High-Order Spectral Difference: Validation and Acceleration using GPU Computing

Ben Zimmerman
M.S. Student
Iowa State University

IOWA STATE UNIVERSITY
Department of Aerospace Engineering

Spectral Difference

- Overview
  - Brief Introduction
  - 1-D Implementation
  - CUDA Implementation
  - CUDA Validation
  - CUDA Acceleration
  - Conclusions

Introduction

- SD3DN
  - Spectral Difference 3-D Navier-Stokes Solver
  - High-order
  - Unstructured hexahedral elements
    - Hexahedral cells offer high efficiency
  - Solves equations in differential form
    - No surface / volume integrals
  - Implemented in:
    - MPI FORTRAN
    - CUDA C++

1-D Formulation

- Consider the 1-D wave equation:

$$\frac{\partial u}{\partial t}+\frac{\partial f(u)}{\partial x}=0 \qquad f(u)=cu \qquad c>0$$

- Discuss 3rd order spacial discretization

- First assume solution takes the following form:

$$u(x)=C_0+C_1 x+C_2 x^2 \quad \longrightarrow \quad \text{3 points}$$

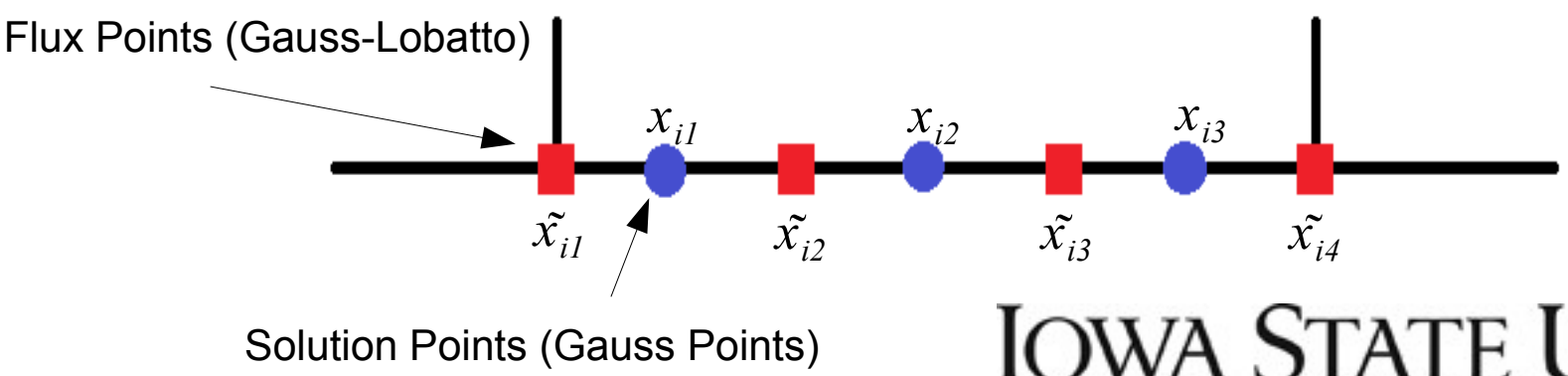
1-D Formulation

$$\frac{\partial u}{\partial t}+\frac{\partial f(u)}{\partial x}=0$$

- From before we have:

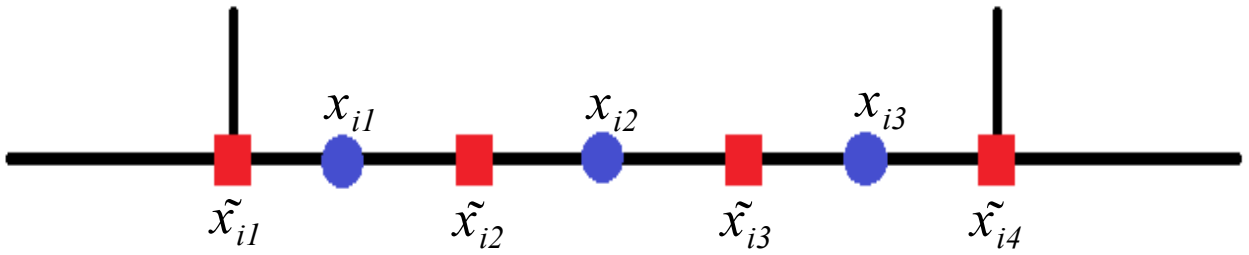
$$u \in P^2 \longrightarrow f(u) \in P^3$$

- To build flux polynomial → 4 points needed
- We use Gauss and Gauss-Lobatto Points



1-D Formulation

- How do we build the flux?



- Lagrange Polynomials (use flux point locations)

$$L_1(x)=\frac{(x-\tilde{x}_{i2})(x-\tilde{x}_{i3})(x-\tilde{x}_{i4})}{(\tilde{x}_{i1}-\tilde{x}_{i2})(\tilde{x}_{i1}-\tilde{x}_{i3})(\tilde{x}_{i1}-\tilde{x}_{i4})} \quad L_2(x)=\frac{(x-\tilde{x}_{i1})(x-\tilde{x}_{i3})(x-\tilde{x}_{i4})}{(\tilde{x}_{i2}-\tilde{x}_{i1})(\tilde{x}_{i2}-\tilde{x}_{i3})(\tilde{x}_{i2}-\tilde{x}_{i4})}$$

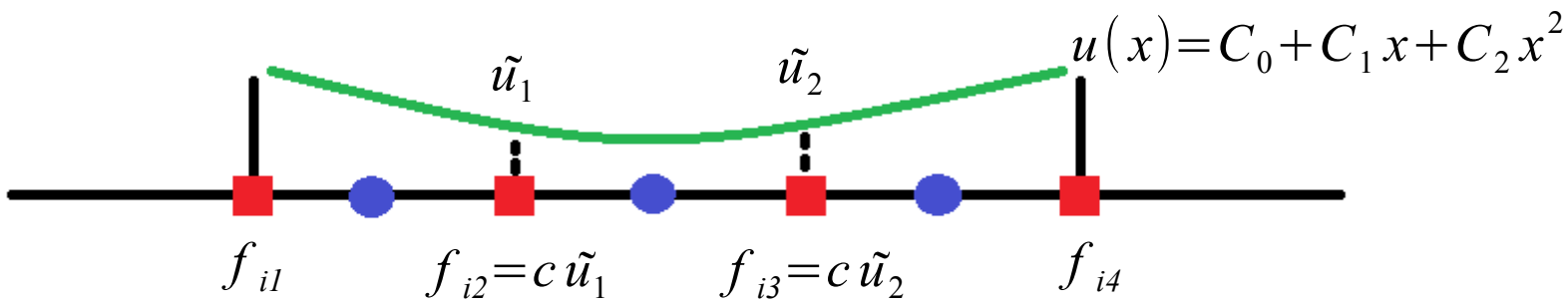
1-D Formulation

- The Flux polynomial becomes:

$$f(x) = f_{i1} L_1(x) + f_{i2} L_2(x) + f_{i3} L_3(x) + f_{i4} L_4(x)$$

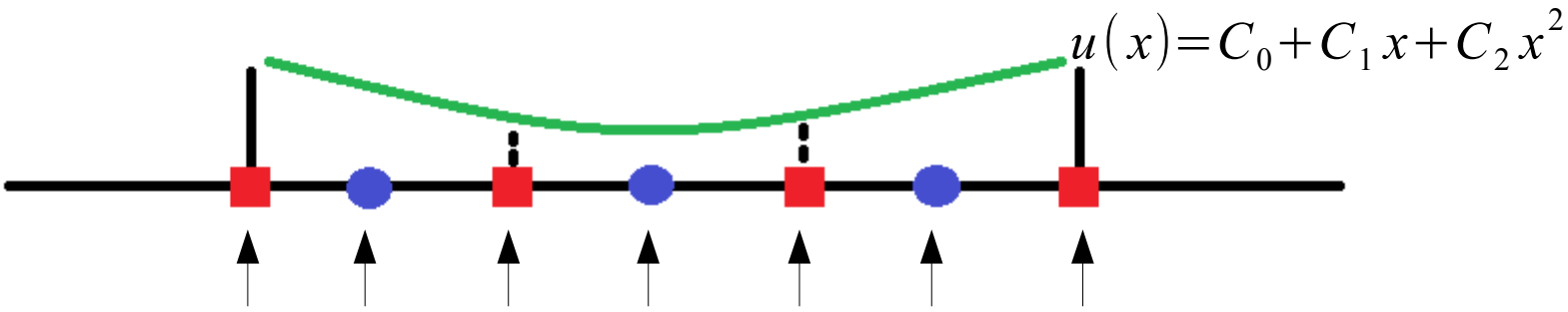
- We require the derivative and flux values:

$$\frac{\partial f(x)}{\partial x} = f_{i1} L'_1(x) + f_{i2} L'_2(x) + f_{i3} L'_3(x) + f_{i4} L'_4(x)$$



1-D Formulation

- Spectral Difference Steps
  - Get solution (u) at flux points

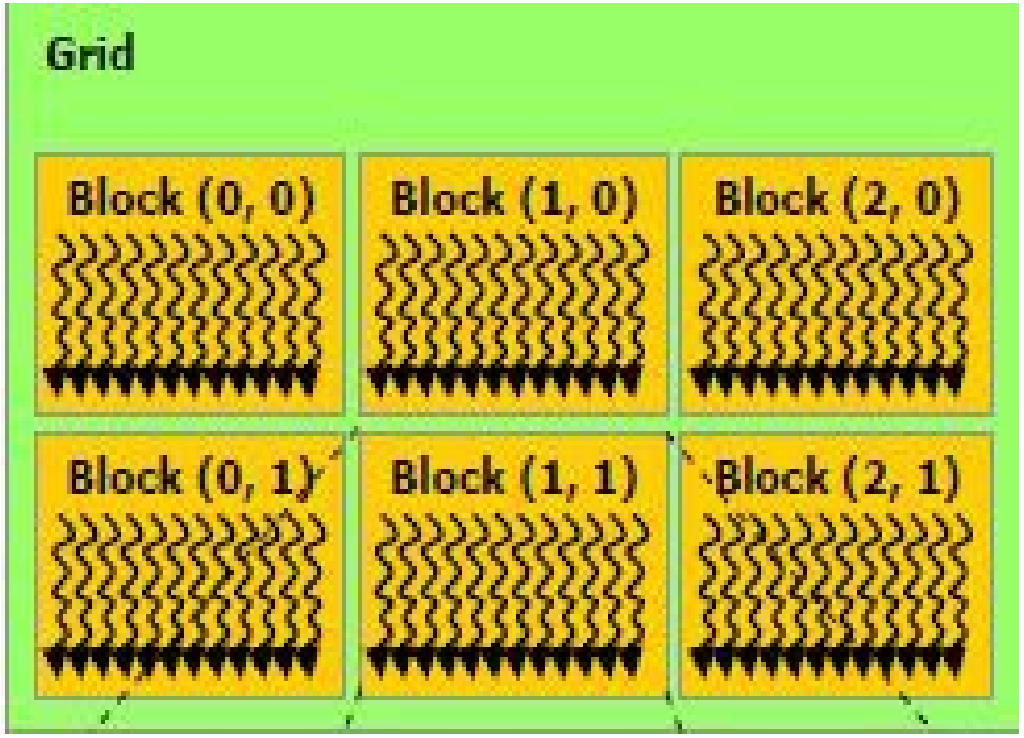


  - Evaluate interior flux using $f(u)$
  - Set end point flux to common Riemann flux
  - Get flux derivative at solution points
  - Update solution at solution points

