GPU Port of A Parallel Incompressible Navier-Stokes Solver based on OpenACC and MVAPICH2

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

June 29, 2014

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

North Carolina State University

Abstract

OpenACC is a directive-based programing standard aim to provide a highly portable programming model 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 stresses a demand for careful planning of data movement and novel approaches of parallel algorithms not commonly involved in scientific computation. By following a similar concept of 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 relatively narrow interconnection bandwidth among GPUs and the very fine granularity of GPGPU architecture. The first is particularly restrictive when cross-node data exchange is involved in a cluster environment. Furthermore, GPGPU's fine-grained parallelism is in conflict with certain types of inherently serial algorithms, posing further restrictions on performance. In our study, an implicit 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

Recently, Graphics Processing Unit (GPU) attracts much attention as a promising technology for large-scale parallel computation (see, for example [1, 2, 3, 4, 5, 6]). GPU has the potential to achieve one or two magnitudes of performance improvement for highly-parallel algorithms. It has been found that, CFD codes can hugely benefit from GPU, with vigorous efforts such as rewriting the codes with low-level GPU programming models such as CUDA and OpenCL. However, large-scale applications of GPU in CFD still remain a challenging subject. First of all, the low-level model approach is not always cost-effective for the task of porting legacy codes to GPU, considering the huge amount of effort on porting the codes to fit new algorithms and data structures. Low-level GPU models are also constantly evolving, which raises concerns of future re-usability of the ported codes. Finally, CUDA is strictly tied to nVidia hardware. The emergence of other accelerator architectures, such as Intel's Many Integrated Core (MIC) technology, renders the porting efforts to CUDA useless if Intel MIC is to be utilized.

Similar situations have been encountered when shared-memory parallelism was first adopted for scientific computing, which were well resolved by the introduction of OpenMP, a directive-based programming model designed to hide the underlying implementation details from the programmers. As a similar approach, a programming standard called OpenACC is developed to meet the needs to simplify GPU programming. Programmers can use a collection of compiler directives to specify loops and regions of their codes to be offloaded from a host CPU to an accelerator. One apparent advantage of directive-based porting is easy maintenance. Only one copy of the code needs 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 utilizing GPU. This fact also facilitates debugging. As many algorithmic problems can be exposed by debugging the pure-CPU binary. Certain bugs related to GPU parallelism only occur when the code is run on a GPU, thus are much harder to debug. Since the generation of GPU binary is carried out by the compiler automatically, the programmer is much less likely to make trivial errors. Finally, OpenACC directives are designed to closely resemble their OpenMP counterparts. As a result, codes written with OpenACC are very easy to port to future OpenMP standards which will incorporate GPU computation capabilities.

Even with the assistance of OpenACC, the migration from CPU to GPU still poses many issues not seen in the adoption of OpenMP. A major difficulty is the bottleneck of the CPU-GPU data transfer, as the data bandwidth across CPU-GPU boundary is much smaller than those of internal CPU or GPU memory. For smaller problems this can be avoided by keeping most of data on GPU throughout the simulation. However, as the size of the simulation grows, memory on a single GPU becomes inadequate, thus requiring cooperation of multiple GPU's spanning over multiple compute nodes in a cluster. As the inter-node communication is handled by Messange Passing Interface (MPI), the symptom usually appears as slow MPI transfer rate [7]. In fact, it is primarily due to two factors.

First, traditional ghost cell data packing schemes, which are essential parts of message-passing parallelism of CFD, are usually very inefficient on GPU. In reality two situations are often encountered. Older CFD codes, written with MPI Version 1, tend to have their own data packing (manual data packing), while more recent CFD codes, written with MPI Version 2, often make use of the Derived Data Type (DDT) feature (automatic data packing [8]). Regardless, these schemes usually assumes serial execution, which needs be to be rewritten to allow efficient parallel execution on GPU. Manual data packing is relatively straightforward to parallelize, while DDT data packing relies on efficient MPI implementation to achieve acceptable performance. Due to the general nature of DDT, the migration of DDT data packing algorithms to GPU proves to be a rather challenging task [9].

