Acceleration in Acoustic Wave Propagation Modelling Using OpenACC/OpenMP and Its Hybrid for the Global Monitoring System

Noriyuki Kushida[1(✉)], Ying-Tsong Lin[2], Peter Nielsen[1], and Ronan Le Bras[1]

[1] Comprehensive Nuclear-Test Ban Treaty Organization, Vienna, Austria
{noriyuki.kushida,peter.nielsen,ronan.lebras}@ctbto.org
http://www.ctbto.org
[2] Woods Hole Oceanographic Institution, Woods Hole, MA 02543, USA
ytlin@whoi.edu

Abstract. CTBTO is operating and maintaining the international monitoring system of Seismic, Infrasound, Hydroacoustic and Airborne radionuclide facilities to detect a nuclear explosion over the globe. The monitoring network of CTBTO, especially with regard to infrasound and hydroacoustic, is quite unique because the network covers over the globe, and the data is opened to scientific use. CTBTO has been developing and improving the methodologies to analyze observed signals intensively. In this context, hydroacoustic modelling software, especially which that solves the partial differential equation directly, is of interest. As seen in the analysis of the Argentinian submarine accident, the horizontal reflection can play an important role in identifying the location of an underwater event, and as such, accurate modelling software may help analysts find relevant waves efficiently. Thus, CTBTO has been testing a parabolic equation based model (3D-SSFPE) and building a finite difference time domain (FDTD) model. At the same time, using such accurate models require larger computer resources than simplified methods such as ray-tracing. Thus we accelerated them using OpenMP and OpenACC, or the hybrid of those. As a result, in the best case scenarios, (1) 3D-SSFPE was accelerated by approximately 19 times to the original Octave code, employing the GPU-enabled Octfile technology, and (2) FDTD was accelerated by approximately 160 times to the original Fortran code using the OpenMP/OpenACC hybrid technology, on our DGX—Station with V100 GPUs.

Keywords: OpenACC/OpenMP hybrid · OpenACC with Octave/Matlab · Split Step Fourier · FDTD · Hydroacoustic modelling

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-030-49943-3_2) contains supplementary material, which is available to authorized users.

S. Wienke and S. Bhalachandra (Eds.): WACCPD 2019, LNCS 12017, pp. 25–46, 2020.
https://doi.org/10.1007/978-3-030-49943-3_2

1 Introduction

The Comprehensive Nuclear-Test-Ban Treaty (CTBT) is the treaty which bans nuclear explosions in any environment over the globe, such as in the atmosphere, in the ocean, and underground. Although the treaty has not entered into force, Preparatory Commission for the CTBT Organization (CTBTO) has been monitoring signs of nuclear explosions using four technologies, namely, seismic, infrasound, hydroacoustic and air-borne radionuclide. The monitoring network of CTBTO, especially with regard to infrasound and hydroacoustic, is quite unique because the network covers over the globe and the data is opened to scientific use. Therefore, CTBTO has been developing and improving the methodologies to analyze observed signals intensively. Because of the complex natures of the oceans and the atmosphere, computer simulation can play an important role in understanding the observed signals. In this regard, methods which depend on partial differential equations, in other words an "*ab-initio*" approach, are preferable in order not to overlook any subtle phenomena. However, there have been only a few groups which perform such computer modelling with the parabolic equation (PE) methods [10,11]. Based on such circumstances, CTBTO has been testing and developing hydroacoustic simulation software packages based on PE called 3D-SSFPE [20], and the finite difference method (FDM) [18] respectively. Lin et al. explained the advantages of 3D-SSFPE over other PE methods in their literature i.e. 3D-SSFPE is designed for long distance modelling.

One of the biggest drawbacks of using such accurate methods is the high demand on computer resources, especially the arithmetic computing performance. Although computer simulation is not considered as a necessary product for the treaty, providing the member states with modelling results promptly may help their decision-making.

At the same time, computing accelerators such as general purpose graphics processing unit (GPGPU), field-programmable gate arrays (FPGA), and so forth are now prevalent in the computer simulation field. Particularly, a GPGPU is available at an affordable price thanks to the active development in deep learning. Thus, we have started evaluating the performance gain with GPGPUs on our simulation programs. Considering the effort we could spare for porting the codes, the directive-based parallel programming is practically the only choice for a non-research organization, even though there is a known gap in the achievable performance between the special languages, such as CUDA and OpenCL, and the directive based parallel programming languages, such as OpenACC. Finally, we have implemented our software on the DGX-station by NVIDIA using OpenACC.

In the following sections, the details of each program and performance evaluation results as well as the computing environment will be described.

2 Computing Environment

In the present study, we have employed the DGX–station produced by NVIDIA [1] as a test–bed of a GPGPU environment. The DGX–station equips four NVIDIA Tesla V100s. In the OpenACC framework, using one GPU per pro-

cess is the standard. In the present study, we employed OpenMP to launch multiple threads and assigned one GPU to a thread for multi-GPU computation in Sect. 4. The CPU installed is Intel Xeon E5-2698 v4 20 cores. The theoretical peak performance of the CPU is 0.576 tera floating point operations per second (TFLOPS) in double precision (0.0288 TFLOPS/core) and the memory bandwidth is 71.53 giga byte per second (GB/s). At the same time, a V100 performs 7.5 TFLOPS in double precision and 900GB/s. The operating system is Ubuntu 18.04.2 LTS. The CUDA version is 10.0. The compilers employed are (1) GCC version 9.1, which is available through the ubuntu-toolchain-r/test repository [5] and (2) PGI compiler version 19.5. Both compilers have the capability of compiling OpenACC-enabled codes. However, the PGI compiler fails to generate an object file which works together with Octave. Therefore, we employed GCC for 3D-SSFPE, while the PGI compiler provides advanced features of OpenACC such as managed memory. The PGI technical team recognizes this issue. Readers who try to implement OpenACC-enabled codes on Octave with the PGI compiler may check the release notes. The PGI compiler for Fortran 90 (pgf90) was used throughout the FDM simulation development and it's performance evaluation. Octave is installed using `apt-get` and the version is 4.2.2.

3 3D-SSFPE

3.1 Overview

3D-SSFPE is a three-dimensional (3D) underwater sound propagation model based on the parabolic equation (PE) combined with the Split-Step Fourier method (SSF) [20]. In the literature, Lin et al. developed the code in the three dimensional Cartesian coordinate as well as a cylindrical coordinate. In the present study, since the long range wave propagation is of interest, we focus on the Cartesian coordinate. The theoretical background as well as the implementation of 3D-SSFPE are given in the literature, and here, we provide readers with a brief explanation, toward the GPU implementation. The 3D SSF solves a linear wave equation by marching a two-dimensional (2D) grid ($Y-Z$ plane) along the perpendicular direction of the grid (X-axis) from the source term to the point of interest, and each 2D solution grid is computed in spatial and wavenumber domains alternately. The spatial and wavenumber domain transform is performed through Fast Fourier Transform. As an analogy, the computation progresses like a wave-front propagates. However, SSF does not solve the time evolution, instead it solves a boundary value problem. In the implementation point of view, 3D-SSFPE repeats the following steps;

1. Calculate the sound pressure at $x_{n+1/2} = x_n + \Delta x/2$ in the wavenumber domain
2. Correct the amplitude and phase of the sound pressure due to sound speed changes (the index of refraction) at $x_{n+1/2}$ in the spatial domain
3. Proceed to x_{n+1} in the wavenumber domain
4. if necessary, update the environment information

where x_n denotes the nth grid point on the X-axis, and Δx denotes the grid length. The conversion of the pressure between the spatial and wavenumber domains is undertaken using the Fast Fourier Transform (FFT) and the inverse FFT (iFFT). Since there are many well-tuned FFT libraries, the point of the discussion of implementation is how to compute the remaining part efficiently. In the following section, we discuss it for our computing environment.

