High Performance Regional Ocean Modeling with GPU Acceleration

Ian Panzer*, Spencer Lines*, Jason Mak†, Paul Choboter*, Chris Lupo*
*California Polytechnic State University, †UC Davis
Email: {ipanzer,slines,pchobote,clupo}@calpoly.edu, jwmak@ucdavis.edu

***Abstract*—The *Regional Ocean Modeling System* (ROMS) is an open-source, free-surface, primitive equation ocean model used by the scientific community for a diverse range of applications [1]. ROMS employs sophisticated numerical techniques, including a split-explicit time-stepping scheme that treats the fast barotropic (2D) and slow baroclinic (3D) modes separately for improved efficiency [2]. 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 [3], which generate reliable results, but require more computational resources than without any assimilation of data.**

The implementation of ROMS supports two parallel computing models; a distributed memory model that utilizes Message Passing Interface (MPI), and a shared memory model that utilizes OpenMP. Prior research has shown that portions of ROMS can also be executed on a General Purpose Graphics Processing Unit (GPGPU) to take advantage of the massively parallel architecture available on those systems [4].

This paper presents a comparison between two forms of parallelism. NVIDIA Kepler K20X GPUs were used for performance measurement of GPU parallelism using CUDA while an Intel Xeon E5-2650 was used for shared memory parallelism using OpenMP. The implementation is benchmarked using idealistic marine conditions. Our experiments show that OpenMP was the fastest, followed closely by CUDA, while the normal serial version was considerably slower.

I. Introduction

The *Regional Ocean Modeling System* (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.

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, parallel implementations of ROMS currently exist 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 Message Passing Interface (MPI) provides access to the computing power of multi-node clusters. Recent research has also shown some success with an implementation using NVIDIA's CUDA framework for a *Graphics Processing Unit* (GPU) [4].

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.

Modern GPUs are well suited to high-performance compute applications. They allow general purpose computation in a massively parallel fashion. Support for fast double-precision calculations and error correcting memories mean that calculations that require high degrees of accuracy can run with performance that can exceed shared memory implementations on sequential or even multi-core CPU architectures. A mature software development environment including compilers, debuggers and profilers allows developers to optimize their code for acceleration on a GPU in a manner similar to that of code running on CPU systems.

This work extends the prior ROMS GPU implementation to utilize features of NVIDIA's newest Kepler architecture [5] in a heterogeneous high performance compute solution. The Kepler architecture has more memory and a larger number of compute cores available than prior GPU offerings. Multiple GPUs are used in addition to multiple parallel threads executing on a traditional CPU. The implementation uses NVIDIA's *Compute Unified Device Architecture* (CUDA) framework [6] and the Accelerated Fortran compiler developed by The Portland Group (PGI) [7]. Like prior work, this implementation focuses on the barotropic 2D stepping that occupies a significant proportion of time in a simulation. Simulations are run on a Kepler K20X compute card as well as the previous generation Fermi C2050 GPU.

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 numerically integrates the Reynolds-averaged Navier-Stokes equations under the hydrostatic and Boussinesq approximations [1],

$$\frac{\partial u}{\partial t} + \mathbf{u} \cdot \nabla u - fv = -\frac{\partial \phi}{\partial x} - \frac{\partial}{\partial z}\left(\overline{u'w'} - \nu \frac{\partial u}{\partial z}\right) + \mathcal{F}_u + \mathcal{D}_u$$

$$\frac{\partial v}{\partial t} + \mathbf{u} \cdot \nabla v + fu = -\frac{\partial \phi}{\partial y} - \frac{\partial}{\partial z}\left(\overline{v'w'} - \nu \frac{\partial v}{\partial z}\right) + \mathcal{F}_v + \mathcal{D}_v$$

$$\frac{\partial C}{\partial t} + \mathbf{u} \cdot \nabla C = -\frac{\partial}{\partial z}\left(\overline{C'w'} - \nu_\theta \frac{\partial C}{\partial z}\right) \mathcal{F}_C + \mathcal{D}_C$$

$$\rho = \rho(T, S, P),$$

$$\frac{\partial \phi}{\partial z} = \frac{-\rho g}{\rho_0},$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0,$$

which correspond, respectively, to the conservation of momentum in the x- and y-directions, time evolution of a scalar concentration field $C(x, y, z, t)$(temperature or salinity), the equation of state, the hydrostatic equation, and the continuity equation.

ROMS models an ocean region and its conditions as finite difference grids in the horizontal and vertical. For efficient time discretization of both the fast free-surface waves and slower internal motions, 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`) [8]. 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 per large time step (*NDTFAST*). 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 [8] 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 are applied 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 [8]. 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. 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. Figure 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 sizes in the OpenMP implementation are limited by the number of processor cores. Therefore, the parallelism offered by OpenMP is coarse-grained and may have difficulty scaling for larger problem sets.

B. MPI

Message Passing Interface (MPI) is a parallelization model used for distributed memory systems. The targeted hardware of this paradigm are 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 [8]. Computations require tiles to use grid points from neighboring tiles. In a shared memory model, these “ghost points” can be accessed in a straightforward manner. MPI, however, requires message passing to retrieve the ghost points, which adds network transfers to the cost of computation. The experiments presented in this paper do not use MPI, as all experiments are executed on a single compute system.

