Dear editor and reviewer,

We thank you for your highly valuable comments. It is our great honor to receive such a careful edit. We have carefully incorporated the referee's comments and suggestions. The point-to-point responses are as follows.

Responses to the comments of Prof. David Webb:

(1) "The standard of English is poor."

[Response]:
Thanks to Prof. Webb for carefully editing our manuscript. Following his suggestions, we have revised the whole paper thoroughly. Almost every sentence of the manuscript has been revised and we learned a lot by following his edit.

In addition, the paper is professionally proof-read for grammatical errors and poor sentence structures. We believe the current version is substantially improved and hope it is suitable for publication.

(2) "There is a lot of repetition and redundant text."

[Response]:
Agreed and now corrected. We removed redundant sentences and reorganized most paragraphs for a more concise presentation. For example, in our previous draft, we repetitively described the optimization of chasing method in Section 4.1(B) and last paragraph of Section 4.1. Following Prof. Webb's suggestions, we now split Section 4.1 into two parts. The first part describes the standard computer improvements and the second describes the optimizations special to the GPU, including the local memory blocking optimization of the chasing method.

(3) The technical description is slanted too much towards computer specialists - who should not need to be told anyway.

[Response]:
We agree and now have shortened the implementation details of communication optimizations (Section 4.2) and I/O optimizations (Section 4.3), as well as the verification of accuracy, which are too much towards the computer specialists. In addition, we have placed the related projects in a table to clearly show their speedups in Section 1 and have added a diagram illustrating the GPU memory architecture in Section 3. With the audience in mind, we further revised large numbers of sentences describing the GPU architecture and the POM.gpu, and tried to make every computer term clear and simple.

We really appreciate your great effort and highly valuable comments. We hope our revised manuscript will be more suitable for the publication of GMD.

Best wishes,

Xiaomeng Huang

Manuscript prepared for Geosci. Model Dev.
with version 2015/04/24 7.83 Copernicus papers of the LaTeX class copernicus.cls.
Date: 10 August 2015

~~gpuPOM1~~POM.gpu-v1.0: a GPU-based Princeton Ocean Model

Shizhen Xu[1], Xiaomeng Huang[1], Lie-Yauw Oey[2,3], Fanghua Xu[1], Haohuan Fu[1], Yan Zhang[1], and Guangwen Yang[1]

[1] Ministry of Education Key Laboratory for Earth System Modeling, Center for Earth System Science, Tsinghua University, 100084, and Joint Center for Global Change Studies, Beijing, 100875, China.
[2] Institute of Hydrological & Oceanic Sciences, National Central University, Jhongli, Taiwan.
[3] Program in Atmospheric & Oceanic Sciences, Princeton University, Princeton, New Jersey, USA.

Correspondence to: Xiaomeng Huang
(hxm@tsinghua.edu.cn)

Abstract. ~~Rapid advances in the performance of the graphics processing unit (GPU) have made the GPU a compelling solution for a series of scientific applications~~ Graphics Processing Units (GPUs) is an attractive solution in many scientific applications due to its high performance. However, most existing GPU ~~acceleration works for climate models are doing partial code porting for certain hot spots, and can only achieve limited speedup for the entire model. In this work, we take~~ conversions of climate models use GPUs for only a few computationally intensive regions. In the present study, we redesign the mpiPOM (a parallel version of the Princeton Ocean Model) ~~as our starting point, design and implement a GPU-based Princeton Ocean Model. By carefully considering the architectural features of the state-of-the-art GPU devices, we rewrite the full mpiPOM model from the original Fortran version into~~ with GPUs. Specifically, we first convert the model from its original Fortran form to a new Compute Unified Device Architecture C (CUDA-C) ~~version. We take several accelerating methods to further improve the performance of gpuPOM, including optimizing memory access in a single GPU, overlapping communication and boundary operations among multiple~~ code, then we optimise the code on each of the GPUs, the communications between the GPUs, and ~~overlapping input/output (~~the I/O ~~) between the hybrid Central Processing Unit (CPU)and the GPU. Our experimental results indicate~~ between the GPUs and the Central Processing Units (CPUs). We show that the performance of the ~~gpuPOM~~ new model on a workstation containing 4 GPUs is comparable to that on a powerful cluster with 408 ~~CPU cores~~ standard CPU cores, and it reduces the energy consumption by a factor of 6.8~~times~~.

1 Introduction

High-resolution atmospheric, oceanic and ~~/or climate modeling remains a~~ climate modelling remain significant scientific and engineering ~~challenge~~ challenges because of the enormous computing, com-

Table 1. Existing GPU porting work in climate fields. The speedups are normalized to one CPU core.

Model Name	Model Description	Porting Modules to GPU	Speedup
WRF	Weather research and forecasting	WSM5 microphysics	8
WRF-Chem	WRF chemical	Chemical kinetics kernel	8.5
POP	Parallel ocean program	Loop structures	2.2
COSMO	Consortium for small-scale modelling	Dynamical core	22.7
NIM	Non-hydrostatic icosahedral model	Dynamical core	34
ASUCA	Non-hydrostatic weather model	Dynamical core & physical	80

munication, and storage requirements ~~. With~~ involved. Due to the rapid development of computer architecture, in particular the development of multi-core and many-core ~~techniques~~hardware, the computing power that can be applied to scientific problems has increased exponentially in recent decades. ~~Some parallel computing techniques~~Parallel computing methods, such as the Message Passing Interface (MPI, Gropp et al. (1999)) and Open Multi-Processing (OpenMP, Chapman et al. (2008)) have been widely used to support the parallelization of ~~numerous~~ climate models. ~~Moreover, as modern massive supercomputers become more and more heterogeneous because of the increasing number of different accelerating~~ However, supercomputers are becoming increasingly heterogeneous involving devices such as the GPU ~~, the Intel many integrated core~~ and the Intel Many Integrated Core (Intel MIC)~~and reconfigurable computing based on field programmable gate array (FPGA),~~ , and new approaches are required to ~~more~~ effectively utilize the ~~emerging novel architecture, communication and input/output (I/O) to achieve order-of-magnitude acceleration required for climate models.~~ new hardware.

In recent years, a number of scientific codes have been ported to the GPU ~~. Different levels of speedup were achieved~~ as shown in Table 1. Most existing GPU acceleration codes for climate models ~~using GPUs. Michalakes and Vachharajani (2008) accelerated a computationally intensive microphysics process of the~~ are only operating on certain hot spots of the program, leaving a significant portion of the program still running on CPUs. The speed of some subroutines reported in the Weather Research and Forecast (WRF) ~~model with a speedup of nearly 25x; but the entire WRF model is sped up by~~ (Michalakes and Vachharajani, 2008) and WRF-Chem(Linford et al., 2009) is improved by a factor of approximately 8 whereas the whole model achieves limited speedup because of partial porting. The speed of POP (Zhenya et al., 2010) is improved by a factor of only ~~1.23x. Shimokawabe et al. (2010) fully accelerated the ASUCA model – a non-hydrostatic weather model – on 528 Nvidia Tesla GT200 GPUs and achieved a speedup of 80x. Linford et al. (2009) accelerated a computationally intensive chemical kinetics kernel from the WRF model with Chemistry on an Nvidia Tesla C1060 and achieved a speedup of 8x. Leutwyler et al. (2014) accelerated a full huge operational weather forecasting model COSMO and achieved 2.8X speedup for its dynamic core. Carpenter et al. (2013) accelerated the spectral element dynamical core of the Community Earth~~

~~System Model (CESM) using the GPU by 3x. Govett et al. (2010) ported the dynamics portion of the Non-hydrostatic Icosahedral (NIM) model to the GPU and achieved a speedup of 34x.Zhenya et al. (2010) adopted~~ 2.2 because the model only accelerated a number of loop structures using the OpenACC Application Programming Interface (OpenACC API)~~, which used simple compiler directives to accelerate some hot-spot functions, to accelerate the parallel ocean program (POP) by 2.2x.~~

~~Most existing GPU acceleration projects for climate models are only working on certain hot spots of the program, leaving a significant part of the program still running on CPUs. Therefore, there are usually frequent data exchange between CPUs and GPUs , which significantly reduces the overall performance~~. The speed of COSMO(Leutwyler et al., 2014) , NIM(Govett et al., 2010) and ASUCA(Shimokawabe et al., 2010) are greatly improved by multiple GPUs. We believe that the elaborate optimization of the memory access of each GPU and the communication between GPUs can further accelerate these models.

The objective of our study ~~is~~ was to shorten the ~~high~~ computation time of ~~high-resolution ocean models by parallelizing their~~ the Princeton Ocean Model (POM) by parallelizing its existing model structures using the GPU. Taking the parallel version of the Princeton Ocean Model (mpiPOM)~~as an example~~, we demonstrate how to ~~parallelize~~ code an ocean model ~~to make it run efficiently on a~~ so that it runs efficiently on GPU architecture. ~~Using the state-of-the-art GPU architecture, we~~ We first convert the mpiPOM from its original Fortran version into a new Compute Unified Device Architecture C (CUDA-C) version, POM.gpu-v1.0. CUDA-C is the dominant programming language for GPUs. We ~~call the new version gpuPOM1.0. Then, we design and implement several optimizing methods: (i) computation optimization in a single GPU; (ii) communication optimization among multiple GPUs, and (iii) I/O optimization between a hybrid GPU and CPU.~~

~~In terms of computing, we concentrate on memory access optimization and making better use of caches in GPU memory hierarchy. We improve memory usage by using read-only data cache, local memory blocking, loop fusion, function fusion and that disables error-correcting code memory (Error Checking Correction, ECC memory). The experimental results demonstrate that high memory access optimization can achieve a speedup of approximately 100x when comparing a single GPU against a single CPU core.~~

~~In terms of communication, we concentrate on the overlapping between the inner-region computation and the outer-region communication and update. With the GPUDirect communication technology, multiple GPUsin one node can communicate directly and bypass the CPU. In addition, with the fine-grained control of the CUDA streams and its priority, inner-region computation can be executed concurrently with outer-region communication and updating.~~

~~In terms of I/O, we choose to split the MPI communicator into computation and~~ then optimise the code on each of the GPUs, the communications between the GPUs, and the I/O ~~processes. One individual computation process and one individual I/O process are attached to one GPU. The computation process is responsible for launching kernels on the GPU and the I/O process is responsible~~

~~for data copy back from the GPU and to write on disk. The computing process and the I/O process execute concurrently.~~between GPUs and the CPUs to further improve the performance of POM.gpu.

To understand the accuracy, performance and scalability of the ~~gpuPOM, we build a customized~~ POM.gpu code, we customized a workstation with four ~~GPU~~ Nvidia K20X ~~devices inside. The experimental~~ GPUs. The results show that the performance of ~~the gpuPOM~~ POM.gpu running on this workstation is comparable to that on a powerful cluster with 408 standard CPU cores.

~~The~~ This paper is organized as follows. In Section 2, we review the mpiPOM model. In Section 3, we briefly introduce the GPU computing model. In Section 4, we present ~~detailed techniques about computation optimization in a single GPU, communication optimization among multiple GPUs, and I/O optimization between a hybrid GPU and CPU. We provide the corresponding experimental results about~~ the detailed optimization techniques. In Section 5, we report on the correctness, performance and scalability ~~in Section 5~~ of the model. We present the code availability in Section 6 and conclude our work in Section 7.

