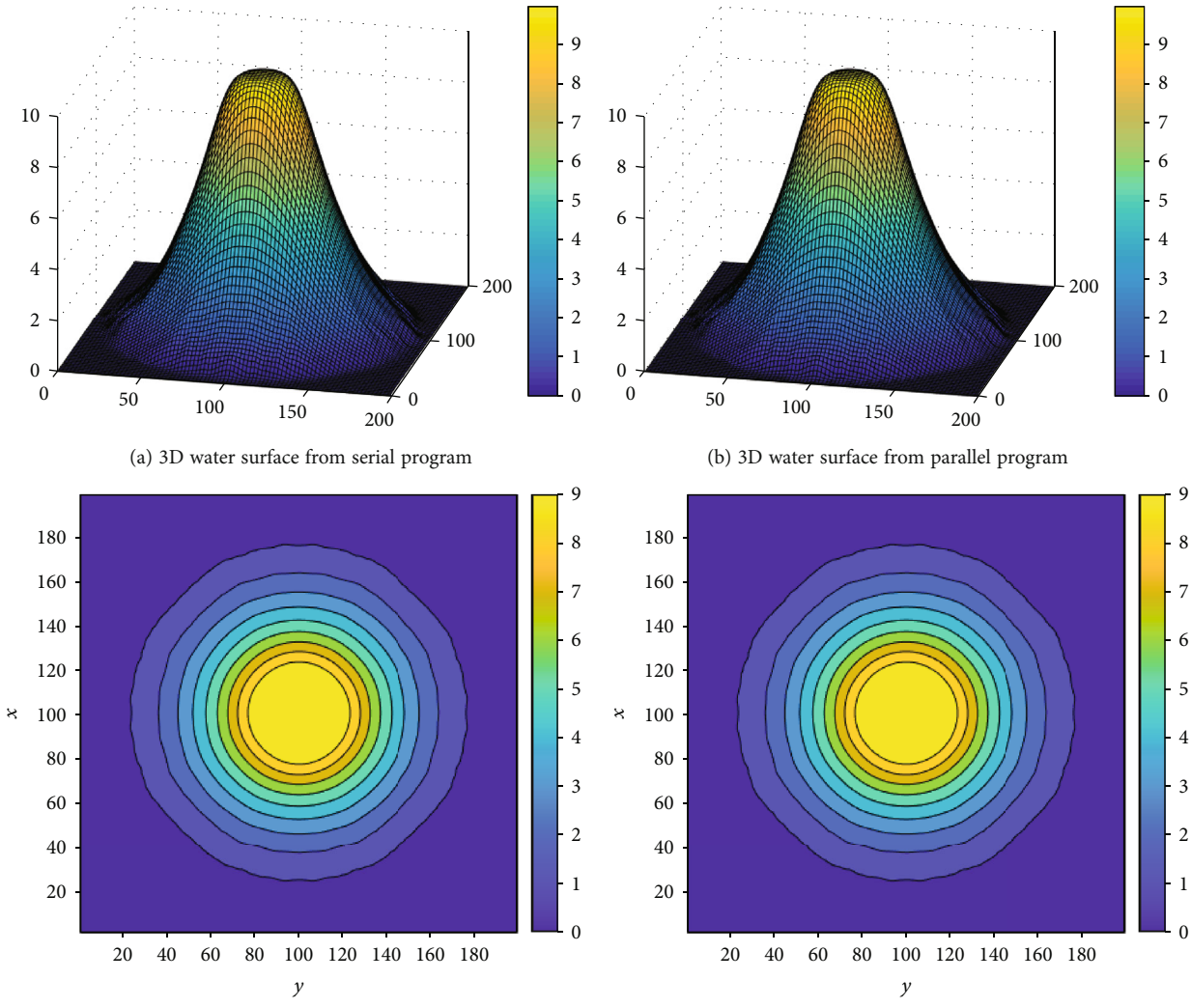
(c) Contour plot by serial program

(d) Contour plot by parallel program

Figure 15: Surface and contour plots for serial and parallel programs of circular dam break on a wet bed.

computes flux in the $y$ direction using $(1, 64)$ thread block as shown in Figure 12(b), and the kernel runs faster. We design to transfer data from CPU to GPU using cudaMemcopy that can transfer data in all cells from CPU to 1-dimensional array on GPU consecutively (see Figure 6).

