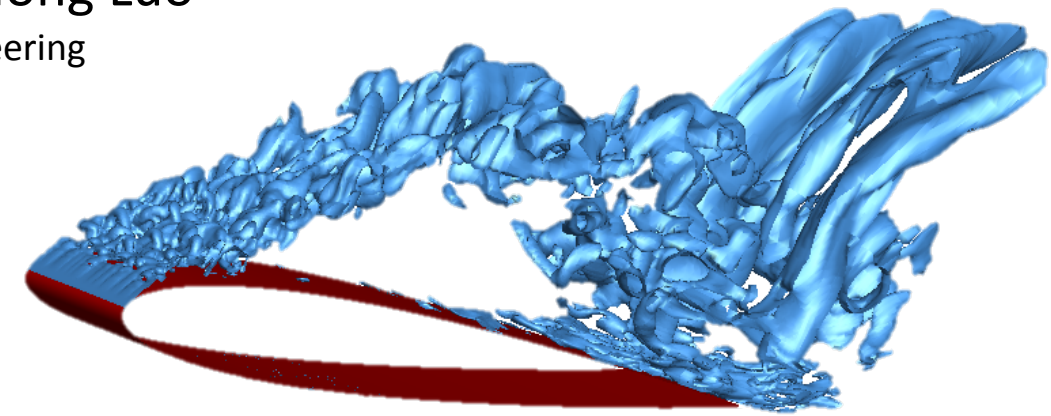
Performance Assessment of A Multi-block Incompressible Navier-Stokes Solver using Directive-based GPU Programming in a Cluster Environment

Lixiang (Eric) Luo, Jack Edwards, Hong Luo
Department of Mechanical and Aerospace Engineering
Frank Mueller
Department of Computer Science

North Carolina State University



Supported by Air Force Office of Scientific Research
Basic Research Initiative program grant FA9550-12-1-0442

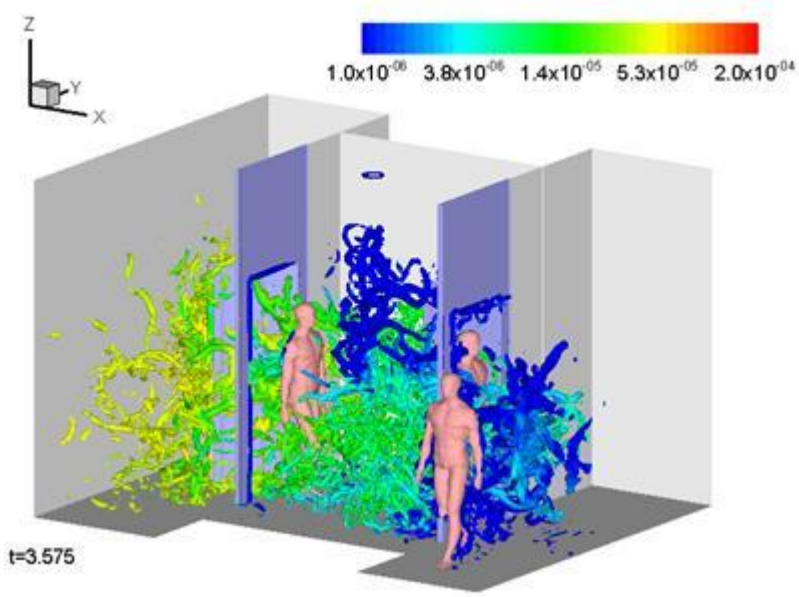
Porting a Legacy Code: INCOMP3D

INCOMP3D is an incompressible Navier-Stokes solver

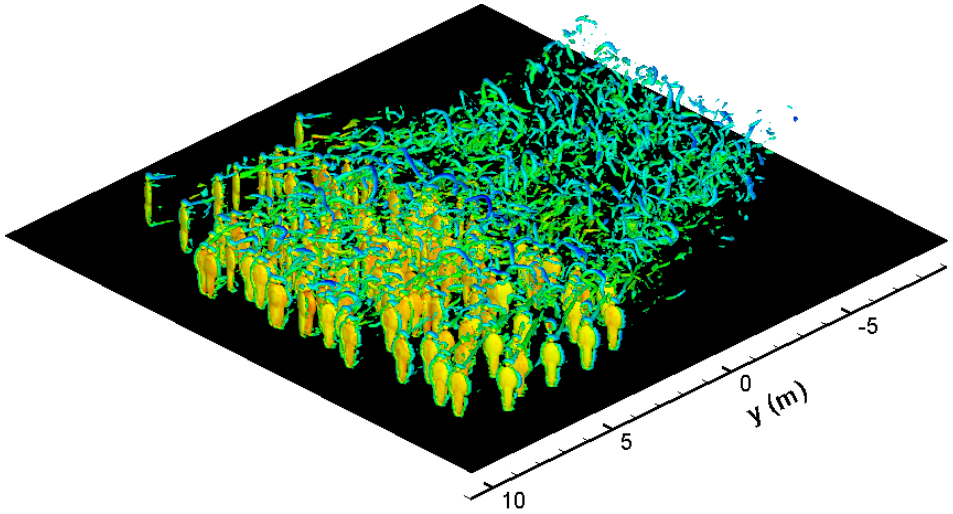
- Based on the Low Diffusion Flux Splitting Scheme (LDFSS) upwind finite volume method
- Second or higher order spatial accuracy
- Immersed boundary (IB) method
- Artificial compressibility (AC) method
- Flexible multi-block computational domains consisting of an arbitrary number of structured mesh blocks
- Implicit time evolution using ILU-preconditioned linear system solvers
- Two-phase flow modeling capability
- Parallelization by MPI