3.2 Implementation

3D-SSFPE has been developed on Matlab, which is a well known commercial scientific software development environment. 3D-SSFPE solves the state of a wave of a specific frequency, whilst hydroacoustic researchers would like to solve problems of a spatial domain with various wave frequencies. It is worth noting that the problem of each frequency is completely independent and can be solved in parallel. Therefore, launching multiple instances at the same time is beneficial. In other words, avoiding the limitation on the number of licenses increases the total computational speed. Thus we employed Octave, which is an open source clone of Matlab. It should be also noted that pre/post processes are implemented with Matlab's unique file format, and we can avoid the re-implementation of such processes by using Octave. 3D-SSFPE calls FFT and iFFT frequently. In our preliminary experiment, the GPU enabled FFT function on Matlab was also examined. However, probably because of the data transfer between GPU and CPU, the total computational speed remained the same level with the non-GPU version. This fact also motivated us to use OpenACC.

Octave, as well as Matlab, provides users with a functionality to call a routine written in C++. The functionality is called "Octfile" in Octave, or "Mex file" in Matlab. Since the Octfile is written in C++, we can apply OpenMP and OpenACC. Considering the similarity in OpenMP and OpenACC, and the complexity appears only in OpenACC, specifically data transfer, we followed the following three steps, namely;

1. Re-write the target functions in C++
2. Apply the OpenMP directives
3. Apply the OpenACC directives based on 2

FFT and iFFT are performed using FFTW [9](Step 1 and 2) or cuFFT [4](in Step 3).

In the following sections, the details will be given to apply OpenMP and OpenACC to the Octfile. For readers' convenience, a 3D SSF implementation for the Lloyd's Mirror Problem in Matlab/Octave and its GPU version are available on Zenodo (Matlab/Octave version [19] and GPU version [15]). Although 3D-SSFPE deals with more complex geometries and the inhomogeneity of medium which lead to additional complexity into the codes, we believe those samples help readers understand the efforts in the present study.

Accessing Arrays. Octave provides users with the functionality to access a multidimensional array in the Fortran style, specifically in the "`array(i,j)`" form, in the Octfile. However, this style involves function calls which prevents OpenACC compilers from generating GPU enabled loops. On the other hand, there is a way to handle a raw pointer of an array. In the present study, we construct all loops with raw pointers of multi-dimensional arrays including vectors and matrices so that all the loops can be computed on GPU. In the following program, a matrix `Mat` is initialized using the raw pointer. In the program, `Mat` is a matrix defined in Octave and passed to the Octfile, `Nc` and `Nr` are the numbers of columns and rows of `Mat`, and `Mat_p` is the pointer, which stores the content of `Mat`. As the name of the method to obtain the raw pointer, `fortran_vec()`, indicates, the matrix is stored in the column-major manner as Fortran. A vector and higher dimensional arrays can be accessed in the same way.

Example of the initialization of a matrix with the raw pointer

```
double _Complex *Mat_p = reinterpret_cast<double _Complex *>
 (const_cast<Complex *>(Mat.fortran_vec()));
octave_idx_type  Nc = Mat.cols();
octave_idx_type  Nr = Mat.rows();
for (int i=0; i< Nr; i++){
   for (int j=0; j< Nc; j++){
      Mat_p[j*Nr+i] = 0.0;
   }
}
```

Double Complex Data Type. Since 3D-SSFPE uses FFT and iFFT, a complex value data type is necessary. In the Octfile, the `Complex` data type is the standard. However, with GCC version 9.1, more precisely, g++ version 9.1, the `Complex` data type prevents the compiler from generating GPU enabled loops. In the present study, we discovered that the `double _Complex` data type can be used as an alternative of the `Complex` data type although this is a data type in C. The binary format of the `double _Complex` data type and the `Complex` data type is identical, and users can use the `double _Complex` data type with matrices by casting the data type. The actual usage can be found in Section **Accessing arrays**. With the `double _Complex` data type, compilers can generate GPU enabled loops, even with mathematical functions such as `cexp`, which computes the exponential of a complex value.

Memory Mapping for OpenACC. Basically, OpenACC compilers should allocate arrays on GPU automatically using directives, such as `#pragma acc data copy`. However, in the Octfile, neither GCC nor PGI handles such directives as expected. More precisely, error messages relevant to the memory address were observed such as "Failing in Thread:1 call to cuMemcpyDtoHAsync returned error 700: Illegal address during kernel execution". In the present study, we

manually allocate arrays on GPU and associate them with corresponding arrays on the main memory. In the following program, `acc_malloc` allocates arrays on GPU, and `acc_map_data` associates arrays on GPU with corresponding arrays on CPU. Finally, users can generate a GPU enabled loop using directives.

Example of the allocation of matrix on CPU and GPU in the Octfile

```
octave_idx_type  Nc = Mat.cols();
octave_idx_type  Nr = Mat.rows();

double _Complex *Mat_p = reinterpret_cast<double _Complex *>
                         (const_cast<Complex *>(Mat.fortran_vec()));
double _Complex *Mat_d = (double _Complex*)
                         acc_malloc(sizeof(double _Complex)*Nc*Nr);
acc_map_data(Mat_p,Mat_d,sizeof(double _Complex)*Nc*Nr);

#pragma acc parallel loop independent present(Mat_p[0:Nc*Nr])
for (int i=0; i< Nr*Nc; i++){
   Mat_p[i] = 0.0;
}
```

Calling cuFFT with OpenACC. CuFFT is the FFT library which is one of the best for GPUs provided by NVIDIA. NVIDIA also provides another FFT library, called cuFFTW. The main difference in those libraries is that cuFFTW takes arrays on CPU as input, while cuFFT takes arrays on GPU as input. Since we are aiming to confine all the arrays into GPU, cuFFT is the choice in the present study. CuFFT is designed to use together with CUDA, which is the language that handles the pointers of arrays on GPU explicitly, whilst OpenACC handles the pointers on GPU implicitly. To circumvent this issue, we employed `#pragma acc host_data use_device` as inspired by the site [2]. An example of the Octfile which calls cuFFT from OpenACC is shown in the following program. In the program, a matrix `Matrix` is given from Octave, and iFFT is performed on it. The result is stored in `out`, and normalized using the number of matrix elements as Octave's native function does. The working example can be found at Zenodo [17]. It is worth noting that, since both the iFFT part and the normalization part are in the same `#pragma acc data copy` region, only the arrays on GPU are accessed during the computation, which is essential for performance.

