Optimization of A Fine-grained BILU by CUDA Inter-block Synchronization

Lixiang Luo, Jack R. Edwards, Hong Luo, Frank Mueller
North Carolina State University

Wu-chun Feng
Virgina Tech

Abstract

A fine-grained block incomplete LU (FGBILU) factorization for solving large-scale block-sparse linear systems resulting from coupled PDE systems with n equations has been recently developed for massively parallel heterogeneous architectures, such as general-purpose graphics processing units (GPGPUs). A straightforward one-sweep wavefront ordering is combined with element-wise block submatrix operations, allowing FGBILU to achieve low-overhead concurrent computation at $O\left(n^2N^2\right)$ scale on a 3D PDE domain with a linear scale of N. Numerical experiments show that FGBILU is less efficient on smaller domains. Besides the inevitable performance penalty of a wavefront ordering, the index reconstruction by each concurrent computation thread causes considerable parallelism overhead. One way to reduce the overhead is to employ thread recycling along with CUDA inter-block synchronization. Dynamic parallelism is also attempted, although with no significant perforamnce benefit. The improved FGBILU is tested for a series of 3D PDE domains extracted from an incompressible Navier-Stokes solver called INCOMP3D. Results show that thread recycling can significantly reduce parallelism overhead and improve the performance of FGBILU on smaller domains.

1 Introduction

It is widely known that implicit time marching can significantly improve CFD simulation speed when the physical time scale is relatively large, as is often the case in incompressible flows. Also, in many complex physics processes governing equations 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 stretched computational grids due to unusual boundary geometry, which tend to cause local stiffness. Implicit methods for a complex physical process usually result in a large block sparse linear system. Because direct solution of a large sparse matrix is prohibitively costly, the best option is usually an iterative method, whose efficiency is primarily determined by the quality of the preconditioner.

Recent development on hardware and software technologies of General Purpose Graphics Processing Unit (GPGPU) has greatly improved its potential for large-scale high performance computation (see, for example [1, 2, 3, 4, 5]). Maturing programming frameworks

have allowed GPGPU algorithm designs to gain more popularity. Particularly, the emergence of OpenACC [6], a directive-based programming model closely resembling OpenMP, significantly reduces the obstacles of efficient GPGPU programming. With the assistance of convenient development tools many software engineering issues of porting CFD codes can be trivially resolved, including data management and efficient data exchange between multiple GPGPUs. Our previous attempts of porting CFD codes to GPGPU using OpenACC and GPGPU-aware MPI implementations have proven the advantages on portability and maintainability of this approach [2, 7]. More fundamental performance restrictions are encountered when porting implicit CFD solvers. Much of the difficulty is contributed by poor performance of the iterative linear solver, a core component of most implicit CFD solvers. Mathematically efficient solvers often involve highly sequential algorithms for preconditioning, which can be very difficulty to implement on fine-grained massively parallel architectures such as GPGPU.

Our study on implicit methods focus on an implicit 3D LES incompressible Navier-Stokes solver called IN3D [7], which has been ported onto the GPU architecture using OpenACC [6] and GPGPU-aware MPI [8, 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. A full GPU port has been implemented [7]. However, the performance of the linear solver is rather poor, as it is found to be extremely memory bound.

The solution is to refine granularity for the linear solver, whose benefit is two-fold. First, to obtain optimal speed of a GPGPU, its cores should be kept as busy as possible. Finer granularity allows the algorithm to be divided into smaller units, thus occupying more cores with the same total amount of work. Meanwhile, the amount of information each core processes is kept as little as possible. Unfortunately, fine granularity also introduces significant parallelism overhead. Particularly, the overhead associated with launching separate kernels for each wavefront and the overhead associated with reconstructing array indices for each concurrent thread, which scales with a rate of n^2N^2. Although index reconstructions are fast integer operations, such scaling would nevertheless cause considerable performance penalty.

It is possible to reduce this overhead by employing thread recycling. Instead of spawning new threads for different grid locations, each thread runs a loop to fetch new grid points to process, avoiding a large portion of the index construction that is constant. Because the size of a wavefront is usually larger than a CUDA block, inter-block synchronization must be manually managed to resolve data dependency. The most common method to achieve this is to use a global variable and atomic operation. Such approach is commonly referred to as "task-based parallelism" by Nvidia. In first-generation Nvidia GPGPUs, this is not recommended as atomic operations are rather inefficiently implemented, resulting in very long synchronization time if the number of streamline multiprocessors is large. However, recently

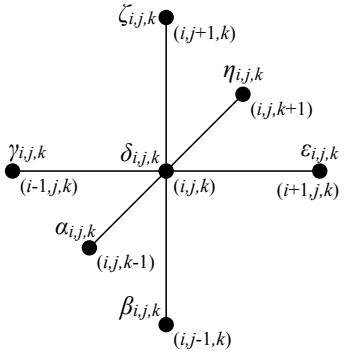


Figure 1: 7-point stencil on a regular 3D structure

Nvidia GPGPUs have greatly improved the performance of atomic operations, making this approach much more desirable.

This paper is organized as follows. First, a brief introduction of BILU is given in Section 2, along with an enhanced FGBILU scheme. The scheme includes algorithms for the factorization, triangular solve and iterative correction processes. In Section 3, FGBILU algorithms with thread recycling are discussed in detail, including two methods of achieving CPU-free inter-block synchronization. The performance of the new scheme is studied in numerical experiments presented in Section 4, where FGBILU is employed as a block sparse linear system solver for an incompressible Navier-Stokes solver called INCOMP3D. The paper concludes with discussions on further improvements and potential applications of FGBILU scheme.

2 BILU on a regular 3D structured domain

The linear solver of IN3D is based on BILU(0), which stands for incomplete LU factorization with zero fill-ins for block-sparse linear systems. The "zero fill-ins" phrase is omitted from now on for clarity. Incomplete LU factorization was originally developed as approximate inversion schemes by matrix splitting for sparse matrices [18]. The block-wise extension arises naturally from coupled multi-component PDE systems and found to be a better choice for this kind of applications in general [19, 20].