INCOMP3D Capability Showcase

- Contaminant Transport



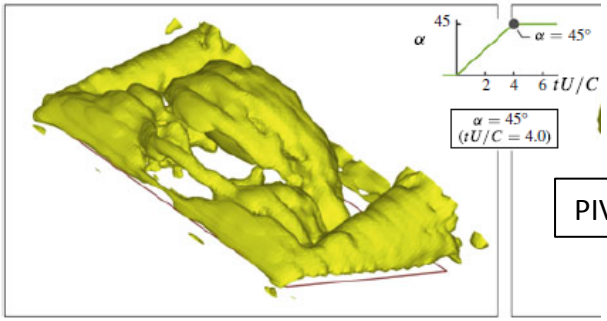
Three people walking through an airlock



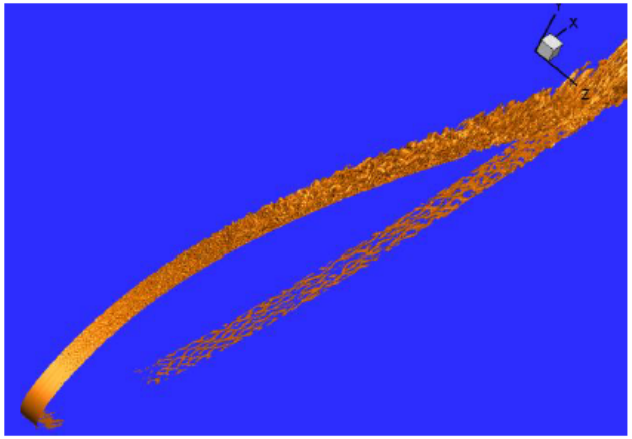
Movement of a crowd of 150 ellipsoidal ‘people’

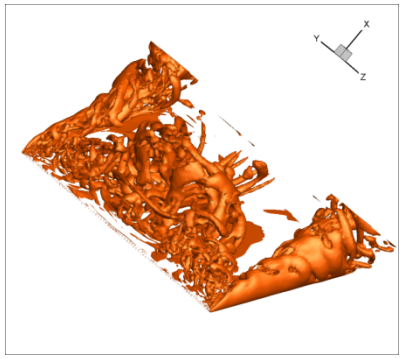
INCOMP3D Capability Showcase

- Applied Aerodynamics

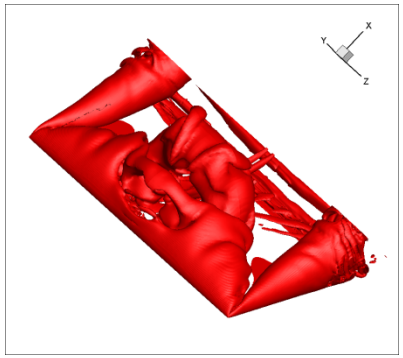


PIV (Rockwell)

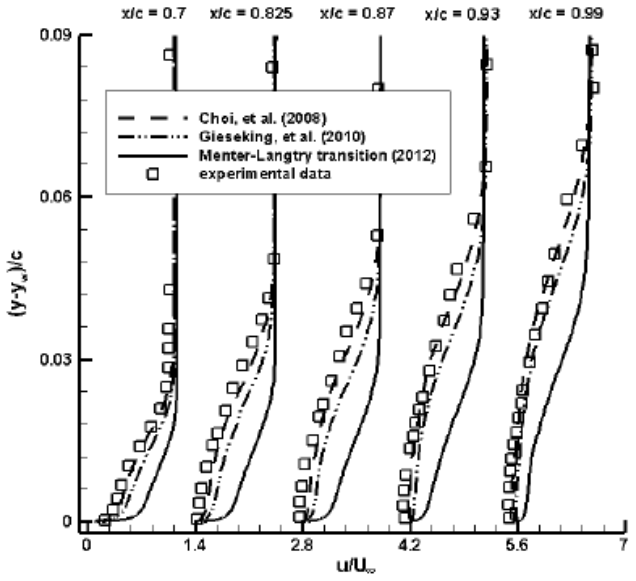




LES – Smagorinsky model



RANS – Spalart Allmaras model

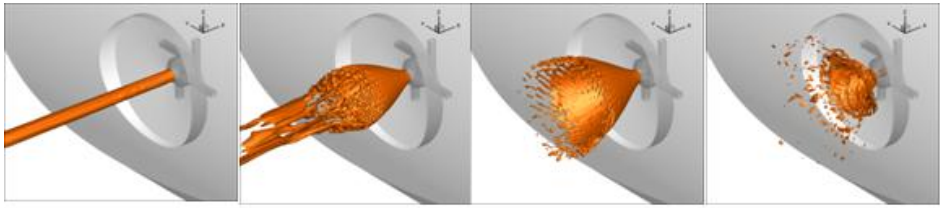


Pitch-up of airfoil at low Reynolds number

LES of Aerospatiale airfoil near static stall

INCOMP3D Capability Showcase

- Two-Phase Flows



Evolution of conical jet structure with increasing Reynolds number (left to right) during pressure-swirl atomization

