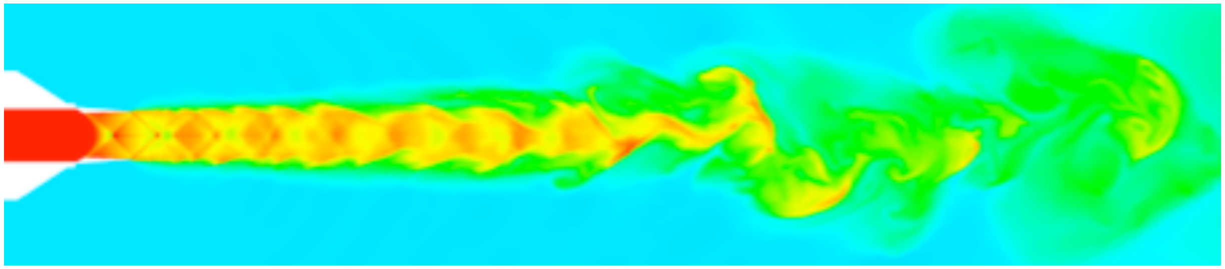
Unstructured Finite Volume Code on a Cluster with Multiple GPUs per Node



Keith Obenschain & Andrew Corrigan

Laboratory for Computational Physics and Fluid Dynamics
Naval Research Laboratory
Washington DC, 20375

Mayhem GPU Cluster Overview

- Division Machine
  - Balance both CPU and GPU requirements for all users
- Each node can be configured as prototype field-deployable system
  - Difficult to deploy a typical HPC machine in the field
  - Severely reduced power and cooling requirements
  - Single/multiple 4U nodes with multiple GPUs a potential solution

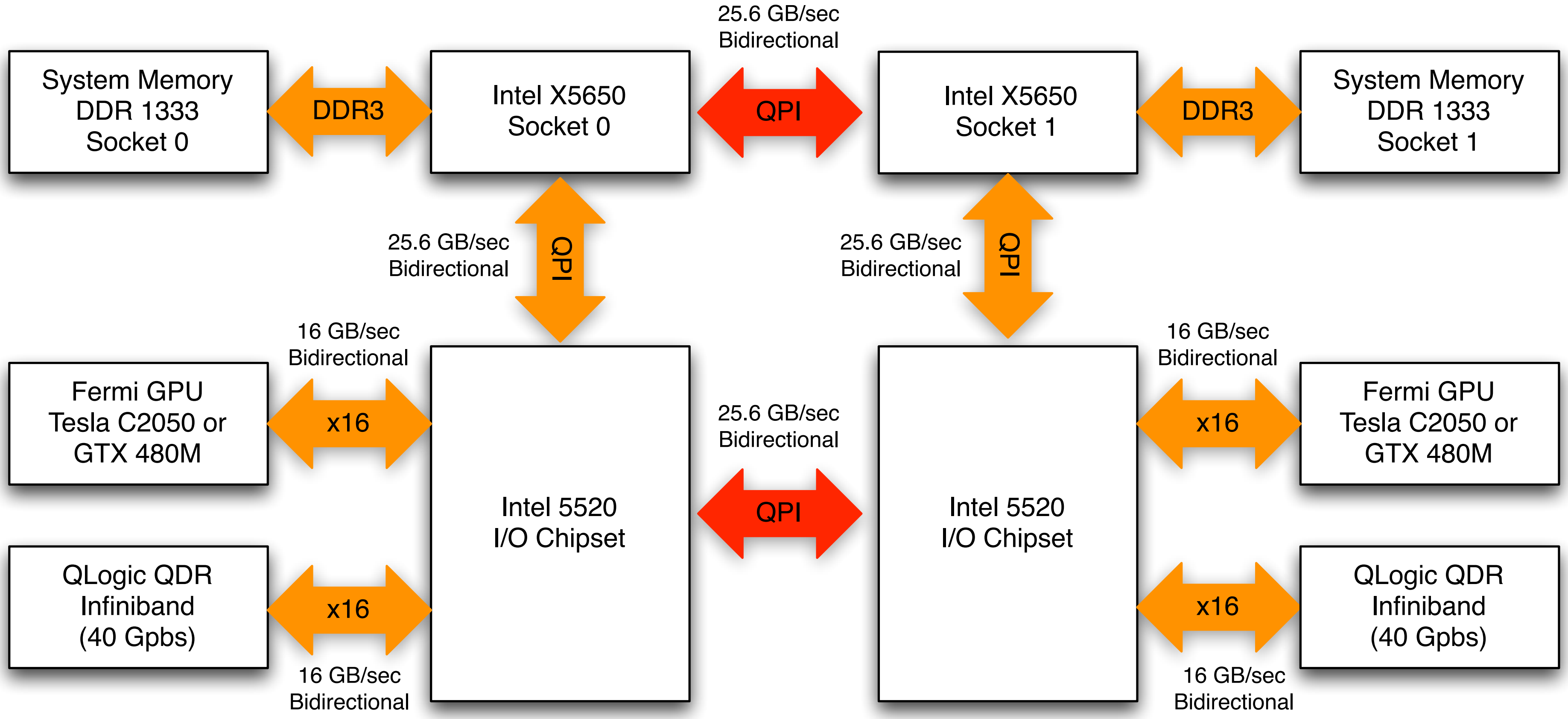
Mayhem GPU Cluster Overview

- Balanced System
  - Fast CPUs (Intel)
  - Fast IO internally
  - Multiple GPUs per compute node
  - Fast IO across compute nodes (40 Gbps Infiniband)
- Node Choices
  - Available chipsets have 36 PCI Express Lanes (2 x16 slots + 1 x4)
    - Choice of 1 GPU and 1 Infiniband Card or 2 GPUs with full bandwidth
    - Use PLX chip share PCI Express lanes
  - Multiple Chipset systems available

Mayhem Cluster Details

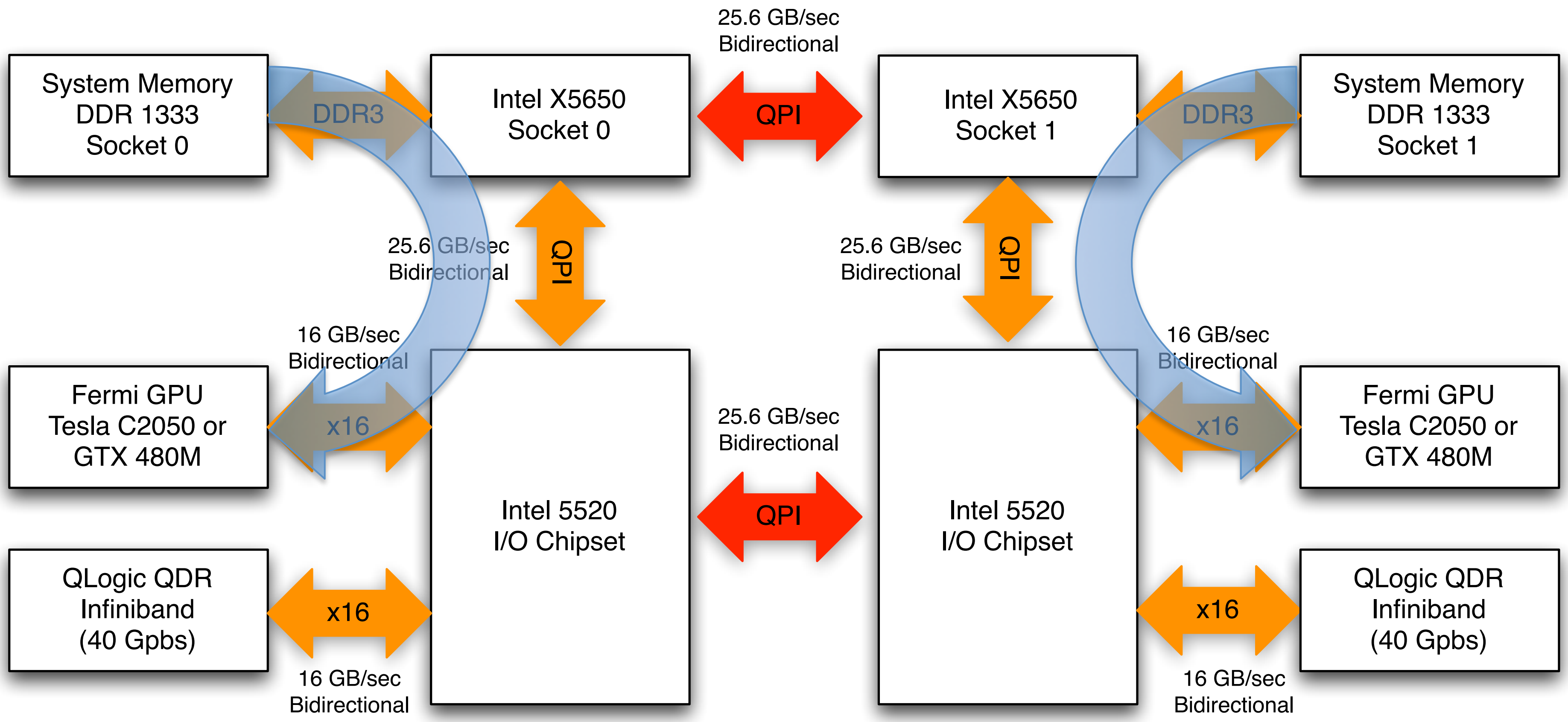
- 24 Compute Nodes
  - Super Micro X8DTG-QF Dual Socket Motherboard
  - 4 PCI Express 2.0 x16 Slots
  - 4U Chassis to accommodate a wide variety of GPUs and interconnects
  - Intel X5650 6 Cores @ 2.67 Ghz with 12 MB L3 Cache
  - 24 GB DDR 1333 memory
  - 2 GPUs (2 Tesla C2050 or 2 GTX 480 GPUs)
  - QLogic QDR Infiniband
    - 1 Infiniband card in GTX 480 Nodes
    - 2 Infiniband cards in Tesla Nodes
  - 36 Port QLogic QDR Switch
  - Centos 5.5 (Diskless)
  - CUDA 3.1

