# Performance Assessment of A Multi-block Incompressible Navier-Stokes Solver using Directive-based GPU Programming in a Cluster Environment

Lixiang Luo[†], Jack R. Edwards[†], Hong Luo[†], Frank Mueller[‡]

December 17, 2013

[†]Department of Mechanical and Aerospace Engineering
[‡]Department of Computer Science

North Carolina State University

**Abstract**

OpenACC, a directive-based GPU programing standard, is emerging as a promising technology for massively-parallel accelerators, such as General-purpose computing on graphics processing units (GPGPU), Accelerated Processing Unit (APU) and Many Integrated Core Architecture (MIC). The heterogeneous nature of these accelerators call for careful designs of parallel algorithms and data management, which imposes a great hurdle for general scientific computation. Similar to OpenMP, the directive-based approach of OpenACC hides many underlying implementation details, thus significantly reduces the programming complexity and increases code portability. However, many challenges remain, due to the very high granularity of GPGPU and the relatively narrow interconnection bandwidth among GPUs, which is particularly restrictive when cross-node data exchange is involved in a cluster environment. In our study, a multi-block incompressible Navier-Stokes solver is ported for GPGPU using OpenACC and MVAPICH2. A performance analysis is carried out based on the profiling of this solver running in a InfiniBand cluster with nVidia GPUs, which helps to identify the potentials of directive-based GPU programming and directions for further improvement.

## 1 Introduction

CFD is a typical computation-intensive task, which has always been constrained by available computer processing capability. It is a constant objective of CFD scholars to achieve better results and performance on limited resources. Recently, General Purpose Graphics Processing Unit (GPGPU) emerges as a promising technology for large-scale parallel computation. GPU is particularly attractive due to its potential of one or two magnitudes of performance improvement with relatively low capital investment. Due to early hardware implementation for graphics processing, GPU used to have poor double-precision performance. For this

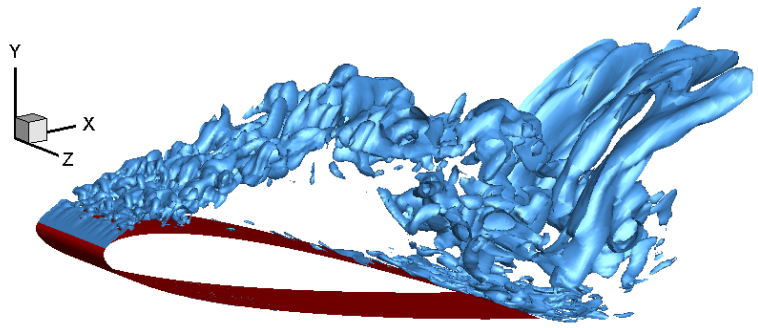


Figure 1: Lambda-2 isosurfaces of a dynamic stall simulation. Re$\sim 10^6$, $8 \times 10^6$ cells.

reason the CFD community had been reluctant to adopt GPU for large-scale, practical applications. Furthermore, the lack of a mature GPU-capable Fortran compiler makes migrating legacy codes particularly difficult. Recent GPU hardware has seen significant improvement on double-precision performance. Meanwhile, Fortran compilers have become capable enough in GPU programming to be considered for production development.

However, GPU programming for production-level CFD applications still remain a challenging subject, due the very different architectures of GPU and CPU and the notorious bottleneck of GPU-CPU memory bandwidth. Further contributing to the difficulty is the nearly inevitable use of Message Passing Interface (MPI) in a cluster environment. Not only the data size of the problem often exceed the memory limit of one GPU, often the computation complexity also calls for the problem to be partitioned to allow better execution speeds. As the inter-node data exchange has always been an overall performance limitation of MPI programs, the GPU-CPU bandwidth bottleneck only adds to the difficulties of porting legacy CFD codes. Recently, several MPI implementations become GPGPU-aware, including Open MPI [1] and MVAPICH2 [2]. However, the underlying limitations on bandwidth and data packing remain.