Flow in an aerated-liquid atomizer (dark blue- aerating gas; light blue – liquid water)

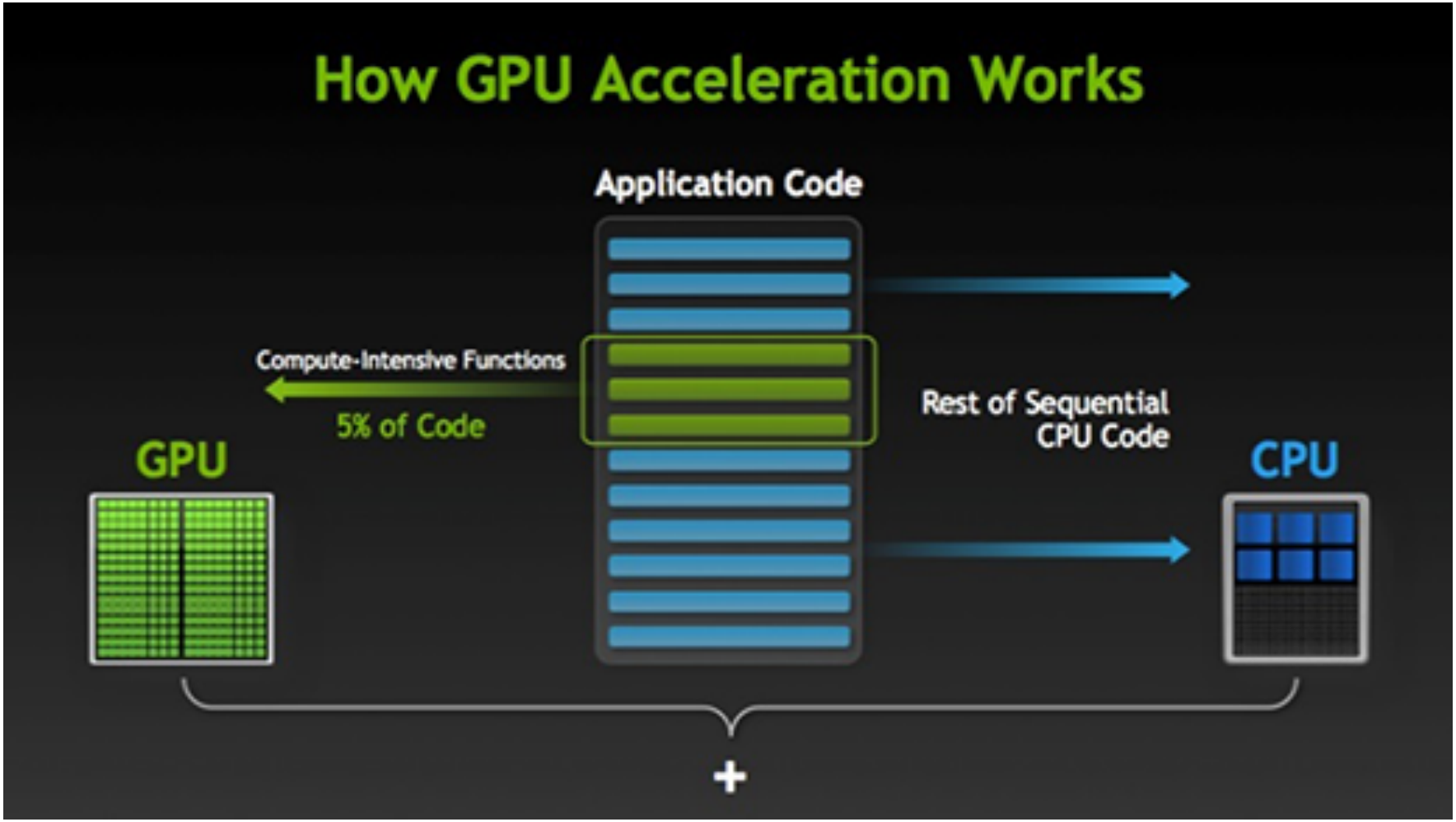
Overarching Goal

- To apply co-design principles to develop a version of INCOMP3D completely adaptable to GPU and GPU/CPU architectures while maintaining the full functionality of the original
- Several challenges to overcome
  - Code re-design for maximum efficiency on GPUs
  - MPI parallelism
  - Implicit solver functionality

Outline

- Introduction
- Porting a simplified 2D version
  - OpenACC implementation
  - CUDA implementation
  - Performance comparison
- Porting the full 3D version
  - Coloring scheme in flux calculation
  - Redesigning the internal data structures
  - Making MPI work with OpenACC
  - Performance comparison
- Summary

Introduction to GPGPU



OpenACC vs. CUDA: Matrix Addition

OpenACC Fortran

```
Real*8, Dimension(N1,N2) :: A, B, C
…
A = … ; B = …
…
!$acc data copyin(A,B) copyout(C)

!$acc kernels
Do i=1:N1
Do j=1:N2

   C(i,j) = A(i,j) + B(i,j)

End Do
End Do
!$acc end kernels

!$acc end data

Print *, C
```