Consider discretization of a coupled PDE system with n unknowns on a structured 3D domain with a straightforward 7-point stencil, as shown in Figure 1. $0 \leq i < I$, $0 \leq j < J$ and $0 \leq k < K$ are grid cell indexes and the corresponding dimensions of the domain. Implicit time integration on such a stencil usually results in a block-sparse matrix linear system $A\mathbf{x} = \mathbf{b}$, where the left-hand-side coefficient matrix takes a form as shown in Figure 2. Each element therein is a $n \times n$ submatrix. Each row corresponds to the discretization centered on a certain grid point. A typical non-boundary grid point (i, j, k) produces totally 7 block submatrices with the same subscript i, j, k, which is reflected in Figure 1.

BILU with zero fill-in attempts to approximate A with LU, such that LU has the exact same zero pattern of A. BILU with zero fill-ins ignores all off-stencil multiples of the submatrices. As a result, the submatrices of the factorization can be found by a comparison of submatrices for a general grid point (i, j, k) on its 7-point stencil. Simple analysis [21] reveals that the factorization only need to be carried out for the diagonal blocks $\delta_{i,j,k}$, using

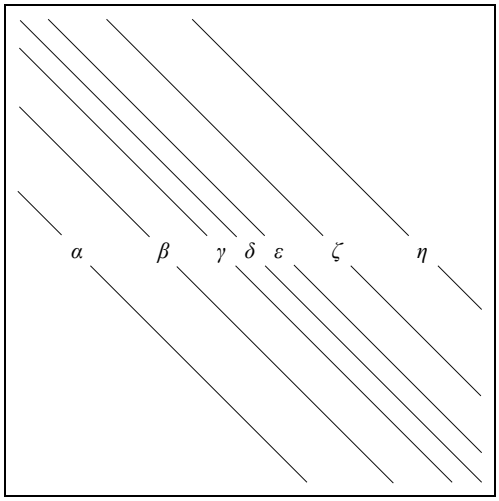


Figure 2: Coefficient matrix A

the following recurrence:

$$d_{i,j,k} = \delta_{i,j,k} - \alpha_{i,j,k} d^{-1}_{i,j,k-1} \eta_{i,j,k} - \beta_{i,j,k} d^{-1}_{i,j-1,k} \zeta_{i,j,k} - \gamma_{i,j,k} d^{-1}_{i-1,j,k} \varepsilon_{i,j,k} \,. \tag{1}$$

Note that all $d_{i,j,k}$ can be safely assumed non-singular for most physical systems. Out-of-bounds array references are treated as zero. The BILU factorization can then be expressed as:

$$LU = (D + E) \left(I + D^{-1} F \right) ,$$

where D is the block-diagonal matrix of the factorized diagonals $d_{i,j,k}$, calculated using Eq. 1. E is the strict lower block-diagonal of A and F is the strict upper block-diagonal of A.

The factorization leads to an approximation of the original linear system, which can be now be solved by two triangle sweeps:

$$\begin{aligned} (D + E)\, \mathbf{y} &= \mathbf{b} \,, \\ \left(I + D^{-1} F \right) \hat{\mathbf{x}} &= \mathbf{y} \,, \end{aligned}$$

which, if written in point-wise form, gives the following recursive formulation:

$$y_{i,j,k} = d^{-1}_{i,j,k} \left[b - (\alpha_{i,j,k} y_{i,j,k-1} + \beta_{i,j,k} y_{i,j-1,k} + \gamma_{i,j,k} y_{i-1,j,k}) \right] , \tag{2}$$

$$\hat{x}_{i,j,k} = y_{i,j,k} - d^{-1}_{i,j,k} \left(\varepsilon_{i,j,k} \hat{x}_{i+1,j,k} + \zeta_{i,j,k} \hat{x}_{i,j+1,k} + \eta_{i,j,k} \hat{x}_{i,j,k+1} \right) . \tag{3}$$

Finally, the approximated solution $\hat{x}$ can be further improved by recursively solving a similar linear system for the residual:

$$M \xi_l = \mathbf{b} - A \hat{\mathbf{x}}_l \,, \quad \hat{\mathbf{x}}_{l+1} = \hat{\mathbf{x}}_l + \xi_l, \tag{4}$$

until the residual $\mathbf{R}_l = \mathbf{b} - A\hat{\mathbf{x}}_l$ diminishes to a certain level. The subscript l is the iterative index. In a multi-block PDE solver, block coupling can be achieved by exchanging $\mathbf{R}_l$ using message passing before each iteration.

Algorithm 1 Generation of wavefront ordering mapping

1: ***for*** $(i = 0 \dots I-1)$ ***for*** $(j = 0 \dots J-1)$ ***for*** $(k = 0 \dots K-1)$ **{**
2: $\quad p \leftarrow i + j + k$;
3: $\quad N_p \leftarrow N_p + 1$;
4: $\quad W_{0,N_p,p} \leftarrow i$; $W_{1,N_p,p} \leftarrow j$; $W_{2,N_p,p} \leftarrow k$;
5: **}**

2.1 Parallel BILU by wavefront ordering

Although the BILU factorization (Eq. 1) and the triangular solves (Eq. 2 and 3) are both in recursive forms, parallelism can be extracted for all grid points with a constant sum of subscripts $p = i+j+k$, as they form a data-independent subset. Such scheme of identifying a sequence of data-independent subsets ("levels") is called *level scheduling*. For a 3D structured domain, a level defined in this way forms a diagonal cross-section, called a hyperplane or a wavefront, which can be uniquely identified by p. The number of data-independent points on hyperplane p is written as N_p. In a parallel algorithm, the grid points are processed in a wavefront ordering. Each grid point is assigned an index q, unique within its hyperplane, so that every grid point in the 3D domain can be uniquely identified by an index pair (p, q). A simple 3D traverse, as given in Algorithm 1, can be used to generate a list of hyperplane sizes N_p, and a mapping $W : (p, q) \rightarrow (i, j, k)$, which is stored as a three-dimensional array.

Parallel factorization and triangular solve are given in Algorithm 2 and 3. In a wavefront scheme, the sequential outer loop resolves data dependency, while the parallel inner loop processes all the data point on a hyperplane concurrently. Such a parallel loop is designated as a "kernel", with index variables and their ranges listed inside parenthesis. One permutation of indices is the smallest unit for concurrent execution in a parallel implementation. For clarity, an implied mapping W is always carried out inside a wavefront scheme kernel.