According to the Nsight report in Figure 13, we found that the control flow efficiency is improved after we change the block dimension from $(64, 1)$ to $(1, 64)$ in the kernel cudaRunKernelFluxy. Control flow efficiency is defined by the ratio of the active thread to the maximum number of threads per warp. Figure 13 shows the difference in the control flow efficiency for the same launch configuration block dimension in both the $x$ and $y$ dimensions; see Figure 13(a). The difference in the launch configuration between the $x$ and $y$ directions is shown in Figure 13(b). When the block dimension is changed to $(1, 64)$, all threads in the same block are executing the code and can access memory consecutively at the same padding. Due to improving the control flow efficiency, our third parallel implementation (called parallel memory pattern) has 1.3x speedup




(a) 3D water surface from serial program

(b) 3D water surface from parallel program

(c) Contour plot from serial program

(d) Contour plot from parallel program

Figure 16: Surface and contour plots for serial and parallel programs of circular dam break on a dry bed.

compared with the parallel occupancy for a rectangular dam break problem. All results will be summarized again in Results.

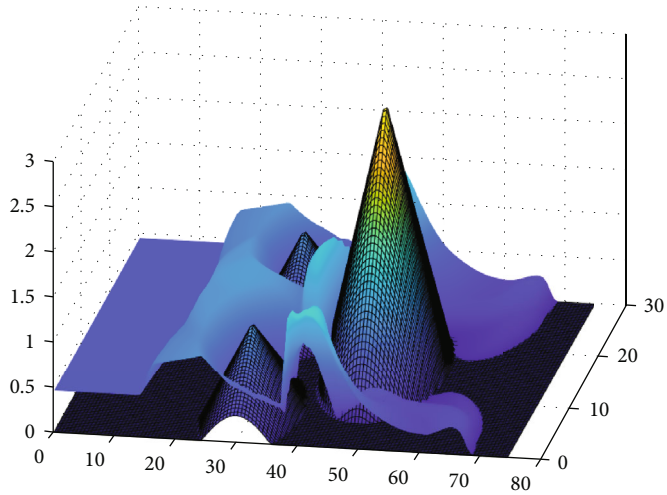
*4.6. Parallel Unroll.* From our various experiments, we found from the profiler results that about 78% of the computations involve calculating flux. This is performed in the kernels cudaRunKernelFluxy and cudaRunKernelFluxx. Both kernels call a device function computefluxxhydro. Inside the function, there is a loop for computing TVD-WAF flux when the number of waves in the Riemann problem ($N_c$) is set to be 2; for details, see equation (7). Thus, we have a small loop to compute the speed of waves as shown in Figure 14(a) for every finite volume cell. To optimize the computational time of this point, we propose to unroll this loop and calculate it separately. The idea is shown in Figure 14(b).

The loop of the wave speed is unrolled, and we call this fourth parallel implementation a parallel loop unroll and its computational time is 1.2 times faster than the parallel memory pattern for a rectangular dam break problem.

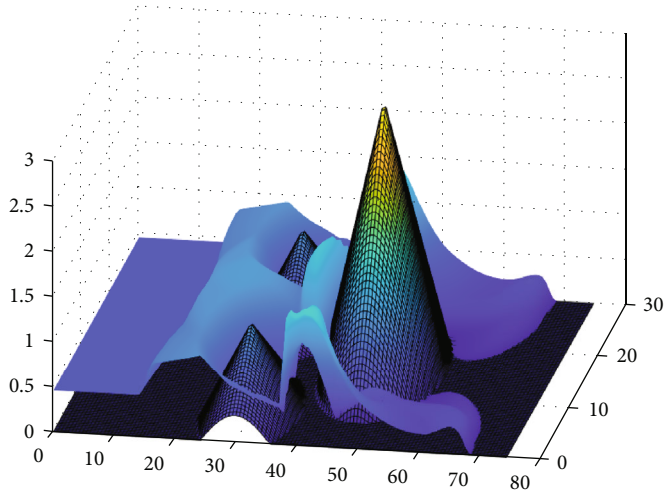
## 5. Accuracy Tests

Before applying our various programs to solve real flow problems, the accuracy of our implementations is verified by comparing the output with the previous serial implementations in [8, 9]. Accuracy tests are performed for 3 problems: a circular dam break on a wet bed, a circular dam break on a dry bed, and dam break flows over three humps.

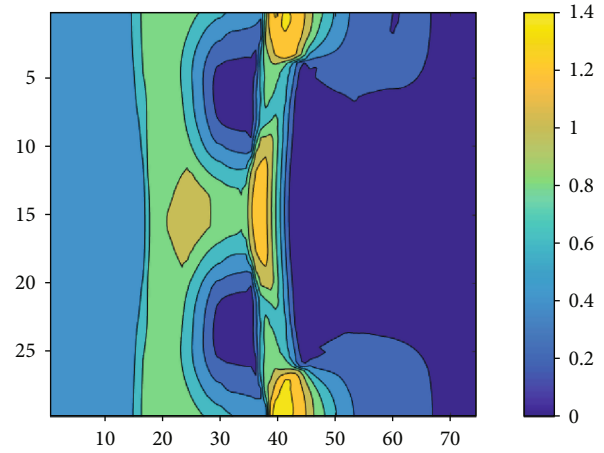
*5.1. Circular Dam Break on Wet Bed.* The domain has 200 $\times$ 200 rectangular grid cells. Inside the domain, there is a cylindrical dam located at the center of the domain. The radius of the dam is 50 meters. The calculation is performed without source terms. The initial water depth inside the dam is 10 meters and outside the dam is 1 meter. The surface and contour plots of both serial and parallel implementations are shown in Figure 15. Both serial implementations (cell- and edge-based) provide the same results as shown in Figures 15(a) and 15(c), while the results from the parallel program are shown in Figures 15(b) and 15(d). The



