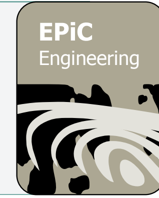
EPiC Series in Engineering

Volume 3, 2018, Pages 2105–2111

HIC 2018. 13th International Conference on Hydroinformatics



Multi-GPU Implementation of 2D Shallow Water Equation Code with Block Uniform Quad-Tree grids

Massimiliano Turchetto[1], Renato Vacondio[1], and Alessandro Dal Palù[2]

[1] Department of Engineering and Architecture, University of Parma, Parco Area delle Scienze 181/A, 43124, Parma, Italy

[2] Department of Mathematical Physical and Computer Sciences, University of Parma, Parco Area delle Scienze 53/A, 43124, Parma, Italy

Abstract

This paper presents a multi Graphic Processing Unit (GPU) implementation of a 2D *shallow water equations* solver which is able to exploit the computational power of modern HPC clusters equipped with several GPUs on different nodes. The domain has been discretized by means of a Block Uniform Quadtree (BUQ) grid which allows to efficiently introduce variable resolution in a GPU-accelerated finite value code. In the present work the BUQ grid is decomposed into different partitions, and each partition is assigned to a dedicated GPU. Communications between different partitions are then handled by means of a Message Passing Interface (MPI) protocol. Computations and communications have been overlapped to reduce the overheads of the multi-GPU implementation. The *strong* scalability test shows an efficiency dropdown better than linear in the number of GPUs adopted by the simulation, and the *weak* scalability test shows that network overheads caused by border communication are completely maskable by GPU calculations.

1 Introduction

Numerical models based on 2D *Shallow Water Equations* have been proven robust and suitable for modeling flooding events. In particular, explicit finite volume schemes allow to accurately simulate flood propagation on real bathymetries. Notwithstanding these simulations have a good accuracy, the requested computational time can be very large, preventing the possibility of performing simulations over large domains. Different authors [1, 6–8] have managed to significantly reduce the simulation time by mapping the computational grid into a parallel architecture. Vacondio et al. developed the PARFLOOD solver, which takes advantage of CUDA GPU architecture [6], by adopting a new type of grid called *block uniform quad tree grid* (BUQ), which discretizes the domain using multiple resolution levels, thus allowing high resolution over areas of interest and lower resolution elsewhere. Simulations carried out using BUQ grids run up to 10 times faster than those performed on Cartesian grids and reach a ratio of physical to computational time of $12 - 15$, thus making quasi real-time simulations over big domains a viable option. However, simulation times on a single GPU can be decreased only via code optimization and hardware upgrade. Similarly, the dimension of input grids is limited by the memory available on the GPU. The aim of this paper is to overcome these limitations further extending the work described in [6] by allowing simulations of flooding events to be carried out on multi GPU architectures.

The key idea behind this project is to decompose the computational domain in different partitions which are simulated on dedicated GPUs (i.e., a different GPU for each partition). GPUs can be located on different nodes over a network and must be able to communicate through a Message Passing Interface (MPI) protocol. Nowadays, HPC systems which satisfy these requirements are common and provide communication frameworks which make the implementation of parallel applications much easier.

2 Numerical Model

In this section we briefly summarize the numerical model adopted, as a detailed description is beyond the purpose of this work. Further details can be found in [7]. The numerical model solves the 2D-*Shallow Water Equations* (SWE) through a finite volume scheme in the integral form [5]:

$$\frac{d}{dt}\int_A \mathbf{U} dA + \int_C \mathbf{H}\cdot\mathbf{n} dC = \int_A (\mathbf{S}_0 + \mathbf{S}_f)\, dA$$

where A is the area of the integration element, C the element boundary, **n** the outward unit vector normal to C, **U** the vector of the conserved variables, $\mathbf{H} = [\mathbf{F}, \mathbf{G}]$ the tensor of fluxes int the x and y directions, $\mathbf{S}_0$ and $\mathbf{S}_f$ the bed and friction slope source term, respectively. Conserved variables and Fluxes are defined as follows:

$$\mathbf{U} = \begin{bmatrix}\eta \\ uh \\ uvh\end{bmatrix}, \mathbf{F} = \begin{bmatrix}\frac{u^2}{h} \\ u^2h + \frac{1}{2}g(\eta^2 - 2\eta z) \\ uvh\end{bmatrix}, \mathbf{G} = \begin{bmatrix}\frac{u^2}{h} \\ uvh \\ v^2h + \frac{1}{2}g(\eta^2 - 2\eta z)\end{bmatrix}, \mathbf{S}_f = \begin{bmatrix}0 \\ -g\eta\frac{\partial z}{\partial x} \\ -g\eta\frac{\partial z}{\partial y}\end{bmatrix}$$

where h is the flow depth, u and v are the velocity components in the x and y directions, g is the gravitational acceleration, z is the bed elevation and $\eta = h + z$ is the free surface elevation above datum.

3 CUDA Implementation

The domain subdivision is done by means of a *Block Uniform Quad-Tree* grid (BUQ) [6]. Our definition of BUQ relies on the notion of *quadtree subdivision* given in [4] with some adaptation to our specific problem. A BUQ *subdivision* is a *quadtree subdivision*, Figure 1-a), in which each *leaf* of the quadtree is a matrix $M \times M$ - Figure 1-b) - where M is constant for all leaves (in our models we use $M \in \{8, 16\}$). We call *block* such matrix and *cells* the entries of the matrix. Each cell represents a square surface whose dimension is $2^i\Delta$ for $0 \leq i < k$ where $\Delta \in \mathbb{R}$ and $k \in \mathbb{N}^+$ are fixed. The integer i identifies the *resolution* of a block. Blocks sharing at least one *corner* are called *neighbours* or *adjacent* blocks, Figure 1-c). Furthermore, we impose an important constraint on our spatial subdivision: each pair of adjacent blocks can differ for at most one resolution level.

The memory representation of a BUQ grid can be seen as a bidimensional array tiled up with blocks having an unique identifier called *index*. Let us note that this representation does not preserve neighbouring relations between blocks [6](i.e., two adjacent blocks in the BUQ can be far away from each other in memory). Thus for keeping track of adjacency relations between blocks a directed graph $G(S, E)$ is defined, where S is a set of *indexes* and E is the set of pairs (i, j), where $i, j \in S$ and i, j are *neighbours*.

The simulation is performed on Nvidia *graphic processing units* (GPUs) which are able to spawn thousands of conurrent threads. In the NVIDIA™ Compute Unified Device Architecture (CUDA) framework herein adopted, threads are grouped inside blocks which in turn are grouped inside a grid.

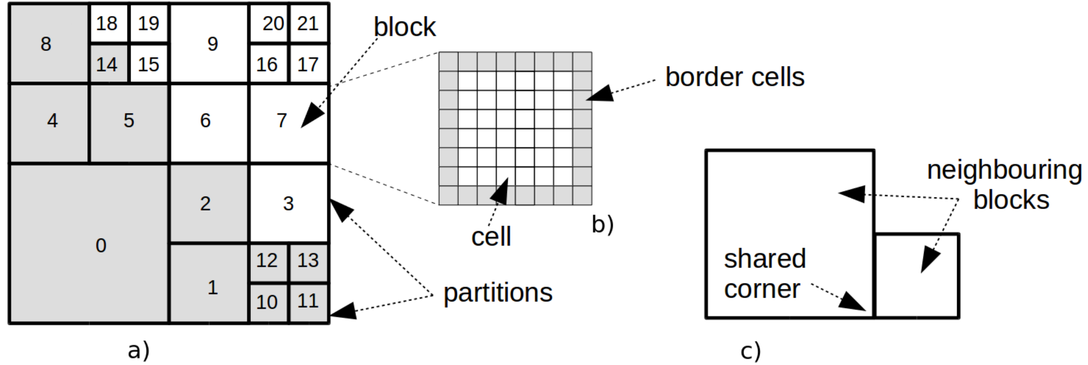


Figure 1: a) Quad Tree Subdivision of a domain in 22 blocks. The set of blocks is split into two partitions respectively highlighted whit white and grey areas. b) Each block is a matrix of $M \times M$ cells, where M is the same for all blocks. c) Neighbours blocks share a corner.

