A fine-grained block ILU scheme on regular structures for GPGPUs

Lixiang Luo[a,*], Jack R. Edwards[a], Hong Luo[a], Frank Mueller[b]

[a]*Department of Mechanical and Aerospace Engineering*
North Carolina State University
[b]*Department of Computer Science*
North Carolina State University

Abstract

Iterative methods based on block incomplete LU (BILU) factorization are considered highly effective for solving large-scale block-sparse linear systems resulting from coupled PDE systems with n equations. However, efforts on porting implicit PDE solvers to massively parallel shared-memory heterogeneous architectures, such as general-purpose graphics processing units (GPGPUs), have largely avoided BILU, leaving their enormous performance potential unfulfilled in many applications where the use of implicit schemes and BILU-type preconditioners/solvers is highly preferred. Indeed, strong inherent data dependency and high memory bandwidth demanded by block matrix operations render naive adoptions of existing sequential BILU algorithms extremely inefficient on GPGPUs. In this study, we present a fine-grained BILU (FGBILU) scheme which is particularly effective on GPGPUs. A straightforward one-sweep wavefront ordering is employed to resolve data dependency. Granularity is substantially refined as block matrix operations are carried out in a true element-wise approach. Particularly, the inversion of diagonal blocks, a well-known bottleneck, is accomplished by a parallel in-place Gauss-Jordan elimination. As a result, FGBILU is able to offer low-overhead concurrent computation at $O\left(n^2N^2\right)$ scale on a 3D PDE domain with a linear scale of N. FGBILU has been implemented with both OpenACC and CUDA and tested as a block-sparse linear solver on a structured 3D grid. While FGBILU remains mathematically identical to sequential global BILU, numerical experiments confirm its exceptional performance on an Nvidia GPGPU.

Keywords: block ILU; block-sparse linear systems; wavefront scheme; GPGPU; OpenACC; CUDA

1. Introduction

Block incomplete LU (BILU) factorization has been considered as one of the best preconditioning techniques for large-scale fully implicit simulations. Its variants are particularly popular for simulations of highly stiff physics systems [1]. One example is multi-phase flows, in which different behaviors of fluids often lead to poorly-conditioned linear systems. Multi-component reactive flows are notorious for being extremely stiff due to vastly different time scales of fluid dynamics and chemical reactions. Stiffness can also be a result of inherent nonlinearity of the Navier-Stokes equations, such as shock and boundary layer interactions. The selection of of linear solver can be rather limited in these situations. While a direct solver is most desirable, it is prohibitively expensive for 3D

*corresponding author.
Email address: lixiang_luo@ncsu.edu (Lixiang Luo)

problems. As a compromise, iterative methods based on BILU-type schemes are often employed. The robustness and efficiency of BILU are largely contributed by its close resemblance of Gaussian elimination. Although a global BILU with natural ordering is very useful in sequential codes, its inherent data dependency poses a daunting challenge for parallelization. One common practice to extract parallelism is level scheduling, such as wavefront ordering, but its benefit is limited by synchronization overhead and still insufficient parallelism. Other techniques such as coloring or graph partitioning allow efficient coarse-grained to mid-grained parallelism with little overhead (see, for example [2, 3, 4, 5, 6]), at the expense of slightly reduced mathematical efficiency. More detailed reviews can be found in [7, 8, 9]. PDE solvers on large domains often employ message passing parallelism. The original computational domain is partitioned into smaller subdomains, each mapped to a single processor core running a sequential BILU, avoiding further parallelism and corresponding overhead. Moreover, in a multi-block solver, the original computational domain is already partitioned into subdomains, so a message passing solution is a natural choice.

Recently, general-purpose graphics processing units (GPGPU) have attracted much attention as a promising technology for large-scale high performance computation (see, for example, [10, 11, 12, 13, 14, 15, 16]). A GPU, which is essentially a shared memory vector machine, has a potential to achieve one or two orders of magnitude of performance improvement for highly parallel algorithms. However, adoption of implicit PDE solvers on GPU remains a challenging subject. It is generally very difficult for an iterative linear solver on a GPU to utilize BILU-type preconditioners, which are recursive in nature. While the general difficulty of extracting parallelism from BILU remains, the massively parallel nature of a GPU renders usual coarse-grained strategies highly ineffective [17], adding to a perception that BILU is not suitable for a GPU implementation.

Fine granularity is the key for BILU to achieve high efficiency on massively parallel architectures. A typical GPU has hundreds of computation cores, each of which has much less computation power and local cache memory than a typical CPU core. Therefore, the benefit of fine granularity is twofold. 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 data each core processes is kept as small as possible. The latter aspect is particularly important for solving a block-sparse system, which consists of a large amount of square submatrices of an identical size. If the operation on one submatrix is mapped to one GPU core, the algorithm easily becomes heavily memory-bounded, resulting in serious performance penalty.

For BILU algorithms, fine granularity can be accomplished by adopting a two-tier approach. At the first tier, level scheduling by wavefront ordering is used to resolve the inherent data dependency of BILU [18]. It allows the parallel processing of a series of subsets, each consisting of data-independent elements, while maintaining mathematical equivalence to the sequential BILU. The number of grid points on each subset is relatively small. To achieve finer granularity more parallelism must be extracted beyond the wavefront scheme. At the second tier, the matrix operations for block submatrices are parallelized at the matrix element level. While the fine-grained algorithms for matrix-matrix and matrix-vector multiplications are rather straightforward, a fine-grained algorithm for matrix inversion is particularly elusive. This is achieved by adopting an in-place Gauss-Jordan elimination with an element-wise thread mapping.

This paper presents a global fine-grained BILU(0) scheme (FGBILU) that can be readily adopted to a wide variety of massively parallel hardware and programming frameworks. While new technologies are constantly being developed, it is often preferable that algorithms are designed to be compatible with open standards that can adapt to ever-changing hardware. With such considerations in mind,

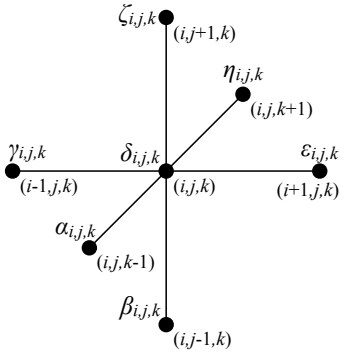


Figure 1: 7-point stencil on a regular 3D structure

algorithms will be presented in two prevailing programming styles for massively parallel architectures. The "ACC style" targets the directive-based programming frameworks, such as OpenACC [19], OpenHMPP [20] and OpenMP for Accelerators [21]. The pseudocode notions are self-explanatory, as the only "directive" is "*kernel*", specifying regions to be executed in parallel, along with necessary thread spawning information. Directive-based programming standards allow the same source code to be compiled into binaries intended for different architectures, greatly improving maintainability and portability. The second style, we call "CL style", targets the more low-level programming frameworks, such as CUDA and OpenCL. Additional programming techniques on thread-level synchronization and local shared memory arrays allow more flexibility in algorithm design and offer better performance potential.

This paper is organized as follows. First, a brief introduction of BILU is given in Section 2, along with a description of a traditional sequential algorithm. This section also includes a coarse-grained BILU scheme based on wavefront ordering, which serves as a starting point of the fine-grained algorithms. In Section 3, the two styles of FGBILU algorithms are discussed in detail. The scheme includes algorithms for the factorization, triangular solve and iterative correction processes. 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 structure

In this section, we revisit BILU(0), the incomplete LU factorization with zero fill-ins for block-sparse linear system, on a regular 3D IJK structure. Since the discussion of other ILU variants is beyond the scope of this study, the "zero fill-ins" notation is omitted from now on. ILU was originally developed as approximate inversion schemes by matrix splitting for sparse matrices [22]. The block-sparse extension was given much attention when iterative solvers started being employed in more complex physical problems [23, 24].

Consider the discretization of a coupled PDE system with n unknowns on a structured 3D domain with a straightforward 7-point stencil, as shown in Fig. 1. $1 \leq i \leq I$, $1 \leq j \leq J$ and $1 \leq k \leq K$ are grid cell indexes and the corresponding dimensions of the domain. Note that all indices, including those in pseudocodes, are one-based. Implicit time integration with a second-order spatial discretization