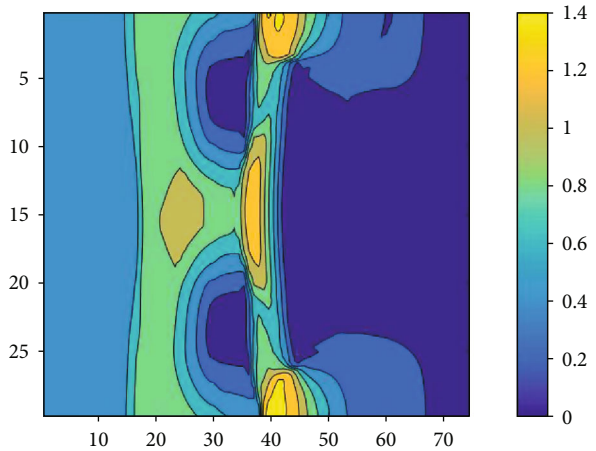

(a) Surface plot from serial program




(b) Surface plot from parallel program




(c) Contour plot from serial program




(d) Contour plot from parallel program

FIGURE 17: Dam break flows over three humps at $t = 12$, see comparisons with [8]: (a) surface plot $h + z$ from serial program, (b) surface plot $h + z$ from parallel program, (c) contour plot from serial program, and (d) contour plot from parallel program.

results from all parallel versions are identical. Thus, we pick one result to represent the results from all the parallel versions. If we regard the serial result as an exact solution, we can then measure the root mean square error (RMSE) at $t = 3$ between the serial and parallel programs. It is found that the RMSE is 0.000001, which shows very close

Table 2: Speedup compared with previous program for various problems.

| | Rectangular dam break | | Circular dam break on wet bed | | Circular dam break on dry bed | | Dam break flows over three humps | |
|---|---|---|---|---|---|---|---|---|
| | 16k | 1M | 16k | 1M | 16k | 1M | 16k | 1M |
| Serial: cell-based | Baseline | Baseline | Baseline | Baseline | Baseline | Baseline | Baseline | Baseline |
| Serial: edge-based | 1.90 | 1.92 | 1.87 | 1.24 | 1.92 | 1.52 | 1.74 | 1.81 |
| Parallel V1 | 6.55 | 9.70 | 11.19 | 15.81 | 9.92 | 14.05 | 7.91 | 9.67 |
| Parallel occupancy | 2.42 | 2.79 | 1.81 | 2.02 | 1.80 | 2.05 | 2.32 | 2.50 |
| Parallel memory pattern | 1.33 | 1.02 | 1.27 | 1.01 | 1.24 | 1.01 | 0.99 | 1.01 |
| Parallel unroll | 1.19 | 1.20 | 1.20 | 1.23 | 1.18 | 1.22 | 1.26 | 1.15 |

Table 3: Speedup compared to baseline program for various problems.

| | Rectangular dam break | | Circular dam break on wet bed | | Circular dam break on dry bed | | Dam break flows over three humps | |
|---|---|---|---|---|---|---|---|---|
| | 16k | 1M | 16k | 1M | 16k | 1M | 16k | 1M |
| Serial: cell-based | Baseline | Baseline | Baseline | Baseline | Baseline | Baseline | Baseline | Baseline |
| Serial: edge-based | 1.90 | 1.92 | 1.87 | 1.24 | 1.92 | 1.52 | 1.74 | 1.81 |
| Parallel V1 | 12.45 | 18.67 | 20.98 | 19.66 | 19.03 | 21.39 | 13.74 | 17.47 |
| Parallel occupancy | 30.18 | 52.20 | 38.04 | 39.71 | 34.23 | 43.82 | 31.87 | 43.58 |
| Parallel memory pattern | 40.24 | 52.56 | 48.19 | 40.22 | 42.55 | 44.12 | 31.49 | 44.02 |
| Parallel unroll | 48.07 | 63.35 | 57.72 | 49.39 | 50.19 | 53.67 | 39.64 | 50.60 |