Second cause of the MPI bottleneck is the lack of reliable direct GPU data transfer mechanisms between compute nodes in a cluster, as any MPI data must pass the CPU-GPU boundary twice, once in the sending end and once in the receiving end. The extremely lim-

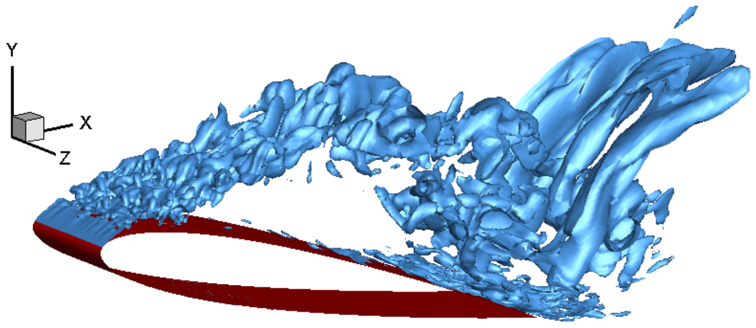


Figure 1: Lambda-2 isosurface of a dynamic stall simulation. Re$\sim 10^6$, 8×10^6 cells.

ited bandwidth across the CPU-GPU boundary greatly impacts the MPI transfer efficiency. This problem is expected to be relieved by direct GPU data transfer technologies in near future. As the underlying technologies are constantly revolving, it is desirable to utilize a general MPI interface which can make use of the optimal mechanism available, while hiding the underlying details from the scientific programmers. As of early 2014, direct GPU communication technology between compute nodes is under development but has yet to reached production maturity. However, latest MPI implementations, such as OpenMPI [10] and MVAPICH2 [11], allow passing variables on GPU memory as MPI subroutine arguments, even though the inter-node GPU data transfers still involves internal CPU-GPU transfers. MVAPICH2 is expected to employ direct inter-node GPU communication once it is available. From the perspective of scientific programmers, their CFD codes will be able to make use of the direct inter-node GPU transfer technology once it reaches production maturity, while requiring minimal changes to the CFD codes.

A more difficult performance restriction is encountered when porting implicit time integration schemes. Employing an implicit time integration instead of explicit time integration can often significantly improve simulation speed. This is particularly true when the physical time scale is relatively large, as is often the case in incompressible flows. A common theme in implicit methods of CFD is to solve a large sparse linear system. The solution of linear systems often involves a strongly serial procedure, which is difficult to parallelize. Furthermore, in 3D problems such linear system easily reaches prohibitive sizes, requiring large amount of storage. Considering the performance of the whole implicit CFD solver is often restricted by the linear solver, it is critical to select an efficient parallel algorithm for solving large sparse linear systems.

2 Governing Equations

Our study attempts to port a 3D LES incompressible Navier-Stokes solver onto the GPU architecture. It is based on an existing solver validated for a range of model problems [12, 13]. 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 [14]. 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 [15, 16], two-phase flows [17], and human-induced contaminant transport [18, 19]. An immersed-boundary method [13] 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, ρ is the density, p is the pressure and τ_{ij} is the viscous stress tensor. The elements of the stress tensor τ_{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 [14]. 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 $\mathbb{U} = \left(p, u_i\right)^{\mathrm{T}}$ and $\mathbb{A}$ is the system Jocobian matrix. $\mathbb{R} = \left(R_C, R_{M,i}\right)^{\mathrm{T}}$ is the residual vector, where R_C and $R_{M,i}$ are the residuals for the continuity and momentum equations, respectively.

3 GPU Implementation

The solver is being ported to GPU using PGI Fortran and C compilers with OpenACC support. The migration generally follows the same methodology as shared-memory parallelism using OpenMP. Directives are added to the source codes to allow the compiler to generate corresponding GPU binary at compile time and automatically offloads the binary to GPU when program is executed. This hides large amount of GPU implementation details from the the programmer, which is particularly desirable for scientific computation purposes. One major difference from OpenMP, however, is that OpenACC provides many directives for managing data transfer between CPU and GPU. Because of the relatively small bandwidth between host and device memory, data transfers is carefully planned in order to avoid any unnecessary transfers. Another difference comes from granularity. The CFD algorithm must be revisited with high level of parallelism, which is critical for achieving high performance on GPU. For comparison, OpenMP usually involves less than 50 threads while GPU is capable of parallelizing thousands of threads. 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, each capable of running one thread. Hence, it is usually advised to parallelize the outside loops first with OpenMP, while it is better to parallelize the complete nested 3D grid loop directly with OpenACC.