CUDA Implementation

- What is CUDA?
  - NVIDIA GPU Computing
    - Compute Unified Device Architecture
  - Massive parallelism of programs
  - Hundreds of “CUDA cores”
  - CUDA functions (“kernals”) contain:
    - Grid
    - Blocks
    - Threads

CUDA Implementation



From NVIDIA Programming Guide

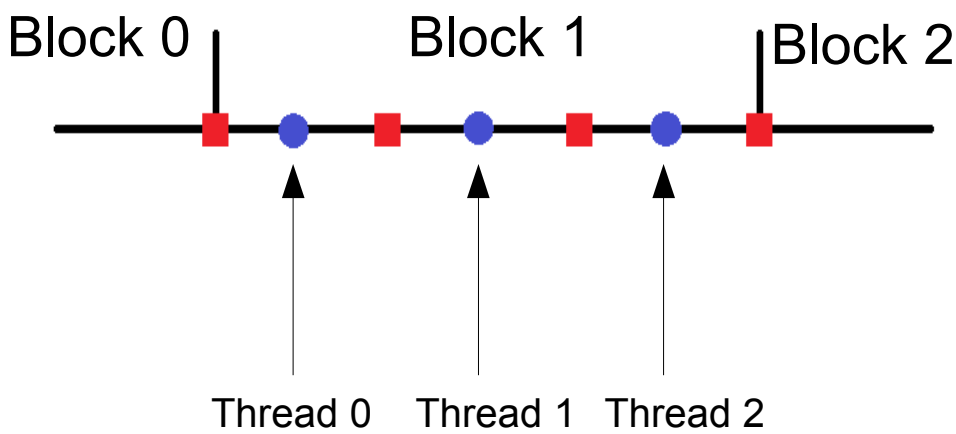
- Function ("kernal") → Grid
- This Grid contains blocks
  - blockIdx.x
  - blockIdx.y
- Each block contains threads
  - threadIdx.x
  - threadIdx.y
  - threadIdx.z

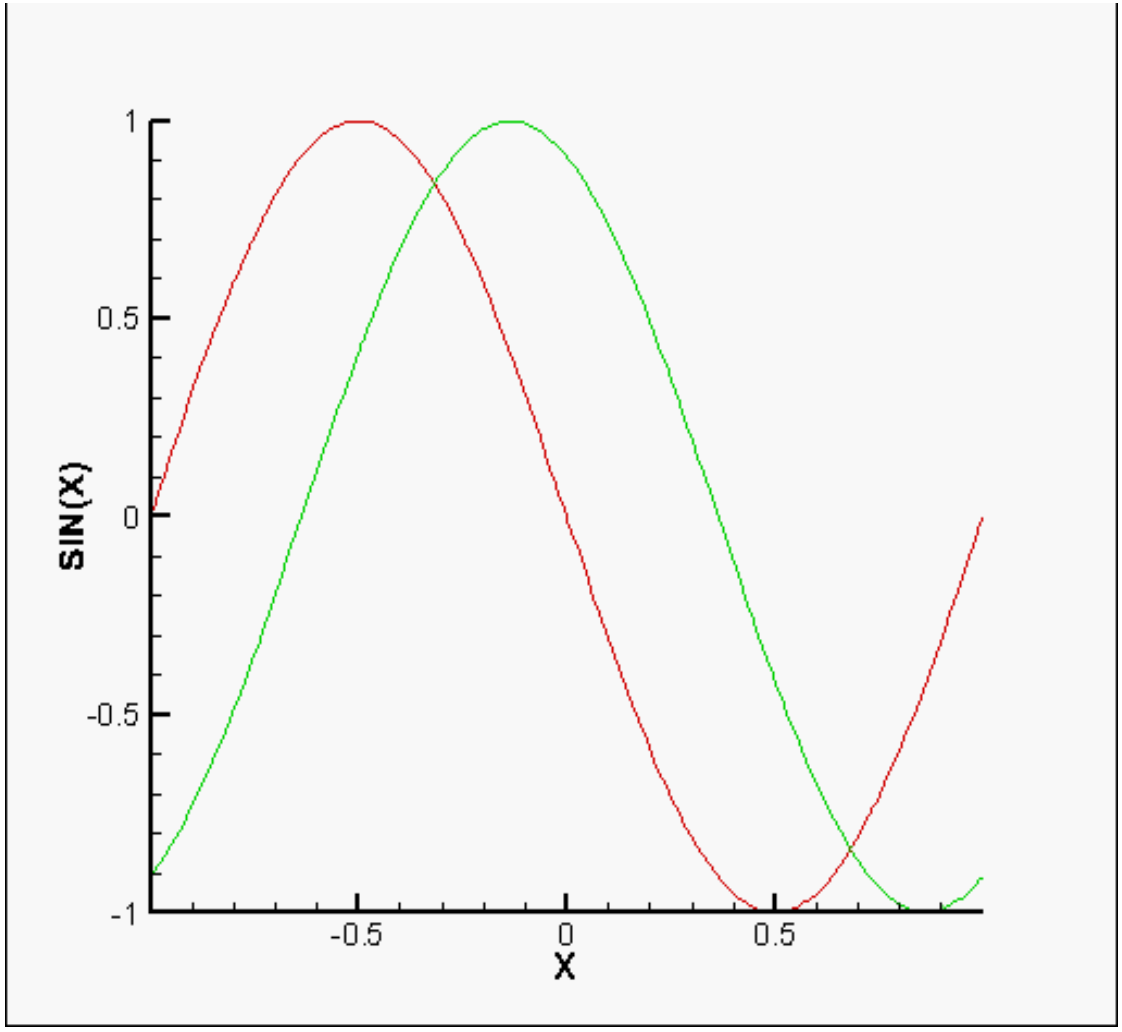
CUDA Implementation

- Memory Importance
  - Global
    - Copied from CPU to GPU and used in calculations
    - Not very fast
  - Texture
    - Bound in CPU code
    - Allows for fast reads
  - Shared
    - Allocated within GPU code
    - 150x faster than global memory

CUDA Implementation

- 1-D Case:





- Evaluate derivative of flux
  - Gauss points

CUDA Implementation

- Example: Calculate flux derivative:

$$\frac{\partial f(x)}{\partial x} = f_{i1} L'_1(x) + f_{i2} L'_2(x) + f_{i3} L'_3(x) + f_{i4} L'_4(x)$$

- Let:
  - sp Current solution point
  - fp Current flux point
  - nsp Total number of solution points per cell
  - cell Current cell
  - nc Total number of cells
  - L Array of Lagrange coefficients [nsp * (nsp+1)]
  - F Array of flux values [# of cells * (nsp+1)]

CUDA Implementation

$$\frac{\partial f(x)}{\partial x} = f_{i1} L'_1(x) + f_{i2} L'_2(x) + f_{i3} L'_3(x) + f_{i4} L'_4(x)$$

- CPU Code

```
for (int cell=0; cell<nc; cell++) {  //Loop over cells

for (int sp=0; sp<nsp; sp++) {   //Loop over solution points
dfdx[sp] = 0.0;

for (int fp=0; fp<nsp+1; fp++) { //Loop over flux points
//Flux derivitive @ solution point per cell
dfdx[sp] += F[fp + cell*(nsp+1)] * L[fp + sp*(nsp+1)];

} //End loop over flux points
dFdX[sp + cell*nsp] = dfdx[sp]; //Save to memory
} //End loop over solution points
} //End loop over Cells
```

- Requires 2 nested loops on CPU for 1-D
  - How can we transfer this to CUDA?

CUDA Implementation

$$\frac{\partial f(x)}{\partial x} = f_{i1} L'_1(x) + f_{i2} L'_2(x) + f_{i3} L'_3(x) + f_{i4} L'_4(x)$$

- GPU Code

```
int sp   = threadIdx.x;  //Solution Point
int cell = blockIdx.x;   //Current working Cell

double dFdX[nsp*nc];

dFdX[sp + cell*nsp] = 0.0;

for (int fp=0; fp<nsp+1; fp++) { //Loop over flux points

    int id=sp + cell*nsp;
    //Get derivative at solution points
    dFdX[id] += F[fp + cell*(nsp+1)] * L[fp + sp*(nsp+1)];
}
```

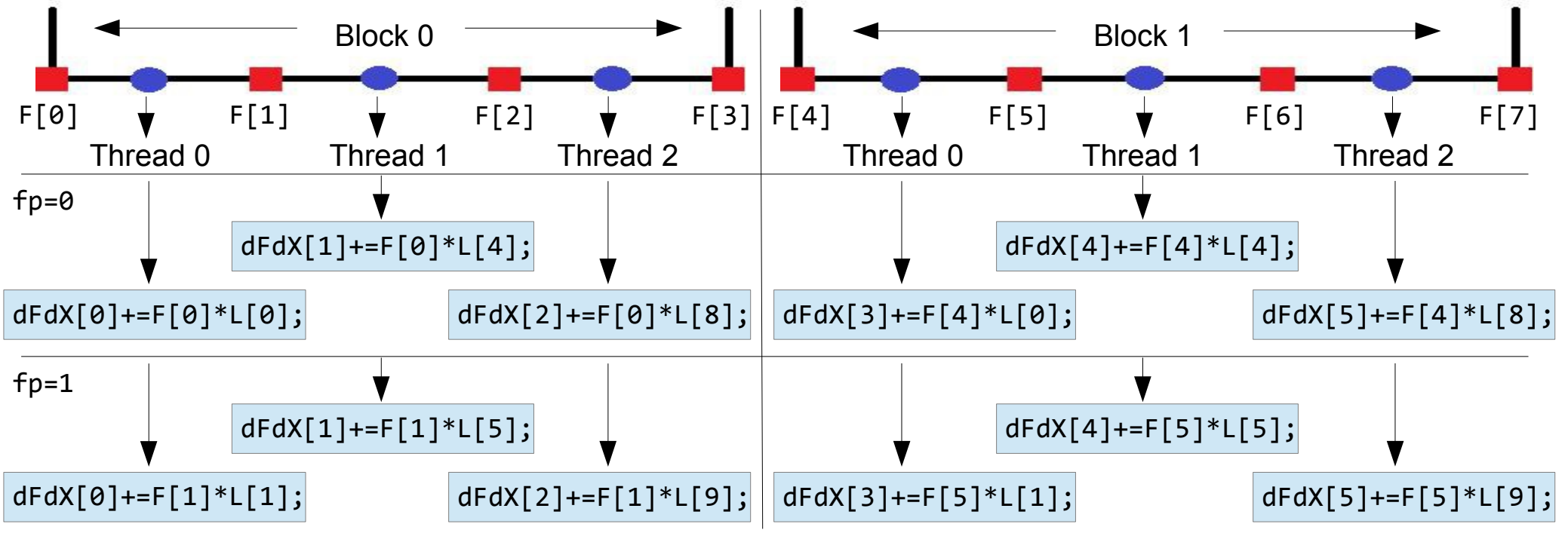
- Number of Blocks
  - nc
- Number of Threads
  - nsp

- Only one loop!
  - But what is going on?