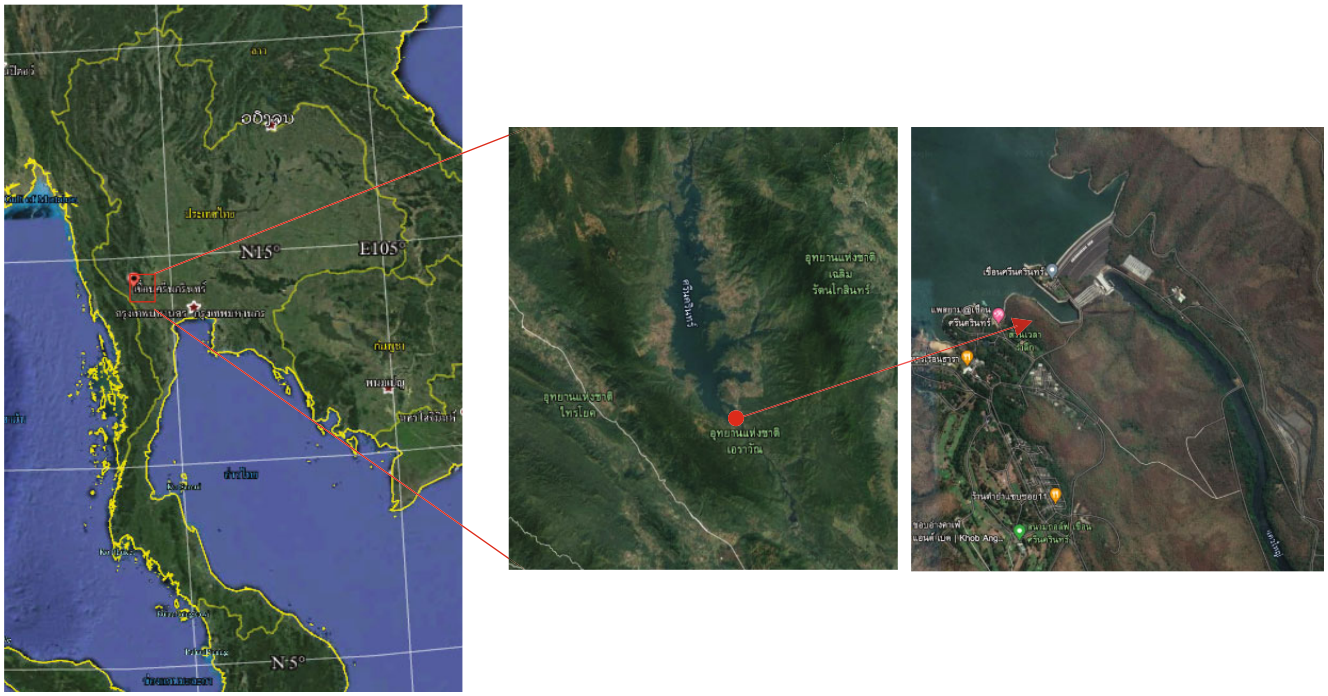
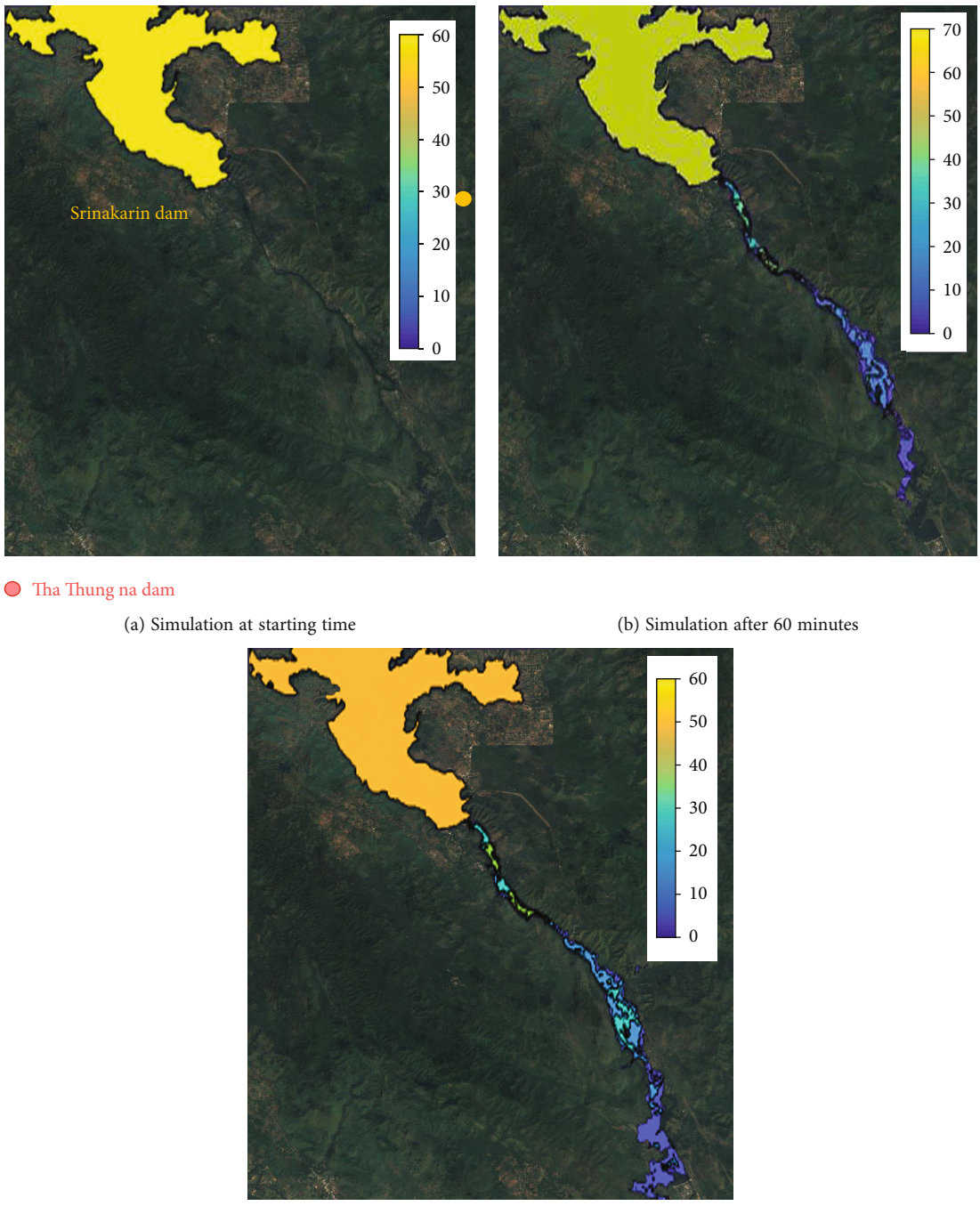