Algorithm 2 Coarse-grained parallel BILU factorization

1: ***for*** $(p = 0 \dots I + J + K - 3)$
2: $\quad$ ***kernel*** $(q = 0 \dots N_p - 1)$ **{**
3: $\quad\quad d_{i,j,k} \leftarrow d_{i,j,k} - \alpha_{i,j,k} d_{i,j,k-1} \eta_{i,j,k} - \beta_{i,j,k} d_{i,j-1,k} \zeta_{i,j,k} - \gamma_{i,j,k} d_{i-1,j,k} \varepsilon_{i,j,k}$;
4: $\quad\quad d_{i,j,k} \leftarrow (d_{i,j,k})^{-1}$;
5: $\quad$ **}**

Algorithm 3 Coarse-grained parallel BILU triangular solve

Forward sweep

1: ***for*** $(p = 0 \dots I + J + K - 3)$
2: $\quad$ ***kernel*** $(\ q = 0 \dots N_p - 1\ )$
3: $\quad\quad \hat{x}_{i,j,k} \leftarrow d_{i,j,k}(b_{i,j,k} - \alpha_{i,j,k} \hat{x}_{i,j,k-1} - \beta_{i,j,k} \hat{x}_{i,j-1,k} - \gamma_{i,j,k} \hat{x}_{i-1,j,k})$;

Backward sweep

4: ***for*** $(p = I + J + K - 3 \dots 0)$
5: $\quad$ ***kernel*** $(\ q = 0 \dots N_p - 1\ )$
6: $\quad\quad \hat{x}_{i,j,k} \leftarrow \hat{x}_{i,j,k} - d_{i,j,k}(\varepsilon_{i,j,k} \hat{x}_{i+1,j,k} + \zeta_{i,j,k} \hat{x}_{i,j+1,k} + \eta_{i,j,k} \hat{x}_{i,j,k+1})$;

Coarse-grained BILU factorization and triangular solve algorithms on CPU have been

developed based on Algorithm 2 and 3, whose kernels are implemented as OpenMP parallel regions. Instead of an direct inversion, as indicated at Line 4 of Algorithm 2, an LU decomposition is employed, which allows the multiplication with $d_{i,j,k}^{-1}$ as two triangular sweeps.

2.2 Fine-grained algorithms on GPGPU

Although GPGPU programming generally follows OpenMP-like shared-memory parallelism, some distinct challenges must be addressed. A GPGPU has hundreds of computation cores, each capable of executing one instance of the kernel code, which is called a "thread". GPGPU threads are logically organized as "workgroups". The size of a workgroup and the total number of workgroups can both be programmed. Within each workgroup, local shared memory can be used for data sharing. A certain number (fixed by hardware design) of threads, called a "warp", are executed in synchronized lock-step, much like a SIMD operation in CPU. The actual execution of warps is out-of-order, but explicit synchronization points can be imposed within a workgroup.

When implementing the BILU algorithms, the primary performance constraint is found to be memory boundedness [22]. If one grid point is mapped to one GPU thread, the coarse-grain factorization would require exceedingly large cache to achieve reasonable locality, which is not practical with current GPGPU hardware. Furthermore, the coarse-grained algorithm quickly becomes intractable as the number of PDE equations increases, because the amount of data per thread scales with n^2. The solution to this problem is fine-grained algorithms that can map the calculation on the element (scalar) level. This cancels the per-thread scaling factor of n^2, greatly easing the demand on cache memory. However, matrix operations have internal data dependencies, which must be addressed correctly and efficiently. For parallel paradigms with a moderate overhead, such as OpenMP, the benefit of extracting parallelism from matrix operation is often negated by heavy synchronization cost. Low-overhead solutions, such as SIMD operations, are often restricted by poor support of branching instructions and lack of local storage [23]. A GPGPU, on the other hand, combines the advantage of very low synchronization overhead (within a workgroup) and the capability of branching operations, which makes the element-level parallelism a much more practical approach.

The basic idea of FGBILU can be considered as a two-tier parallelism. BILU has two inherent granularity levels. As discussed in Section 2.1, concurrent processing of all grid point in a hyperplane allows coarse-grained parallelism. On the next tier, fine-grained parallelism can be achieved by vectorizing submatrix operations in a element-wise fashion. This two-tier structure maps very well to GPGPU, which also has a well-defined two-tier structure. Concurrent processing of grid points can be mapped to multiple workgroups, while vectorized submatrix operations can be mapped to threads. Data dependency across hyperplanes can be implemented as sequential kernel launches, in the same way as OpenMP. It is possible to achieve synchronization across workgroup using other methods, which will be discussed in following sections.

In the descriptions that follow, grid location triplets (i, j, k) in subscripts of domain submatrix and vector variables, namely, α, β, γ, δ, ε, ζ, η, d, b, $\hat{x}$ and R, are simplified to reduce clustering in pseudocodes. Only the deviation from the center of the 7-point stencil is explicitly indicated. Any submatrix variable without the triplet is taken at the stencil center (i, j, k). For example, $\alpha_{i,j-1,k}^{u,v}$ is simply written as $\alpha_{j-1}^{u,v}$, and $\beta^{u,v}$ actually means $\beta_{i,j,k}^{u,v}$. Submatrix and vector variables are easily distinguished from scalar variables, which do not

Algorithm 4 FGBILU factorization

Local shared memory array: $s_{3,L}^{n,n}$. A second subscript, q_w, is implied in all s references.

kernel ($(I_G,\, I_T) \to (i,\, j,\, k,\, u,\, v,\, q_\mathrm{w})$) **{**
  $s_0^{u,v} \leftarrow \langle \alpha^{u,:} \,,\, d_{k-1}^{:,v} \rangle$; $s_1^{u,v} \leftarrow \langle \beta^{u,:} \,,\, d_{j-1}^{:,v} \rangle$; $s_2^{u,v} \leftarrow \langle \gamma^{u,:} \,,\, d_{i-1}^{:,v} \rangle$;
  syncthreads
  $d^{u,v} \leftarrow d^{u,v} - \langle s_0^{u,:} \,,\, \eta_{k-1}^{:,v} \rangle - \langle s_1^{u,:} \,,\, \zeta_{j-1}^{:,v} \rangle - \langle s_2^{u,:} \,,\, \varepsilon_{i-1}^{:,v} \rangle$;
  syncthreads
  for $(r = 0 \ldots n-1)$ **{**
    $t \leftarrow 1 - t$; $a \leftarrow (u = r ? 1 : 0)$; $b \leftarrow (v = r ? 1 : 0)$; $c \leftarrow (1-a)(1-b)$;
    if $(c = 0)$ $s_t^{u,v} \leftarrow d^{u,v}$;
    syncthreads
    $d^{u,v} \leftarrow [ab + (a-b)\, d^{u,v} - c\, s_t^{u,r}\, s_t^{r,v}] / s_t^{r,r} + c d^{u,v}$;
  }
}

