Numerical Ocean Modeling and Simulation with CUDA

Jason Mak Paul Choboter Chris Lupo

***Abstract*—ROMS is software that models and simulates an ocean region using a finite difference grid and time stepping. ROMS simulations can take from hours to days to complete due to the compute-intensive nature of the software. As a result, the size and resolution of simulations are constrained by the performance limitations of modern computing hardware. To address these issues, the existing ROMS code can be run in parallel with either OpenMP or MPI. In this work, we implement a new parallelization of ROMS on a graphics processing unit (GPU) using CUDA Fortran. We exploit the massive parallelism offered by modern GPUs to gain a performance benefit at a lower cost and with less power. To test our implementation, we benchmark with idealistic marine conditions as well as real data collected from coastal waters near central California. Our implementation yields a speedup of up to 8x over a serial implementation and 2.5x over an OpenMP implementation, while demonstrating comparable performance to a MPI implementation.**

I. Introduction

The Regional Ocean Modeling System (ROMS) is a free-surface, primitive equation ocean model widely used by the scientific community for a diverse range of applications [1]. ROMS evolves the components of velocity, temperature, salinity and sea surface height forward in time using a finite-difference discretization of the equations of motion, with curvilinear coordinates in the horizontal and terrain-following coordinates in the vertical [2]. Sophisticated numerical techniques are employed, including a split-explicit time-stepping scheme that treats the fast barotropic (2D) and slow baroclinic (3D) modes separately for improved efficiency [3]. ROMS also contains a suite of data assimilation tools that allow the user to improve the accuracy of a simulation by incorporating observational data. These tools are based on four dimensional variational methods [4], which have a solid mathematical foundation and generate reliable results in practice, but require much more computational resources than a run without any assimilation of data.

The accuracy and performance of any ROMS simulation is limited by the resolution of the finite difference grid. Increasing the grid resolution increases the computational demands not only by the increase of the number of points in the domain, but also because numerical stability requirements demand a smaller time step on a finer grid. The ability to increase the grid size to obtain a finer-grain resolution and therefore a more accurate model is limited by the computing performance of hardware. This is particularly true when performing data assimilation, where the run-time can be orders of magnitude larger than non-assimilating runs, and where accuracy of a simulation is of paramount concern. In order for the numerical model to give a desired accuracy with reasonable performance, ROMS uses parallel computing to take advantage of its grid-based model. A shared memory model using OpenMP enables ROMS to take advantage of modern multi-core processors, and a distributed memory model using MPI provides access to the computing power of multi-node clusters.

Graphics processing units have steadily evolved from specialized rendering devices to general-purpose, massively parallel computing devices (GPGPU). Because of the relative low cost and power of GPUs, these devices have become an attractive alternative to large clusters for high-performance scientific computing. With support for a large amount of threads, an architecture optimized for arithmetic operations, and their inherent data-parallelism, GPUs are naturally suitable for the parallelization of large-scale scientific computing problems.

The success of GPUs as general-purpose computing devices owes not only to their hardware capabilities but also to their improved programmability. NVIDIA's Compute Unified Device Architecture (CUDA) framework has been instrumental in giving developers access to the resources available in GPUs [5]. Until recently, the language of the CUDA API was limited to C/C++. In 2009, the Portland Group (PGI) worked with NVIDIA to develop a CUDA Fortran compiler [6]. Because Fortran is popular in the scientific community, the release of this compiler opens up new avenues for parallelizing existing applications. Researchers have been successful in using the GPU to achieve speedup in a variety of Fortran applications, including grid-based models similar to ROMS. The Weather Research and Forecasting Model (WRF) is a mesoscale open-source application written in Fortran [7] that has been successfully coupled with ROMS [8]. Michalakes and Vachharajani rewrote a module in WRF as CUDA C and obtained speedup [9]. Others have taken a step further and accelerated WRF in its original language using CUDA Fortran [10][11]. In Michalakes' and Vachharajani's work, they explore issues concerning the use of GPUs for weather simulation, including the capabilities of GPUs, the ease of re-engineering existing code to work with CUDA, and the possible improvements to ease development and increase performance [9]. The availability of CUDA Fortran creates new avenues to explore these ideas.