Our study attempts to port a 3D incompressible Navier-Stokes (N-S) solver onto the GPU architecture. It is based on an existing solver validated for a range of model problems [3, 4]. For example, the result of an LES dynamic stall simulation is given in Figure 1.

The 3D incompressible N-S equations are solved in a structured grid using the finite volume method (FVM). Time integration of the discrete equations is carried out by the artificial compressibility scheme [5]. Time-accurate simulation is achieved by employing a dual time stepping procedure (sub-iteration) at each physical time step. A general multi-block grid can be partitioned over a number of allowable processors. MPI is used to achieve parallel computation on a cluster. The original version of the solver has been used in the study of a wide variety of CFD problems, including unsteady aerodynamics [6, 7], two-phase flows [8], and human-induced contaminant transport [9, 10]. An immersed-boundary method [4] is incorporated to enable computations of flow about moving objects.The differential form of

the governing equations is

$$\rho \frac{\partial u_j}{\partial x_j} = 0 \,,$$
$$\frac{\partial \left(\rho u_i\right)}{\partial t} + \frac{\partial}{\partial x_j}\left(\rho u_i u_j + p\delta_{ij} - \tau_{ij}\right) = 0 \,.$$

where $u_i$ is the velocity vector, $\rho$ is the density, $p$ is the pressure and $\tau_{ij}$ is the viscous stress tensor. The elements of the stress tensor $\tau_{ij}$ for a Newtonian fluid can be defined as

$$\tau_{ij} = \mu\left(\frac{\partial u_j}{\partial x_i} + \frac{\partial u_i}{\partial x_j}\right) .$$

A finite-volume representation is obtained by the integration of the governing equations on every control volume $V_C$ and its corresponding control surface $A_C$:

$$\int_{V_C} \rho u_j n_j dA = 0 \,,$$
$$\int_{V_C} \frac{\partial \left(\rho u_i\right)}{\partial t} dV + \int_{A_C}\left(\rho u_i u_j + p - \mu\frac{\partial u_i}{\partial x_j}\right) n_j dA = 0 \,.$$

The 3D incompressible N-S equations are solved on multi-block, structured meshes. Time integration of the discrete equations is carried out by the artificial compressibility scheme [5]. Time-accurate simulation is achieved by employing a dual time stepping procedure (sub-iteration) at each physical time step. The basic formulation at time step $n+1$, sub-iteration $k$ is given by

$$\mathbb{A}\left(\mathbb{U}^{n+1,k+1} - \mathbb{U}^{n+1,k}\right) = -\mathbb{R}^{n+1,k} \,,$$

where $\mathbb{U} = (p, u_i)^{\mathrm{T}}$ and $\mathbb{A}$ is the system Jocobian matrix. $\mathbb{R} = (R_{C,}\, R_{M,i})^{\mathrm{T}}$ is the residual vector, where $R_C$ and $R_{M,i}$ are the residuals for the continuity and momentum equations, respectively.

To achieve parallel computation of CFD on GPU, the CFD algorithm must be redesigned to be massively parallel. Take the high-end nVidia Tesla K20c for example, it has 30 multiprocessors, each of which contains 64 (double-precision) or 192 (single-precision) cores, making a total of 832 (DP) or 2496 (SP) cores. As each core runs one thread, an algorithm must be designed for high level of data parallelism to benefit from GPU.