Motherboard Interconnect Overview



- NUMA architecture
- Two Intel 5520 Chipsets provide 4 x16 PCI Express 2.0 Slots
- Non-Uniform Access to GPUs/Chipsets
- Must go through an additional QPI link if GPU is not associated with the processor’s chipset

Optimal communication



- Processor and memory affinity should be enabled
- Process started on a CPU should select a GPU connected to the CPU’s chipset
- Before we optimize performance across nodes, we need to optimize one node first
- **Is this really important?**

Host<->Device Bandwidth Tests

- Measure transfer rate of data between CPU and GPU memory
  - *N* CPU to GPU memory transfers followed by *N* GPU to CPU transfers
  - CPU to GPU memory transfer followed by a GPU to CPU transfer repeated *N* times
- Bandwidth test developed for GPU Direct modified to run on multiple GPU's
  - Using Pinned Memory
- OpenMPI's processor/memory/socket affinity utilized to configure optimal and non-optimal memory/GPU selection
- Tested with/without socket affinity and non-optimal configurations
- Tested with a node configured with 4 Tesla C2050 GPUs

NRL

Setting CPU Affinity under Linux

- OpenMPI has a rich set of options to set affinity
  - Ability to set memory/core/socket affinity

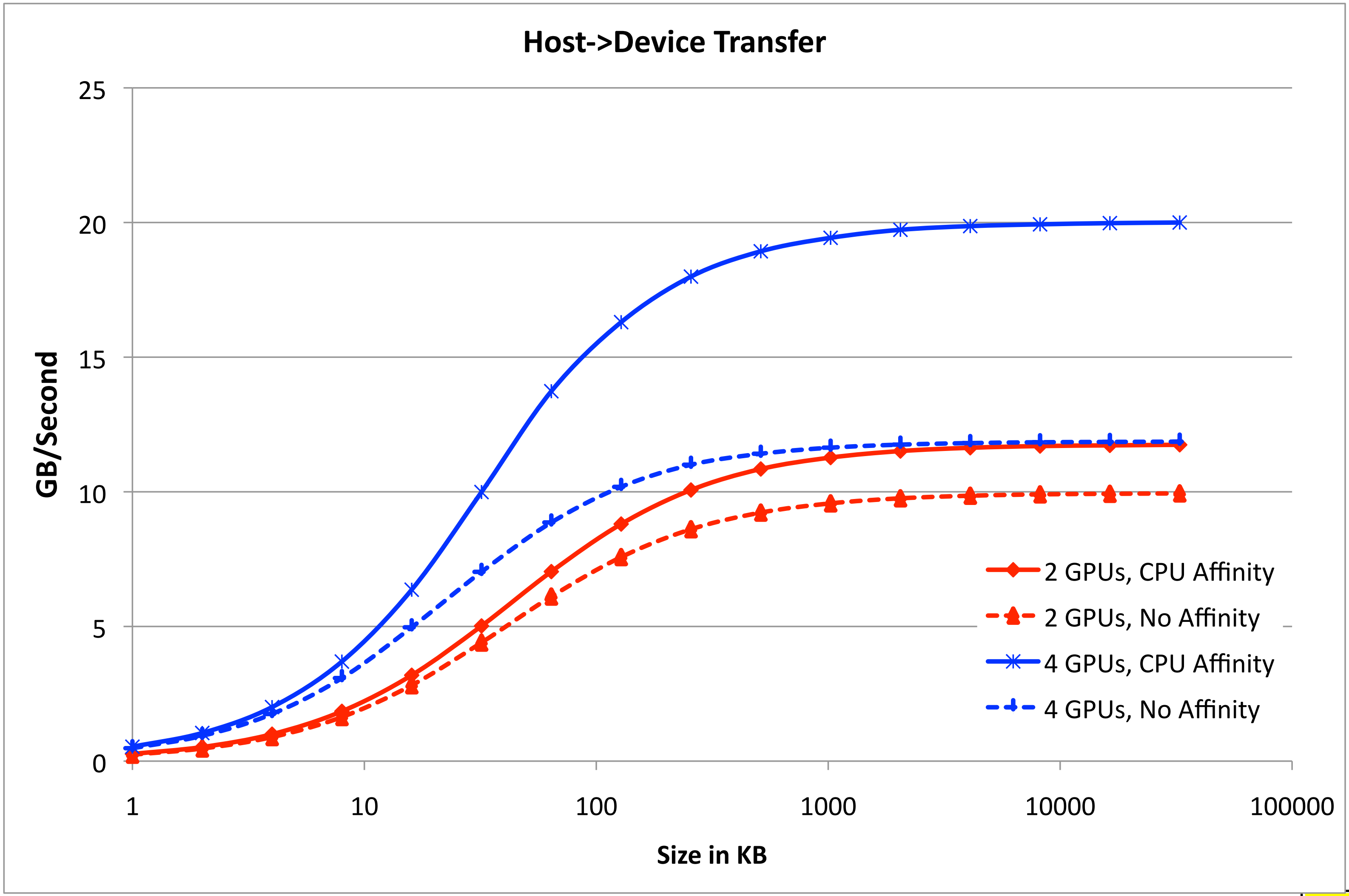
Example:

```
mpirun --mca mpi_paffinity_alone 1
-bind-to-core –npersocket 1 –np 4
command
```

- Use “taskset” to set CPU affinity for single processor tasks

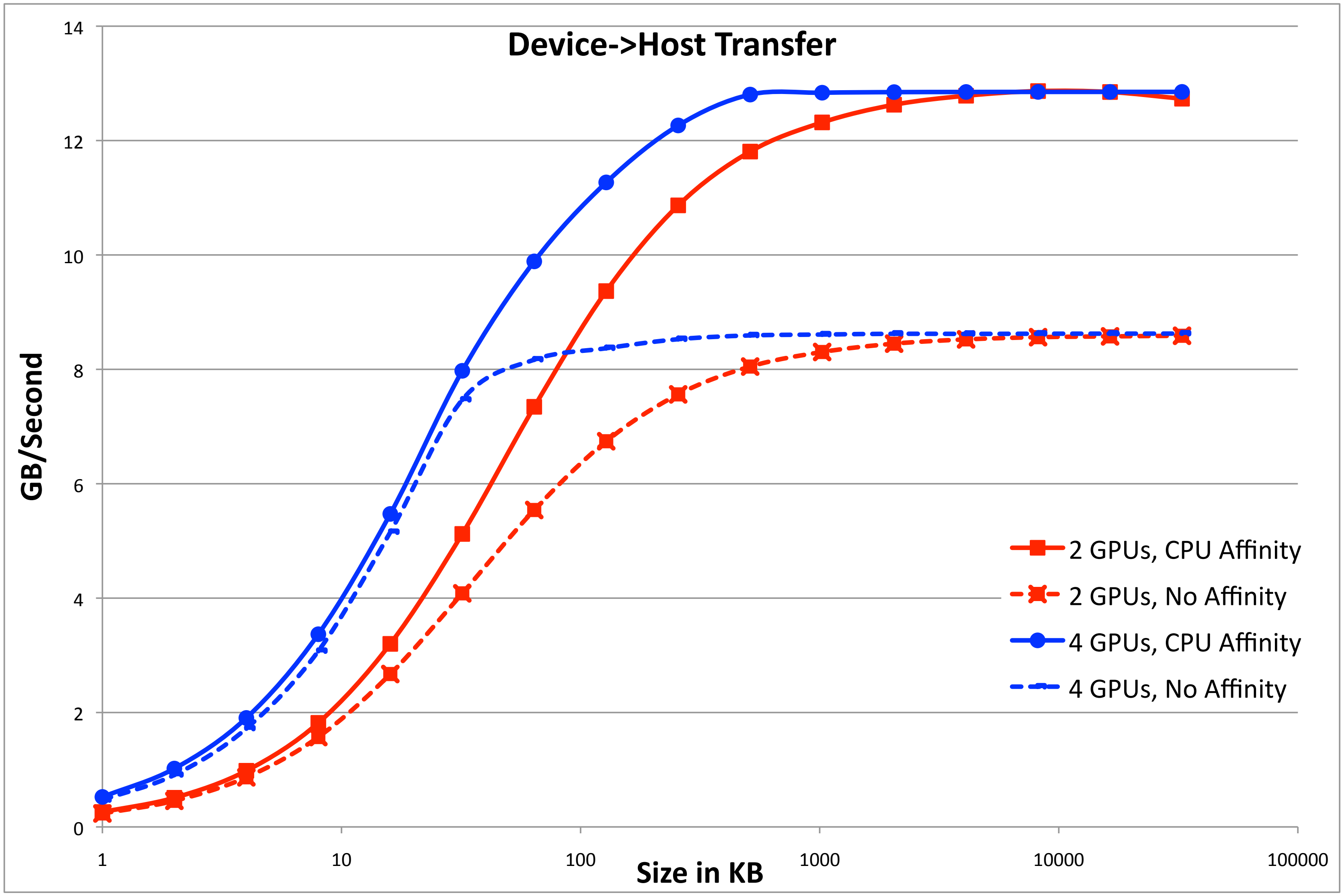
LCP &FD

Total Host to Device Bandwidth (CPU to GPU)