The primary tasks of porting the solver include the redesign of loops for better data parallelism and rewriting Fortran subroutine interfaces to adapt to OpenACC requirements. OpenACC provides special directives to assist modular programming, so that the original structure of the code can be largely preserved. All computation-intensive tasks inside the main loop must be 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 are created directly on GPU memory when necessary, avoiding CPU-GPU transfers. Since OpenACC is essentially shared-memory parallelism, like OpneMP, memory contingency is a common issue when porting legacy sequential codes. A typical solution is to arrange the memory access in groups, so called the "coloring" scheme. 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, which PGI has yet to fully support in their compilers as of early 2014, allows atomic operations, which will provide another possible solution to 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 of OpenACC, Fortran custom data structures are not yet fully 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 [20] ConnectX-2 VPI interfaces, which is expect to provide a bandwidth around 3GB/s and a latency of several μs. To take full advantage of the InfiniBand interconnection, MVAPICH2 is selected as the MPI implementation to handle the inter-process data exchange [21]. 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

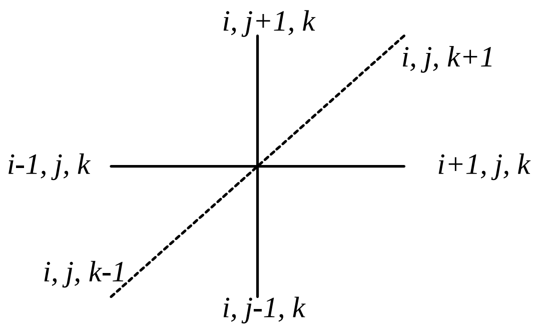


Figure 2: 7-point stencil associated with the structured 3D finite volume method

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. Unfortunately, the data packing algorithms, originally optimized for CPU execution, prove to be much less effective on GPU unless specialized GPU codes are employed [22, 9]. As a results, the manual data packing scheme is retained in the current version. This is, indeed, not the most portable solution. Future versions may choose different approaches, such as optimized MPI DDT or generalized data exchange library designed for GPGPU.

In the manual data packing scheme, ghost cells are copied one-by-one to a continuous buffer set up by the solver. The buffer is then passed to MPI for inter-process transfers. The manual data packing algorithms, however, is a sequential code, thus requiring complete redesign to allow parallel execution. This is achieved by using calculated indexes. All three grid directions are rewritten as nested loops with zero data dependency, allowing maximum parallelism. The physical variables, such as pressure and velocity components, are stored contiguously in GPU memory for each individual grid point, which is called the "array of structures" arrangement. To maximize the memory coalescence, the loop over physical variables is therefore mapped to a sequential loop in each thread. The implementation details can be found in our earlier publication [23].

3.1 Implicit Time Marching

It is known that an implicit time marching scheme can greatly increase simulation speed, if the time scale of physical process is not particularly small. This is often the case for non-reacting incompressible flow, where smooth solutions can be expected. An early attempt of porting the implicit time marching is described in this section.

A first-order fully implicit time-accurate time marching is employed. The time marching eventually reduces to solving a block sparse linear system

$$\mathbb{A}\Delta\mathbb{V} = -\mathbb{R}\,, \tag{1}$$

The 3D finite volume scheme involves a 7-point stencil as shown in Figure 2. This results in A has the pattern given in Figure 3.

The submatrices $A_n, B_n, C_n, D_n, E_n, F_n, G_n$ at a particular row correspond to the contribution by grid cells (i,j,k-1), (i,j-1,k), (i-1,j,k), (i,j,k), (i+1,j,k), (i,j+1,k), (i,j,k+1) for the equations at grid cell (i,j,k), respectively. Each submatrix has the dimension of 6x6, due to the

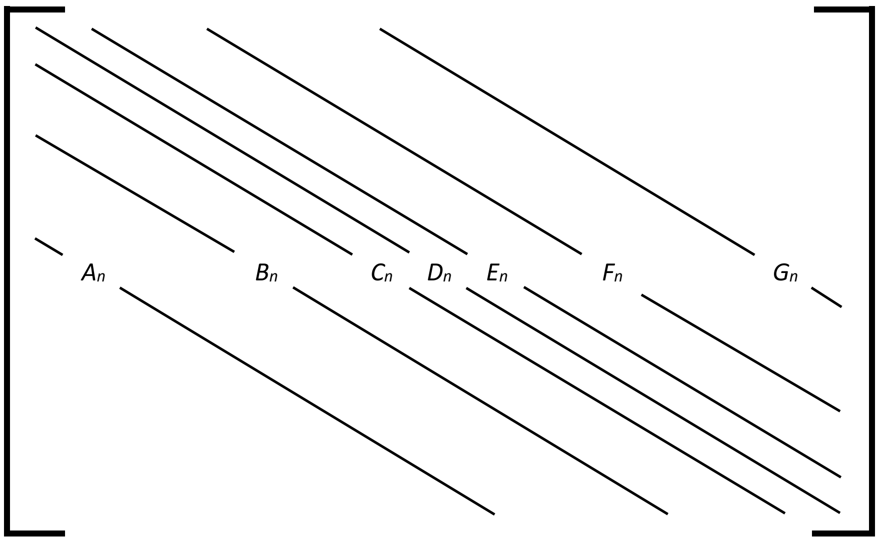


Figure 3: Pattern of matrix A associated with 7-point finite volume method

fact that there are 6 primitive variables to be solved at each grid cell. Block-incomplete-LU decomposition (BILU) with zero fill-in, or so-called BILU(0), is employed as the preconditioner. It involves a recurrence for the diagonal submatricies to be updated by the following relationship:

$$\hat{D}_{i,j,k} = D_{i,j,k} - C_{i,j,k}\hat{D}^{-1}_{i-1,j,k}E_{i,j,k} - B_{i,j,k}\hat{D}^{-1}_{i,j-1,k}F_{i,j,k} - A_{i,j,k}\hat{D}^{-1}_{i,j,k-1}G_{i,j,k} \tag{2}$$

which shows a data dependency of (i,j,k) on (i-1,j,k), (i,j-1,k) and (i,j,k-1). Such an algorithm is inherently sequential and cannot be parallelized. In fact, in the original CPU codes the BILU(0) algorithm is implemented sequentially. This is completely acceptable as each process on runs on one CPU core. However, it poses a serious performance restriction when the codes is ported onto GPU, as a naive implementation would result in sequential execution on GPU.

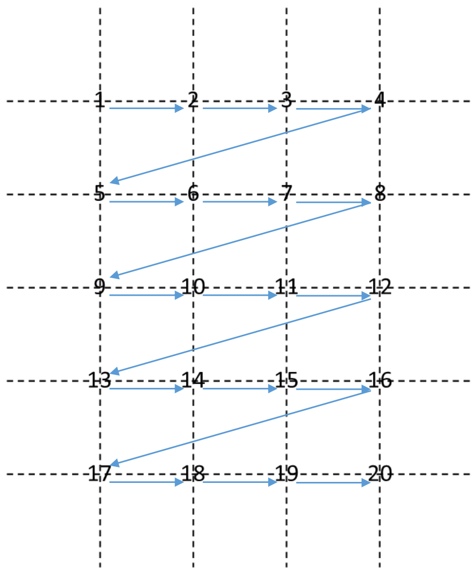
On way to extract parallelism from this type of data dependency is the "wavefront" ordering scheme. Here we use a 2D example to explain this scheme. Both the naive ordering and the wavefront ordering of a 5x4 2D block are shown in Figure 4. Inside any "wavefront line", where i+j being a constant, the grid cells are not data-dependent with each other, thus allowing parallel processing inside the wavefront. For 3D, a "wavefront hyperplane" consists of all grid cells with a constant sum of their index numbers i+j+k, as shown in Figure 5. A detailed study on wavefront ordering scheme can be found in [24].

Once all $\hat{D}$ are found, the BILU(0) factorization can be express as

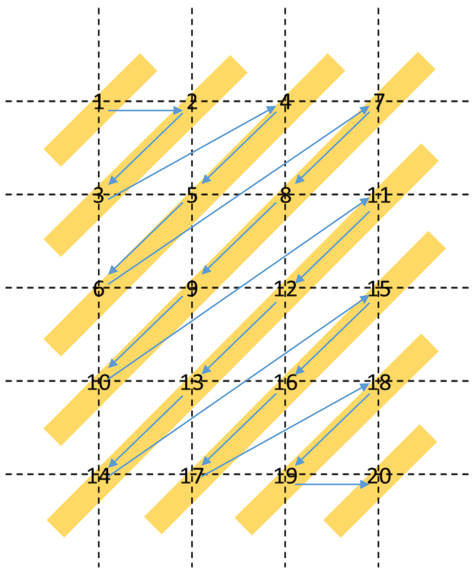
$$\mathbb{M} = (\hat{\mathbb{D}} - \mathbb{L})\hat{\mathbb{D}}^{-1}(\hat{\mathbb{D}} - \mathbb{U})\,, \tag{3}$$

where $\hat{\mathbb{D}}$ is the block-diagonal matrix consists of all $\hat{D}$, $-\mathbb{L}$ is the strict lower triangle of $\mathbb{A}$, and $-\mathbb{U}$ is the strict upper triangle of $\mathbb{A}$. Instead of solving the original linear system (1), this factorization allows a fast approximation solution of the following system by solving two triangle linear systems using simple forward and back substitutions:

$$(\hat{\mathbb{D}} - \mathbb{L})\hat{\mathbb{D}}^{-1}(\hat{\mathbb{D}} - \mathbb{U})\Delta\mathbb{V} = -\mathbb{R}\,. \tag{4}$$



