FUNWAVE-GPU: Multiple-GPU Acceleration of a Boussinesq-Type Wave Model

Ye Yuan1,2, Fengyan Shi1, James T. Kirby1, and Fujiang Yu2

1Center for Applied Coastal Research, Department of Civil and Environmental Engineering, University of Delaware, Newark, DE, USA, 2National Marine Environmental Forecasting Center of China, Beijing, China

Abstract This paper documents development of a multiple-Graphics Processing Unit (GPU) version of FUNWAVE-Total Variation Diminishing (TVD), an open-source model for solving the fully nonlinear Boussinesq wave equations using a high-order TVD solver. The numerical schemes of FUNWAVE-TVD, including Cartesian and spherical coordinates, are rewritten using CUDA Fortran, with inter-GPU communication facilitated by the Message Passing Interface. Since FUNWAVE-TVD involves the discretization of high-order dispersive derivatives, the on-chip shared memory is utilized to reduce global memory access. To further optimize performance, the batched tridiagonal solver is scheduled simultaneously in multiple-GPU streams, which can reduce the GPU execution time by 20–30%. The GPU version is validated through a benchmark test for wave runup on a complex shoreline geometry, as well as a basin-scale tsunami simulation of the 2011 Tohoku-oki event. Efficiency evaluation shows that, in comparison with the CPU version running at a 36-core HPC node, speedup ratios of 4–7 and above 10 can be observed for single- and double-GPU runs, respectively. The performance metrics of multiple-GPU implementation needs to be further evaluated when appropriate.

Plain Language Summary Numerical modeling of surface wave dynamics is necessary for coastal infrastructure design. FUNWAVE-Total Variation Diminishing is a widely accepted open-source wave model for simulating surface wave propagation and wave-driven processes in the nearshore region, as well as tsunami wave propagation at oceanic scales. Due to the complexity of governing equations and corresponding numerical methods, the modeling of wave dynamics usually depends on the use of High Performance Clusters, which are both expensive and power consuming. To address this problem, Graphics Processing Unit (GPU)-accelerated computing is introduced in the FUNWAVE-Total Variation Diminishing for wave dynamics modeling in this study. GPUs were originally used for image processing and visualization purpose in personal computers. Because GPUs have thousands of “Cores” that can implement arithmetic computations simultaneously, they are now widely employed to facilitate computing-intensive tasks such as deep learning and engineering computations. We find that by porting wave model to GPU devices, the modeling of surface wave dynamics over a large domain can be achieved by an affordable stand-alone PC with GPU cards installed.

1. Introduction Phase-resolving Boussinesq-type wave models have proven to be robust tools for modeling surface waves and wave-driven processes in the nearshore region as well as tsunami wave propagation at oceanic scales (Kirby, 2016; Shi et al., 2012). Recently, the Boussinesq-type model FUNWAVE-Total Variation Diminishing (TVD) (Shi et al., 2012) has been extended to simulate sediment transport dynamics (Tehranirad et al., 2015), ship wakes (Shi et al., 2018), meteotsunamis (Woodruff et al., 2018), and landslide tsunamis (Li et al., 2019). Due to the higher-order numerical schemes needed to treat the dispersive terms and the requirement to solve tridiagonal matrix systems, FUNWAVE-TVD is much more computationally demanding compared with shallow water equation solvers (Kirby, 2016). Although the current version of the FUNWAVE-TVD code, parallelized using the Message Passing Interface (MPI), is able to handle a large-scale computation such as wind wave simulations in a ∼ 10 km × 10 km region, a large amount of CPU cores in High-Performance-Computing (HPC) clusters may be required to fulfill the job in reasonable time frames. To address this problem, we have developed a multi-GPU (Graphics Processing Units) accelerated version of FUNWAVE-TVD in this study.
As an efficient, portable yet commercially available substitute for HPC clusters, GPUs have played a central role in implementing massive computation across a wide range of areas, among which GPU acceleration of CFD algorithms was one of the main areas in the past few years. The key to the success of GPU computing is partly attributed to its capability of massive computation characterized by high throughput and low latency when compared to CPUs (Brodtkorb et al., 2013). In particular, this capability can be achieved by lower power consumption and less hardware investment. Furthermore, the release of general-purpose programming languages (e.g., NVIDIA CUDA C, PGI CUDA Fortran, and OpenACC) enormously lowers the barrier to GPU programming for nongraphic applications.

A number of studies on GPU implementation of the shallow water equation solvers have been published in the past decade. In particular, the porting of tsunami models to GPU devices has been a priority of the international tsunami research community in the wake of the devastating 2011 Tohoku Tsunami. Currently, the majority of mainstream tsunami models, mostly based on linear or nonlinear shallow water equations discretized with different orders of accuracy (e.g., Tsunami-HySEA, MOST, GeoClaw, TUNAMI-N1, EASYWAVE, and COMCOT), have been accelerated by GPU devices for real-time tsunami warning purpose at a lower cost (Amouzgar et al., 2014; Castro et al., 2015; de La Asunción et al., 2016; Gidra et al., 2011; Harig et al., 2019; Qin et al., 2019; Satria et al., 2012). Typically, the computational time for basin-scale tsunami propagation modeling can be reduced to several minutes, or even tens of seconds in some simplified cases. By incorporating Tree-based mesh-refinement (Acuña & Aoki, 2018), Adaptive Mesh Refinement techniques (de la Asunción & Castro, 2017; Qin et al., 2019; Vacondio et al., 2017), the efficiency of a tsunami simulation can be further optimized without losing accuracy in the nearshore regions.

The GPU implementation of Boussinesq-type wave models has not been reported until recently. Tavakkol and Lynett (2017) presented an interactive coastal wave simulation and visualization software “Celeris,” which was packed with a Single-GPU-accelerated solver to solve the extended Boussinesq equations using a hybrid finite volume-finite difference scheme. The software, programmed with C++ and Microsoft’s Direct3D library, can be freely accessible with a simple sign-up procedure. Kim et al. (2018) successfully ported their Boussinesq wave model to single GPU via CUDA Fortran. Studies on GPU acceleration of 2-D and 3-D nonhydrostatic model has also been conducted recently. Escalante et al. (2018) revisited the 2-D nonhydrostatic shallow water model proposed by Yamazaki et al. (2011), and developed a GPU-accelerated FVM-FDM algorithm to simulate dispersive wave dynamics using the CUDA C language. A nonhydrostatic 3-D ocean model “kinaco” was implemented on a single GPU with an improved Poisson/Helmholtz solver (Yamagishi & Matsumura, 2016).

In this paper, we describe a porting of FUNWAVE-TVD to single- and multi-GPU systems in order to improve model performance. The paper is organized as follows. Section 2 briefly reviews the governing equations and numerical schemes adopted by FUNWAVE-TVD. Then, the CUDA Fortran implementation of the package and optimization are detailed in section 3. A brief introduction to GPU architecture and features, which are closely related to the current work, is also addressed in this section. To validate the model, section 4 presents three test cases and the performance analysis. Conclusions are made in section 5. The source code is open-source and available at Github for test purpose (via https://github.com/dryuanye/FUNWAVE-GPU).