Example of calling cuFFT from OpenACC in the Octfile

```
void inv_CUFFT(double _Complex *in_data,
               double _Complex *out_data,
               int nc, int nr, void *stream)
{
    cufftHandle plan;
```

```
    cufftResult ResPlan = cufftPlan2d(&plan, nc,nr, CUFFT_Z2Z);
    cufftSetStream(plan, (cudaStream_t)stream);
    cufftResult ResExec = cufftExecZ2Z(plan,
                                       (cufftDoubleComplex*)in_data,
                                       (cufftDoubleComplex*)out_data,
                                       CUFFT_INVERSE);
    cufftDestroy(plan);
}

DEFUN_DLD(testFFTGPU, args, ,
            "main body;")
{
   ComplexMatrix Matrix(args(0).complex_matrix_value());
   octave_value_list retval;
   ComplexMatrix out(Matrix.dims());

   double _Complex *pmat = reinterpret_cast<double _Complex*>
                    (const_cast<Complex *>(Matrix.fortran_vec()));
   double _Complex *pout = reinterpret_cast<double _Complex*>
                    (const_cast<Complex *>(out.fortran_vec()));
   int Nc = pmat.cols();
   int Nr = pmat.rows();
#pragma acc data copy(pmat[0:Nc*Nr],pout[0:Nc*Nr])
{
   void *stream = acc_get_cuda_stream(acc_async_sync);
#pragma acc host_data use_device(pmat,pout)
{
   inv_CUFFT((double _Complex*)pmat,(double _Complex*)pout,Nc,Nr,stream);
}

#pragma acc parallel
   for(int i=0;i<Nr*Nc;i++){
      pout[i] = pout[i]/double(Nr*Nc);
   }
}
   retval(0) = out;
   return retval;
}
```

Conditional Access. It is well known that loops should be avoided in Matlab and Octave, in terms of computational speed. Instead, users are encouraged to use a technique called vectorization (note that, this vectorization is not equivalent to the one in the context of supercomputing, especially on vector supercomputers). In the vectorization technique, users apply built-in operations and functions which perform over the entire elements of vectors and arrays. In the

case that one needs to process only a part of an array, a new array needs to be created, as;

```
smallArray = Array(find(X > 0.0))
```

In the above example, `Array` and `X` are the vectors which have the same number of elements, and if an element of `X` is greater than 0.0, the corresponding element of `Array` is extracted and copied to `smallArray`. Finally, one can apply vectorized operations on `smallArray`. On the other hand, the same operation can be implemented in C++ as follows, where `n` is the size of `Array` and `X`;

```
int idx = 0;
for(int i; i<n; i++){
   if(X[i] > 0.0){
      smallArray[idx] = Array[i];
      idx++;
   }
}
```

This operation cannot be parallelized, because `idx` needs to be incremented sequentially, and therefore, it cannot be implemented on GPU. As a result, `Array` and `smallArray` need to be transferred between CPU and GPU, which should be avoided from the computational performance point of view. In the present study, we avoided creating such small arrays by processing all the elements of `Array` with `if` branch.

3.3 Performance Evaluation and Conclusion

In order to evaluate the performance of the GPU implementation as well as the multi-core implementation with OpenMP, the computation time was measured. Table 1 gives the computation times of each implementation on three sizes of problems, namely $ny \times nz = 1000 \times 2000$, 2000×4000, 4000×8000, where ny and nz are the numbers of grid points along the Y-axis and the Z-axis respectively. In the table, "Octave + FFTW (single)" denotes the computation time of Octave with a single threaded FFTW, "Octave + FFTW (20cores)" denotes the computation time of Octave with multi-threaded FFTW on 20 cores, "OpenMP" denotes the multi-threaded version with OpenMP on 15 cores, and "OpenACC" denotes the GPU version. The numbers listed in the table represent the computation time in seconds. Figure 1 shows the relative acceleration of each implementation to "Octave + FFTW (single)". In all the cases, OpenACC shows the best performance in all implementations. In addition, the larger the problem becomes, the better the acceleration becomes. In the largest case, OpenACC is approximately 19 times faster than Octave + FFTW (single). Regarding OpenMP, the same tendency with OpenACC is observed although the performance is worse than Octave + FFTW (single), in the smallest case. Except in the smallest problem case, OpenMP shows better performance than "Octave + FFTW (20cores)".

At the same time, "Octave + FFTW (20cores)" is always better than "Octave + FFTW (single)". This implies that although FFT is dominant in 3D-SSFPE, the remaining parts are not negligible.

Table 1. Computation time of each implementation on various problem sizes. Computation time in seconds is listed. Problem size indicates the numbers of grid points along the Y-axis and the Z-axis. The marching distance along the X-axis is identical in all cases.

Problem size	Octave + FFTW (single)	Octave + FFTW(20cores)	OpenMP	OpenACC
4000×8000	163	103	25.1	8.60
2000×4000	36.8	25.3	9.52	2.94
1000×2000	5.11	3.68	5.65	1.58

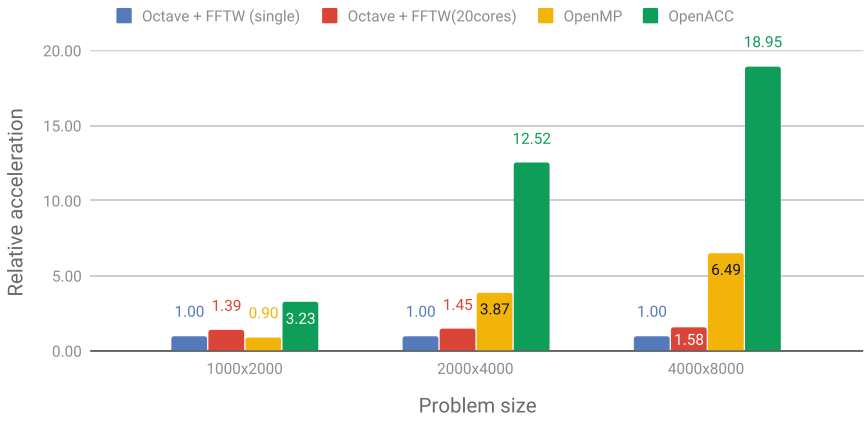


Fig. 1. Relative acceleration of each implementation to Octave + FFTW (single). Problem size indicates the numbers of grid points along the Y-axis and the Z-axis. The marching distance along the X-axis is identical in all cases.

Finally, we can conclude that porting the entire kernel of 3D-SSFPE to GPU is necessary for high performance even though a well-tuned FFT on GPU is provided. This is because (1) the communication between CPU and GPU is expensive as well known and in order to avoid such communication, all the computation should be performed on GPU, and (2) although FFT is dominant, the remaining parts are also perceptible if FFT is sufficiently fast.

4 Global Acoustic Simulation with FDM

4.1 Overview

As discussed, an FDM-based global acoustic model has a favorable nature to analyze CTBTO observation. One of the advantages of employing a FDM model

over PE methods is that one can apply a waveform directly to a source term. However, so far, to our best knowledge, such models have not been developed (pp 147–161 of ref [8,23]). The main difficulties in building such simulation codes are: (1) considering the inhomogeneity of medium including background flows, (2) high aspect ratio of computational domain, (3) stability during long time integration. To overcome these difficulties, we employ a 2D FDM scheme on a spherical coordinate with the Yin-Yang overset grid [12] solving the governing equation of acoustic waves introduces by Ostashev et al. [21]. In the following section, we discuss the formulation and the implementation on CPU as well as GPU.