CUDA Implementation

```
for (int fp=0; fp<nsp+1; fp++) { //Loop over flux points

    int id=sp + cell*nsp;
    //Get derivative at solution points
    dFdX[id] += F[fp + cell*(nsp+1)] * L[fp + sp*(nsp+1)];
}
```

- Recall each cell is a block and nsp threads per block



- Uses global memory

CUDA Implementation

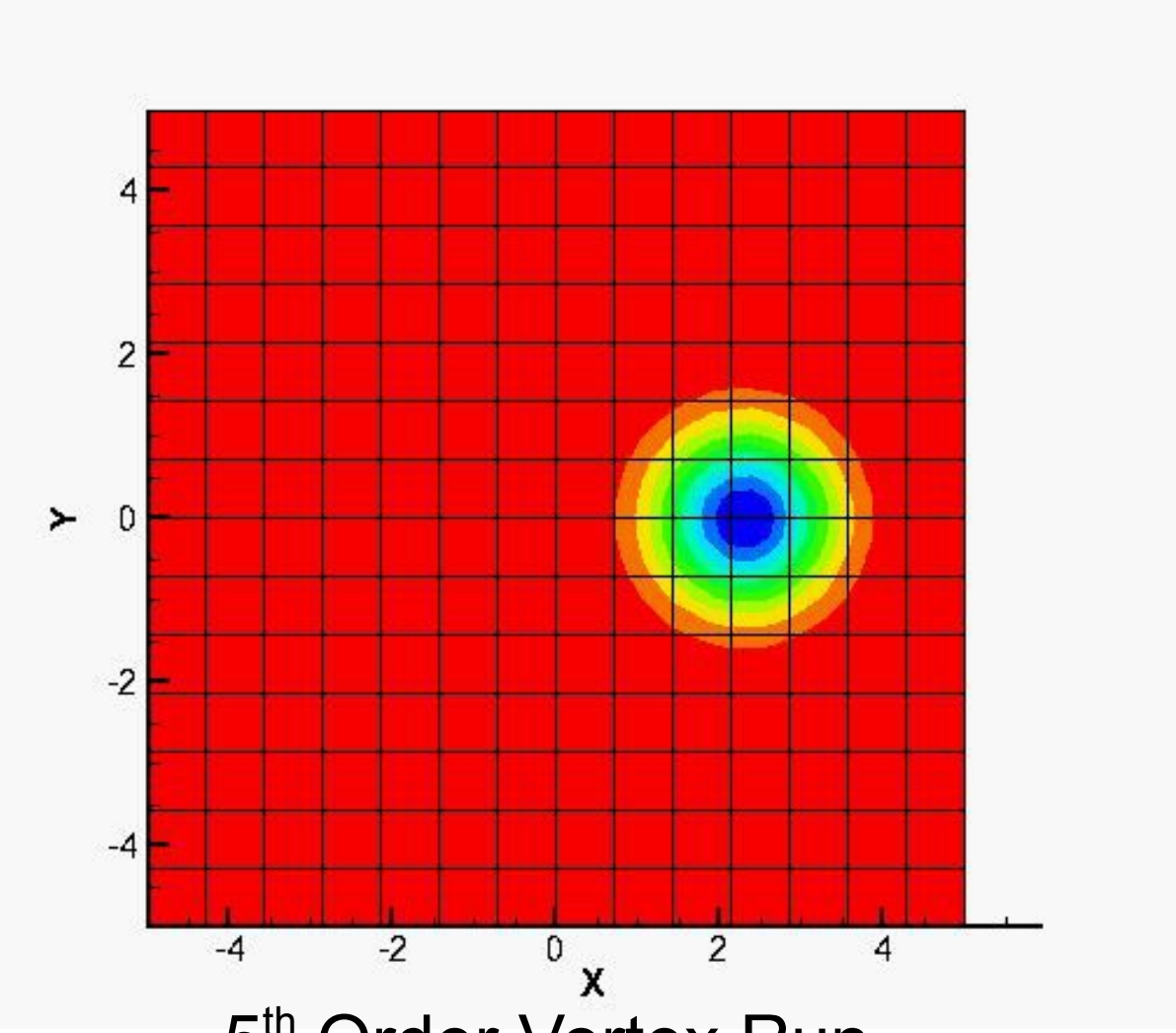
```
int sp   = threadIdx.x;   //Solution Point
int fp   = threadIdx.y;   //Flux point
int cell = blockIdx.x;    //Cell
int2 ii;                  //Dummy Variable

__shared__ double dfdx[nsp];        //dflux Shared memory
__shared__ double Ls[nsp*(nsp+1)];  //Lagrange Shared memory
__shared__ double Fs[nsp+1];        //flux Shared memory

int idx = fp + sp*(nsp+1);     //Index to read Lagrange from textured memory
int idy = fp + cell*(nsp+1);   //Index to read flux from textured memory

//Fetch data from textured memory
ii = tex1Dfetch(t_L, idx); Ls[idx] = __hiloint2double(ii.y, ii.x);
ii = tex1Dfetch(t_F, idy); Fs[fp]  = __hiloint2double(ii.y, ii.x);

__syncthreads();  //Ensure all data is loaded into Shared memory

if (fp == 0) {    //Lock down threads (do not need them)

dfdx[sp] = 0.0;

for (int m=0; m<nsp+1; m++) { //Loop over flux points
    dfdx[sp] += Fs[m] * Ls[m + sp*(nsp+1)];
}
}
```

CUDA Implementation

- What have we gained?
  - We more than doubled our lines of code
  - But...
    - Code 1 kernal launch time (global memory):
      - 8.412 milliseconds
    - Code 2 kernal launch time (shared+texture memory):
      - 3.917 milliseconds
  - Double performance per iteration!
    - Performed on NVIDIA GEFORCE GT 630M

CUDA Validation

- Isentropic Vortex
  - 2-D case
  - Motion in X
  - Two studies
    - H-refinement
    - P-refinement

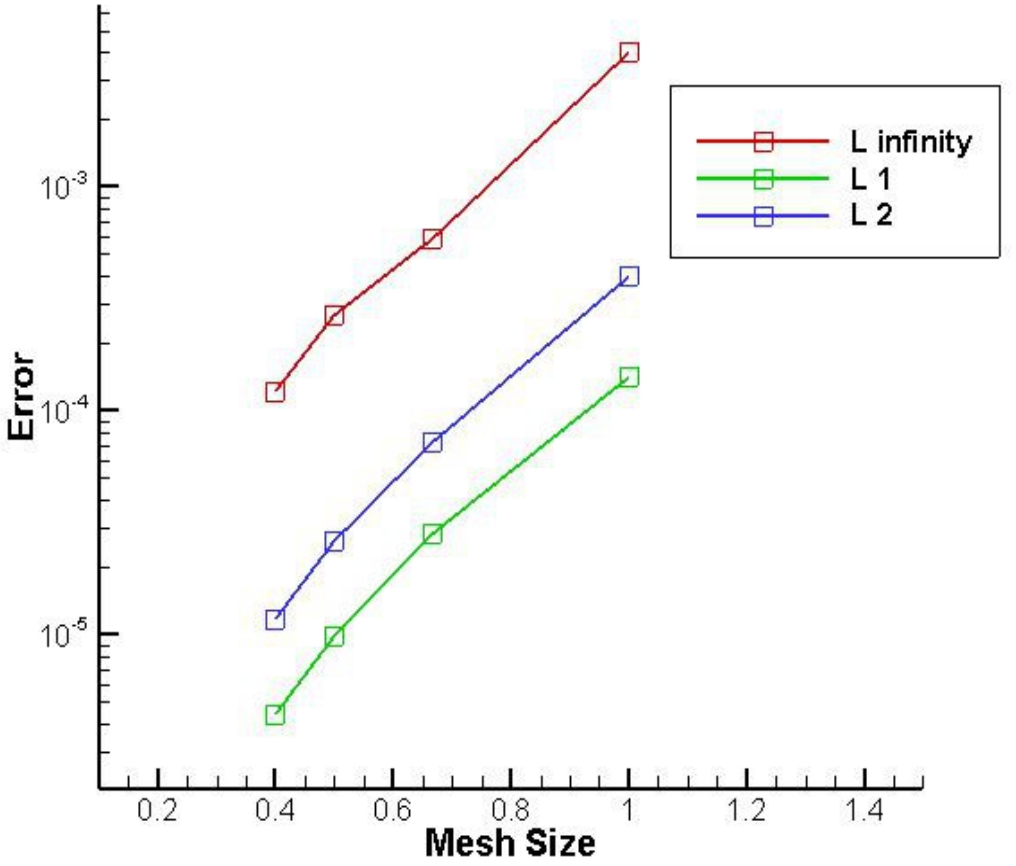


5th Order Vortex Run

CUDA Validation

- H-refinement
  - 4 Grids
    - 10x10 – 15x15
      20x20 – 25x25

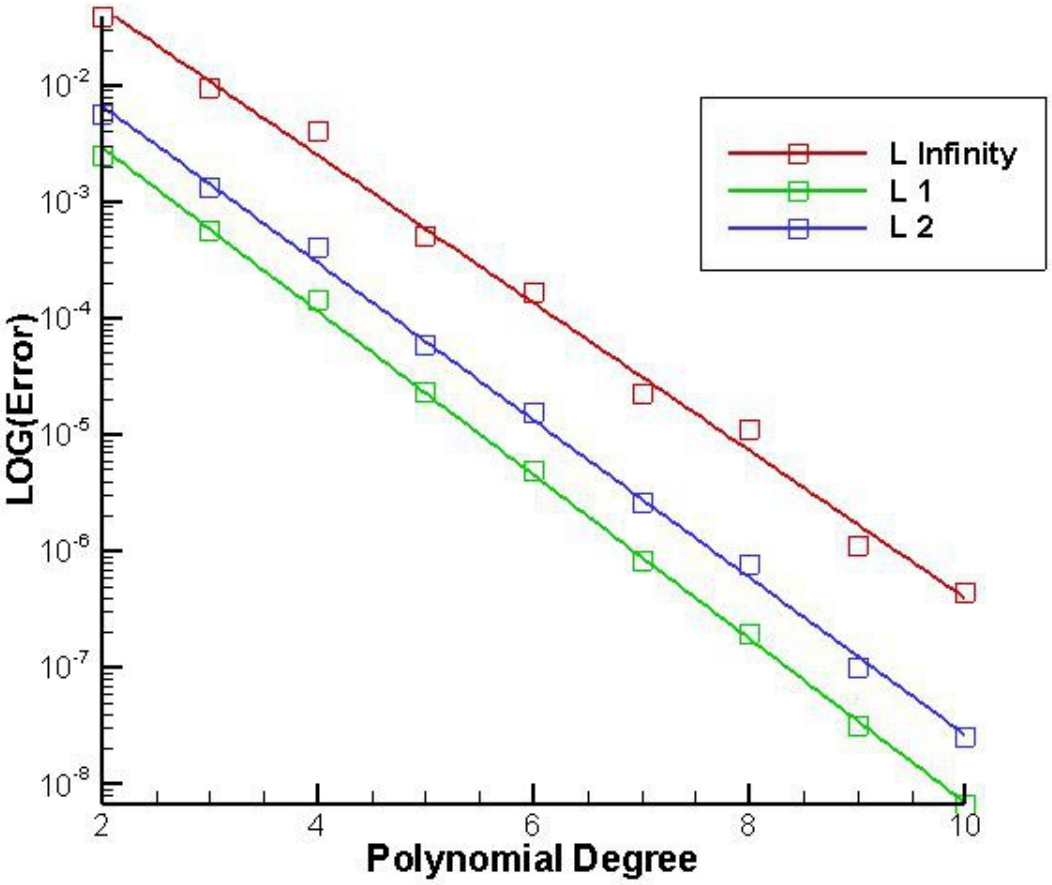
Order	L Inf Slope	L1 Slope	L2 Slope
4	3.94	3.96	3.89
5	4.76	4.98	5.03
6	5.88	6.08	6.09