2 The mpiPOM

The mpiPOM is a parallel version of the ~~Princeton Ocean Model (POM) that is based on MPI~~POM. It retains most of the physics ~~package~~ of the original POM (Blumberg and Mellor, 1983, 1987; Oey et al., 1985a, b, c; Oey and Chen, 1992a, b), ~~but includes also~~ and includes satellite and drifter assimilation schemes from the Princeton Regional Ocean Forecast System (Oey, 2005; Lin et al., 2006; Yin and Oey, 2007), ~~as well as more recently advanced features such as wind-wave induced Stokes drift ,~~ stokes drift and wave-enhanced mixing~~and Localized Ensemble Transform Kalman Filter (Oey et al., 2013; Xu et al., 2013)~~ (Oey et al., 2013; Xu et al., 2013; Xu and Oey, 2014) . The POM code was reorganized and ~~MPI~~ the parallel MPI version was implemented by Jordi and Wang (2012) using a two-dimensional data decomposition of the horizontal domain~~with a halo of ghost cells~~. The MPI is a standard library for message passing and it is widely used to develop parallel programs.The POM is a powerful ocean model that has been used in a wide range of applications: circulation and mixing processes in rivers, estuaries, ~~shelf and slope,~~ shelves, slopes, lakes, semi-enclosed seas and open and global oceans. It is also at the core of various real-time ocean and hurricane forecasting systems, ~~for examples: Japan~~ e.g., the Japanese coastal ocean and Kuroshio ~~(Isobe et al., 2012) ; Adiratic~~ current (Miyazawa et al., 2009; Isobe et al., 2012; Varlamov et al., 2015) ; the Adriatic Sea Forecasting System (Zavatarelli and Pinardi, 2003); the Mediterranean Sea forecasting system (Korres et al., 2007); the GFDL Hurricane Prediction system (Kurihara et al., 1995, 1998), the US' Hurricane Forecasting System (Gopalakrishnan et al., 2010, 2011) and the Advanced Taiwan Ocean Prediction system (Oey et al., 2013). Additionally, the model has been used to study various geophysical fluid dynamical processes (e.g.~~Allen and Newberger, 1996; Newberger and Allen, 2007a, b; Kagimoto and Yamagata, 1997; Guo et~~

Allen and Newberger, 1996; Newberger and Allen, 2007a, b; Kagimoto and Yamagata, 1997; Guo et al., 2006; Oey et al., 2003; Za
For a more complete list, please visit the POM website (http://www.ccpo.odu.edu/POMWEB).

The mpiPOM experiment ~~that is~~ used in this paper is one of ~~the two~~ two that were designed and tested by Professor Oey and students; the codes and results are freely available at the FTP site (ftp://profs.princeton.edu/leo/mpipom/atop/tests/). The reader can ~~see~~ refer to Chapter 3 of the ~~Lecture Notes~~ lecture notes (Oey, 2014) for more detail. The test case is a dam-break problem in which warm and cold waters are initially separated in the middle of a zonally periodic channel ~~$200km \times 50km \times 50m$~~ (200km × 50km × 50m) on an f-plane, with walls at the northern and southern boundaries. Geostrophic adjustment then ensues and baroclinic instability waves amplify and develop into finite-amplitude eddies in 10∼20 days. The horizontal grid sizes are 1 km and there are 50 vertical sigma levels. Although the problem is a test case, the code is the full mpiPOM version ~~that is~~ used in the ATOP forecasting system.

The model solves the primitive equation under hydrostatic and ~~boussinesq~~ Boussinesq approximations. In the horizontal, spatial derivatives are computed either using ~~centered-space~~ centred-space differencing or Smolarkiewicz's positive definite advection transport algorithm (Smolarkiewicz, 1984) on a staggered Arakawa C-grid; both schemes have been tested, but the latter is reported here. In the vertical, the mpiPOM supports terrain-following sigma coordinates and a fourth-order scheme option to reduce the internal pressure-gradient errors (Berntsen and Oey, 2010). The mpiPOM uses the time-splitting technique to separate the vertically integrated equations (external mode) from the vertical structure equations (internal mode). The external mode calculation is responsible for updating the surface elevation and ~~the~~ vertically averaged velocities. The internal mode calculation ~~results in updates for~~ updates the velocity, temperature and salinity, as well as the turbulence quantities. The three-dimensional internal mode and the two-dimensional external mode are both integrated explicitly using a second-order leapfrog scheme. These two modules are the most computationally intensive kernels of the mpiPOM model.

3 ~~GPU programming model overview~~

~~In this section, we describe the basic GPU architecture in a programmer's perspective and focus on how to harness the power of the GPU with NVIDIA's Compute Unified Device Architecture(CUDA), a programming model and computing platform that makes GPU program elegant and simple~~The main computational problem of the mpiPOM is memory bandwidth limited. To confirm this issue, we use the runtime performance API tool to estimate the floating point operation count and the memory access instruction count, as in Browne et al. (2000) . The results reveal that the computational intensity, defined as floating point operations per byte transferred to or from memory, of the mpiPOM is approximately 1:3.3, whereas the computational intensity provided by a modern high-performance CPU (an Intel SandyBridge E5-2670) is 7.5:1. Many large arrays are mostly pulled from main memory and there is poor data reuse in the mpiPOM. In addition, there are no obvious hot spot functions in the mpiPOM, and even the most time-consuming subroutine occupies only 20% of the total execution time. Therefore, porting a handful of subroutines to the GPU is not helpful in improving the model efficiency. This explains why we must port the entire program from the CPU to the GPU.

~~In the GPU hardware design, there are numerous~~

3 GPU computing model overview

Modern GPUs employ a stream-processing model with parallelism. Each GPU contains a number of stream multiprocessors (SMs)~~grouped by large numbers of CUDA cores. As an example, the~~. In this work, we carried out the conversion using four Nvidia's K20X ~~GPU we used has~~ GPUs. Each K20X GPU contains 14 SMs and each SM has 192 ~~CUDA cores for single precision operation. One K20X GPU can achieve 3.93TFlops~~ single precision processors and 64 additional processors for double precision. Although the computational capability of each processor is low, one GPU with thousands of processors can greatly boost the performance compared to the CPU. In computing, FLOPS (FLoating-point Operations Per Second) is a measure of computer performance. The theoretical peak performance ~~with single-precision floating point and 250GB~~of each K20X GPU is 3.93 teraFLOPS (TFLOPS, one trillion floating-point operations per second) for the single precision floating-point calculations. In contrast, a single Intel SandyBridge E5-2670 CPU is only capable of 0.384 TFLOPS.

Each pair of GPUs shares 6 Gigabytes (GB) of memory, with the interface having a potential bandwidth of 250 GB/s~~memory bandwidth~~. Figure 3 illustrates the memory hierarchy of the K20X GPU. Each SM ~~has its own execution units (CUDA cores, load/store units, special function units), warp-schedulers, and various~~ possesses some types of fast on-chip ~~faster memories such as registers~~memory such as register, L1 cache~~/~~, shared memory and ~~texture cache. Various on-chip caches provides more opportunities to implement memory optimizations on GPU platform. Each SM owns 64K 32 bit registers which are the fastest memory in the GPU memory hierarchy~~read-only data cache. In GPUs, the register is the fastest memory, of which the size is 256 Kilobytes (KB) for each SM. The shared memory and the L1 cache ~~share a 64KB on-chip fast memory and can be configured with artificial options such~~ use the common 64 KB space which can be partitioned as 16/~~48KB~~48 KB, 32/~~32KB~~ 32 KB or 48/16 KB. ~~In addition, there are~~ The 48 KB read-only data cache ~~which add the feature for read-only data in global memory to be loaded through the same cache.~~ is useful for holding frequently used values that remain unchanged during each stage of the processing.

There are three ~~common methods to port a program from CPU to GPU~~widely used methods for porting a program to GPUs. The first method uses drop-in libraries provided by CUDA to replace the existing code, ~~such as the work implemented by~~ as in Siewertsen et al. (2012). The second method

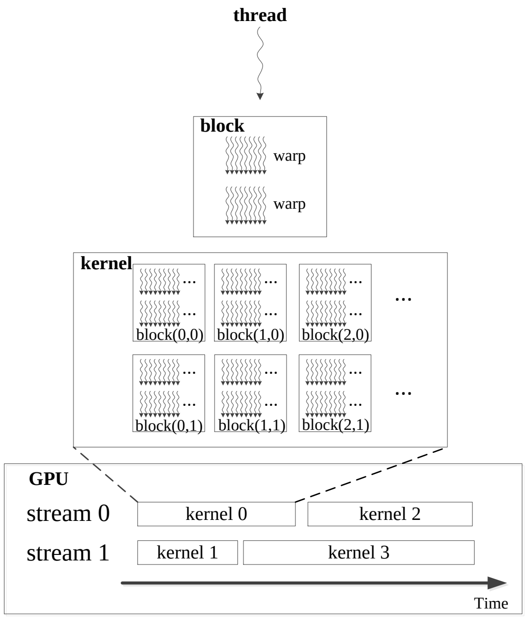


Figure 1. The hierarchy of stream, kernel, block, warp and thread.

uses ~~simple~~ OpenACC directive as hints in the original CPU code ~~, such as the work implemented by~~ as in Zhenya et al. (2010). The last method ~~,~~ is the most complex but also the most effective~~,~~ ~~rewrites the whole program with~~ ; it involves rewriting the entire program using low level CUDA subroutines.

~~In CUDA , a **kernel** is a~~

In CUDA terminology, a kernel is a single section of code or subroutine running on the GPU. ~~Each kernel launch consists of a large number of threads and these~~ The underlying code in a kernel is split into a series of threads each of which deals with different data. These threads are grouped into equal size ~~blocks which~~ thread blocks that can be executed independently. ~~Each~~ A thread block is further divided into warps ~~, which consist~~ as basic scheduled units. A warp consists of 32 consecutive threads ~~. Threads in a warp~~ that execute the same instruction simultaneously~~and can be scheduled as a whole unit. Kernel function~~ . Each kernel and data transfer ~~commands~~ command in CUDA has an optional parameter ~~"stream ID". If "stream ID" is declared~~“stream ID”. If the “stream ID” is set in code, commands belonging to different streams can be executed concurrently. ~~It is usually used to alleviate the kernel launch overhead of subsequent independent kernels~~A stream in CUDA is a sequence of commands executed in order. Different streams can execute concurrently with different priorities. Figure 1 illustrates the hierarchy of these terms.

At present, ~~there are two CUDA platforms to support~~ CUDA compilers are available for C and Fortran~~respectively, which are CUDA-C and CUDA-Fortran~~. Although CUDA-Fortran ~~compiler~~

has been available since 2009 and ~~that can bring about less modification to~~ would involve less modification of the mpiPOM code, we ~~still choose~~ chose CUDA-C ~~at the gpuPOM1~~to convert the POM.gpu-v1.0 because of the following reasons: 1) CUDA-C is free of charge~~while CUDA-Fortran for one workstation costs more than $1000.~~ ; 2) ~~Previous work (Henderson et al., 2011) show that , during the porting of Nondydrostatic Icosahedral Model(NIM), the commercial~~ previous work (Henderson et al., 2011) has shown that the CUDA-Fortran compiler ~~does~~ did not perform as well as the ~~manually converted~~ CUDA-C version ~~in some kernels.~~ for some of the kernels during the porting of NIM; 3) ~~The~~ the read-only data cache ~~utilization~~ is not supported ~~in~~ by CUDA-Fortran, which is the key optimization of Section ~~4.1(A).~~ 4.1.2; and 4) ~~We have already had a lot of previous experiences for deep optimizations~~ we have many previous optimisation experiences with CUDA-C.

4 Full GPU acceleration of the mpiPOM

Figure 2 ~~illustrates the flowchart of the gpuPOM.~~is a flowchart illustrating the structure of POM.gpu. The main difference between ~~mpiPOM and gpuPOM~~ the mpiPOM and the POM.gpu is that the CPU in ~~gpuPOM~~ the POM.gpu is only responsible for the initializing ~~work and the outputting~~ and the output work. The ~~gpuPOM begins with~~ POM.gpu begins by initializing the relevant arrays on the CPU ~~host~~ and then copies data from ~~the CPU host~~ CPU to the GPU. The GPU ~~does all the computations, including the external mode, the internal mode, and their interactions. In the 2D external mode loop, the depth-averaged velocity UA, VA and sea surface height are calculated. In the 3D internal model loop, the fields such as velocities (U,V), temperature (T), salinity (S), and various turbulence variables are time-stepped forward.~~ then performs all of the model computations. Outputs such as velocity and sea surface height, are copied back to the CPU ~~host and~~ and are then written to the disk at a user-specified time interval.

In ~~our implementation, the 3D arrays of variables are stored sequentially in the order of x, y, z and the 2D arrays are stored in the order of x, y, which is the same as the original code.The vertical diffusion is solved by the tridiagonal solver (the Thomas Algorithm) which is calculated sequentially in the z direction. For simplicity, the grid is divided along x and y directions (2D block decomposition) in all kernel functions. Each GPU thread specifies a (x,y) point in the horizontal direction and performs all the calculations from surface to bottom. The thread blocks are divided as (32,4). In the x direction, the block number should be a multiple of 32 threads to perform coalesced memory access within a warp. In the y direction, we tested many thread numbers, such as 4 and 8, and obtained similar performances. We finally choose 4 because we attempt to obtain more blocks to distribute the workload among stream multiprocessors (SM) more uniformly, and also to obtain enough occupancy (Volkov, 2010) . Occupancy is the percentage of threads active per multiprocessor~~the following sections, we introduce the optimizations of the POM.gpu by computation, communication and I/O aspects individually.