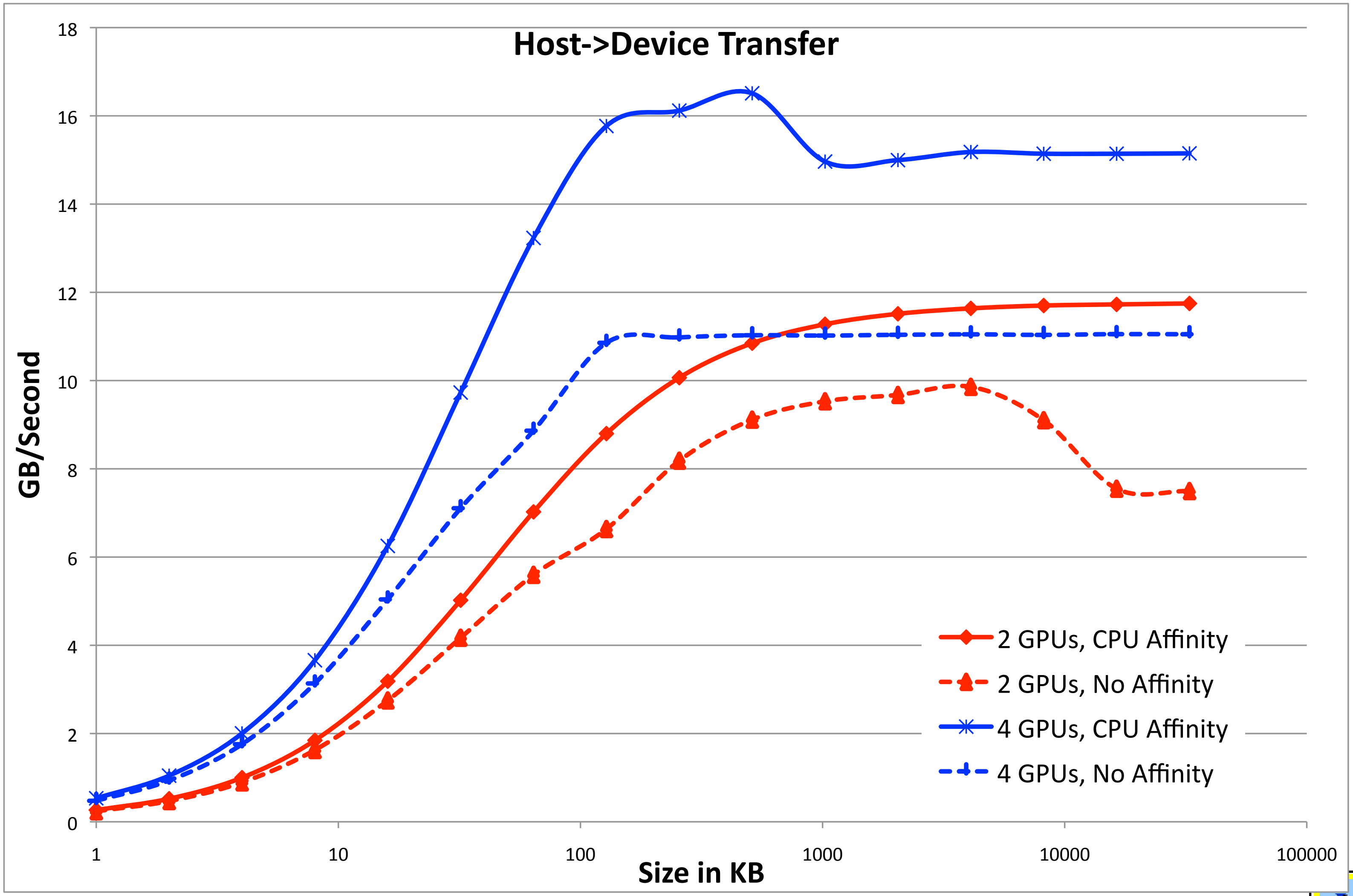
- Host to Device bandwidth scaling is reasonable if CPU affinity is enabled

NRL Total Device to Host Bandwidth (GPU to CPU)



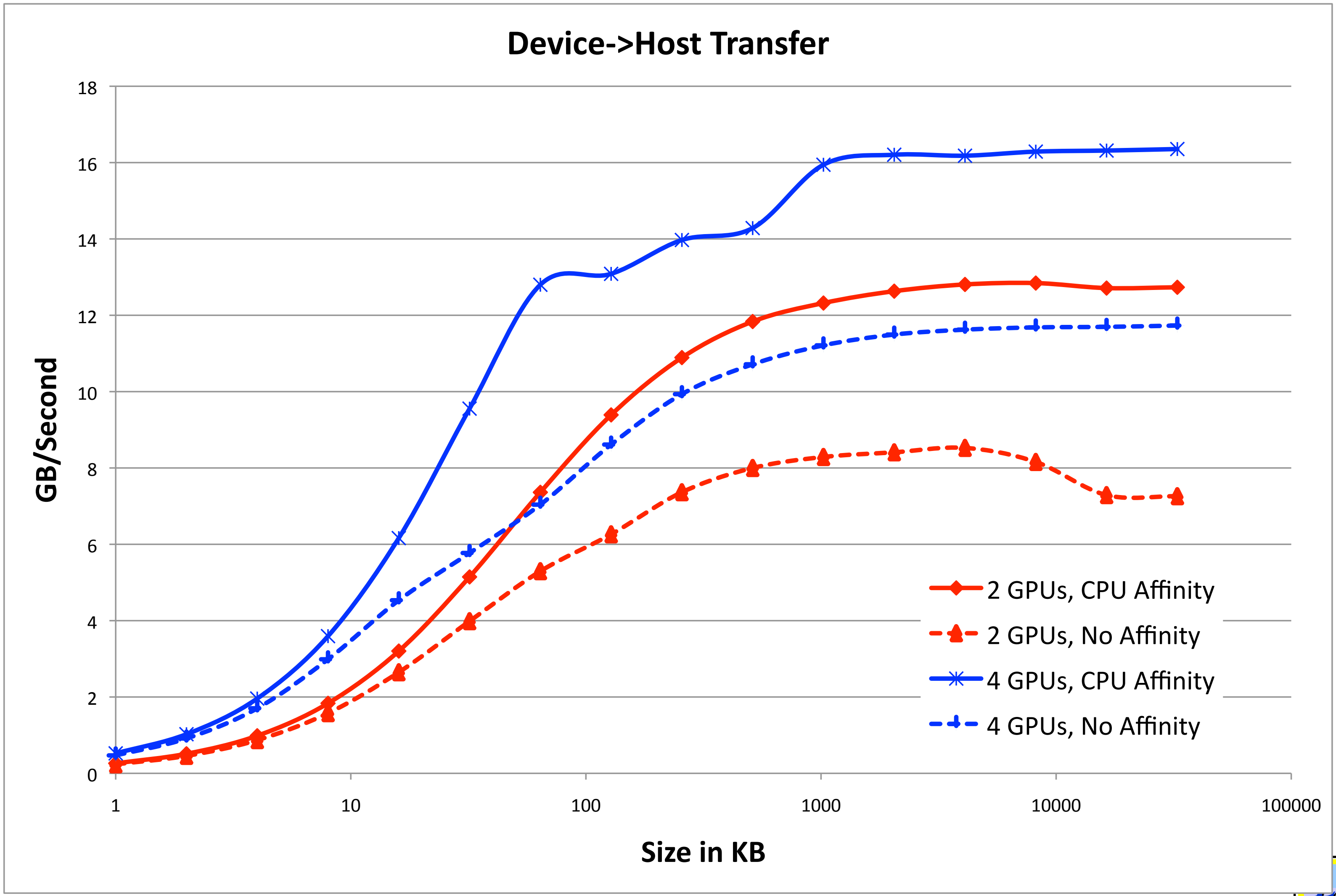
- For larger messages, GPU to CPU transfer becomes saturated even with one GPU