4.2 Formulation

Ostashev et al. give the formulation of the wave propagation over the moving inhomogeneous media as;

$$\left(\frac{\partial}{\partial t} + \mathbf{v}\cdot\nabla\right) p + \rho c^2 \nabla\cdot\mathbf{w} = \rho c^2 Q \tag{1}$$

$$\left(\frac{\partial}{\partial t} + \mathbf{v}\cdot\nabla\right) \mathbf{w} + (\mathbf{w}\cdot\nabla)\,\mathbf{v} + \frac{\nabla p}{\rho} = \frac{\mathbf{F}}{\rho} \tag{2}$$

where p is the pressure, $\mathbf{w}$ is the velocity vector of the wave, $\mathbf{v}$ is the velocity vector of background media, ρ is the density of the background media, c is the adiabatic sound speed, Q is a mass source, and $\mathbf{F}$ is a force acting on the background media. Since we are interested in solving the wave propagation over the globe, we need to know the explicit form of Eqs. 1 and 2 in a spherical coordinate. We now define the spherical coordinate we use in the present study as;

$$\begin{aligned} x &= r\sin\theta\cos\phi \\ y &= r\sin\theta\sin\phi \\ z &= r\cos\theta \end{aligned} \tag{3}$$

where the radial distance r, the inclination θ, and the azimuth ϕ. Based on Eq. 3, we have the operator ∇,

$$\nabla = \left(\frac{\partial}{\partial r}, \frac{1}{r}\frac{\partial}{\partial\theta}, \frac{1}{r\sin\theta}\frac{\partial}{\partial\phi}\right). \tag{4}$$

We write down the explicit form of $(\mathbf{v}\cdot\nabla)\,\mathbf{w}$,

$$(\mathbf{v}\cdot\nabla)\,\mathbf{w} = \left(v_r\frac{\partial}{\partial r} + \frac{v_\theta}{r}\frac{\partial}{\partial\theta} + \frac{v_\phi}{r\sin\theta}\frac{\partial}{\partial\phi}\right)\left(w_r\mathbf{e}_r + w_\theta\mathbf{e}_\theta + w_\phi\mathbf{e}_\phi\right), \tag{5}$$

where $\mathbf{e}_r$, $\mathbf{e}_\theta$ and $\mathbf{e}_\phi$ are the unit vectors of r, θ and ϕ, and v and w with subscripts denote the components of each vector along each direction. Finally,

$$\begin{aligned}(\mathbf{v}\cdot\nabla)\,\mathbf{w} = &\left(v_r\frac{\partial w_r}{\partial r} + \frac{v_\theta}{r}\frac{\partial w_r}{\partial \theta} + \frac{v_\phi}{r\sin\theta}\frac{\partial w_r}{\partial \phi} - \frac{v_\theta w_\theta + v_\phi w_\phi}{r}\right)\mathbf{e}_r \\ &+ \left(v_r\frac{\partial w_\theta}{\partial r} + \frac{v_\theta}{r}\frac{\partial w_\theta}{\partial \theta} + \frac{v_\phi}{r\sin\theta}\frac{\partial w_\theta}{\partial \phi} + \frac{v_\theta w_r}{r} - \frac{v_\phi w_\phi\cot\theta}{r}\right)\mathbf{e}_\theta \\ &+ \left(v_r\frac{\partial w_\phi}{\partial r} + \frac{v_\theta}{r}\frac{\partial w_\phi}{\partial \theta} + \frac{v_\phi}{r\sin\theta}\frac{\partial w_\phi}{\partial \phi} + \frac{v_\phi w_r}{r} + \frac{v_\phi w_\theta\cot\theta}{r}\right)\mathbf{e}_\phi.\end{aligned} \tag{6}$$

In the same way, we have,

$$\begin{aligned}(\mathbf{w}\cdot\nabla)\,\mathbf{v} = &\left(w_r\frac{\partial v_r}{\partial r} + \frac{w_\theta}{r}\frac{\partial v_r}{\partial \theta} + \frac{w_\phi}{r\sin\theta}\frac{\partial v_r}{\partial \phi} - \frac{w_\theta v_\theta + w_\phi v_\phi}{r}\right)\mathbf{e}_r \\ &+ \left(w_r\frac{\partial v_\theta}{\partial r} + \frac{w_\theta}{r}\frac{\partial v_\theta}{\partial \theta} + \frac{w_\phi}{r\sin\theta}\frac{\partial v_\theta}{\partial \phi} + \frac{w_\theta v_r}{r} - \frac{w_\phi v_\phi\cot\theta}{r}\right)\mathbf{e}_\theta \\ &+ \left(w_r\frac{\partial v_\phi}{\partial r} + \frac{w_\theta}{r}\frac{\partial v_\phi}{\partial \theta} + \frac{w_\phi}{r\sin\theta}\frac{\partial v_\phi}{\partial \phi} + \frac{w_\phi v_r}{r} + \frac{w_\phi v_\theta\cot\theta}{r}\right)\mathbf{e}_\phi.\end{aligned} \tag{7}$$

The divergence of $\mathbf{w}$ is,

$$\nabla\cdot\mathbf{w} = \frac{1}{r}\frac{\partial}{\partial r}\left(r^2 w_r\right) + \frac{1}{r\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta w_\theta\right) + \frac{1}{r\sin\theta}\frac{\partial w_\phi}{\partial\phi}. \tag{8}$$

By using Eqs. 6, 7, and 8, we have the explicit form of Eqs. 1 and 2 in the spherical coordinate,

$$\begin{aligned}\frac{\partial p}{\partial t} =& -\left(v_r\frac{\partial p}{\partial r} + \frac{v_\theta}{r}\frac{\partial p}{\partial\theta} + \frac{v_\phi}{r\sin\theta}\frac{\partial p}{\partial\phi}\right) \\ &-\kappa\left(\frac{1}{r}\frac{\partial}{\partial r}\left(r^2 w_r\right) + \frac{1}{r\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta w_\theta\right) + \frac{1}{r\sin\theta}\frac{\partial w_\phi}{\partial\phi}\right) + \kappa Q, \\ \frac{\partial w_r}{\partial t} =& -\left(v_r\frac{\partial w_r}{\partial r} + \frac{v_\theta}{r}\frac{\partial w_r}{\partial\theta} + \frac{v_\phi}{r\sin\theta}\frac{\partial w_r}{\partial\phi} - \frac{v_\theta w_\theta + v_\phi w_\phi}{r}\right) \\ &-\left(w_r\frac{\partial v_r}{\partial r} + \frac{w_\theta}{r}\frac{\partial v_r}{\partial\theta} + \frac{w_\phi}{r\sin\theta}\frac{\partial v_r}{\partial\phi} - \frac{w_\theta v_\theta + w_\phi v_\phi}{r}\right) - b\frac{\partial p}{\partial r} + bF_r, \\ \frac{\partial w_\theta}{\partial t} =& -\left(v_r\frac{\partial w_\theta}{\partial r} + \frac{v_\theta}{r}\frac{\partial w_\theta}{\partial\theta} + \frac{v_\phi}{r\sin\theta}\frac{\partial w_\theta}{\partial\phi} + \frac{v_\theta w_r}{r} - \frac{v_\phi w_\phi\cot\theta}{r}\right) \\ &-\left(w_r\frac{\partial v_\theta}{\partial r} + \frac{w_\theta}{r}\frac{\partial v_\theta}{\partial\theta} + \frac{w_\phi}{r\sin\theta}\frac{\partial v_\theta}{\partial\phi} + \frac{w_\theta v_r}{r} - \frac{w_\phi v_\phi\cot\theta}{r}\right) - b\frac{1}{r}\frac{\partial p}{\partial\theta} + bF_\theta, \\ \frac{\partial w_\phi}{\partial t} =& -\left(v_r\frac{\partial w_\phi}{\partial r} + \frac{v_\theta}{r}\frac{\partial w_\phi}{\partial\theta} + \frac{v_\phi}{r\sin\theta}\frac{\partial w_\phi}{\partial\phi} + \frac{v_\phi w_r}{r} + \frac{v_\phi w_\theta\cot\theta}{r}\right) \\ &-\left(w_r\frac{\partial v_\phi}{\partial r} + \frac{w_\theta}{r}\frac{\partial v_\phi}{\partial\theta} + \frac{w_\phi}{r\sin\theta}\frac{\partial v_\phi}{\partial\phi} + \frac{w_\phi v_r}{r} + \frac{w_\phi v_\theta\cot\theta}{r}\right) - \frac{b}{r\sin\theta}\frac{\partial p}{\partial\phi} + bF_\phi,\end{aligned} \tag{9}$$