Currently several options exist for porting CPU codes to GPU. The two dominant approaches are the high-level directive-based programming standards, such as OpenACC and PGI Accelerator, and the low-level programming frameworks, such as CUDA and OpenCL. OpenACC, much similar to OpenMP, allows programmers to annotate a source code to identify areas that should be accelerated using GPU. The compiler generates corresponding GPU binary at compile time and automatically offloads the binary to GPU when the program is executed. One apparent advantage of directive-based porting is easy maintenance. Only one copy of the code need to be maintained for the two target platforms (CPU and GPU). With appropriate compiler options the source code can be compiled into a pure-CPU binary or an accelerated binary capable of using GPU. This fact also facilitates debugging. As many

algorithmic problems can be exposed by debugging the pure-CPU binary. It is true that some bugs related to GPU parallelism only happen when the code is run on a GPU, which are much harder to debug. However, since the generation of GPU binary is carried out by the compiler automatically, the programmers are much less likely to make trivial errors. Also, the compiler is capable of detecting many common programming errors involving GPU parallelism, so the programmers can correct the errors immediately according to the warnings given by the compiler. On the other hand, OpenACC provides many directives for managing data transfer between the host (CPU) and the device (GPU), which is not present in current OpenMP standards (Version 3.1). Because of the relatively small bandwidth between host and device memory, data transfer must be carefully planned in order to avoid any unnecessary transfers. Another difference comes from granularity, as OpenMP usually involves less than 50 cores, while GPU has a native 2D or 3D grid and is capable of parallelizing thousands of threads in the grid natively. As a result, it is usually advised to parallelize the outside loops first in OpenMP, while it is better to parallelize the complete 2D or 3D grid loop directly on GPU.

As a preliminary study we used a 2D version of the flow solver to study of these two approaches (CUDA and OpenACC), while the full 3D version is ported using OpenACC only. Performance analysis is given based on run times in a Infiniband [11] cluster with nVidia GPUs. The paper is then concluded by the findings of the performance analysis.

# 2 Simplified 2D Version

## 2.1 GPU Implementation

Our first attempt is to port a simplified 2D N-S solver using the OpenACC. The porting of the 2D solver primarily involves the redesign of loops for better data parallelism and rewriting Fortran subroutine interfaces to adapt to OpenACC requirements. OpenACC directives are designed with modular programming in mind, so that the original structure of the code can be largely preserved. All computation-intensive tasks inside the main loop are now carried out by GPU. I/O can only be executed on the host side so they are avoided whenever possible in the main loop. All arrays involved in the main loop remain on GPU memory until finish, and temporary data arrays (such as the time step) are created directly on GPU memory when necessary, avoiding CPU-GPU transfers.

As our second attempt, a CUDA Fortran version of the 2D code is implemented. The primary goal of this version is to explore the potential and limit of GPU. The primary advantage to program in CUDA is its exposure of shared memory, essentially a cache memory between the main GPU memory and the GPU thread processors. As the bandwidth between the thread processors and the shared memory is even larger than that with the main GPU memory, the performance is expected to further increase.

Following a similar approach as in OpenACC programming, the goal is to minimize data transfer between GPU main memory and shared memory. However, the shared memory is only consistent within a "thread block", which is a group of threads that logically runs simultaneously on GPU [12]. Due to size of a thread block being much smaller than any practical mesh, the mesh is divided into smaller blocks which can fit into a thread block. Residual calculation and time marching are carried out locally within a thread block, so that

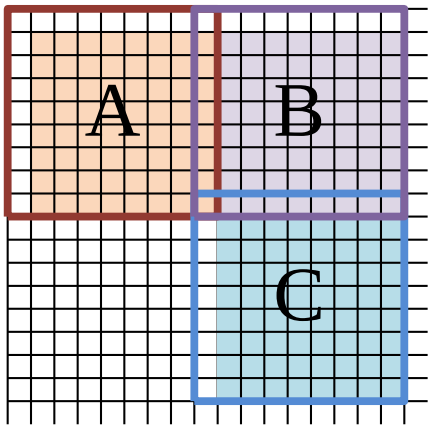


Figure 2: Examples of overlapping blocks. Block A and B are boundary blocks, and C is an internal block.

residual array are not transferred in/out shared memory. Time step calculation is also local to each thread, thus eliminating the time step array in GPU global memory.