### 2. Governing Equations and Numerical Scheme for FUNWAVE-TVD
#### 2.1. Governing Equations

FUNWAVE-TVD is a fully nonlinear Boussinesq-type wave model discretized by a hybrid method combining finite-volume and finite-difference TVD-type schemes (Shi et al., 2012). To facilitate the subsequent introduction on GPU implementation, it is necessary to briefly describe the governing equations in this section. The conservative form of the fully nonlinear Boussinesq equations can be written as

\[
\begin{align*}
\eta_t + \nabla \cdot \mathbf{M} &= 0 \\
\mathbf{M}_t + \nabla \left( \frac{\mathbf{MM}}{H} \right) + gh \nabla \eta &= H \left\{ \tilde{u}_2 + u_a \cdot \nabla \tilde{u} + \tilde{u}_2 \cdot \nabla u_a - V_1 - V_2 - V_3 - R \right\}
\end{align*}
\]

where

\[
\mathbf{M} = H \left\{ u_a + \tilde{u}_2 \right\}
\]
In equation (1), both $\Psi$ and $\Theta$ are the vectors of conserved variables and the flux vector function, respectively, and are given by

$$
\Psi = \left( \begin{array}{c}
\eta \\
U \\
V 
\end{array} \right), \quad \Theta = \left( \begin{array}{c}
\frac{P_i + O_q}{\Pi} \\
\frac{P_i}{\Pi} \mathbf{1} + \left[ \frac{\zeta}{\Pi} + \frac{1}{2} g (\eta'^2 + 2 \eta h) \right] \mathbf{1} + \frac{\zeta}{\Pi} \mathbf{J} \
g_{\eta'} \mathbf{J} + \psi_x + H R_x
\end{array} \right).
$$

$$
S = \left( \begin{array}{c}
0 \\
g_{\eta'} \frac{\partial \ln \psi}{\partial t} + \psi_x + H R_x \\
g_{\eta'} \frac{\partial \psi_y}{\partial t} + \psi_y + H R_y
\end{array} \right).
$$

is the horizontal volume flux. $H = h + \eta$ is the total local water depth, in which $h$ is water depth, and $\eta$ is free surface displacement. $u_\alpha$ is the horizontal velocity vector at a reference elevation $z = z_\alpha$.

$$
\overline{u}_\alpha = \left( \frac{z_\alpha^2}{2} - \frac{1}{6} (h^2 - h \eta + \eta^2) \right) \nabla (\nabla \cdot u_\alpha) + \left( z_\alpha + \frac{1}{2} (h - \eta) \right) \nabla (\nabla \cdot h u_\alpha)
$$

is the depth-averaged $O(\mu^2)$ order of correction to horizontal velocity, in which $\mu$ is the dimensionless parameter for wave dispersion.

In equation (1), both $\mathbf{V}_1$ and $\mathbf{V}_2$ are dispersive Boussinesq terms, and $\mathbf{V}_1$ accounts for the second-order effect of the vertical vorticity. $\mathbf{R}$ denotes diffusive and dissipative terms including bottom friction, subgrid lateral turbulent mixing, as well as wave breaking if the eddy viscosity breaking scheme of Kennedy et al. (2000) is applied. The details of expressions for these terms can be found in Chen et al. (2003) and Shi et al. (2012). It is worth to mention that time derivatives in $\mathbf{V}_1$ can be extracted as $\mathbf{V}'_{1,t}$, and $\mathbf{V}_1$ is expressed as

$$
\mathbf{V}_1 = \mathbf{V}'_{1,t} + \mathbf{V}''_1
$$

As explained in Shi et al. (2012), $\mathbf{V}'_{1,t}$, together with the time derivative dispersive term $\mathbf{V}_2$, is merged in the left hand side of equation (1), which gives

$$
\begin{align*}
\mathbf{V}_1 + \nabla \cdot &\left( \frac{\mathbf{MM}}{H} \right) + \nabla \left[ \frac{1}{2} g (\eta'^2 + 2 \eta h) \right] \\
&= \eta (\mathbf{V}'_1 - \overline{u}_\alpha) + H (u_\alpha - \overline{u}_\alpha + \overline{u}_\alpha \cdot \nabla u_\alpha - \mathbf{V}'_1 - \mathbf{V}_2 - \mathbf{V}_3 - \mathbf{R}) + g \eta \nabla h
\end{align*}
$$

where

$$
\mathbf{V} = H (u_\alpha + \mathbf{V}_1)
$$

As $\mathbf{V}$ is obtained, the velocity $u_\alpha$ can be found by solving tridiagonal systems formed by equation (5).

### 2.2. Numerical Scheme

In FUNWAVE-TVD, the naming convention of variables and numerical scheme basically follow Shi et al. (2012). We present the scheme here in a compact manner, with some mathematical expressions omitted. The vector forms of variables in equation (5) are

$$
\begin{align*}
&\mathbf{u}_\alpha = (u, v), \\
&\overline{u}_\alpha = (U_\alpha, V_\alpha), \\
&M = (P, Q) = H \left[ u + U_\alpha, v + V_\alpha \right], \\
&\mathbf{V}'_1 = (U'_1, V'_1), \\
&\mathbf{V}''_1 = (U''_1, V''_1), \\
&\mathbf{V}_2 = (U_2, V_2), \\
&\mathbf{V} = (\overline{U}, \overline{V}) = H \left[ (u + U'_1), (v + V'_1) \right], \\
&R = (R_x, R_y).
\end{align*}
$$

The generalized conservative form of Boussinesq equations can be written as

$$
\frac{\partial \Psi}{\partial t} + \nabla \cdot \Theta(\Psi) = \mathbf{S}
$$

where $\Psi$ and $\Theta(\Psi)$ are the vectors of conserved variables and the flux vector function, respectively, and are given by

$$
\begin{align*}
\Psi &= \left( \begin{array}{c}
\eta \\
U \\
V 
\end{array} \right), \quad \Theta &= \left( \begin{array}{c}
P_i + O_q \\
\frac{P_i}{\Pi} \mathbf{1} + \left[ \frac{\zeta}{\Pi} + \frac{1}{2} g (\eta'^2 + 2 \eta h) \right] \mathbf{1} + \frac{\zeta}{\Pi} \mathbf{J} \
g_{\eta'} \mathbf{J} + \psi_x + H R_x
\end{array} \right).
\end{align*}
$$
Perfect linear scalability is not achieved for three main reasons. First, for a larger modeling domain that is subdivided into a number of subdomains, time for the exchange of the overlapping ghost cells among MPI ranks is not negligible. In FUNWAVE-TVD, in addition to free surface and momentum fluxes, a number of high-order derivatives need to be transferred across the inner overlapping ghost cells. This is also the case for outer boundaries when the periodic boundary condition is switched on. Second, although binary output format has been enabled in the FUNWAVE-TVD, the more popular and self-contained HDF5 source terms in the right hand side can be discretized by finite difference.

The expanded forms of \(U_i^t, V_j^t, U_i^{n'}, \psi_x, \psi_y, \eta_x, \eta_y\) are discretized by central difference. As mentioned before, term \(R\) breaks into three parts, including bottom stress, subgrid mixing, and wave breaking. Bottom stress can be approximated by quadratic friction law, and subgrid mixing (if included) is handled by the addition of lateral eddy viscosity estimated in terms of wave-induced mean current field. Wave breaking is modeled by two schemes implemented in the CPU code; either the shock-capturing scheme of Tonelli and Petti (2009), or the eddy viscosity scheme following Kennedy et al. (2000).

For the flux terms, FUNWAVE-TVD introduced a shock-capturing MUSCL-TVD scheme with Minmod limiter for local Riemann approximation at the cell interface. In the latest version, FUNWAVE-TVD v3.3, the default scheme is updated to the modified fourth-order MUSCL-TVD with van-Leer limiter for better stability (Choi et al., 2018).