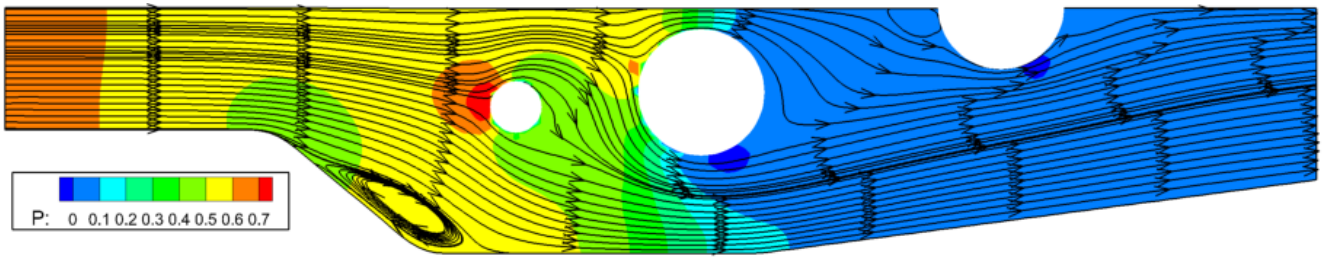
CUDA Fortran

```
                    --- Main ---
Real*8, Dimension(N1,N2) :: A_h, B_h, C_h
Real*8, Device, Dimension(N1,N2) :: A_d, B_d, C_d
…
A_d = A_h; B_d = B_h …
Call cu_madd<<<CUDA args>>>( A_d, B_d, C_d )
C_h = C_d
…
Print *, C_h

                  --- Kernel ---
Attributes(global) Subroutine cu_madd(A,B,C)
Real*8, Device, Dimension(N1,N2) :: A, B, C
Integer :: i, j

i = (blockIdx%x-1)*blockDim%x + threadIdx%x
j = (blockIdx%y-1)*blockDim%y + threadIdx%y

If ( i<=N1 .AND. j<=N2 ) Then
   C(i,j) = A(i,j) + B(i,j)
End If

End Subroutine cu_madd
```

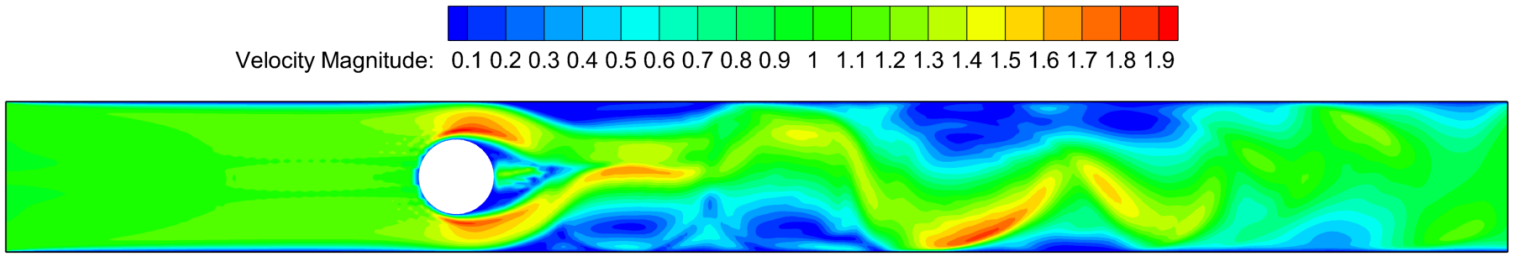
First Attempt: A Simplified Version - IN2D

A much-simplified 2D version using fully explicit time integration is ported to GPGPU as the first attempt. No MPI is involved.

Example1: steady state, Re=200



Example 2: time-accurate, Re=3000, t=20s



OpenACC Implementation of IN2D

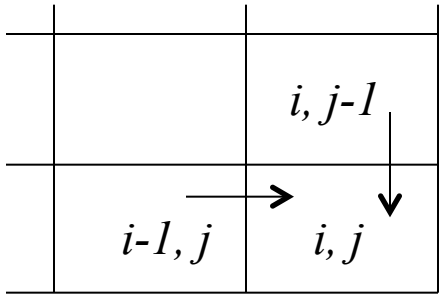
- All computation-intensive tasks inside the main loop must be carried out by GPU
- Calculations in the main loop must be redesigned to allow massive parallelization
  - Parallelizing the computation of the whole grid
- Data management is optimized for minimal CPU-GPU transfer
  - All essential arrays remain on GPU main memory
  - Temporary data arrays are created directly on GPU memory when necessary, avoiding CPU-GPU transfers
- The modular structure of the code is preserved
  - OpenACC directives facilitates writing codes in a modular fashion