(a)



(b)

Figure 4: (a) Naive ordering; (b) wavefront ordering

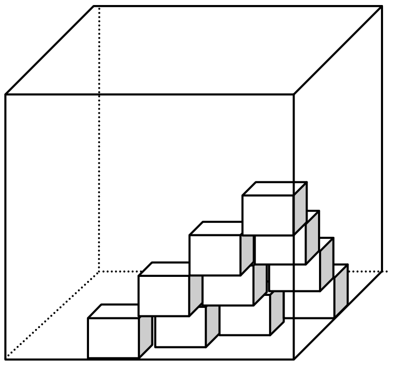


Figure 5: A wavefront plane in a 3D block

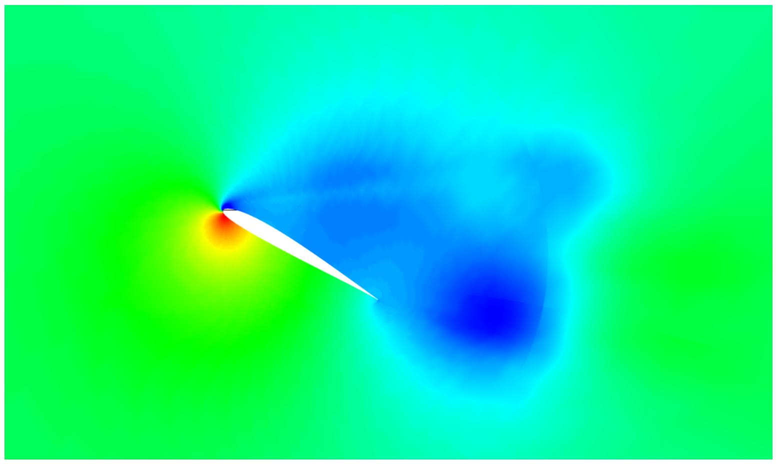


Figure 6: Pressure contour of a high angle-of-attack LES simulation. Airfoil is SD7003 with a chord length of 10cm. 3.3M cells, Re=1000, CFL=15.5.

Note that there is a similar data dependency pattern when solving triangle linear systems as that in the BILU(0) factorization, so that wavefront ordering scheme must be employed to extract parallelism. To improve accuracy of the approximated solution, corrections can be applied iteratively on the approximated solution, using the same $\hat{\mathbb{D}}$ calculated by the BILU(0) factorization.

One limitation imposed by OpenACC during BILU(0) factorization is that it does not allow localization of temporary arrays on the thread level [25, 26]. Temporary arrays are needed during the computation of multiplication of submatrices. Current implementation of OpenACC by PGI only allows localization of these arrays using global CUDA memory, which would easily fill up the GPU global memory, considering that for every cell at least two 6x6 temporary arrays are needed. A better implementation should allow localization using shared memory, which requires explicit CUDA programming and voids the portability of the codes. To resolve this conflict we decided to retain the use to OpenACC, and unroll all thread-level loops to eliminate the use of temporary arrays. Since all scalar variables are localized on the thread-level, the overall memory restriction becomes irrelevant. Note that such manual unrolling of loops is essentially a workaround, since it does not allow easy modification to accommodate more unknown variables to be involved during solution process of the linear system. Future implementation of OpenACC may allow the localization of temporary arrays using CUDA shared memory, which can permanently resolve this problem.

4 Results

Several test cases are simulated to verify the correctness of the porting. One of the test case is shown in Figure 6, where the pressure contour of a high angle-of-attack simulation for SD7003 is shown. The CFL number is set to 15.5 in this case, which allows a much more efficient simulation than an explicit method.

A detailed comparison of computation time of the main loop is given in Figure 7. “6x CPU” time is obtained by the MPI version of the code running on 6 CPU cores on 6 compute

