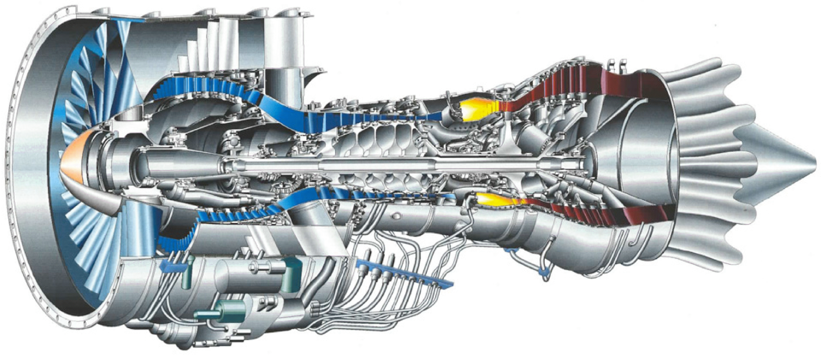
Case Study:
Computational Fluid Dynamics (CFD)

Patrick LeGresley

plegresley@nvidia.com

High Performance Computing with CUDA
Dresden, Germany
June 16, 2008

Challenging Problems in CFD



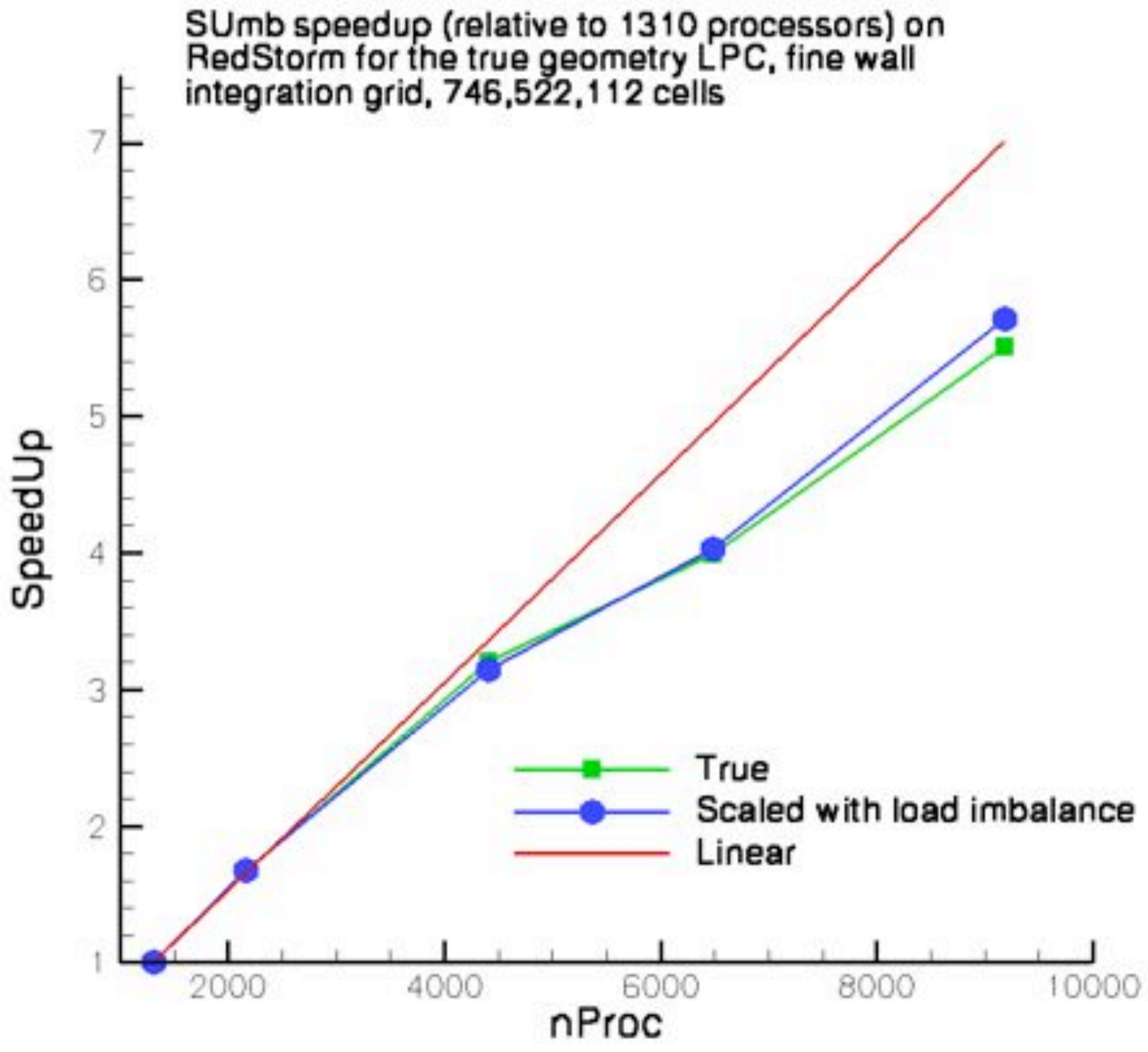
Computational Requirements for a Jet Engine Simulation

- Entire engine geometry (full wheel):
  - Extremely large computational grids (about 1 billion cells)
  - Using 50,000 CPUs, the computational time is about 14 days for each flow-through time

- Reduced model:
  - 20^o sector (by scaling the blade counts) ~ 75 millions cells
  - Coarser grid ~ 16 millions cells
  - Using 1000 CPUs (ALC), the computational time is about 10 days
  - Reduced storage requirements ~ 5 TB

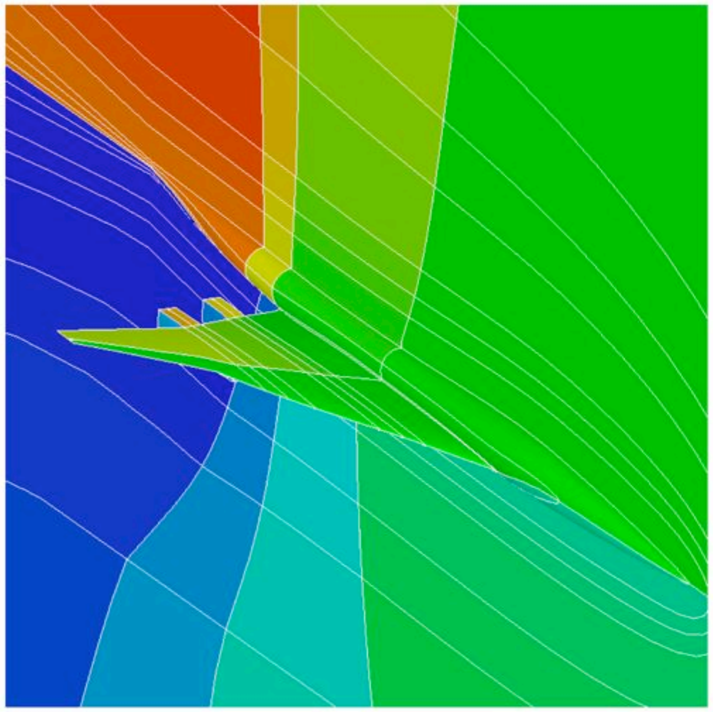
Source: Center for Integrated Turbulence Simulations (CITS), Stanford University