where $\kappa = \rho c^2$ and $b = 1/\rho$. By dropping the radial component in Eq. 9, we have the governing equation of the wave propagation in the horizontal direction,

$$\begin{aligned}
\frac{\partial p}{\partial t} &= -\left(\frac{v_\theta}{r}\frac{\partial p}{\partial \theta} + \frac{v_\phi}{r\sin\theta}\frac{\partial p}{\partial \phi}\right) - \kappa\left(\frac{1}{r\sin\theta}\frac{\partial}{\partial \theta}\left(\sin\theta w_\theta\right) + \frac{1}{r\sin\theta}\frac{\partial w_\phi}{\partial \phi}\right) + \kappa Q, \\
\frac{\partial w_\theta}{\partial t} &= -\left(\frac{v_\theta}{r}\frac{\partial w_\theta}{\partial \theta} + \frac{v_\phi}{r\sin\theta}\frac{\partial w_\theta}{\partial \phi} - \frac{v_\phi w_\phi \cot\theta}{r}\right) - \left(\frac{w_\theta}{r}\frac{\partial v_\theta}{\partial \theta} + \frac{w_\phi}{r\sin\theta}\frac{\partial v_\theta}{\partial \phi} - \frac{w_\phi v_\phi \cot\theta}{r}\right) \\
&\quad - b\frac{1}{r}\frac{\partial p}{\partial \theta} + bF_\theta, \\
\frac{\partial w_\phi}{\partial t} &= -\left(\frac{v_\theta}{r}\frac{\partial w_\phi}{\partial \theta} + \frac{v_\phi}{r\sin\theta}\frac{\partial w_\phi}{\partial \phi} + \frac{v_\phi w_\theta \cot\theta}{r}\right) - \left(\frac{w_\theta}{r}\frac{\partial v_\phi}{\partial \theta} + \frac{w_\phi}{r\sin\theta}\frac{\partial v_\phi}{\partial \phi} + \frac{w_\phi v_\theta \cot\theta}{r}\right) \\
&\quad - \frac{b}{r\sin\theta}\frac{\partial p}{\partial \phi} + bF_\phi.
\end{aligned} \tag{10}$$

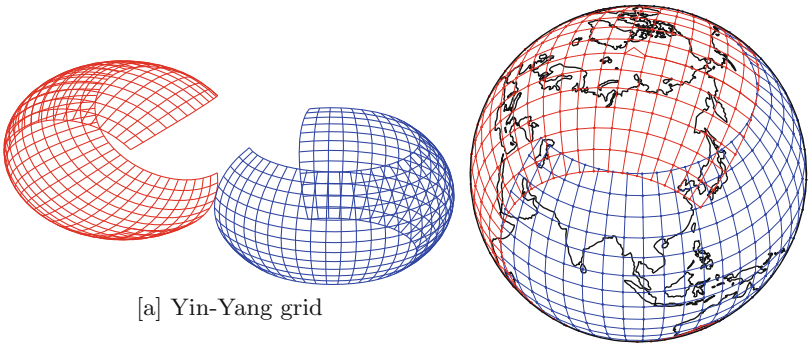
In the present study, we employ Eq. 10 to solve the global acoustic wave propagation.

4.3 Yin-Yang Grid

Kageyama and Sato developed an overset grid Called the Yin-Yang grid to overcome the issues in FDM in a spherical coordinate. Namely, the Yin-Yang Grid resolves the singularity at poles and provides a near uniform grid over the globe. On the other hand, it leads to an additional complexity which originates in combining two identical grids. The Yin-Yang grid, as the name implies, uses two identical grids. In Fig. 2, we visualized the positional relationship of the Yin and Yang grids. As can be seen in Fig. 2 [a], both are identical and a grid can be projected onto the other grid only with rotation. We also visualised the locations of each grid on the globe (Fig. 2 [b]). Since those two grids are identical, there is no need to distinguish them by applying labels. However, for convenience, we put the label of Yin on the blue grid, and Yang on the red grid. In order to comprehend the geographical location of each grid, we projected them with the plate carree projection as well (Fig. 3). With those figures, we see that the poles are covered by the Yang grid, whilst neither the Yin nor Yang grid has poles which appear in the standard spherical coordinate grid. In addition, we can observe that the grid width, which determines the time step length and therefore the total computation time, is uniform. More precisely, in the standard spherical coordinate grid, the grid lengths close to the poles are smaller than the ones around the equator. Because of the stability in numerical computing, one needs to choose a time step length based on the smallest grid length, which results in a larger number of time steps. In this context, a uniform grid length reduces the total computation time.

In the Yin-Yang grid, the computation proceeds by exchanging the physical values on the boundaries of each grid. Kageyama and Sato pointed out that there are regions which are computed twice, around the corners of each grid. Ideally, such wasteful computations should be avoided, and Kageyama and Sato also proposed a grid for this purpose. However, because of the following reasons, we employed the one which is visualized in the figures; (1) The optimized

grid requires additional "if" statement, which may impact on the computational speed, especially on GPU, (2) The number of such doubly computed grids is negligible in practice.



[a] Yin-Yang grid

[b] Mapped on the globe

Fig. 2. Visualization of Yin-Yang grid. [a] Yin and Yang grids are identical and can be projected only with rotation [b] Yin-Yang grid mapped on the globe (Color figure online)

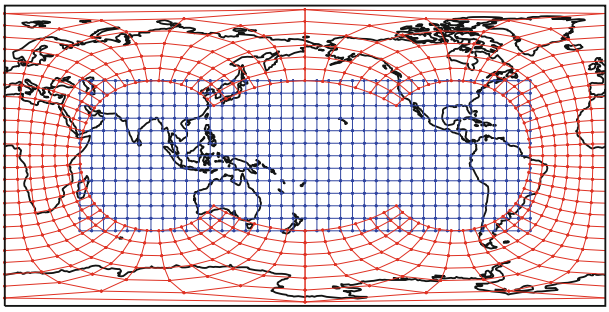


Fig. 3. Yin-Yang grid projected on the globe with the plate carree projection (Color figure online)