For the individual GPUs, we concentrate on memory access optimization by making better use of caches in the GPU memory hierarchy. This involves using read-only data cache, local memory blocking, loop fusion, function fusion, and disabling error-correcting code memory. The test results demonstrate that a single GPU can run the model almost one hundred times faster than a single CPU core.

In ~~the following sections, we introduce the general optimizations of the gpuPOM in a single GPU and the special optimizations of the gpuPOM following the state-of-the-art GPU architecture. Then, we present the design of communications for various processes and multiple GPUs within a node. Finally, we describe the design of~~ terms of communication, we overlapped the sending of boundary data between the GPUs with the main computation. Data is also sent directly between the GPUs, bypassing the CPU.

In terms of I/O, we launched extra MPI processes on the main CPU to output the data. These MPI processes are divided into two categories, the computation processes and the I/O processes. The computation processes are responsible for launching kernels into GPUs and the I/O ~~overlapping for hybrid CPU and GPU architecture.~~ processes are responsible for copying data back from the GPUs and for writing to disks. The computation processes and the I/O processes can execute simultaneously to save output time.

4.1 Computational optimizations in a single GPU

Managing the significant performance difference between ~~off-chip~~ global memory and on-chip fast memory is the primary concern ~~of a GPU programmer.~~ for GPU computing. The ratio of bandwidth between global memory and shared memory is approximately 1:10. Therefore, data reuse in on-chip cache always needs to be seriously considered. As shown on the right side of Fig. 3, we propose ~~five key optimizations to fully utilize the faster on-chip memory of~~ two classes of optimization, including the standard optimization of fusion and the ~~GPU and describe the relationships between the GPUmemory hierarchy and each~~ special optimization of the GPU, to better utilize the fast registers and caches.

4.1.1 Standard optimizations of fusion

Fusion optimization in the ~~following.~~POM.gpu code includes loop fusion and function fusion. The loop fusion merges several loops into one loop and the function fusion merges several subroutines into one subroutine.

~~**(A) Read-only data cache utilization**. Effective use of the new 48KB directly-access and~~ Loop fusion is an effective method to store scalar variables in registers for data reuse. As shown in Fig. 4, if the variable *drhox(k, j, i)* is read several times in multiple loops, we can fuse these loops into one. Therefore, *drhox(k, j, i)* will first be read from the global memory and then repeatedly read from a register. For instance, for the *profq* kernel optimized with loop fusion, the device memory

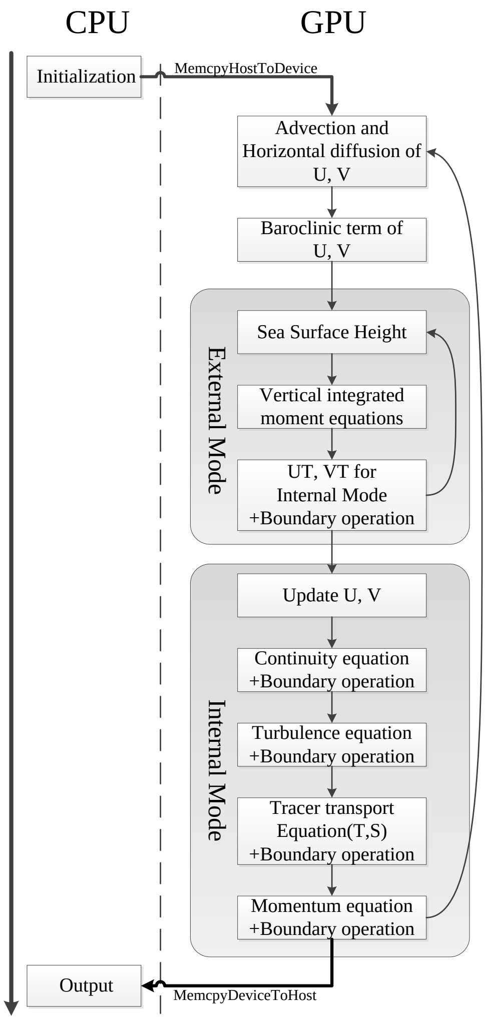

CPU
GPU
Initialization
MemcpyHostToDevice
Advection and Horizontal diffusion of U, V
Baroclinic term of U, V
External Mode
Sea Surface Height
Vertical integrated moment equations
UT, VT for Internal Mode +Boundary operation
Internal Mode
Update U, V
Continuity equation +Boundary operation
Turbulence equation +Boundary operation
Tracer transport Equation(T,S) +Boundary operation
Momentum equation +Boundary operation
Output
MemcpyDeviceToHost

Figure 2. ~~gpuPOM~~ POM.gpu flowchart

transactions decrease by 57%, and the running speed of this kernel is improved by 28.6%. The loop fusion optimization can also be applied in a number of mpiPOM subroutines.

Similar to loop fusion, we can also merge functions in which the same arrays are accessed. For example, the $advv$ and $advu$ functions in the mpiPOM code are used to calculate the advection terms in horizontal directions, respectively. After merging them into one subroutine, the redundant memory access is avoided. The function fusion can also be applied in which one function is called several times to calculate different tracers. The $proft$ function in the mpiPOM code is called twice – one for temperature and one for salinity. Their computing formulas are similar and some common

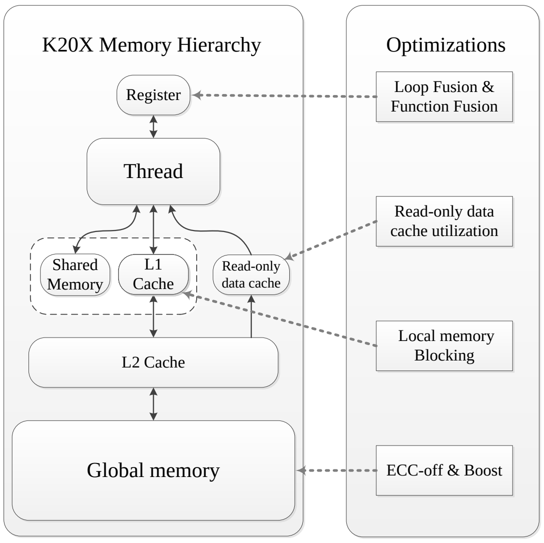


Figure 3. The memory hierarchy of the K20X GPU and the relationships with each ~~optimizations~~optimization

```
/**************************
 *There exist two loops.
 *drhox is visited twice in these loops.
 **************************/
for (k = 1; k < nz-1; k++){
   drhox[k][j][i] = drhox[k-1][j][i] + A[k][j][i];
}

for (k = 0; k< nz-1; k++){
   drhox[k][j][i] = drhox[k][j][i] * B[k][j][i];
}
```

(a) Original CUDA-C code

```
/**************************
 *These loops can be fused into one
 *to reduce global memory access.
 **************************/
for (k = 1; k < nz-1; k++){
   drhox[k][j][i] = drhox[k-1][j][i] + A[k][j][i];
   drhox[k-1][j][i] = drhox[k-1][j][i] * B[k-1][j][i];
}

drhox[k-1][j][i] = drhox[k-1][j][i] * B[k-1][j][i];
```

(b) Optimized CUDA-C code

Figure 4. A simple example of loop fusion.

arrays are accessed. After function fusion, the running speed of the $proft$ kernel is improved by 28.8%.

4.1.2 Special optimizations of the GPU

Our special optimizations mainly focus on the improved utilization of the read-only data cache ~~in the K20X GPUcan improve the performance of memory intensive kernels. This feature will be automatically enabled and utilized~~ and the L1 cache on the GPU. It is useful to alleviate the bottleneck of memory bandwidth that is limited by using these fast on-chip caches.

There is a 48KB read-only data cache in the K20X GPU. We can automatically use this as long as ~~certain conditions are met. We~~ the read-only condition is met. In the POM.gpu, we simply add “const __restrict__~~" qualifiers to~~” qualifiers into the parameter pointers ~~in gpuPOM to explicitly allocate the read-only data cache for our program. The “LDG.E” instruction will then appear in the disassembling code, and Nvidia Visual Profiler(NVVP) software will show that the read-only data cache is actually being utilized.~~

to explicitly direct the compiler to implement the optimization. As an example, consider the calculations of advection and the horizontal diffusion terms. Because mpiPOM adopts the Arakawa C-grid, in the horizontal plane, updating the temperature(T) requires the velocity of longitude(u), the ~~update of $T(i,j,k)$ requires the value of $u(i,j,k)$, $u(i+1,j,k)$, $v(i,j,k)$ and $v(i,j+1,k)$, in addition to the value of~~ velocity of latitude(v) and the horizontal kinematic viscosity~~,~~ (aam~~, from four neighboring~~) on the neighbouring grid points. In one ~~time step, the arrays of~~ kernel, the u and v ~~must be~~ arrays are accessed twice, and the aam array ~~must be~~ is accessed four times. ~~Therefore it is natural to use the~~ After using read-only data cache to improve the data locality~~of gpuPOM. This optimization improves the performance of this part~~ , the running speed of this kernel is improved by 18.8%.

~~**(B) Local memory blocking**. Cache blocking is a common method to improve data reuse in parallel computing~~To reuse the data in each thread, we use local memory blocking to pull the data from global memory to the L1 cache. In this method, a small subset of a dataset is loaded into the fast on-chip ~~faster memory (e.g., the L1/L2 cache in the GPU and the CPU) and~~ memory and then the small data block is repeatedly accessed by the program.~~It is helpful to reduce~~ This method is helpful in reducing the need to access the off-chip with high latency memory~~(e. g., global memory on the GPU). Because regular global memory access cannot be cached in L1 cache for K20X GPU, the method used here is to pull the data from local memory to the L1 cache.~~

~~For the subroutines about~~ . In the subroutines of the vertical diffusion and source/sink terms, the chasing method is used to solve a tridiagonal matrix along the vertical direction for each grid point individually. ~~As shown in Algorithm **??**, the 3D temporary arrays in the original code, such as ee, gg, that store~~ Each thread only accesses its own tiles of row transformation coefficients~~are streamed from memory. However, these arrays are too large to reside in the cache entirely; code efficiency is therefore decreased. We find that each thread performs a column calculation from surface to bottom and there is no communication. Thus, we declare 1D~~ . As shown in Fig. 5, the arrays ~~ee_new, gg_new in local memory to replace the original 3D global arrays . Their size is equal to the level of ocean, $nz-1$, which is typically a very small value.~~

~~In the chasing method, these local arrays~~ are accessed twice within one thread, one from the surface($k=0$~~to~~) to the bottom($k=nz-1$) and another from the bottom($k=nz-1$~~to~~) to the surface($k=0$). After blocking the vertical direction arrays in local memory, the L1 cache is fully utilized ~~although some of them may be spilled to global memory. The performance of the subroutines~~

```
/*************************
 *3D arrays ee and gg represent row transformation
 *coefficients of the chasing method.
 *************************/
for (k = 1; k < nz-2; k++){
   ee[k][j][i] = ee[k-1][j][i]*A[k][j][i];
   gg[k][j][i] = ee[k-1][j][i]*gg[k-1][j][i] - B[k][j][i];
}

for (k = nz-3; k>= 0; k++){
   uf[k][j][i] = (ee[k][j][i]*uf[k+1][j][i]+gg[k]) * C[k][j][i];
}
```

(a) Original CUDA-C code

```
/*************************
 *Each thread pulls its own tile of ee,gg to
 *1D new arrays ee_new, gg_new(local memory).
 *There two new arrays can be cached in L1 for reuse.
 *************************/
for (k = 1; k < nz-2; k++){
   ee_new[k] = ee_new[k-1]*A[k][j][i];
   gg_new[k] = ee_new[k-1]*gg[k-1] - B[k][j][i];
}

for (k = nz-3; k>= 0; k++){
   uf[k][j][i] = (ee_new[k]*uf[k+1]+gg_new[k])*C[k][j][i];
}
```

(b) Optimized CUDA-C code

Figure 5. A simple example of local memory blocking.

~~about vertical diffusion and source/sink terms~~ and the running speed of these subroutines is improved by 35.3%~~when using the local memory blocking technique~~.