Parallel Scaling of SUmb on Red Storm

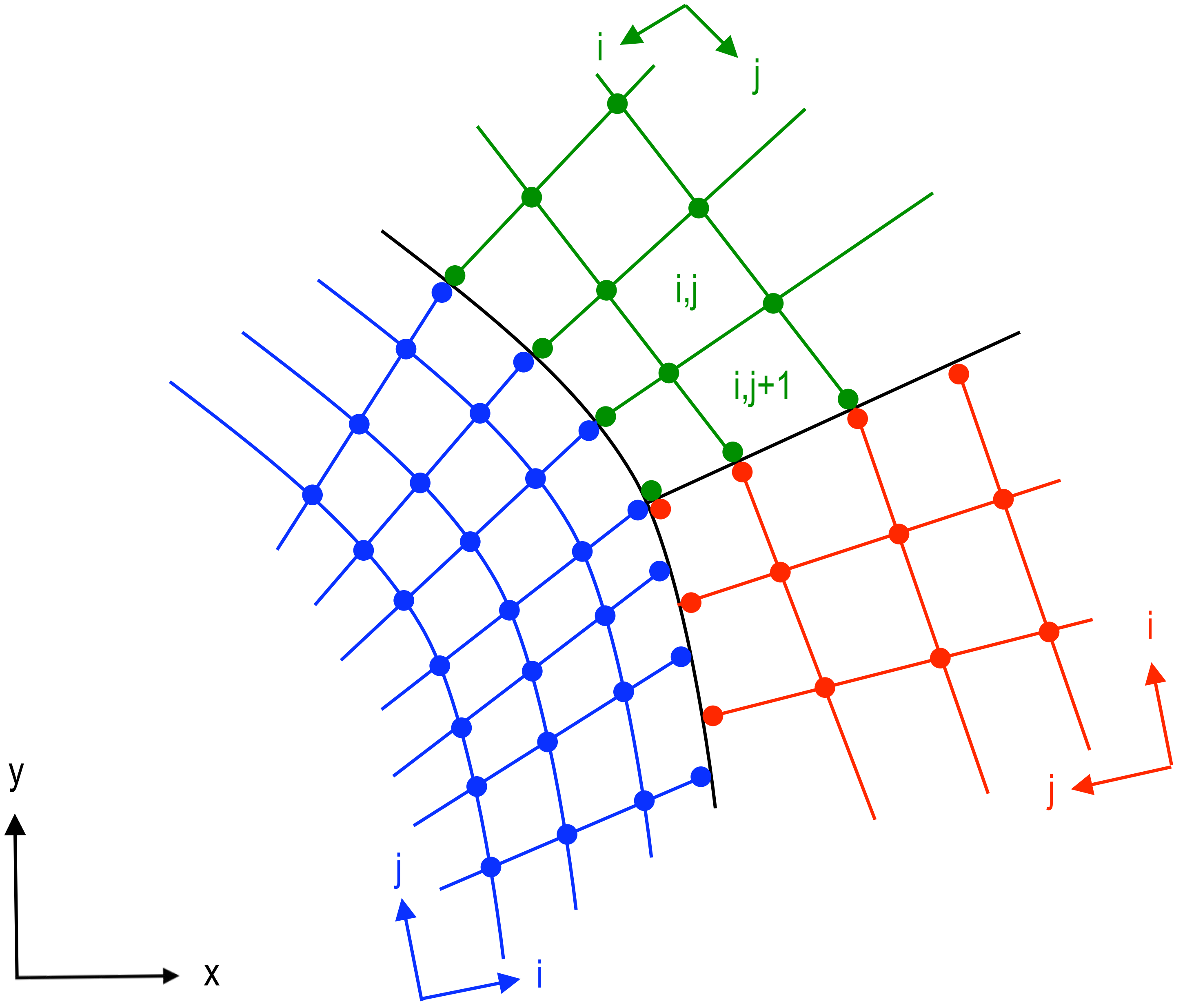


Navier-Stokes Stanford University Solver (NSSUS)

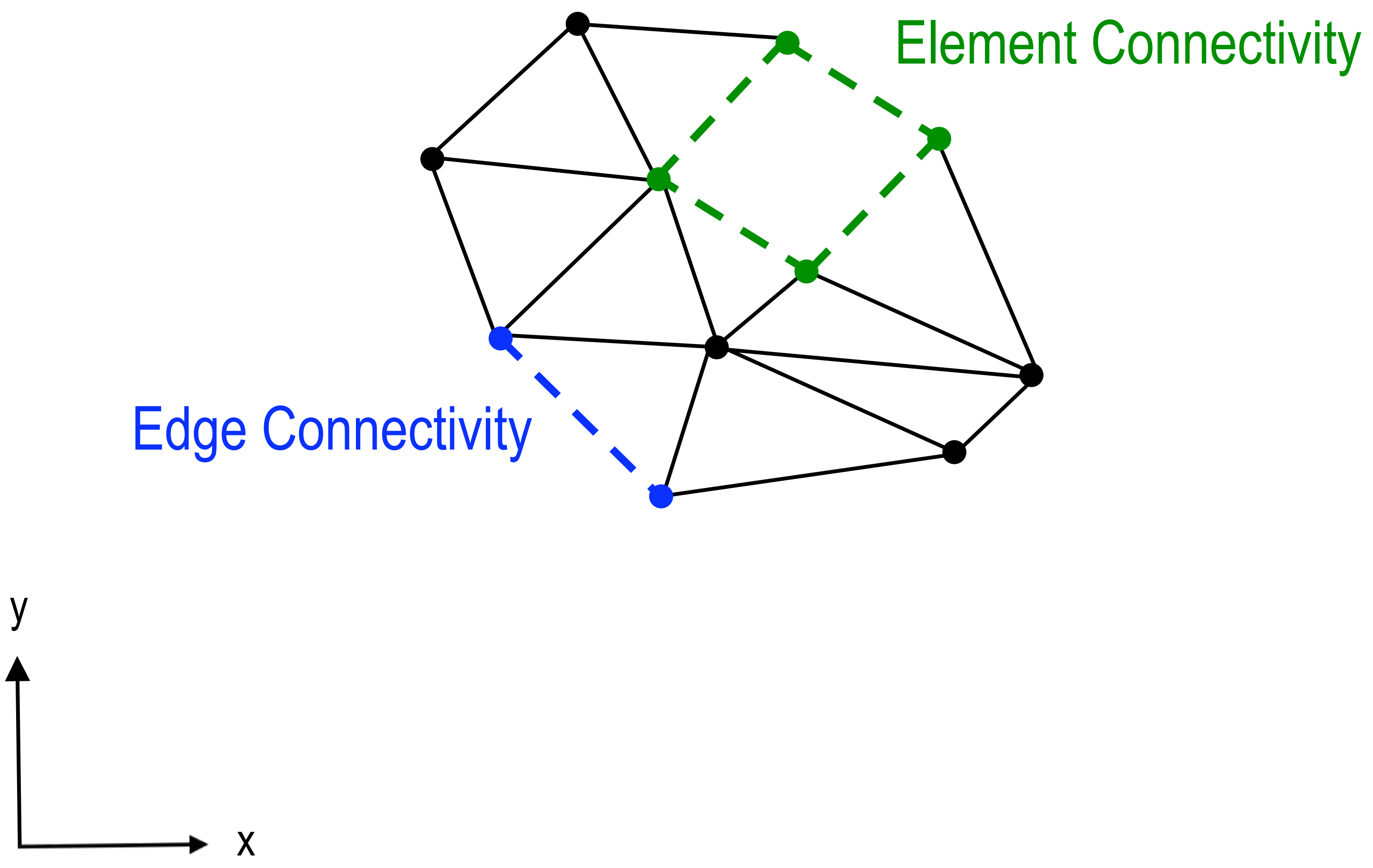
- Solves the Unsteady Reynolds-Averaged Navier-Stokes (URANS) equations via a finite difference formulation on massively parallel platforms
- Uses multi-block meshes which are a hybrid approach that uses blocks of structured meshes linked in an unstructured fashion