Because residual calculation depends on data outside the block, overlapping blocks with one row/column of ghost cells are used. Take the three blocks in Figure 2 for example. The color-shaded areas represent the cells to be updated by the block they reside. The top row and left-most column of each block is not updated by its block. Instead, the blocks on the left/top updates those rows and columns, such as "B" and "C". Boundary blocks relies on boundary conditions to define the left/top rows and columns, such as "A" and "B". An sacrifice due to the use of ghost cells is the redundant calculation on the overlapped rows and columns, which proves to be a good tradeoff to allow complete local calculation inside a thread block.

Note that the boundary fluxes have two directions ($i$ and $j$) . It is possible that the two threads calculating flux differences on $i$ and $j$ both attempt to write into the same memory location, causing a memory contention. One common solution is grouping of surface fluxes, so called the "coloring" scheme. This is straightforward in the case of a 2D structured grid, as $i$ and $j$ directions naturally forms two "colors".

Eventually, the main loop is converted into a monolithic CUDA kernel, which has zero memory exchange between GPU global memory and thread block shared memory within each time step. Data exchanges only happen at the end of each time step, where all updated field variables must be synchronized with GPU global memory.

### 2.2 Performance Analysis

The simplified 2D solver is capable of both steady-state and time-accurate simulations. In Figure 3, a steady-state channel with a slopped back step is simulated. Note that the three circular obstacles, which is achieved by immersed boundary scheme [13]. A time-accurate result is given in Figure 4, which is a simple rectangular channel with one circular obstacle. As shown in the figure, the vortex shedding is well reproduced.

A comparison of computation time of the main loop is given in Figure 5. "CPU" time is

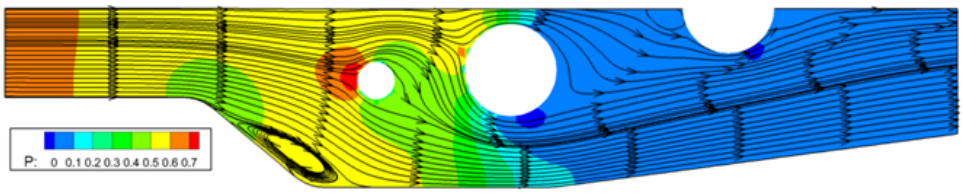


Figure 3: Pressure map of 2D steady-state channel flow. Re~200.

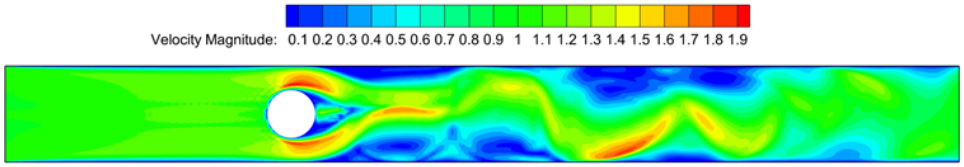


Figure 4: Velocity magnitude map of a 2D time-accurate channel flow. Re~4000.

obtained by the sequential version of the code running on one CPU core. Both OpenACC and CUDA versions are run on a single nVidia K20c GPU. The test case is a steady-state 2D case as in Figure 3, using a $301 \times 101$ mesh and double precision.

The benefit of using GPU is apparent. The OpenACC version achieved an 8x speedup, while the CUDA version achieve 44x. Programming in CUDA Fortran enables more flexibility in parallel algorithm, while shared memory programming is currently impossible with OpenACC. As a result, the performance is significantly better than the OpenACC version. On the other hand, programming in CUDA requires much more effort. It is easy to make trivial mistakes, such as incorrect loop bounds. Another concern is code re-usability, as CUDA is a quickly evolving proprietary platform. Unlike CUDA C, currently no debugger can debug CUDA Fortran natively on GPUs. From a cost-effectiveness aspect, OpenACC provides a good compromise between pure CPU and pure CUDA. It offers good speedup, with minimal to moderate effort on porting legacy codes. It is also easy to debug and maintain, which is particularly desirable for scientific computation.