~~A simple example of local memory blocking~~

~~/********************** * Origin CUDA-C code**********************/ //ee, gg are parameter pointers of the function //that represent the use of global memory for (k~~ In current implementation, as in the original mpiPOM code, the 3D arrays of variables are stored sequentially as east-west(x), north-south(y), vertical(z), i.e., i, j, k ordering. 2D arrays are stored in i, j ordering. The vertical diffusion is solved using a tridiagonal solver which is calculated sequentially in the z direction. For simplicity, in our kernel functions the grid is divided along x and y. Each GPU thread then specifies an (x, y) point in the horizontal direction and performs all of the calculations from the surface to the bottom. The thread blocks are divided as 32×4 subdomains in the x-y plane. In the x direction, the block number must be a multiple of 32 threads to perform consecutive and aligned memory access within a warp (NVIDIA, 2015) . In the y direction, we tested many thread numbers, such as 4 and 8, and obtained similar performances. We ultimately choose 4 because this value produced more blocks and allowed us to distribute the workload more uniformly amongst the SMs. In addition, 128(=~~1; k < nz-2; k++){ eekji= eek-1ji*Akji; ggkji= eek-1ji*ggk-1ji-Bkji; } for (k = nz-3; k >= 0; k++){ ufkji= (eekji*ufk+1ji+ggk)*Ckji; } /******************** * After local memory blocking ********************/ //ee, ggare 1-D array declared in function //that represent the use of local memory for (k = 1; k < kbm1; k++){ ee= ee*Akji gg= ee*gg-Bkji; } for (k = nz-3; k >= 0; k++){ ufkji= (ee*ufk+1ji+gg)*Ckji; }~~ 32×4) threads are enough to maintain the full occupancy, which is the number of active threads in each multiprocessor.

In GPU computing, one is free to choose which arrays will be stored in an on-chip cache. Our experience involves putting the data along the horizontal direction into the read-only cache to reuse among threads, and putting the data along with vertical direction into the local memory for reuse within one thread.

~~**(C) Loop fusion**. Loop fusion is an effective method to store scalar variables in registers for data reuse. Registers are the fastest memory in the GPU memory hierarchy. For example, as shown in Algorithm **??**, if the variable *drhox(k, j, i)* must be read several times in multiple loops, we can fuse these loops into one. Therefore, the *drhox(k, j, i)* will be read from the global memory the first time and then repeatedly read from a register. This method can also be applied in a number of the mpiPOM subroutines.~~

~~Take the kernel *profq* as an example. After rewriting part of source code with loop fusion, the device memory transactions decrease by 57%, while the registers used per thread increase from 46 to 72, as reported in NVVP. Although the occupancy achieved decrease from 61.1% to 42.7%, the performance of this kernel is improved by 28.6%.~~

~~A simple example of loop fusion~~

~~/******************** * Origin cuda-c code ********************/ for (k = 1; k < kbm1; k++) { drhoxkji= drhoxk-1ji+ Akji; } for (k = 0; k < kbm1; k++){ drhoxkji= drhoxkji* Bkji; } /******************** * After loop fusion ********************/ for (k = 1; k < kbm1; k++){ drhoxkji= drhoxk-1ji+ Akji; drhoxk-1ji= drhoxk-1ji* Bkji; } drhox0ji= drhox0ji+ Bkji;~~

~~**(D) Function fusion**. Because we can fuse the loops in which the same arrays are accessed, we can also fuse functions in which similar formulas are calculated and the same arrays are accessed. For example, the *advv* and *advu* functions of the mpiPOM calculate advection in longitude and latitude, respectively, and they can be fused into one subroutine. This optimization benefits from the elimination of the redundancy calculations.~~

~~This optimization is also useful for the situation in which one function is called several times to calculate different tracers. For example,~~ Furthermore, we improve the *proft* ~~functions of the mpiPOM is called twice – once for temperature and once for salinity. Their computing formulas are similar and certain common arrays are accessed; these functions were modified to calculate temperature and salinity simultaneously. The method of Function fusion improves the performance of these functions by 28.8%.~~

~~**(E)ECC-off and GPU boost.** Because ECC memory consumes some amount of memory bandwidth, we can improve the GPU~~ global memory bandwidth by disabling the ~~error checking and memory correcting features. Also, the memory bandwidth that can be achieved is improved by~~ Error Checking and memory Correcting(ECC-off), as well as enhancing the clock ~~of SM core. In our implementation, we overclock the default clock of K20X GPUfrom 732 MHz to 784 MHz. The methods of ECC-off and GPU boost~~on the GPU(GPU boost). This method improves the performance of the ~~whole application~~ POM.gpu by 13.8%.

4.1.3 Results of the computational optimizations

We divide all ~~the gpuPOM subroutines into different~~ of the POM.gpu subroutines into three categories based on their different ~~computation~~ computational patterns. As shown in Table 2, in ~~gpuPOM~~the POM.gpu, we deploy different optimizations in ~~different categories to achieve improved performance~~ these categories to improve the performance of POM.gpu; these categories are ~~now described~~ described as follows.

(1) Category 1: Advection and horizontal diffusion (~~adv~~adv)

This category has 6 subroutines, and calculates the advection~~and~~, horizontal diffusion and ~~in the case of velocity, the~~ the pressure gradient and Coriolis terms ~~. Here~~ in the case of velocity. Here, it is possible to reuse data among adjacent threads, and the subroutines therefore benefit from using the read-only data cache~~and shared memory. Also~~. At the same time, the variables are calculated in different loops ~~of one function~~ or in different functions ~~, so~~ such that the loop fusion and function fusion optimizations ~~apply~~ are applied to this part as well.

(2) Category 2: vertical diffusion (~~ver~~ver)

This category has 4 subroutines, and calculates the vertical diffusion. In this part, the chasing method is used in the tridiagonal solver in the k-direction. The main feature is that ~~data is reused~~ the data are accessed twice within one thread, ~~while data is accessed once from $k = 0$ to $k = nz - 1$ and once from $k = nz - 1$ to $k = 0$~~once from the surface to the bottom and again from the bottom to the surface. The subroutines are significantly sped up after grouping the k-direction variable in the local memories.

(3) Category 3: vorticity (~~vort), baroclinic(baro~~$vort$), baroclinicity ($baro$), continuity equation (~~cont~~$cont$) and equation of state (~~state~~$state$)

This category is less time consuming than the two categories described above, but it also benefits from our optimizations. Because ~~there exists data reuse with adjacent~~ data exists reuse among threads, the use of a read-only data cache improves ~~efficiency. For vort~~data locality. For the $vort$ subroutine, there is data reuse within one thread, and thus the loop fusion improves the ~~efficiency~~data locality.

~~The main bottleneck of the mpiPOM is memory-bound problem. To confirm this issue, we use the Performance API(Browne et al., 2000) to estimate floating point operation count and the memory access(store/load) instruction count. Results reveal that the computational intensity(flops/byte) of the mpiPOM is around 1:3.3, while the computational intensity provided by SandyBridge E5-2670 CPUs is 7.5:1, and large arrays are mostly streamed from memory and shows little locality. According to the roofline model(Williams et al., 2009) , the whole mpiPOM is mainly memory-bounded. In addition, the mpiPOM suffers from a flat profiling results, with even the most time-consuming subroutine just occupying 20% of the total execution time. Namely, there are no obvious hot spot functions in the mpiPOM and porting a handful of subroutines to GPU is not helpful to improve the model efficiency. That is the reason that we need to port the whole program from CPU to GPU.~~

Table 2. Different subroutines adopt different optimizations in ~~gpuPOM~~the POM.gpu

Subroutines	~~A~~ Loop fusion	~~B~~ Function fusion	~~C~~ Read-only data cache	~~D~~ Local memory blocking	~~E~~ ECC-off & GPU boost	Speedup
Adv & Hor diff	√	√	√		√	2.05X
Ver diff	√	√		√	√	2.82X
~~Baroclinic~~ Baroclinicity	√		√		√	2.08X
Continuity ~~equ~~equation	~~√~~		~~-~~√		√	1.39X
Vorticity	√		√		√	3.19X
State ~~equ~~equation	~~√~~		√		√	1.35X

~~To alleviate the memory bound problem, an optimization method that is frequently used is cache blocking. It is usually cache beneficial to use vertical index as the innermost array index(z,x,y ordering). For the mpiPOM with 962 × 722 × 51 test case, one array has 962 × 722 × 4bytes= 2.6MBytes in the x-y plane, while one CPU has a 32KB per-core L1 cache, 256KB per-core L2 cache and 20MB shared L3 caches. Take the chasing method in vertical diffusion terms as an extreme case. If x,y,z ordering is used, in terms of calculation along z-axis, each x-y plane is blocked in L3 cache for reuse. When traversing backwards along z, the data needed are all evicted. If z,x,y ordering is used, in terms of calculation along z-axis, each k column data is blocked in L1 cache for reuse. When traversing backwards along z, the data remains valid in L1 cache. Unfortunately, the mpiPOM uses east-west index as the innermost array index. However, for gpuPOM, z,x,y ordering has to be avoided to satisfy GPU memory coalescing, which is also demonstrated in Shimokawabe et al. (2010) . We make east-west (x) as innermost index(x,y,z ordering). A big difference for memory optimizations between GPU and CPU is that, in GPU, programmers can artificially choose which array to store in cache. Moreover, GPU provides various on-chip caches, such as L1/L2 cache, shared memory, texture cache. Thus, according to how the arrays are used, we can put different arrays in different caches. In the gpuPOM, we have explored a better data placement on different caches for different terms, besides conventional cache blocking optimizations.~~

4.2 Communication optimizations among multiple GPUs

In this section, we present the optimizing strategies ~~used to harness the computing power of~~ for multiple GPUs. ~~With multiple GPUs, the computing domain is divided into smaller blocks than with a single GPU. The performance of GPU computing is faster and the memory requirement for each GPU is reduced. To utilize multiple GPUs, an effective domain decomposition method and communication method should be employed. We split the domain along the x~~ In the mpiPOM, the entire domain is split along the horizontal directions and ~~y directions (2-D decomposition) and assign~~ each MPI process ~~for~~ is responsible for the model's computation of one subdomain, following

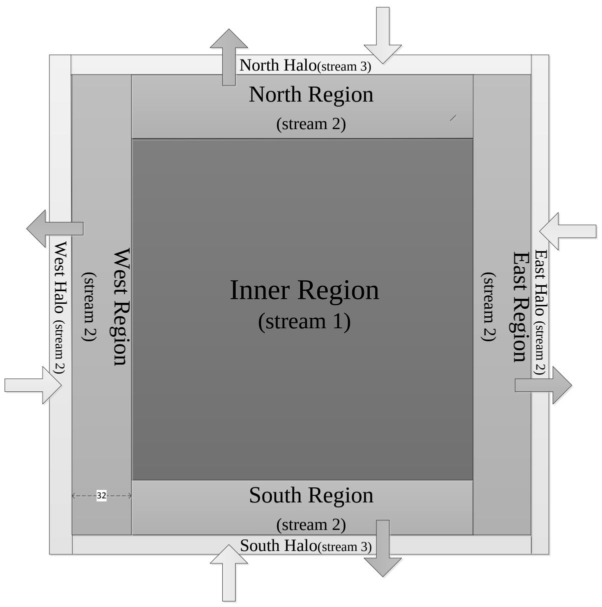


Figure 6. Data decomposition in ~~gpuPOM~~the POM.gpu

Jordi and Wang (2012). ~~Then~~In the POM.gpu, we attach ~~the~~ one MPI process to one GPU and ~~send messages from one GPU to another.~~ move the complete computation to the GPU. The MPI process is in charge of the computation within each subdomain and of the data transfer between the GPU and main memory. The data transfer between subdomains is handled by the GPUs directly. Shimokawabe et al. (2010) and Yang et al. (2013) proposed ~~some~~ fine-grained overlapping methods of GPU computation and CPU communication to improve the ~~simulation~~ computing performance. An important ~~common issue~~ issue in their work is that the communications between multiple GPUs explicitly require the participation of the CPU. In our current work, we ~~hope to implement the communication to~~ simply bypass the CPU ~~to fully employ~~ in implementing the communication to fully exploit the capability of the ~~GPU~~GPUs.

~~State-of-the-art~~ At present, two MPI libraries, ~~such as~~ OpenMPI and MVAPICH2, ~~have announced their support for MPI communication directly from GPU memory, which is known~~ provided support for the direct communication from the GPU to the main memory. This capability is referred as CUDA-aware MPI. We ~~tried~~ attempted to use MVAPICH2 to implement direct communication among multiple GPUs~~at first~~. However, we found that ~~the boundary operation and MPI~~ inter-domain communication occupied nearly ~~15~~18% of the total runtime~~after GPU porting~~.

