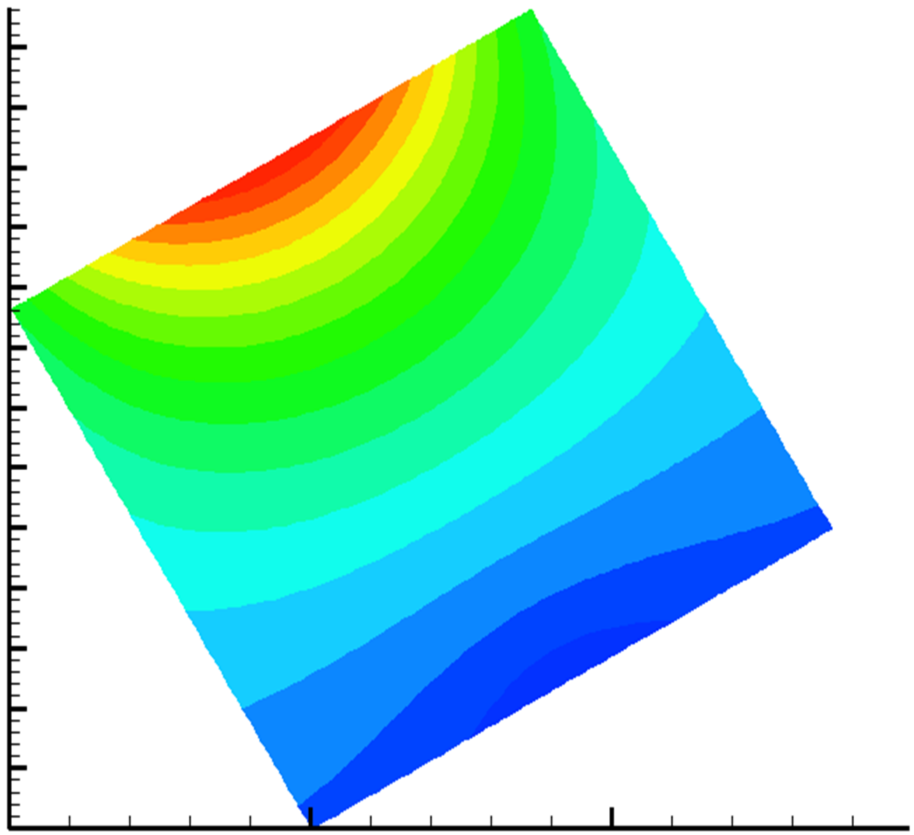
Lecture 17 Programming on GPUs

- **In this lecture, we will show examples of GPU programming using**
  - CUDA-C
  - CUDA-Fortran
- **We will discuss programming basics including**
  - Driver routines in CUDA-C and CUDA-Fortran
  - Kernel routines in CUDA-C and CUDA-Fortran
  - Memory mapping and copies
- **Results on single and multiple GPUs**

Example: Solving the Transient Heat Conduction Equation with GPUs
Jon Kemal (former UCD graduate student)



Hardware

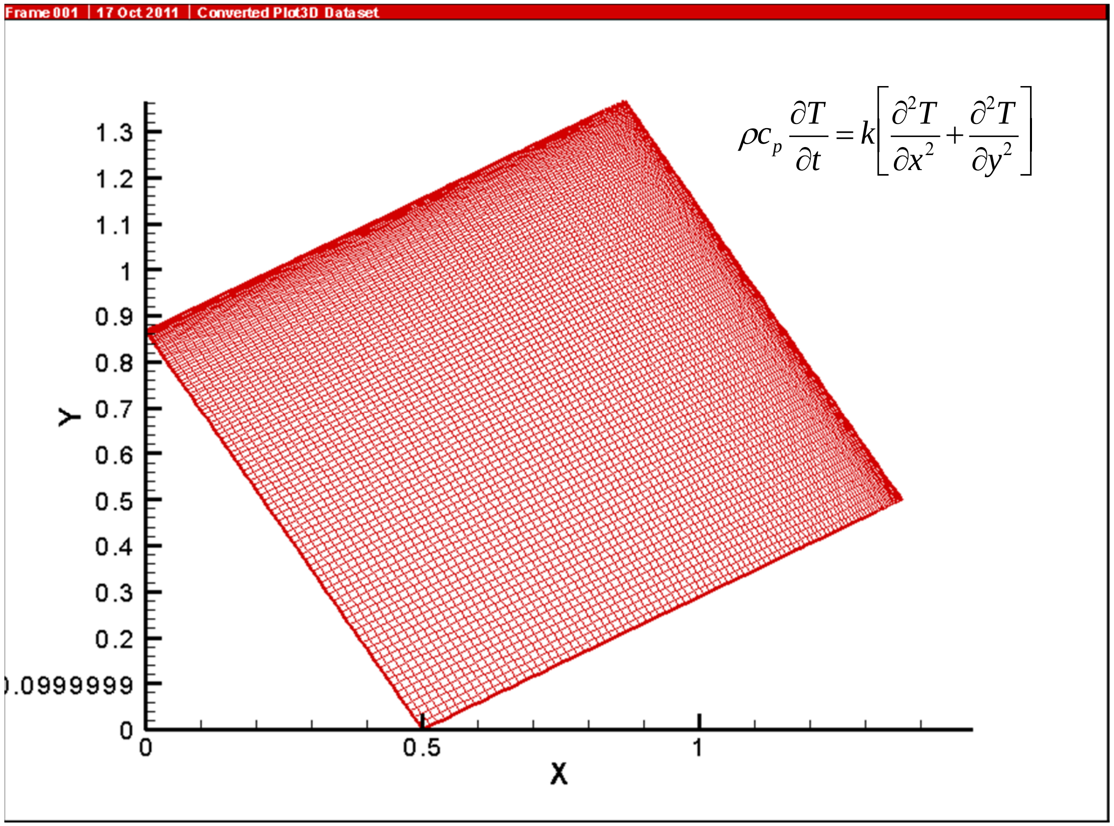
- CPU: Intel Xeon X5667

4 cores, 3.47 Ghz

- GPU: 4 NVIDIA Tesla C2050

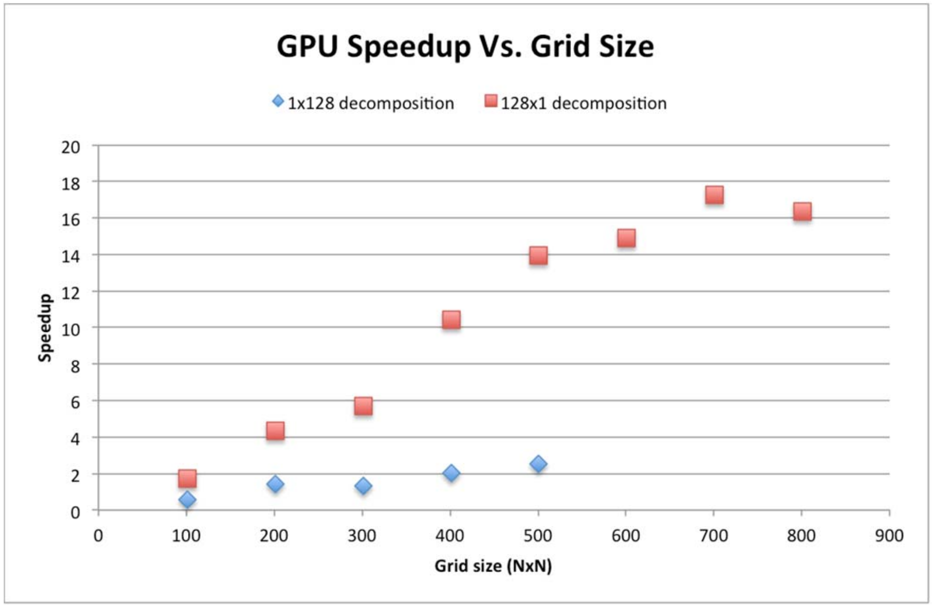
Fermi, 448 ALUs. 14 SM cores.

Single GPU

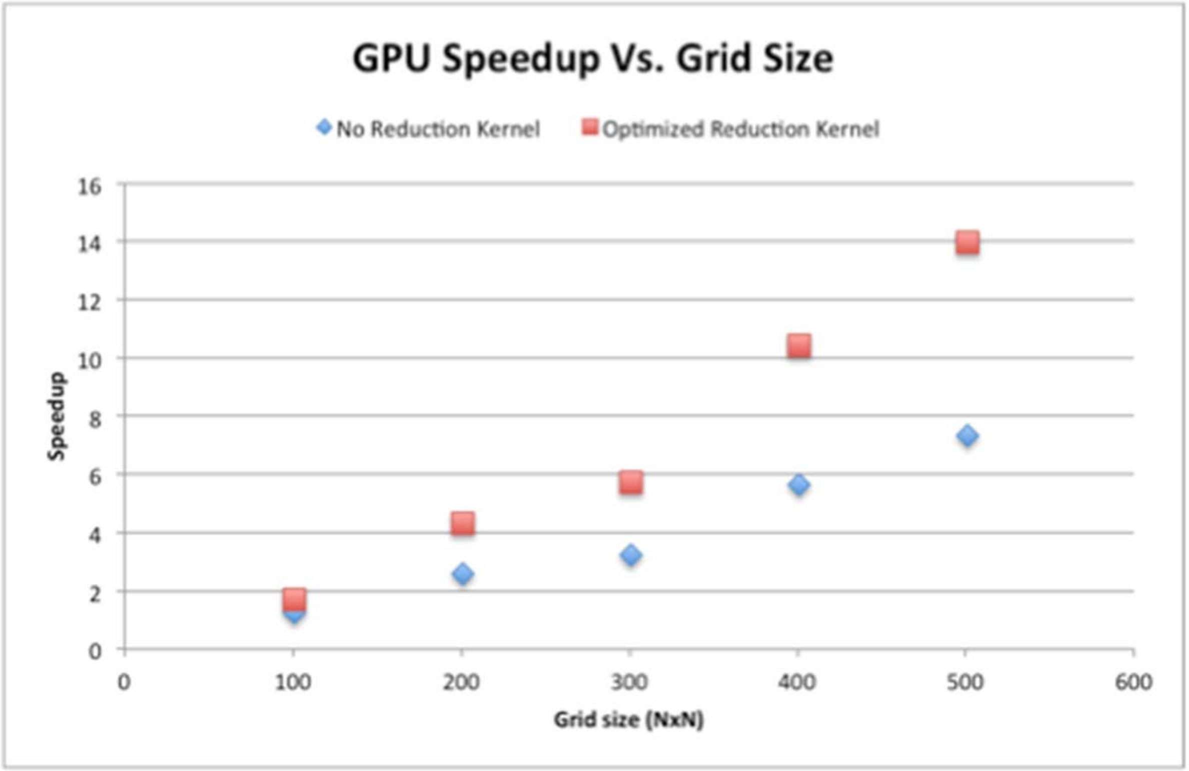


$$\rho c_p \frac{\partial T}{\partial t} = k\left[\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right]$$

- Non-uniform rectangular mesh (101x101 shown)
- Solved using *finite volume* iterative implicit scheme
- Decomposing domain into thread blocks crucial to performance
- Mixed language programming: setup in Fortran, solver in CUDA C.