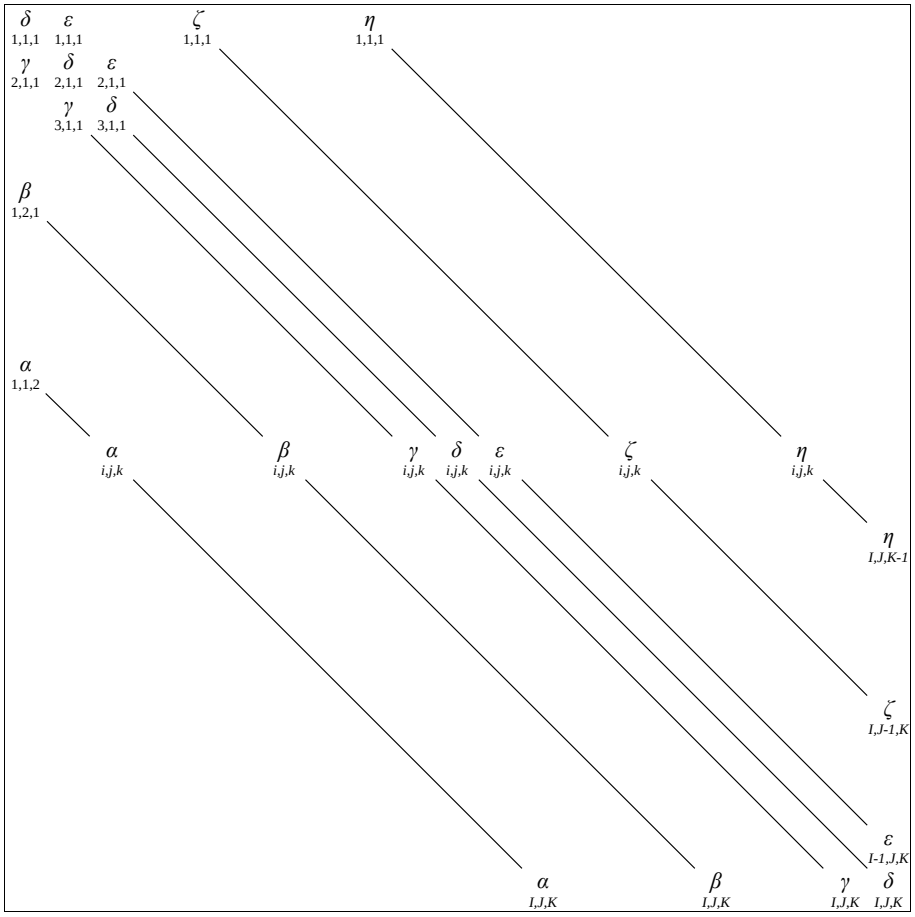


Figure 2: Coefficient matrix A

scheme on such a stencil 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 Fig. 2. Each "element" therein is an $n \times n$ 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 also reflected in Fig. 1. Grid location references in algorithms for a general grid point are always limited to the 7 points in the stencil.

The BILU factorization of A, represented by the multiplication of a lower block-triangle matrix L and an upper block-triangle matrix U, is given in Fig. 3. BILU with zero fill-in attempts to approximate A with LU so the residual matrix $(LU - A)$ takes the exact same zero pattern of A. The submatrices in L and U associate with the stencil in a straightforward fashion, as shown in Fig. 4.

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:

$$\begin{aligned}
a_{i,j,k} d_{i,j,k-1} &= \alpha_{i,j,k}\,, \\
b_{i,j,k} d_{i,j-1,k} &= \beta_{i,j,k}\,, \\
c_{i,j,k} d_{i-1,j,k} &= \gamma_{i,j,k}\,, \\
a_{i,j,k} g_{i,j,k-1} + b_{i,j,k} f_{i,j-1,k} + c_{i,j,k} e_{i-1,j,k} + d_{i,j,k} &= \delta_{i,j,k}\,, \\
e_{i,j,k} &= \varepsilon_{i,j,k}\,, \\
f_{i,j,k} &= \zeta_{i,j,k}\,, \\
g_{i,j,k} &= \eta_{i,j,k}\,.
\end{aligned}$$

Simple manipulations reveal that the factorization only needs to be carried out for the diagonal blocks $\delta_{i,j,k}$, using 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:

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

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:

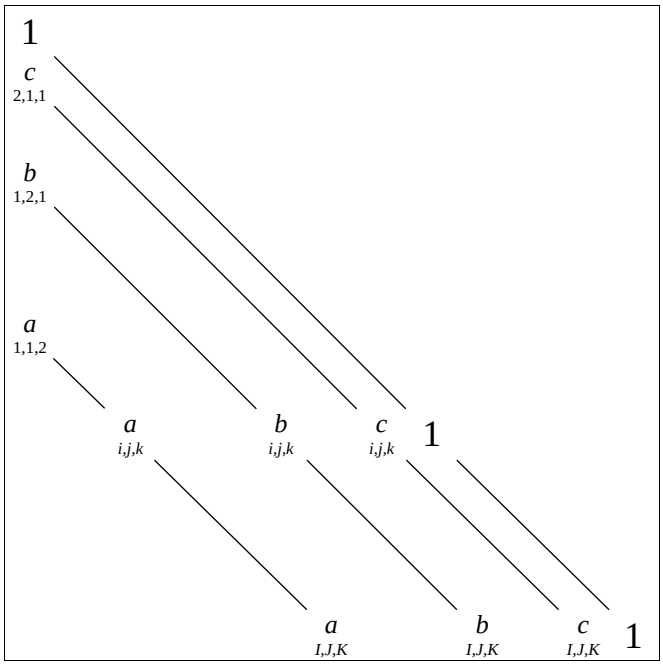
$$M\hat{\mathbf{x}} = \mathbf{b}\,, \tag{2}$$

which can be now be solved by two triangular sweeps. The forward sweep solves an intermediate vector $\mathbf{y}$ using the following recurrence:

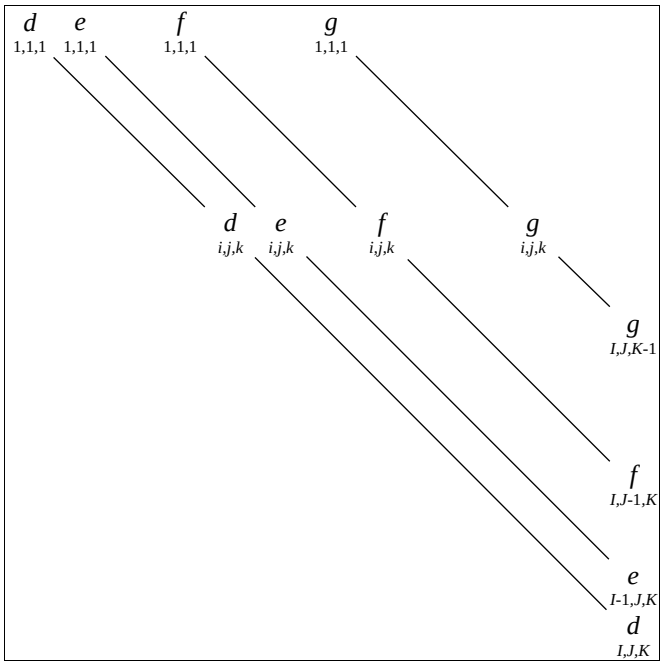
$$y_{i,j,k} = d^{-1}_{i,j,k} \left(b_{i,j,k} - \alpha_{i,j,k} y_{i,j,k-1} - \beta_{i,j,k} y_{i,j-1,k} - \gamma_{i,j,k} y_{i-1,j,k} \right)\,.$$

The backward sweep solves $\hat{x}$ from y in a similar fashion:

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



(a)



(b)

Figure 3: (a) L and (b) U factors of the BILU factorization

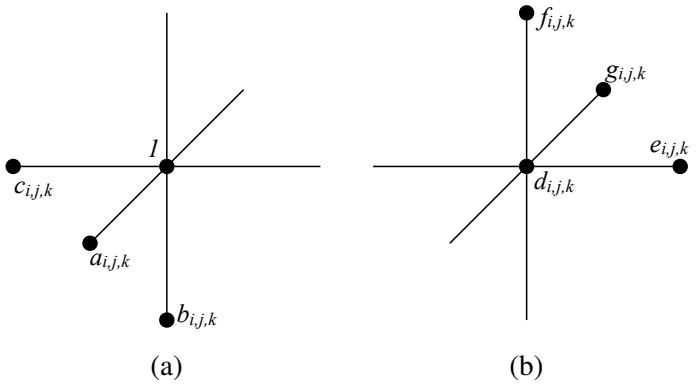


Figure 4: (a) L and (b) U factors of the BILU factorization for the 7-point stencil in Figure 2.

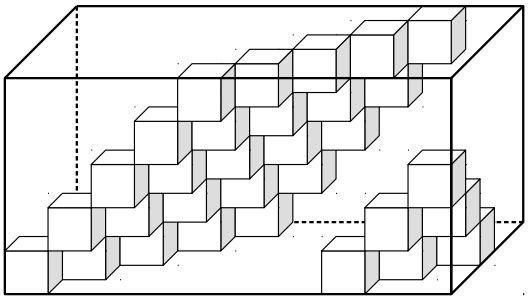


Figure 5: Two hyperplanes in a $10 \times 5 \times 5$ block.

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

$$M\xi_l = \mathbf{b} - A\hat{\mathbf{x}}_l\,, \quad \hat{\mathbf{x}}_{l+1} = \hat{\mathbf{x}}_l + \xi_l, \tag{3}$$

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.

2.1. Parallel BILU by wavefront ordering

We can see that neither the BILU factorization (Eq. 1) nor the triangular solve (Eq. 2) is readily parallelizable, as they are both in recursive forms. In particular, the factorization has a data dependency of (i,j,k) on $(i-1,j,k)$, $(i,j-1,k)$ and $(i,j,k-1)$. The triangular solves have similar data dependencies. On the other hand, all grid points with a constant sum of subscripts $i+j+k$ are data-independent, thus allowing parallel processing. In fact, this is one of the simplest form of leveling scheduling, that is, identifying a sequence of data-independent subsets ("levels"). In our case, we can use the sum of the subscripts as the level index p:

$$p = i + j + k\,.$$

The number of points on hyperplane p is then denoted as N_p.