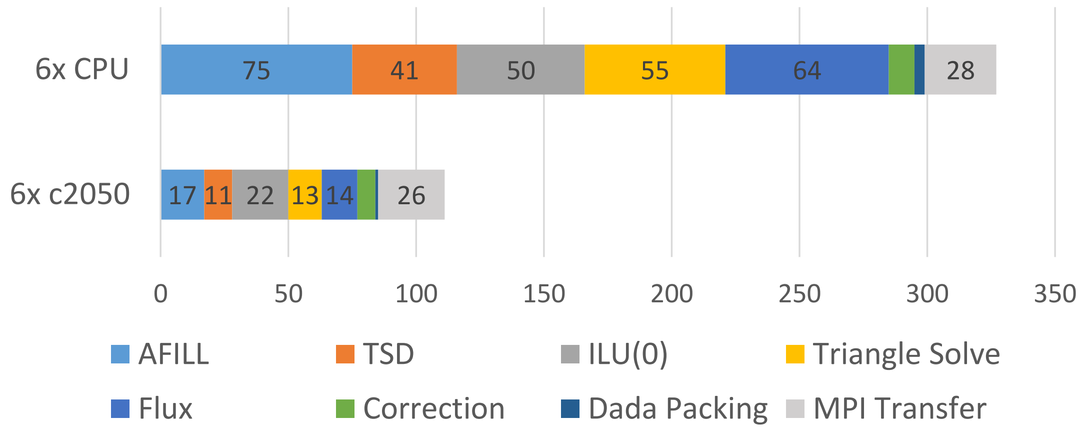


Figure 7: Comparison of computation time of the main loop.

nodes, and "6x c2050" by 6 nVidia c2050 GPU on the same 6 compute nodes. This is to maximize the MPI transfer effects across compute nodes. The test case is a steady-state 3D channel flow, using a 700K mesh partitioned into 6 blocks, double precision and 100 iteration steps.

As we can see the OpenACC version achieved roughly 3x speedup. The profiling results show mostly consistent effects of GPU porting on different parts of the program. First of all, the time spent on MPI data transfer is almost the same for both versions. The data packing, however, takes relatively little time, which is in direct contrast to our previous results given by the explicit solver [23]. One explanation is that, in this particular case, we intentionally choose very large blocks to allow better performance by the wavefront scheme, which effectively lower the ratio of ghost cells to the overall domain. This shows that larger blocks can benefit the implicit methods. The benefit of using GPU is apparent for most computation-intensive subroutines which achieve various speedup from 2.5x to 4.5x.

Among the computation-intensive subroutines, the worst speedup occurs at the BILU(0) subroutine, which only achieve 2.2x speedup. One of the contributing factor may be insufficient load by small hyperplanes. Due to the nature of the wavefront scheme, the parallelization only happens on one wavefront plane at a time. For a block of 80x80x80 cells, the largest wavefront has 3240 cells while the smallest wavefront has only 1 cell. Although we can intentionally use larger blocks during partitioning of the overall simulation domain, a large number of wavefront planes would still has less number of cells than the number of GPU hardware cores, which can cause significant waste of GPU computation resources due to insufficient load. To further investigate the reason for the poor performance we studied the per-thread timing of both BILU(0) and the triangle solver algorithms on GPU, as given in Figure 8. The run time data is obtained from CUDA profiling output. The per-thread time is calculated by dividing the kernel run time of a hyperplane by the number of points/threads on that hyperplane. The results of a test run on a $79 \times 49 \times 49$ block are shown here, whose number of points for different hyperplanes is given in Figure 9. As we can see, the per-thread efficiency is generally good for larger hyperplanes. In fact, per-thread run time generally reaches an optimal value for hyperplanes with more than 400 points. This is consistent with

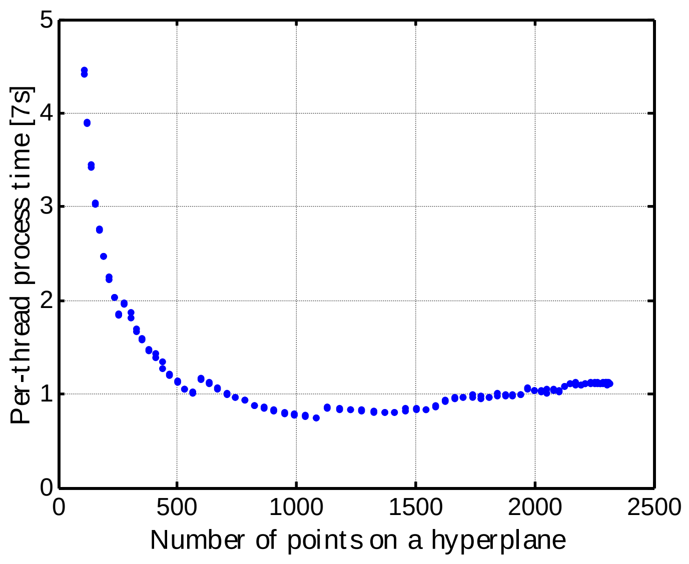
	BILU(0)	Triangle Solver
Total runtime	0.2087	0.03584
Total threads	189679	
Actual per-thread runtime	$1.1\mu s$	$0.19\mu s$
Optimal per-thread runtime	$1.0\mu s$	$0.165\mu s$
Loss	10%	14%

