Accelerating the BLAST code with hybrid MPI + OpenMP + CUDA programming

Tingxing(Tim) Dong

Lawrence Livermore National Laboratory

September 7th, 2012

Mentors: Tzanio Kolev, Robert Rieben

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344

Outline

- Introduction of BLAST
- Motivation
- Implementation
- Results
- Conclusion

Outline

- Introduction of BLAST
- Motivation
- Implementation
- Results
- Conclusion

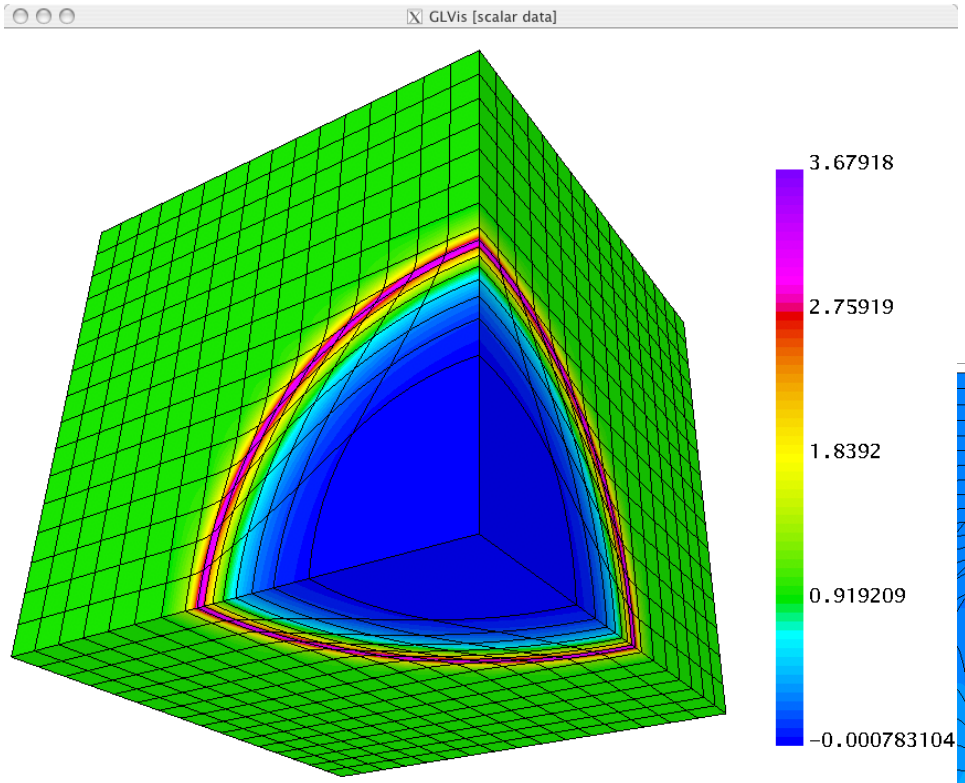
Introduction

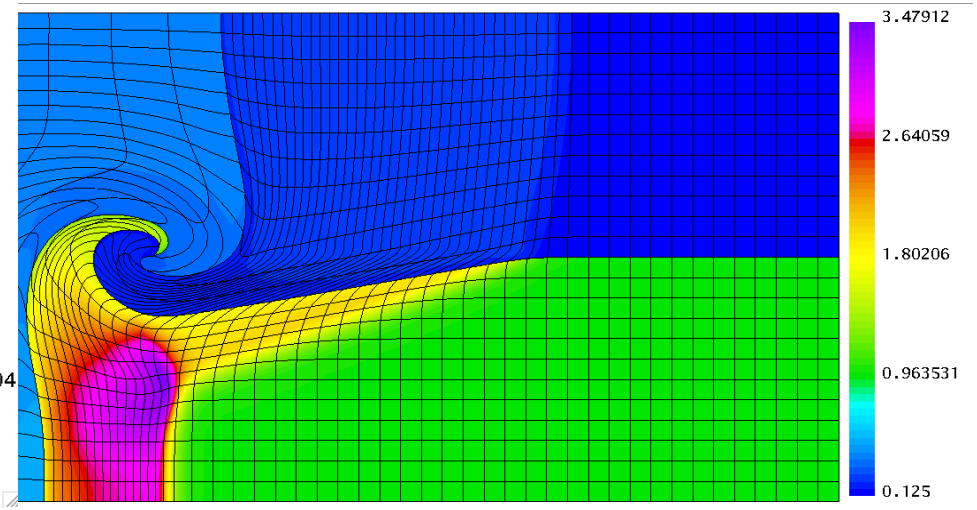
BLAST

- Solve equations of compressible hydrodynamics with Finite Element Method(FEM)
- Based on Lagrangian frame (moving mesh)
- C++ code, parallelized by MPI