To ensure temporal accuracy, a third-order Strong Stability-Preserving Runge-Kutta scheme is adopted for time stepping (Shi et al., 2012). As \(\Psi\) is obtained at each intermediate step, the velocity \(u_x\) can be solved by a system of tridiagonal matrix equations formed by equation (5). The Thomas algorithm is then adopted to solve the systems. As discussed in section 3, an efficient tridiagonal solver is the key to the success of GPU acceleration.

In order to evaluate the MPI efficiency, numerical tests were carried out in the HPC cluster at National Marine Environmental Forecasting Center of China by modeling a wave-averaged mean current field in a 2,048 × 2,048 grid with time duration of 1,000 s (Case 2 in section 4). Each HPC node has 28 cores running at a clock frequency of 2.7 GHz. The task was assigned with 4, 8, 16, 32, 64, and 128 cores separately to test MPI speedup. As shown in Figure 1, the drop in scalability becomes significant starting from the 32-core test. For 128-core case, the speedup ratio is 16 compared to the 4-core test case, which is one half of the linear scaling.

**Figure 1.** Scalability of MPI implementation in FUNWAVE-TVD for a 2,048 × 2,048 domain. Diagonal line denotes linear scalability, and actual speedup is shown by cycles.

The expanded forms of \(U_i^t, V_j^t, U_i^{n'}, \psi_x, \psi_y, \eta_x, \eta_y\) can be found in Appendix A of Shi et al. (2012). Based on equation (6), the source terms in the right hand side can be discretized by finite difference scheme. The high-order dispersive terms in \(\psi_x\) and \(\psi_y\) are discretized by central difference. As mentioned before, term \(R\) breaks into three parts, including bottom stress, subgrid mixing, and wave breaking. Bottom stress can be approximated by quadratic friction law, and subgrid mixing (if included) is handled by the addition of lateral eddy viscosity estimated in terms of wave-induced mean current field. Wave breaking is modeled by two schemes implemented in the CPU code; either the shock-capturing scheme of Tonelli and Petti (2009), or the eddy viscosity scheme following Kennedy et al. (2000).

For the flux terms, FUNWAVE-TVD introduced a shock-capturing MUSCL-TVD scheme with Minmod limiter for local Riemann approximation at the cell interface. In the latest version, FUNWAVE-TVD v3.3, the default scheme is updated to the modified fourth-order MUSCL-TVD with van-Leer limiter for better stability (Choi et al., 2018).

To ensure temporal accuracy, a third-order Strong Stability-Preserving Runge-Kutta scheme is adopted for time stepping (Shi et al., 2012). As \(\Psi\) is obtained at each intermediate step, the velocity \(u_x\) can be solved by a system of tridiagonal matrix equations formed by equation (5). The Thomas algorithm is then adopted to solve the systems. As discussed in section 3, an efficient tridiagonal solver is the key to the success of GPU acceleration.

In order to evaluate the MPI efficiency, numerical tests were carried out in the HPC cluster at National Marine Environmental Forecasting Center of China by modeling a wave-averaged mean current field in a 2,048 × 2,048 grid with time duration of 1,000 s (Case 2 in section 4). Each HPC node has 28 cores running at a clock frequency of 2.7 GHz. The task was assigned with 4, 8, 16, 32, 64, and 128 cores separately to test MPI speedup. As shown in Figure 1, the drop in scalability becomes significant starting from the 32-core test. For 128-core case, the speedup ratio is 16 compared to the 4-core test case, which is one half of the linear scaling.

Perfect linear scalability is not achieved for three main reasons. First, for a larger modeling domain that is subdivided into a number of subdomains, time for the exchange of the overlapping ghost cells among MPI ranks is not negligible. In FUNWAVE-TVD, in addition to free surface and momentum fluxes, a number of high-order derivatives need to be transferred across the inner overlapping ghost cells. This is also the case for outer boundaries when the periodic boundary condition is switched on. Second, although binary output format has been enabled in the FUNWAVE-TVD, the more popular and self-contained HDF5...
or NETCDF formats are not implemented yet. Therefore the output can be time-consuming when using formatted output.

Another important reason is that the tridiagonal solver cannot be fully parallelized with the equal workload. This can be explained by a left-right subdomain configuration. The two-stage Thomas algorithm (forward sweeping and backward substitution) is adopted to solve the tridiagonal systems. The computation of the right subdomain could not start until the forward sweeping of the left side is finished and the \((i-1)\) value is available for MPI transfer. Similarly, the left subdomain then has to wait until the backward substitution of the right side is completed and the \(i\) value for substitution is ready. As stated later in section 3.5.3, percentage of time that is taken by tridiagonal solver is around 22%. Thus, the observed performance decline resulting from the serial nature of the Thomas algorithm is nontrivial.

### 3. GPU Implementation

Prior to GPU programming, it is necessary to ascertain that part of the existing code is the most computationally demanding. By profiling FUNWAVE-TVD using MPI timing function, we found that computations of high-order dispersive terms, the Riemann solver for numerical fluxes, and the tridiagonal solver take more than 90% of the total running time. Although the first two steps are explicit and highly parallelizable, the Thomas algorithm for the tridiagonal system is basically serial. In this paper we explored two different solvers based on previous studies (Chang & Hwu, 2014; Kim et al., 2011). We also expect the GPU speedup would be affected by branching since there are lots of logical conditional structures in FUNWAVE-TVD.

Choice of an appropriate programming tool is important. Before the advent of PGI CUDA Fortran, the majority of GPU programming adopt CUDA C as the coding language. However, for ocean models that are usually developed by Fortran, it is costly not only for the developers to rewrite all the code but also for existing users to reacquaint themselves with the model. We notice that a number of studies on GPU-accelerated hydrodynamic models have been reported which are based on CUDA Fortran (de La Asunción et al., 2016; Kim et al., 2018; Qin et al., 2019; Yamagishi & Matsumura, 2016; Zhang & Jia, 2013; Zhu et al., 2018). We also adopt CUDA Fortran in this study for the aforementioned reasons.

For multiple-GPU programming, the cuda-aware OpenMPI is used to take care of inter-GPU communication. The direct inter-GPU MPI transfer is introduced in Kepler-class GPUs and CUDA 5.0, which can largely minimize the coding effort and improve data transfer efficiency. However, the technology is highly dependent on the GPU hardware and network interfaces. Considering the variety of normal user's computational environment, we use standard two-step copies via CPU memory for code portability.

In this section, the guidelines and strategies for the GPU implementation of FUNWAVE-TVD are given after a brief introduction on GPU notations and features related to the current study.

#### 3.1. GPU Architecture, Memory Hierarchy, and Streams

##### 3.1.1. GPU Architecture

Unlike CPU, GPU is composed of thousands of CUDA cores that can run simultaneously. CUDA core is a fully pipelined integer and floating-point arithmetic logic unit that executes one integer or floating-point operation per clock cycle. While CPU architectures are designed for task parallelism, GPU architectures are designed for data parallelism. GPU employs an execution model called Single Instruction, Multiple Thread. A series of instructions for specific purpose can form a user-defined function which is called CUDA kernel. When launched, it first generates a grid consisting of blocks of threads and then make these threads flow through all the CUDA cores in an organized and synchronized manner. Each thread can be regarded as a clone of kernels associated with a particular block and thread ID. CUDA has a series of built-in variables and structures to define the number of blocks in a grid in each dimension and the number of threads in a block in each dimension. In each CUDA core, there are a batch of active threads executing in serial. When finished, they exit and another batch of threads flows in with zero overhead. This feature is perfect for massive parallelism in computation.