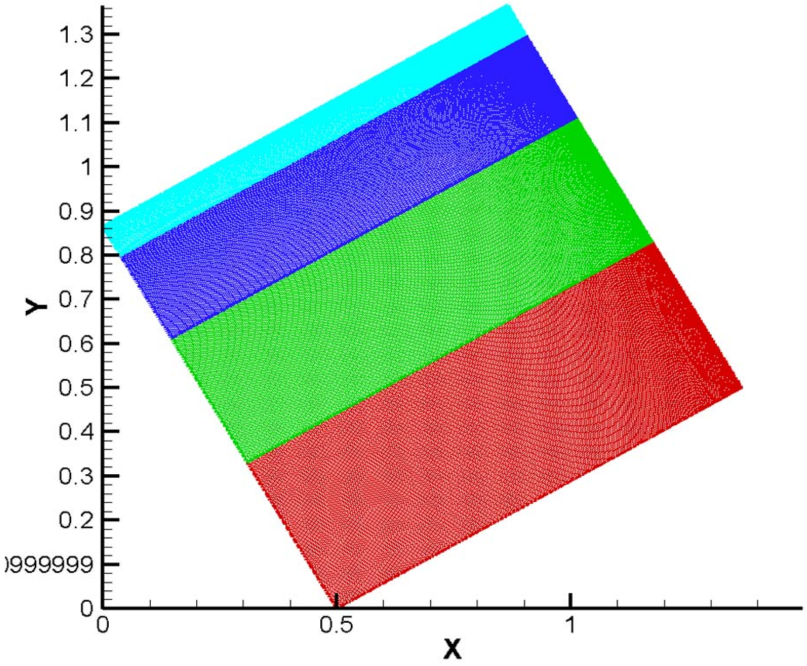


- Best performance with 4 warps per thread block, but how to decompose domain?
- Best results obtained for 128x1 blocks or 32x4 blocks – good memory coalescing.
- Worst results for 1x128 blocks.
- Memory accessed per half-warp. With 128x1 decomposition, a lot of data aligned in linear memory.
- Fortran stores data in column major order!



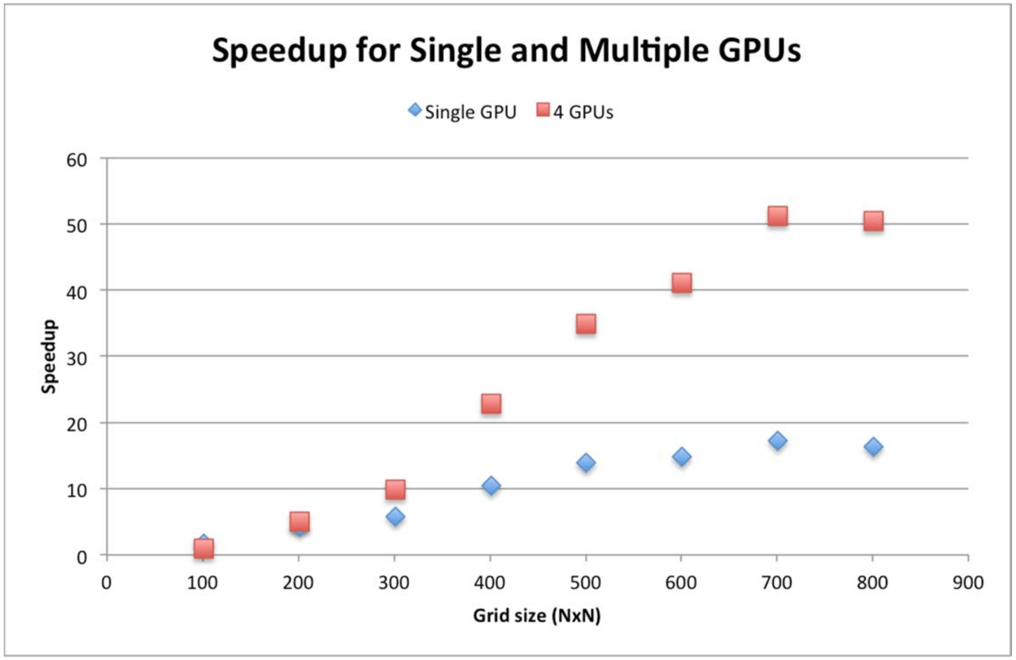
- GPUs great for reduction trees
- Great for finding maximum, minimum, sum, etc.
- MUCH faster than copying back to host!

Using 4 GPUs



- Divide domain into 4 sections (domain blocks)
- Each GPU solves different portion of domain
- Using 1 CPU thread (and core) to manage all 4 GPUs.
- Need to use “halo” or “ghost nodes” to store boundary information for each block
- Need to pass boundary information between GPUs during each iteration. This means additional overhead!

Results!



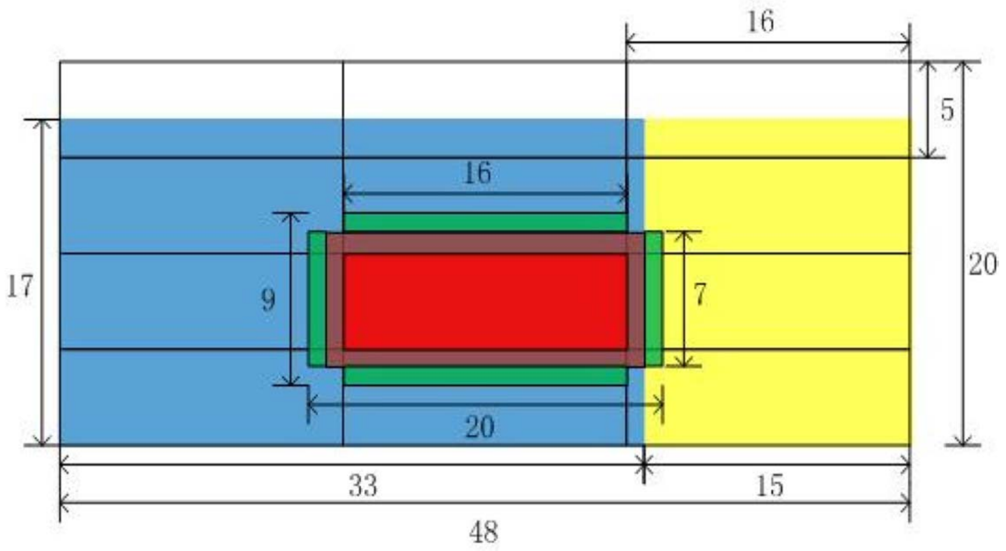
- Multiple GPUs provide additional speedup for large grids, take longer for small grids!
- Maximum speedup, compared to single CPU, 17 for single GPU and 50 for 4 GPUs
- Multiple GPUs thus provide additional speedup factor of 2.9, even though there are 4 GPUs. This is due to additional overhead of message passing etc.

Software Development Using GPUs

- **Example Application of GPUs to Computational Fluid Dynamics (CFD)**
  - Similar numerics to those used in your heat conduction projects
- **Determine optimal performance gains using 2D Euler code constructed specifically for GPUs**
- **Determine “typical” performance gains for existing “general purpose” CFD codes**
  - Use 2D multi-block, structured-grid Navier-Stokes code
    - Arbitrary block connectivity and orientation
    - Several turbulence modeling strategies including 2-equation RANS, DES, and hybrid RANS/LES

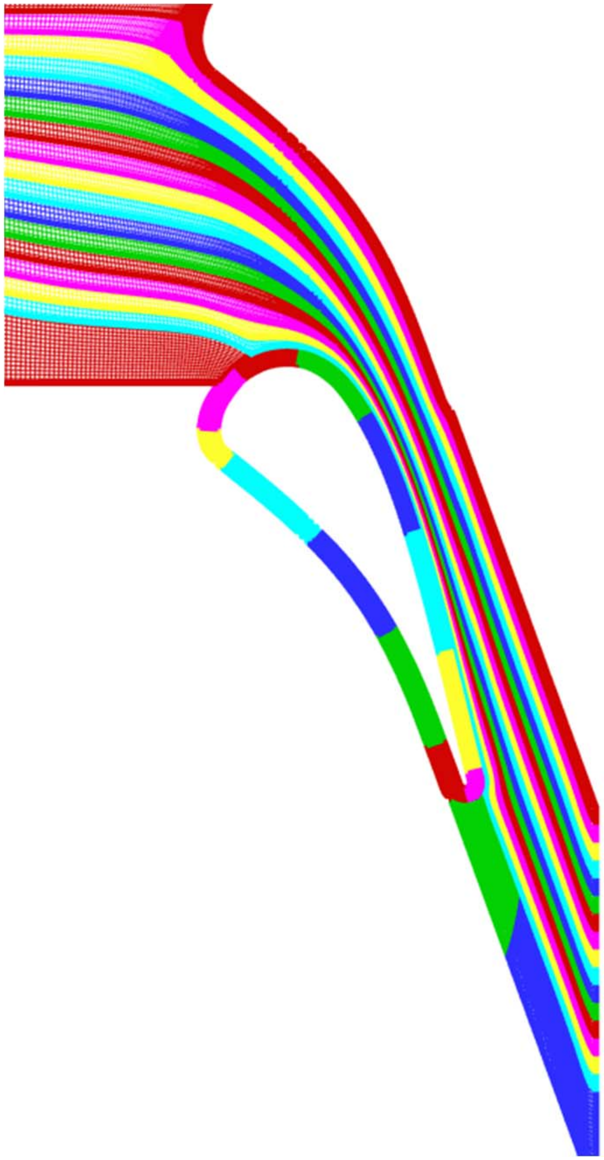
Example-1: Special-Purpose Euler GPU Solver

- **Current GPU implementation uses**
  - GPU data arrays are 1D stripes of linear memory, pointers used to re-construct 2D structure.
  - Kernel Launches are 2D.
  - 2 layers of ghost nodes/cells