Table 1: Estimation of performance loss due to small hyperplanes

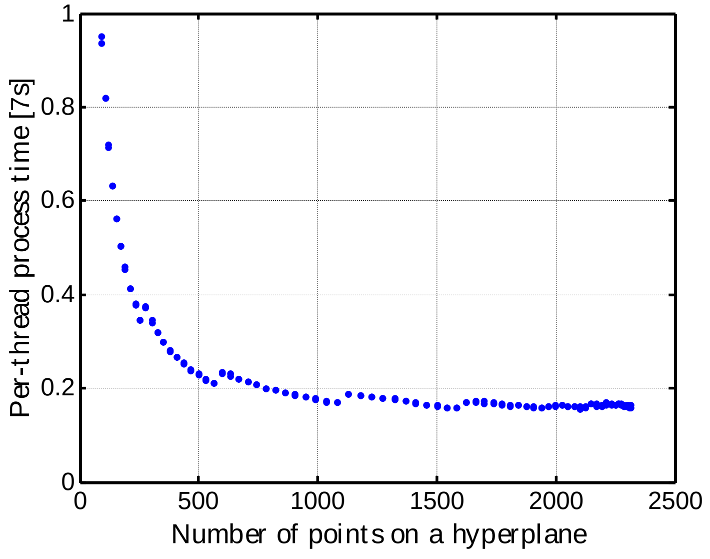
the fact that the nVidia c2050 GPU has 448 double-precision cores, so any hyperplane small than that causes insufficient load. In this test case roughly 70% of hyperplanes can supply enough computation load on this GPU. The remaining 30% cause different degrees of waste of computation resources. At its extreme, the first and last few hyperplanes only have several points to process, which means almost all the GPU cores are idle, waiting the few cores to complete their computation.

An attempt is made to quantify the loss of performance due to inefficient computation load scheduling in wavefront scheme, as given in Table 1. The overall run times of BILU(0) and triangle solver can be obtained by summing all kernel run times. From Figure 8 we can find that the optimal per-thread processing times are roughly $1.0\mu s$ and $0.165\mu s$ for BILU(0) and triangle solver, respectively. Knowing the size of the block, we can now compare the optimal and actual per-thread run times. It is already known that roughly 30% of hyperplanes cannot supply enough computation load. Within these 30% hyperplanes we expect the larger ones causes less performance loss than the smaller ones. Using linear estimation, we can then expect roughly 15% overall performance loss caused by small hyperplanes. This estimation is consistent with our findings using per-thread time analysis.

Assuming we can somehow eliminate as much as 15% performance loss, the speedup by ILU(0) would only be around 2.6x. Note that the performance loss estimation given by Table 1 actually takes into account the overhead of launching multiple kernels for different hyperplanes. This suggest that the algorithms of BILU(0) has other significant source of performance loss. We believe that the primary suspect is the inefficient memory access pattern of BILU(0). Memory coagulation is a very important factor in deciding performance on GPU. The data dependency stencil of BILU(0) as shown in (2) suggest very little memory coagulation, that is, adjacent threads tend to access memory locations far from each other. Low data coagulation leads to low cache reuse. More specifically, for each point (i,j,k) the kernel needs to access seven 6x6 matrices (A~G) and three 6x6 temporary matrices necessary for matrix manipulation. Out of these ten 6x6 matrices, only one may overlap with the thread processing an adjacent point. Assume that per-thread data access amount is nine 6x6 matrices, which is $6 \times 6 \times 9 \times 8 = 2.6$KB of data. To fully load a CUDA wrap (32 threads), at least $2.6 \times 32 = 83$KB of data need to be accessed. Note that per-wrap data access amount is almost two times the L1 cache (48KB for data in the current Nvidia GPU's). It is likely that the actual per-thread data access amount, due to cache overflow, is significantly more than 2.6KB. As an estimation, the actual per-thread data access amount can be calculated as follows. Assume the overall bandwidth is 5GB/s, which is a typical value for codes with very low data coagulation. The actual per-thread execution time is $1.1\mu s$. The per-thread data throughput is then $5\text{GB} \times 1.1\mu s = 5.5$KB, which is more than twice the theoretical data



(a)



(b)

Figure 8: Per-thread process times of wavefront kernels for a $79 \times 49 \times 49$ block: (a) BILU(0), (b) triangle solver