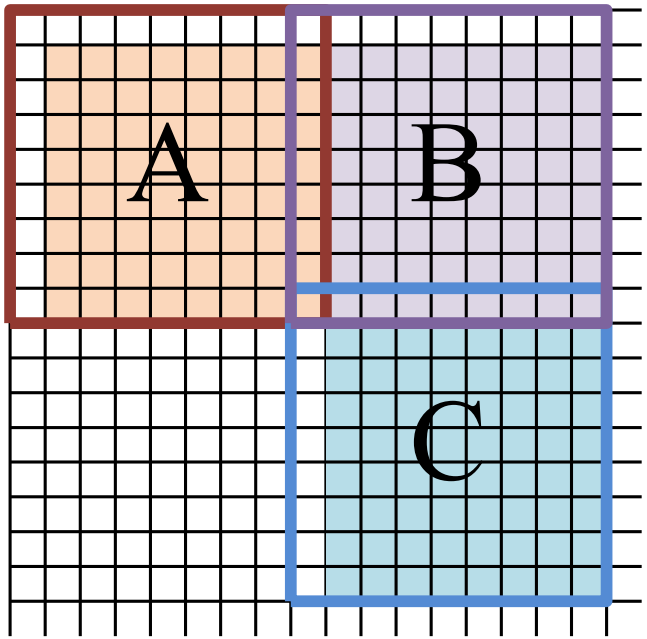
CUDA Implementation of IN2D

- A CUDA Fortran version of INS2D is implemented
  - To understand the inner working of GPGPU
  - To explore the potential and limit of GPGPU
- One step beyond ACC: to minimize data transfer between GPU main memory and L1/shared memory
  - Residual calculation and time marching are carried out locally within a thread block.
  - Residual array are not transferred in/out L1 cache/shared memory; time step is local to each thread, thus eliminating the time step array.
  - Overlapping blocks are used, since residual calculation depends on data outside the block.
  - Main loop consists of one monolithic kernel, since shared memory is only consistent inside a block.

CUDA: Overlapping Blocks

- Why overlapping?
  - Residual calculation at (i,j) depends on fluxes at (i,j), (i-1,j) and (i,j-1):
    $R\downarrow i,j\uparrow = f\downarrow i,j\uparrow x - f\downarrow i-1,j\uparrow x + f\downarrow i,j\uparrow y - f\downarrow i,j-1\uparrow y$
- Blocks overlap by 1 row/column
  - The top row and left-most column is not updated by the current block. Instead, the blocks on the left/top updates those rows and columns
  - Boundary blocks relies on boundary conditions, such as “A” and “B”
- Flux calculation is redundant on the overlapped rows/columns

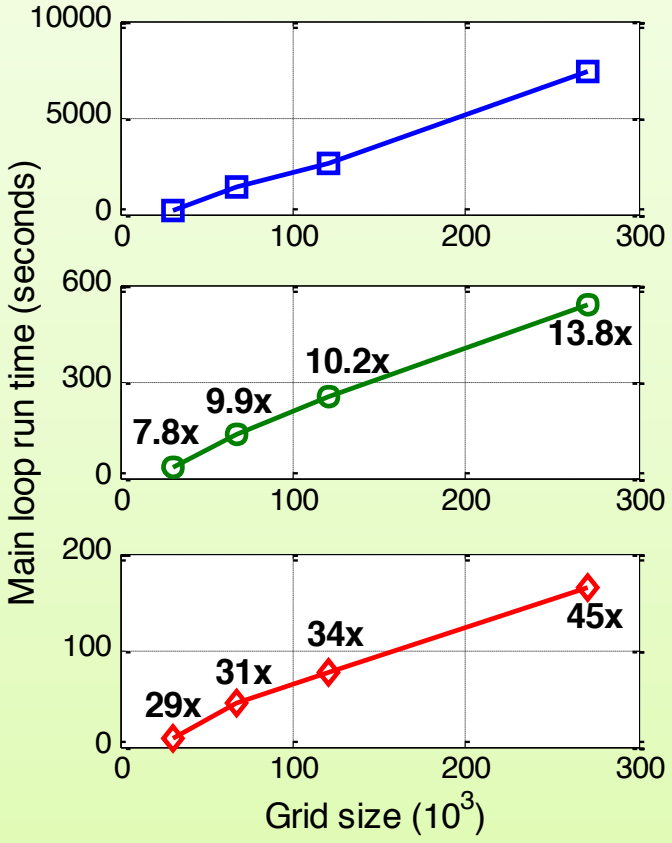




Performance Assessment

- ACC and CUF both achieved significant speedup over CPU
- CUF achieved better performance, but requires much more effort to port and maintain
  - Direct access of shared memory allows greater flexibility on algorithms.
  - Easier to make mistakes; harder to debug.
- ACC provides a good compromise between CPU and CUF
  - Good speedup (~10x), with moderate effort on porting.
  - Easier to debug and maintain.

Example: steady-state flow inside a channel with 3 circular obstacles, Re=200. All results (in seconds) are obtained using nVidia c2050.



Full 3D Version

- Upwind schemes must be carefully selected
  - Must be re-designed for massive parallelization
  - Large stencil must be avoided
- More complex data structures
  - The original version has an internal data structure designed for reactive flow simulations
  - Multiple blocks can be mapped to one processor, which requires careful arrangement of block information
  - Must remain flexible enough for further expansion
- MPI is inevitable
  - Dictated by the size of 3D problems
  - Must be able to work with GPU
  - Calls for efficient data packing algorithm on GPU

Coloring Scheme in Flux Calculation

Fluxes are calculated on 3 directions. On the inner-most direction a difference is carried out to calculate residul. For example, on the "i" direction

```
do k=k0,k1 ; do j=j0,j1
   do i=i0,i1
       q(i) = …
   end do
   do i=i0,i1
       b(i,j,k) = b(i,j,k) – ( q(i)-q(i-1) )
   end do
end do ; end do
```

Data dependence prevents massive parallelization. New algorithm:

```
do ipass=0,1   ! Odd/even passes
   !$acc loop independent
   do k=k0,k1
   !$acc loop independent
   do j=j0,j1
   !$acc loop independent
   do i=i0+ipass,i1,2
       q = …
       b(i,j,k) = b(i,j,k) – q
       b(i+1,j,k) = b(i+1,j,k) + q
   end do ; end do ; end do
end do
```

Expandable Data on GPU: Array of Arrays (AOA)

- A discretized description of a number of field variables $f\downarrow i$ $(i=1...N)$ defined on a given grid can be organized as Fortran arrays $F\downarrow i$ $(i=1...N)$.
- The number of total variables N is determined at runtime.
- The size of $F\downarrow i$ is determined at runtime.
- $F\downarrow i$ can be accessed in different shapes in different parts of the code.
  - A typical scenario: 3D access generally, 1D access in a linear system solver.
- A subroutine interface should maximize its generality on operating on different field variables.

