Advanced Optimizations of An Implicit Navier-Stokes Solver on GPGPU

Lixiang Luo, Jack R. Edwards, Hong Luo, Frank Mueller

September 15, 2014

North Carolina State University

1 Introduction

General-purpose computing on graphics processing units (GPGPU) is a massive fine-grain parallel computation platform, which is is particularly attractive for CFD tasks due to its potential of one or two magnitudes of performance improvement with relatively low capital investment. Many successful attempts have been reported in recent years (see, for example [1, 2, 3, 4, 5, 6]). Although early attempts of utilizing GPGPU for CFD has been hampered by the heterogeneous nature of GPGPU hardware and complex programming tools, recent GPU technology has seen significant improvement on programming toolchain. The emergence of OpenACC [7], a directive-based programming model closely resembling OpenMP, further reduces the obstacles for efficient GPGPU programming. By allowing programmers to use a collection of compiler directives to specify loops and regions of their codes to be offloaded from a host CPU to GPGPU, this programming model offers a good balance between porting efforts and performance gain. As shown in [8, 6], our previous attempts of porting existing CFD codes to GPGPU using OpenACC and MVAPICH2 [9] have proven the advantages on portability and maintainability of this approach.

Even with the assistance of convenient development tools many of the challenges in GPGPU can be resolved, including memory contingency and efficient data exchange between multiple GPU's. A more fundamental performance restriction is encountered when porting implicit time integration schemes. Employing an implicit time marching instead of explicit time marching 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. From a more general point of view, governing equations of many complex systems are so stiff that an explicit method may never converge under any reasonable CFL number. One prime example is multiphase reactive flows, which, due to their incorporation of a large number of variables and equations for describing the complex thermodynamical and chemical processes, often result in highly stiff linear systems. Another example is highly streched computational grids due to unusual boundary shapes of the domain, which also result in stiff linear systems. Even formulated as an implicit problem, the resulting system matrix is often non-diagonally

dominant or weakly diagonally dominant. Because direct solution of a large sparse matrix is prohibitively costly, the best option is usually iterative solution, whose efficiency is primarily determined by the quality of the preconditioner. Unfortunately, good preconditioners that can handle ill-posed linear systems often involve inherently sequential algorithms, causing great difficulty in parallelization. Furthermore, in 3D problems such linear system easily reaches enormous sizes, requiring large amount of storage. Considering the performance of the whole implicit CFD solver is often dictated by the linear solver, it is critical to design efficient parallel algorithms for solving large sparse linear systems.

Our study on implicit methods focus on an implicit 3D LES incompressible Navier-Stokes solver previously ported onto the GPU architecture using OpenACC and MVAPICH2 [9]. The solver was based on an CPU-based version validated for a range of model problems [10, 11]. 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 [12]. 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 [13, 14], two-phase flows [15], and human-induced contaminant transport [16, 17]. An immersed-boundary method [11] is incorporated to enable computations of flow about moving objects.

The implicit solver has been 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. Because of the relatively small bandwidth between host and device memory, data transfers is carefully planned in order to avoid any unnecessary transfers. Due to the fine granularity by GPU, the CFD algorithm must be revisited so that it can be parallelized over hundreds, or preferably, thousands of independent threads. Due to the shared-memory parallelism nature of OpenACC, memory contingency is a commonly encountered issue. 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 two-pass or three-pass coloring scheme can be employed to resolve the memory contingency.

The size of 3D problems demands partitioning, particularly due to the fact that GPU has less memory than the CPU. To take full advantage of the InfiniBand [18] interconnection of our cluster, MVAPICH2 is selected as the MPI implementation to handle the inter-process data exchange [19]. 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, if an accelerated data packing kernel is provided [20, 21]. Although currently a manual data packing scheme is used in the current version [8], upcoming study will likely to utilize the data pipelining capability.

A GPU port of the implicit solver has been completed [22]. However, test results found that the overall performance is only 3x faster than the equivalent code run on CPU. Although the performance gain is only slightly less than the explicit version of the solver, we believe that significant improvements can be made for at least two components of the program.

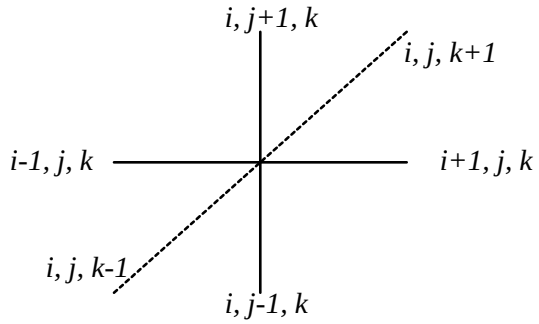

Figure 1: 7-point stencil associated with the structured 3D finite volume method

First, inter-nodal data exchange is slow. To improve this we will attempt to utilize GPUDirect RDMA capability to speed up the MPI data transfer. With the current Nvidia hardware and MPI implementation the data exchange between different GPU's on different compute nodes must go through the GPU-CPU memory transfer bottleneck. This is found to be a significant performance restriction. Recently the GPUDirect RDMA (Remote Direct Memory Access) technology has matured for production applications [23]. It allows GPU's on different cluster nodes to communicate via InfiniBand network without going through the main CPU memory, which can reduce data transfer overhead between GPU's over a cluster. Note that the application of this technology in our CFD code may be restricted by compiler support, library support and hardware readiness.