Multi-block structured mesh

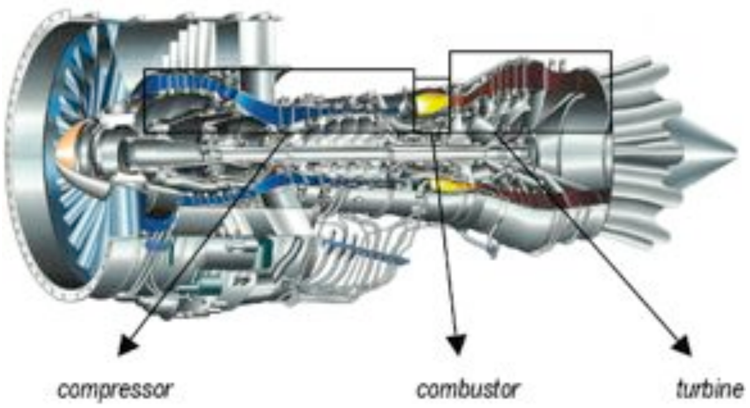


Unstructured mesh



NSSUS

- 1st to 6th order finite difference operators
- Boundary conditions are implemented using penalty terms based on Summation By Parts (SBP) / Simultaneous Approximation Terms (SAT) approach
- Geometric multigrid with support for irregular coarsening of meshes



Euler Equations

- GPU work has initially focused on the Euler Equations, which come about if the viscous and heat transfer terms are neglected
- System of non-linear PDEs:

$$\frac{\partial W}{\partial t}+\frac{\partial E}{\partial x}+\frac{\partial F}{\partial y}+\frac{\partial G}{\partial z}=0$$

$$W=\begin{Bmatrix}\rho\\ \rho u\\ \rho v\\ \rho w\\ \rho E\end{Bmatrix}\quad E=\begin{Bmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ \rho uw\\ \rho uH\end{Bmatrix}\quad F=\begin{Bmatrix}\rho v\\ \rho uv\\ \rho v^2+p\\ \rho vw\\ \rho vH\end{Bmatrix}\quad G=\begin{Bmatrix}\rho w\\ \rho uw\\ \rho vw\\ \rho w^2+p\\ \rho wH\end{Bmatrix}$$

$$H=E+\frac{p}{\rho}\qquad p=(\gamma-1)\rho\left[E-\frac{1}{2}(u^2+v^2+w^2)\right]$$

Euler Equations (cont.)

- For the finite difference discretization a coordinate transformation is made to yield:

$$\frac{\partial \overline{W}}{\partial t} + \frac{\partial \overline{E}}{\partial \xi} + \frac{\partial \overline{F}}{\partial \eta} + \frac{\partial \overline{G}}{\partial \zeta} = 0$$

$$\overline{W} = \frac{W}{J} \qquad E = \frac{1}{J}\left(\xi_x E + \xi_y F + \xi_z G\right) \qquad F = \frac{1}{J}\left(\eta_x E + \eta_y F + \eta_z G\right) \qquad G = \frac{1}{J}\left(\zeta_x E + \zeta_y F + \zeta_z G\right)$$

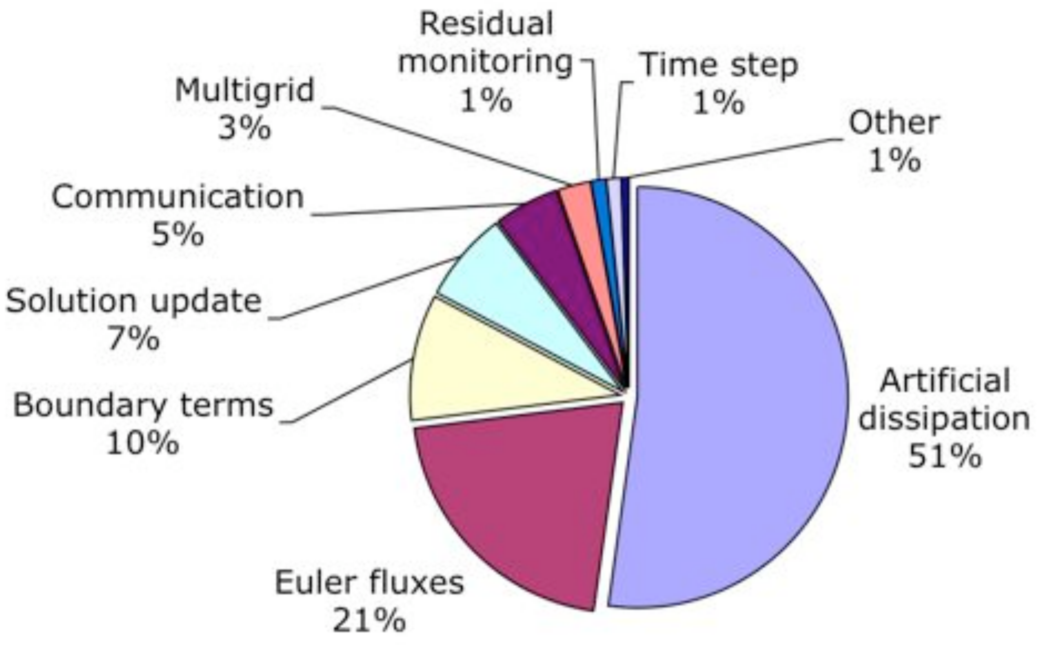
and *J* is the coordinate transformation Jacobian

- The spatial operators are discretized to generate a system of ODEs for *each and every node* in the mesh:

$$\frac{d}{dt}\left(\frac{W_{ijk}}{J_{ijk}}\right) + R_{ijk} = 0$$

Euler Equations (cont.)

- An explicit five-stage Runge Kutta scheme is used to advance the equations to a steady state
- Computing the residual *R* is the main computational cost and includes the inviscid Euler fluxes, the artificial dissipation for stability, and the penalty state and penalty terms for the boundary terms



Porting NSSUS to the GPU

- The idea is to ***port*** the existing CPU implementation of NSSUS to the GPU:
  - Approximately 200,000 lines of Fortran source code
  - Most of the lines of code (> 75%) are not performance critical or not amenable to the GPU
    - Preprocessing
    - MPI Communication
    - Postprocessing

Mapping Algorithms to the GPU

- Classification of the kernel types:
  - **Pointwise:** All of the memory accesses, possibly from many different arrays, are at the same location as the output. For example, computing the momentum from the density and velocity.
  - **Stencil:** Spatially local data is required, typically one or two neighbors in each direction. This data may or may not be contiguous in memory depending on the direction. Difference approximations and multigrid transfer operators are examples of this type of access.
  - **Unstructured gather:** Outputs gather from arbitrary locations in memory. This occurs when exchanging data used for computing block to block penalty terms.
  - **Reductions:** Outputs a single scalar by performing a commutative operation on an array of data. Used to monitor the convergence of the solution.

Simplified Inviscid Time Step Calculation

```
do k=1,kl
  do j=1,jl
    do i=1,il
      cc2         = gamma(i,j,k)*w(i,j,k,ip)/w(i,j,k,irho)
      cc2         = max(cc2,clim2)
      dt(i,j,k) = abs(w(i,j,k,ivx)*dxi  (i,j,k,1)  &
                +     w(i,j,k,ivy)*dxi  (i,j,k,2)  &
                +     w(i,j,k,ivz)*dxi  (i,j,k,3)) &
                + abs(w(i,j,k,ivx)*deta (i,j,k,1)  &
                +     w(i,j,k,ivy)*deta (i,j,k,2)  &
                +     w(i,j,k,ivz)*deta (i,j,k,3)) &
                + abs(w(i,j,k,ivx)*dzeta(i,j,k,1)  &
                +     w(i,j,k,ivy)*dzeta(i,j,k,2)  &
                +     w(i,j,k,ivz)*dzeta(i,j,k,3)) &
                + sqrt(cc2*(dxi  (i,j,k,1)**2      &
                +           dxi  (i,j,k,2)**2      &
                +           dxi  (i,j,k,3)**2))    &
                + sqrt(cc2*(deta (i,j,k,1)**2      &
                +           deta (i,j,k,2)**2      &
                +           deta (i,j,k,3)**2))    &
                + sqrt(cc2*(dzeta(i,j,k,1)**2      &
                +           dzeta(i,j,k,2)**2      &
                +           dzeta(i,j,k,3)**2))
      dt(i,j,k) = one/(dt(i,j,k)*detJinv(i,j,k))
    enddo
  enddo
enddo
```