In this work, we use a GPU to implement a new parallelization of ROMS. Using CUDA Fortran, we exploit the existing

grid partitioning routines and tile access schemes of ROMS' OpenMP implementation to map portions of a simulation onto a GPU. We target GPUs based on NVIDIA's new Fermi architecture, which features enhanced performance for double precision calculations. Our work focuses on the barotropic 2D stepping module that occupies a significant proportion of time in a simulation of a non-linear model. We discuss the existing parallelization options and their tradeoffs. We test our CUDA implementation on both idealized and realistic data and assess its performance against that of OpenMP and MPI.

II. ROMS

ROMS is open-source software that is actively developed by a large community of programmers and scientists. With over 400,000 lines of Fortran code, ROMS employs various equations in its numerical model [1]. ROMS models an ocean region and its conditions as staggered, finite difference grids in the horizontal and vertical. To simulate changing conditions, ROMS uses split-explicit time-stepping in which a large time step (baroclinic) computes 3D equations and iterates through a sequence of smaller time steps (barotropic) that compute depth integrated 2D equations (*step2D*) [12]. Users determine the length and computational intensity of a simulation by setting the resolution of the grid, the number of large time steps (*NTIMES*), the amount of real time each step represents (*DT*), and the amount of small discrete time steps (*NDTFAST*) per large time step (with a minimum number to maintain numerical stability). When translated into source code, ROMS begins a simulation by entering a loop that iterates through *NTIMES* large time steps, where the large time step is represented by a function named *main3D*. This function solves the 3D equations for the finite difference grid and calls the small time step *NDTFAST* times. Because the 3D and 2D equations are both applied to the entire grid, they serve as the targets of parallelization.

III. Parallelization

To run in parallel, ROMS partitions the grid into tiles [12]. Users enter the number of tiles in the *I* direction and the *J* direction. For example, setting *NtileI* to 4 and *NtileJ* to 2 results in a partitioned grid of tiles that consists of 2 rows and 4 columns for a total of 8 tiles. The computations in ROMS applies to each grid point, so tiles can be assigned to separate processors. ROMS can be run in parallel with either OpenMP or MPI, an option that is selected at compile time [12]. Currently, ROMS does not support using both options at once. Each paradigm is discussed in the next sections.

A. OpenMP

OpenMP is a parallelization model designed for shared memory systems. For most modern hardware, this refers to multi-core processors on a single machine. In both its C and Fortran specifications, OpenMP requires programmers to insert directives around *for loops* (or *do loops* in Fortran). The OpenMP library automatically assigns different iterations of a loop to multiple threads, which are then divided among multiple processors. In ROMS, the *for loops* of interest are located in *main3D*, where the 3D equations are solved. Each function is applied to the entire grid by looping over the partitioned tiles. Fig. 1(a) shows OpenMP directives applied to a loop in ROMS. Modern multi-core processors are fast and can perform computations over each tile quickly. However, the ideal minimum tile size in the OpenMP implementation is determined by the number of processor cores. Therefore, the parallelism offered by OpenMP is coarse-grained and may have difficulty scaling for larger problem sets.

TABLE I
Runtime profiling data of a short ROMS simulation

Function	Runtime (s)	Percentage
2D stepping	614	53.3%
GLS vertical mixing parameterization	261	22.6%
Harmonic mixing of tracers, geopotentials	43	3.7%
3D equations predictor step	41	3.5%
Corrector time-step for tracers	38	3.3%
Corrector time-step for 3D momentum	36	3.1%
Pressure gradient	35	3.0%
3D equations right-side terms	34	2.9%
Equation of state for seawater	25	2.2%
Other	26	2.3%

B. MPI

Message Passing Interface (MPI) is a parallelization model used for distributed memory systems. This paradigm commonly targets multiple machines operating in a networked cluster. In ROMS, this paradigm enables a simulation to be parallelized with an arbitrarily large number of processors. As its name implies, MPI uses message passing to facilitate memory management across several machines. The drawback of this distributed model occurs when different processes require data sharing during computation, and network transfers incur overhead. In ROMS, MPI differs from OpenMP because each partitioned tile is sent to a different machine and computed as its own process [12]. Computations require tiles to use overlapping boundary points, or halo points, from neighboring tiles [12]. In a shared memory model, these halo points can be accessed in a straightforward manner. MPI, however, requires message passing to retrieve the halo points, which adds network transfers to the cost of computation.