Algorithm 5 Gauss-Jordan elimination without pivoting for an $n \times n$ matrix

for $(r = 0 \ldots n-1)$ **{**
  for $(u = 0 \ldots n-1)$ ***if*** $(u \neq r)$ $d^{u,r} \leftarrow -d^{u,r} / d^{r,r}$;
  for $(v = 0 \ldots n-1)$ ***if*** $(v \neq r)$ $d^{r,v} \leftarrow d^{r,v} / d^{r,r}$;
  for $(u, v = 0 \ldots n-1)$ ***if*** $(u \neq r \;\&\&\; v \neq r)$ $d^{u,v} \leftarrow d^{u,v} + d^{u,r} d^{r,u} d^{r,r}$;
  $d^{r,r} \leftarrow 1 / d^{r,r}$;
}

have superscripts.

In GPGPU programming, each thread is assigned a workgroup index I_G and a thread index I_T, both 0-based. All other indices for mathematics can be reconstructed with Algorithm 6, where integer division ($\backslash$) is used extensively. Once q is calculated, (i, j, k) can be further determined by mapping W. For clarity, index reconstruction appears as a mapping from $(I_G,\, I_T)$ to $(i,\, j,\, k,\, u,\, v,\, q_\mathrm{w})$ in the kernel header.

FGBILU factorization is given in Algorithm 4. While the calculation of the off-diagonal multiples and their sum can be embarrassingly parallel, the inversion of diagonals is a well-known bottleneck of vectorizing BILU algorithms. In FGBILU, an in-place direct inversion algorithm based on the Gauss-Jordan elimination (GJE) is employed, whose general sequential pseudocode is given in Algorithm 5. Shared memory array$s$ is a temporary storage for resolving data dependency. Its first subscript ranges from 0 to 2, corresponding to the three spatial components. The second subscript q_w, which is always implied in pseudocodes, gives the point index within a workgroup. The two superscripts are submatrix indices, each ranging from 0 to $n-1$. A dot product is written as $\langle \cdot, \cdot \rangle$. Fortran-style colon operators are used to indicate the range of a certain dimension. Note that the use of s_0 and s_1 as read buffers for resolving data dependency in GJE. For each $n \times n$ submatrix this inversion algorithm generates roughly $2n^3$ floating point operations. For comparison, a pure sequential algorithm would generate n^3 floating point operations. Because the parallel GJE uses n^2 threads per matrix, the computation time should be at the scale of $2n^3/n^2 = 2n$, which is much less than a sequential algorithm.

The triangular solves involve mostly matrix-vector multiplications. As shown in Algo-

Algorithm 6 Index reconstruction

1: $q_{\mathrm{w}} \leftarrow I_T \backslash n^2$;
2: $q \leftarrow I_G L + q_{\mathrm{w}}$;
3: $v \leftarrow I_T \backslash n - n q_{\mathrm{w}}$;
4: $u \leftarrow I_T - n^2 q_{\mathrm{w}} - n v$;

Algorithm 7 FGBILU triangular solve

Local shared memory array: $s_L^{n,n}$, z_L^n. A subscript, q_{w}, is implied in all s and$z$ references.
(a) Forward sweep

1: ***kernel*** ($(I_G,\, I_T) \rightarrow (i,\, j,\, k,\, u,\, v,\, q_{\mathrm{w}})$) **{**
2: $\quad s^{u,v} \leftarrow \alpha^{u,v}\, \hat{x}^v_{k-1} + \beta^{u,v}\, \hat{x}^v_{j-1} + \gamma^{u,v}\, \hat{x}^v_{i-1}$;
3: $\quad$ ***syncthreads***
4: $\quad$ ***if*** $(v = 0)$ $z^u \leftarrow b^u - \sum s^{u,:}$;
5: $\quad$ ***syncthreads***
6: $\quad s^{u,v} \leftarrow d^{u,v}\, z^v$;
7: $\quad$ ***syncthreads***
8: $\quad$ ***if*** $(v = 0)$ $\hat{x}^u \leftarrow \sum s^{u,:}$;
9: **}**

(b) Backward sweep

1: ***kernel*** ($(I_G,\, I_T) \rightarrow (i,\, j,\, k,\, u,\, v,\, q_{\mathrm{w}})$) **{**
2: $\quad s^{u,v} \leftarrow \varepsilon^{u,v}\, \hat{x}^v_{i+1} + \zeta^{u,v}\, \hat{x}^v_{j+1} + \eta^{u,v}\, \hat{x}^v_{k+1}$;
3: $\quad$ ***syncthreads***
4: $\quad$ ***if*** $(v = 0)$ $z^u \leftarrow \sum s^{u,:}$;
5: $\quad$ ***syncthreads***
6: $\quad s^{u,v} \leftarrow d^{u,v}\, z^v$;
7: $\quad$ ***syncthreads***
8: $\quad$ ***if*** $(v = 0)$ $\hat{x}^u \leftarrow \hat{x}^u - \sum s^{u,:}$;
9: **}**

rithm 7, the fine-grained algorithm further divides the dot product into element-wise operations, which refines the granularity by a factor of n. The thread-level synchronization with the assistance of a local shared memory array allows the merge of data-dependent operations.

During correction iterations the factorization **d** and the triangular solve procedure remain constant. The residual, $\mathbf{R} = \mathbf{b} - A\hat{\mathbf{x}}$, must be calculated, which is then used as the RHS for the upcoming correction step. The residual takes a simple form (Eq. 4), which can be embarrassingly parallelized without level scheduling, as given in Algorithm 8.