Advective contribution

Acoustic contribution

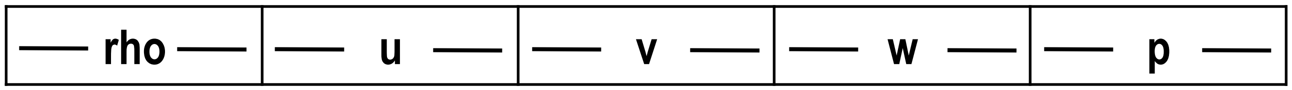
Time step per unit volume

Layout of data in CPU memory

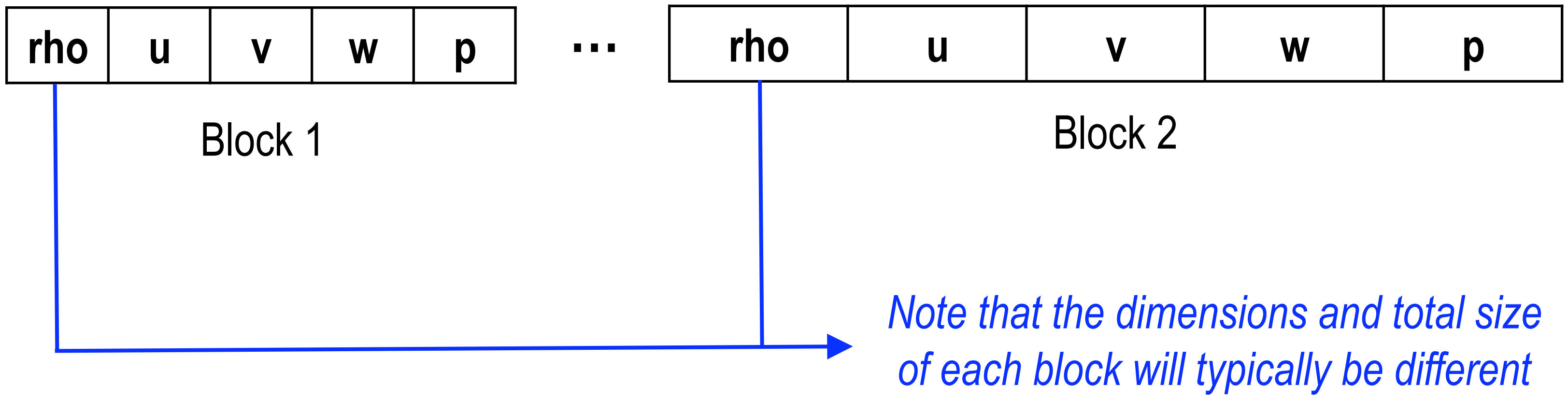
```
type blockType

...

! w(il,jl,kl,1:nw): The primitive variables.
!                   w(i,j,k,1:nwf) are the flow field
!                   variables, rho, u, v, w and p.
!                   w(i,j,k,nt1:nt2) turbulent
!                   variables; also the primitive
!                   variables are stored.

real(kind=realType), dimension(:,:,:,:), pointer :: w

...

end type blockType

...

! flowDoms(:,:,:): array of blocks. Dimensions are
!                  (nDom,nLevels,nTimeInstancesMax)

type(blockType), allocatable, dimension(:,:,:), target :: flowDoms
```

Layout of data in CPU memory (cont.)

Layout of a single block in linear memory:



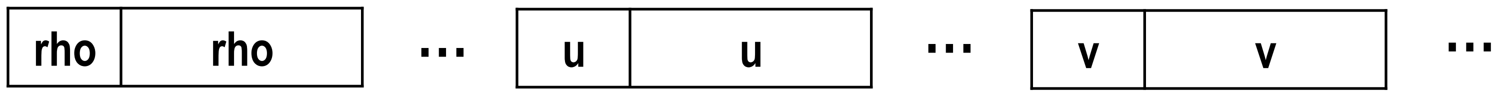
Layout of multiple blocks in linear memory:


rho
u
v
w
p
…
rho
u
v
w
p
Block 1
Block 2
Note that the dimensions and total size of each block will typically be different

Some observations about data layout

- Don't want to make changes in layout that require modifying preprocessing, MPI communication, or postprocessing
- Interop in Fortran 2003 gives us some flexibility
- It may be beneficial to use a different data layout on the GPU…
- But, we need to understand the data sizes and transfer characteristics to assess cost of rearranging data
- *Note that as an iterative solver we touch all data at every RK stage*

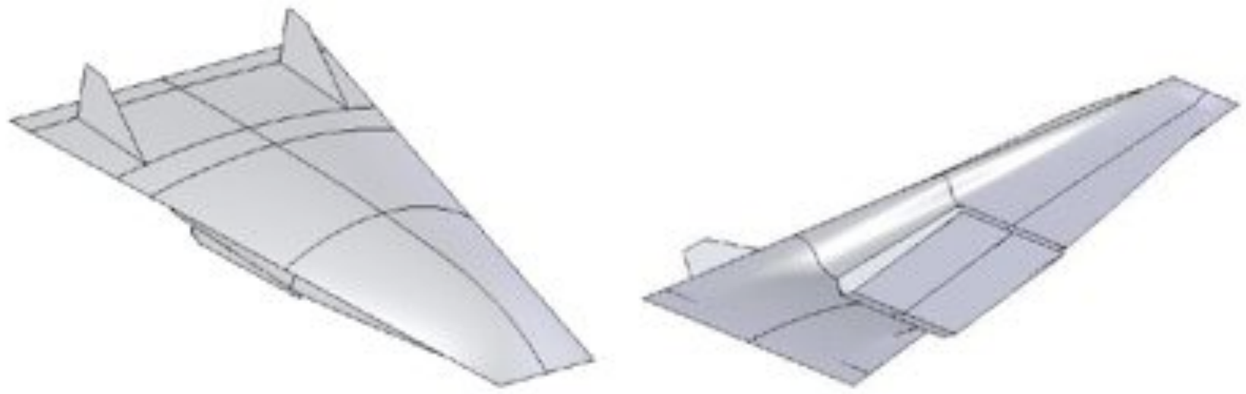
An appealing data layout option for the GPU