In a 3D IJK block, each level resembles a diagonal cross-section, called a hyperplane. The sequence of hyperplanes propagates in the IJK block like a wavefront, thus giving the name of "wavefront scheme". Fig. 5 shows an example of a $10 \times 5 \times 5$ block. Each cube therein represents one grid

Algorithm 1 Generation of wavefront ordering mapping

1: **do** $i = 1, I$; **do** $j = 1, J$; **do** $k = 1, K$
2: $p \leftarrow i + j + k$
3: $N_p \leftarrow N_p + 1$
4: $\mathrm{W}(1, N_p, p) \leftarrow i$; $\mathrm{W}(2, N_p, p) \leftarrow j$; $\mathrm{W}(3, N_p, p) \leftarrow k$
5: **end do** ; **end do** ; **end do**

point. Assuming grid location of (1,1,1) is at the lower-right corner of the bottom plane, it can be found that $N_5 = 6$ for the smaller hyperplane, and $N_{12} = 25$ for the larger hyperplane.

Instead of the natural IJK ordering, the grid points are processed in a wavefront ordering. Each grid point is associated an index pair (p,q), which indicates the hyperplane it belongs to and its position in the hyperplane. For a static IJK grid, a mapping $W : (p,q) \rightarrow (i,j,k)$ and a list of hyperplane sizes N_p can be readily generated *a priori* with Algorithm 1.

At this point factorization and triangular solve can be parallelized inside each hyperplane, as given in Algorithm 2 and 3. The outer loop is sequential over the hyperplanes, which ensures data dependency. A parallelized inner loop is designated as a "kernel", with index variables and their ranges listed inside a pair of curly brackets. One permutation of indices is the smallest unit for concurrent execution in a parallel implementation. As a convention, the mapping $W : (p,q) \rightarrow (i,j,k)$ is implied whenever wavefront scheme is involved. In a sequential or coarse-grained implementation, the factorized diagonals $d_{i,j,k}$ are often stored in-place as its LU decomposition. This allows a matrix multiplication with both $d_{i,j,k}$ and $d^{-1}_{i,j,k}$ to be calculated as two triangular sweeps, without using extra memory space. Note that, in pseudocodes, $d_{i,j,k}$ and $\hat{x}_{i,j,k}$ represents the computer variable instead of its original mathematical meaning. Again, any out-of-bound references should be treated as zero.

Algorithm 2 Coarse-grained parallel BILU factorization

1: **do** $p = 3$, *I+J+K*
2: ***kernel*** { $q = 1 \ldots N_p$ }
3: $d_{i,j,k} \leftarrow d_{i,j,k} - \alpha_{i,j,k} d_{i,j,k-1} \eta_{i,j,k} - \beta_{i,j,k} d_{i,j-1,k} \zeta_{i,j,k} - \gamma_{i,j,k} d_{i-1,j,k} \varepsilon_{i,j,k}$
4: $d_{i,j,k} \leftarrow \left(d_{i,j,k}\right)^{-1}$
5: ***end kernel***
6: **end do**

One can observe that the coarse-grained algorithms can be readily implemented with OpenMP on CPU cores. However, such a straightforward wavefront scheme has been considered insufficient to provide much potential for parallelization [18]. First of all, a coarse-grained wavefront scheme suffers from insufficient load when processing smaller hyperplanes near $(1,1,1)$ and (I,J,K) corners. If the parallel paradigm introduces significant overhead during synchronization (OpenMP), the performance of a parallel algorithm can be rather poor. On the other hand, production-level PDE solvers are often parallelized over a high-performance cluster with message passing parallelism. To achieve finer granularity one can simply partition the overall domain into smaller subdomains. Unless message passing parallelism is absent, coarse-grained shared-memory parallelism such as OpenMP is often uneconomical for improving the overall performance of a BILU-based PDE solver.

Algorithm 3 Coarse-grained parallel BILU triangular solve

Forward sweep

do $p = 3, I+J+K$
  kernel { $q = 1 \ldots N_p$ }
    $\hat{x}_{i,j,k} \leftarrow d_{i,j,k}\left(b_{i,j,k} - \alpha_{i,j,k}\hat{x}_{i,j,k-1} - \beta_{i,j,k}\hat{x}_{i,j-1,k} - \gamma_{i,j,k}\hat{x}_{i-1,j,k}\right)$
  end kernel
end do

Backward sweep

do $p = I+J+K, 3, -1$
  kernel { $q = 1 \ldots N_p$ }
    $\hat{x}_{i,j,k} \leftarrow \hat{x}_{i,j,k} - d_{i,j,k}\left(\varepsilon_{i,j,k}\hat{x}_{i+1,j,k} + \zeta_{i,j,k}\hat{x}_{i,j+1,k} + \eta_{i,j,k}\hat{x}_{i,j,k+1}\right)$
  end kernel
end do

3. Fine-grained algorithms on massively parallel shared-memory architectures

3.1. Introduction to GPGPU computation concepts

Recent development of massively parallel architectures, such as GPGPUs, poses new challenges in high-performance code design. Their heterogeneous nature generally does not provide full capabilities of a traditional CPU core. Current MPI codes designed to run on a CPU core cannot run on a GPU core, forcing programmers to adopt to a GPU programming model.

A GPU has hundreds of computation cores. Each core is capable of executing one instance of the kernel code, which is called a "thread". GPU threads are logically organized as "workgroups". Within each workgroup, local shared memory space can be used for data sharing among threads of the workgroup. The size of a workgroup and the total number of workgroups can both be programmed. The actual execution of threads is out-of-order, even for threads in the same logical workgroup. GPU threads are not executed independently, either. In fact, a certain number (fixed by hardware design) of threads, called a "warp", are executed in synchronized lock-step. One important consequence of warp execution is the significant performance penalty by warp divergence (different threads within a warp need to run different code branches), so conditionals should be avoided if possible.

Our experience shows that a simple port of the coarse-grained BILU on a GPU does not perform very well [17]. The primary performance constraint is found to be memory boundedness. If one grid point is mapped to one GPU thread, the coarse-grain factorization would need to access 7 $n \times n$ submatrices per GPU thread. Assuming that n, the number of PDE equations, is 6 (for 3D RANS using a two-equation model) and that most computations can be carried out in-place, each GPU core would need to access at least $7 \times 6^2 \times 8 = 2016$ bytes of data (each double precision variable uses 8 bytes of memory). Take M2050, a typical second-generation Nvidia GPU used for our numerical experiments, for example. Its 448 cores share a total of 512KB L2 cache. The coarse-grained BILU factorization algorithm on this GPU would need to access $2016 \times 448 = 903168$ bytes of data simultaneously. This number is far more than the size of the L2 cache memory, leading to very poor data locality. Furthermore, the coarse-grained algorithm quickly becomes intractable as the number of PDE unknowns increases, because the amount of data scales with n^2. The high memory bandwidth demand can easily overwhelm the cache subsystem of even the most powerful GPU.

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, $\alpha^{u,v}_{i,j-1,k}$ is simply written as $\alpha^{u,v}_{j-1}$, and $\beta^{u,v}$ actually means $\beta^{u,v}_{i,j,k}$. In pseudocodes, submatrix and vector variables are easily distinguished from scalar variables, which do not have superscripts.

3.2. ACC-style algorithms

The directive-based GPGPU programming approach closely resembles the shared-memory parallelism for CPUs. Since all parallel sections appear as simple loops, such approach allows the codes to be written with high maintainability and portability. Indeed, this approach proved invaluable for our development effort on FGBILU. On the other hand, the ACC style is rather restricted in synchronization options, due to the underlying hardware limitations of GPGPUs. In ACC style, the only formal synchronization method is to end a kernel altogether. As a result, an ACC-style kernel does not allow any inter-thread data dependency, and any synchronization barrier must be translated into a separate kernel launch.

3.2.1. Factorization

Algorithm 4 ACC version of FGBILU factorization

Temporary storage: $s^{n,n}_{3,N_p}$. A second subscript, q, is implied in all s references.

kernel #1 $\{ q = 1 \dots N_p, u = 1 \dots n, v = 1 \dots n \}$
  $s^{u,v}_1 \leftarrow \langle \alpha^{u,:} , d^{:,v}_{k-1} \rangle$; $s^{u,v}_2 \leftarrow \langle \beta^{u,:} , d^{:,v}_{j-1} \rangle$; $s^{u,v}_3 \leftarrow \langle \gamma^{u,:} , d^{:,v}_{i-1} \rangle$
end kernel
kernel #2 $\{ q = 1 \dots N_p, u = 1 \dots n, v = 1 \dots n \}$
  $d^{u,v} \leftarrow d^{u,v} - \langle s^{u,:}_1 , \eta^{:,v}_{k-1} \rangle - \langle s^{u,:}_2 , \zeta^{:,v}_{j-1} \rangle - \langle s^{u,:}_3 , \varepsilon^{:,v}_{i-1} \rangle$
end kernel
do $r = 1, n$
  kernel #3 $\{ q = 1 \dots N_p, \hat{w} = 1 \dots 2(n-1) \}$
    $a \leftarrow (\hat{w} \geq n ? 1 : 0)$; $w \leftarrow \hat{w} - a\,(n-1)$; $w \leftarrow w + (w \geq r ? 1 : 0)$
    $b \leftarrow a\,(w - r)$; $u \leftarrow r + b$; $v \leftarrow w - b$
    $d^{u,v} \leftarrow (1 - 2a)\, d^{u,v} / d^{r,r}$
  end kernel
  kernel #4 $\{ q = 1 \dots N_p, \hat{u} = 1 \dots (n-1), \hat{v} = 1 \dots (n-1) \}$
    $u \leftarrow \hat{u} + (\hat{u} \geq r ? 1 : 0)$; $v \leftarrow \hat{v} + (\hat{v} \geq r ? 1 : 0)$
    $d^{u,v} \leftarrow d^{u,v} + d^{u,r}\, d^{r,v}\, d^{r,r}$
  end kernel
  kernel #5 $\{ q = 1 \dots N_p \}$
    $d^{r,r} \leftarrow 1.0 / d^{r,r}$
  end kernel
end do

ACC version of the fine-grained factorization algorithm for a hyperplane with index number p is given in Algorithm 4. The temporary array s, allocated in the global GPU memory, is used for storing intermediate results during the matrix multiplications. Its first subscript ranges from 1 to 3, corresponding to the three spatial components. The second subscript q gives the point index within a hyperplane, which is always implied in pseudocodes. The two superscripts are submatrix indices,

each ranging from 1 to n. A dot product is written as $\langle \cdot, \cdot \rangle$. Fortran-style colon operators are used to indicate the range of a certain dimension.