The "Extra Dimension" Approach

Since all variables are defined on the same grid, the variables naturally forms an extra dimension. On a structured i-j-k grid, all variables can be stored on a 2D Fortran array:

```
Real, Dimension(:,:), Allocatable :: F_ALL
```

The subroutine which access one field variable in 3D can be written as

```
Subroutine DoSth3D( F,i0,i1,j0,j1,k0,k1 )
   Real, Dimension(i0:i1,j0:j1,k0:k1) :: F
   …
End Subroutine
```

The 3D subroutine can be called

```
Program MyCFDCode
   Real, Dimension(:,:), Allocatable :: F_ALL
   …
   Allocate( F_ALL(ii*jj*kk,1:N) )
   …
   Call DoSth3D( F(1,i_var),i0,i1,j0,j1,k0,k1 )
   …
End Program
```

Index of the field variable to be accessed

where ii=i1-i0+1, jj=j1-j0+1 and kk=k1-k0+1.

It doesn't work with OpenACC!

The "Array of Arrays" Approach

Instead of using an extra dimension, each field variable has its own array. The subroutine which assesses one field variable in 3D remains the same. Extra allocation statements are needed:

```
Program MyCFDCode
   Type AR1D
      Real, Allocatable, Dimension(:) :: a
   End Type
   Type(AR1D), Dimension(:), Allocatable :: F_ALL
   …
   Allocate( F_ALL(N) )
   Do i_var = 1,N
      Allocate( F_ALL(i_var)%a(ii*jj*kk) )
   End Do
   …
   Call DoSth3D( F(i_var)%a,i0,i1,j0,j1,k0,k1 )
   …
End Program
```

The difference is that in AOA a whole field array is passed instead of being a portion as in the "extra dimension" approach. DoSth3D remains the same.

AOA in OpenACC Fortran

Sample codes:

```
Subroutine DoSth3D_GPU( F,i0,i1,j0,j1,k0,k1 )
   Real, Dimension(i0:i1,j0:j1,k0:k1) :: F
   …
   !$acc kernels present( F )
   …
   !$acc end kernels
End Subroutine

Program MyCFDCode
   Use openacc
   Type(AR1D), Dimension(:), Allocatable :: F_ALL
   …
   Allocate( F_ALL(N) )
   Do i_var = 1,N
      Allocate( F_ALL(i_var)%a(ii*jj*kk) )
   End Do
   … ! Initialization on CPU
   Do i_var = 1,N
      Call acc copyin( F_ALL(i_var)%a )
   End Do
   …
   Call DoSth3D_GPU( F(i_var)%a,i0,i1,j0,j1,k0,k1 )
   …
End Program
```

Getting OpenACC to Work with MPI

- MPI implementation must be aware of GPGPU
  - Automatic GPU memory detection and proper handling of GPU data over a cluster
  - Better performance
  - Reduced code maintenance effort
  - Unified interface maximize portability for further development
  - GPU-aware MPI implementations: OpenMPI, MVAPICH2
- MVAPICH2 is selected for our project
  - Widely adopted
  - Better utilization of InfiniBand interconnection used by our cluster
  - MVAPICH2 is actively updated
  - Expert support, which is critical for any state-of-art endeavor, is readily available by collaborators in Virginia Tech

Detection of OpenACC Variables by MVAPICH2

GPU memory detection in MVAPICH2 is based on address. However, the "host_data use_device" construct is only implemented in PGI C compiler, but not in Fortran. A custom Fortran interface, written in C is used to resolve this problem.

First, write a C wrapper:

```
void accmpi_isend( double* restrict accbuf,
   int cnt, int datatype, int dest, int tag, int comm,
   int* request, int* istat) {
#pragma acc host_data use_device( accbuf )
   *istat = MPI_Isend(accbuf,cnt,datatype,dest,tag,comm,request);
}
```

Then, define a Fortran interface for the C function above:

```
interface
subroutine accmpi_isend(accbuf,cnt,datatype,dest,tag,comm,request,istat) &
      bind(c,name='accmpi_isend')
   use, intrinsic :: iso_c_binding
   real(c_double), dimension(*) :: accbuf
   integer(c_int), value :: cnt, datatype, dest, tag, comm
   integer(c_int) :: request, istat
end subroutine
end interface
```

Rewriting Data Packing of Ghost Cells

The original INCOMP3D ghost cells are copied one-by-one to a continuous buffer set up by the solver. It is strictly sequential:

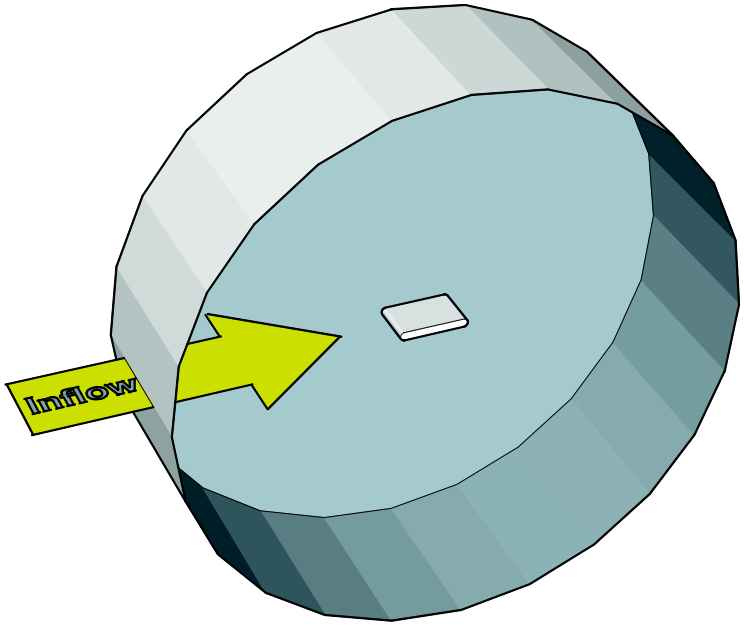
```
size_buffer = 0
do nv = nv0, nv1 ; do k = k0,k1,ks ; do j = j0,j1,js ; do i = i0,i1,is
    size_buffer = size_buffer + 1
    MPI_buffer(size_buffer) = data(i,j,k,n)
end do ; end do ; end do ; end do
```

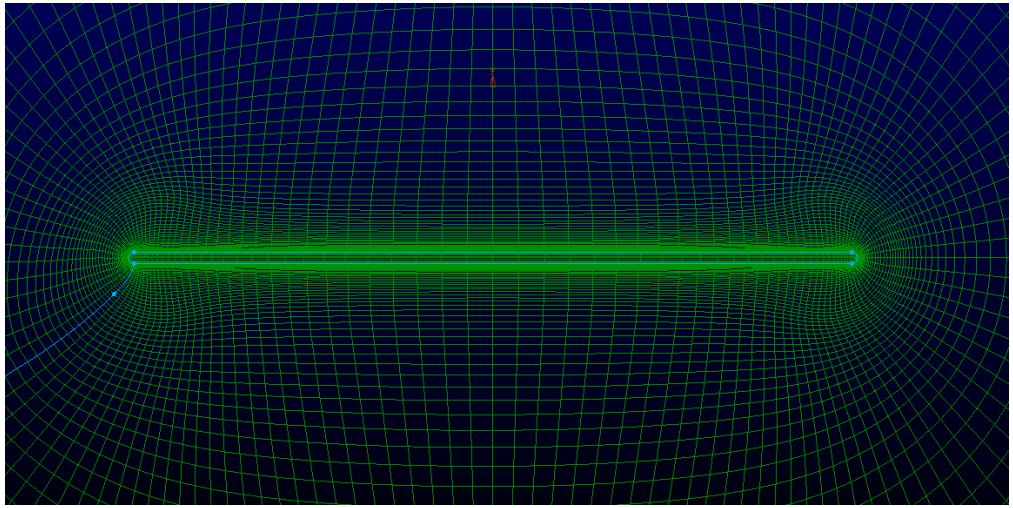
A parallel algorithm using calculated-index:

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

3D Test Domain

- Based on a dynamic stall study.
  - The airfoil is reduced to a flat plate. Symmetry allows better block partitioning and load balancing.
- 7M structured grid cells. 152 blocks mapped to 32 processes.

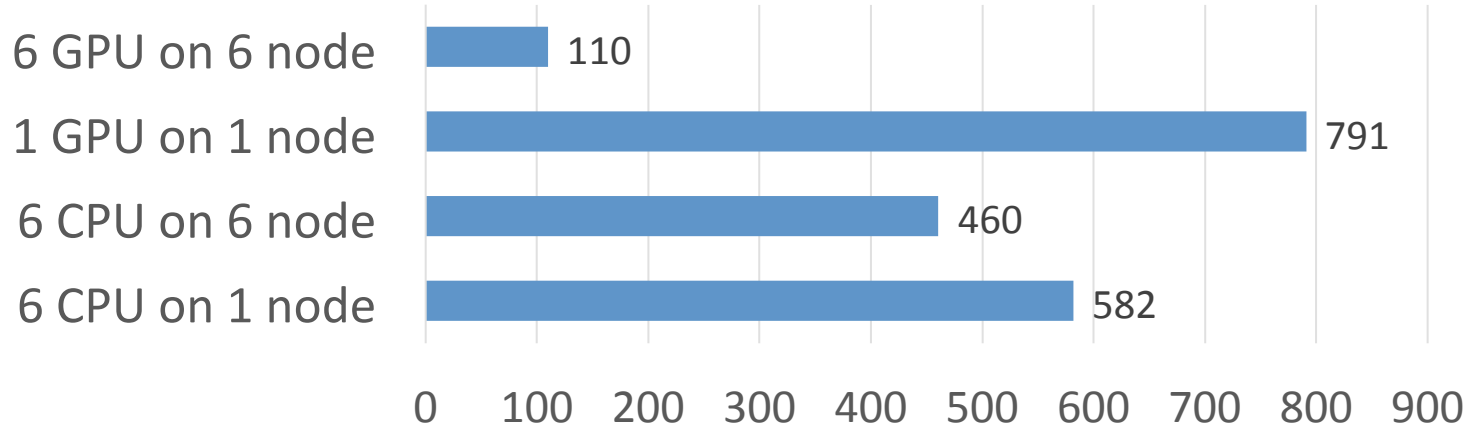




Performance Assessment

- Full 3D simulations
- Full GridPro-style block connection is supported
- Performance tests show moderate speedup of 4x (6GPU/6nodes vs 6CPU/6nodes)