Algorithm 8 FGBILU residual calculation in correction steps

1: ***kernel*** ($i = 0 \ldots I-1,\, j = 0 \ldots J-1,\, i = 0 \ldots K-1,\, u = 0 \ldots n-1$)
2: $\quad R^u \leftarrow b^u - \langle \alpha^{u,:}\,,\, \hat{x}^{:}_{k-1} \rangle - \langle \beta^{u,:}\,,\, \hat{x}^{:}_{j-1} \rangle - \langle \gamma^{u,:}\,,\, \hat{x}^{:}_{i-1} \rangle$
$\quad -\langle \delta^{u,:}\,,\, \hat{x}^{:} \rangle - \langle \varepsilon^{u,:}\,,\, \hat{x}^{:}_{i+1} \rangle - \langle \zeta^{u,:}\,,\, \hat{x}^{:}_{j+1} \rangle - \langle \eta^{u,:}\,,\, \hat{x}^{:}_{k+1} \rangle$;

3 CUDA-specific Optimizations

The major concern of FGBILU is its use of wavefront ordering, which is always less efficient on smaller domains. However, there are methods to mitigate its performance impact. Specifically, we can try to lower the synchronization overhead. In this section, we introduce two attempts to improve the performance of FGBILU. The first the a feature called dynamic parallelism, recently introduced to CUDA devices with compute capability 3.5 or above. It allows kernel to be launched on a GPGPU directly, without any involvement of the CPU. The second is a CUDA programming technique called thread recycling, which allows a workgroup to "fetch" multiple grid points to work on. Combined with a lock-based synchronization scheme, this technique can achieve CPU-free synchronization and overhead reduction by eliminating redundant index reconstruction. To expose the technical details of these optimizations, the pseudocodes in this section will be written in a more detailed manner.

3.1 CPU-free inter-workgroup synchronization by dynamic parallelism (DP)

In FGBILU's two-tier parallelism, the data dependency of hyperplanes are resolved by synchronization across workgroups working on the hyperplane, which is traditionally achieved by separate kernel launches for each hyperplane, initiated from host (CPU) side. As arguments must be passed from CPU memory to GPU memory, host-side kernel launching is always associated with an overhead. If the kernel launching can avoid data exchange between CPU and GPU, this overhead can be largely eliminated.

CUDA Dynamic Parallelism is introduced as a mechanism to launch new kernels (child kernel) within an active kernel (parent kernel). The launch is fully nested, so that data consistency is preserved for consecutive child kernels. This capability can be used to move the level scheduling from CPU to GPGPU completely. The migration from a traditional level scheduling using CPU kernel launches is trivial. First, the hyperplane size list N_p needes to be copied to GPGPU. Then, a master control kernel with only one thread is launch, which carries out the kernel launches, in the exact same way as CPU kernel launches. By removing any intervention of the CPU during level scheduling, it may be able to reduce overheads and improve the overall performance of the wavefront scheme.

3.2 Thread recycling (TR)

Although implied in the pseudocodes, the index reconstruction is a significant overhead of the fine-grained parallel algorithms. In a one-to-one mapping, each thread must calculate the same grid location indecies (i, j, k) and the local matrix element index (u, v) for *every* matrix element. Instead, thread recycling establish a one-to-many mapping from a thread to elements in many grid points. In such approach, each GPGPU workgroup repeatedly "fetches" new grid points to process. One can observes that the local matrix element index (u, v) can be fixed while (i, j, k) is changed for new grid points. In fact, u, v and q_w can all be fixed as the mapping to a matrix element is solely determined by the relative position of a thread within a workgroup. This can potentially save a significant amount of overhead on index reconstruction. To differentiate from traditional data parallelism, this programming approach is called task-based parallelism by Nvidia.

However, overhead reduction by thread recycling is only beneficial if each workgroup

...	25~28	12N					
...	25~28	29~32	33~36	12N			
...	25~28	29~32	33~36	37~40	41~44	45, 3N	12N

Figure 3: Padded wavefront mapping $\hat{W}$. Fill-in points are represented by N.

remains active to process large amount of grid points without exiting. This requirement is in direct conflict with the wavefront scheme. To resolve data dependency of two consecutive hyperplanes, all GPGPU workgroups must be synchronized at the boundary of a hyhperplane. In standard GPGPU programming, synchronization across multiple workgroups is achieved by launching another kernel for the subsequent hyperplane. To resolve this conflict, a method for inter-workgroup synchronization without launching new kernels must be employed.

GPGPU is not a flat collection of computation cores. Instead, a certain number of cores are physically organized in a streamline multiprocessor (SM). A GPGPU usually have a number of SMs. During kernel execution, each logical GPGPU workgroup is mapped to one SM and the mapping remains constant until that workgroup finishes. The execution of workgroups for each kernel launch is out-of-order, meaning that SMs work most independently on different workgroups. This hardware arrangement of two-level granularity directly corresponds to the workgroup arrangement in the programming model.

Atomic operations are globally sequential, which can provide a mechanism for creating a global barrier for inter-workgroup synchrnoization [24]. This is achieved by a global mutex variable to count the number of workgroups that reach the synchronization point. A CUDA C template of this approach is given in Algorithm 9, which is used by both the factorization and the forward solve. Backward solve use a slightly different version. Line 13 is replaced by the specific computation task. Assume that there are totally N_{sm} active SMs working on the problem. At the beginning of the loop, the head thread ($I_T = 0$) of each workgroup obtains a new batch, which contains L grid points, by adding L to the global index g_p_elm. Note that function atomicAdd() returns the value of the global mutex before it is added.

The wavefront ordering map is padded, so that N_{sm} workgroups of purely fill-ins are placed at each hyperplane boundary. For example, if SM=3, and each workgroup process 4 data points, then three hyperplanes with 28, 36, 45 points would be padded in the way shown in Figure 3. Note that each table cell corresponds to one workgroup batch. The mapping can then be flatten to a single long list $\hat{W}$ with only one index, facilitating the synchronization algorithm. N_{total} is the total number of grid points, including both "real" points and fill-ins. If the head thread detects a fill-in, it adds 1 to the global mutex g_mutex. When all SMs finish adding to the global mutex, the global mutex is increased by N_{sm}, so that the modulo operation should result in zero. An inter-workgroup synchronization is therefore achieved.

We also need to ensure that each SM has a one-to-one mapping to an active workgroup, so that, from a programmer's perspective, a workgroup is simply the logical equivalence of an SM. This is achieved by launching the kernel with exactly N_{sm} workgroups, and allocating all available local shared memory on an SM to each workgroup. The latter technique effectively prevents two workgroups from being scheduled to the same SM.