There is a one-to-one mapping between the CUDA environment and our spatial models, namely each spatial block is mapped into a CUDA block and each computational cell is mapped into a CUDA thread.

Although a GPU offers the possibility to perform a high number of calculations in parallel, the simulation time can still be unsatisfactory for input grids having order of 10^6 - 10^7 cells. Furthermore, the memory of a single GPU (up to 16GB) is rather limited for our application and so is the dimension of the input models. To overcome these limitations, the Section 4 presents an extension of the PARFLOOD simulator [6] which takes advantage of the spatial decomposition to perform Multi GPU simulations over a network.

Example 3.1. In Figure 1 the set of block indexes is $S = \{0, 1, \ldots, 21\}$. The neighbors of the block 1 are $0, 2, 3, 10, 12$. Each block has $8 \times 8 = 64$ cells. The GPU simulation spawns 22×64 CUDA threads, each one computing the content of a cell based on the content of neighboring cells.

4 Multi GPU Implementation

In Section 3 a single GPU implementation of the *shallow water equations* solver has been briefly discussed. The aim of this section is to generalize the single GPU implementation to a scalable multi GPU version. While the former version stores the domain and the related data structures on a single GPU, the latter version decompose the global domain into different pieces, called *partitions*. Each partition is then assigned to a different GPU. The preprocessing stage presented in Section 4.1 describes the data structures which allow different partitions to exchange border information. In Section 4.2 a multi GPU algorithm is discussed. Finally, the Section 4.3 evaluates the performances of the algorithm by presenting both strong and weak scalability tests.

4.1 Preprocessing Stage

Before describing in detail the preprocessing stage, let us introduce some notation that will help to make the description more clear. The symbol S denotes the set of blocks indexes in a domain and will be referred to as the global set of blocks. The multi GPU implementation is done by partitioning the set S into a sequence of non-empty disjoint sets $S_1, \ldots, S_n$ where n is the number of processes over which the simulation is performed (e.g., in Figure 1-a $n = 2$). The symbol S_t will be called *partition*. Let us

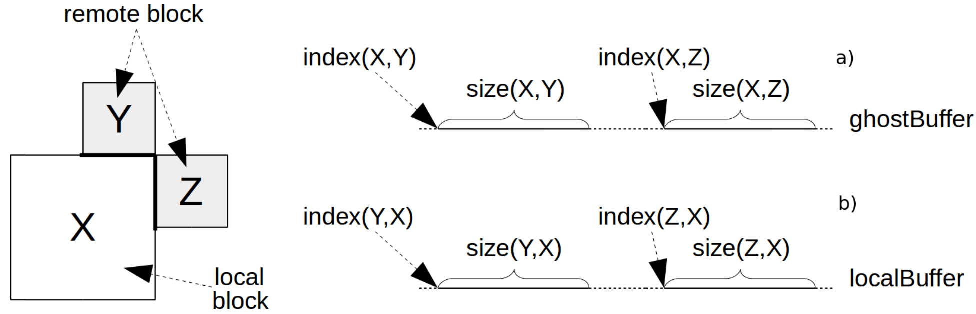


Figure 2: a) Mapping of remote blocks Y and Z into the *ghostBuffer* inside an area reserved for X readings. b) Mapping of local block X into the localBuffer which is sent to the partitions containing Y and Z.

note that n is an input to the multi GPU simulation and does not change during the computation. Each partition S_t is simulated on a different node p_t equipped with a GPU. Recall that a global domain can be represented by means of an adjacency graph $G(S, E)$, where S denotes the global set of blocks and E denotes the neighboring relations between blocks. The notation $neigh(b)$ indicates the set of neighbors of a block $b \in S$. We further assume that the global domain is *statically partitioned* into n sets (i.e., the content of each partition does not change during the simulation).

During the partitioning stage, a *blocks distribution table* T is computed for keeping track of the distribution of blocks among different partitions. Formally, for each block $b \in S$, $T(b) = t$ if and only if $b \in S_t$. In our implementation t is an integer, called *rank*, which uniquely identifies the process p_t. During the preprocessing stage further information is to be computed to allow neighboring domains to exchange border information. With this purpose each process p_t creates three different disjoint sets which we call *internalBlocks*, *borderBlocks*, *ghostBlocks*. The first is the set of blocks $b \in S_t$ such that all neighbors of b belong to S_t (i.e., $neigh(b) \subset S_t$). The second is the set of blocks $b \in S_t$ such that b has at least one remote neighbor, (i.e., $\exists r \in neigh(b)$ s.t. $r \notin S_t$). The third is the set of blocks $r \in S \setminus S_t$ s.t. r has at least one neighbor in S_t (i.e., $\exists b \in neigh(r)$ s.t. $b \in S_t$).