BLAST's features

- Curvilinear zone geometries
- Higher order field representations
- Exact discrete energy conservation by construction
- Reduces to classical SGH under simplifying assumptions
- Support for 2D/3D meshes
- Multiple options for basis functions / quadrature order


GLVis [scalar data]
3.67918
2.75919
1.8392
0.919209
-0.000783104


3.47912
2.64059
1.80206
0.963531
0.125

Outline

- Introduction of BLAST
- Motivation
- Implementation
- Results
- Conclusion

Target Cluster: Edge

CPU	GPU	Memory/Node	Switch	Nodes
2x 6Core X5660	2 M2050	96GB	IB QDR	216

Table: Overview of Edge Cluster in LLNL

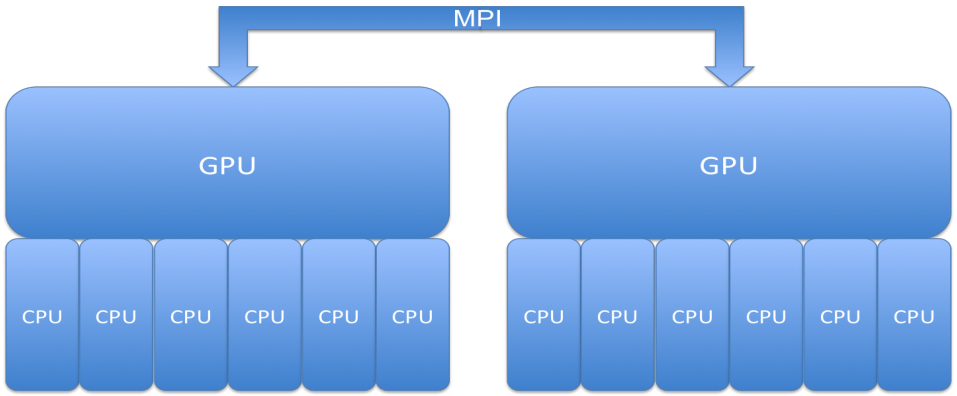


Figure: Architecture hierarchy

Two Strategies

MPI + CUDA

- 1 GPU is dedicated to 1 MPI task(before Kepler)
- 6 MPI task run on 6 cores
- 6 cores + 1 GPU : 1 MPI runs with GPU, 5 do not
- Load balance problem between MPIs
- We have to modify METIS, which takes care of load balance of BLAST

MPI + OpenMP + CUDA

- 6 OpenMP threads run on 6 cores
- 1 MPI task calls 6 OpenMP threads and 1 GPU

Outline

- Introduction of BLAST
- Motivation
- Implementation
- Results
- Conclusion

Euler's Equations in a Lagrangian Frame

Euler's Equations

Momentum Conservation: $$\rho \frac{\mathrm{d}\vec{v}}{\mathrm{d}t} = \nabla \cdot \sigma$$

Mass Conservation: $$\frac{1}{\rho}\frac{\mathrm{d}\rho}{\mathrm{d}t} = -\nabla \cdot \vec{v}$$

Energy Conservation: $$\rho \frac{\mathrm{d}e}{\mathrm{d}t} = \sigma : \nabla \vec{v}$$

Equation of State: $$p = EOS(e, \rho)$$

Equation of Motion: $$\frac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \vec{v}$$

Semi-discrete finite element method in BLAST

Momentum Conservation: $$\mathbf{M_v} \cdot \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\mathbf{F} \cdot \mathbf{1}$$

Energy Conservation: $$\frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t} = \mathbf{M_e}^{-1}\mathbf{F}^{\mathbf{T}} \cdot \mathbf{v}$$