CUDA cores are managed by streaming multiprocessors (SMs) in GPU. The SM typically contains a group of CUDA cores, caches, warp scheduler, and registers, which makes them be compared with CPU cores to some extent. The latest NVIDIA Tesla V100 GPU, which is used in our study, contains 80 SMs, each with 64 CUDA cores and a 128 KB on-chip memory. The SM groups adjacent 32 threads within a block into warps. These threads read in the consecutive memories and execute the same floating-point instructions. Thus, memory
accesses in chunks of 32, 64, and 128 bytes are necessary. For NVIDIA V100, each SM includes four warp scheduler, and each scheduler can issue a set of arithmetic instruction to a warp every two clock cycles. The maximum number of concurrent warps per SM is 64, which suggests there are 2,048 active threads in a SM every two cycle. Different warps are executed independently by scheduler unless there is explicit synchronization. The memory latency can thus be effectively hidden by high floating-point throughput. In our case, as illustrated by Figure 2, it is natural to map the grid of threads to the modeling domain with each thread corresponding to a node or cell. The stencil problem can usually be written as a CUDA kernel. When called, in each thread the value at a cell centroid or interface can be estimated by values of the neighboring cell.

3.1.2. Memory Hierarchy

It is necessary to manipulate a combination of memory types to avoid memory latency in GPU programming. Here we briefly introduce three types of memories that are used in the present study, which is listed in decreasing order by speed: registers, shared memory, as well as global memory.

In a CUDA kernel, locally declared variables are stored in registers. Although each SM contains tens of thousands of registers, there is a trade-off between the amounts of active threads and register usage. For thousands of active threads in a SM, registers are actually a limited resource to each thread.

The SM has a fixed amount of fast-retrieving shared memory shared by all residing threads within a block. Arrays declared explicitly with attribute `shared` are stored in shared memory. Shared memory is private for each block, and visible to each thread within the same block. This feature allows thread collaboration when neighboring threads are interdependent. To minimize the global memory transaction and reuse the data, the best practice is allocating shared memory for unknown variables in a stencil problem. Since it is a per-SM resource, shared memory usage can affect occupancy, which is defined as the number of warps that the SM can keep resident. For complicated stencil problems involving multiple variables, careful design of kernel function is necessary to maintain a high occupancy.

The main GPU memory is refer to as the global memory. All the field variables need to be declared in global memory. Even though NVIDIA V100 has a theoretical memory bandwidth of 900 GB/s, there is still huge latency for memory transactions. As a rule of thumb, maximum memory performance should be pursued through coalescing of memory transactions, which means every successive 128-bytes (32 single precision words) memory is accessed by a warp in a single transaction.

3.1.3. Streams and Concurrent Kernels

Due to separate engines to schedule device-host memory transfers and CUDA kernel, multiple CUDA operations can be performed simultaneously on GPU. The concurrency can be accomplished by issuing a sequence
of operations in streams. Memory transfer can be effectively hidden by overlapping kernel execution in the same stream, while CUDA kernels in different streams may run concurrently if there still are SM resources available. Kernel launch are asynchronous, which means the task dispatcher returns immediately without waiting it to be completed. For NVIDIA V100, up to 32 CUDA kernels can be launched asynchronously on a GPU. For independent kernels that work on different data sets, concurrent kernels can improve GPU performance in some degree. This virtue will be further illustrated in section 3.5.

3.2. CUDA Fortran Implementation

Following section 2, the algorithm for solving the Boussinesq equations is numerically advanced as follows in the FUNWAVE-GPU version. For model initialization and host memory declaration, the original CPU code is intact since they are executed only once. The flowchart is illustrated in Figure 3.

1. Map MPI ranks to different GPU cards at different nodes. It is necessary to know the topology of GPUs in the current GPU cluster before running the model. This can be accomplished by running nvidia-smi or cudaDeviceQuery beforehand.
2. Allocate device memory and upload variables. Here, host-to-device memory transfer is referred to as upload. For a large modeling domain, this step is time-consuming since FUNWAVE involves a wide range of field variables due to complexity of Boussinesq equations.
3. Update free surface elevation and flow velocity components for the previous time step.
4. Estimate adaptive time step $\delta t$ following the Courant-Friedrichs-Lewy criterion. The minimal value must be collected from each GPU by MPI reduction, to ensure the synchronization among different subdomains. The corresponding subroutine is Estimate_dt_gpu.
5. Calculate high-order dispersive terms discretized by finite difference method at cell centroids, including \((U'_1, V'_1), (U'_2, V'_2), (U'_3, V'_3), (U'_4, V'_4), (U'_5, V'_5)\). These high-order terms, together with pressure gradients and dissipating terms, are considered as source terms \(S\) on the right hand side of the generalized conservative form of Boussinesq equations. The corresponding subroutine is \texttt{Cal\_Dispersion}.

6. Construct horizontal fluxes \(\Theta(\Psi)\) at cell interfaces by the modified fourth-order MUSCL-TVD with van-Leer limiter, then use the HLL approximate Riemann solver to estimate numerical fluxes across the cell interfaces. This two-step computation is completed in subroutine \texttt{Fluxes}. Unlike the previous step, CUDA threads are mapped to all cell interfaces, rather than cell centroids.

7. Compute source terms, including hydrostatic pressure gradient contributed by bathymetry, bottom stress estimation using the quadratic friction law, high-order dispersive terms yielded by Step 5. The source term is also the place to incorporate Smagorinsky-like subgrid turbulent mixing algorithm, apply wavemakers or wind forces. The corresponding subroutine is \texttt{SourceTerms}.

8. Derive \(\eta^{n+1}, (\psi^{n+1}, \Psi^{n+1})\) by a third-order Runge-Kutta time stepping scheme.

9. Compute the velocity field by solving tridiagonal systems along the \(x\) and \(y\) directions using the Thomas algorithm. For the multiple-GPU version, exchange of velocity components along the boundaries of neighboring subdomains is necessary since the Thomas algorithm is actually sequentially advanced along the \(x\) and \(y\) directions.

10. Implement wave breaking algorithm and exchange variables along the inner and outer boundaries among the neighboring sub-domains by MPI.

11. Make statistics on field variables such as \(\eta\) and velocity components, and check model blowup conditions. Steps 1–2 are only called once, while Steps 3–11 are advanced repeatedly until time integration finished. In each time step, Steps 4–10 are iterated triple times.

### 3.3. Extension to Multiple GPUs by MPI

Generally, there are several strategies of using multiple GPUs for computation. For a single computational node hosting multiple GPUs via its PCI express (PCIe) slots, it is recommended that using Peer-to-Peer communication to transfer data among GPUs. For short, NVIDIA GPU’s Peer-to-Peer feature allows CUDA programs to access and transfer data from one GPU’s memory to another without having to go through CPU’s host memory. This can be realized by designating different GPU devices (\texttt{cuda\_set\_device(n)}) and then asynchronously launching a bundles of kernels and memory copies in an alternating way. However, this way of programming is inflexible and only confined to single-node multiple-GPU configuration. For multiple-node multiple-GPU programming, using CUDA-aware MPI for multiple-GPU communication is necessary. In analogy with MPI+CPU applications, in this case each GPU device is regarded as a MPI rank, and assigned by a subdomain. For CUDA-aware OpenMPI version 3.1.3 and above, the programming on assigning MPI ranks to GPUs on different nodes is largely simplified by introducing \texttt{MPI\_COMM\_SPLIT\_TYPE}.