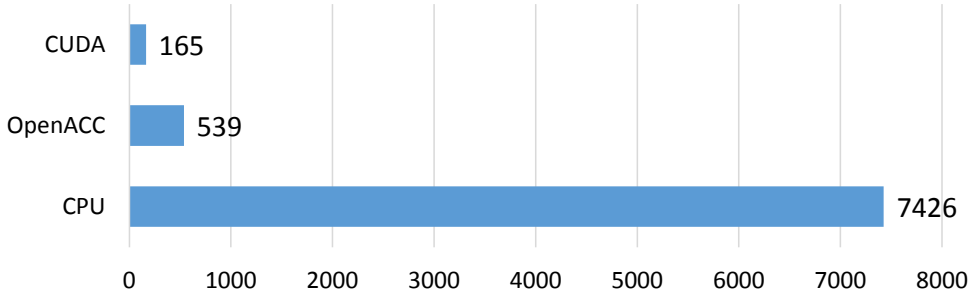


Figure 5: Comparison of run times (in seconds) of the main loop.

# 3 Complete 3D Version

## 3.1 GPU Implementation

The porting effort of the IN3D code is similar to the 2D version, with a focus on using OpenACC directives to identify computation-intensive regions and schedule data exchanges between CPU and GPU memory. Not unlike the 2D version, the finite volume flux calculation also causes memory contingency unless a coloring scheme is used. Because of the structured nature of our grids, a straightforward odd-even coloring scheme is employed to resolve the memory contingency. Note that the new OpenACC 2.0 standard [14], which has yet to receive wide compiler support as of late 2013, allows atomic operations, which will provide another possible solution to avoid memory contingency without the use of coloring scheme.

More challenges arise due to the large amount of data a full 3D version processes. One direct consequence is a more complex data structure. In the current version (1.0) of OpenACC, Fortran custom data structures are not supported on GPU memory, and partial reference of Fortran arrays can often cause problems during subroutine argument passing. On the other hand, the data structure must allow enough flexibility to store variable amount of data, depending on the size of the simulation problem. A carefully designed scheme based on the idea of "array of arrays" is adopted.

Another consequence of the size of 3D problems is that the complete domain easily exceed the memory limit of one compute node in a cluster, thus forcing the domain to be partitioned into smaller blocks. This limitation is more prominent as most GPU has less memory than the CPU.

On NCSU's ARC cluster, the nodes are equipped with InfiniBand ConnectX-2 VPI interfaces, which is expect to provide a bandwidth around 3GB/s and a latency of several $\mu s$. To take full advantage of the InfiniBand interconnection, MVAPICH2 is selected as the MPI implementation to handle the inter-process data exchange [15]. Not only MVAPICH2 is actively optimized for the latest interconnect hardware and software, it includes a pipelining capability at user level with non-blocking MPI and CUDA interfaces, which allows MPI subroutines to operate on variables residing on GPU memory directly. Combined with the MPI's Derived Data Type (DDT) features, explicit data packing and manual GPU-CPU data transfers can be completely avoided, which can greatly reduce efforts on porting MPI-related codes.

Data packing of ghost cells can be carried out either manually or automatically by using DDT. When using DDT, the data to be transferred are indicated by its locations in the overall data structure, and the locations are organized as the "derived data types" in MPI [16]. Internally MPI arranges the necessary memory buffers, data copying and inter-process exchanges, freeing the programmers from many trivial and repetitive tasks, often with optimized data transfer efficiency. Although DDT is implemented for GPU variables, the data packing algorithms, originally optimized for CPU execution, prove to be much less effective on GPU [17]. In the current general version of MVAPICH2, the CUDA memory copy based mechanism are used for DDT data packing [18]. It is found that, with this mechanism the MPI DDT data exchange in IN3D is unacceptably slow, simply due to the CPU-side overhead incurred by CUDA memory copy subroutines. It is possible to write a purpose-built CUDA kernel-based data packing patch [19] in MVAPICH2 for IN3D, which can indeed give optimized performance. However, doing so requires extra effort to maintain the patch, greatly reducing the

portability of the IN3D code.