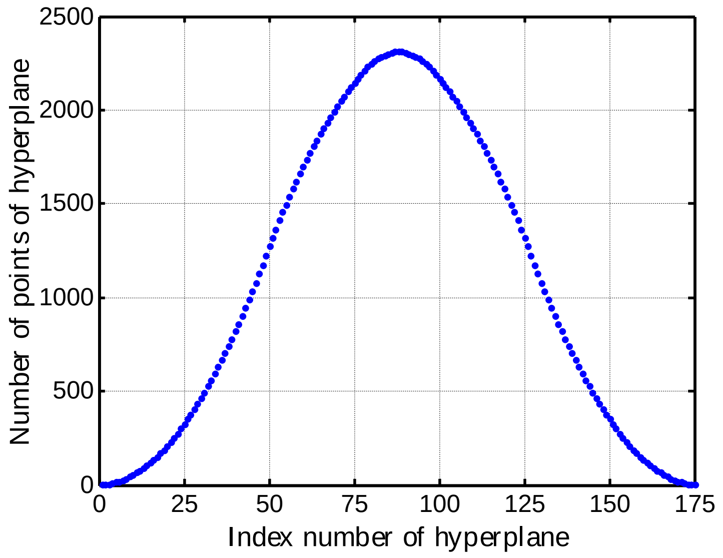


Figure 9: Number of points on different hyperplanes

access amount. For comparison, the CPU version of BILU(0) does not suffer a significant speed penalty due to cache overflow, because the CPU (Opteron 6128 in our case) has more than 16MB (4M L2 + 12M L3) of data cache available, usually enough to hold all the submatrices for the entire hyperplane (2.6KB per point).

5 Conclusion

In this study, an implicit multi-block incompressible Navier-Stokes solver is ported to nVidia GPGPU platform using OpenACC. 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. The GPU version of the code is about 3.5x faster than a CPU version, while the performance of the implicit method is found to be less impressive. The relatively poor performance of BILU(0) is analyzed in detail, which points out low data coagulation is deciding factor restricting speed, while insufficient load due to small hyperplanes also causes some degree of performance loss.

There are two directions which may lead to performance improvement of BILU(0). First of all, it may be possible to schedule multiple blocks on one GPU, which should allow more efficient scheduling of computation load as different wavefront planes from different blocks can be processed simultaneous, which may mitigate part of the performance loss due to insufficient load. Second, the BILU(0) can be improved by a more rigorous implementation on GPU to increase data coagulation and avoid cache overflow. One possibility is to use two-step implementation, which separate the inversion and multiplication of submatrices, so that neither of them can cause cache overflow.

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] T. Brandvik and G. Pullan, "Acceleration of a 3d euler solver using commodity graphics hardware," in *46th AIAA aerospace sciences meeting and exhibit*, 2008, p. 607.

[2] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens, "Rapid aerodynamic performance prediction on a cluster of graphics processing units," in *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, 2009, pp. 2009–565.

[3] D. Goddeke, S. H. Buijssen, H. Wobker, and S. Turek, "GPU acceleration of an unmodified parallel finite element navier-stokes solver," in *International Conference on High Performance Computing & Simulation, 2009. HPCS09.* IEEE, 2009, pp. 12–21.

[4] J. C. Thibault and I. Senocak, "CUDA implementation of a navier-stokes solver on multi-gpu desktop platforms for incompressible flows," in *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, 2009, pp. 2009–758.

[5] A. Corrigan, F. Camelli, R. Löhner, and F. Mut, "Semi-automatic porting of a large-scale fortran CFD code to GPUs," *International Journal for Numerical Methods in Fluids*, vol. 69, no. 2, pp. 314–331, 2012.

[6] Y. Xia, H. Luo, L. Luo, J. Edwards, J. Lou, and F. Mueller, "OpenACC-based GPU acceleration of a 3-D unstructured discontinuous galerkin method," in *52nd Aerospace Sciences Meeting*, 2014.

[7] D. A. Jacobsen, J. C. Thibault, and I. Senocak, "An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters," in *48th AIAA Aerospace Sciences Meeting and Exhibit*, vol. 16, 2010.

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

[9] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, no. PrePrints, p. 1, 2013.

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

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

[12] 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.

[13] 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.

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

[15] 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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] L. Luo, J. R. Edwards, H. Luo, and F. Mueller, "Performance assessment of a multiblock incompressible navier-stokes solver using directive-based gpu programming in a cluster environment," in *52nd Aerospace Sciences Meeting*, 2013.

[24] Y. Saad, *Iterative methods for sparse linear systems.* SIAM, 2003.

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

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