4th order refinement

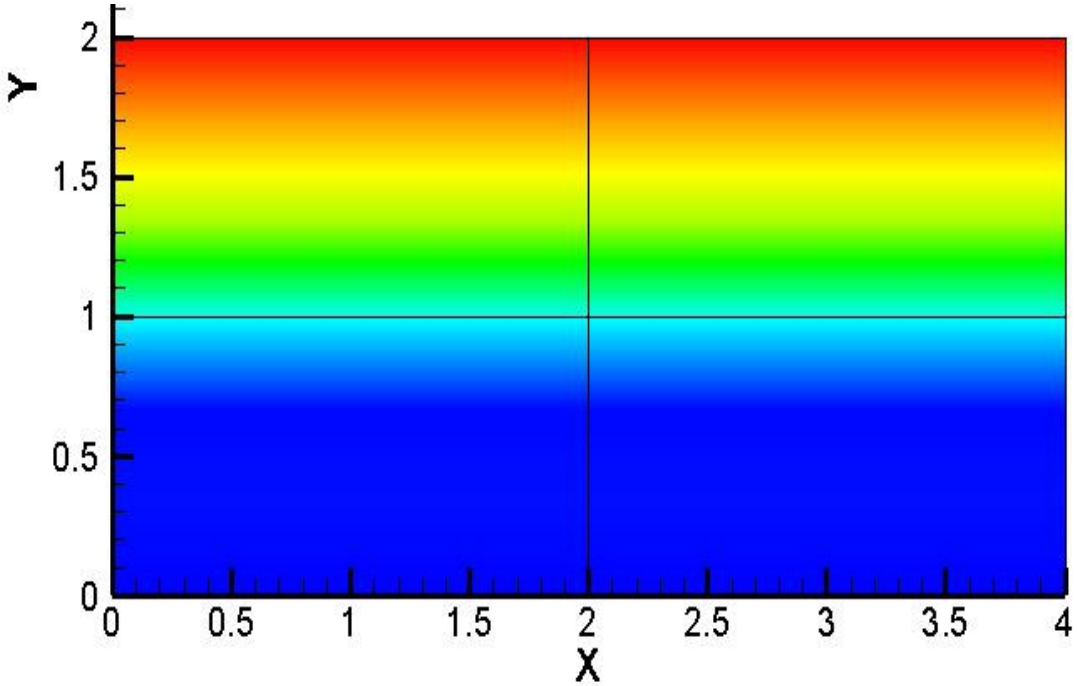
CUDA Validation

- P-refinement
  - Studied on coarse grid (10x10)
  - Polynomial increased to degree 10
  - Exponential decay of error



CUDA Validation

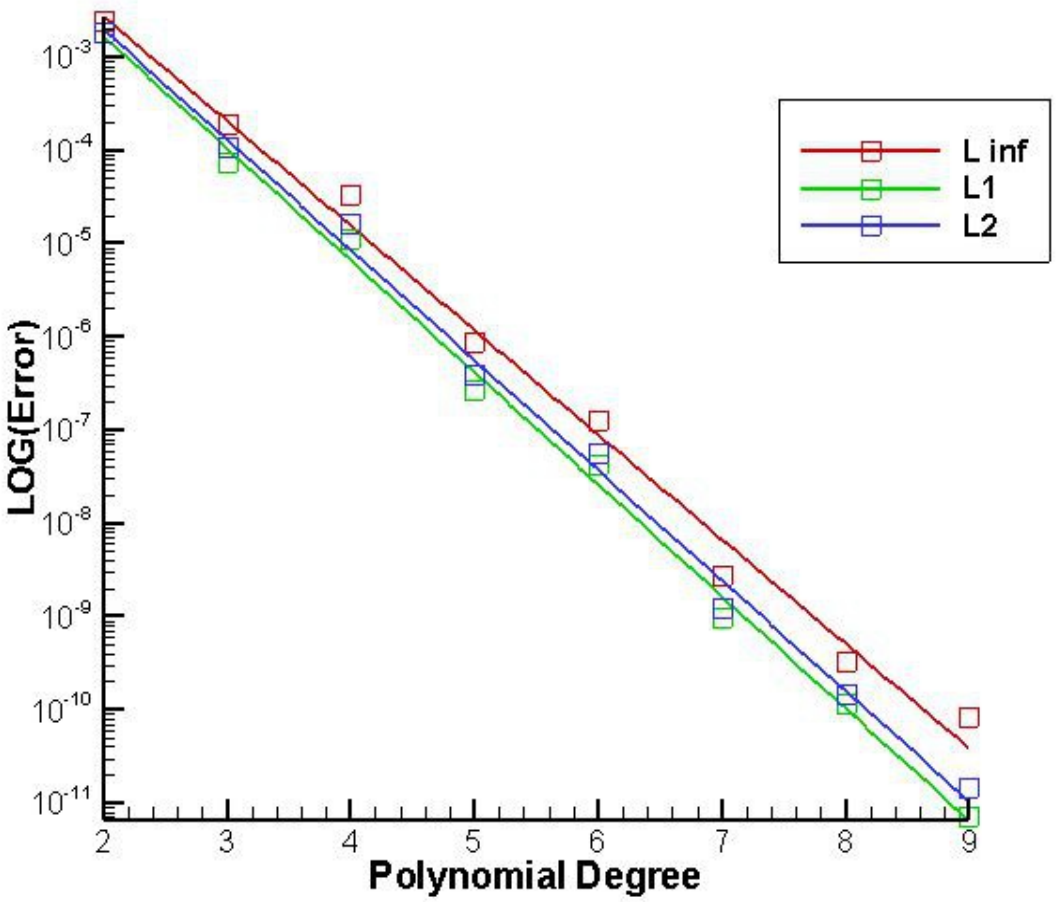
- Couette Flow
  - 3-D case
  - Flow between two parallel plates
  - Viscous validation
  - P-refinement study



Velocity Profile

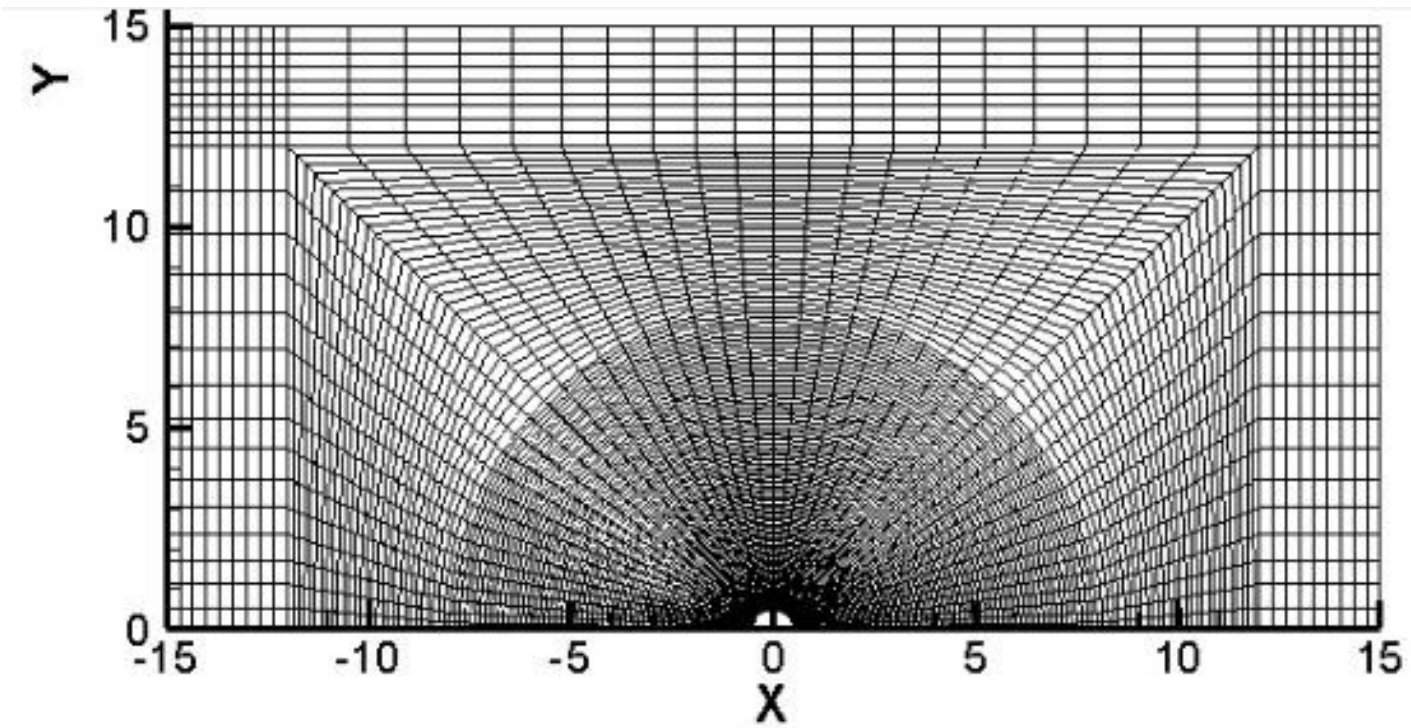
CUDA Validation

- P-refinement
  - Studied on coarse grid (2x2x1)
  - Order increased to 9th
  - Exponential decay of error