4.4 Computational Schemes

In the present study, we followed the computational schemes which Ostashev had employed, namely, the first order staggered grid for spatial discretization, and the fourth order Runge-Kutta explicit time integration. Figure 4 shows the schematic figure of the configuration of grid points. In the figure, the black dots represent p and other scalar values, and the triangles represent the θ and ϕ components of $\mathbf{w}$ and other vector values. A set of numbers in brackets indicates

the addresses of arrays in the Fortran notation. The dashed line indicates the region which need to be computed, and values on the grid points outside the region need to be imported from the other Yin-Yang grid. Equation 11 shows the time integration with the Runge–Kutta method, where n denotes the time step number, ψ_n denotes the physical value at the time step "n", t_n denotes the time at the time step, and f represents a function. ψ^* and ψ^{**} are physical values at intermediate steps which are only for computation. If we assume that the time derivative can be derived using the first order differential, we can introduce the explicit form of f using Eq. 10 without any difficulty.

$$
\begin{aligned}
\psi^*_{n+\frac{1}{2}} &= \psi_n + \frac{\Delta t}{2} f\left(t_n, \psi_n\right) \\
\psi^{**}_{n+\frac{1}{2}} &= \psi_n + \frac{\Delta t}{2} f\left(t_{n+\frac{1}{2}}, \psi^*_{n+\frac{1}{2}}\right) \\
\psi^*_{n+1} &= \psi_n + \Delta t f\left(t_{n+\frac{1}{2}}, \psi^{**}_{n+\frac{1}{2}}\right) \\
\psi_{n+1} &= \psi_n + \frac{\Delta t}{6}\left[f\left(t_n, \psi_n\right) + 2f\left(t_{n+\frac{1}{2}}, \psi^*_{n+\frac{1}{2}}\right) + 2f\left(t_{n+\frac{1}{2}}, \psi^{**}_{n+\frac{1}{2}}\right) + f\left(t_{n+1}, \psi^*_{n+1}\right)\right]
\end{aligned}
\tag{11}
$$

4.5 Performance Optimization

Merging Function Evaluations. Equation 10 gives the functions that should be evaluated in Eq. 11. In a naive implementation, those three functions are implemented into three individual functions. However, at the same time, one can point out that most variables appear in all three equations. For instance, variables loaded to evaluate the first equation can be used in the remaining equations. As well known, modern computers including GPUs, have high arithmetic intensity (the number of floating point operations per word). In other words, reducing the number of memory instructions is the key technique to achieve high performance. Thus, we evaluate those three equations within the same loop so that a variable is reused as many times as possible. In Table 2, the computation times of the naive implementation and the merged function implementation are listed, as well as the speedup from the naive implementation to the merged implementation. The computational time is measured on Intel Xeon CPU E5-1620 v3 one core, only with the Yin grid. The grid sizes used are $n\phi \times n\theta = 1000 \times 3000$ (10 km/grid on the equator), and 3000×9000 (3.3 km/grid) where $n\phi$ and $n\theta$ are the numbers of grid points along ϕ and θ directions, respectively. In the smaller problem case, we obtained 4.85 times acceleration while we obtained 3.31 times acceleration in the larger problem case. The reason why the larger case shows smaller improvement can be attributed to the data cache memory working more effectively in the smaller problem. Since the number of floating point operations required stays the same in both implementations, we believe

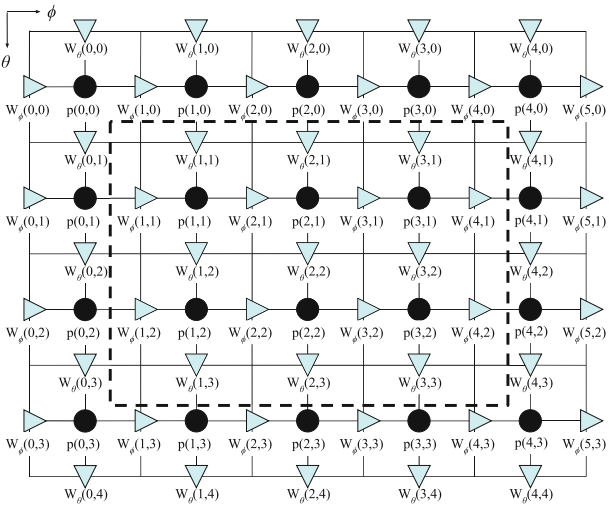


Fig. 4. Configuration of the staggered grid.

that the improvement originates in the reduction of memory access. Therefore, we can expect that this optimization is also effective on GPU, although we are not able to perform this test on GPU because the optimization was applied at an early stage of the development.

Table 2. Computational times of the naive implementation and the merged functions implementation. Times in seconds are listed.

	Naive	Merged	Speedup
1000×3000	0.80	0.17	4.85
3000×9000	7.25	2.19	3.31

Memory Management. The structure data type is efficient in building software. However, currently, handling the allocatable array in a structure is cumbersome, namely, users need to allocate arrays on GPU manually, and transfer the data. In the present study, we employed the managed memory (the unified memory in CUDA is equivalent technology), although this is only supported by the PGI compiler currently. Using the managed memory, all the arrays used on GPU are automatically uploaded. Once an array is uploaded onto GPU, no communication between CPU and GPU is necessary except the case that output files are created.

Double GPU. In the implementation point of view, we need to launch an identical function twice, namely once with the Yin grid and also with the Yang grid. In other words, those two grids can be evaluated independently, if the boundary values are exchanged correctly. Thus, using the two GPUs and assigning a grid to a GPU is quite natural. However, at the same time, one needs to implement a way to import and export the boundary values to the other. Thanks to the managed memory technology which we employed in the present study, communication among GPUs and CPU is carried out automatically. On the other hand, as we will discuss later, the communication cost among GPUs with the managed memory technology is high and the total computational speed is worse than the single GPU configuration. Thus, we implemented a communication functionality with a lower level function "`cudaMemcpyPeer`". `cudaMemcpyPeer` enables us to send and receive data directly bypassing the CPU, whilst the data path with the management memory technology is not visible for users. On the other hand, `cudaMemcpyPeer` requires us to manage the memory space of GPU manually. In the present study, we employ the device array and `!$acc deviceptr` directive to use `cudaMemcpyPeer`.

Because of the prevalence of the GPU-cluster (multi-node) type supercomputers, there have been many reports of success in MPI–OpenACC hybridization [7] and even training [3]. Here, MPI stands for Message Passing Interface. Nevertheless, since the DGX–station we employed in the present study as well as DGX–1 and DGX–2 by NVIDIA, which have a similar architecture with DGX–station whilst more powerful, are single-node-multi-GPU computers, discussing the hybridization of OpenMP and OpenACC may draw attention from researchers. One of the biggest advantages of using the OpenMP/OpenACC hybridization is that users can progressively implement their codes based on their sequential version, especially users who can use the managed memory technology. As well known, MPI requires users to reconstruct the data structure which may lead to the major code rewrite.