Equation of Motion: $$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{v}$$

Generalized corner forces on the GPU

Semi-discrete finite element method in BLAST

Momentum Conservation: $\mathbf{M_v} \cdot \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\mathbf{F} \cdot \mathbf{1}$

Energy Conservation: $\frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t} = \mathbf{M_e}^{-1}\mathbf{F}^{\mathsf{T}} \cdot \mathbf{v}$

Equation of Motion: $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{v}$

Matrix F is highly floating point operation intensive and thread independent

F is constructed by two loops:
- Loop over zones in the domain(in each processor)
  -Loop over quadrature points in this zone
    Compute hydro forces associated with this quadrature point

on each point we compute this value absoutely independently

$$(\mathbf{F}_z)_{ij} = \int_{\Omega_z(t)} (\sigma : \nabla \vec{w}_i)\, \phi_j = \sum_k \alpha_k \hat{\sigma}(\hat{\vec{q}}_k) : \mathbf{J}_z^{-1}(\hat{\vec{q}}_k) \hat{\nabla} \hat{\vec{w}}_i(\hat{\vec{q}}_k)\, \hat{\phi}_j(\hat{\vec{q}}_k) |\mathbf{J}_z(\hat{\vec{q}}_k)|$$

Pre-computed constant values that can go in constant memory

have to solve eigen vectors/values, SVD.

Profile of BLAST

Method	Corner Force Matrix	CG Solver	Total time
Q4Q3	198.6	53.6	262.7
Q3Q2	72.6	26.2	103.7
Q2Q1 3D	90	56.7	164

Table: Corner force tasks 55%-75% of total time. CG solver takes 20%-34%. Measurment is based on three hundred iterations. Time is in seconds.

Hybrid programming model

two layers of parallelism

- MPI-based parallel domain-partition and communica- tion between CPUs
- CUDA and OpenMP based parallel corner force inside each MPI task
- CUDA OpenMP can be turned on off

OpenMP & CUDA to accelerate Corner Force

- Host thread distributes work on GPU and return immediately
- Spawn 6 threads running on 6 cores
- No communication during compuatation of corner force
- OpemMP and GPU are done; Sychronization of CPU and GPU.
- host thread resumes
- Auto balance to maintain load balance between OpenMP and GPU

GPU implementation

Semi-discrete finite element method in BLAST

Momentum Conservation: $$\mathbf{M_v} \cdot \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\mathbf{F} \cdot \mathbf{1}$$

Energy Conservation: $$\frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t} = \mathbf{M_e}^{-1}\mathbf{F}^{\mathsf{T}} \cdot \mathbf{v}$$

Equation of Motion: $$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{v}$$

CUDA kernel 1: Loop over qudrature points. Compute part of **F** based on **v**, **e**, **x** (transferred from CPU), and need to solve eigen vec/values and SVD.

CUDA kernel 2: Loop over zones. each zone does a DGEMM and assemble the **F**

CUDA kernel 3 : Compute $\mathbf{F} \cdot \mathbf{1}$ and either return result to the CPU or keep on the GPU depending on the CG solver settings.

CUDA kernel 4 : Compute $\mathbf{F}^{\mathsf{T}} \cdot \mathbf{v}$

Mass matrix solve on the GPU

Semi-discrete finite element method in BLAST

Momentum Conservation: $$\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\mathbf{M_v}^{-1}\mathbf{F}\cdot\mathbf{1}$$

Energy Conservation: $$\frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t} = \mathbf{M_e}^{-1}\mathbf{F}^{\mathbf{T}}\cdot\mathbf{v}$$

Equation of Motion: $$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{v}$$

CUDA kernel 5 : CUDA CG solver for $\mathbf{M_v}^{-1}\mathbf{F}\cdot\mathbf{1}$ based on CUBLAS/CUSPARSE/MAGMA, with a diagonal preconditioner.