Algorithm 9 Lock-based wavefront kernel with thread recycling

```
1: ___device___ volatile int g_p_elm, g_mutex; // Both initialized to zero
2: ___device___ void wavefront_kernel(...)
3: {
4:    ___shared___ volatile int q_0;
5:    q_w ← I_T\n^2 ; q ← I_G L + q_w ; v ← I_T\n − n q_w ; u ← I_T − n^2 q_w − n v ;
6:    for () {
7:       if (I_T==0) q_0 ←atomicAdd((int*)&g_p_elm,L);
8:       ___syncthreads();
9:       if (q_0 > N_total) break ;
10:      Ŵ : (q_0 + q_w) → (i, j, k)
11:      if (not a fill-in) {
12:         ...do computation...
13:      } else if (I_T==0) {
14:         atomicAdd((int*)&g_mutex,1);
15:         while (g_mutex%N_sm!=0) {}
16:      }
17:   ___syncthreads();
18: }
```

4 Numerical experiments

The fine-grained algorithms have been implemented in Fortran and C with OpenACC and CUDA support. All floating-point operations are in double precision. For baseline comparison, a sequential BILU code is run on one of 16 cores of an Opteron 6128. A 16-thread OpenMP version is also included as a benchmark of coarse-grained algorithms. The fine-grained algorithms is tested on an Nvidia GTX Titan, which has 24 SMs and 12GB of global device memory. GTX Titan is a representative fourth-generation GPGPU, with a CUDA compute capability of 5.2. Due to improved atomic operations and warp scheduling, task-based parallelism becomes a much more attractive approach to write GPGPU programs.

4.1 Verification of FGBILU in INCOMP3D

FGBILU has been implemented as the block-sparse linear solver in INCOMP3D, which is an implicit 3D incompressible Navier-Stokes solver validated for a range of model problems [10, 11]. The 3D incompressible Navier-Stokes equations are solved on a multi-block structured grid using finite volume methods. 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 around moving objects, but is not employed in this study.

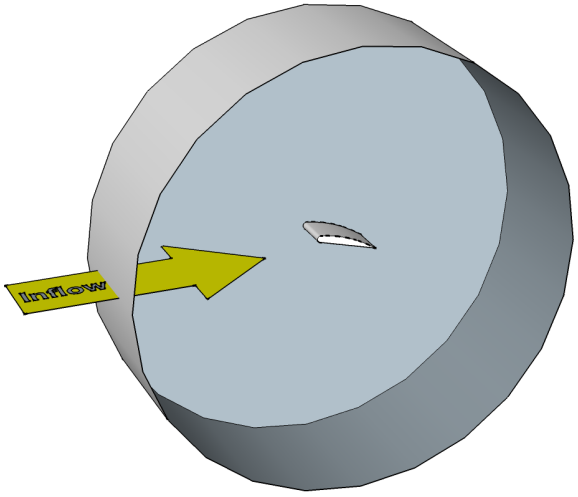


Figure 4: Illustration of the 3D domain for verification

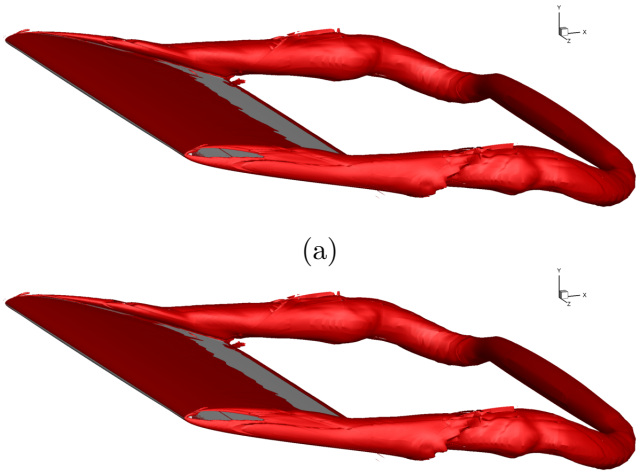
An LES test case based on a dynamic stall study [25] is used for the verification. An illustration with exaggerated dimensions is given in Figure 4. An SD7003 airfoil, with a chord length of 100mm and a span of 200mm, is placed in the center of the domain, which has a radius of around 1200mm. On the spanwise direction there is an additional 200mm of free flow space on each side of the airfoil, to allow enough room for finite-span effects. The Reynolds number based on the chord length is 10^6, with a free-stream velocity of 0.1m/s. The O-type mesh has 23 million cells, divided into 1152 blocks. Static load balancing is achieved by mapping multiple blocks of various sizes to a processor. 32 computation nodes are used, each of which executes two computation processes. Each of the nodes has one Xeon E5645 and two M2050 GPU. Only two of the six cores of each E5645 is used for the CPU version of the code, so that the impact of the interconnection is similar for both CPU and GPU simulations. A qualitative comparison is given in Figure 5, showing Q=4 isosurfaces for the same setting of the SD7003 case. As we can see, the results are virtually indistinguishable.

4.2 Scaling tests for different domain sizes

The primary concern of algorithms based on a wavefront scheme is synchronization overhead and performance penalty due to smaller wavefronts near domain corners. The total process time T varies greatly for different domain sizes. To evaluate an average process time for one grid point in the domain, a per-point process time is defined:

$$t_p = \frac{T}{IJK}\,.$$

Test domains are extracted from INCOMP3D solving a series of simple steady-state 3D RANS channel flows in cubic domains, ranging from 15^3 (3.4K) to 55^3 (85K). The number of PDE unknowns is fixed at $n = 6$. Once the linear system is extracted it is solved in the standalone solver. To minimize the impact of random time measure error, we repeat each task (factorization or triangular solve) 100 times and take an average.