In fact, the original IN3D code already include the capability of manual data packing, as ghost cells are copied one-by-one to a continuous buffer set up by the solver. It is strictly sequential, whose Fortran pseudo code is as follows

```
size_buffer = 0
do nv = nv0, nv1
  do k = k0, k1, ks
  do j = j0, j1, js
  do i = i0, i1, is
    size_buffer = size_buffer + 1
    MPI_buffer(size_buffer) = data(i,j,k,n)
  end do
  end do
  end do
end do
```

where i, j, k are the spatial indices and nv is the field variable index. Depending on the block face being packed, the loop bounds and strides take different values. Unpacking is carried out in a similar fashion, but allowing more variations in bounds and strides for complex block connections. Following the kernel-based data packing approach, this sequential manual data packing algorithm can be completely redesigned to allow parallel execution. This is achieved by using calculated-index scheme, whose pseudo code is as follows:

```
!$acc kernels
!$acc loop seq independent
do nv = nv0, nv1
  !$acc loop independent
  do k = k0, k1, ks
  !$acc loop independent
  do j = j0, j1, js
  !$acc loop seq independent
  do i = i0, i1, is
    MPI_buffer( Ki_b*i + Kj_b*j + Kk_b*k + Kn_b*n) = &
          data( Ki_d*i + Kj_d*j + Kk_d*k + Kn_d*n)
  end do
  end do
  end do
end do
!$acc end kernels
```

The loops over physics variables and two outer grid directions are rewritten as nested loops with zero data dependency, allowing maximum parallelism. Note that the most inner loop is forced to remain sequential to avoid parallelizing 4 nested loops, which caused compiler

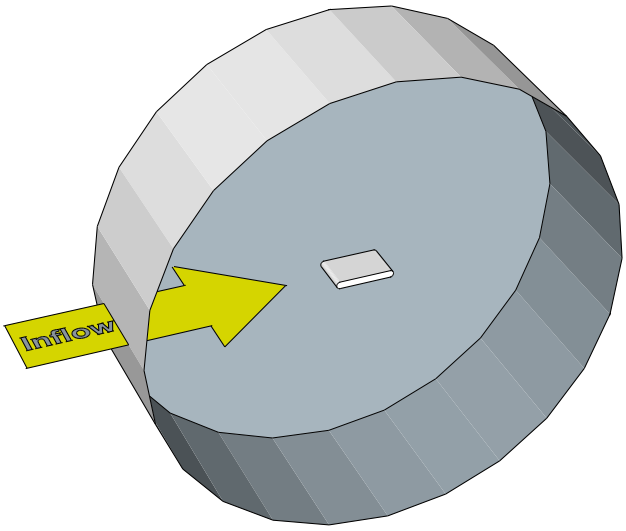


Figure 6: Illustration of the 3D domain for performance assessment.

errors.

### 3.2 Performance Analysis

A simplified O-type mesh based on previous dynamic stall studies is used for the performance analysis. To reduce the impact of possible load imbalance, the airfoil is reduced to a flat plate with rounded leading and trailing edges. An illustration with exaggerated dimensions is given in Figure 6. The grid has 7 million cells, divided into 152 blocks. Multiple blocks can be mapped to a CPU or GPU, which facilitates load balancing among compute nodes.