In the following program, we show the structure of the hybridization with a pseudo code resembling Fortran. In the program, `rungeKutta` is the function which computes the physical values of the next step using the Runge–Kutta algorithm on the Yin or Yang grid. Since the Yin grid and the Yang grid are identical, we call the same function twice but with separate arrays (`yin` and `yang`) in each iteration. In our hybrid program, each grid is assigned on each OpenMP thread, so that two GPUs are used in parallel. The function `acc_set_device_num` is a function provided within the OpenACC framework, and this specifies the device that is used by a thread. The arrays, `yin2yangSend`, `yin2yangRecv`, `yang2yinSend` and `yang2yinRecv` are device arrays which are to exchange the boundary values. Since device arrays should be allocated on target devices, they are allocated after `acc_set_device_num` is called. `calcEdge` is the function which computes the boundary values on the other grid, and stores such values on `yin2yangSend` and `yang2yinSend`. Those arrays are copied

to `yang2yinRecv` and `yin2yangRecv` directly using `cudaMemcpyPeer`. Finally the boundary values are copied to the Yin and Yang grids using the function `recoverBoundary` so that we can move on to the next time step. Since we should avoid the communication between CPU and GPUs, the computation in the functions `rungeKutta`, `calcEdge` and `recoverBoundary` are all written in OpenACC. Thus the hybridization of OpenMP/OpenACC is achieved on our FDM code.

Hybridization of OpenMP/OpenACC on the Yin-Yang grid FDM

```
program acousticFDM

! Structure that stores each grid values
type(yinYangGrid) :: yin, yang
! Device array to exchange boundary values
real(8),dimension(:),allocatable, device :: yin2yangSend, yin2yangRecv
real(8),dimension(:),allocatable, device :: yang2yinSend, yang2yinRecv
! GPU number 1 and 2 are used
integer,parameter, dimension(0:1) :: iDev = (/1,2/)

nPhi = 100      ! Number of grid points along Phi-axis
nTheta = nPhi*3 ! Number of grid points along Theta-axis
integer :: nEdge = 2*nPhi + 2*nTheta !Number of grid points on the all edges

!$omp parallel num_threads(2)
do ITER = 1, MAX_ITERATION ! Time integration loop
   if(ITER==1)then
      call acc_set_device_num(iDev(omp_get_thread_num() ) , &
           acc_device_nvidia)
      if(omp_get_thread_num()==0)then
         allocate(yin2yangSend(nEdge))
         allocate(yang2yinRecv(nEdge))
      else
         allocate(yang2yinSend(nEdge))
         allocate(yin2yangRecv(nEdge))
      endif
    endif

    if(omp_get_thread_num()==0)then
! Compute the Yin grid
       call rungeKutta(yin)
! Compute the values of grid point on the edged in the Yang grid
! and store in yin2yangSend
       call calcEdge(yin,yin2yangSend)
! Copy yin2yangSend to yin2yangRecv which is on the other device
       istat = cudaMemcpyPeer(yin2yangRecv,iDev(1), &
                         yin2yangSend,iDev(0),nEdge)
       call recoverBoundary(yang2yinRecv,yin)
    else
       call rungeKutta(yang)
       call calcEdge(yang,yang2yinSend)
       istat = cudaMemcpyPeer(yang2yinRecv,iDev(0), &
                         yang2yinSend,iDev(1),nEdge)
```

```
      call recoverBoundary(yin2yangRecv,yang)
    endif
enddo
!$omp end parallel
end program acousticFDM
```

4.6 Software Evaluation

Accuracy Evaluation with a Live Experiment. Since this software has been developed from scratch, we first would like to check the validity of our code from the modelling point of view. In the present study, we refer to the experiment performed in 1960 [22]. In this experiment, a chemical explosive (amatol) was used as a source of the hydroacoustic wave, and it was detonated by the research vessel Vema offshore Perth, Australia. A hydrophone was set in the Bermuda area to catch the hydroacoustic wave. In the literature, authors identified the propagation path and the effective sound speed was estimated. In the present study, we employed the parameters given in the literature to perform the modelling, namely,

- sound speed: 1485 m/s
- location of hydrophone: 32.10 N and 64.35 W
- location of detonation: 33.36 S and 113.29 E in the SOFAR channel
- travel time: 3 h 41 min 18 s to 3 h 42 min 24 s.

Since our FDM code does not take into account the radial direction, we assume that the wave propagates at 1,000 m depth, close to where the SOFAR channel is situated. ETOPO1 [6] is employed as the bathymetry data. The solid earth is treated as the rigid body, in other words, the velocity of wave becomes zero at the boundary.

As a result, the wave travels from Vema to Bermuda in 3 h 43 min 0 s, which is slightly longer than the experiment. Although further evaluation and development are necessary, we believe the implementation so far is successful. A visualization animation of the wave propagation of this experiment is uploaded to Zenodo [16].

Computational Speed. In order to evaluate the improvements in computational speed, we measured the computation time per iteration of each implementation with various problem sizes (Table 3. Figures in the table are in seconds). In the present study, we used the following grids: $n\phi \times n\theta = 1000 \times 3000$(10 km/grid on the equator), 3000×9000(3.33 km/grid), 6000×18000(1.67 km/grid), 8000×24000(1.25 km/grid) and 9000×27000 (1.11 km/grid), where $n\phi$ and $n\theta$ are the numbers of grid points along the ϕ-axis and the θ-axis. In the table, "CPU" denotes FDM implementation with one core CPU, "OpenMP" denotes 20 cores parallel with OpenMP, "Multicore" denotes 20 cores parallel with OpenACC with the `-ta=multicore` option,

"Single GPU" denotes one GPU implementation with OpenACC, "managed memory" denotes the dual-GPU implementation with the managed memory technology, and "cudaMemcpyPeer" denotes the dual-GPU implementation with the cudaMemcpyPeer function (please note that single GPU also relies on the managed memory technology). In all the cases, CPU shows the slowest. The multicore parallel implementations (OpenMP and Multicore) are the second slowest. Managed memory is faster than CPU, but slower than Single GPU although two GPUs are involved in. Finally, cudaMemcpyPeer shows the best performance. We observed that Single GPU with the 8000×24000 and 9000×27000 grids took over 5 min for one time step computation. Thus we aborted the computation and "Not Available (NA)" is indicated in the table. Thanks to the managed memory technology, although the required memory size is larger than the actual device memory size, the processes continued working. However, because many communication instructions were issued between CPU and GPU, the performance became worse than CPU.

We listed the acceleration of each implementation to CPU (Table 4). In most cases cudaMemcpyPeer and Single GPU show over 100 times acceleration to one core CPU. In the best case scenario, we obtained approximated 160 times acceleration with two GPUs. In this case, cudaMemcpyPeer shows 1.4 better performance than Single GPU, although it is still lower than the ideal acceleration. We observed that the ideal acceleration can be obtained by ignoring `cudaMemcpyPeer` and `recoverBoundary` in our preliminary test. This implies that there is a space for further optimization in the boundary exchange phase. Finally, managed memory shows worse performance than Single GPU while the problem size is sufficiently small. At the same time, managed memory allows us to solve larger problem than Single GPU, and it runs faster than CPU. The multicore implementations show approximately 10 times acceleration to CPU, whilst Multicore is slightly faster than OpenMP.

For readers' evaluation, we provide the CPU version [14], and the `cudaMemcpyPeer` version [13] on Zenodo.