Second, BILU(0) factorization is found to have significantly less speedup when ported to GPU, due to extremely low data coagulation. We plan to introduce a redesigned a two-stage algorithm which has much lower granularity, in order to gain better data coagulation and reduce memory bandwidth demand. In the following sections, an brief study on performance limitations of BILU(0) is given, followed by the optimization proposal which can significantly improve the performance of BILU(0).

2 Performance Limitations of Current Implicit Codes

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.

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

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

The 3D finite volume scheme involves a 7-point stencil as shown in Figure 1. This stencil results in a system matrix A with the pattern given in Figure 2.

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 same dimension of 6x6, due to the fact that there are 6 primitive variables to be solved at each grid cell.

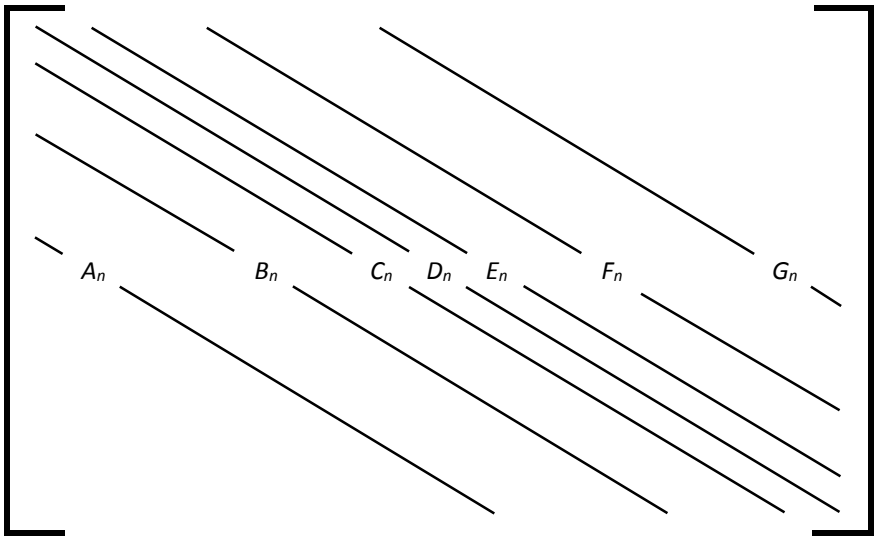

Figure 2: Pattern of matrix A associated with 7-point finite volume method

In an implicit solver, major portion of the computation resource is used to solve the linear system as a result of the discretized governing equation. Hence, an efficient block-sparse linear system solver becomes the focus of implicit solver porting effort, which will be further discussed in the next section.

Block-incomplete-LU decompsition 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 directly.

To extract parallelism from this type of data dependency the "wavefront" ordering scheme is often used. 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 3. 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 4. A detailed study on wavefront ordering scheme can be found in [24]

Once all $\hat{D}$ are found, the ILU(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 lower block-diagonal of $\mathbb{A}$, and $-\mathbb{U}$ is the upper block-diagonal of $\mathbb{A}$. This factorization allows a fast approximation solution of the original linear system by solving two block-triangle linear systems using simple forward and back substitutions. 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