C. CUDA

GPUs have evolved from graphics accelerators to general purpose compute units that provide massive parallelism with their large number of cores. GPUs are designed for data parallelism, where the same instructions are executed simultaneously on multiple data. In addition, the graphics legacy of GPUs enable them to be well optimized for arithmetic operations. Therefore, GPUs are well suited for performing mathematical computations on large grid-based structures. To provide high level access to the hardware resources of its GPUs, NVIDIA provides the CUDA framework, an API based on C/C++ [5]. Like OpenMP, the parallelization abstractions of CUDA target a shared memory architecture. However, CUDA differs from OpenMP in that it requires a large number of

```
DO my_iif=1,nfast(ng)+1
[...]
!$OMP PARALLEL DO
  DO thread=0,numthreads-1
    subs=numtiles/numthreads
    DO tile=subs*thread,subs*(thread+1)-1,+1
      CALL step2d (ng, tile)
    END DO
  END DO
!$OMP END PARALLEL DO
[...]
END DO
```

(a) OpenMP

```
CALL step2d_host_to_device()
DO my_iif=1,nfast(ng)+1
[...]
  CALL step2d_kernel<<<dim_grid, dim_block>>>
   (num_tiles,krhs(ng),kstp(ng),knew(ng),
    nstp(ng), nnew(ng),PREDICTOR_2D_STEP(ng),
    iif(ng), Lm(ng), Mm(ng), iic(ng),
    nfast(ng),dtfast(ng), ntfirst(ng),
    gamma2(ng), rho0, work_dev)
[...]
END DO
CALL step2d_device_to_host()
```

(b) CUDA

Fig. 1. OpenMP and CUDA parallelization

threads for optimal performance [5]. Unlike multi-threaded environments on the CPU, context switching on the GPU bears little cost. Therefore, GPUs facilitate large numbers of threads and have better support for fine-grained parallelism.

Recently available technologies in both hardware and software motivate us to pursue a CUDA implementation of ROMS. In hardware, NVIDIA released a new generation of graphics cards in 2010 based on an architecture named Fermi. Fermi GPUs feature greater numbers of cores than previous models and also have enhanced performance for double precision calculations [13]. As a result, these compute cards have the potential to enhance scientific computing applications like ROMS, which require numerical precision. In software, PGI ported the CUDA API into Fortran and released the CUDA Fortran compiler in 2009. The compiler allows us to develop our CUDA implementation directly in Fortran and to integrate our work with the existing ROMS code.

1) Approach: In a typical CUDA application, the programmer allocates memory on the GPU, copies the relevant data onto GPU memory, performs the compute intensive function (a CUDA kernel) on the data, and copies the results back to CPU memory [5]. We isolate a compute intensive portion of ROMS to rewrite as a CUDA kernel that will run on the GPU. We determine that the depth integrated 2D equations occupy over 50% of the total runtime in a simulation. The 2D equations are computed as a function in the short time step, which is called many times in the large time step. Therefore, *step2D* is one of the most frequently called functions in a simulation. Table I shows various simulation functions and the percentage of runtime they contribute to a short, serial run of ROMS.

In our approach, we rewrite the *step2D* function as a CUDA kernel that is launched in place of the function call. First, we determine the variables that are used by *step2D* and allocate appropriate space for them on the GPU. For *step2D*, memory allocation needs to occur only once at the beginning of the program. This may change in future work when more functions are ported onto the GPU, because the GPU's memory may not be large enough to contain the data used by all functions. Array variables are stored in global memory, the largest but slowest memory on the GPU. We pass scalar variables as arguments to the CUDA kernel so that they will be stored in faster memory locations including registers and cache. Since multiple iterations of *step2D* are called per large time step, copying variable data between GPU memory and CPU memory needs to take place only before and after the iterations.