The BILU factorization has two primary steps. The first step of BILU factorization is the calculation of $\alpha d_{k-1}\eta$, $\beta d_{j-1}\zeta$, $\gamma d_{i-1}\varepsilon$ and their sum. Computation of each element in the result matrix is mapped to one concurrent thread. Because these terms appear in the form of $M_1M_2M_3$, a synchronization point must be placed between the two consecutive multiplications.

The second step of the BILU factorization is the inversion of diagonal submatrices. $d_{i,j,k}$ is a relatively small but dense matrix, and all inversion algorithms invariably involve recursive processes. This step is a well-known bottleneck of parallel BILU algorithms. For parallel solutions such as OpenMP, which has a moderate overhead, the benefit of extracting parallelism from the inversion algorithm is often negated by heavy synchronization cost. Low-overhead solutions, such as early vector machines and the later SIMD instructions available in CPUs, are often restricted by poor support of branching instructions and lack of local storage [25]. A GPU, on the other hand, combines low-overhead parallelism with a high degree of programming flexibility, allowing it to carry out a fine-grained BILU factorization efficiently.

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

1: **do** $r = 1, n$
2: $\quad \forall u = 1 \ldots n, u \neq r : d^{u,r} \leftarrow -d^{u,r}/d^{r,r}, d^{r,u} \leftarrow d^{r,u}/d^{r,r}$
3: $\quad \forall u = 1 \ldots n, v = 1 \ldots n, u \neq r, v \neq r : d^{u,v} \leftarrow d^{u,v} + d^{u,r}d^{r,v}d^{r,r}$
4: $\quad d^{r,r} \leftarrow 1/d^{r,r}$
5: **end do**

To achieve fine granularity in FGBILU factorization, an in-place direct inversion algorithm based on the Gauss-Jordan elimination (GJE) is employed, whose general sequential pseudocode is given in Algorithm 5. This method is highly compact and does not require any extra storage, making it particularly efficient on GPUs. The outer r-loop corresponds to the index of the row being transformed in GJE. Pivoting is generally not required, but it can be included if necessary. Although GJE is not the fastest method to calculate inversions, it is rather parallelizable. Also, it can be performed in-place, avoiding any thread-private arrays. Both factors are highly beneficial for GPU implementations.

Due to data dependencies, inversion must be written as three kernels (#3~5), which are simply parallel versions of Line 2, 3 and 4 in Algorithm 5, respectively. These three kernels have different mappings between GPU threads and loop iterations, mapping $2(n-1)$, $(n-1)^2$ and 1 thread per grid point, respectively. Conditionals have been translated into index manipulations, avoiding idle threads for skipped iterations. Because the r-loop runs on a CPU, there are $3n$ kernel launches to complete the inversion for one hyperplane.

3.2.2. Triangular solves

While the factorization requires rigorous algorithm design effort, the triangular solves are rather straightforward, as they involve mostly matrix-vector multiplications. The ACC-style algorithms for a fine-grained triangular solves are given in Algorithm 6. Forward and backward sweeps are very similar in structure. Each element in the solution vector is mapped to one GPU thread. Due to data dependencies, the computation on each hyperplane must be written as two kernels. Similar to the factorization algorithm, an temporary storage space z is used to store intermediate results.

Algorithm 6 ACC version of triangular solve

Temporary storage: $z^n_{N_p}$. A subscript, q, is implied in all z references.

(a) Forward sweep

1: ***kernel*** $\{q = 1 \ldots N_p, u = 1 \ldots n\}$
2: $\quad z^u \leftarrow b^u - \langle \alpha^{u,:}, \hat{x}^:_{k-1} \rangle - \langle \beta^{u,:}, \hat{x}^:_{j-1} \rangle - \langle \gamma^{u,:}, \hat{x}^:_{i-1} \rangle$
3: ***end kernel***
4: ***kernel*** $\{q = 1 \ldots N_p, u = 1 \ldots n\}$
5: $\quad \hat{x}^u \leftarrow \langle d^{u,:}, z^: \rangle$
6: ***end kernel***

(b) Backward sweep

1: ***kernel*** $\{q = 1 \ldots N_p, u = 1 \ldots n\}$
2: $\quad z^u \leftarrow b^u - \langle \varepsilon^{u,:}, \hat{x}^:_{i+1} \rangle - \langle \zeta^{u,:}, \hat{x}^:_{j+1} \rangle - \langle \eta^{u,:}, \hat{x}^:_{k+1} \rangle$
3: ***end kernel***
4: ***kernel*** $\{q = 1 \ldots N_p, u = 1 \ldots n\}$
5: $\quad \hat{x}^u \leftarrow \hat{x}^u - \langle d^{u,:}, z^: \rangle$
6: ***end kernel***

Algorithm 7 ACC version of residual calculation in correction steps

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

3.2.3. Residual calculation in correction steps

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 computation can be written in ACC style, as shown in Algorithm 7. The residual takes a simple form (Eq. 3), which can be trivially parallelized without level scheduling. In total, $n \times IJK$ threads can be spawned by this kernel, usually far more than the number of GPU cores. It is thus not necessary to write this algorithm in CL style separately.

3.3. CL-style algorithms

Unlike the more restrictive ACC style, the CL style allows thread-level synchronizations across all threads within the same workgroup, and local shared memory can be used to exchange data within the same workgroup. This is a major advantage in GPGPU programming, as it greatly increases the options in parallel algorithms and offers significantly more optimization potential.

When designing CL-style algorithms the programmer must specify the size of workgroups. To avoid warp divergence, the workgroup size should be a multiple of the warp size (N_{warp}). On the other hand, local shared memory is only shared within a workgroup. To use local shared memory for resolving data dependencies within a kernel, the workgroup size has to be a multiple of the number of threads needed for one grid point. A common multiple is usually chosen. Note that the workgroup size has a preferred range, outside which GPU performance can suffer from inefficient work scheduling. For larger n the best choice for workgroup size would shift from the common multiples of N_{warp} and n^2 to simply the smallest multiple of n^2. Hardware also imposes a hard limit on the number of threads per computation block. For current Nvidia GPU, this limit is 1024, so that in a CL-style algorithm

$n \leq \sqrt{1024} = 32$. Note that ACC-style algorithms are not restricted by this limit, because thread-level synchronization is not involved.

Temporary arrays are used as intermediate storage for computing matrix multiplications. Unlike the ACC version, however, temporary arrays only need to be allocated from the local shared memory, which is private to each workgroup. As a result, the second subscript q_{w} (implied in the pseudocodes) ranges from 1 to L, the number of points handled by one workgroup. From our discussion on the workgroup size, we can see that L is simply the workgroup size divided by the number of threads mapped to one grid point. For example, in matrix-matrix multiplications, if $n = 6$ and workgroup size is 288, then $L = 288/n^2 = 9$.

Algorithm 8 Index reconstruction in CL kernels

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

Because of the workgroup arrangement, each CL thread is assigned a workgroup index I_G (shared within a workgroup, distinct for different workgroups) and a thread index I_T (distinct for different threads within a workgroup). Assuming both I_G and I_T are 1-based, all other indices for mathematics can be reconstructed with Algorithm 8. Note that integer division (\) 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_{w}) in the kernel header.

3.3.1. Factorization

The CL version of the fine-grained factorization algorithm is given in Algorithm 9, which differs from the ACC version significantly. The most important feature of the CL version is that each submatrix element is mapped to one constant GPGPU thread. The constant mapping allows the r-loop to exchange position with the loops over grid points and submatrix elements, while data dependencies are resolved by thread-level synchronization and local shared memory. Because the inversion algorithm shares the same thread mapping with the multiplication algorithm, they can be merged into a single kernel. Note that $s_1^{\vdots\vdots}$ and $s_2^{\vdots\vdots}$ are re-purposed as two alternating read buffers, which resolve data dependency within each r iteration. As a result, only one synchronization point is required within the r-loop.