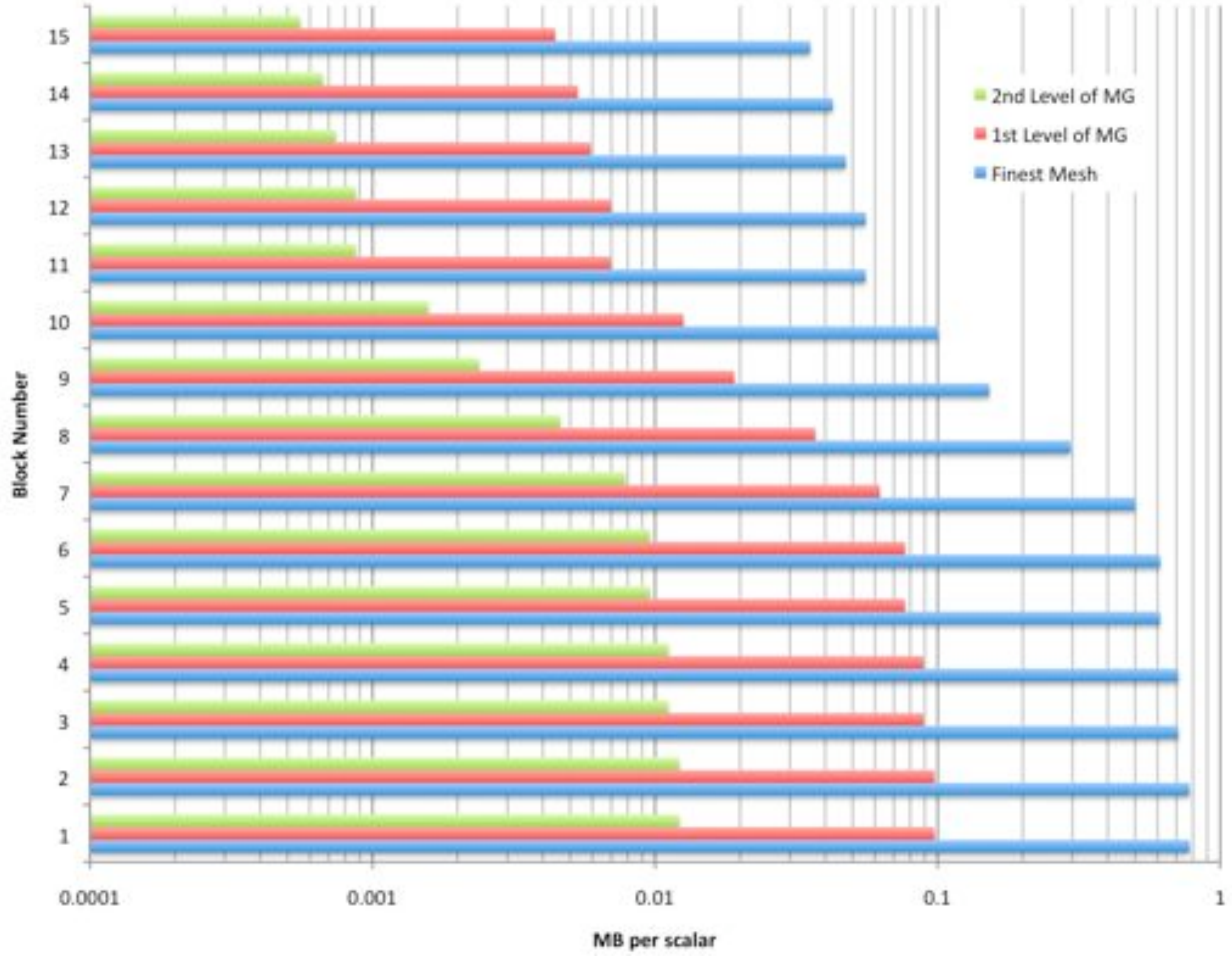
- Allows one kernel launch to easily process pointwise data for all blocks
- There is no way to get this data layout via Fortran interop so data will have to be rearranged every RK stage if using multiple GPUs
- Need to know more to assess the cost

Hypersonic vehicle mesh

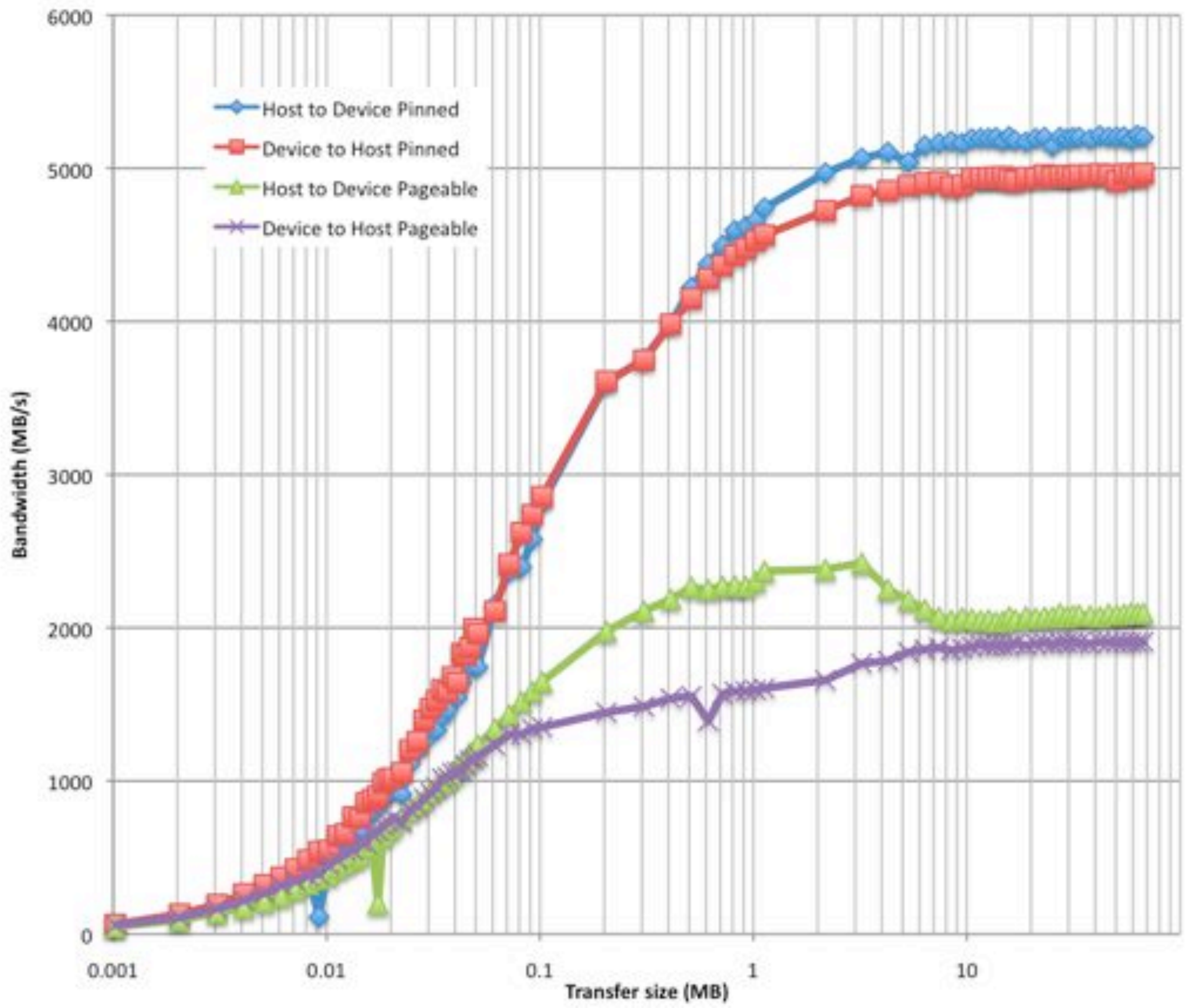
- 15 block meshes with approximately 720,000 and 1.5 million nodes
- For the latter, the average block size is approximately 100,000 nodes with a minimum of 10,000 and a maximum of 200,000 nodes



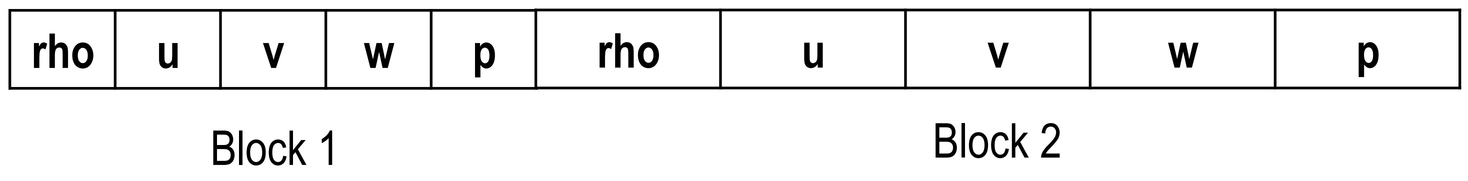
Data sizes for a typical mesh



Data transfer between CPU and GPU



A compromise for data layout on the GPU



- Can use the same data layout on the GPU and the CPU (via Fortran 2003 interop)
- One large data transfer between CPU and GPU is very efficient
- Will blocks have to be processed sequentially since the matching variables from multiple blocks aren’t contiguous?

Processing multiple blocks with a single kernel call

```
__constant__  float* cRhoPtrs[100];
__constant__  int    cBlockSizes[100];

__global__  void myKernel(int nBlocks)
{
  int idx, n, totalVertices = 0, blockSize;
  float *rho = NULL;
  idx = blockIdx.x*blockDim.x + threadIdx.x;
  for (n=0; n<nBlocks; n++)
  {
    blockSize = cBlockSizes[n];
    if (idx - totalVertices < blockSize)
    {
      idx -= totalVertices;  rho = cRhoPtrs[n];   break;
    }
    totalVertices += blockSize;
  }
  if (rho)
  {

    …

  }
}
```

Determines pointer to data for this thread

Make sure pointer is valid before proceeding with computation!