Figure 18: Location of Srinakarin Dam, Thailand, and study area in our simulation.

agreement for all our implementations. This small discrepancy may be from the difference in floating-point architecture between GPU and CPU as reported in [27].

*5.2. Circular Dam Break on Dry Bed.* In this problem, the water depth inside a dam is 10 meters and outside the dam is 0 meters. Figure 16 compares the surface and contour plots of both the serial and parallel implementations. Both serial implementations (cell- and edge-based) provide the same results. All parallel implementations give the same output as well. Both serial and parallel programs provide nearly equal outputs at time $t = 3$ with the RMSE = 0.0050. Since




(a) Simulation at starting time

(b) Simulation after 60 minutes

(c) Simulation after 90 minutes

Figure 19: Contour plots of Srinakarin Dam break simulations.

we have applied the rectangular grid cell in this simulation, a better result of circular shape can be obtained by increasing the mesh resolutions. Another way is the application of unstructured mesh that can capture the complex shape of water level; see, for instance, [2, 3].

*5.3. Dam Break Flows over Three Humps.* In this problem, the domain size is $75 \times 30$ meters. Unlike the previous test, the bottom slope is not flat. The initial water depth $h + z$ is 1.875 meters for $0 < x < 16$, where $z$ is a bottom depth defined by equation (15). The computational domain is divided into $128 \times 128$ uniform grid cells with the Manning coefficient set at 0.018. This problem demonstrates a strong interaction between the high gradient water depth and the friction bottoms for the wet and dry areas. Figure 17 compares the surface and contour plots from both the serial and parallel implementations. The visual results are consistent, and they agree well with the results in [8]. Both serial implementations provide the same result. All parallel implementations also provide very similar results with RMSE = 0.0006.