Due to the nature of warp execution, avoiding conditionals is usually beneficial. Conditional branching inside kernels can often be avoided by two optimization techniques. First, very short conditional branches, such as Line 8 in Algorithm 9, are likely to be converted into fast predicate operations by the compiler. Second, computational branches can often be converted to a linear combination with logical coefficients, such as Line 10 in Algorithm 9, at the cost of a slightly increased amount of computation.

3.3.2. Triangular solves

Similar to the matrix inversion algorithm, the CL version of triangular solves, given in Algorithm 10, further divide the dot products into element-wise operations, which decreases the granular-

Algorithm 9 CL version of FGBILU factorization

Local shared memory array: $s^{n,n}_{3,L}$. A second subscript, $q_{\rm w}$, is implied in all s references.

kernel { $(I_G, I_T) \rightarrow (i, j, k, u, v, q_{\rm w})$ }
  $s_1^{u,v} \leftarrow \langle \alpha^{u,:}, d_{k-1}^{:,v} \rangle$; $s_2^{u,v} \leftarrow \langle \beta^{u,:}, d_{j-1}^{:,v} \rangle$; $s_3^{u,v} \leftarrow \langle \gamma^{u,:}, d_{i-1}^{:,v} \rangle$
  syncthreads
  $d^{u,v} \leftarrow d^{u,v} - \langle s_1^{u,:}, \eta_{k-1}^{:,v} \rangle - \langle s_2^{u,:}, \zeta_{j-1}^{:,v} \rangle - \langle s_3^{u,:}, \varepsilon_{i-1}^{:,v} \rangle$
  syncthreads
  do $r = 1, n$
    $t \leftarrow (r \bmod 2) + 1$; $a \leftarrow (u = r?1 : 0)$; $b \leftarrow (v = r?1 : 0)$; $c \leftarrow (1-a)(1-b)$
    if $(c = 0)$ $s_t^{u,v} \leftarrow d^{u,v}$
    syncthreads
    $d^{u,v} \leftarrow [ab + (a-b)\, d^{u,v} - c\, s_t^{u,r} s_t^{r,v}] / s_t^{r,r} + c d^{u,v}$
  end do
end kernel

ity 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.

Note that it is also possible to rewrite the ACC version of triangular solves to achieve finer granularity, similar to the CL version. Doing so would result in four kernels per hyperplane, instead of two as given in Algorithm 6. This reflects a common challenge in designing massively parallel algorithms, as finer granularity usually introduces more synchronization and memory access overhead. Kernel launches, as the only synchronization mechanism in ACC algorithms, are much more expensive than thread-level synchronization inside a CL kernel. Algorithm designers must weigh costs and benefits to make a balanced choice. Our experience shows that, in this case, the extra overhead associated with ACC kernel launches outweighs the benefit of finer granularity.

4. Numerical experiments

Because FGBILU retains identical mathematical properties of the global sequential BILU, whose behavior has been well studied [23, 26], this paper will not attempt an extensive study of the BILU scheme itself. Instead, the tests in this section will focus on understanding the numerical performance of FGBILU algorithms.

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 6 cores of an Xeon E5645. An OpenMP implementation of BILU based on the coarse-grained wavefront scheme is also included in the tests, in order to reveal the full potential of CPU computations if all cores of the CPU are dedicated to solving linear systems. The fine-grained algorithms is tested on Nvidia M2050 GPGPUs, which has 448 double-precision cores and 3GB of global device memory. M2050 is a representative second-generation GPGPU, with a CUDA compute capability of 2.0. Due to much-improved double-precision and cache memory performance over the first generation, second-generation GPGPUs have been widely deployed in many high-performance computer systems. Although third-generation GPGPUs have been available, we choose to carry out our tests on this GPU to reflect typical performance expected from the most widely adopted GPGPU hardware.

Algorithm 10 CL version of 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** $(v = 1)$ $z^u \leftarrow b^u - \sum s^{u,:}$
5: $\quad$ ***syncthreads***
6: $\quad s^{u,v} \leftarrow d^{u,v} z^v$
7: $\quad$ ***syncthreads***
8: $\quad$ **if** $(v = 1)$ $\hat{x}^u \leftarrow \sum s^{u,:}$
9: ***end kernel***

(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** $(v = 1)$ $z^u \leftarrow \sum s^{u,:}$
5: $\quad$ ***syncthreads***
6: $\quad s^{u,v} \leftarrow d^{u,v} z^v$
7: $\quad$ ***syncthreads***
8: $\quad$ **if** $(v = 1)$ $\hat{x}^u \leftarrow \hat{x}^u - \sum s^{u,:}$
9: ***end kernel***

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 [27, 28]. 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 [29]. 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 [30, 31], two-phase flows [32], and human-induced contaminant transport [33, 34]. An immersed-boundary method [28] is incorporated to enable computations of flow around moving objects, but is not employed in this study.

A simplified test case based on a dynamic stall study [35] is used for the verification. An illustration with exaggerated dimensions is given in Figure 6. 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 15 million cells, divided into 204 blocks. Static load balancing is achieved by mapping multiple blocks of various sizes to a processor. 24 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

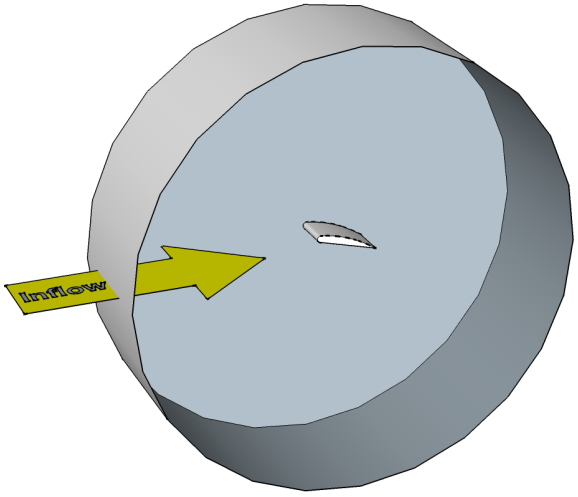


Figure 6: Illustration of the 3D domain for verification

CPU and GPU simulations. A qualitative comparison is given in Figure 7, showing Q=4 isosurfaces for the same setting of the SD7003 case. As we can see, the results are virtually indistinguishable.

A breakdown of the main loop run times of the first 100 physical time steps is given in Figure 8. In this case the number of sub-iterations within each physical time step is 7. The fine-grained linear solver components ("BILU FACT", "BILU SOLV" and "BILU CORR") based on wavefront scheme achieve very good performance, reaching an average 14.2X speedup over the sequential version of the linear solver, while preserving identical mathematical behavior. This is a huge improvement over our previous naive attempt using a coarse-grained implementation of BILU [17]. The flux calculation (RHS) reaches a moderate 6.7X speedup. MPI and data packing (labeled simply as "MPI"), performs slightly (30%) faster on GPUs. Data exchange speed is largely determined by the underlying interconnection of the cluster, regardless of how computation is carried out. Hence, such result is particularly encouraging, as all data exchanges between GPUs must be internally copied to CPUs first. The improved performance is mostly due to the faster data packing procedure on GPUs, which can greatly reduce inter-nodal wait time.

In the current GPU implementation of INCOMP3D, the two components responsible for filling the coefficient matrix, AFILL (invisid flux Jacobian) and TSD (viscous flux Jacobian and time derivative linearization), account for the primary cause of the performance bottleneck. They are similar in nature, as both take field variables as input and generate derivatives, causing large amounts of memory accessed for a relatively small amount of computation. In the current implementation, both tasks are implemented in a coarse-grained fashion, as the computation for each grid point is mapped to one GPU thread, making them highly memory-bound. To obtain full performance on GPUs, an ongoing effort is being undertaken to develop fine-grained versions of these subroutines.

4.1.1. Convergence test on a standalone domain

To demonstrate that FGBILU remains mathematically identical to the original global sequential BILU, a simple convergence test is carried out. The test data is generated by INCOMP3D, a multi-block incompressible Navier-Stokes solver [28]. Due to incorporating a two-equation RANS model we have $n = 6$. Left hand side and right hand side arrays are extracted directly for a $51\times97\times63$ domain and stored offline, which are then used as the input of a standalone linear system solver

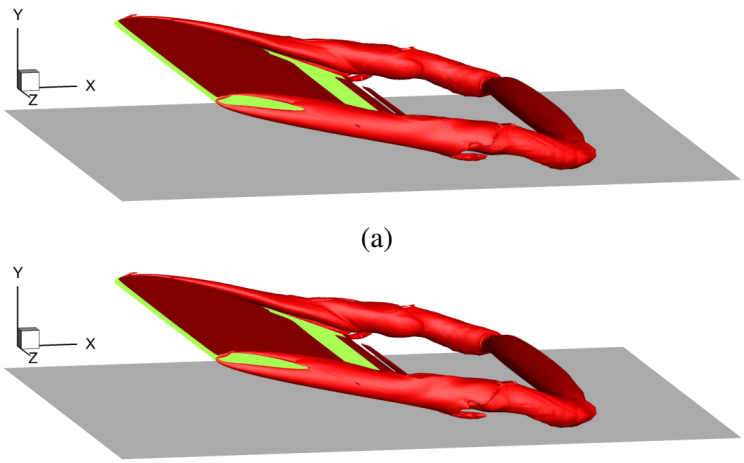


Figure 7: Unsteady RANS simulation results of the (a) unmodified CPU version of INCOMP3D, and (b) GPU version of INCOMP3D. $t = 0.675$s.