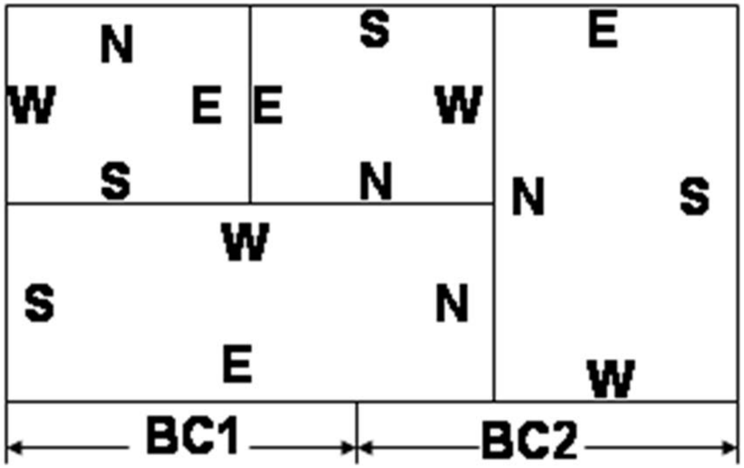
Example-2: General-Purpose Navier-Stokes Approach

- **Explicit Lax-Wendroff (Ni) Finite-Volume**
- **Multiple-Grid Acceleration (Steady and Inner Iteration Unsteady)**
- **Dual Time-Step Scheme (Unsteady)**
- **Blended 2nd and 4th-Difference Added Dissipation**
  - New approach for decaying dissipation in viscous regions



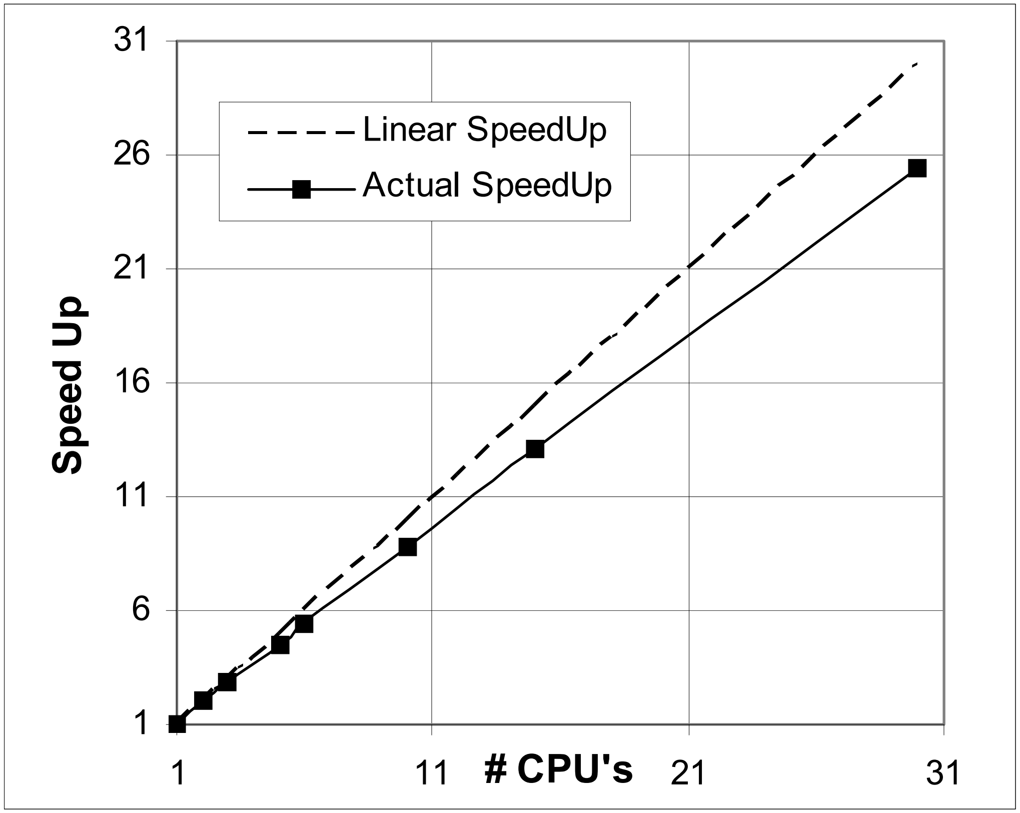
Data Structure

- **Multi-Block Structured and Overlaid Grids**
  - Assembled using unstructured data structure
- **Arbitrary Orientation**
- **Arbitrary Number of Sub-Faces Per Edge**
- **Automated Decomposition During Grid Generation to Create Equal-Size, Equal Multi-gridable Blocks**
- **Automated Connectivity Data Structure Generated During Grid Generation**



CPU Parallel Scalability

- **MPI (Message Passing Interface) Used to Pass Information Between CPU Processors**
- **85% Efficiency on 2D Grid of ~35,000 points**
- **Turn-around of time-averaged URANS or DES in under 1 hour using 30 CPUs (2.6 GHz Athlons)**



Parallelization Strategy

- **A combined shared/distributed parallelization strategy is used in this investigation where**
  - Computations within a block are performed using shared-memory parallelization on the GPUs
  - Computations across blocks and low-volume computations such as boundary conditions are performed using distributed memory parallelization on the CPUs

CUDA-C, CUDA-Fortran, or OpenCL?

- **GPU programming can be performed using CUDA-C, CUDA-Fortran, or OpenCL**
  - CUDA-C is the fundamental language that can be linked to C- or Fortran-codes for NVIDIA GPUs
    - The advantage of CUDA-C is that it is compiler independent
    - The disadvantage of CUDA-C is that C-programming structure may not be familiar to developers using Fortran
  - CUDA-Fortran is only available with the Portland Group compiler (PGI) for NVIDIA GPUs
    - (see http://www.pgroup.com/resources/cudafortran.htm)
  - OpenCL is a general-purpose language for any GPU
    - The advantage is that code is portable to any GPU platform
    - However, the programmer can not take full advantage of the architectural abilities of a particular GPU. In other words, it’s slower.

CUDA Integration with Solvers

Algorithm Kernels

Main routine calls kernel

```
if(gpu==1) then
    call gpu_function(...)
else
    call function
endif
```

Algorithm kernels are done on the GPU

Data is mapped from CPU to GPU during call to Kernel

Multi-block Communication (MPI) performed on CPU (host)

In some cases, only block edges must be copied

```
if(gpu==1) then
    call copy_corners_host(buffer_d,buffer)
    OR call copy_to_host(buffer_d, buffer)
    call blkbnd
    call copy_corners_gpu(…)
    OR call copy_to_gpu(buffer_d, buffer)
else
    call blkbnd
endif
```

Communication between blocks is done by the CPU

GPU Subroutines Using CUDA-C

Example of Embarrassingly Parallel Routine – Laminar Viscosity

CPU Code

```
subroutine lamvis

do j = 1,jmax(n)
  do i = 1,imax(n)
    tott = Long Calculation (no room)
   xmu(1,i,j,n) = xmufree*(tott**1.5d0)/(tott + suthcnst)
  enddo
enddo
```

GPU C-Code

Subroutine Kernel (execution of thread)

```
__global__ void lamvis_kernel( ... )
{
 int i = blockDim.x * blockIdx.x + threadIdx.x;
 int j = blockDim.y * blockIdx.y + threadIdx.y;
 double tott;

if(i<imax && j<jmax)
  {
     tott = (gama–1.0)*(u[j+2][i+2][5] – 0.5*(pow(u[j+2][i+2][6],2) +
pow(u[j+2][i+2][7],2) + s1*pow(w[j+2][i+2][1],2)) +
0.5*pow(omega*(y[j+2][i+2]*s1+r[j+2][i+2]*s2),2))/rttovfree/gama;
    xmu[j+2][i+2][0]=xmufree*pow(tott,1.5)/(tott + suthcnst);  }
}
```

Pointers to variables on GPU

Kernel executed for every thread

Main routine (sets up thread execution)

```
extern "C" void gpu_lamvis_( ... )
{
  dim3 threadsPerBlock(block_rows,block_cols);
  LOOP OVER FLUID BLOCKS
  {
    dim3 numBlocks(nblocks_i[i],nblocks_j[i]);
    lamvis_kernel<<<numBlocks, threadsPerBlock>>>( ... );
  }
}
```

Pointers to variables on CPU are input arguments

3D array structure re-constructed using pointers.

Variables are mapped to GPU grids

GPU Subroutines Using CUDA-Fortran

Example of Embarrassingly Parallel Routine – Laminar Viscosity Main routine

Main routine (sets up thread execution)

```
subroutine gpu_lamvis
     threadBlock = dim3(BLKX,4,1)
     grid = dim3((imax(1)+threadBlock%x-1) / threadBlock%x,
(jmax(1)+threadBlock%y-1) / threadBlock%y, 1)

     call gpu_lamvis_kernel<<<grid,threadBlock>>>

end subroutine gpu_lamvis
```

GPU Subroutines Using CUDA-Fortran

Example of Embarrassingly Parallel Routine – Laminar Viscosity Kernel

CPU Code

```
subroutine lamvis

do j = 1,jmax(n)
  do i = 1,imax(n)
    tott = (gama – 1.0)*(u6 – 0.5*(u7**2 u8**2)
    xmu(1,i,j,n) = xmufree*(tott**1.5d0)/(tott + suthcnst)
  enddo
enddo
```

GPU Fortran-Code

Subroutine Kernel (execution of thread)

```
attributes(global) subroutine gpu_lamvis_kernel
  integer :: i,j,n
  real*8 :: tott
  i = threadIdx%x + blockDim%x*(blockIdx%x – 1)
  j = threadIdx%y + blockDim%y*(blockIdx%y – 1)
  n = 1
  if(i<=imax_d .and. j<=jmax_d) then
    if(gamma_d>0.0d0) then
      tott = (gama_d – 1.d0)*(_u_d(6,i,j,n)
&            – 0.5d0*(_u_d(7,i,j,n)**2 + _u_d(8,i,j,n)**2))
&              /rttovfree_d/gama_d
      xmu_d(1,i,j,n) = xmufree_d*(tott**1.5d0)/(tott + suthcnst_d)
    else
      xmu_d(1,i,j,n) = xmufree_d
    endif
  endif
end subroutine gpu_lamvis_kernel
```

Pointers to variables on GPU are mapped via grid and threadBlock (see next slide)