~~To~~ Instead, to fully overlap the boundary operations and MPI communications with computation, we adopt the data decomposition method shown in Fig. 6. The data region is decomposed into three ~~parts~~regions: the inner ~~part~~region, the outer ~~part, and the halo part. The outer part includes east/west/north/south part, and the halo part also includes east/west/north/south halos to exchange~~

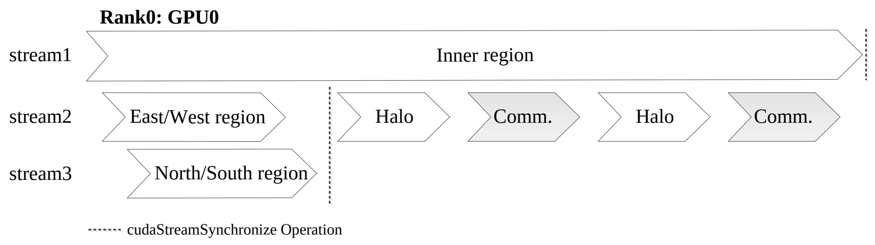


Figure 7. The workflow of multiple streams on the GPU. The “inner/east/west/north/south ~~part"~~ region” and “Halo~~"~~” refer to the computation and update of the corresponding ~~part~~region. “Comm.~~"~~” refers to the communication between processes, which implies synchronization.

~~data with neighbors. In CUDA, a stream is a sequence of commands that execute in order; different streams can also execute concurrently with different priorities. In~~ region, and a halo region which exchanges data with its neighbours. In our design, the inner ~~part~~region, which is the most time-consuming ~~part with the largest workload~~ is allocated to “stream 1~~in which to execute~~”. The east/west outer ~~part~~ region is allocated to “stream 2” and the north/south outer ~~part~~ region is allocated to ~~stream 3.~~ “stream 3”. In the east/west outer ~~part~~region, the width is set to 32 to ensure ~~coalesced~~ consecutive and aligned memory access in a warp~~to improve performance. The halo part is~~. All of the halo regions are also allocated to ~~stream 2.~~ “stream 2”.

The workflow of multiple streams on the GPU is shown in Fig. 7. The east/west/north/south ~~parts are normal~~ regions are common kernel functions that can run in parallel with the inner ~~part~~ region through different streams. The communication operations between domains are implemented by ~~*cudaMemcpyAsync*, which is~~ an asynchronous CUDA memory copy~~function~~. The corresponding synchronization ~~operation~~ operations between the CPU and the GPU or among the MPI processes are implemented ~~with *cudaStreamSynchronize* function and *MPI_barrier*~~ by a synchronization CUDA function and a MPI barrier function. To ~~hide~~ overlap the subsequent communication ~~by the inner part,~~ with the inner region, “stream 2~~and~~” and “stream 3” for the outer ~~part~~ region have higher priority ~~to preempt~~ in preempting the computing resource from “stream 1” at any time.

~~Current CUDA-aware MPI implementation such as MVAPICH2 is not suitable for the “Comm." part in Fig. 6. We found the two-sided MPI functions *MPI_Send* and *MPI_Recv* will block the current stream so that the concurrency pipeline is broken. The probable cause is synchronous *cudaMemcpy* function is called in the current implementation of *MPI_Send* and *MPI_Recv*, according to Potluri et al. (2012) . Moreover, the implementation of non-contiguous MPI datatype for communication between GPUs is not efficient enough for the gpuPOM. The computation time of many kernels is about a few hundred microseconds to a few milliseconds while MPI latency for our message size is about the same, which means the outer part update and communication can not be fully overlapped .~~

~~From CUDA 4.1, the Inter-Process Communication (IPC) feature has been introduced to facilitate direct data copy among multiple GPU buffers that are allocated by different processes. The IPC is implemented by creating and exchanging memory handles among processes and obtaining the device buffer pointers of others. This feature has been utilised in CUDA-aware MPI libraries to optimise communications within a node. Therefore, we decided to implement the communication among multiple GPUs by calling the low-level IPC functions and asynchronous CUDA memory copy functions directly, instead of using high-level CUDA-aware MPI functions. Our communication optimizations among multiple GPUs are mainly implemented with the following two optimizations.~~

~~.~~

~~First, we put the phases of creating, exchanging and opening relevant memory handles into the initialization phase of the gpuPOM, which is executed only once. This method can remove the overhead of IPC memory handle operations during each MPI communication operation. The *cudaMemepyAsync* function with the corresponding device buffer pointers of neighbor processes replaces the original MPI functions.~~

~~Second, we take full consideration of the architecture of our platform in which 4 GPUs are connected with two I/O Hubs (IOHs). As illustrated in Fig. **??**, there are two Intel SandyBridge CPUs that connect two GPUs. Both the CPUs are themselves connected through Intel QuickPath Interconnect (QPI). Notation ① means that the communications between GPUs are connected with the same IOH support Peer-to-Peer (P2P) access. Notation ② represents the communications in which P2P access is not supported. If MPI0 (context on GPU-0) sends data to MPI2 (context on GPU-2), rank 0 must switch its context to GPU-2 temporally and opens the corresponding memory handles to obtain the device buffer pointers of rank 2. For those GPUs that do not support P2P access between one another, we must switch context to the same GPU before opening the corresponding memory handles. We then call regular *cudaMemepyAsync* functions to fulfill data communications. For communications between GPUs on the same IOH, the switching context is not necessary. Although the function cudaMemepyAsync is used in the communication of both ① and ②, the NVVP software shows that ① does a device-to-device memory copy that bypasses the CPU, whereas ② does a device-to-host and a host-to-device memory copy that involves the CPU. The 2-D decomposition introduced in Fig.~~ **~~??~~** ~~is an example to demonstrate~~ Based on this workflow, the inter-domain communication is overlapped with the computation. The experimental results show that our design can ~~easily extend to 8 or more GPUs within one node~~remove the communication overhead taken by MVAPICH2.

~~Communications pattern among multiple GPUs in one node~~

4.3 I/O optimizations between ~~hybrid GPU~~ the GPUs and ~~CPU~~the CPUs

The time consumed for I/O in the ~~original~~ mpiPOM is not significant~~because the output frequency is relatively low~~. However, after we fully accelerate the model by GPU, ~~the I/O overhead, which is~~

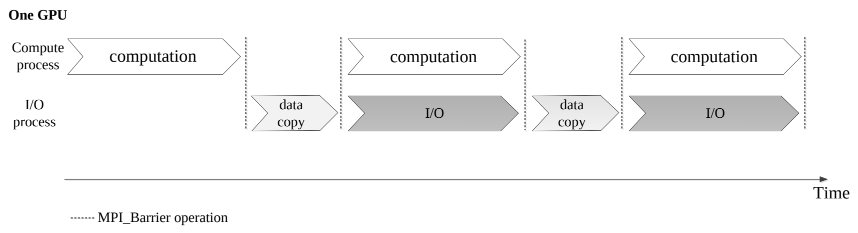


Figure 8. One computing process and one I/O process both set their contexts on the same GPU. During the data copy phase, the computing process remains idle and the I/O process will copy data from the GPU to the CPU ~~host~~ through the $cudaMemcpy$ function.

it accounts for approximately 30% of the total runtime~~, cannot be ignored. As described in Sec. 4.2, each MPI process sets its context on one GPU and is responsible for launching kernel functions on this GPU, and the CPU is used to collect and output data. In fact, in most climate models, including the mpiPOM, the~~ . The computing phase and the I/~~O phase run alternately. In a sense, the computing phase and the I/~~O phase are serial, which means that the GPU will remain idle until the CPU finishes the I/O operations. ~~Huang et al. (2014) designed a fast~~ Motivated by previous work on I/O ~~library for climate models and provided automatic overlapping of I/O and computing. Motivated by their work, we design a method so that computations on~~ overlapping (Huang et al., 2014) , we designed a similar method following computations on a GPU and I/O operations on ~~CPU can~~ a CPU to run in parallel.

~~Because MPI processes are blocked during the output phase and cannot launch kernels to GPUs, we choose~~ In the POM.gpu, we chose to launch more MPI processes. ~~We divide all the MPI processes~~ The MPI processes are divided into computing processes and I/O processes with different MPI communicators. The computing processes are responsible for launching kernel functions as usual, and the I/O processes are responsible for output. One I/O process attaches to one computing process and these two processes set their contexts on ~~one single GPUthrough *cudaSetDevice* function. The total number of MPI processes are twice the size they were before.~~ the same GPU.

~~Since~~ Because the I/O processes must fetch data from the GPU, ~~where the data are allocated by the computing processes,~~ communication is necessary between them. ~~Here, we again utilize the feature of CUDA IPC, as introduced in Sec. 4.2. Through CUDA IPC, the~~ The I/O processes obtain the device buffer pointers from the computing processes during the initialization phase. When ~~there is a need to output data~~writing history files, the computing processes are blocked and ~~kept~~ remain idle for a short time~~while~~ , waiting for I/O processes to fetch data. Then, the computing processes continue their computation, and the I/O processes complete their output in the background, as illustrated in Fig. 8. This method can be further optimized by placing the archive data into a set-aside buffer and carry on the main calculation. However, the method requires more memory, which is not abundant in current K20X GPUs.

The advantage of this method is that it overlaps the I/O on the CPU with ~~computation~~ the model calculation on the GPU. In ~~the~~ serial I/O, the GPU computing processes are blocked while data are ~~brought to the host~~ sent to the CPU and written to disk. In ~~the~~ overlapping I/O, the computing processes only wait for the data to be ~~brought~~ sent to the host. ~~In addition, the~~ The bandwidth of data brought to the host ~~through the PCI-express bus~~ is approximately 6 ~~GBps~~GB/s, but the output bandwidth to the disk is approximately 100 ~~MBps~~MB/s, as determined by the speed of the disk. Therefore, the overlapping method significantly accelerates the entire application.

~~The 3 cases when a subroutine is broken into small kernels.~~

~~Note that there are more than 50 kernel functions in the current version of gpuPOM. The main reason that we have a large number of kernels in gpuPOM is that there exist numerous subroutines in mpiPOM. Since we port the entire model one subroutine by one subroutine, which is a convenient way to debug the gpuPOM and to guarantee its bit-by-bit identical results to mpiPOM, we need to write a large number of GPU kernels. Further more, we break several subroutines of mpiPOM into several GPU kernels of gpuPOM in 3 cases: when subroutine B is invoked in subroutine A(illustrated in Fig. **??**(a)), when a MPI function call is invoked in subroutine A(illustrated in Fig. **??**(b)), and when interior array is first written by one thread and later read by adjacent threads, in the mean while caching this array in shared memory makes no sense(illustrated in Fig. **??**(c)). Although function fusion has been done as described in Section 4.1 (**D**), aggressive function fusion to make use of data locality between functions is a promising optimization(Wahib and Maruyama, 2014) . But, a redesign of the code structure of mpiPOM is needed and it is a part of our future work.~~

5 Experiments

In this section, we first describe the specification of our platform and comparison ~~methods used~~ methodology to validate the correctness of the ~~gpuPOM.~~POM.gpu. Furthermore, we present the performance and scalability of the ~~gpuPOM on the GPU platform in comparison~~ POM.gpu comparied with the mpiPOM~~on the CPU platform~~.

5.1 Platform Setup

The ~~GPU platform used in our experiments is a super workstation computer~~ POM.gpu runs in a workstation consisting of two CPUs and ~~4 GPUs, as illustrated in Fig. ??.~~ four GPUs. The CPUs are 2.6 GHz 8-core Intel ~~E5-2670 (architecture code-named SandyBridge), which can turbo to 3.0 GHz when all 8 cores are utilized. The peak single-precision performance of the Intel SandyBridge CPU is 384 GFlops and the peak memory bandwidth is 51.2 GBps. The~~ SandyBridge E5-2670. The GPUs are Nvidia ~~Telsa~~ Tesla K20X~~, equipped with 2,688 GPU-cores and 6 GB GDDR5 fast on-board memory. The peak single-precision performance of K20X GPU is 3.95 TFlops and the peak memory bandwidth is 250 GBps. Therefore, the aggregated performance provided with 4 GPUs can~~

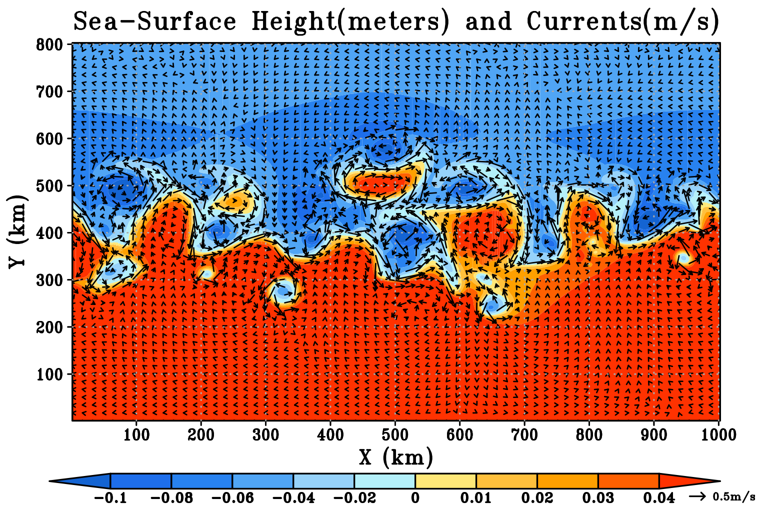
~~reach 16 TFlops and 1 TBps memory bandwidth, which is sufficient to execute the general simulation research for regional ocean models thus far. The~~ . The operating system is RedHat Enterprise Linux 6.3 x86_64. ~~The programs on this platform~~ All programs are complied with Intel compiler v14.0.1, CUDA 5.5 Toolkit, Intel MPI Library v4.1.3 and ~~CUDA 5.5 Toolkit. For the purposes of~~ MVAPICH2 v1.9.