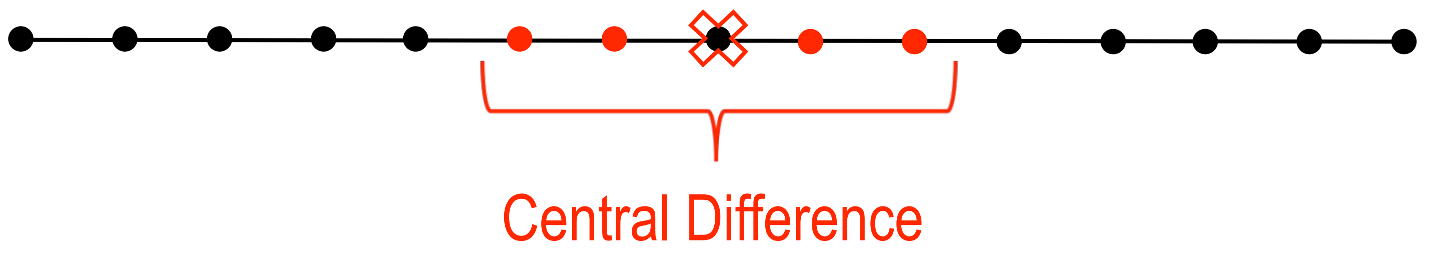
Summary of pointwise kernels

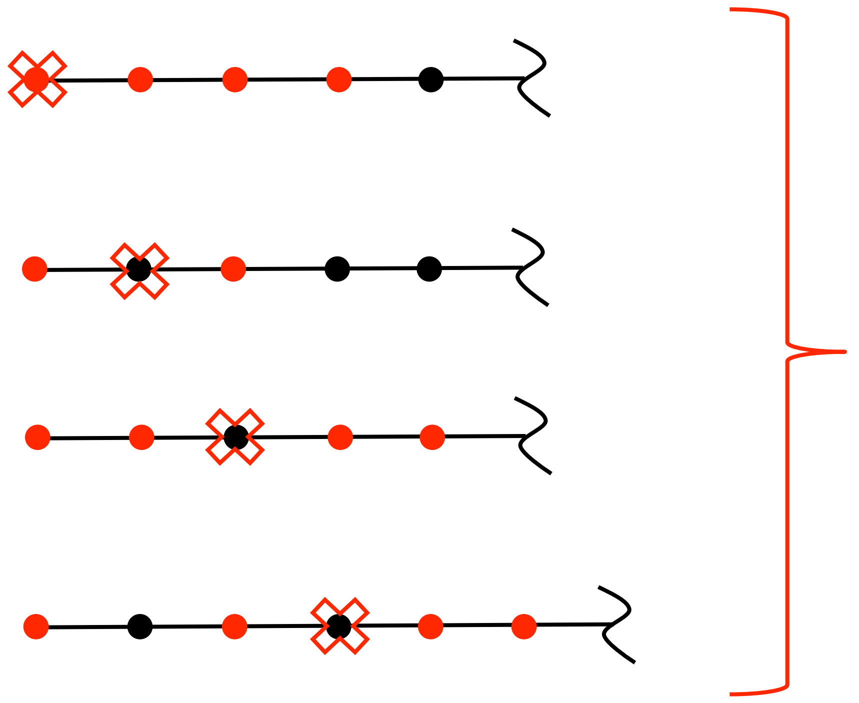
- Straightforward to write and fairly easy to get good performance relative to CPU
- Additional things can be done to handle address alignment for coalescing (not covered here)
- Register pressure becomes an issue:
  - Full Euler time step calculation uses 16 32-bit registers in single precision
  - Things get worse for the rest of the kernels!
  - Good arithmetic intensity is sometimes at odds with keeping occupancy high
  - May need to split big kernels into smaller kernels

Differencing operations

- Depending on block size and multigrid level different orders of differencing may be used in different directions, so each direction is performed independently
  - Too hard to implement 27 different combinations for 1st through 3rd order alone
  - Lose opportunity for on chip data reuse, but retain flexibility for handling “thin” blocks
- *The stencil and weights vary near the edge of blocks*

Stencils for third / fourth order differencing





Stencil and weights are different for each of the four vertices on the boundaries!!

Differencing on the GPU

- The operation can always be expressed as a dot product of weights and field values specified by a given stencil
- A single if/else-if/else branch is used to establish whether a vertex is in the left set of points, middle, or right set of points
- The different stencils and weights associated with the four boundary cases are read from auxiliary arrays accessed based on the index of the node in the differencing direction
- A dot product is computed using the weights and the field values accessed by the stencil

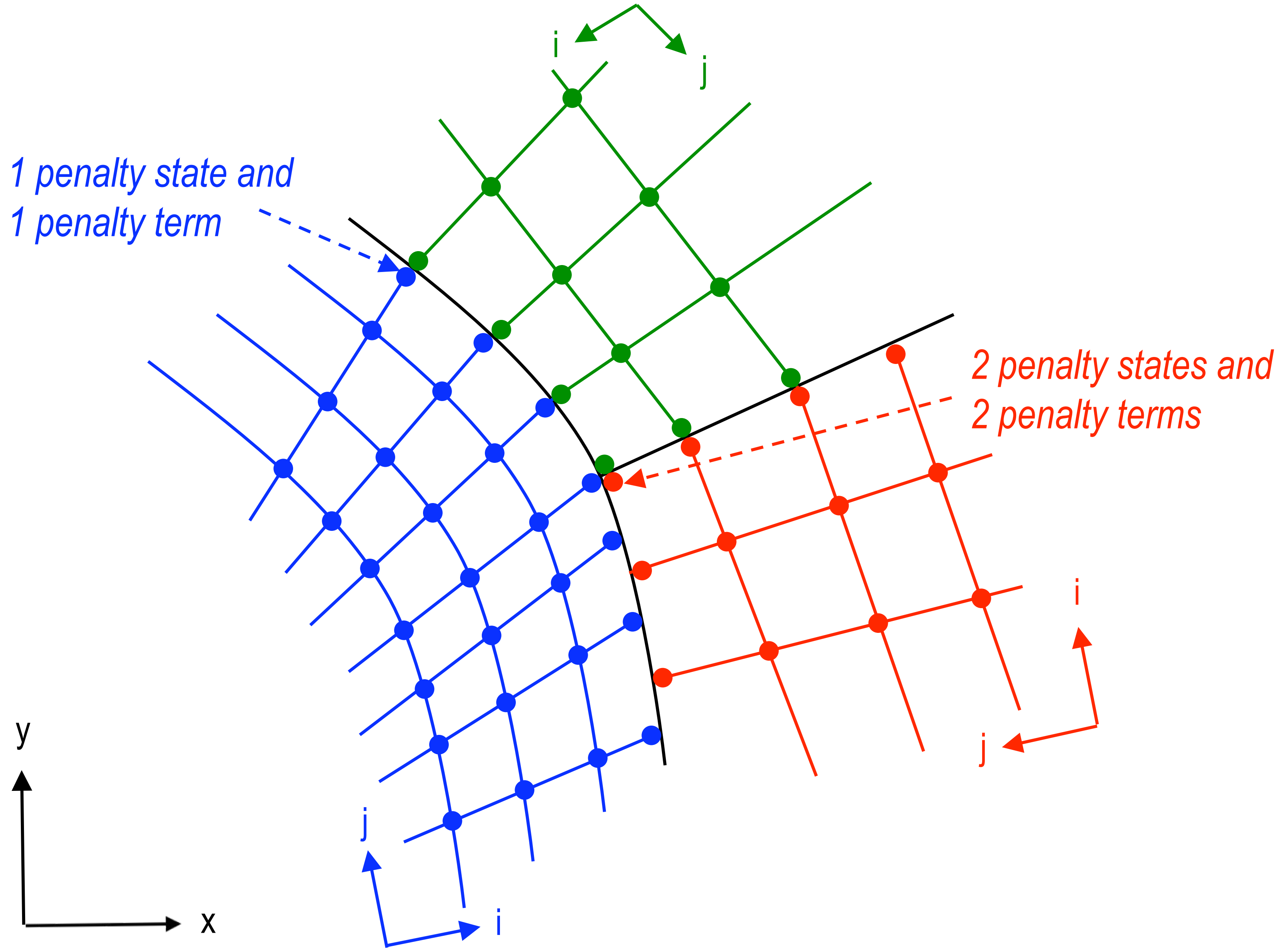
Performance of differencing operations

- Limited by memory bandwidth because number of operations is about the same as number of memory references
- The original CPU implementation does not pad the array dimensions so alignment for coalescing is an issue (can't treat this as 1D problem like pointwise calculations)
- 10 series architecture is more flexible in terms of coalescing, but extra indexing operations for checking alignment definitely helpful
- Textures can also be useful in this situation