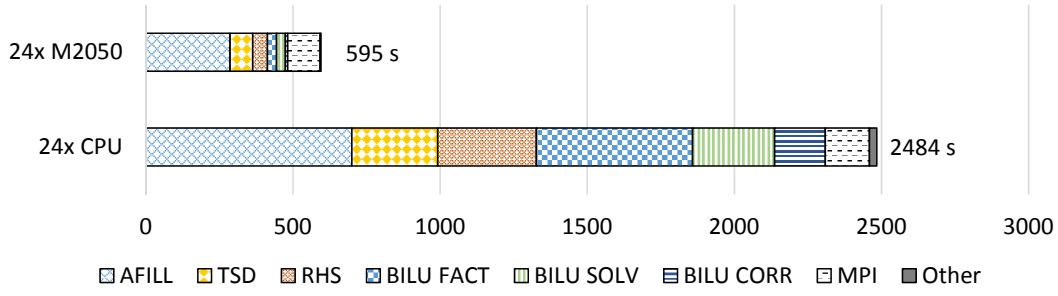


Figure 8: Main loop run times (in seconds).

Iteration	Sequential BILU	FGBILU
0	0.542674993335E+05	0.542674993335E+05
1	0.170814598038E+04	0.170814598038E+04
2	0.131446936261E+03	0.131446936261E+03
⋮	⋮	⋮
11	0.138239874038E-05	0.138239874038E-05
12	0.180227555902E-06	0.180227555901E-06
⋮	⋮	⋮
19	0.126561189429E-12	0.126561185003E-12

Table 1: Sum of squares of residual vector

capable of utilizing all four BILU variances (sequential, OpenMP, OpenACC and CUDA). Table 1 gives the sum of squares of the residual vector $\mathbf{R}_l$ for 20 solution steps, including the initial estimation step and 19 correction steps. We can see that the error does not appear at the 12th digit after the decimal point for 11 correction steps. These are round-off errors due to different numerical operation orders, which are contributed by two sources. First, there are some extremely subtle differences between LU and GJE [36], and their computational procedures take different form. Second, the order of concurrent parallel execution is not deterministic, causing random round-off errors. For this reason the exact numbers given by FGBILU (both OpenACC and CUDA versions) are not reproducible. Neither of these two contributions should cause issues for solving physical PDE.

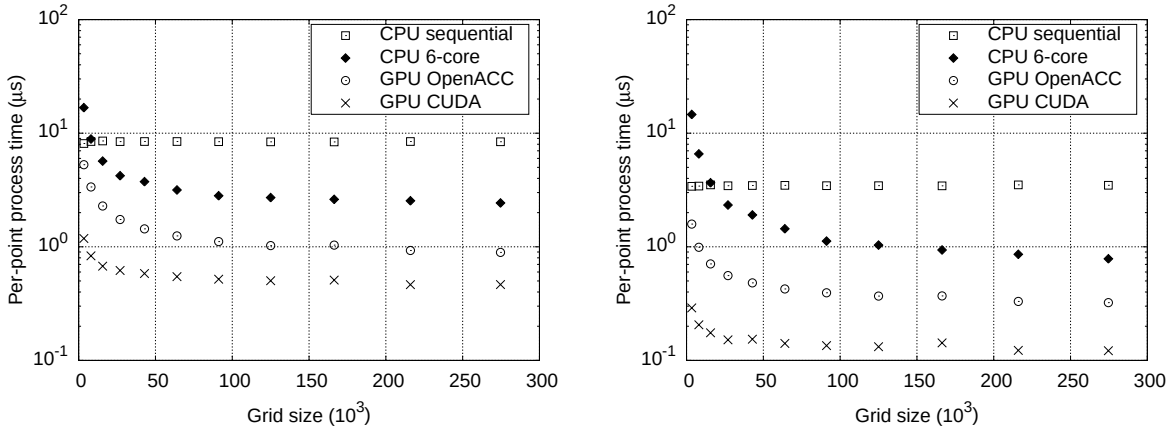
4.2. Scaling tests for different domain sizes

The sequential BILU does not involve level scheduling, while OpenMP, OpenACC and CUDA versions use the same wavefront ordering. The primary concern of algorithms based on a wavefront scheme is synchronization overhead and performance penalty due to smaller wavefronts near the 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 65^3 (275K). The number of PDE unknowns is fixed at $n = 6$. Once the linear system is extracted it is solved in the standalone solver. Each test run includes 3 correction iterations. For kernels that are executed multiple times over the iterations, T is taken as an averaged total kernel run time of one sweep over the domain.

The overall t_p is given in Fig. 9, where the effects of insufficient work load caused by the wavefront scheme near the domain corners is clearly visible. As a general trend, sequential BILU shows almost constant t_p for various domain sizes, while t_p of algorithms with wavefront ordering gradually decreases as the domain size increases. The performance of the 6-core 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. On the GPU, the CUDA version constantly performs better than the OpenACC version. Similar to OpenMP, the OpenACC version is also very sensitive to the domain size. For the 3.4K domain it is only slightly faster than the sequential CPU case.



(a) Total

(b) Factorization

(c) Triangular Solve

(d) Residual calculation

Figure 9: t_p as a function of grid size ($n = 6$, 3 correction steps)

Task	# of threads	FLOP	Data (Bytes)
Factorization	n^2	$8n^3$	$8 \times 7n^2$
Triangular solve	n^2	$8n^2$	$8 \times \left(8n^2 + 3n\right)$
Residual	n	$7n^2$	$8 \times \left(7n^2 + 3n\right)$

Table 2: Complexity and global data access amount

Fig. 9(b) gives the per-point process time of the factorization algorithm, which runs only once in each test, as all the correction steps use the same factorization. While the CUDA version shows moderate sensitivity on domain size, the OpenACC version performs significantly worse on smaller domains. In addition to extra overhead by wavefront ordering, this is also caused by the poor memory access pattern (low cache reuse) in ACC-style algorithms. GPU cores fetch data in batches, called a "cache line", which has a size of 128 bytes on the M2050. Each thread in kernel#5 only processes one DP variable (8 bytes), but it has to fetch the whole cache line of 128 bytes, because $d_{i,j,k}$ for different grid points are located far away from each other. As a result, kernel#3 and #5 become heavily memory-bounded. In fact, it is very common to observe similar kernel run times for kernel #3~5, while their computation loads are very different.

The per-point process times of the triangular solve is given in Fig. 9(c). Note that the triangular solve component runs 4 times, including the initial estimation and correction steps. The average t_p of one run is shown in the figure. Here we can observe a very similar trend of performance penalty due to insufficient load. The CUDA version has a constant mapping of n^2 threads per point, while the two kernels in the OpenACC version both have only n threads per point. The 6-core OpenMP version also suffers from small domain sizes, albeit to a lesser degree than factorization.

The process times of the correction component, shown in Fig. 9(d), offer a sharp distinction from factorization and triangular solve. Note that CUDA version is not necessary for this component, as discussed in Section 3.2.3. Because residual calculation does not require level scheduling, parallelism can be achieved for the whole domain. Even smaller domains can easily supply enough work for all GPU cores. As a result, both CPU and GPU codes show almost constant per-point process time for all domain sizes. Interestingly, t_p of the 6-core OpenMP version decreases slightly for smaller domains, probably due to cache effects.

If we assume that t_p reaches its limit for the largest grid tested, we can do a rough theoretical performance analysis of the three CL-style algorithms. The estimated numbers of computation complexity and global data access amount are given in Table 2. The factorization involves six $n \times n$ multiplications and one $n \times n$ inversion. Counting only floating point multiplications and divisions, each multiplication has n^3 FLOPS and each inversion has roughly $2n^3$ FLOPS, which leads to a computation complexity of $8n^3$ per grid point. We can estimate the complexity of triangular solve (forward + backward) and the residual calculation to be $8n^2$ and $7n^2$, respectively. The amount of global data accessed can be roughly estimated in a similar fashion. Note that triangular solve has two kernels, thus the access to the same array in the two kernels must be counted twice. Access to local shared memory is much faster than global memory, so they are not counted towards the global memory throughput. t_p and speedup numbers by test runs for $n = 6$ are given in Table 3. They are then normalized by the number of threads per grid point, whose results are given in Table 4. Global memory throughput and GFLOPS, can also be determined based on per-thread process times. The speedup number is calculated from the comparison of t_p by sequential BILU on one CPU core and CL-style algorithms.