For comparison, the ~~CPU platform used in our experiments is~~ mpiPOM runs on the $Tansuo100$ ~~supercomputer~~ cluster at Tsinghua University ~~, which consists~~ consisting of 740 nodes~~, each of which has~~ . Each node is equipped with two 2.93 GHz 6-core Intel Xeon X5670 ~~processors~~ CPUs and 32 GB of memory. The nodes are connected through an ~~Infiniband network, which provides a maximum bandwidth of 40 Gbps. The node~~ infiniband network. The operating system is RedHat Enterprise Linux 5.5 x86_64. ~~All the programs~~ Programs on this platform are compiled with Intel compiler v11.1 ~~, and the MPI environment is~~ and Intel MPI v4.0.2. The ~~Original~~ mpiPOM code is ~~benchmarked with its initial compiler flags(~~compiled with its original compiler flags, i.e., "-O3 ~~-precise) and also with the same Intel compiler. We also use the GPUDirect technology within MVAPICH2 v1.9 to test the communication effects among multiple GPUs, and compare the results with our implementation.~~ -fp-model precise".

5.2 The test case and the verification of accuracy

The ~~"dam break"~~ "dam break" simulation (Oey, 2014) is conducted to verify the correctness and test the performance and ~~the~~ scalability of the ~~gpuPOM.~~ POM.gpu. It is a baroclinic instability problem ~~which~~ that simulates flows produced by horizontal temperature gradients. The model domain is configured as a straight channel with a uniform depth of 50 m. Periodic boundary conditions are used in the east-west direction, and the channel is closed in the north and south. Its horizontal resolution is ~~$1km \times 1km$. To test large computational grid, the~~ 1km$\times$1km. The domain size of this test case is ~~increased to~~ 962×722 horizontal grid points and 51 vertical sigma levels, which is limited by the capacity of ~~on-board~~ one GPU's memory. Initially, the temperature in the southern half of the channel is 15^{o}C and 25^{o}C in the northern half. The salinity is fixed at 35 psu. The fluid is then allowed to adjust. In the first 3-5 days, geostrophic adjustments ~~occurs. Then~~occur. Then, an unstable wave develops due to baroclinic instability. Eventually, eddies are generated. Figure 9 shows the sea-surface height (SSH), sea-surface temperature (SST), and currents after 39 days. The ~~development of a gravity wave is manifest. Noticeably, The gravity wave is confined in the middle of the channel by Rossby radius deformation.~~ scales of the frontal wave and eddies are determined by the Rossby radius of deformation. This dam break case uses a single-precision format.

To verify the accuracy, we check the binary output files ~~output from the original mpiPOM and the gpuPOM. This testing method is also used in the GPU-porting of ROMs (Mak et al., 2011) . As introduced in Whitehead and Fit-Florea (2011) , the same inputs will give identical results for individual IEEE-754 operations except in a few special cases. These cases can be classified into~~



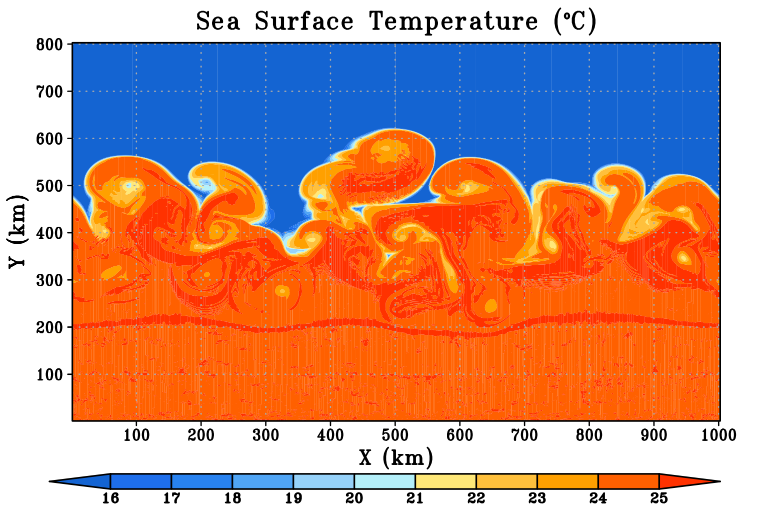


Figure 9. The model results after 39 days of simulation. For the ~~up~~ top figure, ~~color~~ the colour shading is the ~~Sea Surface Height~~ sea-surface height (SSH)~~. Vectors~~ , and vectors are ocean ~~current~~currents. For the ~~low~~ bottom figure, ~~color~~ the colour shading is the ~~Sea Surface Temperature~~ sea surface temperature (SST). Several warm and cold eddies are generated in the middle of the domain where the SST gradient is largest~~. Noticeably, the gravity wave is confined in the middle of the channel~~ ; their scales are determined by the Rossby radius of deformation.

~~three categories: different operations orders, different instructions and different implementations of math libraries. For the first in our study, the parallelization of the mpiPOM does not change the order of each floating point operation and we benefit from this. For the second case in our study, the GPUs~~

~~have fused multiply-add (FMA) instruction while the CPU does not in our CPU platform.Because this instruction might cause a difference in the numerical results, we disable FMA instructions with the "-fmad=false" compiler flag for the GPUs. For the third case in our study, the value of exponent used in the GPU has a maximum of 2 rounding errors NVIDIA (2014) . Fortunately, in the execution path of our dam break simulation, the power of the exponent functions remains unchanged over the entire simulation. Therefore, we accomplish this function on the CPU during the initialization phase and copy the results to the GPU for later data reuse. The experimental~~ of the mpiPOM and the POM.gpu, as in Mak et al. (2011) . The test results demonstrate that the ~~output variables regarding~~ variables velocity, temperature, salinity and sea surface height are all identical.

5.3 Model Performance

To understand the advantages of the ~~optimizing methods introduced~~ optimizations in Sec. 4, we ~~test the dam break case with different experiments . The current dam break case uses single-precision format. The~~ conducted different tests. The metrics of seconds per simulation day ~~, which is the walltime it requires to obtain 24 hours in the simulation, is measured and used~~ are measured to compare the model performance.

~~In the first experiment~~

5.3.1 Single GPU performance

In our first test, we compare the ~~gpuPOM with the mpiPOM on different hardware platforms, including K20X GPU, the Intel Westmere 6-cores~~ performance of the mpiPOM using two different CPUs, the Intel X5670 CPU (6 cores) and the Intel ~~SandyBridge~~ E5-2670 CPU ~~.~~ (8 cores), with that obtained from the POM.gpu using one single GPU. Fig. 10 shows that one K20X GPU can compete with approximately 55 ~~Intel SandyBridge CPU cores or~~ E5-2670 CPU cores to 95 ~~Intel Westmere CPU cores . Obtaining such a speedup on a pure CPU platform is reasonable. Taking the SandyBridge CPU platform as an example, the theoretical memory bandwidth of one 8-core~~ X5670 CPU cores in the simulation. From the parameters of the Intel E5-2670 CPU ~~is 51.2 GBps, and the peak single-precision floating point performance is 384 GFlops with all 8 cores turbo to 3.0 GHz. However, for~~ and Nvidia K20X GPU, ~~the memory bandwidth and peak single-precision floating point performance are 250 GBps and 3.95 TFlops, respectively. The approximate~~ we find that, the ratio of memory bandwidth ~~between one SandyBridge CPU and one K20X GPU is 1 : 5, and~~ and the ratio of floating points performance ~~between one SandyBridge CPU and one K20X GPU is~~ are approximately 1:~~10. Namely,~~ 5 and 1:10, respectively. This means, if an application is strictly memory ~~bound, one K20X~~ bandwidth limited, one GPU can compete with 5 ~~SandyBridge CPUs. In addition,~~ CPUs; if an application is strictly ~~computing bound~~computation limited, it can compete with 10 ~~SandyBridge CPUs. As~~ CPUs. Since the mpiPOM is memory ~~bound, according to the memory bandwidth ratio between the CPU and the GPU, our gpuPOM~~ bandwidth limited, the

POM.gpu should provide equivalent performance to ~~5x8 = 40 CPU cores. Combining our careful memory optimizations, our final design achieves another performance boost of 25%, and one GPU provides similar performance to more than 50 Intel 8-core SandyBridge~~ the mpiPOM running on up to $5 \times 8 = 40$ CPU cores. ~~Compared with Intel Westmere 6-cores CPU, our results provide similar performance to more than 95 CPU cores.~~ Our procedure attempts to optimize memory access and we can further increase this number to 55.

~~The performance API tool (PAPI) shows that the performance of the gpuPOM on single K20X is 107.3Gflops in single-precision for the 962*722*51 grid size. The low performance in Gflops reflects the memory-bound problem in climate models. Previous work such as time skewing (McCalpin and Wonnacott (1999); Wonnacott (2 can make a stencil computation compute bound by making use of data locality between different time-steps. However, for real-world climate models including mpiPOM, the code is usually tens to hundreds of thousands lines and analyzing the dependency manually is tough. Designing an automated tool to further analyze and optimize the mpiPOM and the gpuPOM is a part of our future work.~~

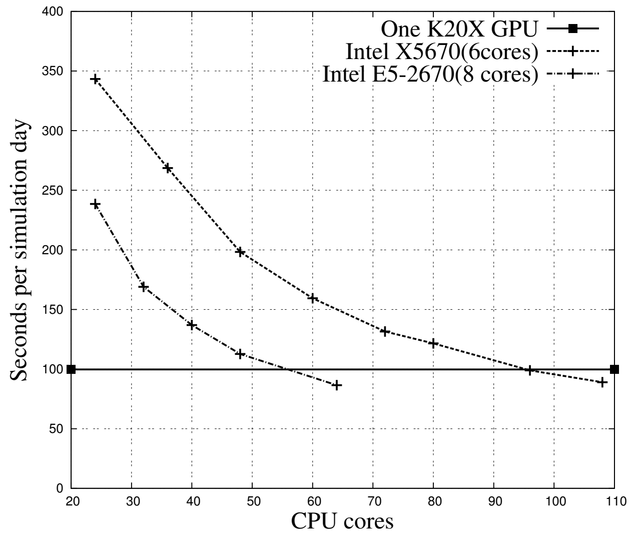


Figure 10. Performance comparison with different hardware ~~platform~~platforms

5.3.2 Multiple GPUs performance

In the second ~~experiment, we test the~~ test, we compare our communication overlapping method ~~used in the gpuPOM and compare it~~ with the MVAPICH2 ~~. In the current MVAPICH2, the communication and boundary operations are not overlapping with computing.~~ library. Fig. 11 ~~shows~~ presents the weak scaling performance ~~of the gpuPOM~~ on multiple GPUs~~. To maximize performance,~~, where the grid size for each GPU is ~~set to~~ kept at $962 \times 722 \times 51$. When ~~using~~ 4 GPUs ~~with the implementation of~~ are used with MVAPICH2, approximately 18% of the total runtime is consumed ~~in executing~~

~~the~~ by inter-domain communication and boundary operations. This overhead ~~does not exist in~~ can be greatly reduced by our communication overlapping method. ~~Fig. 11 shows that it spends almost the same time when using different GPUs because the communication and boundary operations are almost fully overlapped with the inner part of the computation.~~

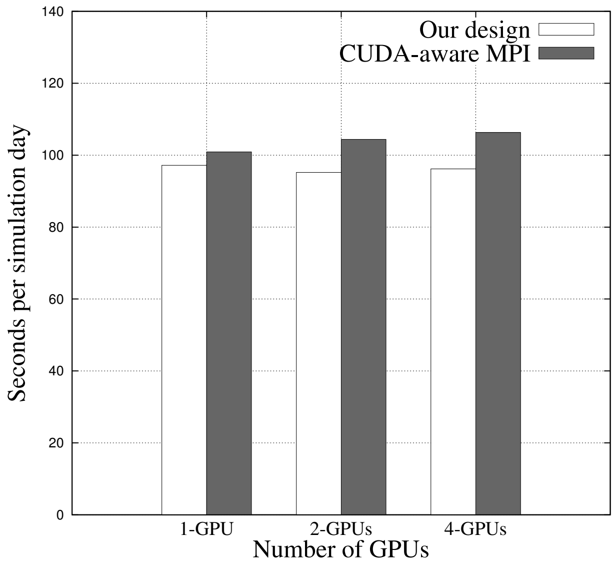


Figure 11. The weak scaling test between our communication overlapping method and the MVAPICH2 subroutines.