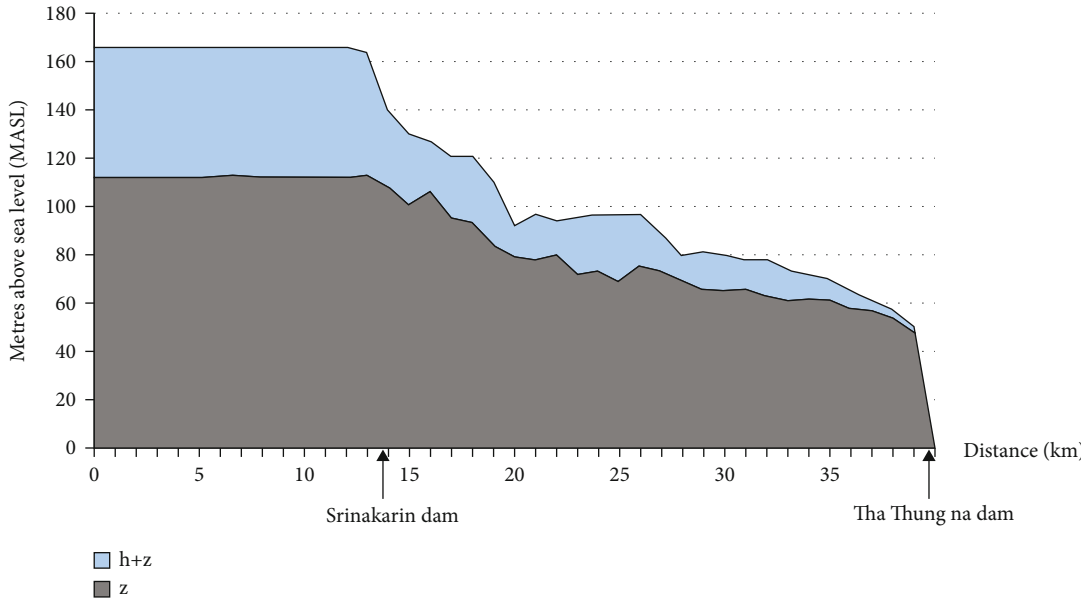



Figure 20: Simulation results after 90 minutes (real time), plots of water depth and bottom elevation.

$$z(x,y) = \max \begin{cases} 0, 1 - \frac{1}{8}\sqrt{(x-30)^2 + (y-6)^2}, \\ 1 - \frac{1}{8}\sqrt{(x-30)^2 + (y-24)^2}, \\ 3 - \frac{3}{10}\sqrt{(x-47.5)^2 + (y-15)^2}. \end{cases} \quad (21)$$

## 6. Performance of Implementations

The performance for implementations is compared based on their individual running times. The calculations can be divided into 3 parts: initialization, numerical computation, and visualization. For the serial programs, initialization includes reading input parameters and allocating and initializing data in the main memory. For parallel programs, initialization adds the overhead of transferring the data from the host memory (main memory) to the device memory (graphics card memory). However, we do not take initialization and visualization times into account. We compare only the numerical computation of the serial and parallel implementations because the initialization and visualization times are small compared with the numerical computation (about 0.003%). Another reason is that the overall running time is varied by some parameters such as the number of iterations or grid cells. Therefore, comparing only numerical computation gives a better view of the code optimization effect.

All implementations are tested with four dam break problems as shown in Table 2. The domain sizes are $128 \times 128$ (16k) and $1024 \times 1024$ (1M) cells. The CPU specifications in our experiments are AMD Ryzen 5 1500x Quad-Core Processor 3.5 GHz with 16 GB memory, and for the GPU, the specifications are GeForce GTX 1050Ti, with 4 GB memory (see Table 1). Since the running time is not equal for every run, we run the program 10 times and report the average. The size of the thread block is $16 \times 16$, $64 \times 1$, or $1 \times 64$.

As shown in Table 2, all parallel versions are faster than the serial versions and the improved parallel version is faster than the previous version. As shown in Table 3, the parallel unroll for the $1024 \times 1024$ domain shows the best performance (63.35x) for the rectangular dam break problem. The ratio of computational time for the serial to the parallel program is close to the ratio of the domain. The large domain achieves a better speedup than the small domain. The optimal number of finite volume cells in each direction also affects the speedup of parallel algorithms.

It should be noted that the speedup from CPU to GPU is depended on both hardware architecture and computer code. If one compares the same code on different hardware, speedup will be different. Then, our objective is to implement the CUDA program using Nsight report to develop better parallel algorithms from the first serial program.

## 7. Srinakarin Dam Break Simulation

In this section, we will apply our developed algorithms to simulate a real-world problem. We study the Srinakarin Dam located on the KkwaeYai River, Kanchanaburi province, Thailand. The Electricity Generating Authority of Thailand (EGAT) constructed this dam for water supply and power generation over forty years ago. Its height from the ground is about 140 meters, and it has an embankment dam 610 meters long. The study area covers 419 $km^2$ from 14.125000°N to 14.300000°N and from 99.000000°E to 99.142300°E; see Figure 18 for the location.

The experiment is set up for a partial dam break simulation. The bottom elevation was obtained from the NASA Shuttle Radar Topographic Mission (SRTM) in a Digital Elevation Model (DEM) with a resolution of $90 \times 90$ m (http://srtm.csi.cgiar.org). The domain size is $512 \times 320$ cells. From the EGAT report, we set the initial water height in the Srinakarin Dam at 60 meters and the flow velocities in the $x$ and $y$

Table 4: Running time and speedup for Srinakarin Dam break simulation.

| | Time (ms) | Speedup compared with previous program | Speedup compared with baseline |
|---|---|---|---|
| Serial: cell-based | 29,156,017 | Baseline | Baseline |
| Serial: edge-based | 15,845,661 | 1.84 | 1.84 |
| Parallel V1 | 1,689,703 | 9.38 | 17.26 |
| Parallel occupancy | 647,045 | 2.61 | 45.06 |
| Parallel memory pattern | 621,090 | 1.04 | 46.94 |
| Parallel unroll | 508,607 | 1.22 | 57.33 |

directions are set at zero. This implies that the water is stationary over the dam, and a mass of water is released suddenly to the KkwaeYai River downstream. The Manning coefficient is assumed to be zero, since we consider a very strong interaction between water depth and bottom elevation on the KkwaeYai River. The sensitivity in time step size is tested and found that $dt < 0.2$ provides the same results in our simulation. Then, we set $dt = 0.1$. The better choice is the application of adaptive time step to satisfy the CFL condition during time integration; see [11]. But for easy control of the measurement of speedup between parallel and serial algorithms, we fix the value of time step in all programs to avoid a floating-point error for calculating time step that will result in different computational time for the same problem.