Task	Sequential CPU t_p	FGBILU t_p	Speedup
Factorization	$3.48\mu s$	$0.122\mu s$	29X
Triangular solve	$0.84\mu s$	$0.0604\mu s$	14X
Residual	$0.51\mu s$	$0.0324\mu s$	16X

Table 3: Per-point process times and speedup ($n = 6$)

Task	Time	FLOP	Data	B/F	Throughput	GFLOPS
Factorization	$3.38ns$	48	56 B	1.17	16.6 GB/s	14.2
Triangular solve	$1.68ns$	8	68 B	8.50	40.5 GB/s	4.8
Residual	$5.41ns$	42	360 B	8.57	66.5 GB/s	7.8

Table 4: Per-thread process times, complexity, global data access amount, estimated overall throughput and GFLOPS ($n = 6$)

An important characteristic number of GPGPU computing is the memory-to-compute (Bytes/FLOP or B/F) ratio, defined as

$$\mathrm{B/F} = \frac{\text{data accessed per thread}}{\text{FLOP per thread}},$$

which is a direct indicator of the algorithm towards being computation-bound (smaller B/F) or memory-bound (larger B/F). Apparently, factorization is a heavily computation-bound algorithm, which carries out many more floating point operations due to the $O(n^3)$ complexity of matrix-matrix multiplications and inversions. On the other hand, the residual calculation is mostly memory-bounded. The triangular solve seems to be a memory-bounded algorithm. However, it has several synchronization barriers, and its sequential summing operations contribute considerable to computation complexity. B/F values can be significantly overestimated. In fact, the triangular solve is a moderately computation-bound algorithm. Algorithms with different B/F ratios require different strategies for performance optimization. To achieve better speedup in a memory-bound algorithm, efforts should be focused on its memory access pattern to improve coalescence. Although the memory access pattern in residual calculation is not perfectly coalescent, it is still rather efficient, delivering nearly half of the theoretical throughput of M2050 (144GB/s). On the other hand, a computation-bound algorithm must minimize non-mathematical operations, which include kernel launching (and, consequently, index reconstructions) overheads, synchronization barriers, redundant operations and warp divergence caused by conditional branches. It is thus not surprising for the triangular solve to have a lower speedup and GFLOPS number than factorization, because it involves more kernel launches and much more warp divergence during the summing steps.

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. This finding is consistent with the fact that, for CPU-only multi-block implicit PDE solvers, shared-memory parallelism (OpenMP) is less preferred than message-passing parallelism (MPI) when ILU-type preconditioners are used. On the other hand, the CUDA implementation performs very well for most of the domain sizes, due to low overhead, constant thread mapping and a highly coalescent memory access pattern. OpenACC offers a compromise between sequential code and well-optimized CUDA codes. While it shows a reasonable performance boost for larger domains, its high sensitivity on small domains renders it somehow less desirable for general scientific computational applications, where domain partitioning does not always allow larger domains.

4.3. Scaling tests for different n

The number of PDE unknowns, n, also has a significant impact on the algorithm efficiency. A larger n allows finer granularity even for smaller domains, and thus has a positive impact on fine-grained algorithms. It is very important to investigate the solver's performance for larger values of n, which are often encountered in multi-component flows.

To study this, we need to generate block-sparse linear system data for various values of n. However, it is rather difficult to compile a collection of PDE solvers with different numbers of unknowns. Instead, a simple technique is employed to generate test data using existing data extracted from INCOMP3D. When RANS modeling is turned off, the number of unknowns in INCOMP3D drops to 4 (u, v, w, p). Along with the data generated by RANS simulation, we can generate test domains of any size for $n = 4$ or $n = 6$. As solutions of Navier-Stokes equations, such data sets are found to converge reliably. Assume that we have two of these test data sets on the same 3D grid: $\{\alpha_1, \beta_1, \gamma_1, \delta_1, \varepsilon_1, \zeta_1, \eta_1, b_1\}$ and $\{\alpha_2, \beta_2, \gamma_2, \delta_2, \varepsilon_2, \zeta_2, \eta_2, b_2\}$, whose number of PDE unknowns are n_1 and n_2, respectively. We can simply construct a new data set $\{\alpha_3, \beta_3, \gamma_3, \delta_3, \varepsilon_3, \zeta_3, \eta_3, b_3\}$, such that, on any grid location (i, j, k):

$$\alpha_3 = \begin{bmatrix} I_{n_1} & 0 \\ 0 & I_{n_2} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} = \begin{bmatrix} \alpha_1 & 0 \\ 0 & \alpha_2 \end{bmatrix}, \beta_3 = \begin{bmatrix} \beta_1 & 0 \\ 0 & \beta_2 \end{bmatrix}, \ldots, b_3 = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.$$

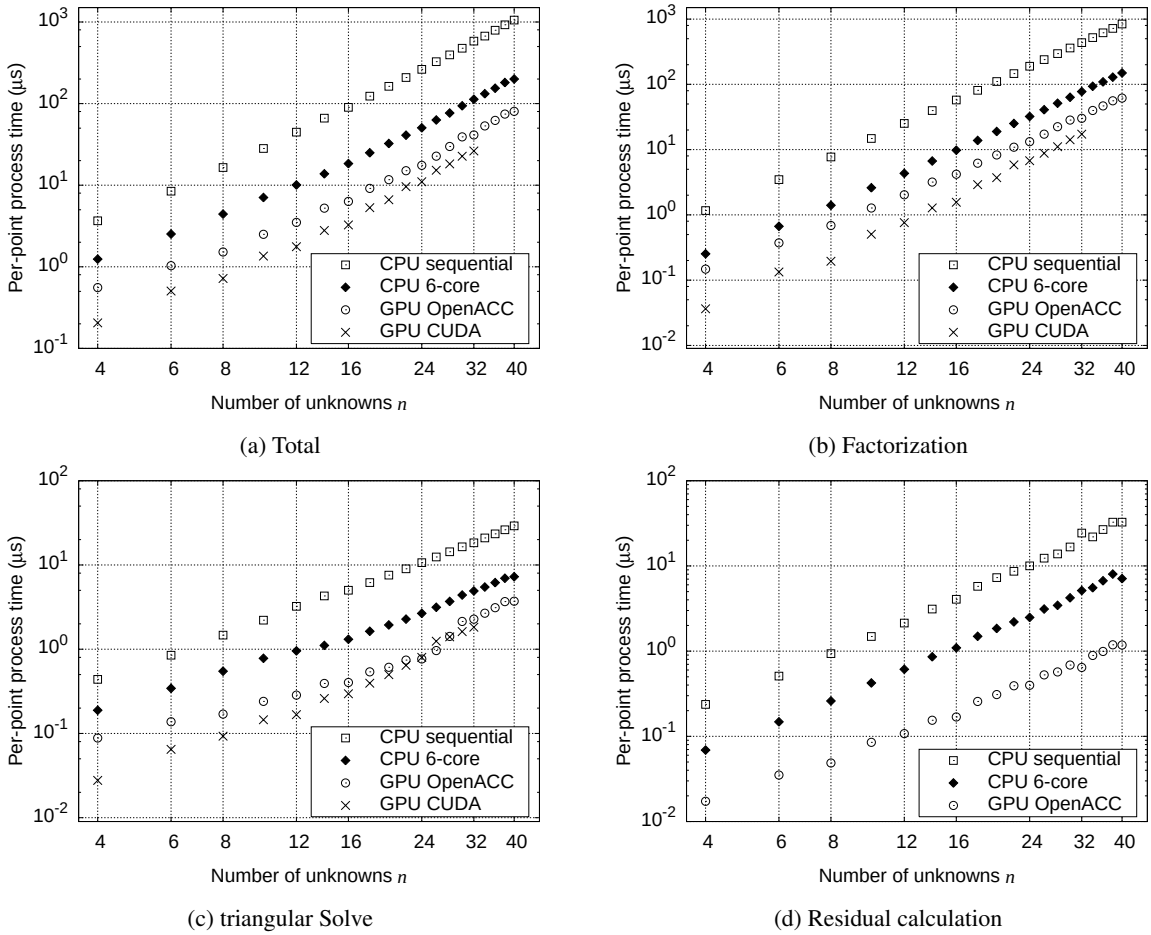
The estimated solution of this new data set, on any grid point, would be

$$\hat{x}_3 = \begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix}.$$

Hence, this procedure can be used to construct a block-sparse linear system for $n_3 = n_1 + n_2$, which has the same grid dimensions as the two original data sets. Using this technique recursively on existing INCOMP3D data for $n = 4$ and 6, we can easily construct test data sets that are guaranteed to converge for any even number (> 4) of PDE unknowns.

Fig. 10 gives the results of a series of runs intended to investigate the effects of varying n. 3 correction steps are used in each run. All test domains therein have roughly constant $n^2IJK \sim 1.8 \times 10^7$, to offer as much work as possible within the limit of GPU memory. Due to the averaging over the domain size IJK, the per-point process time should roughly scale with n^2. To confirm this, Fig. 10 is plotted in log-log scale. Both CPU and GPU versions show good agreement with the n^2 scaling law for larger values of n, while algorithms based on the wavefront scheme deviate from the n^2 scaling for smaller values of n. Note that the domain size decreases for larger n in these tests, so the benefit of larger n clearly outweighs the performance penalty caused by smaller domains.

As shown in Fig. 10(a), the CUDA version clearly out-performs the OpenACC version for all n values up to 32. Fig. 10(b) and 10(c) give the process times of the factorization and the triangular solve components. While the CUDA factorization algorithm performs significantly better than the OpenACC version for all tested values of n, the CUDA version of the triangular solve algorithm does not have a clear advantage for larger values of n. This is mostly due to the two summing stages in the CUDA version, where only 1 out of n threads is doing the summing. For larger n the performance penalty due to idle cores outweighs the efficiency advantage of the CUDA version on kernel launch and synchronization. It is possible, however, to mix the CUDA version of factorization and the OpenACC version of triangular solve to maximize performance for large values of n. In residual calculation, t_p scales with n^2 almost perfectly, as shown in Fig. 10(d), because the algorithm does not involve a wavefront scheme and its associated overhead. Note that CUDA version is not necessary for



(a) Total

(b) Factorization

(c) triangular Solve

(d) Residual calculation

Figure 10: t_p as a function of PDE unknowns n

this component, as discussed in Section 3.2.3. From the fact that t_p of all three FGBILU components scales rather smoothly with n^2, we can see that the negative impact of large workgroup size due to large n values is not particularly concerning for the CL-style algorithms. The primary conclusion drawn from this investigation is that FGBILU algorithms can be expected to perform equally well for PDE with a wide range of n values. In fact, as n increases, the difference in performance between OpenACC and CUDA versions gradually decreases, and they both become less sensitive to small domains.