CUDA Validation

- Pulse and cylinder – 2074 cells

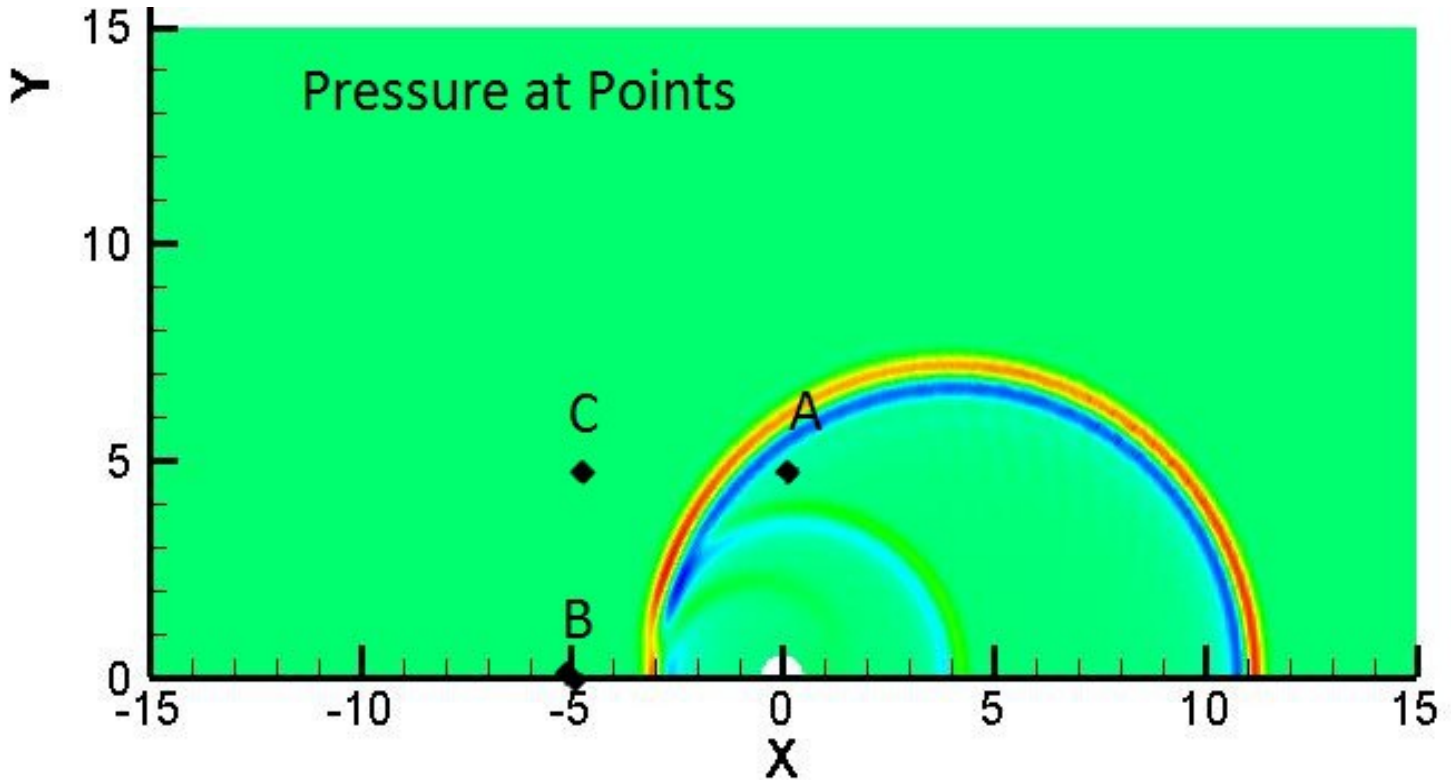


Order	# DOF
2nd	16,592
3rd	55,998
4th	132,736
5th	259,250
6th	447,984

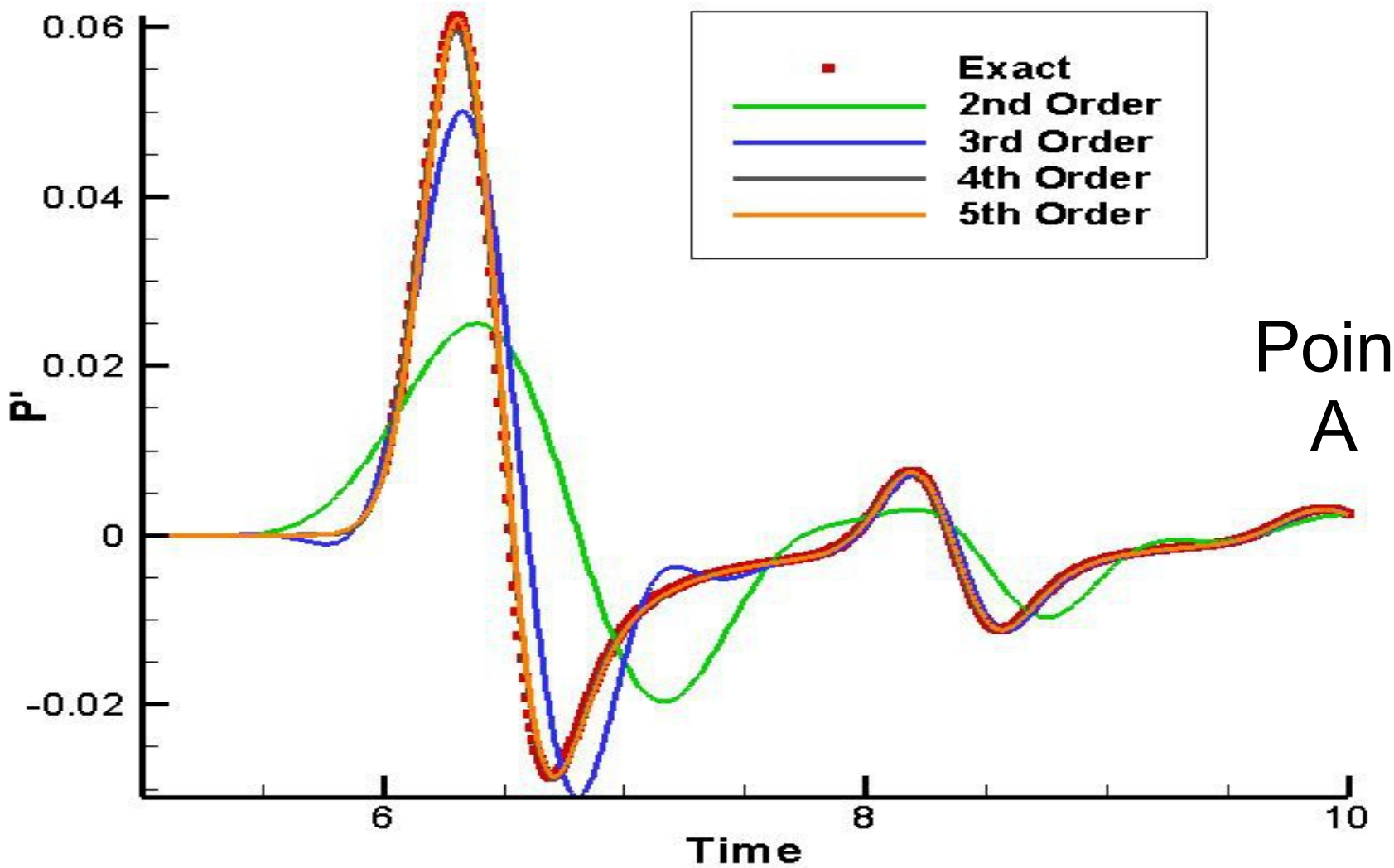
$$P = P_{inf} + \epsilon e^{\ln(-2)\left(\frac{x^2+y^2}{r0}\right)}$$