The simulation after 90 minutes shows that the water in the Srinakarin Dam flows along the river and reaches the Tha Thung na Dam about 27 kilometers downstream from the Srinakarin Dam. We compare the results of the serial and all of our parallel programs and find they all provide the same result. Figures 19(a)–19(c) show the water flow and flood plain along the river at starting times of 60 and 90 minutes, respectively. The water height at the Tha Thung na Dam is about 4 meters (see Figure 20). The water height depends on the initial conditions and the dam break type.

We compare the performance of each implementation. The running time of the serial programs on the AMD Ryzen 5 1500x Quad-Core Processor 3.5 GHz with 16 GB memory is more than the real time (5400 sec), while the running time of parallel programs on the GeForce GTX 1050Ti with 4 GB memory is faster. Table 4 shows the results from the optimized parallel program. The parallel unroll program provides the best performance, being 57.3x faster than the baseline of the serial program. Furthermore, it is faster than the first parallel program (parallel V1) at 3.32x.

## 8. Conclusion

Serial and parallel programs for simulating shallow water flows are presented in this work. We develop two serial programs which are cell-based and edge-based versions. A cell-based program is a logical implementation of the governing equations. The edge-based application is 1.9x faster than the cell-based program. Our first CUDA parallel program is 6.55 times faster than the edge-based program. We use Nsight to find an area of improvement. The hotspot has low occupancy which can be improved by decreasing the number of registers per thread. This is counterintuitive because using more registers can increase speedup for a serial program. However, for the CUDA architecture, registers are the sharing resources between threads in the same block. Lowering the number of registers causes more threads to be run at the same time. Our second parallel implementation is aimed at better memory coalescence. However, coalescence access is 1.33 times faster due to the improvement in control flow efficiency. Finally, the last CUDA program uses loop unrolls in the main computation, resulting in being 3.84 times faster than our first CUDA implementation. The loop unroll approach is proposed to reduce the computational time for calculating fluxes in the Riemann problem of the weighted average flux (WAF) method. Generally, the WAF method contains a small loop of wave components in each cell and we unroll this approximation separately to provide optimized performance using the last parallel program. This concept can be extended to other problems that apply the WAF method to obtain numerical flux at cell interfaces.

## Data Availability

The data that support the findings of this study are openly available in http://srtm.csi.cgiar.org.

## Disclosure

This study extends our preliminary work presented at the 11th International Conference on Computer Science & Education (ICCSE) 2016 according to the following link: https://ieeexplore.ieee.org/document/7581572.

## Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this article.

## Acknowledgments

This research was partially supported by International SciKU Branding (ISB), Faculty of Science, Kasetsart University, Bangkok, Thailand.

## References

[1] E. Gaburro, M. J. Castro, and B. M. Dumbser, "A well balanced diffuse interface method for complex nonhydrostatic free surface flows," *Computers and Fluids*, vol. 175, pp. 180–198, 2018.

[2] A. Lacasta, M. Morales-Hernández, J. Murillo, and P. García-Navarro, "An optimized GPU implementation of a 2D free

surface simulation model on unstructured meshes," *Advances in Engineering Software*, vol. 78, pp. 1–15, 2014.

[3] B. F. Sanders, J. E. Schubert, and R. L. Detwiler, "ParBreZo: a parallel, unstructured grid, Godunov-type, shallow-water code for high-resolution flood inundation modeling at the regional scale," *Advances in Water Resources*, vol. 33, no. 12, pp. 1456–1467, 2010.

[4] J. Langguth, N. Wu, J. Chai, and X. Cai, "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 120–131, 2015.

[5] A. Navas-Montilla, C. Juez, M. J. Franca, and J. Murillo, "Depth-averaged unsteady RANS simulation of resonant shallow flows in lateral cavities using augmented WENO-ADER schemes," *Journal of Computational Physics*, vol. 395, pp. 511–536, 2019.

[6] J. Murillo and A. Navas-Montilla, "A comprehensive explanation and exercise of the source terms in hyperbolic systems using Roe type solutions. Application to the 1D-2D shallow water equations," *Advances in Water Resources*, vol. 98, pp. 70–96, 2016.