In the third ~~experiment, we test the efficiency of the gpuPOM on multiple GPUs. Table 3 shows the strong scaling result of the gpuPOM on multiple GPUs. We~~ test, we fix the global grid size at $962 \times 722 \times 51$~~and increase the amount of GPUs gradually.The results show~~ , and measure the strong scaling performance of POM.gpu. Table 3 shows that the strong scaling efficiency is 99% on 2 GPUs and 92% on 4 GPUs. ~~A smaller subdomain will decrease~~ When more GPUs are used, the size of each subdomain becomes smaller. This decreases the performance of ~~the gpuPOM~~ POM.gpu in two aspects. First, ~~communication time can easily~~ the communication overhead may exceed the computation time ~~in the inner part and cannot be overlapped. As the subdomain size decreases, the inner part computation time decreases, but the communication time will not decrease because latency is the dominant factor~~of the inner region as the size of each subdomain decreases. As a result, the overlapping method in Section 4.2 are not effective. Second, ~~the latency of kernel launching and overhead of implicit synchronization after kernel execution will not decrease. There are a series of small~~there are many "small" kernels in the ~~gpuPOM, and the execution time is close to launching latency and synchronization overhead. When the subdomain size decreases, the impact of these delays expands~~POM.gpu code, in which the calculation is simple and less time consuming. With fewer inner region computations, the overhead of kernel launching and implicit synchronization with kernel execution must be counted.

Table 3. The strong scaling result of ~~gpuPOM~~POM.gpu

Number of GPUs	1-GPU	2-GPUs	4-GPUs
Time(s)	97.2	48.7	26.3
Efficiency	~~1.00~~ 100%	~~0.99~~ 99%	~~0.92~~ 92%

5.3.3 I/O performance

In the fourth ~~experiment, we test the performance of the~~ test, we compare our I/O overlapping method ~~and compare it with the default~~ with the parallel NetCDF (PnetCDF) method and NO-I/O~~method~~. NO-I/O means that all I/O operations are disabled in the program and that the time measured is the pure computing time. ~~We simulated the experiment~~ This simulation is run for 20 days~~and output~~ , and the history files ~~daily in the netCDF format. The variables included in the output netCDF files are 2-dimensional arrays of size 722 × 482 and 3-dimensional arrays of size 722 × 482 × 51. The~~ are output daily. The final history files in NetCDF format are approximately 12 GB. Fig. 12 shows that the I/O overlapping method outperforms the ~~default~~ PnetCDF method. For 1 GPU and 2 GPUs, the overall runtime decreases ~~from1694~~from 1694/1142 seconds to 1239/688 seconds, which is close to the NO-I/O~~method. The small difference between our design and~~. The extra overhead of our method compared with NO-I/O ~~is that~~ involves the computing processes ~~must~~ that need to be blocked until the I/O processes ~~bring~~ obtain data from the ~~GPU. For the case of~~ GPUs. When running with 4 GPUs, the output time ~~is longer than computational time because the latter is fast and the I/O timeis relatively large such that~~ exceeds the computation time. Then, the I/O phase cannot ~~fully overlap with the computing~~ be fully overlapped with the model computation phase. The overall runtime equals the sum of the computation time and the non-overlapped I/O time.

5.3.4 Comparison with a cluster

In the last ~~experiment, we test different workloads with the gpuPOM and compare the results with the mpiPOM on~~ test, we compare the performance of POM.gpu on a workstation containing 4 GPUs with that on the $Tansuo100$ ~~platform. The available global grid size are choosen from the three~~ cluster. Three different high-resolution ~~sets~~ grids (Grid-1: 962 × 722 × 51, Grid-2: 1922 × 722 × 51, Grid-3: 1922 × 1442 × 51) are used. Fig. 13 shows that our workstation with 4 GPUs is comparable to ~~a powerful cluster with~~ 408 standard CPU cores (= 34 nodes ~~*~~ × 12 cores/node) ~~for the simulationof mpiPOM. Since the Thermal Design Power(TDP)~~ in the simulation. Because the thermal design power of one X5670 CPU ~~(6-cores) is 95W~~ is 95 W and that of one K20X GPU is ~~235W, it means using 4 GPUs brings 6.8 times less energy consumption compared with 408 CPU cores. Small subdomains will decrease the performance of the gpuPOM as discussed in the strong scaling test, but it may greatly benefit the mpiPOM on the CPU. The last level cache of one SandyBridge CPU in~~

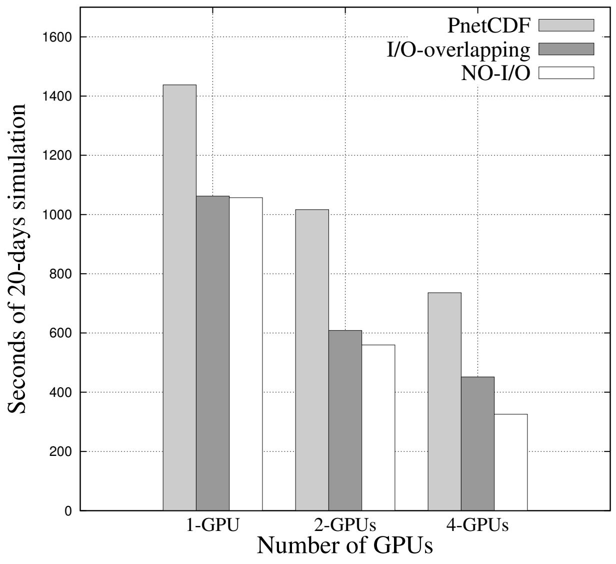


Figure 12. I/O ~~Test~~ test for ~~gpuPOM~~the POM.gpu

~~our platform is 20 MB, whereas that of K20X GPU is only 1.5 MB. As the subdomain size for~~ 235 W, we reduce the energy consumption by a factor of 6.8. Theoretically, as the subdomain of each MPI process ~~decreases~~becomes smaller, the cache hit ratio ~~will increase on a pure CPU platform, which can surely improve the performance especially for the memory-bound~~ of the mpiPOM code will increase. This will greatly alleviate the memory bandwidth-limited problem. However, ~~for~~ in the simulation on 408 standard CPU cores, the MPI communication ~~time~~ may occupy more than 40% of the total execution time. ~~With the number of cores increasing~~ When scaling to over 450 cores, the mpiPOM simulation may instead become slower, ~~the execution time may increase instead ,~~ as shown in Fig. 13. ~~As a result, our GPU solution has an overwhelming~~ Therefore, for high-resolution ocean modelling, our POM.gpu has a clear advantage compared to the ~~CPU because the communication overhead is less expensive and overlapped~~original mpiPOM.

6 Code ~~availablity~~availability

~~The gpuPOM used to simulate the regional ocean dynamic and physical process releases with the~~

The POM.gpu version 1.0 ~~series, which is freely available at . Note that the testing script "~~is available at https://github.com/hxmhuang/POM.gpu. To reproduce the test case in Section 5, the script “run_exp002.sh~~" can be downloaded~~” is provided to compile and execute the ~~codes, and to reproduce the test case.~~POM.gpu code.

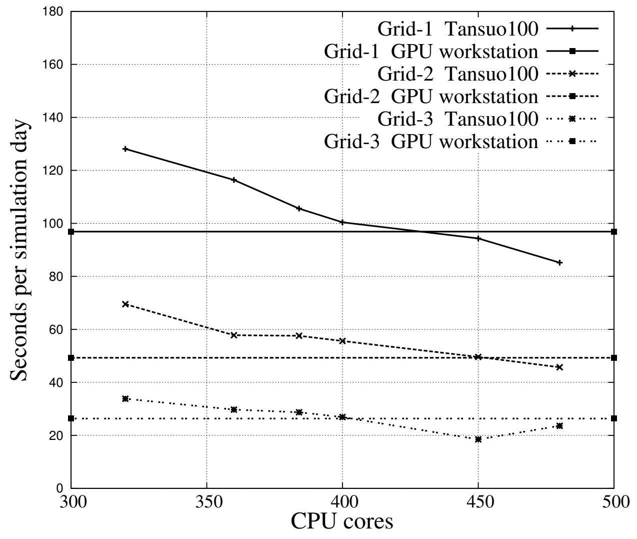


Figure 13. ~~Four GPUs performance~~ Performance test of four GPUs compared with $Tansuo100$ ~~clusters(Intel Westmere CPUs)~~cluster

7 Conclusions and future work

In this paper, we ~~provide~~ develop POM.gpu, a full GPU ~~accelerated solution of POM. Unlike partial~~ solution based on the mpiPOM. Unlike previous GPU porting, ~~such as WRF and ROMs, the gpuPOM does all the~~ the POM.gpu code distributes the model computations on the GPU. ~~The main contribution of our work includes a better use of state-of-the-art GPU architecture, particularly regarding the memory subsystem, a new design of a communication and boundary operations overlapping approach and a new design of an~~ Our main contributions include: optimizing the code on each of the GPUs, the communications between GPUs, and the I/O ~~overlapping approach. With the~~ process between the GPUs and the CPUs. Using a workstation with 4 GPUs, we achieve ~~over 400x speedup against a single CPU core, and provide equivalent performance to~~ the performance of a powerful CPU cluster with ~~more than 400 coresand reduce~~ 408 standard CPU cores. Our model also reduces the energy consumption by a factor of 6.8~~times. This work provides~~ . It is a cost-effective and ~~efficient ways in ocean modeling.~~ energy-efficient strategy for high-resolution ocean modelling. We have described the method and tests in details and, with the availability of the POM.gpu code, our experiences may hopefully be useful to developers and designers of other general circulation models.

In our current POM.gpu, we design a large number of kernel functions because we port the entire mpiPOM one subroutine at a time. This was done to simplify the debugging of POM.gpu and to check that the results are consistent with the mpiPOM. In our future work, we will adjust the code structure of POM.gpu and adopt aggressive function fusion to further improve the performance.

Previous studies proposed to take advantage of data locality between time steps by time skewing (McCalpin and Wonnacott, 1999; Wonnacott, 2000) , thus transforming the problem of memory bandwidth into the problem of computation. However, the real-world ocean models, including the mpiPOM, often involve hundreds of thousands lines of code, and analysing the data dependency and applying time skewing in such a context are challenging and difficult. We leave that to the next-generation POM.gpu.

Acknowledgements. ~~This work is supported in part by a grant from the~~ The author would like to thank David Webb, Robert Marsh and the anonymous reviewer for their valuable comments and improvements regarding the presentation of this manuscript. This study was supported by funding from the National Natural Science Foundation of China(41375102), the National Grand Fundamental Research 973 Program of China (No. 2014CB347800), and the National High Technology Development Program of China (2011AA01A203).

References

Allen, J. S. and Newberger, P. A.: Downwelling Circulation on the Oregon Continental Shelf. Part I: Response to Idealized Forcing, Journal of Physical Oceanography, 26, 2011–2035, doi:10.1175/1520-0485(1996)026<2011:DCOTOC>2.0.CO;2, 1996.

Berntsen, J. and Oey, L.-Y.: Estimation of the internal pressure gradient in σ-coordinate ocean models: comparison of second-, fourth-, and sixth-order schemes, Ocean dynamics, 60, 317–330, 2010.

Blumberg, A. F. and Mellor, G. L.: Diagnostic and prognostic numerical circulation studies of the South Atlantic Bight, Journal of Geophysical Research: Oceans (1978–2012), 88, 4579–4592, 1983.

Blumberg, A. F. and Mellor, G. L.: A description of a three-dimensional coastal ocean circulation model, Coastal and estuarine sciences, 4, 1–16, 1987.