Depending on the compilation options, the same code is used to generate executables targeted for either CPU execution or GPU execution. The simulations are then carried out on the same set of 32 GPU-capable compute nodes, so that the impact of interconnection is the same for both CPU and GPU simulations. The primary purpose of such a comparison is to illustrate the cost of data exchange between compute nodes in a cluster. Figure 7 shows the breakdown of main loop run times of a 2000-step steady state run. The overall run time is listed at the end of the stacked bars, which shows an overall speedup of 4x. The flux calculation remains the dominant portion, achieving a speedup of 3.4x. The total data exchange cost, including data packing and MPI transfers, is reduced from 170s to 34s - a speed up of 5. The data packing itself has even better performance boost, thanks to the efficient data packing scheme on GPU. The data exchange accounts roughly 1/4 of the total run time in GPU, a moderate improvement over the CPU version.

The MPI data exchange imposes a difficult limit on the overall speedup. The data packing kernel on GPU comprises simple memory copies, which has very limited potential for further optimization, even using direct CUDA programming. If the flux calculation of the full 3D code could eventually achieve the same 14x speedup as seen in 2D by OpenACC, the overall speed up would be roughly 7.3x. This is not surprising. In fact, the inter-node data exchange bottleneck has always been around, whose impact is only amplified by GPU's high perfor-

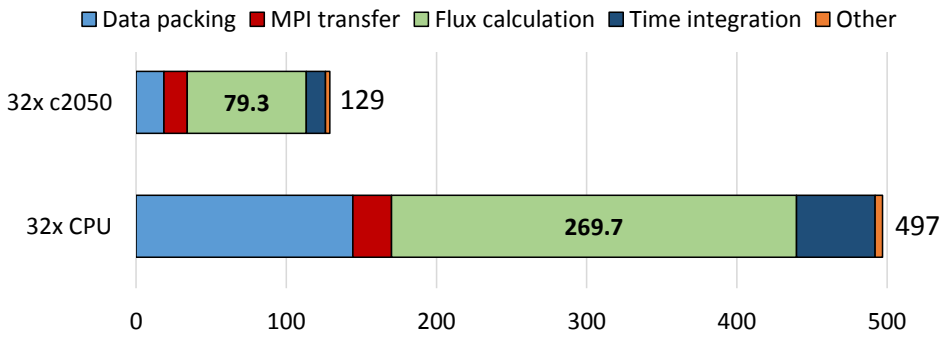


Figure 7: Main loop run times (in seconds).

mance inside one compute node. Task overlapping can mask some of the data exchange time, which, nevertheless, requires extensive changes to code structure and is beyond the scope of this study.

# 4 Conclusion

In this study, a multi-block incompressible Navier-Stokes solver and its simplified 2D version are ported to nVidia GPGPU platform, using both OpenACC (both 3D and 2D solvers) and CUDA (2D solver only). The 3D version incorporates two levels of parallelism. On the block level MPI is employed to shared work among compute nodes, while on process level GPU is used to massively parallelize the computation on grid cells. Using MVAPICH2, the MPI subroutines can directly operate on variables residing in GPU memory, allowing highly portable codes. It is found that, for a general MPI implementation, manual data packing is much more effective in packing ghost cells in 3D CFD applications.

3D performance comparisons are studied using a modern GPU-capable cluster environment with nVidia GPUs. It is found that a highly optimized CUDA code gives a much better performance between the two GPU porting methods. OpenACC, on the other hand, gives a good performance improvement while requiring much less programming and maintaining effort, thus offering a balanced approach on porting legacy codes in terms of cost-effectiveness. Furthermore, OpenACC may facilitate future porting efforts on other massive-parallel architectures due to its high similarity to OpenMP, further increasing the re-usability of the ported codes.

The performance study also reveals two directions to achieve further performance improvement. First, the CFD calculation inside each node can still be improved. However, this becomes less a trivial task beyond off-loading computation-intensive areas to GPU. More extensive redesign of algorithms, or even a complete change of CFD methods must be involved for any dramatic speed improvement. Second, while the speed of CFD calculation can be greatly improved theoretically, the inevitable data exchange across compute nodes in a cluster imposes an even more difficult limit on overall performance improvement, which can only be achieved by more efficient interconnection. From this aspect, the high portability of the directive-based programming approach will prove to be highly valuable for the scientific community to adapt to the ever-changing computer technology landscape.