Total Device to Host Bandwidth Interleaved with CPU to GPU Transfer



- Overall performance less when CPU->GPU and GPU->CPU are interleaved

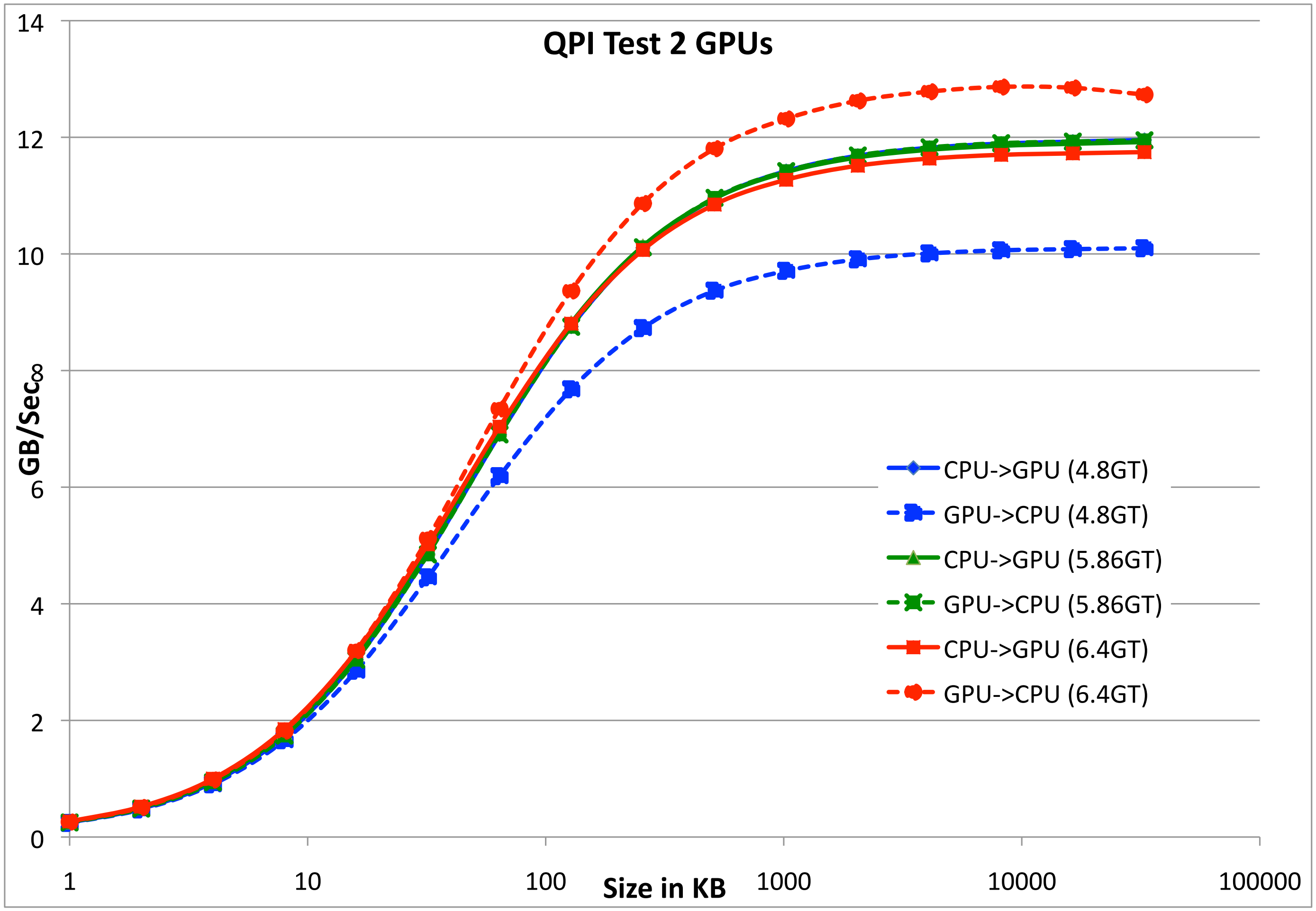
Total Host to Device Bandwidth Interleaved with GPU to CPU Transfer



QPI Bandwidth

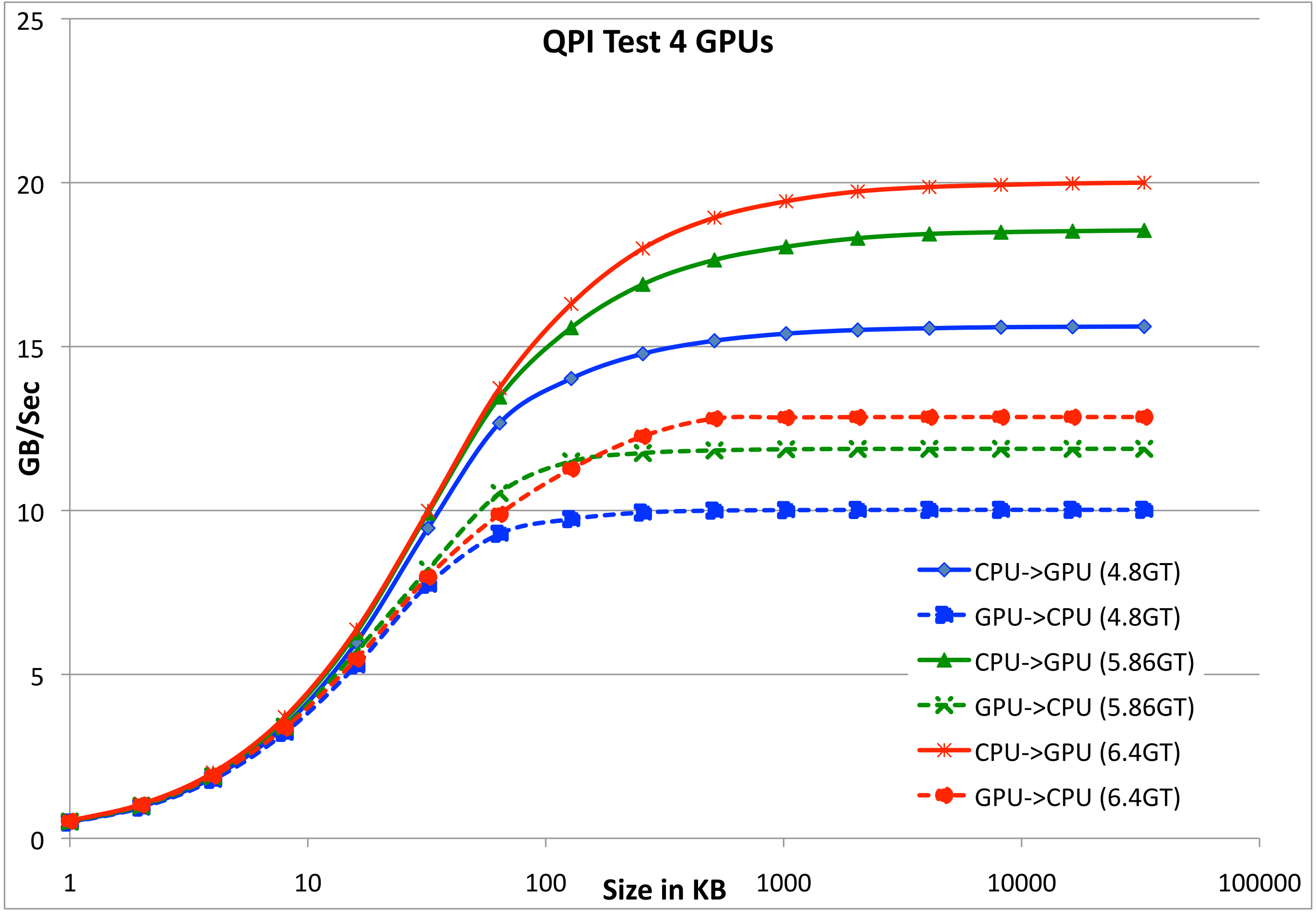
- Are the more expensive chips necessary?
- Tested with QPI Speeds of 4.8, 5.86 and 6.4 GT/sec
- *N* CPU to GPU memory transfers followed by *N* GPU to CPU transfers
- Test with 2 and 4 GPUs
- CPU Affinity Set

QPI Test: 2 GPUs



- CPU->GPU Tests almost identical
- For larger messages GPU->CPU QPI speed limited transfers

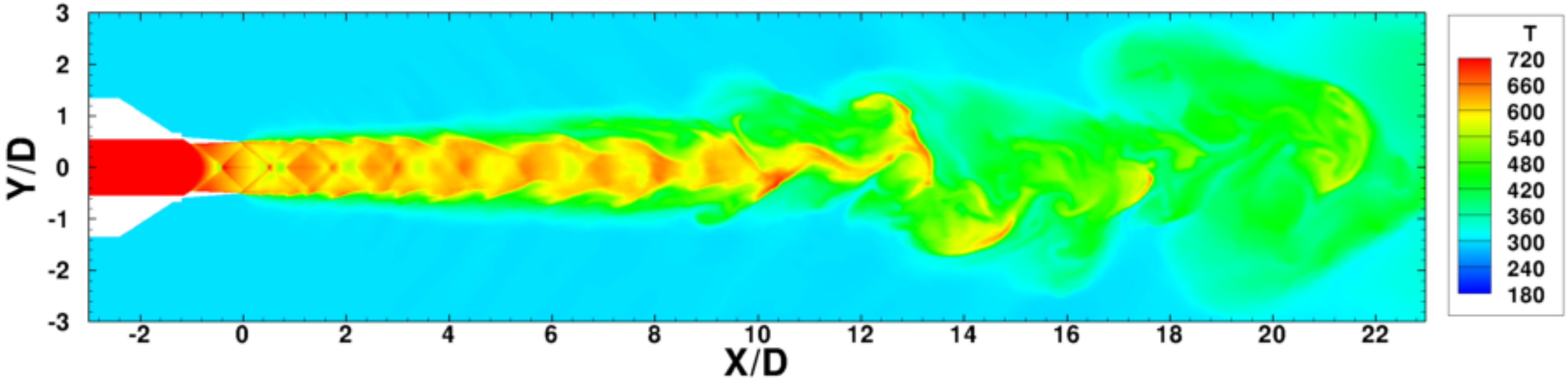
QPI Test: 4 GPUs



- Difference in QPI speed is approximately the same as measured difference in bandwidth for both CPU->GPU and GPU->CPU transfers