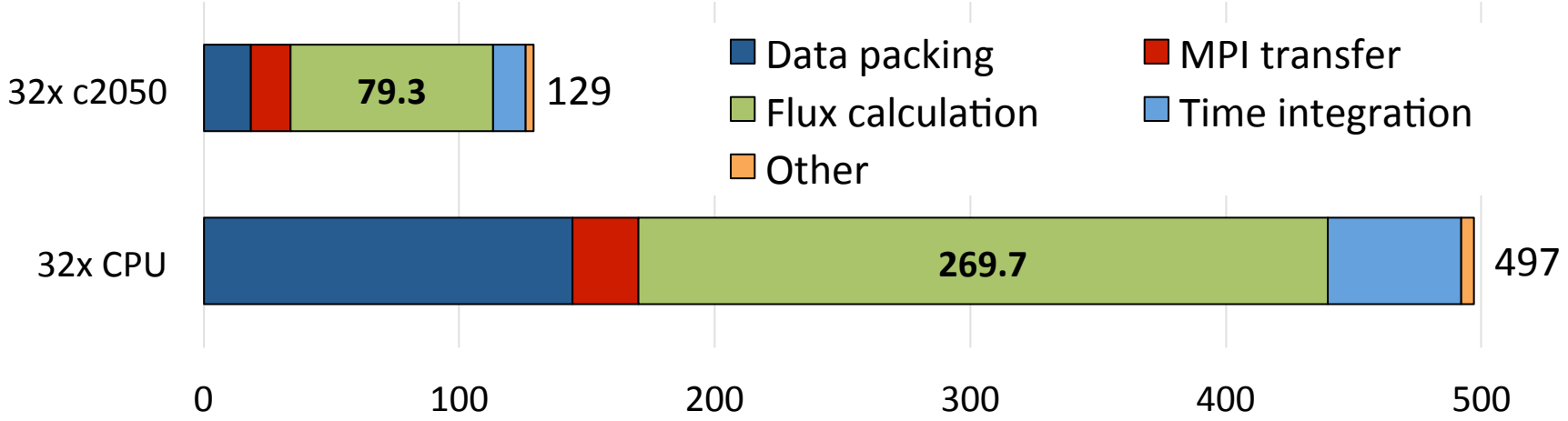
Test case: 200x50x70 (700K) grid, 2000 steps, steady state



Breakdown of Run Times

- Flux calculation are still dominant
  - Flux calculation: 3.4x, time integration: 4x.
- Manual data packing is very efficient
  - 7.8x speed up
- MPI transfer is faster with GPU, even with extra GPU-CPU exchanges.

Test case: 7M grid, 2000 steps, steady state, run on 32 nodes.



Tackling Implicit Methods: BILU

- Implicit methods inherently require some kind of matrix factorization when solving linear systems.
- In the case of INCOMP3D, block incomplete LU factorization (BILU) is used as the preconditioner for solving linear systems.
- For each grid point (i,j,k), one 6x6 BILU is carried out, which is similar to a 6x6 matrix inversion w.r.t. computation complexity.
- BILU is inherently recursive, with limited parallelism to be extracted.
  - Wavefront ordering is being considered for BILU parallelism

OpenACC Version of BILU

- OpenACC implementation restrictions prevent direct porting of existing BILU algorithms
  - In PGI's currently implementation of OpenACC, any indirectly addressed arrays must be allocated in GPU global memory.
  - For a domain block of reasonable size ($10\hat{}6$), the temporary arrays used by the BILU algorithm will take huge amount of GPU global memory ($10\hat{}6 \times 6 \times 6 \times 3 \times 8 = 864\text{Mbytes}$). Also, caching this amount of data is very slow.
- Hence, temporary arrays must be local to GPU threads
  - BILU must be manually unrolled.
  - References to elements of three temporary 6x6 arrays must be converted to scalar variables.
  - Think of an SIMD scenario, where no indirect addressing is allowed.

Summary

- Certain CFD algorithms must be redesigned to fit the characteristics of GPGPU
- OpenACC is a good compromise between speed and code maintainability/reusability
- Directive-based is a promising approach to port legacy codes onto GPGPU
- Technical issues still pose hurdles, but the overall situation is much improved than several years ago

Upcoming Tasks

- Further profiling
  - More profiling data is needed for better understanding of performance limitations
  - MPI profiling can be very tricky
- More complex meshes
  - Data packing and MPI transfer bottlenecks may return if data volume increases significantly
  - Unusual connectivity will likely reduce data packing efficiency
- LES simulations and implicit methods
- Further optimization of flux calculation kernels
  - Current speed up of flux calculation is only 3.4x, further optimization should be possible.