## Acknowledgment

This work is supported by the Air Force Office of Scientific Research under their Basic Research Initiative program grant FA9550-12-1-0442 (PI - Wuchun Feng, Virginia Tech), monitored by Dr. Doug Smith and Dr. Fariba Fahroo. The authors acknowledge the use of the NCSU's ARC CPU/GPU computing cluster, operated by Dr. Frank Mueller and supported primarily by NSF under CRI Grant 0958311. Special thanks to Hao Wang (Virginia Tech) for his support on MVAPICH2.

## References

[1] Open MPI: Open source high performance computing. The Open MPI Project. [Online]. Available: http://www.open-mpi.org/

[2] MVAPICH2: High performance MPI over InfiniBand, 10GigE/iWARP and RoCE. OSU. [Online]. Available: http://mvapich.cse.ohio-state.edu/

[3] J. R. Edwards and M.-S. Liou, "Low-diffusion flux-splitting methods for flows at all speeds," *AIAA journal*, vol. 36, no. 9, pp. 1610–1617, 1998.

[4] J.-I. Choi, R. C. Oberoi, J. R. Edwards, and J. A. Rosati, "An immersed boundary method for complex incompressible flows," *Journal of Computational Physics*, vol. 224, no. 2, pp. 757–784, 2007.

[5] A. J. Chorin, "Numerical solution of the navier-stokes equations," *Mathematics of computation*, vol. 22, no. 104, pp. 745–762, 1968.

[6] K. Ramesh, A. Gopalarathnam, J. Edwards, M. Ol, and K. Granlund, "An unsteady airfoil theory applied to pitching motions validated against experiment and computation," *Theoretical and Computational Fluid Dynamics*, vol. 27, no. 6, pp. 843–864, 2013. [Online]. Available: http://dx.doi.org/10.1007/s00162-012-0292-8

[7] G. Z. McGowan, K. Granlund, M. V. Ol, A. Gopalarathnam, and J. R. Edwards, "Investigations of lift-based pitch-plunge equivalence for airfoils at low reynolds numbers," *AIAA journal*, vol. 49, no. 7, pp. 1511–1524, 2011.

[8] D. A. Cassidy, J. R. Edwards, and M. Tian, "An investigation of interface-sharpening schemes for multi-phase mixture flows," *Journal of Computational Physics*, vol. 228, no. 16, pp. 5628–5649, 2009.

[9] J.-I. Choi and J. R. Edwards, "Large eddy simulation and zonal modeling of human-induced contaminant transport," *Indoor air*, vol. 18, no. 3, pp. 233–249, 2008.

[10] ——, "Large-eddy simulation of human-induced contaminant transport in room compartments," *Indoor air*, vol. 22, no. 1, pp. 77–87, 2012.

[11] The infiniband architecture. [Online]. Available: http://www.infinibandta.com

[12] NVIDIA. (2013) CUDA C programming guide.

[13] C. S. Peskin, "The immersed boundary method," *Acta numerica*, vol. 11, no. 0, pp. 479–517, 2002.

[14] *The OpenACC Application Programming Interface*, 2013.

[15] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[16] "MPI: A message-passing interface standard," Message Passing Interface Forum, Tech. Rep., 2009.

[17] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. Panda, "GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, 2013.

[18] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda, "Optimized non-contiguous mpi datatype communication for gpu clusters: Design, implementation and evaluation with mvapich2," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 2011, pp. 308–316.

[19] J. Jenkins, J. Dinan, P. Balaji, N. F. Samatova, and R. Thakur, "Enabling fast, noncontiguous GPU data movement in hybrid MPI+GPU environments," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE, 2012, pp. 468–476.