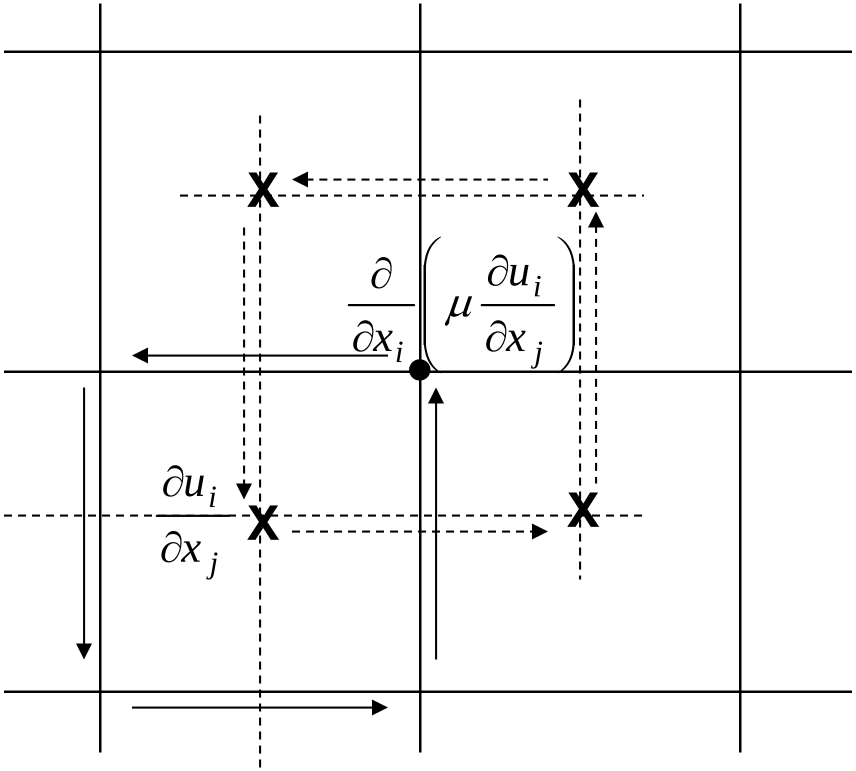
Equivalent CPU indices determined

Kernel executed for every thread

GPU Kernel for Calculating Second Derivatives Using CUDA-C

Example of More Complex Kernel – Viscous Stresses

- **Viscous forces in a flow solver are determined from two-step integration to find**
  - Stresses
  - Viscous forces
- **These are similar to second derivatives in your heat conduction solver**


X
X
$\frac{\partial}{\partial x_i}\left(\mu \frac{\partial u_i}{\partial x_j}\right)$
$\frac{\partial u_i}{\partial x_j}$
X
X

Second Derivative GPU Subroutines Using CUDA-C

Example of More Complex Kernel – Viscous Stresses (Main routine)

```
/* Compute execution configuration */
  dim3 dimBlock1(16, 4, 1);
  dim3 dimGrid1 ((imax+dimBlock1.x-2)/(dimBlock1.x-1), (jmax+dimBlock1.y-
     2)/(dimBlock1.y-1));

  /* Execute the kernel */
```

Main routine (sets up thread execution)

```
   stress_kernel_1<<<dimGrid1, dimBlock1>>>(u, w, vol, x, y, r, sth, dudx, dudy, dvdx,
     dvdy, dhdx, dhdy, dwdx, dwdy, s1, s2, imax, jmax);
```

Variables are mapped to GPU grids

```
   CUT_CHECK_ERROR("Kernel execution failed");
   stress_kernel_2<<<dimGrid1, dimBlock1>>>(tau, taut, xmu, u, w, y, dudx, dudy, dvdx,
     dvdy, dhdx, dhdy, dwdx, dwdy, pran, s1, s2, imax, jmax);
```

Variables are mapped to GPU grids

```
   CUT_CHECK_ERROR("Kernel execution failed");
   stress_kernel_3<<<dimGrid1, dimBlock1>>>(shr, dudx, dudy, dvdx, dwdx, dwdy, s1, s2,
     imax, jmax);
```

Variables are mapped to GPU grids

```
   CUT_CHECK_ERROR("Kernel execution failed");

  return;
```

Second Derivative GPU Subroutines Using CUDA-C

Example of More Complex Kernel – Viscous Stresses (Kernel-1)

```
__global__ void stress_kernel_1(float* u, float* w, float* vol, float* x, float* y, float* r, float*
     sth, float* dudx, \
float* dudy, float* dvdx, float* dvdy, float* dhdx, float* dhdy, float* dwdx, float* dwdy, float
     s1, float s2, int imax, int jmax)
{
     unsigned int lx = threadIdx.x;
     unsigned int ly = threadIdx.y;
//   unsigned int  n = threadIdx.z;
     unsigned int gx = lx + (blockDim.x - 1)*blockIdx.x;
     unsigned int gy = ly + (blockDim.y - 1)*blockIdx.y;
     unsigned int index_c  = (gx) + (gy)*(imax-1);
     unsigned int index_cg = (gx+1) + (gy+1)*(imax+1);
     unsigned int index_ng = (gx+2) + (gy+2)*(imax+4);
//   unsigned int offset = (imax+4)*(jmax+4);

     __shared__ float    u_sh[4][4][16];
     __shared__ float    xyrs[4][4][16];
     __shared__ float   ars_we[3][4][16];
     __shared__ float   ars_ns[3][4][16];
     __shared__ float  side_we[2][4][16];
     __shared__ float  side_ns[2][4][16];
//
```

Local GPU indicies determined

Subroutine Kernel-1 (execution of threads)

Equivalent Global CPU indicies determined

Allocating shared memory on GPU

Second Derivative GPU Subroutines Using CUDA-C

Example of More Complex Kernel – Viscous Stresses (Kernel-1)

```
//   //load into shared memory
     xyrs[0][ly][lx] = x[index_ng];
     xyrs[1][ly][lx] = y[index_ng];
     xyrs[2][ly][lx] = r[index_ng];
     xyrs[3][ly][lx] = sth[index_ng];
```

X, Y, R, and ST are copied to GPU memory

```
     u_sh[0][ly][lx] = u[index_ng*9 + 6];
     u_sh[1][ly][lx] = u[index_ng*9 + 7];
     u_sh[2][ly][lx] = u[index_ng*9 + 8];
     u_sh[3][ly][lx] = w[index_ng*2 + 1];
```

U (flow variable) sub-array 6, 7, 8, and 1 are copied to GPU memory

```
     float ovol = 1.0/vol[index_c];

     __syncthreads();
```

Second Derivative GPU Subroutines Using CUDA-C

Example of More Complex Kernel – Viscous Stresses (Kernel-1)

```
ars_we[0][ly][lx] = abs((0.5f*(xyrs[1][ly][lx] + xyrs[1][ly+1][lx]))*s1+s2);
ars_we[1][ly][lx] =      (0.5f*(xyrs[2][ly][lx] + xyrs[2][ly+1][lx]))*s2+s1;
ars_we[2][ly][lx] =      (0.5f*(xyrs[3][ly][lx] + xyrs[3][ly+1][lx]))*s2+s1;

ars_ns[0][ly][lx] = abs((0.5f*(xyrs[1][ly][lx] + xyrs[1][ly][lx+1]))*s1+s2);
ars_ns[1][ly][lx] =      (0.5f*(xyrs[2][ly][lx] + xyrs[2][ly][lx+1]))*s2+s1;
ars_ns[2][ly][lx] =      (0.5f*(xyrs[3][ly][lx] + xyrs[3][ly][lx+1]))*s2+s1;

float arca = 0.5f*(ars_we[0][ly][lx]+ars_we[0][ly][lx+1]);

__syncthreads();
xyrs[1][ly][lx] = xyrs[1][ly][lx]/xyrs[2][ly][lx];

__syncthreads();
```

Face values of average R and ST are determined

Second Derivative GPU Subroutines Using CUDA-C

Example of More Complex Kernel – Viscous Stresses (Kernel-1)

```
side_we[0][ly][lx] = ars_we[0][ly][lx]*ars_we[1][ly][lx]*ars_we[2][ly][lx]
                     *(xyrs[1][ly+1][lx]-xyrs[1][ly][lx]);
side_we[1][ly][lx] = ars_we[0][ly][lx] *ars_we[2][ly][lx]
                     *(xyrs[0][ly+1][lx]-xyrs[0][ly][lx]);

side_ns[0][ly][lx] = ars_ns[0][ly][lx]*ars_ns[1][ly][lx]*ars_ns[2][ly][lx]
                     *(xyrs[1][ly][lx+1]-xyrs[1][ly][lx]);
side_ns[1][ly][lx] = ars_ns[0][ly][lx] *ars_ns[2][ly][lx]
                      *(xyrs[0][ly][lx+1]-xyrs[0][ly][lx]);

float uc = 0.25f*(u_sh[0][ly][lx]+u_sh[0][ly][lx+1]
                 +u_sh[0][ly+1][lx]+u_sh[0][ly+1][lx+1]);
float vc = 0.25f*(u_sh[1][ly][lx]+u_sh[1][ly][lx+1]
                 +u_sh[1][ly+1][lx]+u_sh[1][ly+1][lx+1]);
float hc = 0.25f*(u_sh[2][ly][lx]+u_sh[2][ly][lx+1]
                 +u_sh[2][ly+1][lx]+u_sh[2][ly+1][lx+1]);
float wc = 0.25f*(u_sh[3][ly][lx]+u_sh[3][ly][lx+1]
                  +u_sh[3][ly+1][lx]+u_sh[3][ly+1][lx+1]);

__syncthreads();
```

Face areas of cells are computed to prepare for integration to get flow derivatives at cell centers

Cell-centered values of flow variables are determined

Second Derivative GPU Subroutines Using CUDA-C

Example of More Complex Kernel – Viscous Stresses (Kernel-1)

```
    if(lx<blockDim.x-1 && ly<blockDim.y-1 && gx<imax-1 && gy<jmax-1)
    {
            dudx[index_cg] = 0.5f*ovol*(
         -(u_sh[0][ly+1][lx]+u_sh[0][ly+1][lx+1])*side_ns[0][ly+1][lx]
        - (u_sh[0][ly][lx]    +u_sh[0][ly+1][lx])*side_we[0][ly][lx] \
         +(u_sh[0][ly][lx+1]+u_sh[0][ly][lx]    )*side_ns[0][ly][lx]
        +(u_sh[0][ly+1][lx+1]+u_sh[0][ly][lx+1])*side_we[0][ly][lx+1]);
            dudy[index_cg] = ........
            dvdx[index_cg] = ........
            dvdy[index_cg] = ........
            dhdx[index_cg] = .......
            dhdy[index_cg] = ........
            dwdx[index_cg] = .......
            dwdy[index_cg] = ........
    }

    __syncthreads();
}
```

Cell-centered derivatives of flow variables are determined

Calls to kernel-2 and kernel-3 follow and code is similar in structure to kernel-1

Second Derivative GPU Subroutines Using CUDA-Fortran

Example of More Complex Kernel – Viscous Stresses (Main routine)