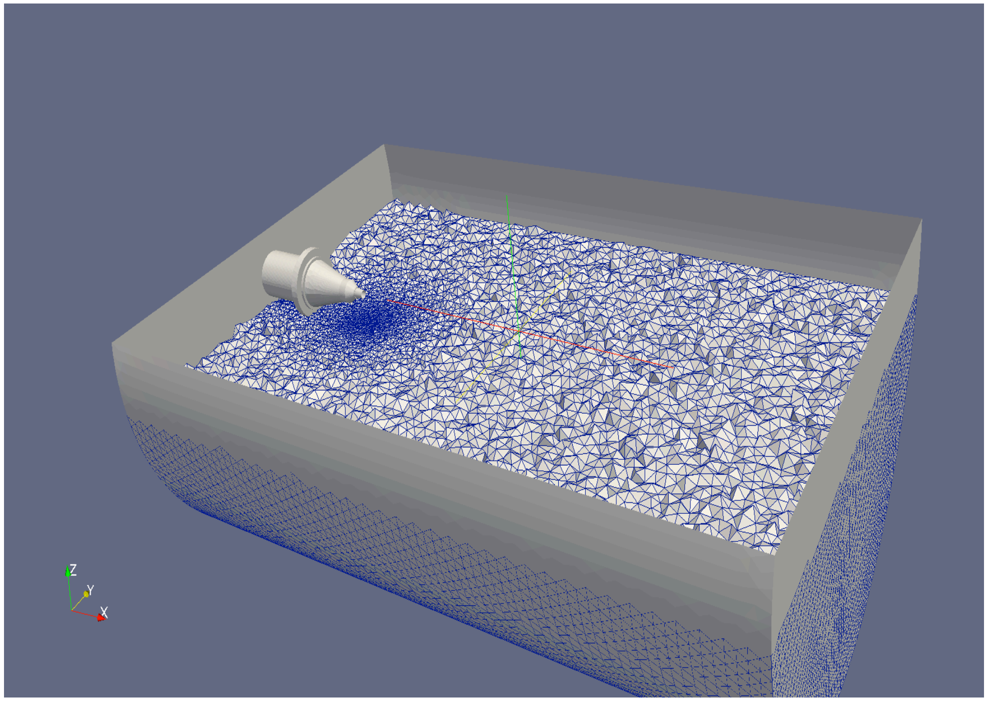
TBD – A New CFD Solver

- A new CFD solver is under development at NRL for supersonic jet noise and other high-speed compressible flow problems.



Solver Features

- Solves the Euler equations for inviscid, compressible flow.
- The equations are discretized using the finite volume method.
- Computes fluxes using *Flux-Corrected Transport* to achieve monotonicity while limiting excessive diffusion
- Extensible to other physics and numerical methods.



Solver Features

- Implemented in C++ using templates.
- Operations are performed using standard algorithms provided by either:
  - The ***C++ Standard Library*** and ***Boost*** to allow for portable C++ code.
  - ***Thrust*** to achieve GPU and multi-core CPU parallelization
  - This allows for a unified GPU-CPU codebase.
- Uses ***MPI*** to achieve multi-CPU/GPU parallelization.

Solver Features

- Arbitrary mesh geometry needed to support complex jet nozzle geometry
  - Arbitrary face shapes
  - Arbitrary cell shapes: tetrahedral, hexahedral, pyramids, etc.

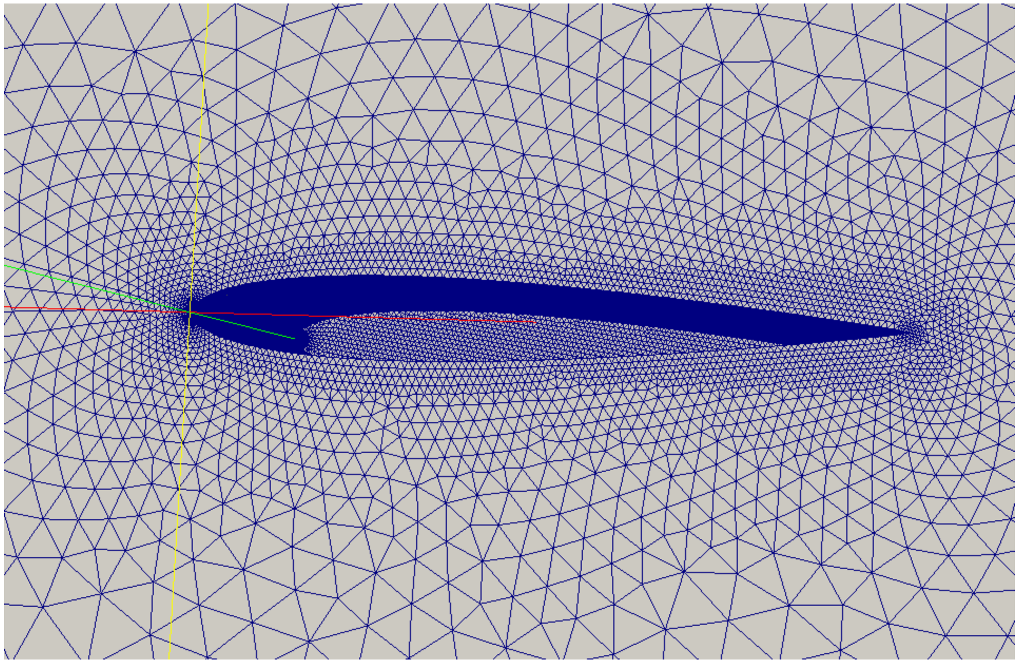
Arbitrary mesh topology

Unstructured grids (to allow for complex geometry).

Structured grids (to reduce storage and bandwidth usage).

Finite Volume Discretization

- The domain geometry is discretized into cells, forming a mesh.



A NACA 0012 mesh

Finite Volume Discretization

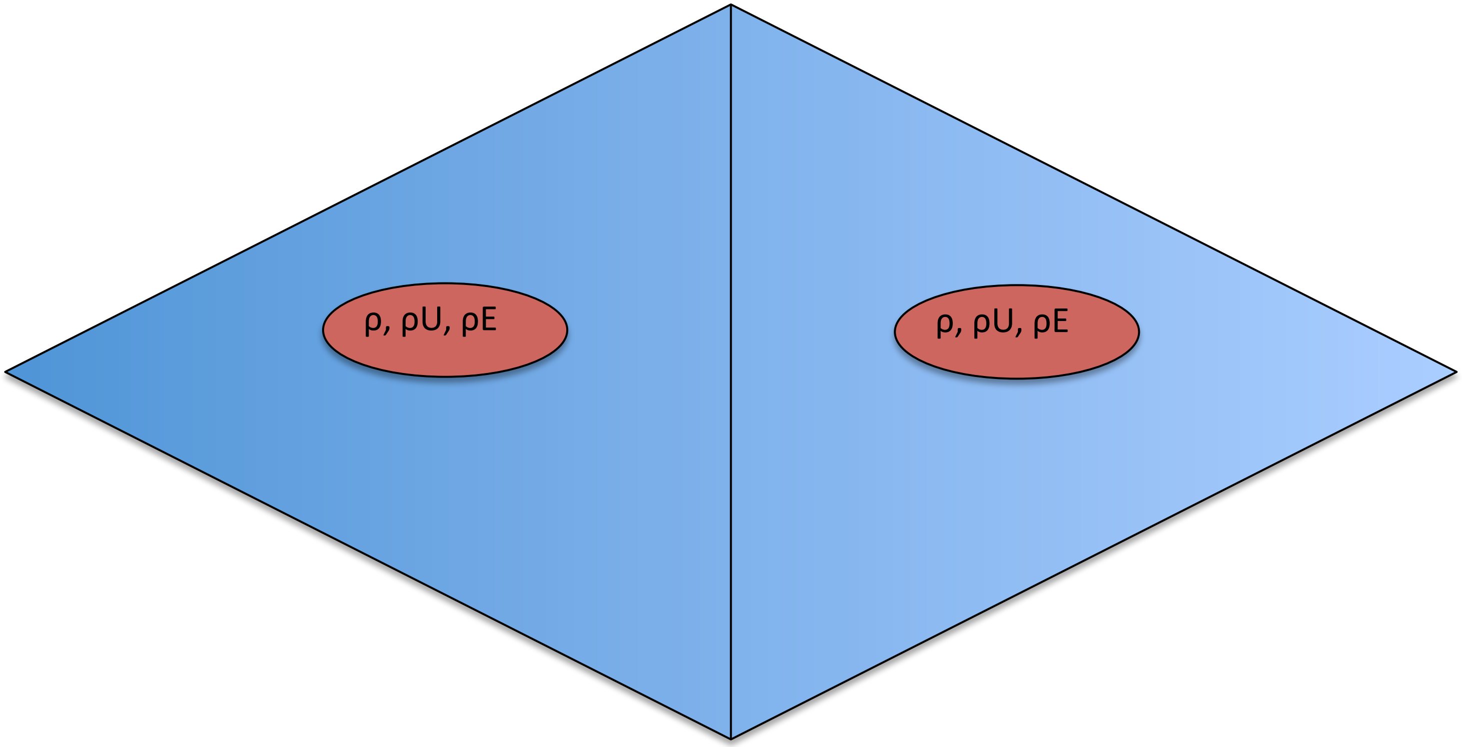
- The flow field is integrated in time by exchanging fluxes across faces, as governed by the Euler equations.