As a key step in any CUDA application, we must partition the problem into subdomains that can be assigned to separate GPU threads. For the task of dividing the model grid, we save coding effort by taking advantage of the tile partitioning used in the existing parallel implementations of ROMS. Because both OpenMP and CUDA target a shared memory model, we deduce that they can manage and use the tiles in similar ways. In ROMS, the tiling abstraction used by *step2D* and other functions is implemented via tile numbers that are passed as arguments. Within *step2D*, the tile number is mapped to a set of boundaries, which define the tile or subset of the grid. To apply a function to every tile, the program loops through the tile numbers and calls the function for each tile number. To parallelize the loops and process the tiles concurrently, the OpenMP implementation encloses the loops with directives. In our GPU implementation, we assign each tile to an individual CUDA thread. Each CUDA thread has a self-identifying ID that can be retrieved from within the kernel. Instead of passing a tile number as an argument, we use each thread's unique ID as the tile number. Fig. 1 shows the OpenMP parallelization of *step2D* and the equivalent CUDA code.

To test the correctness of our implementation, we use a small simulation to compare our output with that of the OpenMP implementation. During a simulation, ROMS prints out diagnostics messages at each large time step, which can be used for basic testing. For more extensive verification, we use the main simulation data that stores the values of grid variables at specific times. To output this large set of data, ROMS periodically writes to files in the NetCDF format at a frequency specified by the user [12]. We dump the contents of these NetCDF files into ASCII files to use for comparison. The contents produced by our CUDA implementation matches that of the OpenMP implementation by approximately 82%. We found that the differences were due to the order in which OpenMP processes tiles, from higher numbered to lowered numbered tiles. When we switched the

loop ordering to process tiles in ascending order, the outputs from CUDA and OpenMP were identical. To address this issue of thread ordering, additional work is needed to explore GPU synchronization for ROMS.

Because the design of ROMS was not intended for CUDA, we found that porting a function into CUDA and integrating it into the code proved to be difficult tasks. The *step2D* function is over 2,000 lines long, which meant the equivalent CUDA kernel would be roughly the same length. Although the majority of the original code is directly reused in the kernel, any changes to the code to make the CUDA implementation work (such as the renaming of variables) must be propagated throughout the function. In addition, the *step2D* function has over 50 parameters, and the total size of these parameters are greater than the 256 KB limit allowed for CUDA kernel arguments. To work around this limitation, we take advantage of a feature in the CUDA Fortran compiler that allows the GPU to access device variables outside the kernel but within the same Fortran module. We change many of the parameters in *step2D* to module variables. We also encountered challenges relating to encapsulation. Many module variables in ROMS are accessible from any file that imports the module. Because *step2D* has a length of over 2,000 lines, the function uses module variables from a large number of files. Since GPU memory is separate from CPU memory, all variables used by the function (and therefore the CUDA kernel) must be identified and copied to GPU memory. These variables are often spread across various modules of the program. This lack of encapsulation may need to be considered in future applications of CUDA in ROMS.

Another feature of the Fortran language that posed a challenge for our CUDA implementation is the ability for programmers to determine the way arrays are indexed. In the OpenMP parallelization of *step2D*, threads index local arrays differently depending on the tile they were assigned. For example, the following is a possible mapping of tiles to index ranges for an array of size *n*: *tile 1* to [*-2...n-3*], *tile 2* to [*1...n*], *tile 3* to [*4...n+3*], and so on. In Fortran, the index ranges of array variables must be specified at the beginning of the function in the variables' declarations as local variables or dummy arguments. One possible solution is to declare the arrays with a common indexing and rewrite all array accesses to assume this common indexing. To save time, we instead collapse all local arrays into one (to save kernel parameter space) and store it in global memory. We pass a pointer to the array as a single argument to the kernel, so that we can declare it as a dummy argument inside the kernel. In the declaration of the dummy argument, we use the thread ID (tile number) to specify the array indices. By doing so, we maintain the existing array accesses that expect different threads to use different indexing. As this solution indicates, many of our approaches are designed to save effort in coding.