Main routine (sets up thread execution)

```
   subroutine gpu_stress

     threadBlock = dim3(STRESSBLKX,4,1)
     grid = dim3((imax(1)+threadBlock%x-2) / (threadBlock%x-1),
(jmax(1)+threadBlock%y-2) / (threadBlock%y-1), 1)

     call gpu_stress_kernel_1<<<grid,threadBlock>>>
     call gpu_stress_kernel_2<<<grid,threadBlock>>>
     call gpu_stress_kernel_3<<<grid,threadBlock>>>


   end subroutine gpu_stress
```

Second Derivative GPU Subroutines Using CUDA-Fortran

Example of More Complex Kernel – Viscous Stresses (Kernel-1)

```
 attributes(global) subroutine gpu_stress_kernel_1
   integer :: lx, ly, gx, gy, n
   real*8 :: ovol, arca, uc, vc, wc
   real*8, shared :: u_sh(STRESSBLKX,4,4), xyrs(STRESSBLKX,4,4), ars_we(STRESSBLKX,4,3),
                         ars_ns(STRESSBLKX,4,3),
&    side_we(STRESSBLKX,4,2), side_ns(STRESSBLKX,4,2)
   lx = threadIdx%x
   ly = threadIdx%y
   gx = threadIdx%x + (blockDim%x - 1)*(blockIdx%x - 1)
   gy = threadIdx%y + (blockDim%y - 1)*(blockIdx%y - 1)

   !load to shared memory
   if(gx<=imax_d .and. gy<=jmax_d) then
     xyrs(lx,ly,1) = x_d(gx,gy,1)
     xyrs(lx,ly,2) = y_d(gx,gy,1)
     xyrs(lx,ly,3) = r_d(gx,gy,1)
     xyrs(lx,ly,4) = sth_d(gx,gy,1)

     u_sh(lx,ly,1) = _u_d(7,gx,gy,1)
     u_sh(lx,ly,2) = _u_d(8,gx,gy,1)
     u_sh(lx,ly,3) = _u_d(9,gx,gy,1)
     u_sh(lx,ly,4) = w_d(2,gx,gy,1)

     ovol = 1.d0/vol_d(gx,gy,1)
   endif

   call syncthreads
```

Load grid and variables to GPU

Second Derivative GPU Subroutines Using CUDA-Fortran

Example of More Complex Kernel – Viscous Stresses (Kernel-1)

```
if(ly<blockDim%y) then
  ars_we(lx,ly,1) = abs((0.5d0*(xyrs(lx,ly,2) + xyrs(lx,ly+1,2)))*s1_d+s2_d)
  ars_we(lx,ly,2) =     (0.5d0*(xyrs(lx,ly,3) + xyrs(lx,ly+1,3)))*s1_d+s2_d
  ars_we(lx,ly,3) =     (0.5d0*(xyrs(lx,ly,4) + xyrs(lx,ly+1,4)))*s1_d+s2_d
endif

if(lx<blockDim%x) then
  ars_ns(lx,ly,1) = abs((0.5d0*(xyrs(lx,ly,2) + xyrs(lx+1,ly,2)))*s1_d+s2_d)
  ars_ns(lx,ly,2) =     (0.5d0*(xyrs(lx,ly,3) + xyrs(lx+1,ly,3)))*s1_d+s2_d
  ars_ns(lx,ly,3) =     (0.5d0*(xyrs(lx,ly,4) + xyrs(lx+1,ly,4)))*s1_d+s2_d
  arca = 0.5d0*(ars_we(lx,ly,1) + ars_we(lx+1,ly,1))
endif

call syncthreads

xyrs(lx,ly,2) = xyrs(lx,ly,2)/xyrs(lx,ly,3)

call syncthreads
```

Calculate face average values of radius and span-wise height

Second Derivative GPU Subroutines Using CUDA-Fortran

Example of More Complex Kernel – Viscous Stresses (Kernel-1)

```
    if(ly<blockDim%y) then
      side_we(lx,ly,1) = ars_we(lx,ly,1)*ars_we(lx,ly,2)*ars_we(lx,ly,3)*(xyrs(lx,ly+1,2)-
xyrs(lx,ly,2))
      side_we(lx,ly,2) = ars_we(lx,ly,1)*ars_we(lx,ly,3)*(xyrs(lx,ly+1,1)-xyrs(lx,ly,1))
    endif

    if(lx<blockDim%x) then
      side_ns(lx,ly,1) = ars_ns(lx,ly,1)*ars_ns(lx,ly,2)*ars_ns(lx,ly,3)*(xyrs(lx+1,ly,2)-
xyrs(lx,ly,2))
      side_ns(lx,ly,2) = ars_ns(lx,ly,1)*ars_ns(lx,ly,3)*(xyrs(lx+1,ly,1)-xyrs(lx,ly,1))
    endif

    if(lx<blockDim%x .and. ly<blockDim%y) then
      uc = 0.25d0*(u_sh(lx,ly,1)+u_sh(lx+1,ly,1)+u_sh(lx,ly+1,1)+u_sh(lx+1,ly+1,1))
      hc = 0.25d0*(u_sh(lx,ly,3)+u_sh(lx+1,ly,3)+u_sh(lx,ly+1,3)+u_sh(lx+1,ly+1,3))
      wc = 0.25d0*(u_sh(lx,ly,4)+u_sh(lx+1,ly,4)+u_sh(lx,ly+1,4)+u_sh(lx+1,ly+1,4))
    endif

    call syncthreads
```

Calculate
face lengths

Calculate
cell-centered
variables

Second Derivative GPU Subroutines Using CUDA-Fortran

Example of More Complex Kernel – Viscous Stresses (Kernel-1)

```
      if(lx<blockDim%x .and. ly<blockDim%y .and. gx<imax_d .and. gy<jmax_d) then
        dudx_d(gx,gy,1) = 0.5d0*ovol*( -(u_sh(lx  ,ly+1,1) + u_sh(lx+1,ly+1,1)) *
side_ns(lx,ly+1,1)
    &                             -(u_sh(lx  ,ly  ,1) + u_sh(lx  ,ly+1,1)) * side_we(lx  ,ly,1)
    &                             +(u_sh(lx+1,ly  ,1) + u_sh(lx  ,ly  ,1)) * side_ns(lx,ly  ,1)
    &                             +(u_sh(lx+1,ly+1,1) + u_sh(lx+1,ly  ,1)) * side_we(lx+1,ly,1))
        dudy_d(gx,gy,1) = 0.5d0*ovol*( +(u_sh(lx  ,ly+1,1) + u_sh(lx+1,ly+1,1)) *
side_ns(lx,ly+1,2)
    &                             +(u_sh(lx  ,ly  ,1) + u_sh(lx  ,ly+1,1)) * side_we(lx  ,ly,2)
    &                             -(u_sh(lx+1,ly  ,1) + u_sh(lx  ,ly  ,1)) * side_ns(lx,ly  ,2)
    &                             -(u_sh(lx+1,ly+1,1) + u_sh(lx+1,ly  ,1)) * side_we(lx+1,ly,2))
    &                             -s1_d*uc/arca
        dvdx_d(gx,gy,1) = 0.5d0*ovol*( -(u_sh(lx  ,ly+1,2) + u_sh(lx+1,ly+1,2)) *
side_ns(lx,ly+1,1)
    &                             -(u_sh(lx  ,ly  ,2) + u_sh(lx  ,ly+1,2)) * side_we(lx  ,ly,1)
    &                             +(u_sh(lx+1,ly  ,2) + u_sh(lx  ,ly  ,2)) * side_ns(lx,ly  ,1)
    &                             +(u_sh(lx+1,ly+1,2) + u_sh(lx+1,ly  ,2)) * side_we(lx+1,ly,1))
        dvdy_d(gx,gy,1) = 0.5d0*ovol*( +(u_sh(lx  ,ly+1,2) + u_sh(lx+1,ly+1,2)) *
side_ns(lx,ly+1,2)
    &                             +(u_sh(lx  ,ly  ,2) + u_sh(lx  ,ly+1,2)) * side_we(lx  ,ly,2)
    &                             -(u_sh(lx+1,ly  ,2) + u_sh(lx  ,ly  ,2)) * side_ns(lx,ly  ,2)
    &                             -(u_sh(lx+1,ly+1,2) + u_sh(lx+1,ly  ,2)) * side_we(lx+1,ly,2))
```

Calculate first derivatives of velocity

Second Derivative GPU Subroutines Using CUDA-Fortran

Example of More Complex Kernel – Viscous Stresses (Kernel-1)