CUDA Validation

- Pressure taken at 3 points
  - Compare with analytic solutions

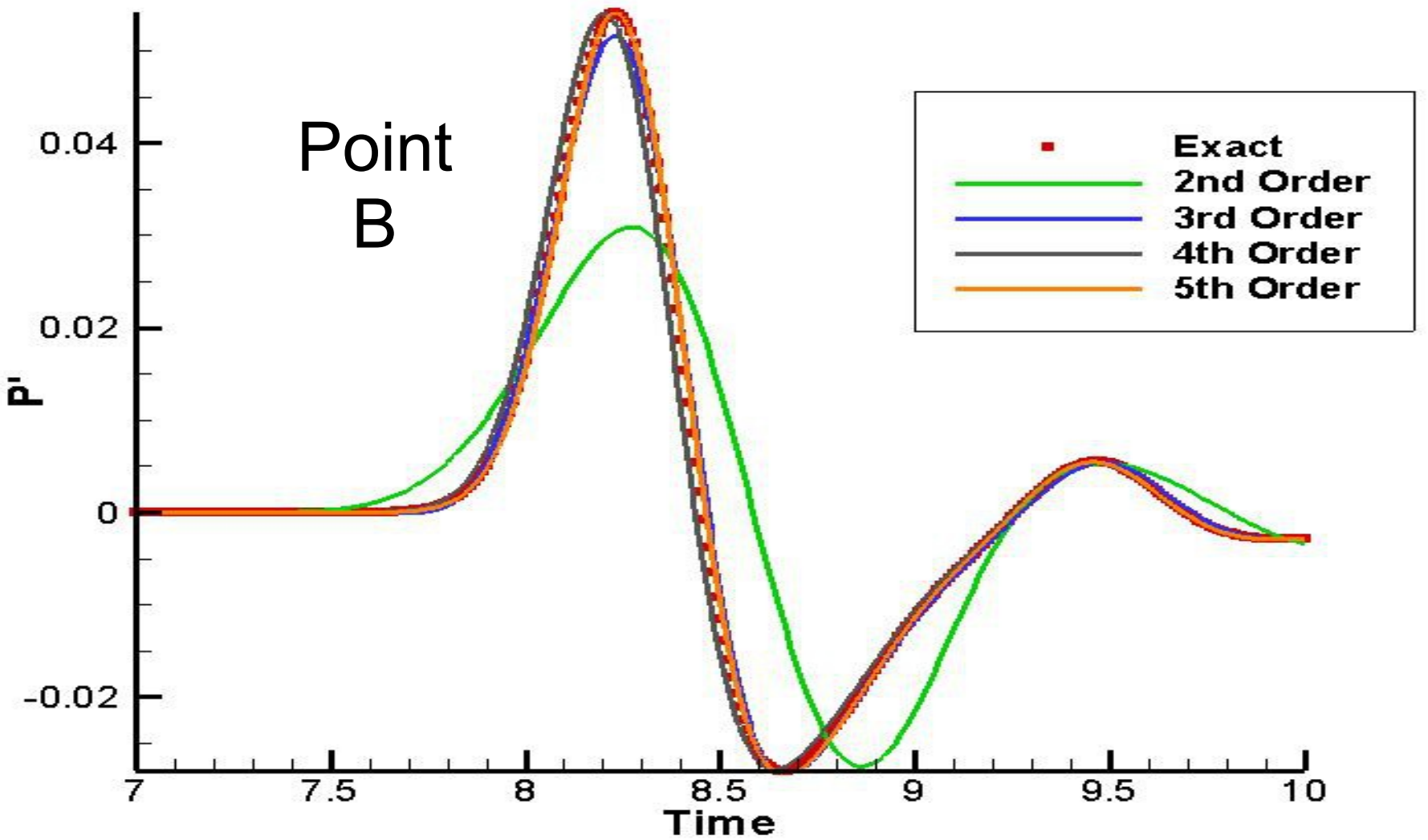


CUDA Validation

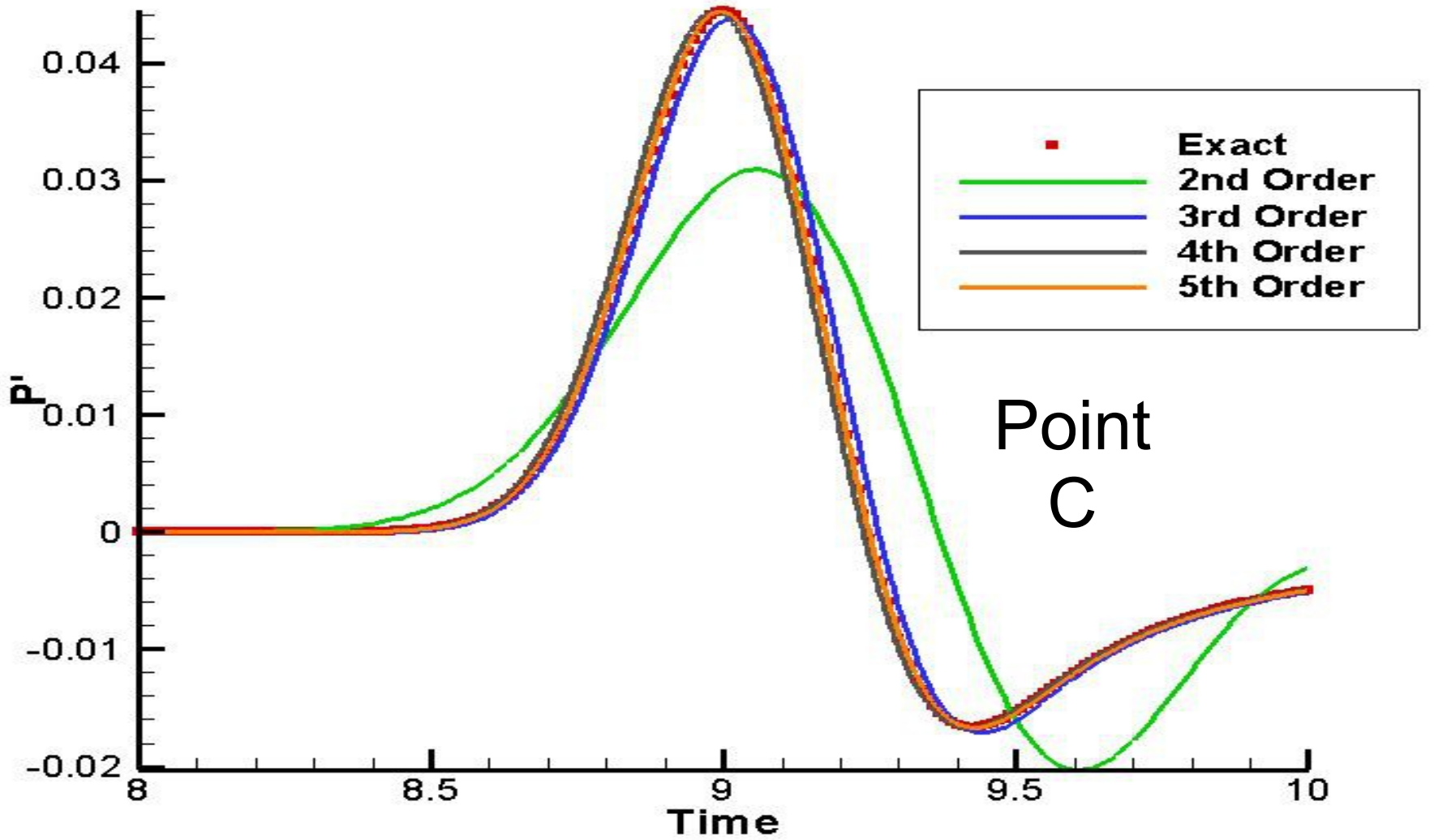


Point A

CUDA Validation

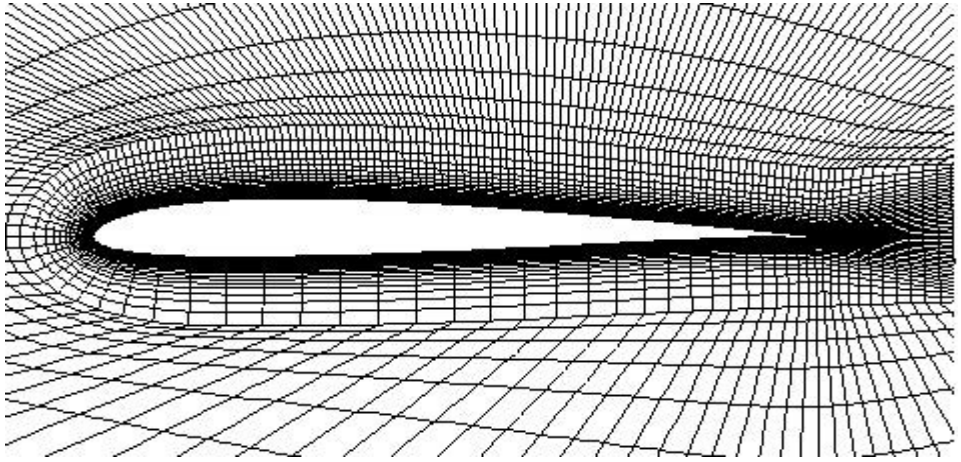


CUDA Validation



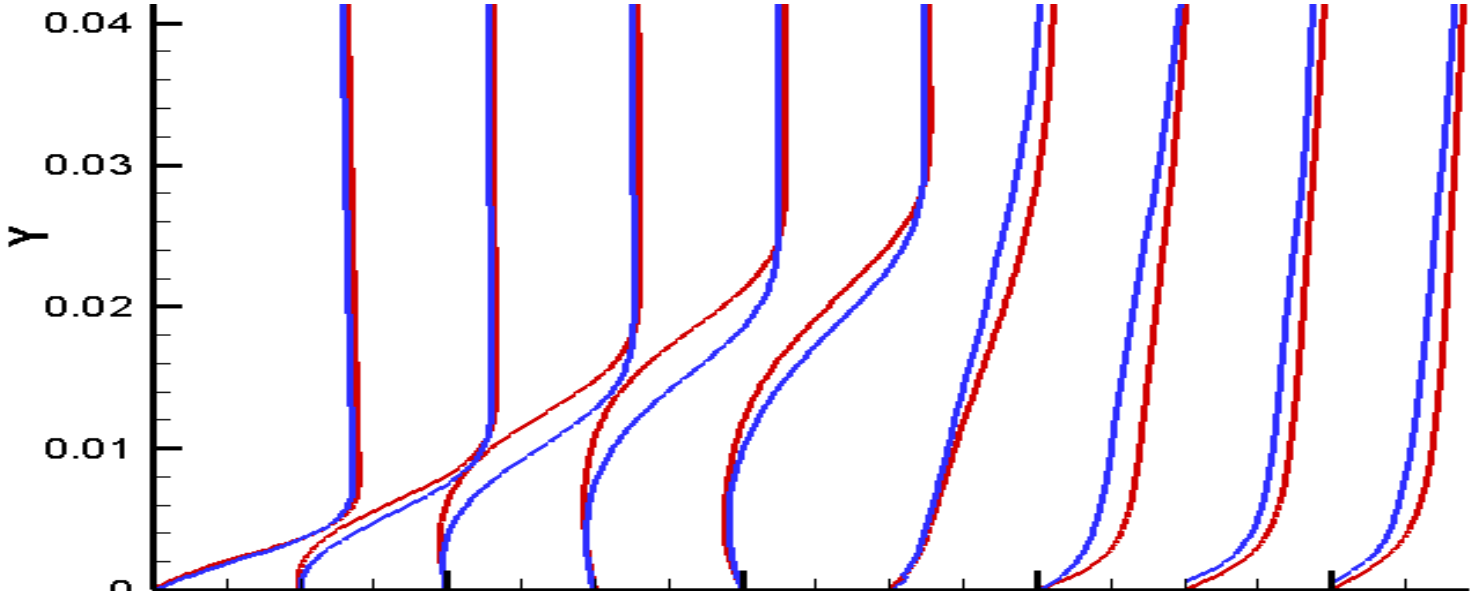
CUDA Validation

- SD7003 Wing
  - 60,000 Re
  - Mach 0.1 at 4 degree AoA
  - Comparison with FDL3DI
  - 293,590 cells



Order	# DOF
2nd	2,348,720
3rd	7,926,930

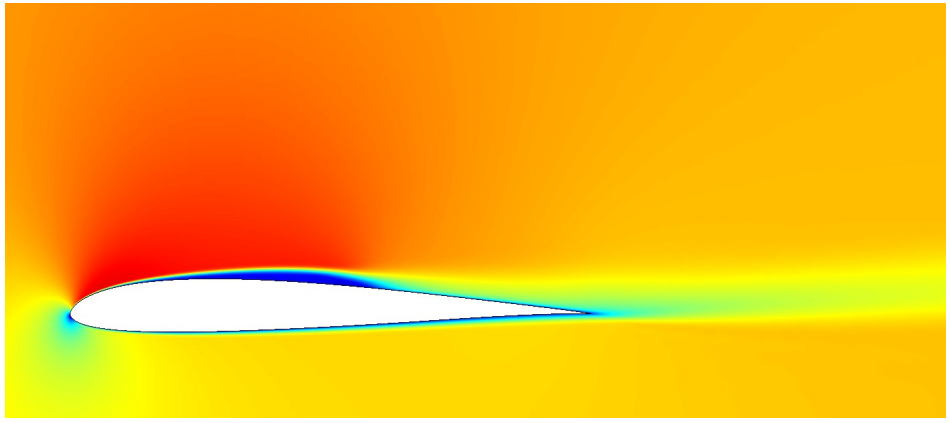
CUDA Validation

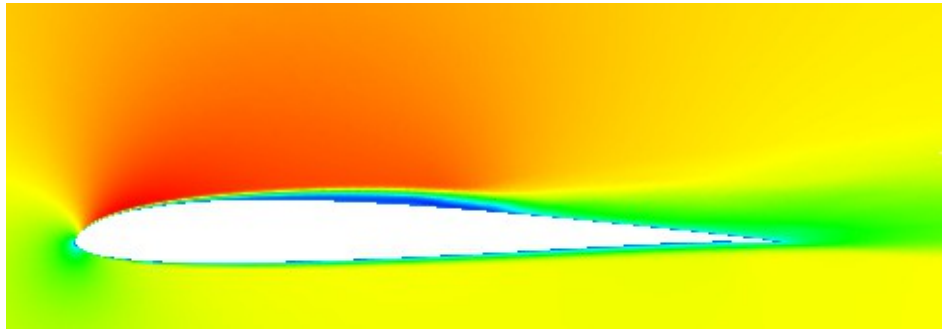


Mean U velocity profile (Data at every 0.1c)