(a)

(b)

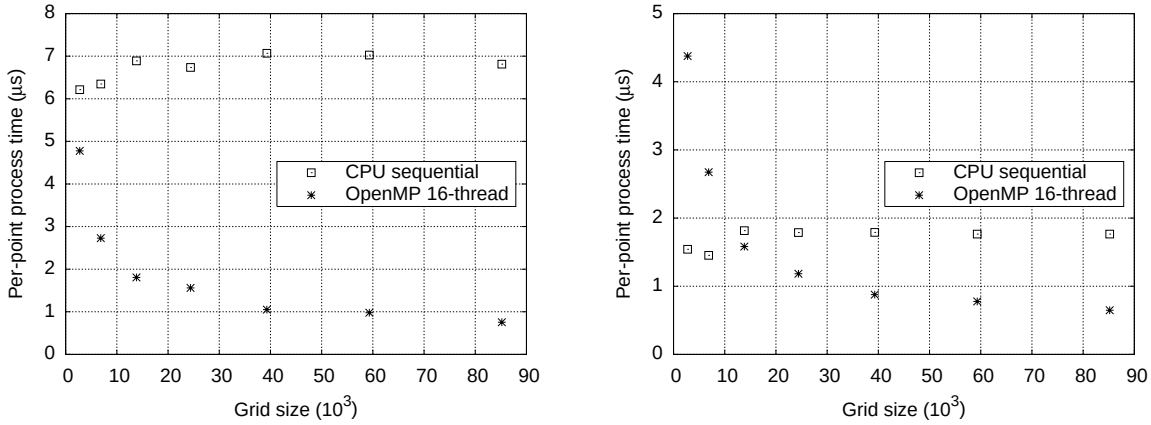
Figure 5: LES simulation results of the (a) unmodified CPU version of INCOMP3D, and (b) GPU version of INCOMP3D. $t = 1.125$s.

Run times on CPU are given in Fig. 6. The effects of insufficient work load caused by the wavefront scheme near the domain corners is clearly visible: sequential BILU shows almost constant t_p for various domain sizes, while t_p for wavefront algorithms gradually increases as the domain size decreases. The performance of the 16-thread OpenMP version is particularly poor for smaller domains, sometimes even slower than the sequential code running on one core, indicating heavy overhead incurred by a coarse-grained wavefront scheme. A coarse-grained wavefront-based implementation of BILU on a CPU remains inefficient for any domain with less than 25K grid points. Considering 25K is actually not a very small domain, such requirement is rather discouraging. Indeed, for multi-block implicit PDE solvers, shared-memory parallelism (OpenMP) is less preferred than message-passing parallelism (MPI) when ILU-type preconditioners are used.

Run times on GPU are given in Fig. 7. For larger domains, the performance by all three versions are mostly the same. This is very much expected, because t_p is largely determined by floating point computation, which is the same for all three versions. It is very important to investigate the performance for smaller domains, as they are often encountered in multi-block PDE solvers.

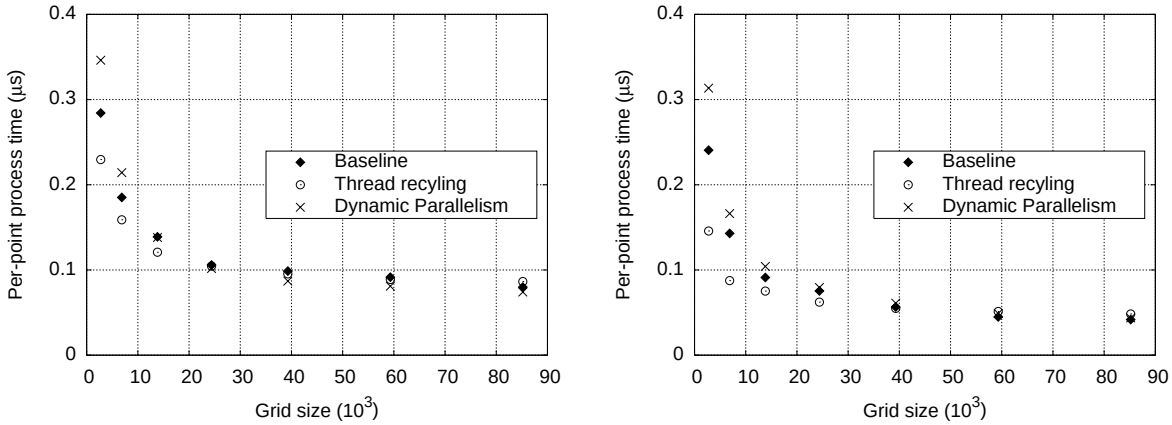
From Fig. 7 we can see that TR version constantly performances better than the baseline version for smaller domains. This indicates a clear benefit of TR, which is reducing index reconstruction overhead. In fact, only the grid point indices (i, j, k) need to be updated each time a new grid point is fetched. The relative element location (u, v) is always constant for each thread.

We can also see that the DP version is slower than the baseline for smaller domains. The reason for DP to under-perform is straightforward. First of all, the overhead associated with kernel launches in the baseline version is not very significant. Note that kernel launches are asynchronized, meaning that the kernel launch overhead often overlaps with kernels that is still running. Because the computation load of all three FGBILU kernels is substantial, almost all of the kernel launch overhead can be masked by this asynchronized launching



(a) Factorization (b) Triangular solve

Figure 6: t_p as a function of grid size for CPU algorithms



(a) Factorization (b) Triangular solve

Figure 7: t_p as a function of grid size for GPU algorithms

mechanism. Therefore, the actual impact of kernel launch overhead in the baseline version is minimal. On the other hand, DP is merely a mechanism to launch kernels from GPGPU. As DP requires an master thread to launch wavefront kernels, allowing slightly less computation resource for actual computation. For smaller domains, such loss of computation resource clearly outweights any benefits of CPU-free kernel launching, resulting in less efficient overall performance in the DP version.

Apparently, a coarse-grained wavefront-based implementation of BILU, such as the OpenMP version on a CPU, is rather inefficient if many small domains are to be processed. On the other hand, wavefront scheme shows clear advantage when fine-grained algorithms are adopted on GPGPU. Among the three versions of FGBILU, the TR version provides an effective optimization for smaller domains, while the DP version shows no clear benefit for all domain sizes. Speedup numbers, calculated by comparing run times of FGBILU and 16-thread OpenMP version, are compiled in Table 1. The DP version is omitted altogether, as it is unlikely to be adopted in an optimized version of FGBILU. Speedup numbers show the biggest difference

	Factorization		Triangular Solve	
Size of grid (10^3)	Baseline	Thread Recycling	Baseline	Thread Recycling
2.7	17	20	18	30
6.9	17	21	19	31
13.8	15	17	17	21
24.4	13	15	16	19

Table 1: List of speedup numbers against 16-thread OpenMP version of BILU

for triangular solve. This is not surprising, because it has less floating point operations, which amplifies any improvement on overhead reduction. As the domain size grows, the speedup numbers quickly converge. There is very little difference between all three FGBILU versions beyond 30K.