IV. Results

To evaluate the performance of our CUDA implementation, we compare it to the existing serial, OpenMP, and MPI implementations running on high performance hardware. For the serial runs, we use an Athlon II processor clocked at 2.9 GHz. For OpenMP, we use 2 Intel Xeon E5504 processors, priced at approximately $450, for a total of 8 cores. To run ROMS with MPI, we use a cluster of Intel Xeon 5130 processors that have a total of 64 cores. The price of this cluster easily exceeds $10,000. To test our CUDA implementation, we target the GTX 470, a commodity GPU priced at approximately $300. This card, based on the Fermi architecture, features 1280 MB of memory and 448 cores clocked at 1.22 GHz [14]. In theory, the GTX 470 requires 21,504 threads for full saturation.

TABLE II
THE UPWELLING CASE IS USED TO TEST STEP2D WITH DIFFERENT GRID SIZES. THE TOTAL RUNTIME IN SECONDS IS OF 60 STEP2D ITERATIONS.

Grid size	Serial	OpenMP	MPI	CUDA
256 x 128	3.01	0.82	0.15	0.42
384 x 192	6.64	1.92	0.26	0.80
512 x 256	11.31	3.52	0.53	1.45

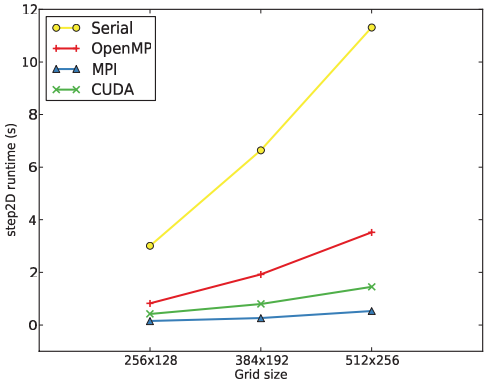

Fig. 2. Upwelling benchmark with various grid sizes.

In addition to CUDA, we use PGI's Fortran compiler to compile ROMS to run in serial and with OpenMP. We also apply the -O3, -fastsse, and -Mipa=fast optimization flags. The MPI implementation is compiled with gfortran using the -O3, -frepack-arrays, and -ffast-math flags. When choosing tiling layouts, we use the same number of tiles as there are processors for the non-CUDA implementations. Because our CUDA implementation maps each tile to a separate thread, we set as many tiles as possible to saturate the GPU. Therefore, the ideal tile size for the GPU is 2 x 2, the smallest allowed in ROMS. In our second benchmark, we are unable to use the smallest tile size due to the GPU running out of memory and instead, we use 4 x 2 tiles. This limitation suggests that more powerful GPUs may be needed for larger grid sizes.

A. Benchmarks

The upwelling ocean model, featuring a periodic channel with shelves on each side, is a basic example used to test ROMS [1]. The upwelling case uses internal, analytic functions to create idealized conditions, which enables us to easily modify its simulation parameters. We take advantage of this by

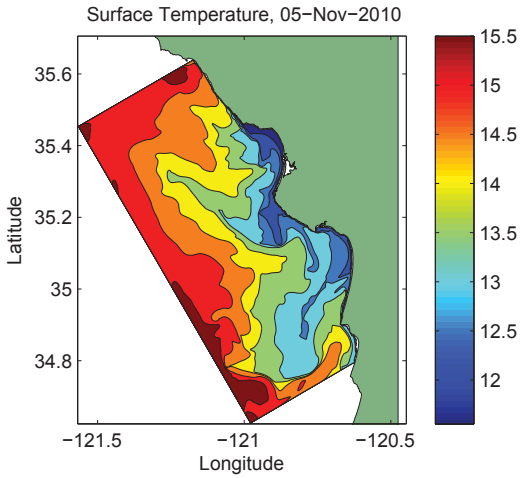

Fig. 3. Ocean surface temperature of a realistic ROMS simulation.