Browne, S., Dongarra, J., Garner, N., Ho, G., and Mucci, P.: A portable programming interface for performance evaluation on modern processors, International Journal of High Performance Computing Applications, 14, 189–204, 2000.

Carpenter, I., Archibald, R., Evans, K. J., Larkin, J., Micikevicius, P., Norman, M., Rosinski, J., Schwarzmeier, J., and Taylor, M. A.: Progress towards accelerating HOMME on hybrid multi-core systems, International Journal of High Performance Computing Applications, 27, 335–347, 2013.

Chang, Y.-L. and Oey, L.-Y.: Instability of the North Pacific subtropical countercurrent, Journal of Physical Oceanography, 44, 818–833, 2014.

Chapman, B., Jost, G., and Van Der Pas, R.: Using OpenMP: portable shared memory parallel programming, vol. 10, The MIT Press, 2008.

Ezer, T. and Mellor, G. L.: A numerical study of the variability and the separation of the Gulf Stream, induced by surface atmospheric forcing and lateral boundary flows, Journal of physical oceanography, 22, 660–682, 1992.

Gopalakrishnan, S., Liu, Q., Marchok, T., Sheinin, D., Surgi, N., Tuleya, R., Yablonsky, R., and Zhang, X.: Hurricane Weather Research and Forecasting (HWRF) model scientific documentation, L Bernardet Ed, 75, 2010.

Gopalakrishnan, S., Liu, Q., Marchok, T., Sheinin, D., Surgi, N., Tong, M., Tallapragada, V., Tuleya, R., Yablonsky, R., and Zhang, X.: Hurricane Weather Research and Forecasting (HWRF) model: 2011 scientific documentation, L. Bernardet, Ed, 2011.

Govett, M., Middlecoff, J., and Henderson, T.: Running the NIM next-generation weather model on GPUs, in: Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on, pp. 792–796, IEEE, 2010.

Gropp, W. D., Lusk, E. L., and Thakur, R.: Using MPI-2: Advanced features of the message-passing interface, vol. 2, Globe Pequot, 1999.

Guo, X., Miyazawa, Y., and Yamagata, T.: The Kuroshio Onshore Intrusion along the Shelf Break of the East China Sea: The Origin of the Tsushima Warm Current., Journal of Physical Oceanography, 36, 2006.

Henderson, T., Middlecoff, J., Rosinski, J., Govett, M., and Madden, P.: Experience applying Fortran GPU compilers to numerical weather prediction, in: Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on, pp. 34–41, IEEE, 2011.

Huang, S.-M. and Oey, L.: Right-side cooling and phytoplankton bloom in the wake of a tropical cyclone, Journal of Geophysical Research: Oceans, 2015.

Huang, X., Wang, W., Fu, H., Yang, G., Wang, B., and Zhang, C.: A fast input/output library for high-resolution climate models, Geoscientific Model Development, 7, 93–103, 2014.

Isobe, A., Kako, S., Guo, X., and Takeoka, H.: Ensemble numerical forecasts of the sporadic Kuroshio water intrusion (kyucho) into shelf and coastal waters, Ocean Dynamics, 62, 633–644, 2012.

Jordi, A. and Wang, D.-P.: sbPOM: A parallel implementation of Princenton Ocean Model, Environmental Modelling & Software, 38, 59–61, 2012.

Kagimoto, T. and Yamagata, T.: Seasonal transport variations of the Kuroshio: An OGCM simulation, Journal of physical oceanography, 27, 403–418, 1997.

Korres, G., Hoteit, I., and Triantafyllou, G.: Data assimilation into a Princeton Ocean Model of the Mediterranean Sea using advanced Kalman filters, Journal of Marine Systems, 65, 84–104, 2007.

Kurihara, Y., Bender, M. A., Tuleya, R. E., and Ross, R. J.: Improvements in the GFDL hurricane prediction system, Monthly Weather Review, 123, 2791–2801, 1995.

Kurihara, Y., Tuleya, R. E., and Bender, M. A.: The GFDL hurricane prediction system and its performance in the 1995 hurricane season., Monthly weather review, 126, 1998.

Leutwyler, D., Fuhrer, O., Cumming, B., Lapillonne, X., Gysi, T., Lüthi, D., Osuna, C., and Schär, C.: Towards Cloud-Resolving European-Scale Climate Simulations using a fully GPU-enabled Prototype of the COSMO Regional Model, in: EGU General Assembly Conference Abstracts, vol. 16, p. 11914, 2014.

Lin, X., Xie, S.-P., Chen, X., and Xu, L.: A well-mixed warm water column in the central Bohai Sea in summer: Effects of tidal and surface wave mixing, Journal of Geophysical Research: Oceans (1978–2012), 111, 2006.

Linford, J. C., Michalakes, J., Vachharajani, M., and Sandu, A.: Multi-core acceleration of chemical kinetics for simulation and prediction, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, p. 7, ACM, 2009.

Mak, J., Choboter, P., and Lupo, C.: Numerical ocean modeling and simulation with CUDA, in: OCEANS 2011, pp. 1–6, IEEE, 2011.

McCalpin, J. and Wonnacott, D.: Time skewing: A value-based approach to optimizing for memory locality, Tech. rep., Technical Report DCS-TR-379, Department of Computer Science, Rugers University, 1999.

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

Miyazawa, Y., Zhang, R., Guo, X., Tamura, H., Ambe, D., Lee, J.-S., Okuno, A., Yoshinari, H., Setou, T., and Komatsu, K.: Water mass variability in the western North Pacific detected in a 15-year eddy resolving ocean reanalysis, Journal of oceanography, 65, 737–756, 2009.

Newberger, P. and Allen, J. S.: Forcing a three-dimensional, hydrostatic, primitive-equation model for application in the surf zone: 1. Formulation, Journal of Geophysical Research: Oceans (1978–2012), 112, 2007a.

Newberger, P. A. and Allen, J. S.: Forcing a three-dimensional, hydrostatic, primitive-equation model for application in the surf zone: 2. Application to DUCK94, Journal of Geophysical Research-Oceans, 112, 2007b.

NVIDIA: CUDA C Programming Guide Version 5.5, available at http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2014.

NVIDIA: CUDA C Best Practices Guide, available at http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#coalesced-access-to-global-memory, 2015.

Oey, L., Chang, Y.-L., Lin, Y.-C., Chang, M.-C., Xu, F.-H., and Lu, H.-F.: ATOP-the Advanced Taiwan Ocean Prediction System based on the mpiPOM Part 1: model descriptions, analyses and results, Terr Atmos Ocean Sci, 24, 2013.

Oey, L.-Y.: A wetting and drying scheme for POM, Ocean Modelling, 9, 133–150, 2005.

Oey, L.-Y.: Geophysical Fluid Modeling with the mpi version of the Princeton Ocean Model (mpiPOM). Lecture Notes, 70 pp, ftp://profs.princeton.edu/leo/lecture-notes/OceanAtmosModeling/Notes/GFModellingUsingMpiPOM.pdf, 2014.

Oey, L.-Y. and Chen, P.: A model simulation of circulation in the northeast Atlantic shelves and seas, Journal of Geophysical Research: Oceans (1978–2012), 97, 20 087–20 115, 1992a.

Oey, L.-Y. and Chen, P.: A nested-grid ocean model: With application to the simulation of meanders and eddies in the Norwegian Coastal Current, Journal of Geophysical Research: Oceans (1978–2012), 97, 20 063–20 086, 1992b.

Oey, L.-Y., Mellor, G. L., and Hires, R. I.: A three-dimensional simulation of the Hudson-Raritan estuary. Part I: Description of the model and model simulations, Journal of Physical Oceanography, 15, 1676–1692, 1985a.

Oey, L.-Y., Mellor, G. L., and Hires, R. I.: A three-dimensional simulation of the Hudson-Raritan estuary. Part II: Comparison with observation, Journal of Physical Oceanography, 15, 1693–1709, 1985b.

Oey, L.-Y., Mellor, G. L., and Hires, R. I.: A three-dimensional simulation of the Hudson-Raritan estuary. Part III: Salt flux analyses, Journal of physical oceanography, 15, 1711–1720, 1985c.

Oey, L.-Y., Lee, H.-C., and Schmitz, W. J.: Effects of winds and Caribbean eddies on the frequency of Loop Current eddy shedding: A numerical model study, Journal of Geophysical Research: Oceans (1978–2012), 108, 2003.

Potluri, S., Wang, H., Bureddy, D., Singh, A. K., Rosales, C., and Panda, D. K.: Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication, in: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, pp. 1848–1857, IEEE, 2012.

Shimokawabe, T., Aoki, T., Muroi, C., Ishida, J., Kawano, K., Endo, T., Nukada, A., Maruyama, N., and Matsuoka, S.: An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code, in: High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for, pp. 1–11, IEEE, 2010.

Siewertsen, E., Piwonski, J., and Slawig, T.: Porting marine ecosystem model spin-up using transport matrices to GPUs, Geoscientific Model Development Discussions, 5, 2179–2214, 2012.

Smolarkiewicz, P. K.: A fully multidimensional positive definite advection transport algorithm with small implicit diffusion, Journal of Computational Physics, 54, 325–362, 1984.

Sun, J., Oey, L., Xu, F., Lin, Y., Huang, S., and Chang, R.: The Influence of Ocean on Typhoon Nuri (2008), in: AGU Fall Meeting Abstracts, vol. 1, p. L3360, 2014.

Sun, J., Oey, L.-Y., Chang, R., Xu, F., and Huang, S.-M.: Ocean response to typhoon Nuri (2008) in western Pacific and South China Sea, Ocean Dynamics, 65, 735–749, 2015.

Varlamov, S. M., Guo, X., Miyama, T., Ichikawa, K., Waseda, T., and Miyazawa, Y.: M2 baroclinic tide variability modulated by the ocean circulation south of Japan, Journal of Geophysical Research: Oceans, 2015.

Volkov, V.: Better performance at lower occupancy, in: Proceedings of the GPU Technology Conference, GTC, vol. 10, 2010.

Wahib, M. and Maruyama, N.: Scalable kernel fusion for memory-bound GPU applications, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 191–202, IEEE Press, 2014.

Whitehead, N. and Fit-Florea, A.: Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs, rn (A+ B), 21, 1–1874919 424, 2011.

Williams, S., Waterman, A., and Patterson, D.: Roofline: an insightful visual performance model for multicore architectures, Communications of the ACM, 52, 65–76, 2009.

Wonnacott, D.: Using time skewing to eliminate idle time due to memory bandwidth and network limitations, in: Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International, pp. 171–180, IEEE, 2000.

Xu, F.-H. and Oey, L.-Y.: The origin of along-shelf pressure gradient in the Middle Atlantic Bight, Journal of Physical Oceanography, 41, 1720–1740, 2011.

Xu, F.-H. and Oey, L.-Y.: State analysis using the Local Ensemble Transform Kalman Filter (LETKF) and the three-layer circulation structure of the Luzon Strait and the South China Sea, Ocean Dynamics, 64, 905–923, 2014.

Xu, F.-H. and Oey, L.-Y.: Seasonal SSH variability of the Northern South China Sea, Journal of Physical Oceanography, 2015.

Xu, F.-H., Oey, L.-Y., Miyazawa, Y., and Hamilton, P.: Hindcasts and forecasts of Loop Current and eddies in the Gulf of Mexico using local ensemble transform Kalman filter and optimum-interpolation assimilation schemes, Ocean Modelling, 69, 22–38, 2013.

Yang, C., Xue, W., Fu, H., Gan, L., Li, L., Xu, Y., Lu, Y., Sun, J., Yang, G., and Zheng, W.: A peta-scalable CPU-GPU algorithm for global atmospheric simulations, in: Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 1–12, ACM, 2013.

Yin, X.-Q. and Oey, L.-Y.: Bred-ensemble ocean forecast of Loop Current and rings, Ocean Modelling, 17, 300–326, 2007.

Zavatarelli, M. and Mellor, G. L.: A numerical study of the Mediterranean Sea circulation, Journal of Physical Oceanography, 25, 1384–1414, 1995.

Zavatarelli, M. and Pinardi, N.: The Adriatic Sea modelling system: a nested approach, Annales Geophysicae, 21, 345–364, 10.5194/angeo-21-345-2003, 2003.

Zhenya, S., Haixing, L., Xiaoyan, L., et al.: The Applica tion of GPU in Ocean General Circulation Mode POP, Computer Applications and Software, 27, 27–29, 2010.

(R1)