5 Conclusion

The baseline FGBILU has been designed with two granularity levels, which are perfectly mapped to the two-tier parallelism model of CUDA. The first tier resolves data dependency among grid points, which is mapped to CUDA workgroups. The second tier resolves data dependency among elements of submatrices, which is mapped to threads within a workgroup. An efficient implementation of the second tier parallelism requires very low-overhead synchronization mechanism, which is readily provided by CUDA thread-level synchronization.

The baseline CUDA implementation of FGBILU performs very well for larger domains. However, the nature of wavefront scheme imposes inevitable performance loss near domains corners. This issues is particularly serious for smaller domains. As less computation work can be generated in smaller domains, the impact of overhead is magnified. The primary objective of improving performance of FGBILU on smaller domains is therefore determined by how the impact of overhead can be mitigated.

In this study, two CUDA-specific techniques are investigated. The first technique, CUDA dynamic parallelism, allows GPGPU to launch kernels without CPU involvement. However, the benefit of this technique is marginal at best, because the kernel launching overhead has been masked very well by the asynchronized launch mechanism. The extra involvement of a master control thread on GPGPU reduces the effective computation resource for floating point computation, causing overall performance loss. The second technique is called thread recycling, which allows each SM on GPGPU to remain mapped to one workgroup so that it continues to fetch grid points to process. A global lock based on atomic operations is adopted to achieve synchronization at the boundary of each hyperplanes. Because threads remain active all the time, a large portion of the index reconstruction can remain constant, avoiding repetitive calculation and improving overall performance on smaller domains.

Further improvement of FGBILU are being investigated. Currently, the wavefront scheme of FGBILU only applies to structured domains. With proper level scheduling techniques FGBILU can be modified to work with unstructured grids. Also, it may be possible to adopt FGBILU using vector processing operations in the latest CPUs. It will be interesting to see if manually optimized algorithms on CPU can lead to similar performance improvement.

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 Virginia Tech's Hokiespeed computing cluster.

References

[1] S. Georgescu, P. Chow, H. Okuda, GPU acceleration for FEM-based structural analysis, Archives of Computational Methods in Engineering 20 (2) (2013) 111–121.

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

[3] D. A. Jacobsen, J. C. Thibault, 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.

[4] T. Brandvik, G. Pullan, Acceleration of a 3D Euler solver using commodity graphics hardware, in: 46th AIAA aerospace sciences meeting and exhibit, 2008, p. 607.

[5] A. Corrigan, F. Camelli, R. Löhner, F. Mut, Semi-automatic porting of a large-scale Fortran CFD code to GPUs, International Journal for Numerical Methods in Fluids 69 (2) (2012) 314–331.

[6] The OpenACC Application Programming Interface (2013).

[7] L. Luo, J. R. Edwards, H. Luo, F. Mueller, Advanced optimizations of an implicit Navier-Stokes solver on GPGPU, in: AIAA Aviation, 2014.

[8] MVAPICH2: High performance MPI over InfiniBand, 10GigE/iWARP and RoCE.
URL `http://mvapich.cse.ohio-state.edu/`

[9] Open MPI: Open source high performance computing.
URL `http://www.open-mpi.org/`

[10] J. R. Edwards, M.-S. Liou, Low-diffusion flux-splitting methods for flows at all speeds, AIAA Journal 36 (9) (1998) 1610–1617.

[11] J.-I. Choi, R. C. Oberoi, J. R. Edwards, J. A. Rosati, An immersed boundary method for complex incompressible flows, Journal of Computational Physics 224 (2) (2007) 757–784.

[12] A. J. Chorin, Numerical solution of the Navier-Stokes equations, Mathematics of Computation 22 (104) (1968) 745–762.

[13] K. Ramesh, A. Gopalarathnam, J. R. Edwards, M. V. Ol, K. Granlund, An unsteady airfoil theory applied to pitching motions validated against experiment and computation, Theoretical and Computational Fluid Dynamics 27 (6) (2013) 843–864.

[14] G. Z. McGowan, K. Granlund, M. V. Ol, A. Gopalarathnam, J. R. Edwards, Investigations of lift-based pitch-plunge equivalence for airfoils at low reynolds numbers, AIAA journal 49 (7) (2011) 1511–1524.

[15] D. A. Cassidy, J. R. Edwards, M. Tian, An investigation of interface-sharpening schemes for multi-phase mixture flows, Journal of Computational Physics 228 (16) (2009) 5628–5649.

[16] J.-I. Choi, J. R. Edwards, Large eddy simulation and zonal modeling of human-induced contaminant transport, Indoor air 18 (3) (2008) 233–249.

[17] J.-I. Choi, J. R. Edwards, Large-eddy simulation of human-induced contaminant transport in room compartments, Indoor Air 22 (1) (2012) 77–87.

[18] R. S. Varga, Matrix iterative analysis, Vol. 27, Springer Science & Business, 2009.

[19] O. Axelsson, S. Brinkkemper, V. Iln, On some versions of incomplete block-matrix factorization iterative methods, Linear Algebra and its Applications 58 (0) (1984) 3–15.

[20] G. Wittum, On the robustness of ILU smoothing, SIAM Journal on Scientific and Statistical Computing 10 (4) (1989) 699–717.

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

[22] L. Luo, J. R. Edwards, H. Luo, F. Mueller, GPU port of a parallel incompressible Navier-Stokestokes solver based on OpenACC and MVAPICH2, in: AIAA Aviation and Aeronautics Forum and Exposition, 2014.

[23] G. Meurant, The block preconditioned conjugate gradient method on vector computers, BIT Numerical Mathematics 24 (4) (1984) 623–633.

[24] S. Xiao, W. Feng, Inter-block GPU communication via fast barrier synchronization, in: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, IEEE, 2010, pp. 1–12.

[25] S. Narsipur, A. Gopalarathnam, J. R. Edwards, A time-lag approach for prediction of trailing edge separation in unsteady flow, in: AIAA Aviation and Aeronautics Forum and Exposition, AIAA, 2014.