Memory transfers are necessary for those overlapping ghost layers among GPUs. We adopt a two-stage copies to transfer data among GPUs via CPU memory. As device-host memory copies are inevitable, the best practice is to declare the transferred variables as page-locked memory for faster transfer speed or asynchronous operations. In combination with MPI nonblocking ISEND/IRECV, it is expected that time for inter-GPU MPI transfer could reduce. However, it is actually dependent on the size of transferred variables. Experiments show that the performance advantage of using pinned memory only shows up when the amount of memory transfer per call is substantially large (tens of megabytes). Besides, pinned memory is much more expensive to allocate and deallocate due to overhead on memory management and transfer. Since the amount of data exchange for the ghost cells of each GPU per call is small, the standard combination of \texttt{cuda\_Memcpy2D} and MPI nonblocking transfer is satisfactory.

### 3.4. Source Code and Compilation

Instead of developing a new package solely for GPU version, we pack the rewritten CUDA Fortran code with the existing source code in the same folder (/src). The GPU source files are marked with “gpu” as suffix, thus can be distinguished from their CPU counterparts easily. By a minor modification of Makefile for compilation, either CPU- or GPU-based executable can be generated as required. When looking into the subroutines in the source code, it is fairly straightforward to switch over the original code and the corresponding CUDA Fortran code by locating their names.

The two versions of source code share the same input file setup and generate the same output files. Generally, the original architecture of FUNWAVE-TVD has been retained as much as possible. This seamless
Table 1

| Hardware Configurations for the CPU and GPU Implementation of FUNWAVE-TVD in the Current Study |
|-------------------------------------------------|-------------------------------------------------|-------------------------------------------------|-------------------------------------------------|
| Hardware specifications                         | Core per CPU / Clock Frequency / CPU/GPU Performance | Hardware specifications                         | Core per CPU / Clock Frequency / CPU/GPU Performance |
|                                                  | (Cores per GPU / GHz) / Number (TeraFlops) / Compiler |                                                  | (Cores per GPU / GHz) / Number (TeraFlops) / Compiler |
| CPU                                              | 18 / 2.7 / 2 / 3.11 / PGI Fortran                  | GPU                                              | 5120 / 1.38 / 2 / 28.0 / PGI Fortran               |

merging is beneficial for the normal users to switch between two versions and save time on preprocessing and postprocessing. In this paper we refer to the new version as FUNWAVE-GPU hereafter.

The recommended software and hardware requirements for running the FUNWAVE-GPU includes PGI Fortran Compiler 18.4, CUDA toolbox Toolkit 9.1, CUDA-aware OpenMPI 3.3, and NVIDIA® graphic card with Compute Capability above 6.0. The source code has not been compiled extensively at different software and hardware configurations.

In this study, the FUNWAVE-GPU version has been compiled and then tested on the NVIDIA Tesla V100 using CUDA Version 10.1. The NVIDIA V100 model consists of 80 streaming multiprocessors each having 64 CUDA cores clocked at 1.38 GHz. The GPU has 32 GB of device memory with a memory bandwidth of 900 GB/s. Two GPU devices are hosted by a HPC node with 2 Intel Xeon Gold 6150 Processors with 36 CPU cores running at 2.7 GHz in total. As suggested in Table 1, the peak performance of single-precision processing for CPU and GPU in the current HPC server is approximately 3.11 and 28.0 TeraFlops (floating-point operations per second), respectively. For single- and double-GPU applications, the nominal speedup in terms of FP processing can reach 4.5 (14.0/3.11) and 9.0 (28.0/3.11), respectively. In practical cases, whether the peak performance can be achieved both for CPU and GPU highly depends on optimization of parallel algorithm.

3.5. Optimization and Runtime Analysis

In this section, fundamental rules on CUDA Fortran programming of FUNWAVE-GPU is presented at first, followed by the introduction of our efforts to optimize tridiagonal solver. Performance improvement is then illustrated by comparison between timelines of both optimized and nonoptimized CUDA kernels.

3.5.1. General Optimization

To fully utilize GPU resource, some rules of thumb for GPU programming are followed in the present study.

1. Taking full advantage of on-chip memories.
   In FUNWAVE-TVD, computation on both source terms and numerical fluxes can be classified as stencil problem. For a five-point stencil problem, the unknowns in the current cell is updated based on the values of its 4 neighboring cells. To complete the computation on a \( m \times n \) grid, the CUDA kernel has to access the global memory for \( 5 \times m \times n \) times or more. As most of stencils are memory-bound kernels, the strategy to approach GPU’s peak performance is by dividing grid into “tiles” through shared memory in each block. To use the resource, each tile is explicitly declared as shared with the same dimensions as the residing block. At first, every thread stores its point value to the tile by reading the global memory in a coalesced manner. The tile is visible to all threads in this block. Then, the stencils can be implemented without access to global memory after that. For five-point stencils, each inner point value is reused 4 times, transfers to and from global memory can thus be reduced.
   A straightforward CUDA implementation of the stencils is illustrated in Figure 2 (left). The modeling domain is mapped into 2D blocks with dimensions of \( 8 \times 8 \) (squares with colors). Assuming the value at one node is updated using the adjacent four nodes, the ith tile consists of \( 6 \times 6 \) inner nodes and halo cells with depth of 1 along its four boundaries. For each tile, while the local index along the \( x \) direction \( (x_{local}) \) ranges from 1–8, its global index increases by \( x_{local} + (i - 1) \times (BlockDim - 2) \). Here \( (BlockDim - 2) \) suggests the overlapped halo cells. In each time step, only values of inner nodes (local index from 2 to 7) are updated using the neighboring values by accessing shared memory. For the whole modeling domain, the grid dimensions (here GPU terminology “grid” refers to as the number of blocks along two directions) should be \( \lfloor m/(BlockDim - 2) \rfloor + 1 \) and \( \lfloor n/(BlockDim - 2) \rfloor + 1 \).

2. Keeping a high occupancy by designating appropriate grid/block dimensions, aligning memory access pattern and avoiding conditional branching.
Since each SM can execute at most eight blocks of threads simultaneously, choosing too small blocks prevents a high occupancy, while too large blocks may cause register spilling and oversized share memory. The number of threads per block should also be a multiple of 32; thus, global memory access can be in full warp. Generally, we set the block dimensions as 16 in most of CUDA kernels and CUF kernels. For ghost cells fillings and tridiagonal solvers, the block dimensions of 64 and 32 are used, respectively. This configuration is based on a comprehensive consideration on usage of on-chip memories and global memory access patterns, which helps to maintain a higher GPU occupancy and better performance. Besides, branching within a kernel is very expensive since the threads along the false branch is idle until next synchronizing point. We minimize the branching by moving the conditional operations out of the loop. It should be noted that only loop-irrelevant conditional statement can be safely treated in this way.

3. Using streams to launch concurrent kernels.

To maintain a high GPU occupancy, there must be sufficient active threads running simultaneously in each SM. Sometimes it is necessary to handle a bundle of smaller yet independent kernels. Instead of launching these kernels serially, we can issue them as concurrent kernels in different streams to fully saturate the GPU resource. In the FUNWAVE code, CUDA kernels for construction of numerical fluxes are called in batches for a series of conserved variables along the $x$ and $y$ directions. We dispatch them in several streams for concurrency. As shown in Figure 4 (pink patches), GPU profiling suggests that time for subroutine `fluxes_gpu` is decreased by approximately 15% for single- and double-GPU runs. Besides, streams are also used in the trivial kernels such as boundary and ghost-cell value fillings. In particular, the application of concurrent kernels in solving tridiagonal systems results in a substantial improvement on GPU performance, which will elaborated in the following section.