$$\frac{d}{dt}\int_{\Omega} \mathbf{u}\, d\Omega + \int_{\Gamma} \mathbf{F}\cdot\mathbf{n}\, d\Gamma = 0$$

$$\frac{d\mathbf{u}_{el}}{dt} = \frac{1}{V_{el}}\sum_{faces} \mathbf{F}_{face}\cdot\mathbf{s}$$

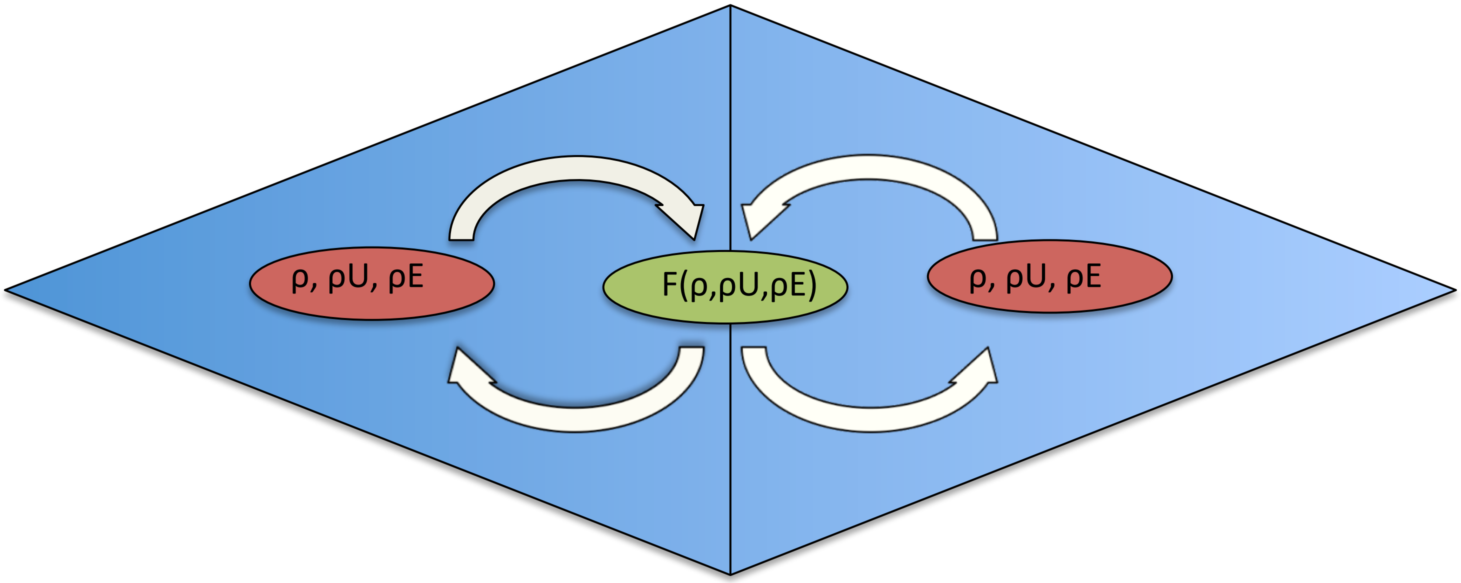
Implementation

- Conserved flow quantities are stored at the cell centers of the mesh.



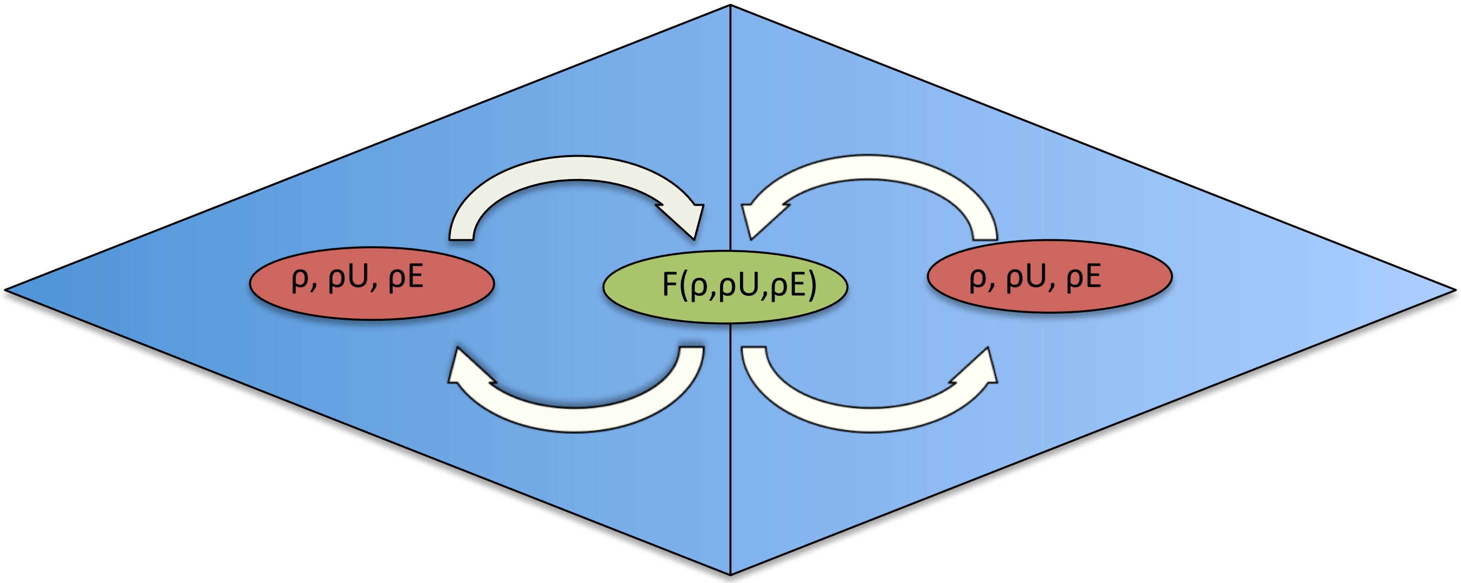
Implementation

- Flow quantities are interpolated from the cells onto the faces.
- A flux is computed and integrated and the conserved variables are updated.



Implementation

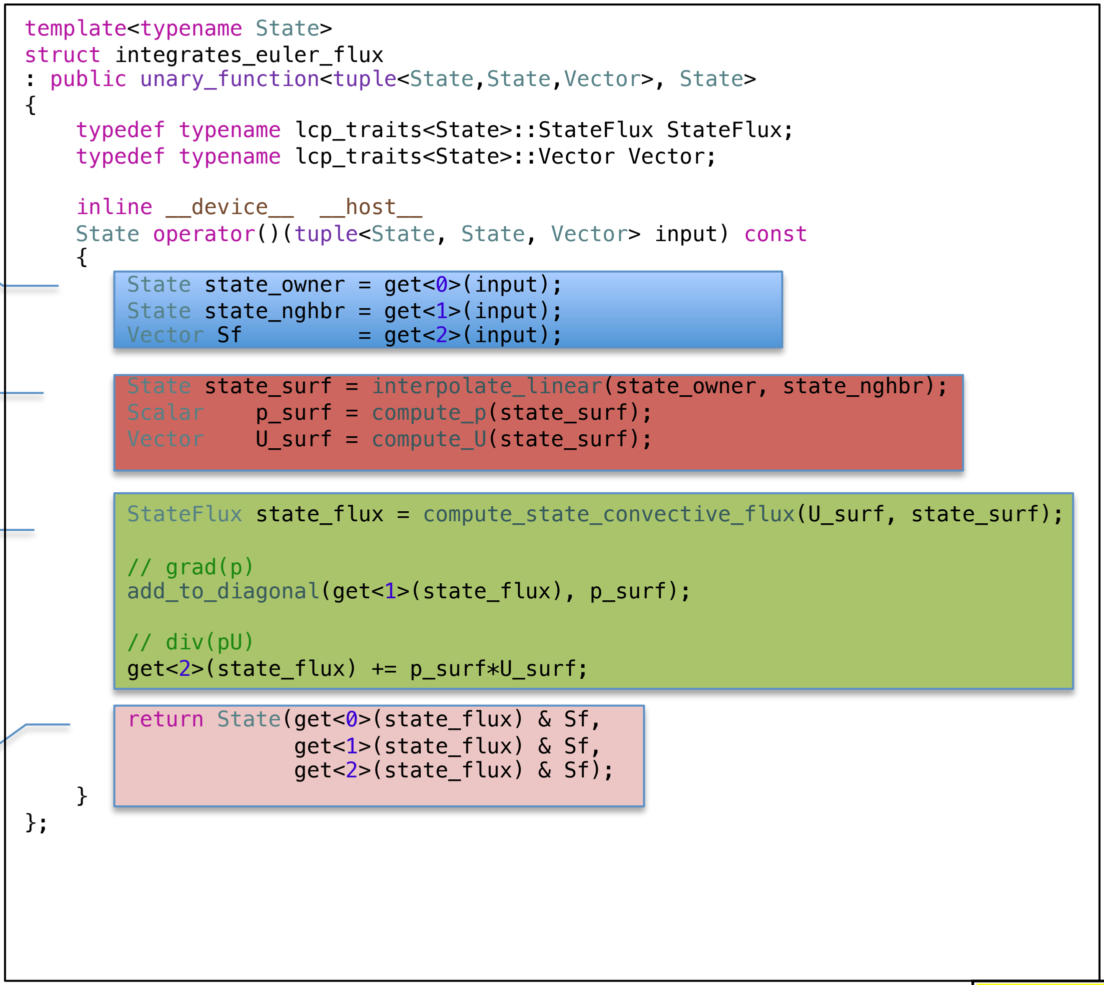
- All computation is performed by either:
  - Looping over cells and updating the cell values
    - Runge-Kutta Time integration
  - Looping over faces, and gathering and scattering cell values between adjacent cells
    - Flux calculation



Implementation

- All algorithms are expressed in terms of:
  1. *Function objects* which represent operations on a per-cell or per-face basis
  2. *Iterators* which provide input and output to these operations.
  3. *Data parallel primitives* (copy, transform, for_each) perform these operations over a range of values specified by iterators, and are used to implement loops over the cells and faces.