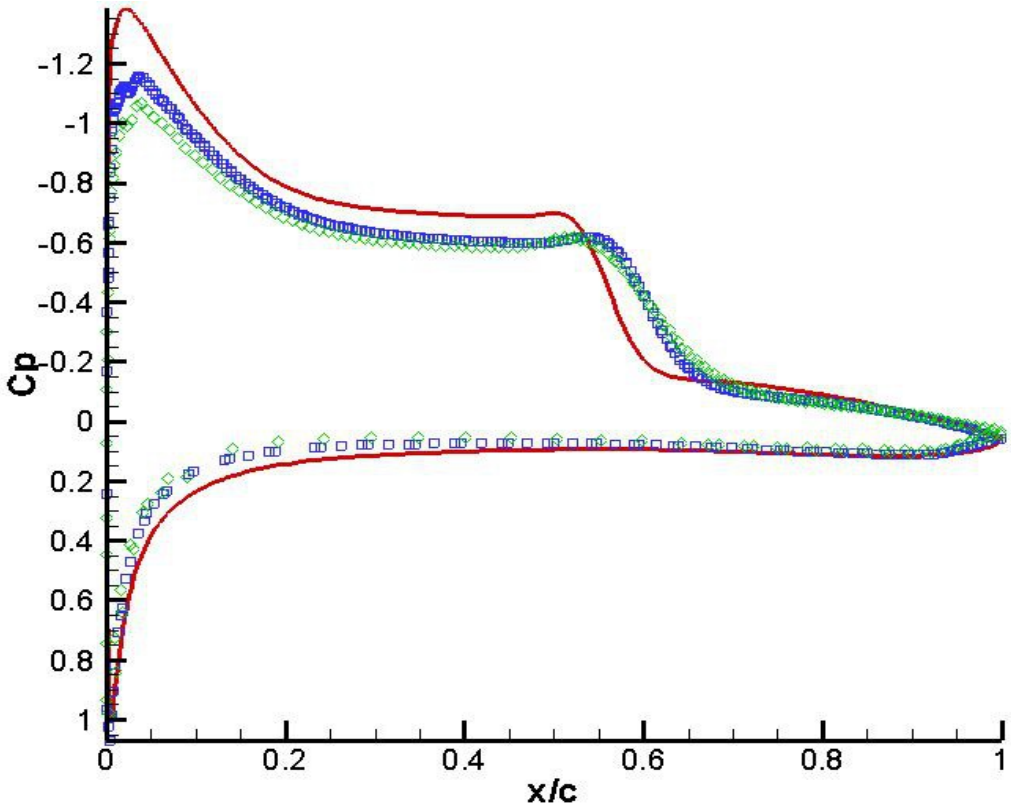
FDL3DI

SD3DN

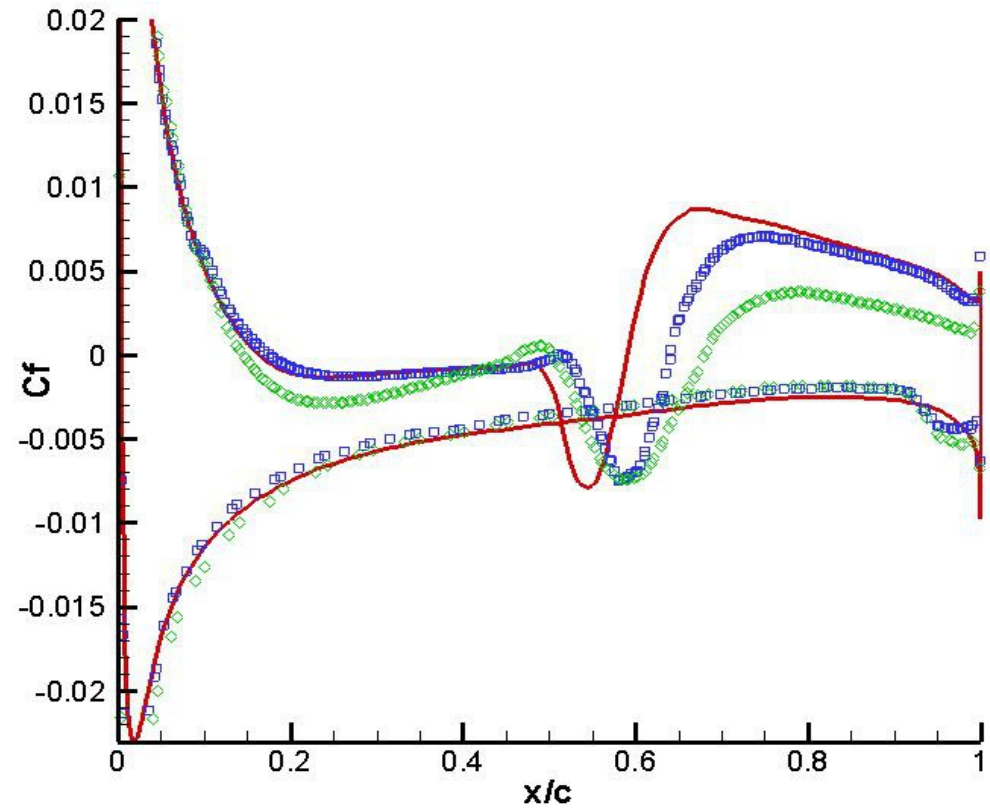




CUDA Validation



Mean Pressure Coefficient

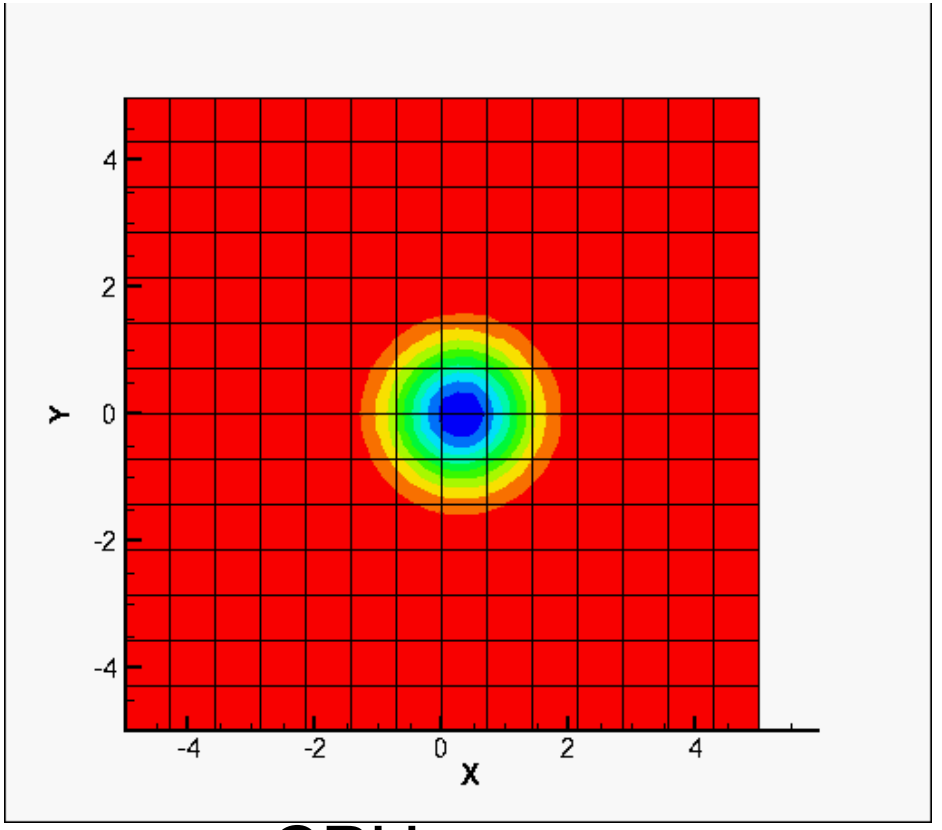


Mean Friction Coefficient

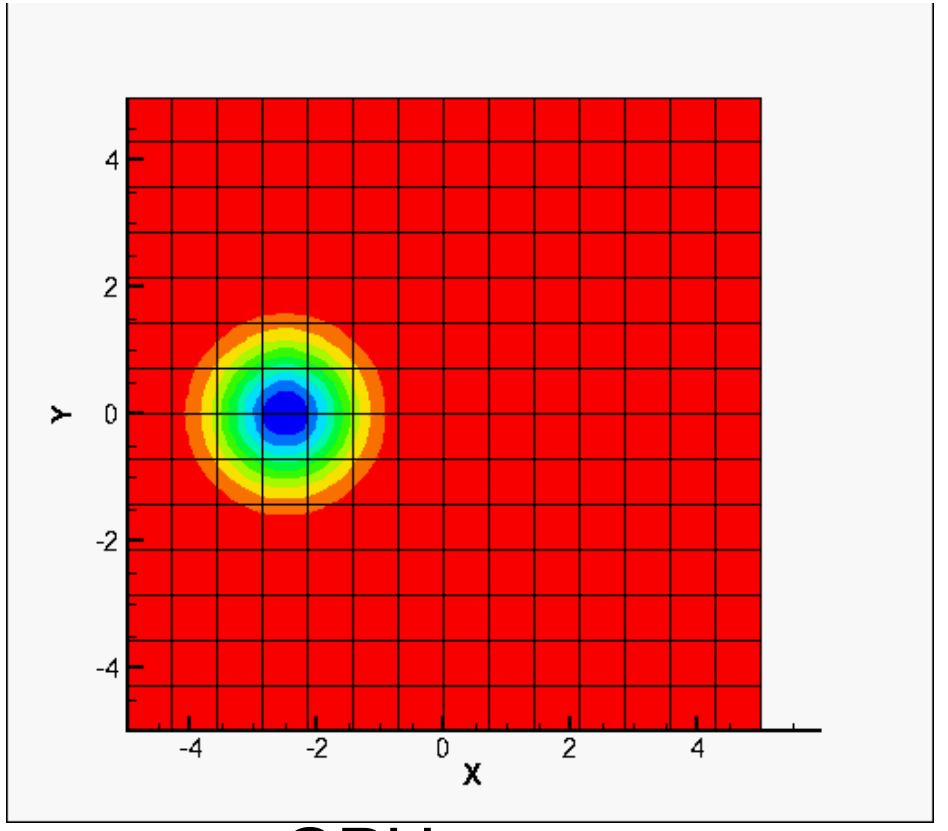
CUDA Acceleration

- CUDA is fast
  - 10x, 20x, 30x, 40x
- One-to-one
  - # GPU's = # CPU's
  - Use all resources
    - All cores on CPU
- Case I
  - Propagating vortex
- CPU (8 cores)
  - Xeon 2.27GHz
- GPU (240 cores)
  - Tesla C1060

CUDA Acceleration



CPU run

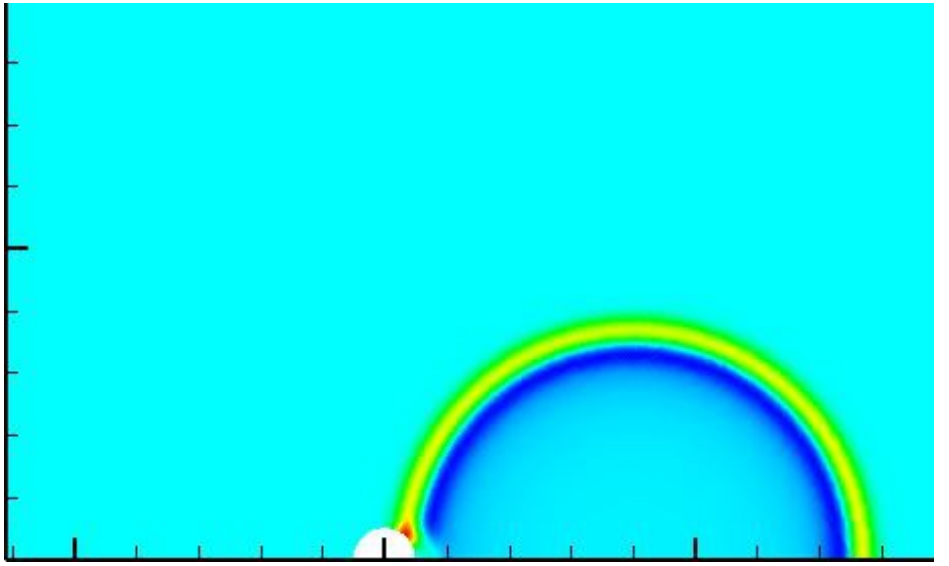


GPU run

- 5th order (28,125 DOF)
  - Roughly 13x faster

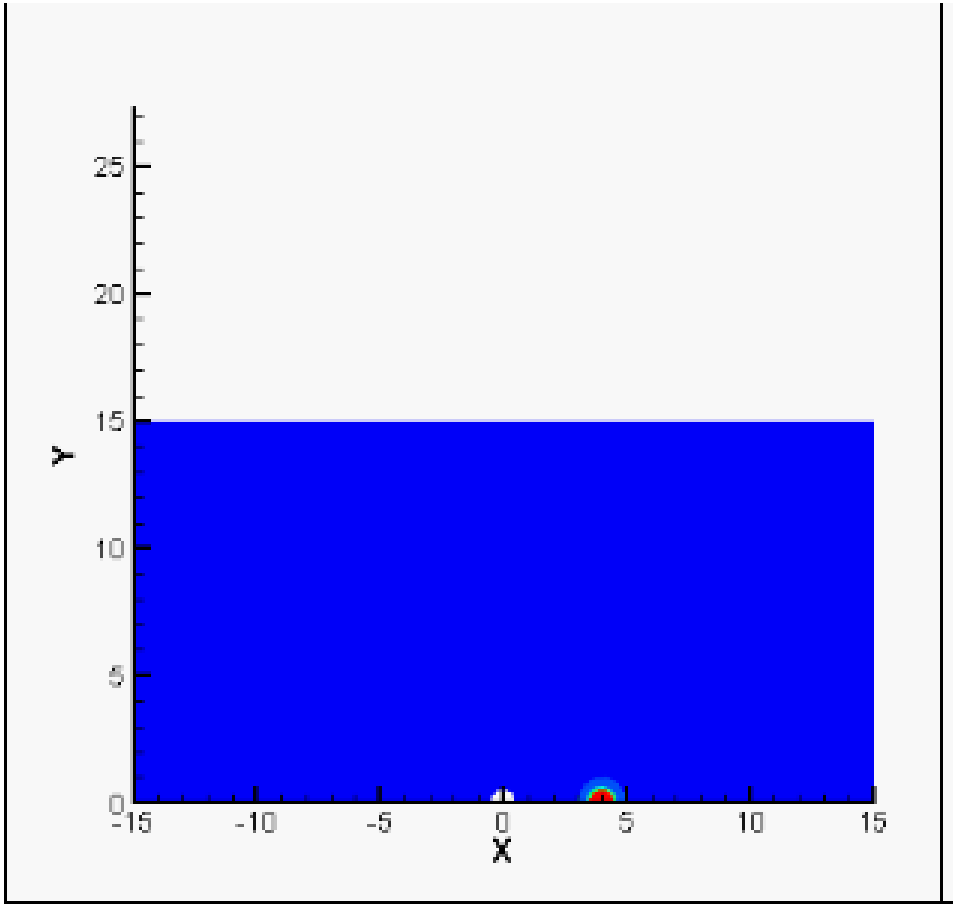
CUDA Acceleration

- Case II
  - Acoustic Pulse
- CPU
  - Xeon 2.27GHz
  - 5th order → 3.55 hrs
- GPU
  - GTX 550TI
    - 192 CUDA Cores
  - Tesla C2075
    - 448 CUDA Cores