4. Minimizing device-host memory copies and MPI exchange.

To avoid expensive overhead on device-host data transfer via PCIe, the best practice is to keep all the data resident on the GPU as much as possible. Reducing model output can yield better speedup performance. The original FUNWAVE-TVD code requires MPI exchange of a dozen of high-order dispersive terms among GPUs. In GPU version we remove all inter-GPU exchange of dispersive terms along the ghost cells except for $U_4$. Instead, we calculate these ghost-cell values in the current time step based on the other known values. The reason we keep $U_4$ is that the depth of ghost cells (NGhost) need to be set as 4 in this case. To reduce time for data exchange to a minimal level, we further optimized the MPI programming. One of the ongoing improvement on current code package is to add parallel I/O by HDF5 or NETCDF4.

### 3.5.2. Optimization on Tridiagonal Solver

The Thomas algorithm has been used in FUNWAVE-TVD. The algorithm involves two-phase sequential computation including forward sweeping and backward substitution. Although the Thomas algorithm is inherently serial, it is widely accepted as standard solver for tridiagonal problem in hydrodynamic models due to its simplicity in numerical scheme and lower communication traffic for MPI parallelization. There are also a range of parallel algorithms (e.g., Cyclic Reduction (CR) and Parallel Cyclic Reduction (PCR)) for
Table 2

Performance of Thomas Algorithm and cuSPARSE’s cusparseGtsvStridedBatch API in Solving Tridiagonal Systems

| Grid resolution          |           |           |           |           |
|--------------------------|-----------|-----------|-----------|-----------|
|                          | 512 × 512 | 1,024 × 1,024 | 2,048 × 2,028 | 4,096 × 4,096 |
| Thomas algorithm         | 3.2 × 10⁻⁴ s | 7.8 × 10⁻⁴ s | 2.2 × 10⁻³ s | 7.2 × 10⁻³ s |
| cusparseGtsvStridedBatch | 4.0 × 10⁻⁴ s | 1.1 × 10⁻³ s | 2.1 × 10⁻³ s | 6.8 × 10⁻³ s |
| \( T_{Thomas} / T_{gtsv} \) | 0.80      | 0.71      | 1.04      | 1.05      |

aThomas algorithm is implemented by concurrent kernels.

solving tridiagonal systems on GPU (Chang & Hwu, 2014). NVIDIA’s cuSPARSE uses a combination of the CR and PCR algorithms to solve massive tridiagonal systems. However, it requires a significant amount of temporary extra storage, and does not support asynchronous execution.

Considering a 2-D case with mesh grid of \((m \times n)\), there are \(n\) tridiagonal systems of size \(m\) along the \(x\) direction, and \(m\) tridiagonal systems of size \(n\) along the \(y\) direction. For GPU implementation of the Thomas algorithm, as the solution of each system is found by a serial process, the common practice is mapping each tridiagonal system to a CUDA thread. In this case, the significant workload imbalance occurs among GPU’s SMs. For a small modeling domain (e.g., 512×512), only 512 CUDA cores are occupied, and each core needs to implement the Thomas algorithm serially over a 512-element tridiagonal system. On the other hand, for NVIDIA V100, there are more than 4,000 cores idle and waiting for thread assignment. GPU profiling reveals a fairly low occupancy of 32% for this test case.

As shown in Table 3, for FUNWAVE-GPU the most time-consuming aspect of the numerical schemes is the tridiagonal solver, which takes as much as 52.1% of computational time if not optimized due to its serial nature. Thus, the optimization of the current tridiagonal solving procedures is necessary. Two ways of optimization are explored. One strategy is to try different algorithm. CUDA Fortran provide implicit interface to cuSPARSE’s cusparseGtsvStridedBatch. Table 2 shows the performance comparison for solving tridiagonal systems with different sizes using the Thomas Algorithm and cuSPARSE tridiagonal solver on a NVIDIA V100. \( T_{Thomas} \) and \( T_{gtsv} \) are time required for the two algorithms per call. The values are averaged over 10,000 times of execution. The Thomas algorithm and cuSPARSE are comparable in terms of elapsed time per call. Considering code portability and computational accuracy, we keep the Thomas algorithm by default.

Another strategy is to re-organize code structure in FUNWAVE-GPU’s tridiagonal solver and try concurrent kernels. The current method solves a tridiagonal system by a thread, which can not fully occupy the GPU

Table 3

Per centages of Time\(^a\) Consumed by Major Subroutines for FUNWAVE MPI+CPU, Single GPU, and MPI+GPUs

| Subroutines/processes          | MPI+CPU (%) | Single GPU (%) | MPI+GPUs (%) |
|--------------------------------|-------------|----------------|--------------|
| Output                         | 0.4         | 1.1            | 0.4          |
| Variables update               | 0.8         | 2.7            | 3.5          |
| MPI exchange and periodic BC   | 2.2         | 0.9            | 13.4         |
| Estimate \( \delta t \)        | 0.5         | 0.8            | 0.6          |
| Dispersion terms               | 24.9        | 2.7            | 12.7         |
| Fluxes and Riemann solver      | 42.9        | 53.0           | 30.1         |
| Source terms                   | 3.2         | 9.2            | 6.3          |
| Tridiagonal solvers            | 22.3        | 28.3 (42.0\(^b\)) | 32.2 (52.1\(^b\)) |
| Wave breaking                  | 2.3         | 0.1            | 0.1          |
| Direct sponge                  | 0.3         | 0.0            | 0.0          |
| Check blowup                   | 0.1         | 1.2            | 0.6          |

\(^a\)Percentage values are different with those in Figure 4 due to different test cases (Rip-2D case with periodic boundary), as well as measuring methods. \(^b\)Nonoptimized tridiagonal solvers launch serially along \(x\) and \(y\) directions.
by a smaller number of systems. By solving the tridiagonal systems along the x and y directions by two concurrent kernels simultaneously launched through two streams, GPU occupancy improved considerably. As indicated by Figure 4 and Table 3, the execution of tridiagonal solver for two directions are completely overlapped (gray and light gray patches), resulting in a dramatic drop on executing time.

3.5.3. CUDA Kernels Runtime Analysis

Profiling is an effective approach to identify bottlenecks of major CUDA kernels. In Table 3, profiling on FUNWAVE’s CPU and GPU version is summarized in terms of percentage of time taken by each subroutines during the modeling period. FUNWAVE-TVD (denoted as MPI+CPU) is profiled by MPI_Wtime, while FUNWAVE-GPU (Single GPU and MPI+GPU) is profiled by nvprof. The profiling is based on CASE 2 (Rip-2D) with mesh grid of 1024 × 1024, which is presented in section 4.

For MPI+CPU version, the model is basically computation bound. Subroutines related to memory transfers, including Output, Variables Update, and MPI exchange, only take a modest share of time consumption. The computation-related kernels contribute 96.6% of running time. For the single-GPU version, the most prominent feature is the share of subroutine on Dispersion terms decreases sharply from 24.9% to 2.7%, suggesting advantage of GPU’s massive parallelism. Percentage of Tridiagonal solver changes in an opposite way, increasing from 22.3% to 42.0% for nonoptimized code. By adopting strategies introduced in Section 3.5.3, the percentage value drops to 28.3%. The variation suggests inherently-serial tridiagonal solver is part of the bottlenecks for GPU implementation. Statistics on multi-GPU version shows more interesting variations. Generally MPI exchange becomes the major limiting factor for speedup. Not only subroutines MPI exchange and Periodic BC exhibits a slowdown from 2.2% to 13.4%, but the shares of Dispersion terms and Tridiagonal Solver show a similar trend. The reason is that MPI exchange also exists in above two subroutines.

Improvement of FUNWAVE-GPU efficiency after the optimization can be clearly observed in Figure 4. The profiling is based on CASE 3 (Tohoku Tsunami) in section 4 with mesh grid of 3,200 × 2,400. For this case, it is worth noting that tridiagonal solver is implemented only once along the x and y direction separately. While in CASE 2, cyclic tridiagonal systems need to be solved along the y direction due to the periodic boundary condition at the South/North edges, which requires tridiagonal solver executed twice along y directions. Thus, the spherical mode of FUNWAVE is less computation-intensive than its Cartesian version.