5. Conclusion

In this study, a fine-grained block ILU(0) preconditioning scheme on a structured 3D grid is developed for GPGPUs. The parallelization of BILU has been a historically challenging subject, and the fine-grained nature of GPGPUs only adds to the overall difficulty of the adoption of BILU on GPGPUs. However, our study indicates that, by using level scheduling techniques and carefully designing fine-grained algorithms, it is possible for BILU to provide a granularity that is fine enough for GPGPU adoption. In the newly developed FGBILU scheme, the inherent data dependency of

BILU is resolved by straightforward one-sweep wavefront ordering. In addition, submatrix operations are further parallelized at the element (scalar) level, significantly refining the granularity.

To achieve true element-level parallelism, efforts are made to parallelize the matrix operations involved in the BILU factorization and triangular solve processes. Three primary tasks are identified: matrix-matrix multiplications, matrix-vector multiplications and multiplications with matrix inversions. While the first two tasks are rather trivial for GPGPU programming, the last task is particularly difficult to parallelize as the original sequential BILU algorithm does an LU decomposition for the factorized diagonal submatrices. As a less obvious solution, instead of an LU decomposition, a matrix inversion is adopted, which is carried out by a compact in-place direct inversion algorithm based on the Gauss-Jordan elimination.

The redesigned fine-grained algorithms are presented in two prevailing programming styles for massively parallel architectures, which are expected to remain relevant as open standards. With the directive-based programming style the codes can be written in traditional loop structures, thus allowing the same source code to be compiled into both sequential CPU binaries and parallel GPU binaries, which greatly facilitates maintainability and portability. The more low-level CL style, on the other hand, allows more massively parallel programming techniques not available in the ACC style, which are found to provide significantly more performance potential.

The ACC and CL algorithms are implemented in OpenACC and CUDA, respectively. The correctness of FGBILU is verified in an incompressible Navier-Stokes solver called INCOMP3D. Standalone tests are carried out with a focus on studying the effects of domain sizes and the number of PDE unknowns n. It is found that increasing domain size and increasing n both have positive impact on performance, because they generally provide more concurrent work and reduce the performance penalty caused by wavefront ordering. ACC algorithms, due to their higher synchronization overhead and less efficient memory access pattern, are found to be more sensitive to smaller domain sizes and smaller n, while CL algorithms, on the other, manage to retain efficiency throughout most of the tested cases. The difference, however, gradually vanishes for n that is sufficiently large.

To the best of our knowledge, FGBILU is the first successful attempt for a pure GPGPU adoption of the original BILU(0) preconditioner to achieve significant speedup. Its combined advantages of mathematical robustness and high performance clears a key roadblock for utilization of GPGPUs in many important subjects of computational physics, where implicit schemes are preferred. The use of two programming styles highly compatible with open standards allows straightforward adoption of FGBILU in most implicit PDE solvers based on structured grids. Because of the inclusion of correction iterations, FGBILU can also work with block coupling based on message passing parallelism.

There is much room for further improvement of FGBILU. First of all, level scheduling of FGBILU is currently carried out on CPUs. Using advanced CUDA programming [37], it is possible to rewrite the algorithms to carry out the level scheduling on the GPU directly, eliminating most of the kernel launch overhead due to level scheduling. Secondly, FGBILU's wavefront scheme in its current form only applies to structured domains. With proper level scheduling techniques FGBILU can be readily modified to work with unstructured grids.

Finally, the development of FGBILU points out a general direction for adoption efforts of similar algorithms on massively parallel architectures. Level scheduling is often considered computationally expensive and inadequate for providing enough parallelism on massively parallel architectures such as GPGPUs. However, if submatrix operations can be efficiently parallelized at the element level, the combined granularity of level scheduling and element-level operations can easily fill GPU cores with well-balanced parallel computations. As stagnation of single core performance persists, such a

approach can be particularly useful for algorithm designers when facing the ever-increasing demand for fine-grained adoptions of sequential ILU-like algorithms.

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.

[1] T. F. Chan, H. A. van der Vorst, Approximate and incomplete factorizations, in: D. Keyes, A. Sameh, V. Venkatakrishnan (Eds.), Parallel Numerical Algorithms, Vol. 4 of ICASE/LaRC Interdisciplinary Series in Science and Engineering, Springer Netherlands, 1997, pp. 167–202. 1

[2] D. Hysom, A. Pothen, A scalable parallel algorithm for incomplete factor preconditioning, SIAM Journal on Scientific Computing 22 (6) (2001) 2194–2215. 2

[3] P. Hénon, Y. Saad, A parallel multistage ILU factorization based on a hierarchical graph decomposition, SIAM Journal on Scientific Computing 28 (6) (2006) 2266–2293. 2

[4] M. M. monga Made, H. A. van der Vorst, Parallel incomplete factorizations with pseudo-overlapped subdomains, Parallel Computing 27 (8) (2001) 989–1008. 2

[5] J. Thies, F. Wubs, Design of a parallel hybrid direct/iterative solver for CFD problems, in: IEEE 7th International Conference on e-Science, 2011, pp. 387–394. 2

[6] V. Heuveline, D. Lukarski, N. Trost, J.-P. Weiss, Parallel smoothers for matrix-based geometric multigrid methods on locally refined meshes using multicore CPUs and GPUs, in: R. Keller, D. Kramer, J.-P. Weiss (Eds.), Facing the Multicore - Challenge II, Vol. 7174 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 158–171. 2

[7] Y. Saad, H. A. van der Vorst, Iterative solution of linear systems in the 20th century, Journal of Computational and Applied Mathematics 123 (2000) 1 – 33, numerical Analysis 2000. Vol. III: Linear Algebra. 2

[8] I. Duff, G. A. Meurant, The effect of ordering on preconditioned conjugate gradients, BIT Numerical Mathematics 29 (4) (1989) 635–657. 2

[9] S. Doi, T. Washio, Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorizations, Parallel Computing 25 (1999) 1995–2014. 2

[10] 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

[11] 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. 2

[12] 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. 2

[13] 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. 2

[14] 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. 2

[15] A. C. Duffy, D. P. Hammond, E. J. Nielsen, Production level CFD code acceleration for hybrid many-core architectures, Tech. rep., NASA/TM-2012-217770 (2012). 2

[16] L. Fu, Z. Gao, K. Xu, F. Xu, A multi-block viscous flow solver based on GPU parallel methodology, Computers & Fluids 95 (0) (2014) 19 – 39. 2

[17] 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. 2, 9, 16

[18] H. van der Vorst, High performance preconditioning, SIAM Journal on Scientific and Statistical Computing 10 (6) (1989) 1174–1185. 2, 8

[19] The OpenACC Application Programming Interface (2013). 3

[20] CAPS Enterprise, The OpenHMPP Open Standard (2009). 3

[21] J. Beyer, E. Stotzer, A. Hart, B. de Supinski, OpenMP for accelerators, in: B. Chapman, W. Gropp, K. Kumaran, M. M uller (Eds.), OpenMP in the Petascale Era, Vol. 6665 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 108–121. 3

[22] J. Meijerink, H. A. van der Vorst, An iterative solution method for linear systems of which the coefficient matrix is a symmetric m-matrix, Mathematics of computation 31 (137) (1977) 148–162. 3

[23] 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. 3, 14

[24] G. Wittum, On the robustness of ILU smoothing, SIAM Journal on Scientific and Statistical Computing 10 (4) (1989) 699–717. 3

[25] G. Meurant, The block preconditioned conjugate gradient method on vector computers, BIT Numerical Mathematics 24 (4) (1984) 623–633. 11

[26] P. Concus, G. Golub, G. Meurant, Block preconditioning for the conjugate gradient method, SIAM Journal on Scientific and Statistical Computing 6 (1) (1985) 220–252. 14

[27] J. R. Edwards, M.-S. Liou, Low-diffusion flux-splitting methods for flows at all speeds, AIAA Journal 36 (9) (1998) 1610–1617. 15

[28] 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. 15, 16

[29] A. J. Chorin, Numerical solution of the Navier-Stokes equations, Mathematics of Computation 22 (104) (1968) 745–762. 15

[30] 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. 15

[31] 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

[32] 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. 15

[33] J.-I. Choi, J. R. Edwards, Large eddy simulation and zonal modeling of human-induced contaminant transport, Indoor air 18 (3) (2008) 233–249. 15

[34] J.-I. Choi, J. R. Edwards, Large-eddy simulation of human-induced contaminant transport in room compartments, Indoor Air 22 (1) (2012) 77–87. 15

[35] 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. 15

[36] G. E. Forsythe, C. B. Moler, Computer solution of linear algebraic systems, Vol. 7, Prentice-Hall Englewood Cliffs, NJ, 1967. 18

[37] 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. 24