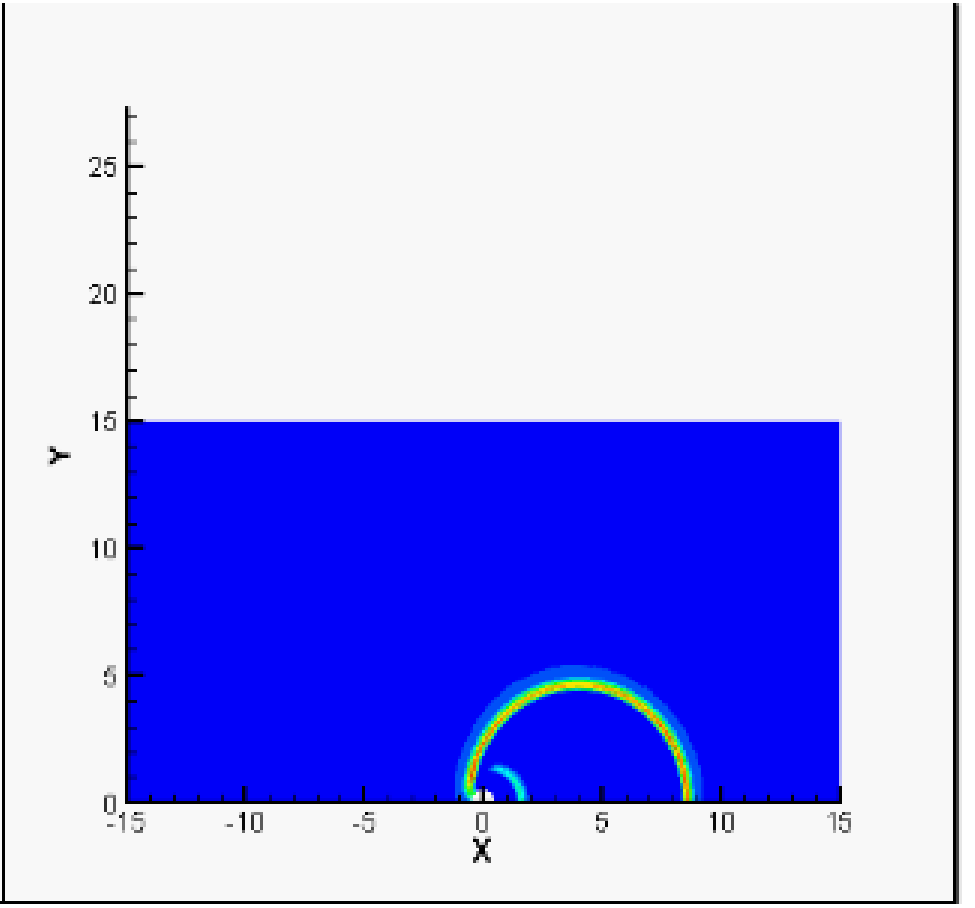


CUDA Acceleration

- 4th order



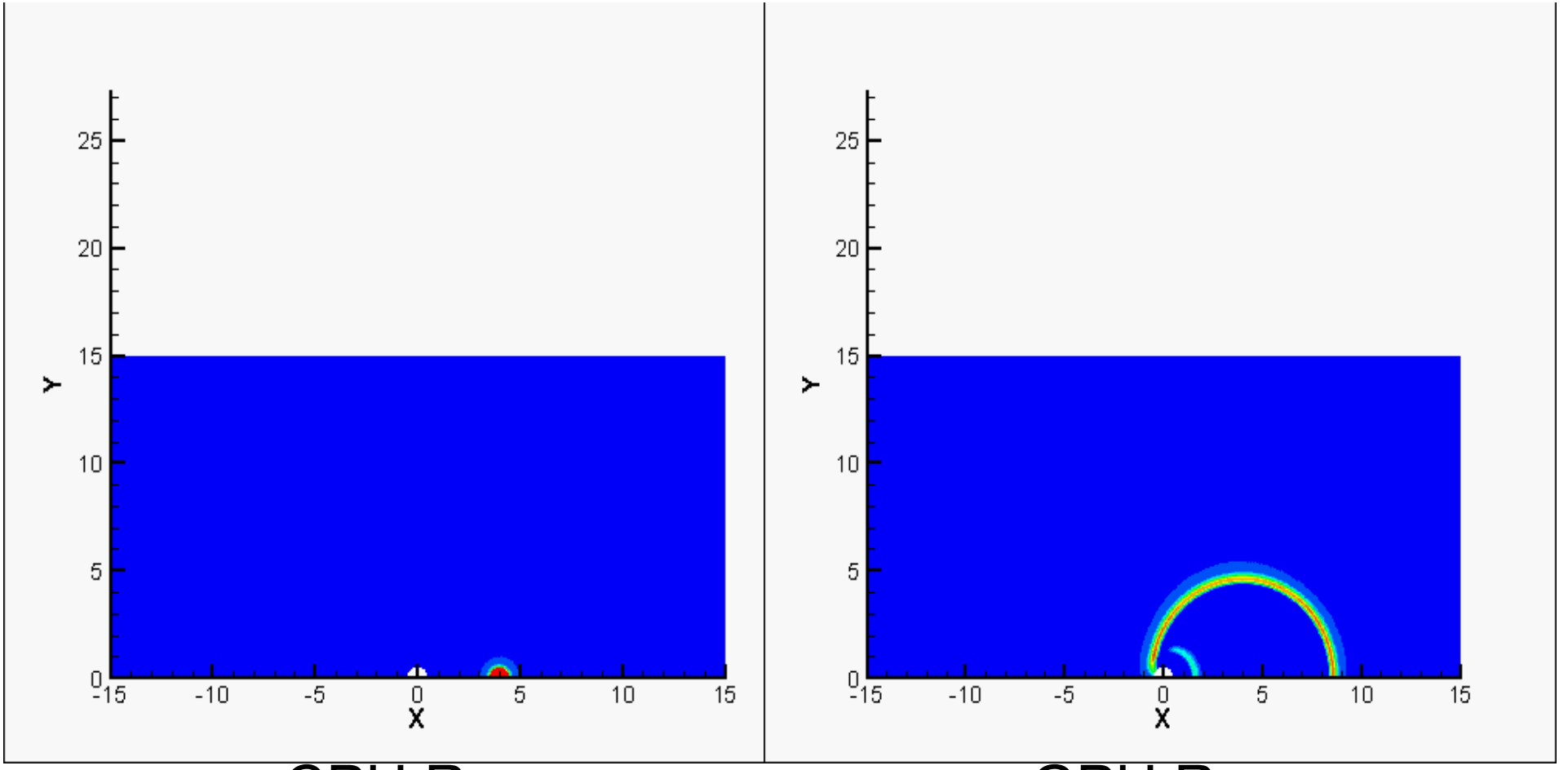
CPU Run



GPU Run

CUDA Acceleration

- 4th order



CPU Run GPU Run

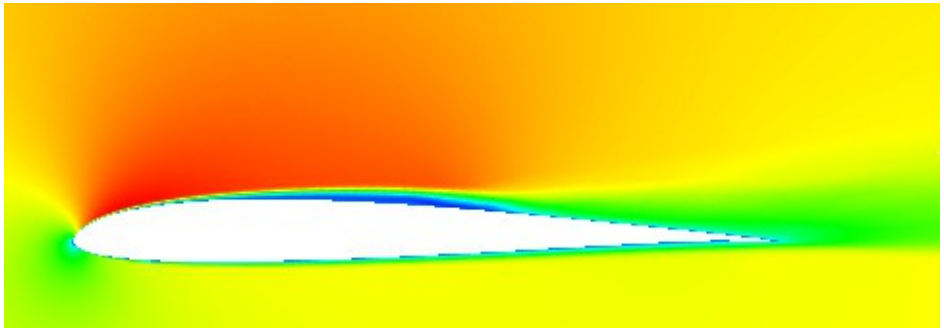
CUDA Acceleration

Type/Order	2nd Order	3rd Order	4th Order	5th Order
GPU (GTX)	0.0082766	0.018596	0.041478	0.07455
GPU (Tesla)	0.0083267	0.019119	0.0321213	0.066518
CPU (8 Cores)	0.205342	0.57619	1.23375	2.34552
Speed Up (GTX)	24.81 x	30.98 x	29.74 x	31.46 x
Speed Up (Tesla)	24.66 x	30.14 x	38.41 x	35.26 x

*Timings in seconds / iteration

CUDA Acceleration

- Case III
  - SD7003 airfoil
- CPU
  - Xeon 2.27GHz
  - 4 CPUs → 32 cores
- GPU
  - 4 Tesla C2075



Mean U Velocity 3rd Order

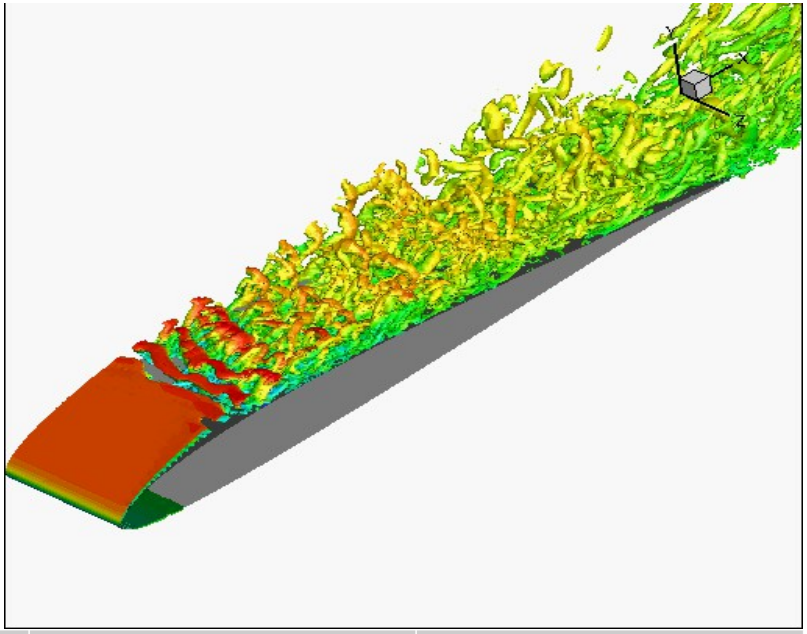
Order	# DOF
2nd	2,348,720
3rd	7,926,930

CUDA Acceleration

Q-Criterion

4 NVIDIA Tesla 3rd order

8 degree AoA



Order	Required Steps	CPU Total Time	GPU Total Time	Speed Up
2nd Order	400,000	607.78 hrs (25 days)	27.86 hrs	21.81x
3rd Order	800,000	3483.29 hrs (145 days)	137.14 hrs (~6 days)	25.4x

Conclusions

- GPU CFD Computing
  - Disadvantages
    - Limited by GPU memory
    - Rewrite from scratch → Best performance
    - Copy data to GPU is slow
  - Advantages
    - Huge increase in processing power over CPUs
    - Can out-preform CPU servers
      - Saving time, space, and money
    - Compiler flags to optimize CPU codes for GPU

Questions?