Boundary terms

- Different logically rectangular regions on the face of a block, called sub-faces, may have different boundary conditions associated with them
- The CPU code loops over blocks and then sub-faces to compute the penalty state for physical boundary conditions and then the penalty terms, which weakly enforce the boundary conditions

2D example of penalty state / terms

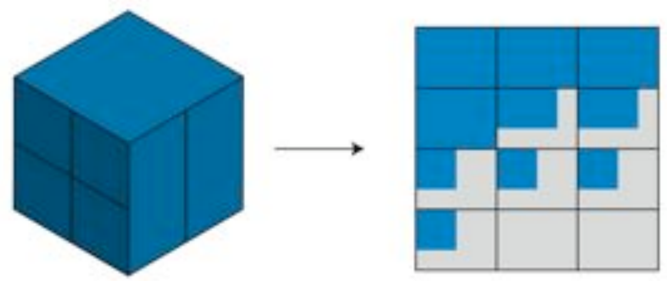


Can we inline the boundary terms?

- From a memory access point of view it would be convenient to "inline" the boundary terms with the volumetric computation
- But the solver supports about 30 different physical boundary conditions:
  - Some are simple (a few lines of code)
  - Some are very complicated (thousands of lines for sliding meshes in turbomachinery)
  - Most are somewhere in between
  - In any case, would greatly increase register pressure
- Not a realistic solution!

Boundary terms: copying surface data

- The CPU approach for boundary terms involves looping over blocks and then sub-faces
- Implemented on the GPU this has horrible performance because with so few values (100s) the computation time is dominated by the overhead to start kernels
- Index in the volume data of every node in each sub-face is pre-computed and used by a kernel to copy the appropriate data into an array containing surface data



- Note that a given surface node may appear multiple times where sub-faces meet at edges and corners!

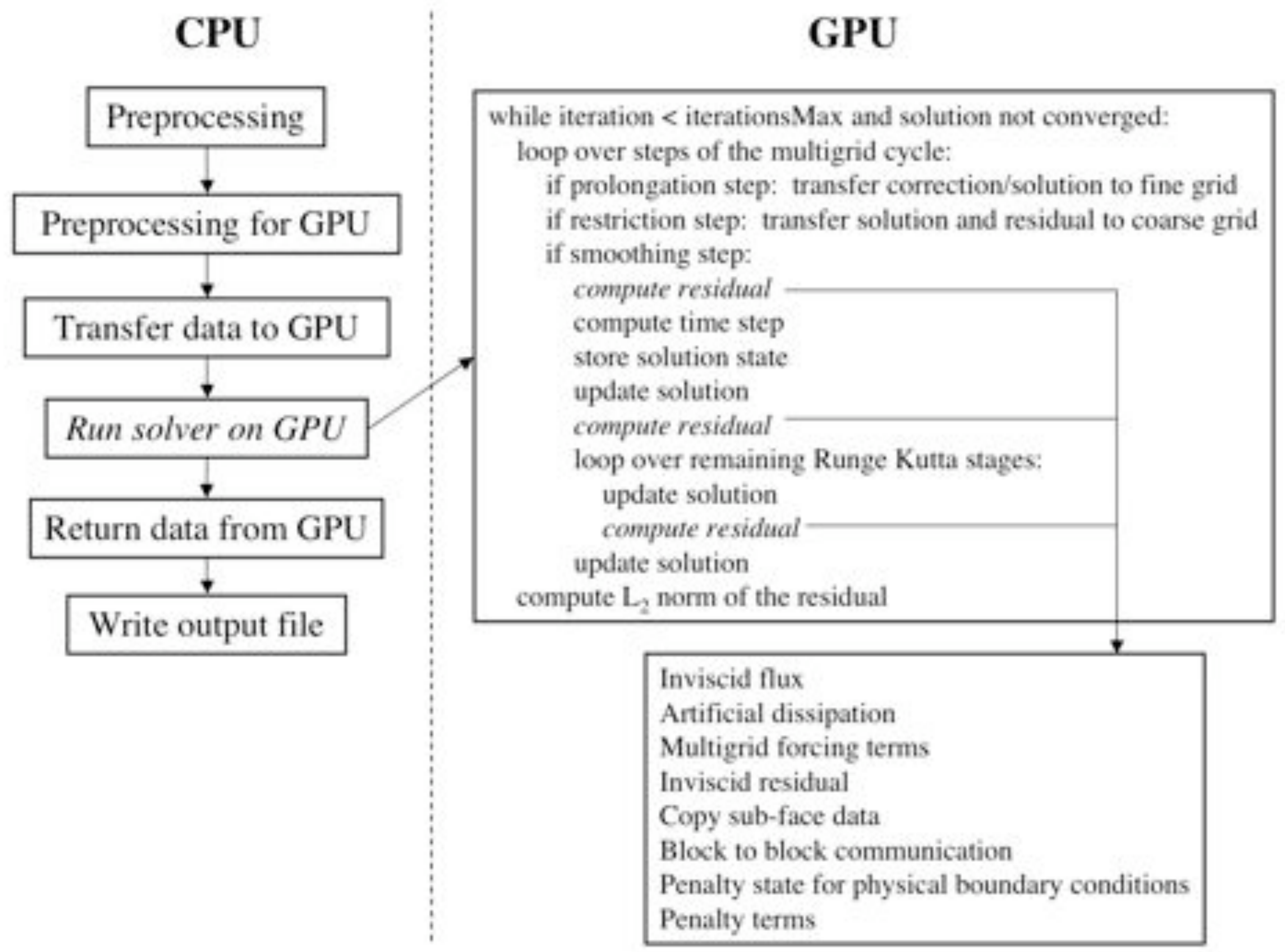
Computing penalty state/terms

- Groups of physical boundary conditions for the entire surface of a block are computed at once by using an auxiliary array that specifies the physical boundary condition for the nodes that are members of each sub-face
- All of the penalty terms are computed at once using the same auxiliary array
- The penalty terms are returned to the volumetric array by assigning a thread to each of the vertices in the volume array and:
  - Computing if the vertex is on the boundary (faster to compute than read in an auxiliary array)
  - If so, looping over the sub-faces and gathering the appropriate values if it is part of a given sub-face
  - Wastes a lot of threads but makes use of some coalescing

Performance on boundary terms

- Performance for the boundary terms (communicating or computing penalty state, and the penalty terms) is *the major limiting factor in overall application performance*
  - Typically see 1 to 5x speedup compared to CPU (depending on mesh topology)
- It may be better to simply treat the block surface boundaries in a completely unstructured way with a list of vertices and connectivity

Summary of code execution



```
while iteration < iterationsMax and solution not converged:
    loop over steps of the multigrid cycle:
        if prolongation step:  transfer correction/solution to fine grid
        if restriction step:  transfer solution and residual to coarse grid
        if smoothing step:
            *compute residual*
            compute time step
            store solution state
            update solution
            *compute residual*
            loop over remaining Runge Kutta stages:
                update solution
                *compute residual*
            update solution
    compute $L_2$ norm of the residual
```