Table 3. Computational times at one time step of each implementation with various problem sizes. Figures are in second. NA denotes "Not Available"

	CPU	OpenMP	Multicore	Single GPU	Managed memory	cuda Memcpy Peer
1000×3000	6.10E–01	7.28E–02	6.76E–02	7.65E–03	1.07E–01	1.42E–02
3000×9000	6.76E+00	7.40E–01	6.51E–01	5.91E–02	6.48E–01	5.26E–02
6000×18000	2.72E+01	3.00E+00	2.69E+00	2.45E–01	2.02E+00	1.71E–01
8000×24000	4.95E+01	5.52E+00	5.12E+00	NA	8.93E+00	3.87E–01
9000×27000	6.04E+01	7.12E+00	6.73E+00	NA	9.44E+00	4.43E–01

Table 4. Acceleration of each implementation to CPU with various problem sizes. NA denotes "Not Available"

	CPU	OpenMP	Multicore	Single GPU	Managed memory	cudaMemcpyPeer
1000 × 3000	x1.00	x8.38	x9.03	x79.82	x5.71	x43.00
3000 × 9000	x1.00	x9.13	x10.39	x114.32	x10.43	x128.36
6000 × 18000	x1.00	x9.08	x10.12	x111.30	x13.50	x158.71
8000 × 24000	x1.00	x8.97	x9.67	NA	x5.54	x127.76
9000 × 27000	x1.00	x8.49	x8.99	NA	x6.40	x136.35

5 Conclusion

In the present study, we have implemented two hydroacoustic modelling codes on GPU with OpenACC and gained better performance than on CPU. Since those two modelling codes will be used to understand the observed signals in the international monitoring system of CTBTO, the larger the number of hypothetical events we can solve, the better we can understand the observed signals. In this context, the acceleration obtained in this study contributes to the mission of the organization. The summaries of the achievements are: (1) In 3D-SSFPE, we succeeded in implementing GPU enabled code which works together with Octave, which is a high-level computer language. As a result, we gained approximately 19 times acceleration to the original Octave code, in the best case scenario. Although the obtained acceleration is lower than that can be observed in our sample codes (50 times acceleration), which are relatively simpler than 3D-SSFPE, we are proud of achieving high performance in a realistic problem. In addition, we may gain further acceleration with updates on compilers. (2) In the in-house FDM code, we succeeded in implementing an OpenMP/OpenACC hybrid code to use two GPUs. As a result, we gained approximately 160 times speedup to one core CPU in the best case scenario. Although there have been many research projects which successfully implemented OpenACC codes on supercomputers, our experience might be of interest to researchers, especially those who are not familiar with supercomputing.

Disclaimer. The views expressed on this article are those of the authors' and do not necessarily reflect the view of the CTBTO.

Acknowledgement. One of the authors, Noriyuki Kushida, would like to express his gratitude to Dr Tammy Taylor, the director of the International Data Centre of CTBTO, on her encouragement on the work. And also he would like to express his gratitude to CEA in France as well as PRACE, on their support for the development of FDM by awarding the machine times on Irena Skylake. Finally, he would like to thank Dr Yuka Kushida for her English correction. She has pointed out errors which had been overlooked even by a native speaker.

References

1. DGX Station web page. https://www.nvidia.com/en-us/data-center/dgx-station/. Accessed 12 July 2019
2. Interoperability between OpenACC and cuFFT. https://www.olcf.ornl.gov/tutorials/mixing-openacc-with-gpu-libraries/. Accessed 16 July 2019
3. MULTI GPU PROGRAMMING WITH MPI AND OPENACC. https://gputechconf2017.smarteventscloud.com/connect/sessionDetail.ww?SESSION_ID=110507&tclass=popup. Accessed 22 July 2019
4. The API reference guide for cuFFT. https://docs.nvidia.com/cuda/cufft/index.html. Accessed 13 July 2019
5. ubuntu-toolchain-r/test web page. https://launchpad.net/~ubuntu-toolchain-r/+archive/ubuntu/test. Accessed 13 July 2019
6. Amante, C.: Etopo1 1 arc-minute global relief model: procedures, data sources and analysis (2009). https://doi.org/10.7289/v5c8276m, https://data.nodc.noaa.gov/cgi-bin/iso?id=gov.noaa.ngdc.mgg.dem:316
7. Calore, E., Gabbana, A., Kraus, J., Schifano, S.F., Tripiccione, R.: Performance and portability of accelerated lattice Boltzmann applications with OpenACC. Concurr. Comput.: Pract. Exper. **28**(12), 3485–3502 (2016). https://doi.org/10.1002/cpe.3862
8. Etter, P.: Underwater Acoustic Modeling and Simulation, 4th edn. Taylor & Francis (2013)
9. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**(2), 216–231 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation"
10. Heaney, K.D., Campbell, R.L.: Three-dimensional parabolic equation modeling of mesoscale eddy deflection. J. Acoust. Soc. Am. **139**(2), 918–926 (2016). https://doi.org/10.1121/1.4942112
11. Heaney, K.D., Prior, M., Campbell, R.L.: Bathymetric diffraction of basin-scale hydroacoustic signals. J. Acoust. Soc. Am. **141**(2), 878–885 (2017). https://doi.org/10.1121/1.4976052
12. Kageyama, A., Sato, T.: "Yin-Yang grid": an overset grid in spherical geometry. Geochem. Geophys. Geosyst. **5**(9) (2004). https://doi.org/10.1029/2004GC000734
13. Kushida, N.: Globalacoustic2D dual GPU (2019). https://doi.org/10.5281/zenodo.3351369
14. Kushida, N.: Globalacoustic2D OpenMP (2019). https://doi.org/10.5281/zenodo.3351284
15. Kushida, N.: GPU version of "Split Step Fourier PE method to solve the Lloyd's Mirror Problem", August 2019. https://doi.org/10.5281/zenodo.3359888
16. Kushida, N.: Hydroacoustic wave propagation from Vema to Bermuda using FDM, July 2019. https://doi.org/10.5281/zenodo.3349551
17. Kushida, N.: OpenACC enabled oct file (2019). https://doi.org/10.5281/zenodo.3345905
18. Kushida, N., Le Bras, R.: Acoustic wave simulation using an overset grid for the global monitoring system. In: AGU Fall Meeting. Oral Presentation: AGU Fall Meeting 2017, New Orleans, USA, 11–15 December 2017 (2017)
19. Lin, Y.T.: Split Step Fourier PE method to solve the Lloyd's Mirror Problem, August 2019. https://doi.org/10.5281/zenodo.3359581

20. Lin, Y.T., Duda, T.F., Newhall, A.E.: Three-dimensional sound propagation models using the parabolic-equation approximation and the split-step Fourier method. J. Comput. Acoust. **21**(01), 1250018 (2013). https://doi.org/10.1142/s0218396x1250018x
21. Ostashev, V.E., Wilson, D.K., Liu, L., Aldridge, D.F., Symons, N.P., Marlin, D.: Equations for finite-difference, time-domain simulation of sound propagation in moving inhomogeneous media and numerical implementation. J. Acoust. Soc. Am. **117**(2), 503–517 (2005). https://doi.org/10.1121/1.1841531
22. Shockley, R.C., Northrop, J., Hansen, P.G., Hartdegen, C.: Sofar propagation paths from Australia to Bermuda: comparison of signal speed algorithms and experiments. J. Acoust, Soc. Am. **71**(1), 51–60 (1982). https://doi.org/10.1121/1.387250
23. Wang, L.S., Heaney, K., Pangerc, T., Theobald, P., Robinson, S.P., Ainslie, M.: Review of underwater acoustic propagation models. NPL report, October 2014. http://eprintspublications.npl.co.uk/6340/