Calculate first derivatives of enthalpy and w-velocity

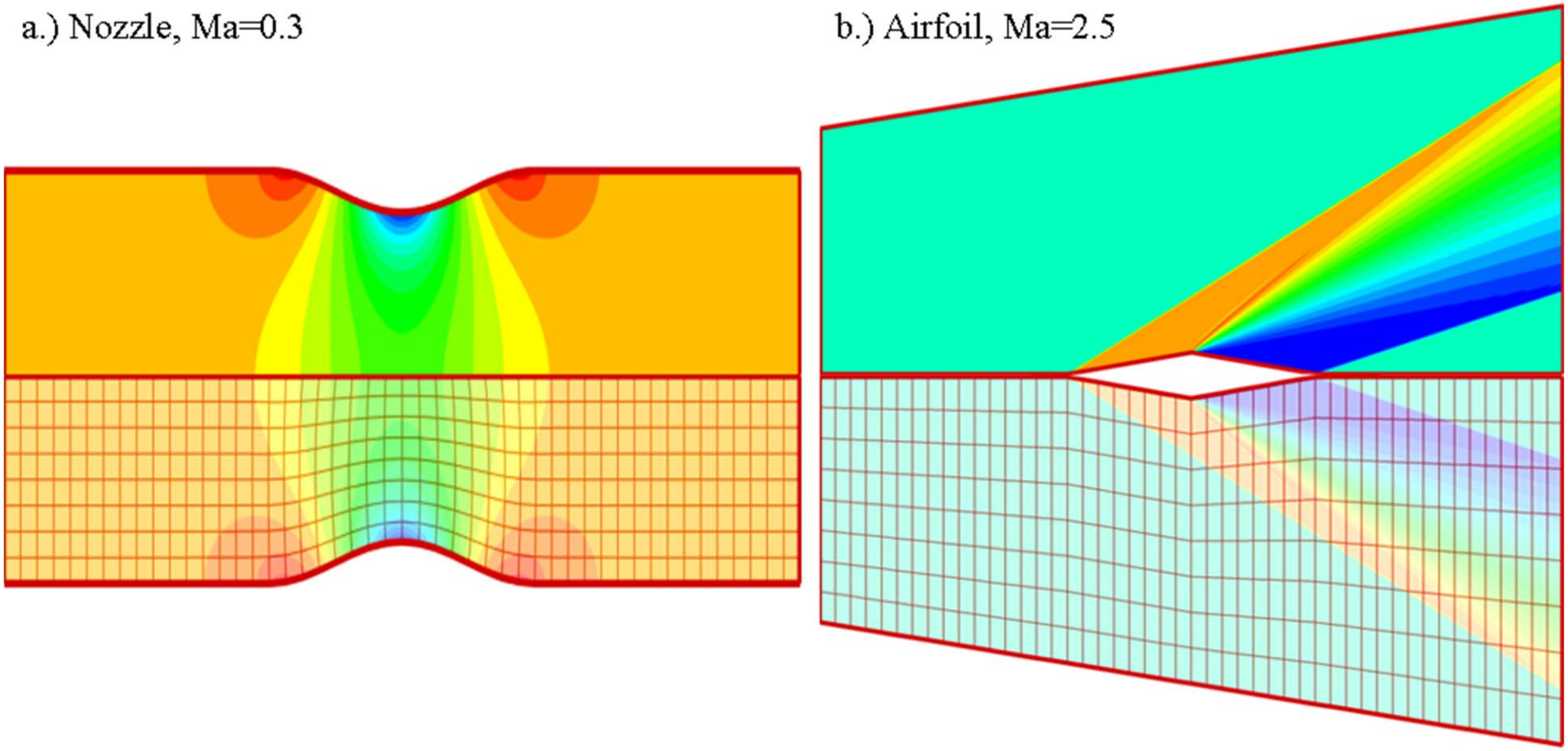
```
        dhdx_d(gx,gy,1) = 0.5d0*ovol*( -(u_sh(lx  ,ly+1,3) + u_sh(lx+1,ly+1,3)) *
side_ns(lx,ly+1,1)
    &                            -(u_sh(lx  ,ly  ,3) + u_sh(lx  ,ly+1,3)) * side_we(lx  ,ly,1)
    &                            +(u_sh(lx+1,ly  ,3) + u_sh(lx  ,ly  ,3)) * side_ns(lx,ly  ,1)
    &                            +(u_sh(lx+1,ly+1,3) + u_sh(lx+1,ly  ,3)) * side_we(lx+1,ly,1))
        dhdy_d(gx,gy,1) = 0.5d0*ovol*( +(u_sh(lx  ,ly+1,3) + u_sh(lx+1,ly+1,3)) *
side_ns(lx,ly+1,2)
    &                            +(u_sh(lx  ,ly  ,3) + u_sh(lx  ,ly+1,3)) * side_we(lx  ,ly,2)
    &                            -(u_sh(lx+1,ly  ,3) + u_sh(lx  ,ly  ,3)) * side_ns(lx,ly  ,2)
    &                            -(u_sh(lx+1,ly+1,3) + u_sh(lx+1,ly  ,3)) * side_we(lx+1,ly,2))
    &                            -s1_d*uc/arca
        dwdx_d(gx,gy,1) = 0.5d0*ovol*( -(u_sh(lx  ,ly+1,4) + u_sh(lx+1,ly+1,4)) *
side_ns(lx,ly+1,1)
    &                            -(u_sh(lx  ,ly  ,4) + u_sh(lx  ,ly+1,4)) * side_we(lx  ,ly,1)
    &                            +(u_sh(lx+1,ly  ,4) + u_sh(lx  ,ly  ,4)) * side_ns(lx,ly  ,1)
    &                            +(u_sh(lx+1,ly+1,4) + u_sh(lx+1,ly  ,4)) * side_we(lx+1,ly,1))
        dwdy_d(gx,gy,1) = 0.5d0*ovol*( +(u_sh(lx  ,ly+1,4) + u_sh(lx+1,ly+1,4)) *
side_ns(lx,ly+1,2)
    &                            +(u_sh(lx  ,ly  ,4) + u_sh(lx  ,ly+1,4)) * side_we(lx  ,ly,2)
    &                            -(u_sh(lx+1,ly  ,4) + u_sh(lx  ,ly  ,4)) * side_ns(lx,ly  ,2)
    &                            -(u_sh(lx+1,ly+1,4) + u_sh(lx+1,ly  ,4)) * side_we(lx+1,ly,2))
    &                            -2.d0*s1_d*wc/arca

      endif

    end subroutine gpu_stress_kernel_1
```

Data Manipulation on the GPU

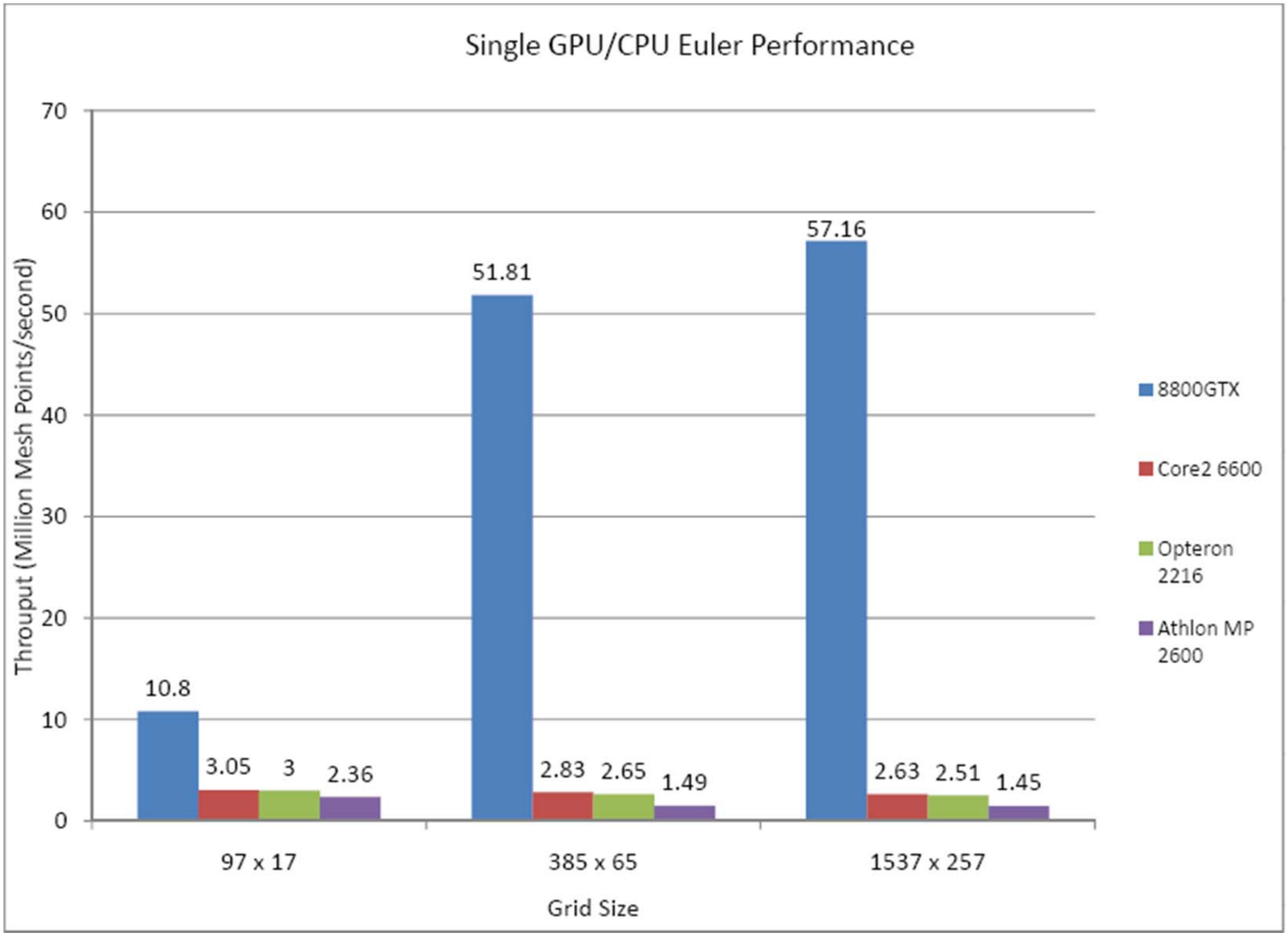
- **Copying arrays to/from the CPU from/to the GPU can be performed with**
  - cudaMemcpy using CUDA-C
  - MPI_Send and/or MPI_Recv with CUDA-Fortran
- **The movement of data to/from the GPU is an expensive operation**
  - This overhead must be minimized or made asynchronous as much as possible to obtain high speed-ups on the GPU
  - All pertinent arrays, scalars, etc. should be moved as infrequently as possible.
- **Allocation of arrays on GPU are performed using**
  - Malloc using CUDA-C
  - Allocate using CUDA-Fortran
- **Also, data locality is important on the GPU. A stride greater than 3 or 4 can cause GPU code to be slower by a factor of 2! The GPU prefers to see individual variable arrays or arrays with a right-most variable index.**