This implementation does not need any assumption on how blocks are distributed among different partitions, however we prefer to minimize exchanged boundaries by using partitions with a single self-avoiding closed fronteer (Figure 1-a).

Example 4.1. In Figure 1 the global set of blocks is $S = \{0, \dots, 21\}$. The domain is split into two partitions, $S_1 = \{0, 1, 2, 4,5, 8, 10,\dots, 14\}$ simulated on node p_1, and $S_2 = S \setminus S_1$, simulated on node p_2. The set of blocks associated with the partition S_1 are *internalBlocks*$= \{0, 4, 10, 11\}$, *ghostBlocks*$=$ $\{3, 6, 7, 9, 15, 18, 19\}$ and *borderBlocks*$= \{1, 2, 5, 8, 12, 13, 14\}$. For all $b \in S_1$ $T(b) = 1$ and for all $b \in S_2$ $T(b) = 2$.

During the preprocessing stage, each process p_t defines two distinct memory buffers, the *ghostBuffer* for storing information received from remote partitions and the *localBuffer* for writing local border information which needs to be sent to remote partitions. (Figure 2).

We detail how to precompute some additional information, in order to let each each block efficiently locate the correct offset inside *ghostBuffer* (resp. *localBuffer*) where to read (resp. to write) the data received from (resp. to send to) some remote block. For this purpose, we introduce the *remote adjacency map* which is computed for each partition S_t (see Example 4.2). In particular, given a partition S_t, for each block index $x \in S$, the *remote adjacency map* $L_t(x)$ is defined as a list of tuples of the form

$(y, coord, index_{(x,y)}, size_{(x,y)})$. The number of tuples stored in $L_t(x)$ is equal to the number of remote neighbors of x. The first element y indicates a remote neighbor block located at the coordinate $coord$ from the point of view of x. If x is local to S_t then $index_{(x,y)}$ indicates the offset inside *localBuffer* where the information about x has to be written and sent to the block y. Otherwise, if x is not local to S_t, $index_{(x,y)}$ indicates the offset inside the *ghostBuffer* where y has to to read the information about x received from a remote partition. The last element, $size_{(x,y)}$, indicates either the number of grid cells to write into the *localBuffer* (if x is local), or the number of grid cells to read from the *ghostBuffer* (if x is remote). Let us note that $L_t(x)$ can be also the empty list, which means that the block x does not have any remote neighbors. Once a map L_t has been computed for each partition S_t, the information about how to read and write *ghostBuffer* and *localBuffer* is known. Notice that is also implicitly known how much space must be allocated for each one of these buffers. In particular, the length of the *ghostBuffer* is the sum of the $size$ elements of each tuple $t \in L(x)$, for all $x \in ghostBorder$. Similarly, the length of the *localBuffer* is the sum of the $size$ elements of each tuple $t \in L(x)$, for all $x \in localBorder$.

Example 4.2. In Figure 2 a local block x and two remote blocks y, z are shown. The map $L(x)$ contains the tuples $(y, N, index(x, y), size(x, y))$ and $(z, E, index(x, y), size(x, y))$ which tell x how to read y and z from the *ghostBuffer*. The map $L(y)$ contains the tuple $(x,S, index(y, x),size(y, x))$ and $L(z)$ contains the tuple $(x, W,index(z, x), size(z, x))$ which tells x how to write information for y and z in the *localBuffer*. Notice that y and z may belong to two different partitions.

Each process builds his own *remote adiacency map* with a local viewpoint. When exchanging buffers, a certain process p_t needs to know how other processes will pack and send their local information. To address this, we let adjacent partitions exchange their respective maps before the the simulations starts. In this way a node p_t can send directly his *localBuffer* to each neighbor, which in turn uses the map belonging to p_t to read it, and their local map to save the data into their *ghostBuffer*. If on one hand this method simplifies the implementation of the communication procedures, on the other hand it increases the network overhead of a factor proportional to the number of neighbors of the sending node. However we will show that such overheads are small and completely maskable by the computation of the GPU kernels.