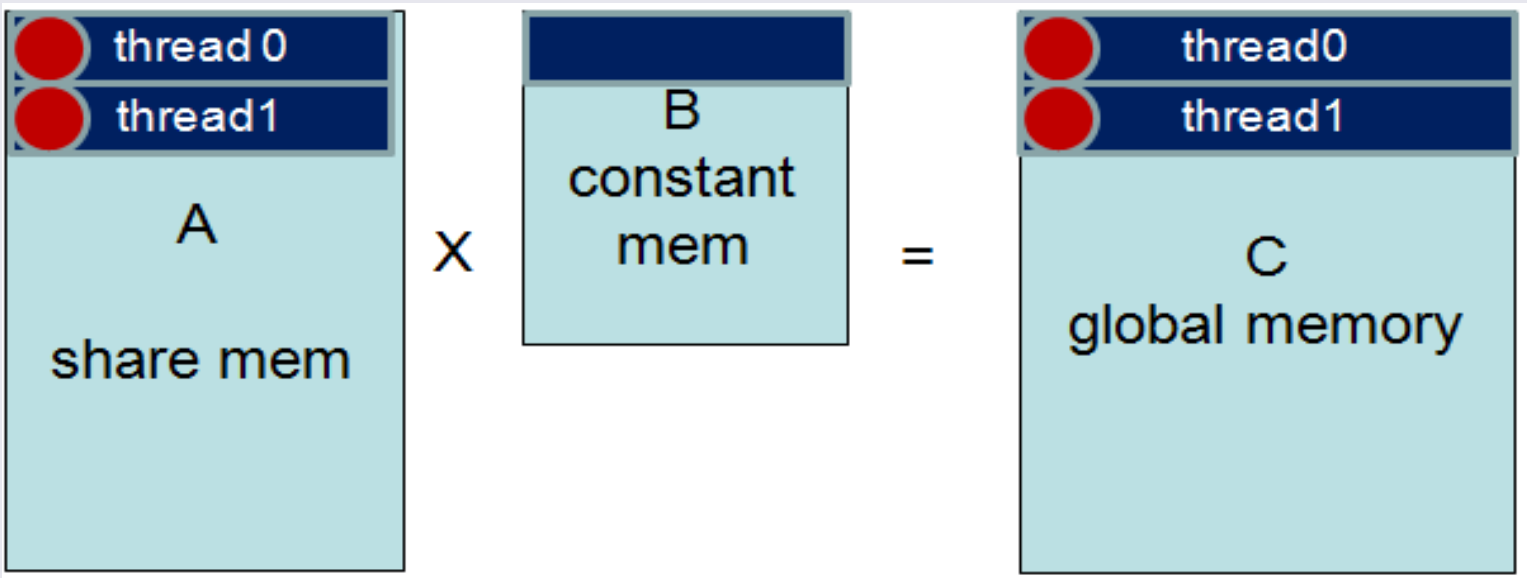
CUDA kernel 6 : SpMV to solve $\mathbf{M_e}^{-1}\mathbf{F}^{\mathbf{T}}\cdot\mathbf{v}$ by calling CUSPARSE

Notice:

$\mathbf{M_v}$ and $\mathbf{M_e}^{-1}$ are computed once and read only thereafter (stay on GPU)
$\mathbf{M_v}^{-1}$ is dense, so we did not use it directly
$\mathbf{M_e}$ is a diagonal local dense matrix, so $\mathbf{M_e}^{-1}$ is a sparse one, can be used directly

Kernel2: ABt = C

- Each thread block(zone) does a ABt=C
- A is updated in each time step; B is read only and common to every blocks
- Accessing global memory is coalesced



Optimization of ABt = C

Problem	Kernel 2.1	Kernel 2.2	Kernel 2.3	CUBLAS	Kernel 1
Sedov 2D	0.63	0.42	0.34	27.2	2.06
Triple-pt	0.14	0.095	0.08	5.7	0.48

Table: Running time of different versions of kernel 2. In kernel 2.3 the bandwidth is highly improved by taking advantage of the memory hierachy. For comparison, time of Kernel 1 is also presented. Kernel time is measured with CUDA event in milliseconds.

Kernel 2.1 only use global memory to read A and B. Kernel 2.2 use shared memory to read A. Kernel 2.3 used shared memory to read A and constant memory to read B.

CUDA + OpenMP implementation

- In corner force, each zone computes independently;
- Some zones go to GPU; The others go to CPU (how to find the optimal ratio is the key) .
- Because GPU runs asynchronously with CPU, control will return to host thread prior to GPU completing work.
- After launch CUDA kernels, host thread will spawns OpenMP threads and distributes the zones (loop) among threads.
- Each OpenMP thread executes like normal serial code.
- Synchronization between CPU and GPU.