Euler flux surface integration



```
template<typename State>
struct integrates_euler_flux
: public unary_function<tuple<State,State,Vector>, State>
{
    typedef typename lcp_traits<State>::StateFlux StateFlux;
    typedef typename lcp_traits<State>::Vector Vector;

    inline __device__  __host__
    State operator()(tuple<State, State, Vector> input) const
    {
        State state_owner = get<0>(input);
        State state_nghbr = get<1>(input);
        Vector Sf         = get<2>(input);

        State state_surf = interpolate_linear(state_owner, state_nghbr);
        Scalar    p_surf = compute_p(state_surf);
        Vector    U_surf = compute_U(state_surf);

        StateFlux state_flux = compute_state_convective_flux(U_surf, state_surf);

        // grad(p)
        add_to_diagonal(get<1>(state_flux), p_surf);

        // div(pU)
        get<2>(state_flux) += p_surf*U_surf;

        return State(get<0>(state_flux) & Sf,
                     get<1>(state_flux) & Sf,
                     get<2>(state_flux) & Sf);
    }
};
```

Input: Euler state at each adjacent cell, the face normal.

Interpolates to the face, derives pressure and velocity.

Computes the convective flux and pressure terms

Returns the flux integrated over the surface

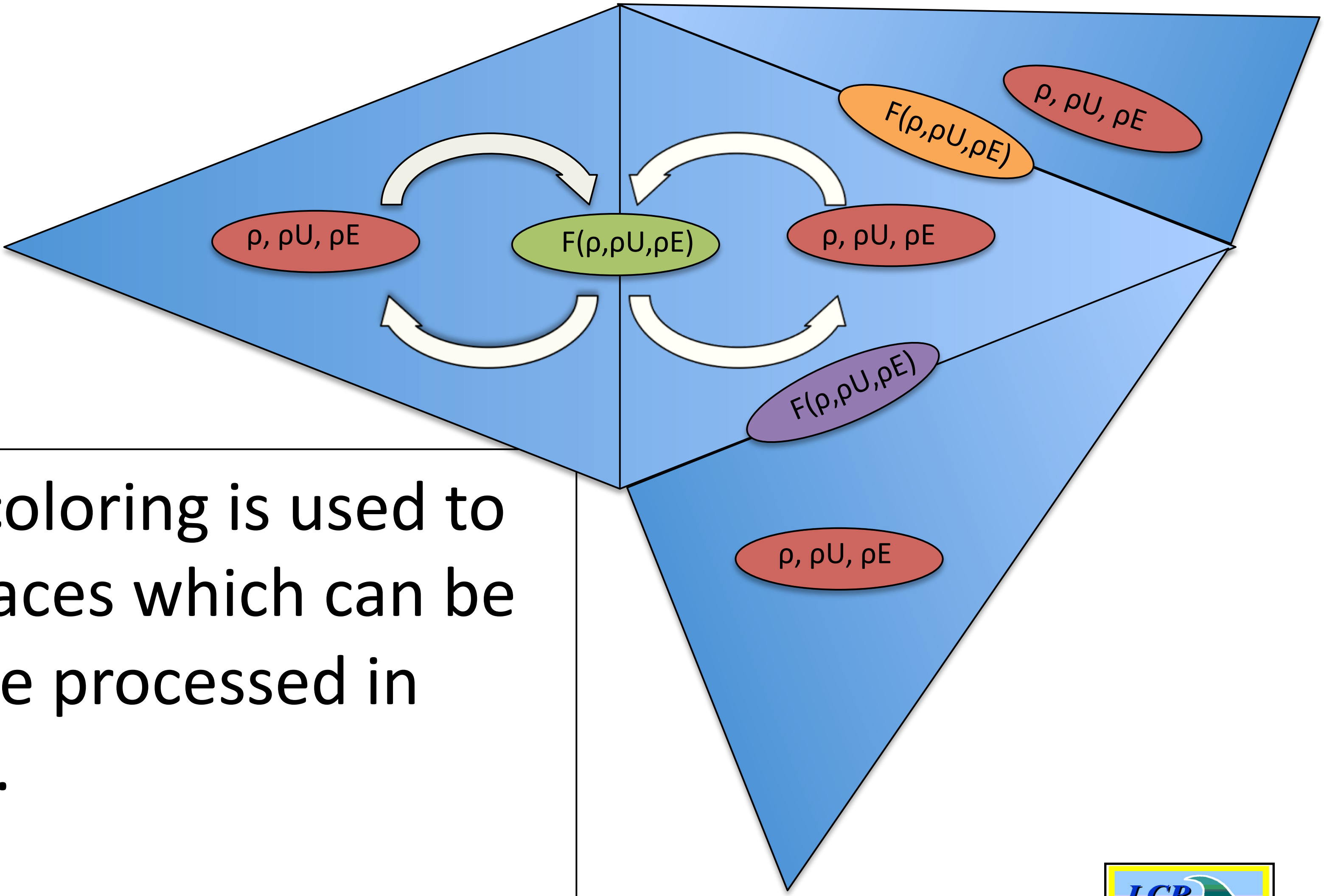
Shared Memory Parallelism:

GPUs and CPUs

- Portability is achieved using the Standard C++ Library and Boost: any platform with a C++ compiler can compile and run the code.
- Analogous implementations of these algorithms are provided by the Thrust library to run on multi-core CPUs and many-core GPUs.

Shared Memory Parallelism:

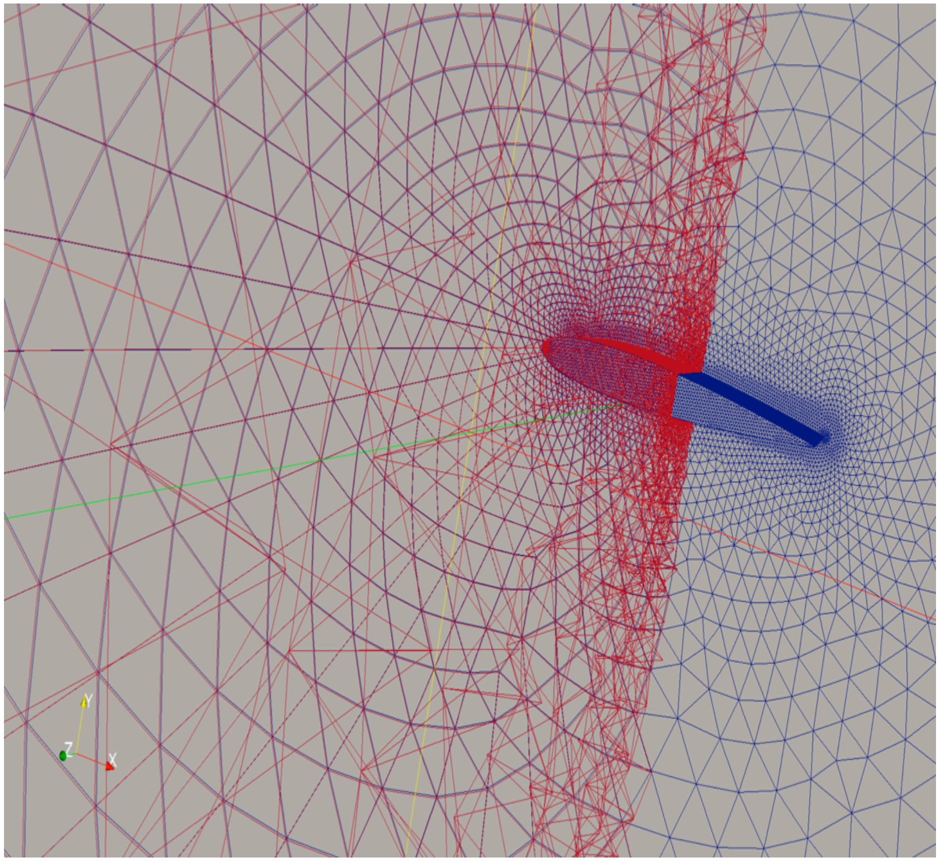
Coloring



- A face coloring is used to group faces which can be safely be processed in parallel.