4.2 Simulation Stage

The Listing 1 shows the pseudocode of the multi GPU implementation. Only the simulation halfstep is described as our main goal is to discuss the communication between different processes. For a complete version of the PARFLOOD implementation on a single GPU, please refer to [6].

Listing 1: Simulation Halfstep of the Multi GPU Implementation

```
1  // DeltaT calculation
2      dt = gpu_deltaT();
3  // Compute the minimum delta T among all partitions (blocking call).
4      dtmin = deltaT_Reduction(dt);
5  // BDO procedure
6      define_wettable_blocks( );
7  // MUSCL Reconstruction for internal blocks
8      gpu_muscl_reconstruction( internalBlocks );
9  // Sending and receiving the conserved variables
10     communication( localBuffer, ghostBuffer );
11 // MUSCL Reconstruction for border blocks
12     gpu_muscl_reconstruction( borderBlocks );
13 // Update Conserved Variables for internal blocks, flux and source terms
14     gpu_fluxes_calculation( internalBlocks);
15 // Sending and receiving the reconstructed variables
16     communication( localBuffer, ghostBuffer );
```

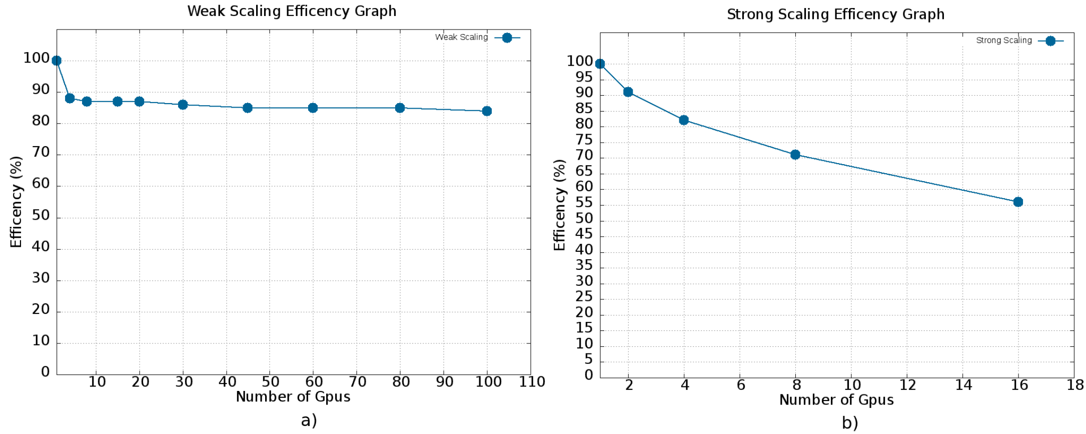


Figure 3: a) Resoluts of weak scalability tests up to 100 GPUs performed over a uniform resolution BUQ grid. b) Resoults of strong scalability tests performed over a circular dam break discretized using a multi resolution BUQ grid.

```
17 // Update Conserved Variables for border blocks, flux and source terms
18     gpu_fluxes_calculation( borderBlocks );
```

At the beginning of the temporal loop, the timestep Δt has to be calculated by each partition (line 2), and then an MPI reduction has to be performed in order to obtain the global minimum value (line 4). Since all the processes have to wait the response from the master process, this call represents the only one which can not be masked. The call plays a critical role in the execution time, especially if the computational load between different GPUs is unbalanced. The other communication calls (line 10, 16) are performed in a point-to-point nonblocking fashion, without the need to synchronize the *send* calls with the *receive* calls among neighbors. Instead, one process can perform all the *send* communications and then wait for all the remote data to arrive. In line 10 the local border *reconstructed variables* stored in the *localBuffer* are sent to the neighbor processes. Similarly remote *reconstructed variables* are received from neighbor processes into the *localBuffer*. The same thing happens for the communication performed in line 16, with the only difference that the buffers contain the *conserved variables*. Thanks to the asynchronous nature of GPU kernels, both the MUSCL reconstruction (line 8) and the Fluxes integration kernels (line 14) can execute concurrently with the communication calls right after them. This can be done because such kernels execute on *internalBlocks*, thus they don't need the border data. Once the MPI communication has been completed, CUDA kernels which operate on *ghostlBlocks* can be executed. In Section 4.3 we discuss the performance of this algorithm.