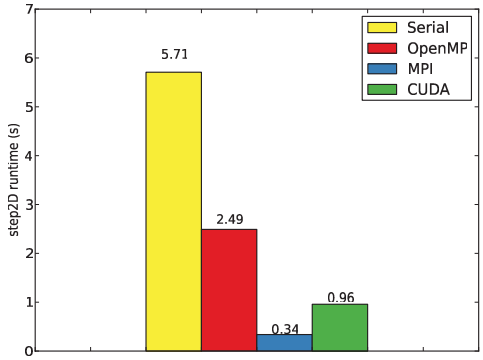

Fig. 4. Comparison of step2D runtime in a realistic benchmark.

using the test case to benchmark the different implementations of *step2D* with increasing grid sizes. When testing CUDA, we include the transfer time between GPU and CPU memory. Because not all functions have been ported to the GPU, the functions on the CPU cannot proceed without the processed data from the GPU and vice versa, so the costs of these transfers are unavoidable. As Fig. 2 shows, the performance of the GTX 470 falls between the multi-core processors that run OpenMP and the MPI-enabled cluster. As the grids grow in size, the performance gaps also increase. With the 512 x 256 grid, the CUDA implementation of *step2D* is nearly 8x faster than the serial implementation and 2.5x faster than the OpenMP implementation.

In addition to analytic functions, ROMS also allows users to use externally collected data to establish the conditions of a simulation, including initial conditions, boundary conditions, and wind forcing. To test CUDA on a more realistic benchmark, we use observational data collected by autonomous underwater vehicles from waters near California's central coast. This data had been collected to simulate the state of an ocean region for the time period of 1–7 November 2010. The simulation is configured to represent a rectangular domain 68 km wide and 170 km long, on a 256 x 512 grid, with approximately 250 m between grid points. The model is also forced with NOAA/NCDC blended 6-hourly 0.25-degree sea surface wind stress [15]. Initial and boundary conditions are taken from the Hybrid Coordinate Ocean Model (HYCOM) 1/12 degree global hindcast reanalysis [16]. The simulation exhibits realistic features typically seen along the central California coastal ocean, including cold upwelled water next to the coast. Fig. 3 shows a graphical depiction of the ocean region's temperature properties. Although this benchmark is more realistic than upwelling, Fig. 4 indicates that the performance characteristics are roughly the same. Once again, the CUDA implementation demonstrates a 2.5x speedup over OpenMP, and despite being a much less costly device, the GPU's performance remains close to that of the MPI cluster.

V. Future Work

Our work leaves many opportunities to increase the performance of ROMS and demonstrate the power of GPUs. Because the *step2D* kernel is over 2,000 lines long, many possible optimizations remain including loop unrolling and divergence removal. We have not attempted such optimizations in our work. Shared memory is another promising optimization found in various CUDA applications. Because GPUs are often limited by memory latency, and shared memory acts as a cache for slower global memory, there is great potential for additional speedup [5]. In addition to kernel-level optimizations, we can investigate other parallelization models involving CUDA. We can explore the use of multiple GPUs, which would enable us to further divide the grid and have each GPU process a smaller piece. We can also combine the use of CUDA with MPI by implementing a model that uses a cluster of GPUs. Integrating the two implementations would enable an MPI process to use a GPU to perform the heavy computations on the tile assigned to the process. Such a solution may be scalable for very large grid sizes. Our work focuses specifically on the *step2D* function that occupies a large percentage of the runtime in a simulation. Although we succeeded in speeding up the function, we can improve overall performance further by running more functions on the GPU. Because we reused the existing tile partitioning in ROMS to convert OpenMP loops to CUDA code for *step2D*, all computations of ROMS that are parallelized with OpenMP can be rewritten in a similar fashion to run on the GPU. Therefore, it is possible to have the majority of a ROMS simulation run entirely on the GPU.