We conducts four runs of CASE 3, including single-GPU with/without streams, and double-GPU with/without streams. While for single-GPU runs the profiling shows a favorable occupancy, in the double-GPU runs there is considerable idle period with percentage of time ranging between 8.0% and 19.0% in either card when solving the tridiagonal systems. The reason is detailed in section 2.3 at great length. Besides, application of streams for concurrent launching of tridiagonal solvers is obviously beneficial to the model efficiency. Since multi-GPU implementation is encumbered by MPI exchange and serial tridiagonal solver, the double-GPU run only saves 30–33% of time compared with single-GPU. However, for the modeling domain of 9, 600 × 7, 200, double-GPU run can exhibit a near-linear speedup over single-GPU (54%). In this case, because of heavy workload, each GPU turns to be computation-bound at most of time.

4. Numerical Validation and Performance Evaluation

As an evaluation of the model performance, we conduct three two-dimensional test cases with two purposes. The first is to validate the FUNWAVE-GPU using laboratory and field observations, and the second is to conduct performance evaluation of GPU implementation in comparison to its CPU counterpart. For the first purpose, we present two cases including solitary wave runup on a shelf with a conical island (Case 1), as well as Pacific-wide propagation of Tohoku Tsunami on 11 March 2011 (Case 3). Abundant of observations are available in these two cases, making them ideal to examine the FUNWAVE-GPU in Cartesian and Spherical coordinates. Second, in the case of rip current modeling at a 2-D beach (Case 2), we make an analysis on the FUNWAVE-GPU efficiency by running the case at four different grid resolutions. Besides, the Case 3 is also conducted at two different spatial resolutions (3,200 × 2,400 and 9,600 × 7,200 separately) to exhibit the speedup metrics. It should be noted that all the benchmark tests are simulated with single precision.

Generally, the efficiency of the GPU implementation of FUNWAVE-TVD over the CPU is evaluated by computing the speedup factor \[ S = \frac{T_{\text{CPU}}}{T_{\text{GPU}}}, \] where \( T_{\text{CPU}} \) and \( T_{\text{GPU}} \) are the time used by the CPU and GPU, respectively. Time for model initialization and variable allocation is excluded, and the beginning of the main loop is taken as the onset of time measurement. Nevertheless, it should be noted that how to showcase the performance of GPU acceleration over CPU is somewhat difficult. The reported speedup ratios
Figure 5. Model bathymetry contours (in meters) for Case 1. The wave-gauge locations are marked with triangles (G1–G9).

Figure 6. Time series comparison of $\eta$ among observations (black solid lines), FUNWAVE-GPU (black dashed lines), and FUNWAVE-TVD (red dash-dotted lines) at nine gauges.
mainly depend on the hardware, workload, and algorithm that are used for CPU versions and their GPU counterparts, respectively. For example, the GPU codes with improved algorithm may be compared with the inefficient CPU code; a recently-released GPU card may compete with a low-performance CPU node, and vice versa. Furthermore, a memory-bound task may not fully occupy the GPU device and thus underestimate the GPU’s peak performance. In this study both of GPU and CPU are the latest models, thus the comparison between them is regarded as reasonable in terms of hardware temporalness. For simplicity, we stick to the metrics of speedup ratios, and compare these values to the ratio of theoretical peak performance between GPU and CPU used.

4.1. Case 1: Solitary Wave Runup on a Shelf With an Island

To validate FUNWAVE-GPU, we performed a simulation of solitary wave runup measured in a large wave basin. The basin is 48.8 m long, 26.5 m wide, and 2.1 m deep. A complex bathymetry consisting of a 1:30...
Figure 9. Percentage of time that more than one kernel is executing in the device for the single-GPU (cycles) and double-GPU (triangles) computation.

4.2. Case 2: Rip Current Modeling on a 2-D Beach

FUNWAVE-TVD has been proven to be able to model surf zone flows such as alongshore and rip currents (Chen et al., 1999, 2003; Geiman et al., 2011; Shi et al., 2012). In this case, we applied an idealized rip-channel bathymetry, which has been used for a standard test of rip current generation in FUNWAVE-TVD. The purpose of the test is to examine the model efficiency for different model configurations of the GPU version. The periodic boundary condition is used at the north and south edges, and the absorbing boundary condition is applied at the offshore edge. An internal wavemaker for directional irregular wave generation was located far from the shoreline. This case accounts for the nonlinear and dispersive features of wave shoaling, as well as wave breaking at the surf zone. The heavy computational load makes the case ideal for efficiency evaluation. To test the speedup of FUNWAVE-GPU over the original code, we conduct four runs with different mesh grid sizes with a time duration of 1,000 s, including grid dimensions of 512 × 512, 1,024 × 1,024, 2,048 × 2,048, and 4,096 × 4,096.

Snapshots of surface elevation, vorticity and wave-induced current modeled using the 512 × 512 grid are shown in Figure 7. Waves approach the beach in a shore-normal direction with wave front transforming
gradually. Wave breaking-generated vortices are generally confined to the surf zone, and some small-scale vortices are evolved and then shed toward the offshore direction. The vector plots of wave-averaged current reveal offshore-directed rip current is formed and strongest along the rip channels.

Figure 8 illustrates performance of single- and double-GPU code with different mesh grid sizes. Overall, the larger the mesh grid size is, the better the speedup shows. GPU’s massive parallelism on computation begins to show its superiority when mesh grid larger than 1,024 × 1,024. For modeling tasks with smaller mesh domains of 512×512, due to insufficient occupancy of the CUDA cores, sequential algorithm for tridiagonal systems, and unavoidable overhead, GPU implementation can only achieve a minor speedup of 2.7. This is the case especially for the 2-GPU run, which only shows a speedup of 3.5. The performance exhibits minor improvement with the optimization on concurrent kernels (Figure 8b). The peak performance of GPU is not achieved as suggested by Figure 9. If we measure the GPU performance by a metric of percentage of time that more than one kernel is executing, only 76% and 65% of time that GPU is busy with computational kernels for single- and double-GPU cases, respectively.

For the single-GPU run with concurrent kernels, the metric of speedup ratio climbs up to 7.1 and stay stable above 6.6 as the mesh grid size larger than 1,024,024. For double-GPU run, a speedup of 14.2 for 4,096 × 4,096 suggests a near-linear scalability over the single-GPU run (7.8). However, this perfect scalability cannot be achieved for runs of smaller modeling domain. The reasons are presented in the previous section. When referring to Figure 9, we find the GPU is occupied 97% and 91% of the executing time for the single- and double-GPU runs. For double-GPU run, the global domain is partitioned as two smaller subdomains, the GPU occupancy lowers slightly as expected.

4.3. Case 3: Tohoku Tsunami Modeling Using Spherical Coordinates

In order to validate the CUDA implementation of the code in spherical coordinates, numerical simulation of $M_w = 9.0$ Tohoku tsunami on 11 March 2011 at 05:46 UTC is made with two spatial resolutions.
Figure 11. Time series comparison of tsunami wave height for observations (black dashed lines), FUNWAVE-GPU (red solid lines), and FUNWAVE-TVD (blue dash-dotted lines) at 14 near- and far-field DART buoys.
(3 and 1 arc-min, respectively). The tsunami source is based on the finite fault solution proposed by Fujii et al. (2011), obtained by inverting tsunami waveforms recorded at tide and wave gauges, GPS wave gauges, and deep water DART buoys (available at http://equake-rc.info/srcmod/). The initial surface condition is then determined by a summation of each subfault's vertical displacement by Okada's rectangular fault model. The model bathymetry from (130°E, −60°S) to (−70°W, 60°N) is extracted from GEBCO's 30-arc-sec gridded bathymetric data set, and then spatially averaged over 1 and 3 arc-min. The far-field propagation of tsunami is then simulated by FUNWAVE-GPU at two resolutions, with a duration of 24 hr.