4.3 Numerical Tests

To assess the performance of our code we preformed both *strong* and *weak* scalability tests on the Piz Daint supercomputer located at the Swiss National Super Computing Centre. Piz Daint is an hybrid Cray XC40/XC50 system equipped with NVIDIA P100. Our GPU code has been tested on CUDA 8 and cray-mpich 7.6 as MPI implementation.

The weak scaling test shown in Figure 3-a has been performed by defining a fixed partition size of 125×32 blocks (i.e., $\approx 10^6$ cells) with the same resolution. This size is more than enough to mask the communication overhead. The efficency drops to 88% for two GPUS and from that point on there is a

small degradation which keeps the efficiency around 85% up to 100 GPUs. By profiling the code with *nvprof* we show that asynchronous communication steps (i.e., lines 10 and 16 of the Listing 1) do not affect the simulation time at all. The drop of efficiency is entirely caused by the reduction performed at line 4 (plus some minor overheads caused by data exchange between CPU and GPU).

The strong scalability test has been performed over a circular dam break model, containing a total of 24700 blocks (i.e $\approx 6.3 \cdot 10^6$ cells). For this test a BUQ grid (Section 3) is used and the partitioning is performed along one dimension. We enforce that each partition contains the same number of blocks. Figure 3-b shows that the efficiency dropdown varies from 10 to 15% by doubling the number of GPUs. By profiling our code with `nvprof` we noticed that the main factor which causes the loss of efficiency is the 1D partitioning. In fact, by doubling the number of GPUs involved in the simulation, the number of blocks is halved for each partition, however the the size of the borders remains almost the same (i.e., a small variation can happen due to the multiresolution grid). Thus the running time for the border kernels (lines 12 and 18 in the Listing 1) remains constant. Another factor which causes a loss of efficiency is related to the computational load of the kernels which execute on internal blocks (lines 8 and 14). In particular we observed that those kernels stick to an efficiency of 100% as long as there are at least 3000 blocks for each partition. This explains the efficiency dropdown increase for a number of GPUs higher than 8. The network overhead is caused only by the synchronization time, especially for 2 and 4 GPUs and then it becomes negligible. Instead, the asynchronous communication (lines 10 and 16) is completely masked by kernel execution, thus it does not affect the execution time, supporting our choice to send all the border data to each partition.

References

[1] M. Asunción, M.J. Castro, E. Fernández-Nieto, J.M. Mantas, S.O. Acosta, J.M. González-Vida (2013). Efficient gpu implementation of a two waves tvd-waf method for the two-dimensional one layer shallow water system on structured meshes, Comput. Fluids 80, 441-452

[2] A.R. Brodtkorb, M.L. Sætra, M. Altinakar (2012). Efficient shallow water simulations on GPUs: Implementation, visualization, verification and validation Comput. Fluids 55, 1-12

[3] NVIDIA CUDA, 2018, CUDA C Programming Guide. docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[4] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, 2008, Computational Geometry. Algorithms and Applications. Third Edition. Springer. 313–314.

[5] E.F. Toro (1999). Shock Capturing Methods for Free Surface Shallow Water Flows. Wiley, New York.

[6] R. Vacondio, A. Dal Palù, A. Ferrari, P. Mignosa, F. Aureli, S. Dazzi (2017). A non-uniform efficient grid type for GPU-parrallel Shallow Water Equations models. Adv. Water. Resour. 88, 119-137

[7] R. Vacondio, A. Dal Palù, P. Mignosa, (2014). GPU-Enhanced Finite Volume Shallow Water solver for fast flood simulations. Environ. Model. Softw. 57, 60-75.

[8] M. Altinakar A.R. Brodtkorb, M.L. Sæ tra. Efficient shallow water simulations on gpus: Implementation, visualization, verification and validation comput. fluids 55, 1-12, 2012.