Euler Results

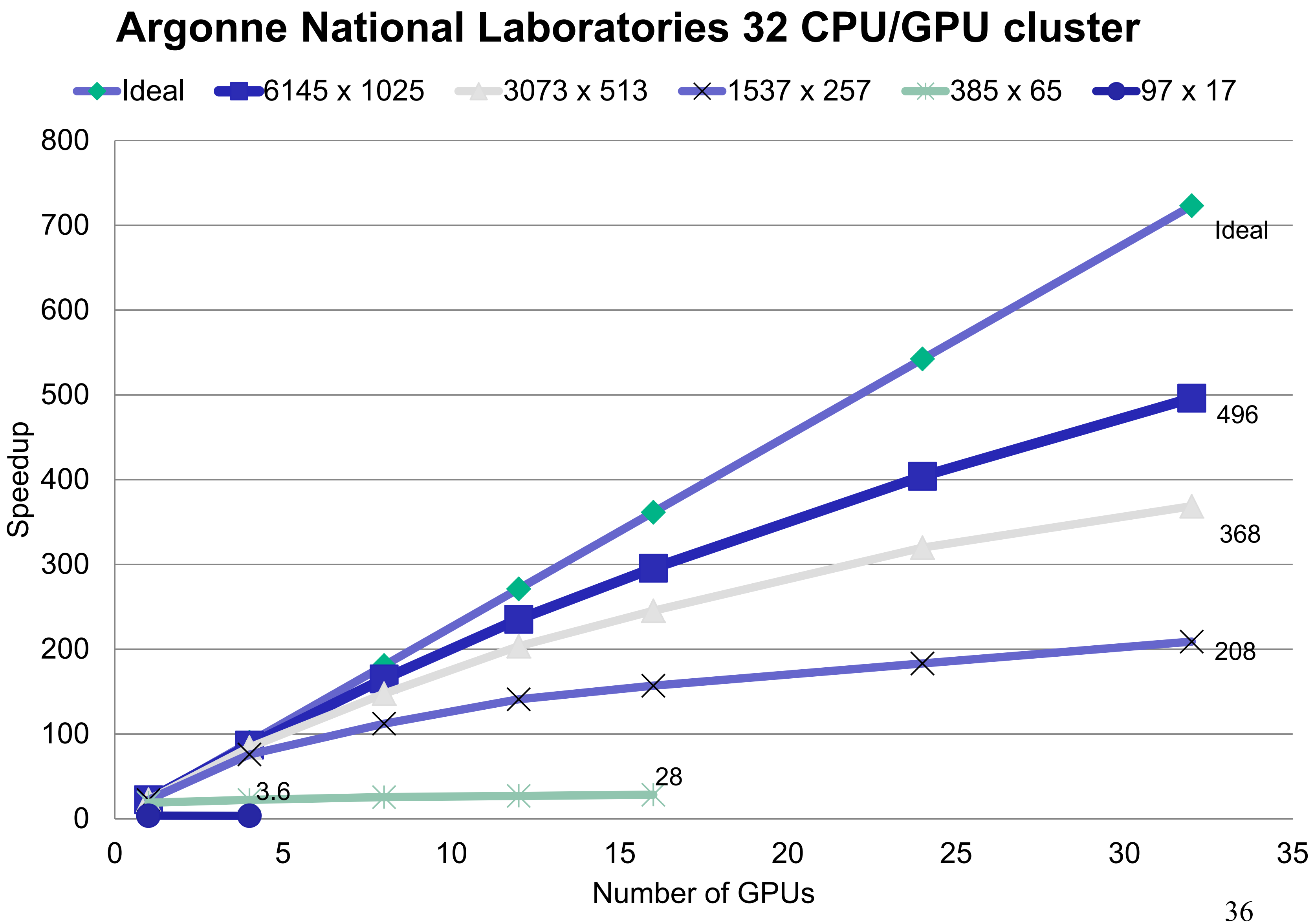
- **Subsonic nozzle and supersonic diamond airfoil**
  - Grids up to 6.4M points