Complexity of porting

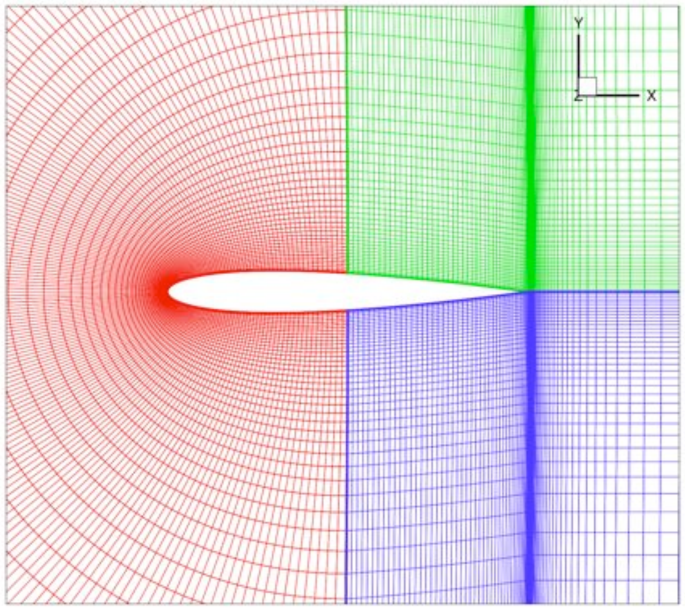
- The GPU port for the steady solution of the Euler equations and a few major boundary conditions (a subset of the full capabilities of NSSUS) involves approximately:
  - 4,000 lines of GPU code
  - 1,500 lines of new Fortran code
  - 5,000 lines of supporting C code

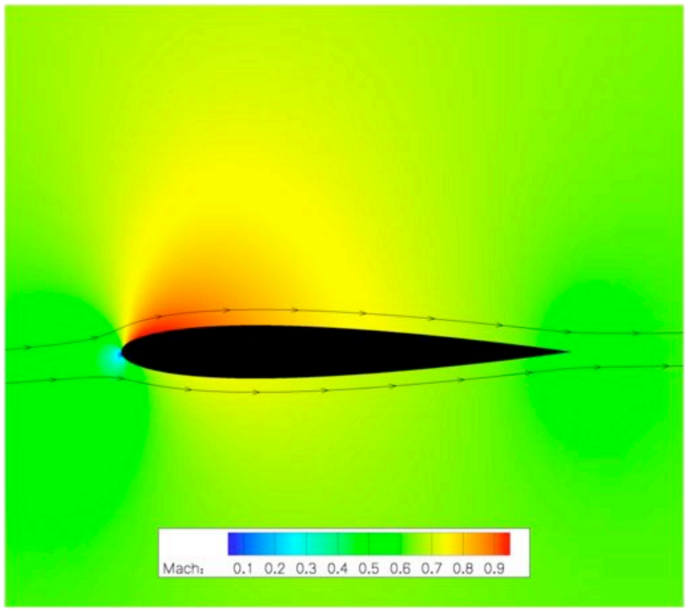
Hardware for performance results

- CPU:
  - Intel Xeon 5160 (3.0 GHz, 4MB L2 cache)
  - Intel Fortran compiler
- GPU:
  - NVIDIA Tesla C1060 (240 cores at 1.33 GHz, 4 GB of device memory)

Performance on real meshes

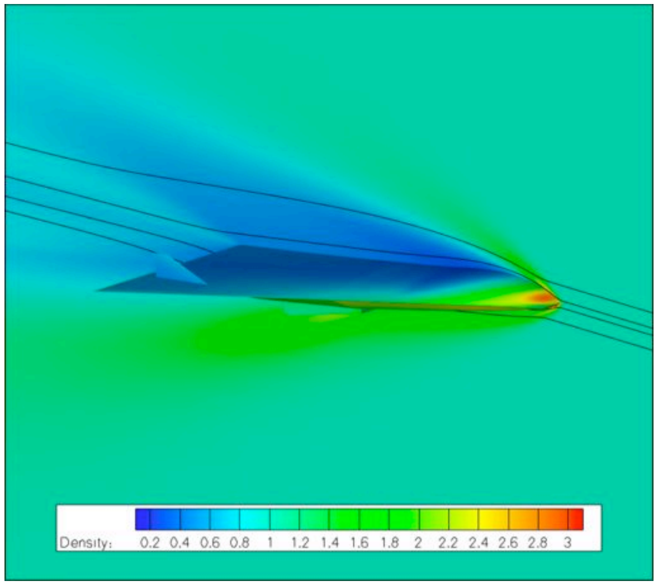
- NACA 0012 airfoil, Mach 0.63, alpha 2. degrees

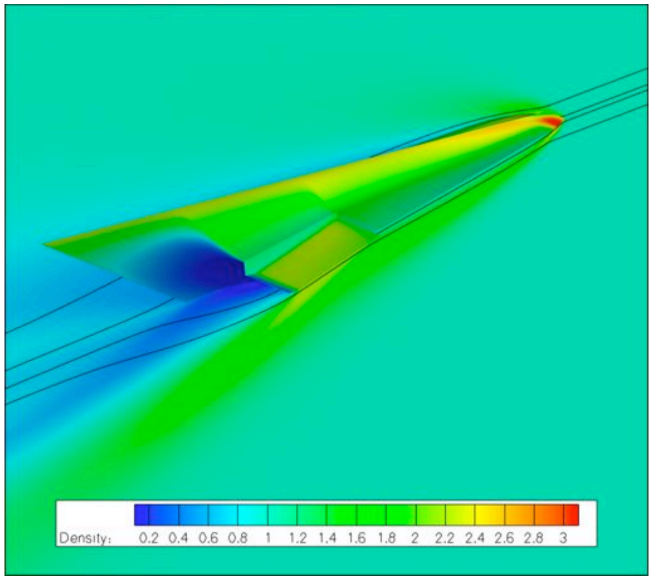


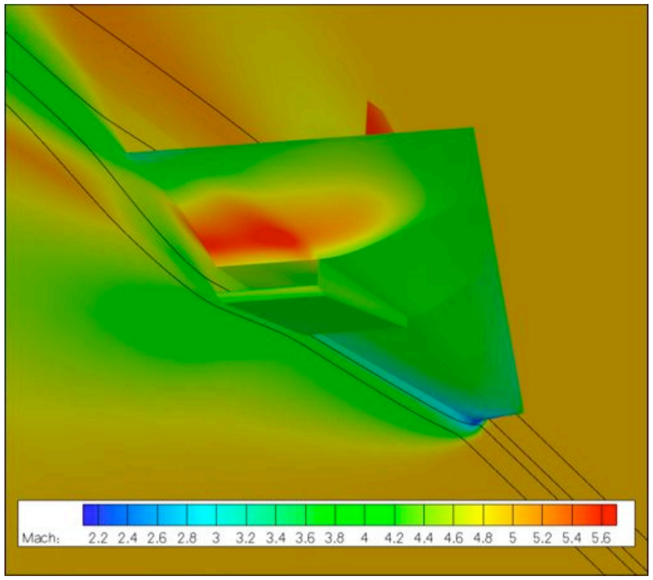


Order	Multigrid cycle	Speed-up
1st	single grid	24.5
3rd	single grid	18.6
1st	2V	19.3
3rd	2V	18.1

Hypersonic vehicle







Mesh	Multigrid cycle	Speed-up
720k	single grid	19.3
720k	2V	14.4
1.5M	single grid	26.9
1.5M	2V	20.2

Closing thoughts

- Data layout is very important to achieving good performance, but it's very expensive in terms of developer time to experiment with a large application
- Suggestions:
  - Profile your code and decide on an overall strategy for using the GPU:
    - Will you port the entire solver loop? Or just some key kernels?
    - Are you targeting a single GPU or multiple GPUs?
  - Categorize your kernels in terms of data access: pointwise, stencil, completely irregular, reduction, etc.
    - Experiment with one of each kind to see what works
  - Use your experiments to decide on an overall strategy and then implement the rest of your kernels