[7] M. de la Asunción, M. J. Castro, E. D. Fernández-Nieto, J. M. Mantas, S. O. Acosta, and J. M. González-Vida, "Efficient GPU implementation of a two waves TVD-WAF method for the two- dimensional one layer shallow water system on structured meshes," *Computers & Fluids*, vol. 80, pp. 441–452, 2013.

[8] S. Vichiantong, T. Pongsanguansin, and M. Maleewong, "Flood simulation by a well-balanced finite volume method in Tapi River Basin, Thailand, 2017," *Modelling and Simulation in Engineering*, vol. 2019, 13 pages, 2019.

[9] T. Pongsanguansin, M. Maleewong, and K. Mekchay, "Shallow-water simulations by a well-balanced WAF finite volume method: a case study to the great flood in 2011, Thailand," *Computational Geosciences*, vol. 20, no. 6, pp. 1269–1285, 2016.

[10] D. H. Zhao, H. W. Shen, J. S. Lai, and G. Q. T. III, "Approximate Riemann solvers in FVM for 2D hydraulic shock wave modeling," *Journal of Hydraulic Engineering*, vol. 122, no. 12, pp. 692–702, 1996.

[11] E. F. Toro and P. L. Roe, "Riemann problems and the WAF method for solving the two-dimensional shallow water equations," *Series A: Physical and Engineering Sciences*, vol. 338, no. 1649, pp. 43–68, 1992.

[12] H. R. Vosoughifar, A. Dolatshah, and S. K. Sadat Shokouhi, "Discretization of multidimensional mathematical equations of dam break phenomena using a novel approach of finite volume method," *Journal of Applied Mathematics*, vol. 2013, Article ID 642485, 12 pages, 2013.

[13] M. Hasert, K. Masilamani, S. Zimny et al., "Complex fluid simulations with the parallel tree-based Lattice Boltzmann solver Musubi," *Journal of Computational Science*, vol. 5, no. 5, pp. 784–794, 2014.

[14] A. R. Brodtkorb, M. L. Sætra, and M. Altinakar, "Efficient shallow water simulations on GPUs: implementation, visualization, verification, and validation," *Computers & Fluids*, vol. 55, pp. 1–12, 2012.

[15] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using GPU architectures," *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2014.

[16] L. S. Smith and Q. Liang, "Towards a generalised GPU/CPU shallow-flow modelling tool," *Computers & Fluids*, vol. 88, pp. 334–343, 2013.

[17] M. Burtscher and K. Pingali, "An efficient CUDA implementation of the tree-based Barnes Hut *n*-body algorithm," in *GPU Computing Gems Emerald Edition*, Morgan Kaufmann Publishers Inc., 2011.

[18] K. Treenath, S. Worasait, and M. Montri, "Lattice Boltzmann method for two-dimensional incompressible Navier-Stokes equation with CUDA," in *TENCON 2014 - 2014 IEEE Region 10 Conference*, pp. 1–5, Bangkok, Thailand, 2014.

[19] M. Joselli, J. R. . S. Junior, E. W. Clua, A. Montenegro, M. Lage, and P. Pagliosa, "Neighborhood grid: a novel data structure for fluids animation with GPU computing," *Journal of Parallel and Distributed Computing*, vol. 75, pp. 20–28, 2015.

[20] M. Lastra, J. M. Mantas, C. Ureña, M. J. Castro, and J. A. García-Rodríguez, "Simulation of shallow-water systems using graphics processing units," *Mathematics and Computers in Simulation*, vol. 80, no. 3, pp. 598–618, 2009.

[21] R. Vacondio, A. Dal Palù, and P. Mignosa, "GPU-enhanced finite volume shallow water solver for fast flood simulations," *Environmental Modelling & Software*, vol. 57, pp. 60–75, 2014.

[22] S. Ha, J. Park, and D. You, "A GPU-accelerated semi-implicit fractional-step method for numerical solutions of incompressible Navier-Stokes equations," *Journal of Computational Physics*, vol. 352, pp. 246–264, 2018.

[23] C. H. Pan, S. Q. Dai, and S. M. Chen, "Numerical simulation for 2D shallow water equations by using Godunov-type scheme with unstructured mesh," *Journal of Hydrodynamics*, vol. 18, no. 4, pp. 475–480, 2006.

[24] E. Audusse, F. Bouchut, O. Bristeau, R. Klein, and B. Perthame, "A fast and stable well-balanced scheme with hydrostatic reconstruction for shallow water flows," *SIAM Journal on Scientific Computing*, vol. 25, no. 6, pp. 2050–2065, 2004.

[25] NVIDIA Corporation, *Cuda C Best Practices Guide, DG-05603-001_v10.1*, 2019.

[26] NVIDIA Corporation, *Cuda C Programming Guide, PG-02829-001_v10.1*, 2019.

[27] NVIDIA Corporation, *Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs, TB-06711-001_v11.4*, 2021.