Distributed Memory Parallelism:

Multi GPUs and CPUs

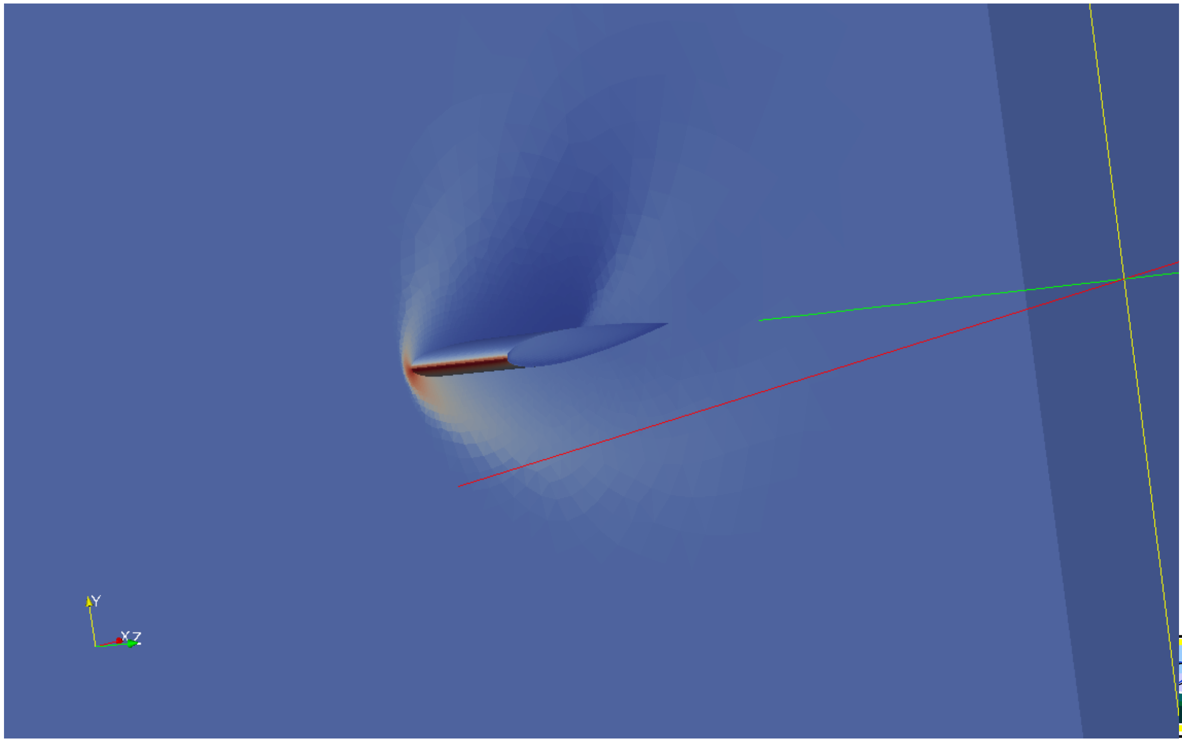


A partitioned NACA 0012 mesh

- Jet noise problems require enormous meshes.
- Requires multi-CPU/GPU parallelization.
- Special boundary conditions are implemented which make MPI calls to exchange data across processor boundaries
- MPI support is entirely orthogonal to shared memory multicore CPU/GPU parallelization.
- The code can run either in a number of configurations
  - MPI
  - MPI + CUDA
  - MPI + OpenMP

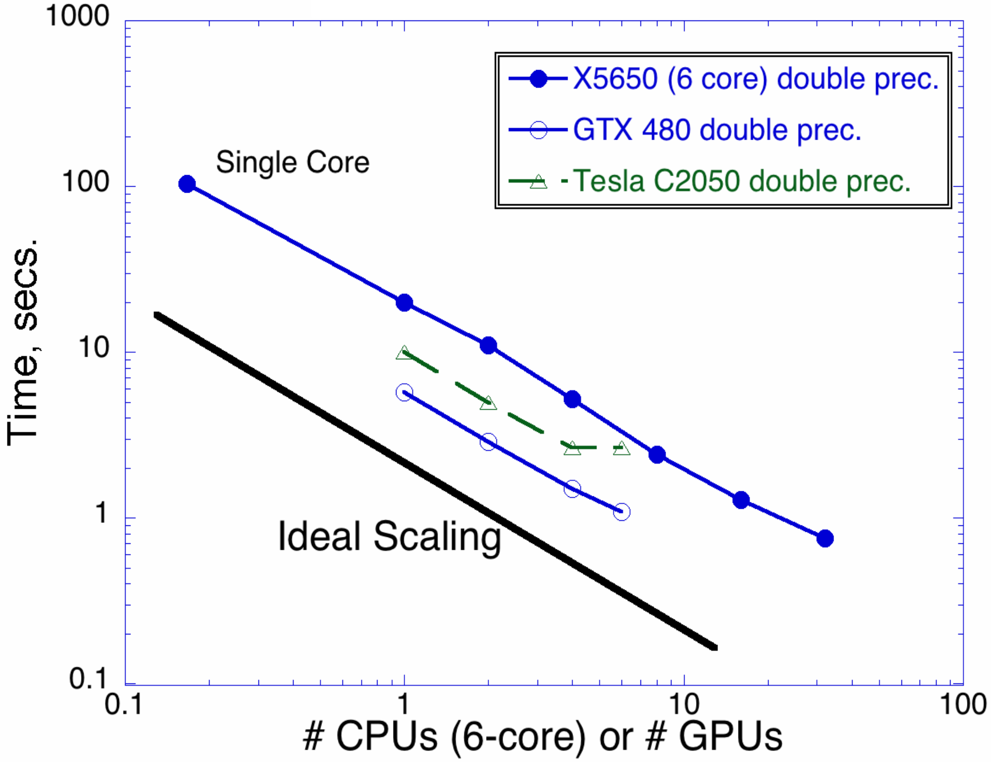
Performance Results

- Performance was measured on a benchmark NACA 0012 air foil case.
- Two mesh sizes, 1 million and 11 million elements.



Performance Results

Double precision, 1 million cells

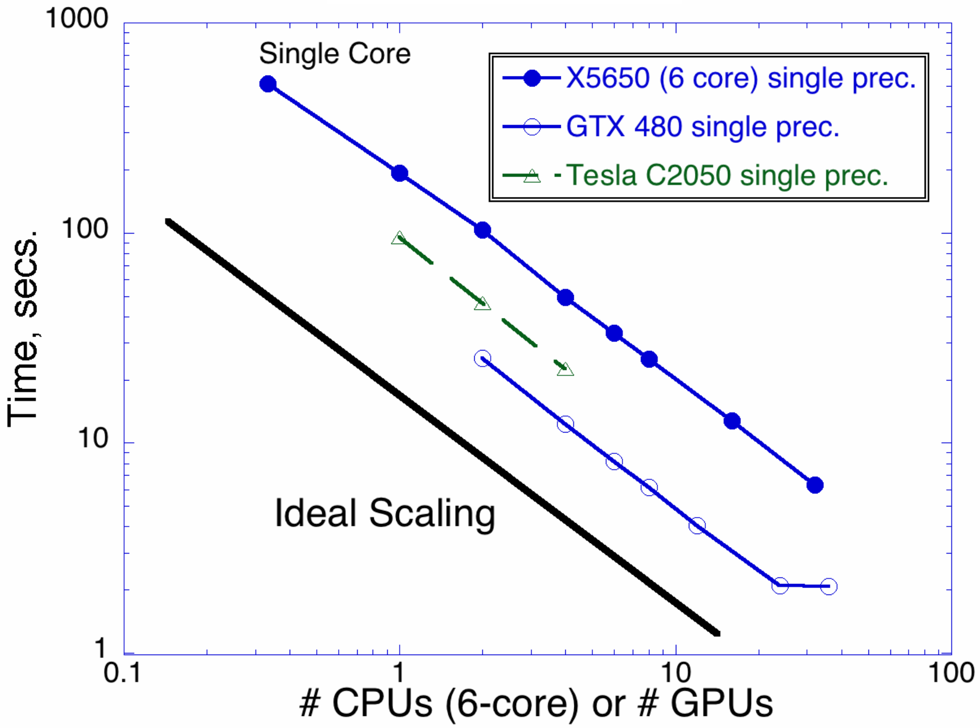


GTX 480 achieves roughly a factor of 4 over the X5650 6-core processor

The Fermi C2050 achieves roughly a factor of 2 performance over the X5650

Performance Results

Single precision, 11 million cells



GTX 480 achieves roughly a factor of 4 over the X5650 6-core processor

The Fermi C2050 achieves roughly a factor of 2 performance over the X5650

Conclusions/Future Work

- Good scaling achieved with a 3D unstructured grid code across many nodes on a GPU cluster.
- The code appears to be memory bandwidth bound.
  - Numbering schemes for meeting coalescing requirements on unstructured grids are needed.
- Continued development of the physics capabilities

Conclusions/Future Work

- CPU affinity must be set for optimal transfer performance
- One GPU per CPU scales linearly
- Open Issue: Performance with multiple GPUs per chipset
- Test GPU Direct/Node to Node Bandwidth
  - Allows for Pinned memory enabled for both GPU and Infiniband Buffers

Acknowledgements

- The code was developed in collaboration with Johann Dahm (University of Michigan and NRL)
- Rainald Löhner (George Mason University) for the NACA 0012 mesh.
- Work was funded by Office of Naval Research and Naval Research Laboratory