Figure 10 shows the maximum tsunami wave height over the Pacific Ocean. To validate the model result, the 14 DART buoys located in the path of the tsunami are used for comparison (Figure 10). The records are detided by applying a second-order Butterworth band-pass filter with cutoff periods of 2 and 120 min. Figure 11 shows a comparison of computed and observed tsunami waveforms at the 14 DART buoys. Overall, results from FUNWAVE's GPU and CPU version are overlapped exactly, and both of them agree reasonably well with observations. Notably, the Fujii’s source underpredicts the leading wave crest elevation at DART Buoy 21418.

We also conduct a simulation at a higher spatial resolution of 1 arc-min across the Pacific Ocean. The mesh grid size is 9, 600 × 7, 200, which is nearly 70 million cells. The performance of GPU and CPU code are summarized in Table 4. It is worth noting that the GPU occupancy for simulation of 3, 200 × 2, 400 is only 71%. During the simulation, we also found that there is slight imbalance of workload between the two GPUs. The masked land grids are excluded from the computation, which results in the imbalance of workload for different GPUs. The CUDA threads corresponding to the land grids are actually idle at most times. For the latter case, a high-resolution simulation can complete around 8.2 hr on two GPUs We reran the case with 36 CPU cores on 2 nodes, which manages to complete the task at a cost of more than 4 days. The speedup can reach up to 12 over the original CPU code.

5. Conclusions

As computation-demanding Boussinesq wave model for studying nearshore wave dynamics and wave-induced current, FUNWAVE-TVD is accelerated by single- and multi-GPU with CUDA Fortran in this study. Based on a brief introduction on numerical scheme of FUNWAVE-TVD, the details on GPU implementation and its optimization are presented. Three test cases are carefully selected to validate the numerical schemes for Cartesian and spherical coordinates with different order of accuracy. The performance evaluation is measured by a metric of speedup ratio with increasing mesh grid sizes, which reveals that the GPU version shows its superiority over the original CPU code with the increase of modeling domain and growth in computation workload. Generally, a speedup of 5–7 over a fully loaded HPC 36-core node can be achieved for a large modeling domain. With the bridging of MPI, multiple GPUs can work together to further reduce the modeling time, making it possible to solve practical problems over large computational domains in an affordable stand-alone machine. It is worth noting that speedup ratios obtained by this study may show considerable variability in different hardware environments, as the evaluation of GPU performance by a fair metric is still an open issue.

As the future work, more improvement on the current GPU version will be considered, including optimization on model input/output, workload balance on GPUs, as well as inter-GPU communication.

References

Acuña, M. A., & Aoki, T. (2018). Tree-based mesh-refinement GPU-accelerated tsunami simulator for real-time operation. Natural Hazards and Earth System Sciences, 18(9), 2561–2602. https://doi.org/10.5194/nhess-18-2561-2018

Amouzgar, R., Liang, Q., & Smith, L. (2014). A GPU-accelerated shallow flow model for tsunami simulations. Proceedings of the Institution of Civil Engineers - Engineering and Computational Mechanics, 167(3), 117–125. https://doi.org/10.1680/eacm.13.00024

Ye Yuan is supported by the National Science and Technology Major Project of China (2016YFC14015), China Scholarship Council (2018041800016), and Innovative Youth Talents Scholarship Council (201804180016), China Science and Technology Major Project (3 and 1 arc-min, respectively). The tsunami source is based on the finite fault solution proposed by Fujii et al. (2011), obtained by inverting tsunami waveforms recorded at tide and wave gauges, GPS wave gauges, and deep water DART buoys (available at http://equake-rc.info/srcmod/). The initial surface condition is then determined by a summation of each subfault's vertical displacement by Okada's rectangular fault model. The model bathymetry from (130°E, −60°S) to (−70°W, 60°N) is extracted from GEBCO's 30-arc-sec gridded bathymetric data set, and then spatially averaged over 1 and 3 arc-min. The far-field propagation of tsunami is then simulated by FUNWAVE-GPU at two resolutions, with a duration of 24 hr.

Figure 10 shows the maximum tsunami wave height over the Pacific Ocean. To validate the model result, the 14 DART buoys located in the path of the tsunami are used for comparison (Figure 10). The records are detided by applying a second-order Butterworth band-pass filter with cutoff periods of 2 and 120 min. Figure 11 shows a comparison of computed and observed tsunami waveforms at the 14 DART buoys. Overall, results from FUNWAVE's GPU and CPU version are overlapped exactly, and both of them agree reasonably well with observations. Notably, the Fujii’s source underpredicts the leading wave crest elevation at DART Buoy 21418.

We also conduct a simulation at a higher spatial resolution of 1 arc-min across the Pacific Ocean. The mesh grid size is 9, 600 × 7, 200, which is nearly 70 million cells. The performance of GPU and CPU code are summarized in Table 4. It is worth noting that the GPU occupancy for simulation of 3, 200 × 2, 400 is only 71%. During the simulation, we also found that there is slight imbalance of workload between the two GPUs. The masked land grids are excluded from the computation, which results in the imbalance of workload for different GPUs. The CUDA threads corresponding to the land grids are actually idle at most times. For the latter case, a high-resolution simulation can complete around 8.2 hr on two GPUs We reran the case with 36 CPU cores on 2 nodes, which manages to complete the task at a cost of more than 4 days. The speedup can reach up to 12 over the original CPU code.

5. Conclusions

As computation-demanding Boussinesq wave model for studying nearshore wave dynamics and wave-induced current, FUNWAVE-TVD is accelerated by single- and multi-GPU with CUDA Fortran in this study. Based on a brief introduction on numerical scheme of FUNWAVE-TVD, the details on GPU implementation and its optimization are presented. Three test cases are carefully selected to validate the numerical schemes for Cartesian and spherical coordinates with different order of accuracy. The performance evaluation is measured by a metric of speedup ratio with increasing mesh grid sizes, which reveals that the GPU version shows its superiority over the original CPU code with the increase of modeling domain and growth in computation workload. Generally, a speedup of 5–7 over a fully loaded HPC 36-core node can be achieved for a large modeling domain. With the bridging of MPI, multiple GPUs can work together to further reduce the modeling time, making it possible to solve practical problems over large computational domains in an affordable stand-alone machine. It is worth noting that speedup ratios obtained by this study may show considerable variability in different hardware environments, as the evaluation of GPU performance by a fair metric is still an open issue.

As the future work, more improvement on the current GPU version will be considered, including optimization on model input/output, workload balance on GPUs, as well as inter-GPU communication.

References

Acuña, M. A., & Aoki, T. (2018). Tree-based mesh-refinement GPU-accelerated tsunami simulator for real-time operation. Natural Hazards and Earth System Sciences, 18(9), 2561–2602. https://doi.org/10.5194/nhess-18-2561-2018

Amouzgar, R., Liang, Q., & Smith, L. (2014). A GPU-accelerated shallow flow model for tsunami simulations. Proceedings of the Institution of Civil Engineers - Engineering and Computational Mechanics, 167(3), 117–125. https://doi.org/10.1680/eacm.13.00024