Single GPU Special-Purpose Euler Solver Performance

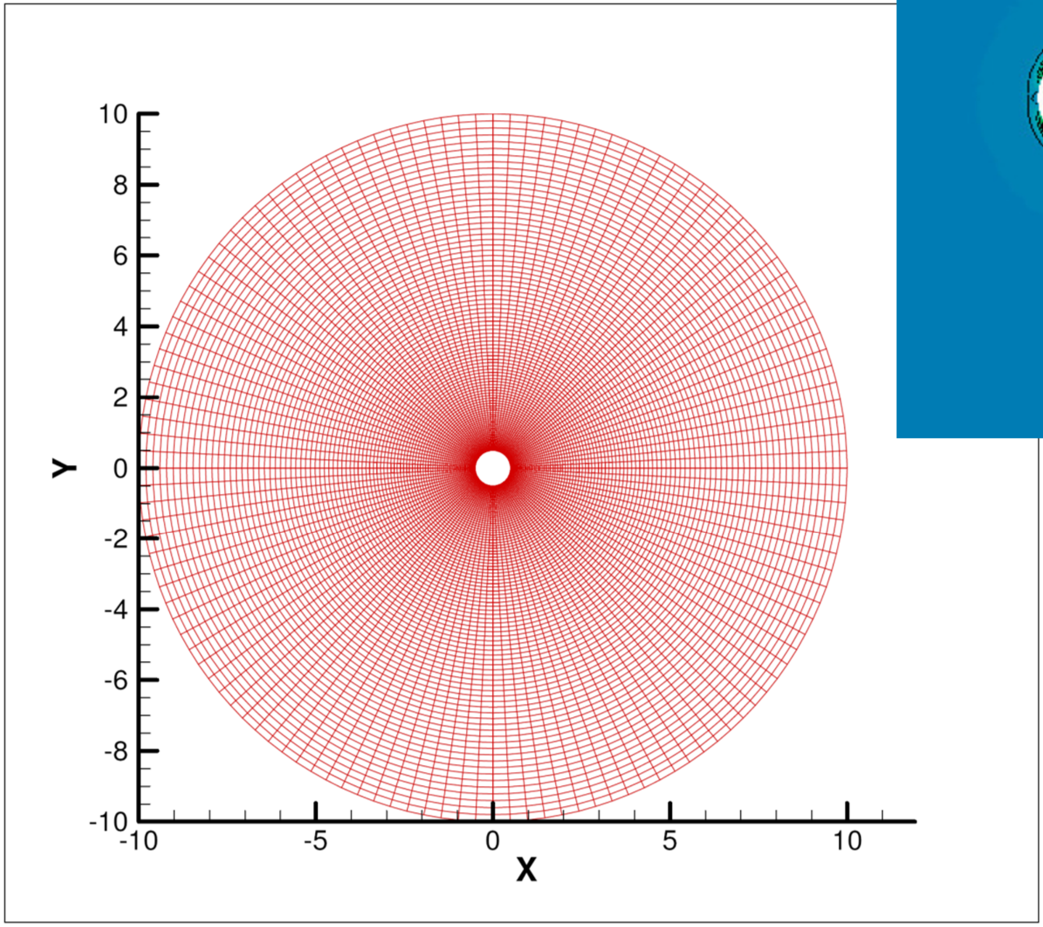


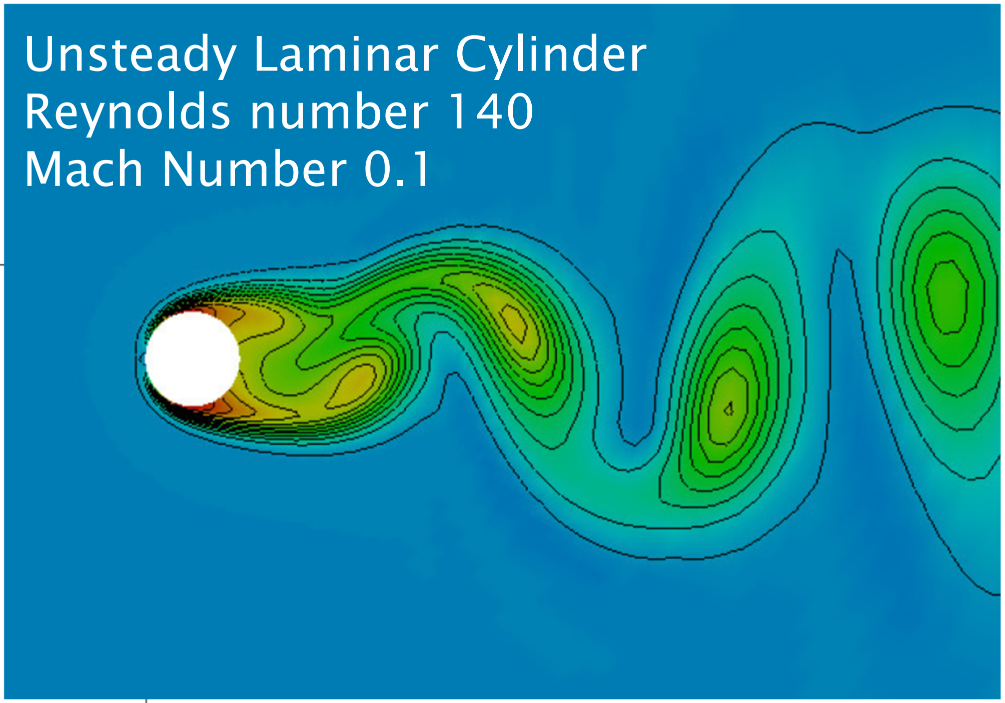
Parallel Euler Performance



General-Purpose Navier-Stokes Results

Up to 32 Blocks in Computational Grid

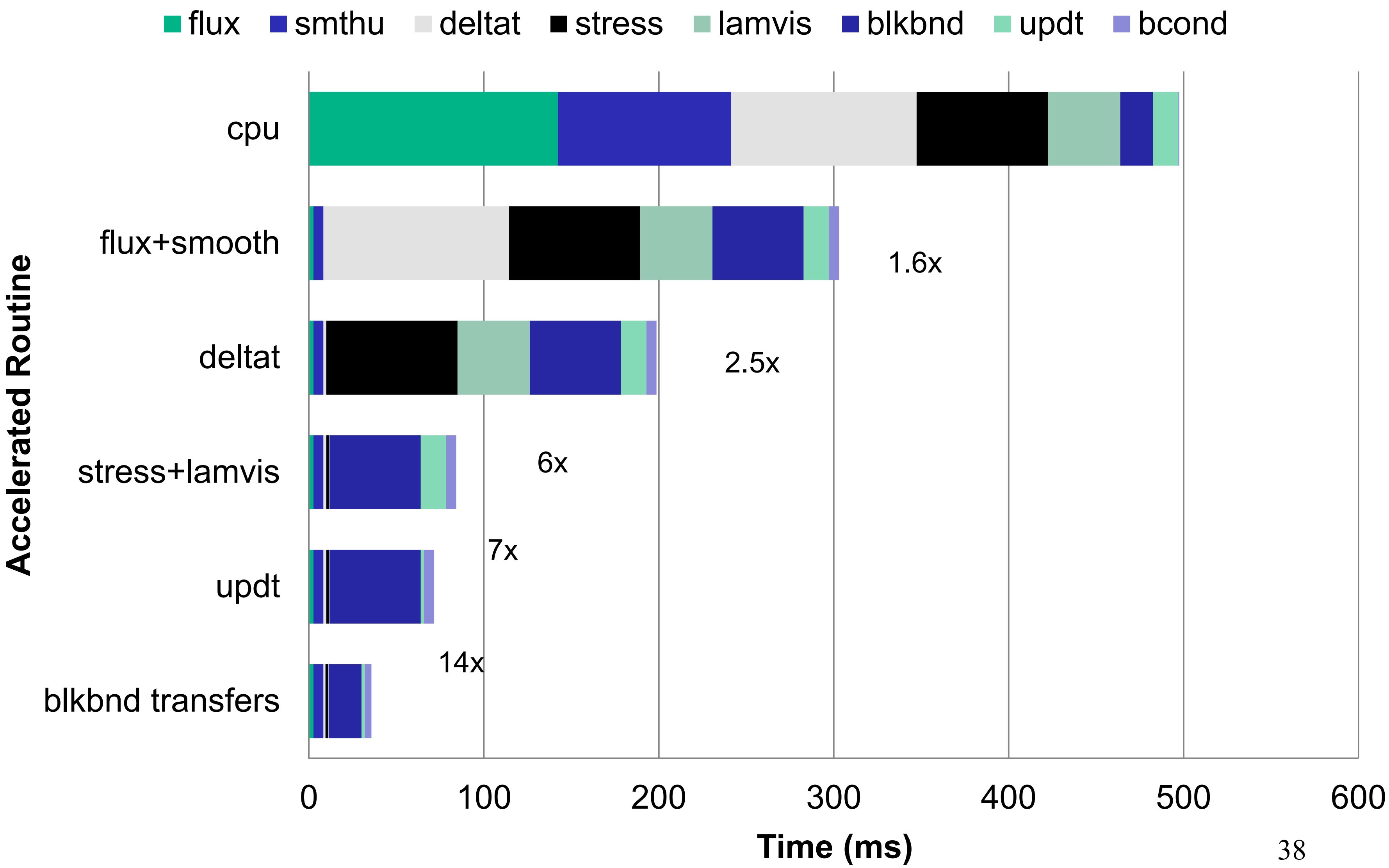




Unsteady Laminar Cylinder
Reynolds number 140
Mach Number 0.1

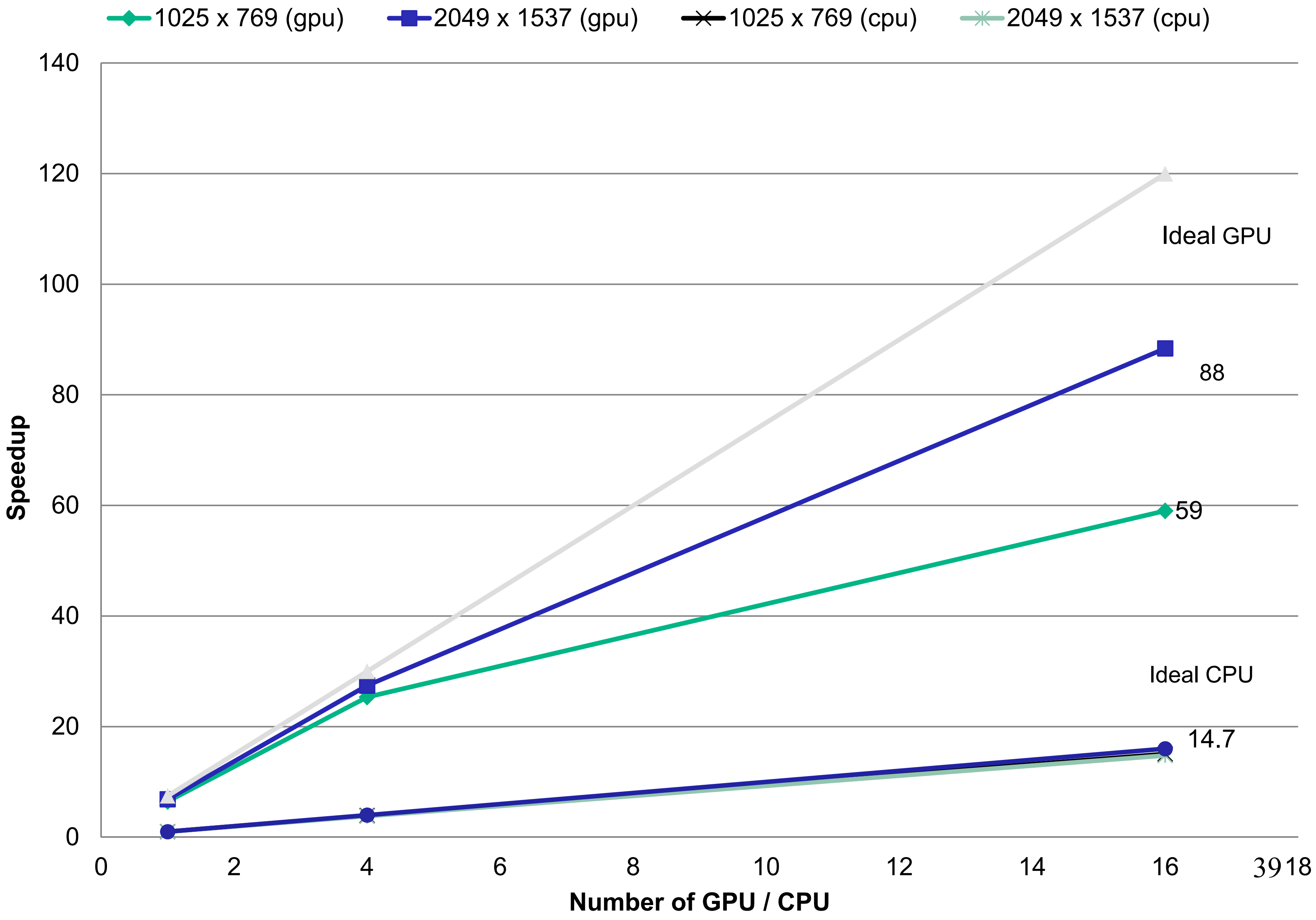
Entropy Contours

Single GPU Performance



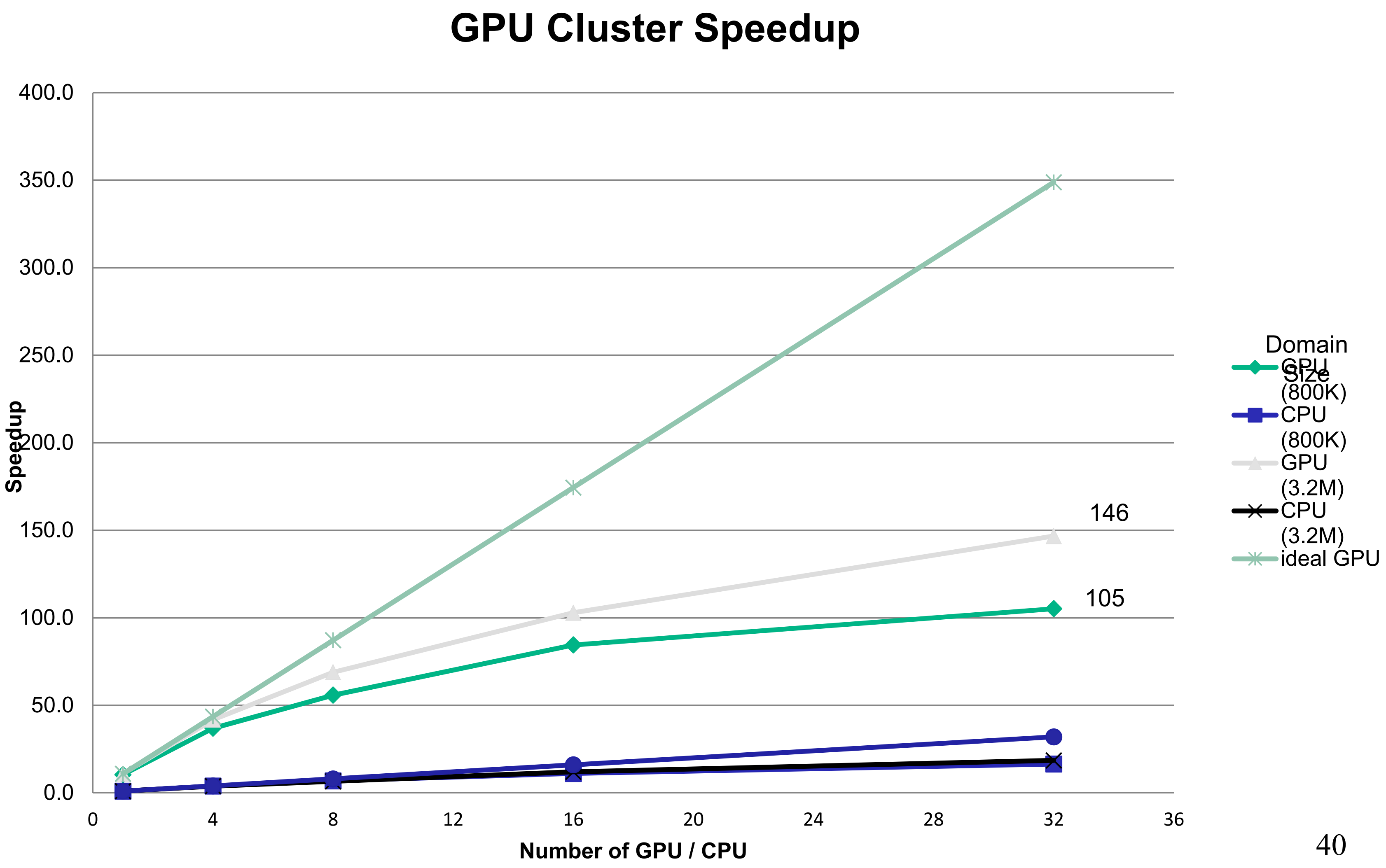
Multi-GPU Parallel Performance

ECE Cluster at UC Davis



Further Multi-GPU Parallel Results

Argonne National Laboratories 32 CPU/GPU cluster



Summary of Performance Gains

- **For 16 CPU/GPU combined processors (ECE Cluster)**
  - Euler (optimal) speed-up of ~300 (6.4 M points) over single CPU
    - A factor of 18.75 over 16 CPUs at 100% efficiency.
  - Navier-Stokes (typical) speed-up of ~88 (3.1 M points) over single CPU
    - A factor of 6.5 over 16 CPUs at 85% efficiency.
- **For 32 CPU/GPU combined processors (Argonne Cluster)**
  - Euler (optimal) speed-up increased to ~496 (65% improvement over 16 combined processors)
    - A factor of 15.5 over 32 CPUs at 100% efficiency.
  - Navier-Stokes speed-up increased to ~146 (66% improvement over 16 combined processors) on Argonne National Laboratory cluster.
    - A factor of 5.4 over 32 CPUs at 85% efficiency.
- **Greater speed-ups with larger data sets and more computing**

Effort to GPU

- **GPU parallelization of fluid-dynamics code (inviscid and laminar capability) at the time of this investigation took approximately 3 man-months each for CUDA-C and CUDA-Fortran including**
  - Time to learn original Navier-Stokes programming structure
  - Time to implement GPU routines
  - Verify solutions with CPU code

Overall Summary

- **The GPU shows great promise in increasing performance/price ratio by multiple orders in magnitude**
- **Research underway to demonstrate**
  - Generality for different algorithms
- **GPU computing could also likely reduce turn-around of other engineering software**
  - Multi-disciplinary simulations
  - Adaptive grid (AMR)
  - Embarrassingly parallel algorithms
  - Other computational science algorithms

Additional Information

- **I have put the manuals for CUDA-C, CUDA-Fortran, and OpenCL in the Additional Material/GPU folder on smartsite along with other additional reference materials.**