C. CUDA

Figure 1 shows the previous implementation of the ROMS call to the `step2D` function in CUDA. The experiments in this paper also use this approach. The profiling data for ROMS is shown in Table III, where it is clear to see that the `step2D` occupies the largest percentage of the ROMS runtime.

IV. Approach

After the initial setup of ROMS, ROMS was compiled to use OpenMP and tested. The results showed a significant speedup over previous serial runtimes. Some fine-tuning of the input parameters allowed for slightly more gains. When attempting to compile using CUDA, there were some difficulties at first,

```
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

TABLE I: Runtime profiling data of a ROMS simulation.

Function	Runtime (sec)	Percentage
2D stepping	614	53.3%
GLS vertical mixing parameterization	261	22.6%
Harmonic mixing of tracers	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%

which resulted in the acquisition of erronous data. After overcoming those challenges, the results closely paralleled those of OpenMP. The parameters of the input were also varied in order to find a more optimal solution. Finally, the CUDA version was compiled for a different architecture and the results were compared.

V. Results

A. Hardware

ROMS running in serial is compared with OpenMP and with CUDA runnign one a single GPU. The CPU used for all experiments is an Intel Xeon E5-2650 processor with 64 GB of RAM. The CUDA implementation of ROMS uses Kepler K20X GPUs. These cards feature 6 GB of memory and 2688 cores clocked at 732 MHz [5]. These cards support up to 28672 threads running concurrently.

B. Simulations

The upwelling example was contributed by Anthony Macks and Jason Middleton and consists of a periodic channel with shelves on each side [1]. There is along-channel wind forcing and the Coriolis term leads to upwelling on one side and downwelling on the other side. The upwelling case is idealized and enables us to easily modify our simulation parameters. Therefore, we appropriately use the example to test the performance of the parallel implementations of step2D using different grid sizes.

TABLE II: Runtime of ROMS Simulations using Various Compute Techniques.

Technique	Runtime (sec)	Percentage of Serial Runtime
Serial	6700	100.0%
CUDA on Fermi	130	1.95%
CUDA on Kepler	117	1.75%
OpenMP	35	0.522%

C. Summary

Considering overall runtime as compared to the serial version, OpenMP was slightly faster than CUDA, by 1.45% and 1.25% compared to the Fermi and Kepler architectures respectively, and both were much faster the the serial version, OpenMP by 99.5% and CUDA be 98.15%. When compiled with CUDA, ROMS ran faster on the newer Kepler architecture than on the the Fermi architecture by 0.1%.

TABLE III: Step2d Runtime for Various Compute Techniques.

Technique	Percentage of Runtime	Percentage vs Serial
Serial	82.8%	100%
OpenMP	55.9%	67.6%
CUDA on Fermi	20.0%	24.1%
CUDA on Kepler	12.4%	15.0%

However, most of the optimizations occur in the `step2D` function. It is useful to see how much time each version of ROMS spends in this function, to compare the efficiency of the various optimizations. The serial version spent 82% of the runtime in the `step2D` function. When using OpenMP, ROMS only spent 55% of the total runtime in `step2D`, a 33% improvement. CUDA takes this optimization even further, with the Fermi architecture spending 24% of runtime in `step2D` and Kepler spending 15%. The alterations

VI. Future Work

This work leaves several opportunities to increase the performance of ROMS and demonstrate the power of GPUs. Because the `step2D` kernel is over 2000 lines long, many possible optimizations remain including loop unrolling and divergence removal. 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 [6].

In addition to kernel-level optimizations, other parallelization models involving CUDA should be investigated. The use of a larger number of GPUs should be explored, which would enable further subdivision of the ROMS grid and have each GPU process a smaller piece.

The use of CUDA with MPI should also be investigated by implementing a model that uses a cluster of servers with GPUs. Combining the two implementations would enable an MPI process to use a GPU to perform the heavy computations on the ROMS tile assigned to the process. This solution may be scalable for very large grid sizes. The work in this paper focuses specifically on the shared memory implementation of the `step2D` function that occupies a large percentage of the runtime in a simulation. Overall performance may be further improved by running more functions on the GPU. Because the existing tile partitioning in ROMS was reused 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.

VII. Conclusion

This paper presents a comparison between two types of parallelism, OpenMP and CUDA as applied to the ocean modeling software, ROMS, using both CPU and GPU based models. The work is motivated by the limitations on grid sizes and accuracy caused by the increased runtimes of simulations. The challenges faced in craeting a stable working environment and obtaining valid data are discussed. The results demonstrate that both OpenMP and CUDA have great potential for drastically decreasing the runtime of this and other simulations. The advantage of using either of these methods is that the hardware is relatively inexpansive and the gains are immense. As both GPUs and CPUs continue to evolve, the interoperability between them will continue to benefit the ROMS community. This work shows the potential of these devices used as a substitue for traditionally more expensive methods.

References

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

[2] 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

[3] 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

[4] J. Mak, P. Choboter, and C. Lupo, "Numerical ocean modeling and simulation with CUDA," in *OCEANS 2011, MTS/IEEE KONA - Oceans of Opportunity: International cooperation and partnership across the Pacific*, September 2011.

[5] NVIDIA Kepler compute architecture. [Online]. Available: http://www.nvidia.com/object/nvidia-kepler.html

[6] "CUDA parallel programming and compute platform," NVIDIA, 2013. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

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

[8] 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.