VI. Conclusion

We accelerate a compute intensive portion of the ocean modeling software, ROMS, using a GPU and CUDA Fortran. Our work is motivated by the limitations on grid sizes and accuracy caused by the increased runtimes of simulations. To show the applicability of our work on real-world simulations, we benchmark our implementation on observational data collected from an ocean region. We discuss the approaches we took and challenges we faced in integrating CUDA into an

existing Fortran project. We demonstrate that the use of GPUs for massive fine grained parallelism in ocean modeling can yield comparable performance to that of multi-core systems and multi-node clusters. Furthermore, commodity GPUs can be found in the consumer market at a fraction of the cost of a cluster. These GPUs can run on a single machine, require relatively little maintenance, and consume much less power. GPUs continue to evolve with increasing numbers of cores, higher clock speeds, larger memory sizes, and improved programmability. We have shown the potential of these devices in enhancing the performance of ROMS and similar large scale ocean modeling applications. We believe these devices can aid scientists by removing the performance bottlenecks that limit the accuracy and scale of ocean simulations.

ACKNOWLEDGMENT

The authors would like to thank NVIDIA for equipment donations.

REFERENCES

[1] ROMS. [Online]. Available: http://www.myroms.org

[2] D. B. Haidvogel, H. Arango, W. P. Budgell, B. D. Cornuelle, E. Curchitser, E. Di Lorenzo, K. Fennel, W. R. Geyer, A. J. Hermann, L. Lanerolle, J. Levin, J. C. McWilliams, A. J. Miller, A. M. Moore, T. M. Powell, A. F. Shchepetkin, C. R. Sherwood, R. P. Signell, J. C. Warner, and J. Wilkin, "Ocean forecasting in terrain-following coordinates: Formulation and skill assessment of the Regional Ocean Modeling System," *J. Comput. Phys.*, vol. 227, pp. 3595–3624, March 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1347465.1347771

[3] A. F. Shchepetkin and J. C. McWilliams, "The Regional Oceanic Modeling System (ROMS): A split-explicit, free-surface, topography-following-coordinate oceanic model," *Ocean Modelling*, vol. 9, no. 4, pp. 347 – 404, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1463500304000484

[4] E. D. Lorenzo, A. M. Moore, H. G. Arango, B. D. Cornuelle, A. J. Miller, B. Powell, B. S. Chua, and A. F. Bennett, "Weak and strong constraint data assimilation in the inverse Regional Ocean Modeling System (ROMS): Development and application for a baroclinic coastal upwelling system," *Ocean Modelling*, vol. 16, no. 3-4, pp. 160 – 187, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1463500306000916

[5] "CUDA programming guide version 3.0," NVIDIA, 2010.

[6] PGI CUDA Fortran compiler. [Online]. Available: http://www.pgroup.com/resources/cudafortran.htm

[7] Weather research and forecasting model. [Online]. Available: http://www.wrf-model.org/index.php

[8] M. Li, J. Hsieh, R. Saravanan, P. Chang, and H. Seidel, "Atlantic hurricanes using a coupled regional climate model." [Online]. Available: http://sc.tamu.edu/research/chang/new/

[9] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1 – 7.

[10] G. Ruetsch, E. Phillips, and M. Fatica, "GPU acceleration of the long-wave rapid radiative transfer model in WRF using CUDA Fortran," NVIDIA. [Online]. Available: http://www.pgroup.com/lit/articles/nvidia_paper_rrtm.pdf

[11] M. Wolfe and C. Toepfer, "Porting the WRF WSM52d kernel to GPUs using PGI Accelerator Fortran," The Portland Group, Oct. 2009. [Online]. Available: http://www.pgroup.com/lit/articles/insider/v1n3a1.htm

[12] K. S. Hedström, "Technical manual for a coupled sea-ice/ocean circulation model (version 3)," U.S. Department of the Interior Minerals Management Service, 2010.

[13] "NVIDIA's next generation CUDA compute architecture: Fermi," White Paper, NVIDIA, 2009.

[14] GeForce GTX 470. [Online]. Available: http://www.nvidia.com/object/product_geforce_gtx_470_us.html

[15] Blended sea winds. [Online]. Available: http://www.ncdc.noaa.gov/oa/rsad/air-sea/seawinds.html

[16] Hybrid coordinate ocean model. [Online]. Available: http://www.hycom.org