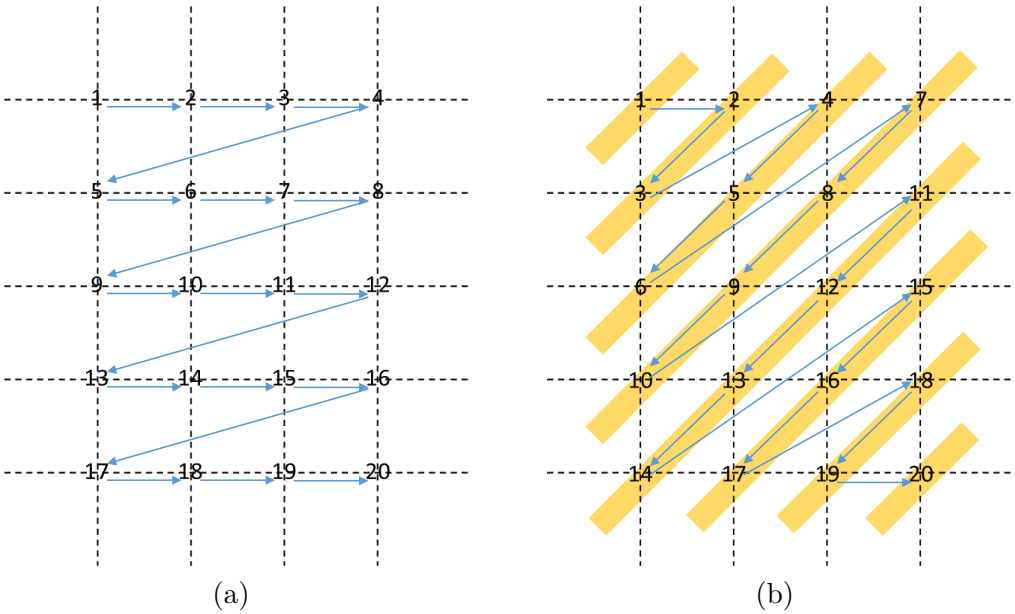

Figure 3: (a) Naive ordering; (b) wavefront ordering

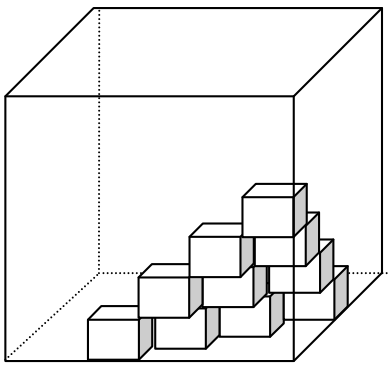

Figure 4: A wavefront plane in a 3D block

solution, corrections can be applied iteratively on the approximated solution, using the same $\hat{\mathbb{D}}$ calculated by the BILU(0).

A naive version of BILU(0) has been implemented using OpenACC. Our most recent study [22] found that its performance gain is significantly less than other computation-intensive subroutines, reaching about only 2.2x speedup. The same study concludes that the primary reason for poor performance is extremely low data coagulation and serious cache overflow. Another contributing factor is insufficient load due to small hyperplanes at the two corners of the block.

3 Proposed Optimizations

To improve the performance of BILU(0), the algorithm must be redesigned for finer granularity. In the current implementation, each CUDA thread must carry out the entire calculation of (2) for one grid cell, thus demanding large amount of data which can not be coagulated with adjacent grid cells (threads). Essentially, the current algorithm stores the LU decomposition of each $\hat{D}_{i,j,k}$. The computations of terms such as $C_{i,j,k}\hat{D}^{-1}_{i-1,j,k}$ are actually carried out as solving a linear system by two triangle sweeps. Due to the sequential nature of triangle sweeps it is not possible for this algorithm to be parallelised beyond the cell level.

On the other hand, if the inverse of $\hat{D}_{i,j,k}$ is stored instead, BILU(0) iteration can then be expressed as purely matrix multiplications and matrix inversions, each of which can be further parallelized. Specialized GPU kernels algorithms that can carry out matrix multiplication very efficiently, with high level of data coagulation. Cache overflow should not occur in fine-grain matrix multiplication.

Such algorithm, however, require the calculation the inverse of submatrices $D_{i,j,k}$, $D_{i,j-1,k}$ and $D_{i,j,k-1}$ in advance of the matrix multiplications. Fortunately, the required inversed matrices are all on the adjacent hyperplane defined by $i + j + k = Z - 1$. This allows us to use a temporary array to store all the $\hat{D}^{-1}$ matrices on the $Z-1$ hyperplane. Highly efficient algorithm for direct matrix inversion on GPU is readily available (for example, [25]).

In this new two-stage BILU(0) algorithm, we sacrifices storage of $\hat{D}^{-1}$ for one hyperplane, in exchange for higher granularity in the matrix multiplication stage, which will have very high data coagulation and zero cache overload. Even for a large hyperplane with 5000 points, the extra storage would be under 1.5MB. Although the inversion stage will likely suffer from relatively low data coagulation due to the size of the each D submiatrix (6x6), there should be no cache overflow as each thread only carry out one 6x6 inversion. Significant benefit is expected in the matrix multiplication stage. If designed properly, matrix multiplications on GPU have the potential to achieve over 100x speedup. Finally, this algorithm can well adapt systems which have more unknowns (must be less than 12 for current CUDA architectures, otherwise inversion calculation itself causes cache overflow). Eventually, the overall performance of BILU(0) will likely to be restricted only by the efficiency of the inversion stage.

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] *The OpenACC Application Programming Interface*, 2013.

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

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

[10] 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.

[11] 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.

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

[13] 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

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

[22] L. Luo, J. R. Edwards, H. Luo, and F. Mueller, "Gpu port of a parallel incompressible navier-stokes solver based on openacc and mvapich2," in *AIAA Aviation and Aeronautics Forum and Exposition*, 2014.

[23] Product brief: Mellanox ofed gpudirect rdma. Mellanox.

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

[25] G. Sharma, A. Agarwala, and B. Bhattacharya, "A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA," *Computers & Structures*, vol. 128, pp. 31–37, 2013.