Auto balance

- Repeated iterations of the same compuational component;
- First, work load(zones) is distributed among CPU and GPU in an arbitrary ratio, say half to half.
- In each iteration, GPU and CPU are timed seperately.
- If the ratio of their running time exceed the interval say $[0.9 - 1.1]$, move zones will go to the one who finished earlier.
- Just a few sampling periods.
- This idea can be extend to other tuning parameters.

Problem	Optimal ratio	Convergence period
Sedov 3D	0.45	3
Sedov 2D	0.75	14
Triple-pt	0.77	12

Table: The optimal ratio refers to the percentage of zones ditributed on GPU out of total zones. The starting tentative ratio is 0.5, and target interval is $[0.9 - 1.1]$. One sampling period consists of forty interations.

Outline

- Introduction of BLAST
- Motivation
- Implementation
- Results
- Conclusion

Validation: Engery preservation

Platform	Time out	Kinetic	Internal	Total	Total Change
CPU/GPU	Initial	0.0000000e+00	1.0050000e+01	1.005e+01	N/A
CPU	0.6	5.0423596e-01	9.5457640e+00	1.005e+01	-9.21929199648e-13
GPU	0.6	5.0418618e-01	9.5458131e+00	1.005e+01	-4.93827201353e-13

Table: Results of CPU and GPU for triple-pt problem with Q3Q2 method; Total Energy includes kinetic energy and internal energy. Both CPU and GPU results preserve energy conservation very well.

Validation

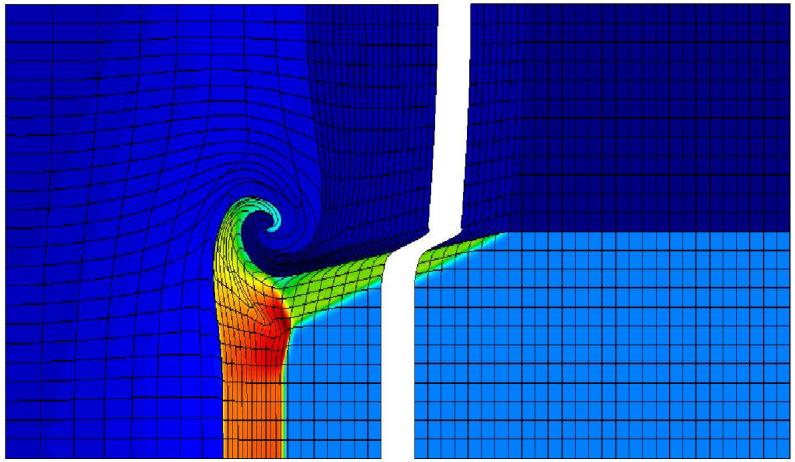


Figure: 2MPI tasks each with 1 M2050 and 6 Xeon Cores

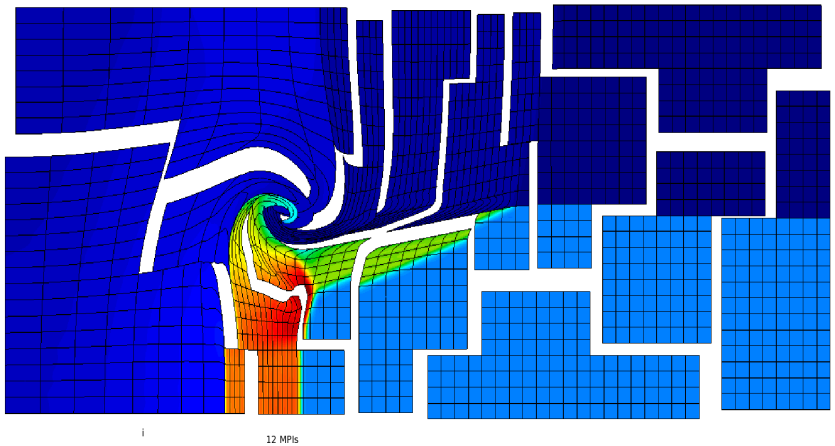


Figure: 12 MPI CPU tasks

Performance of Corner Force

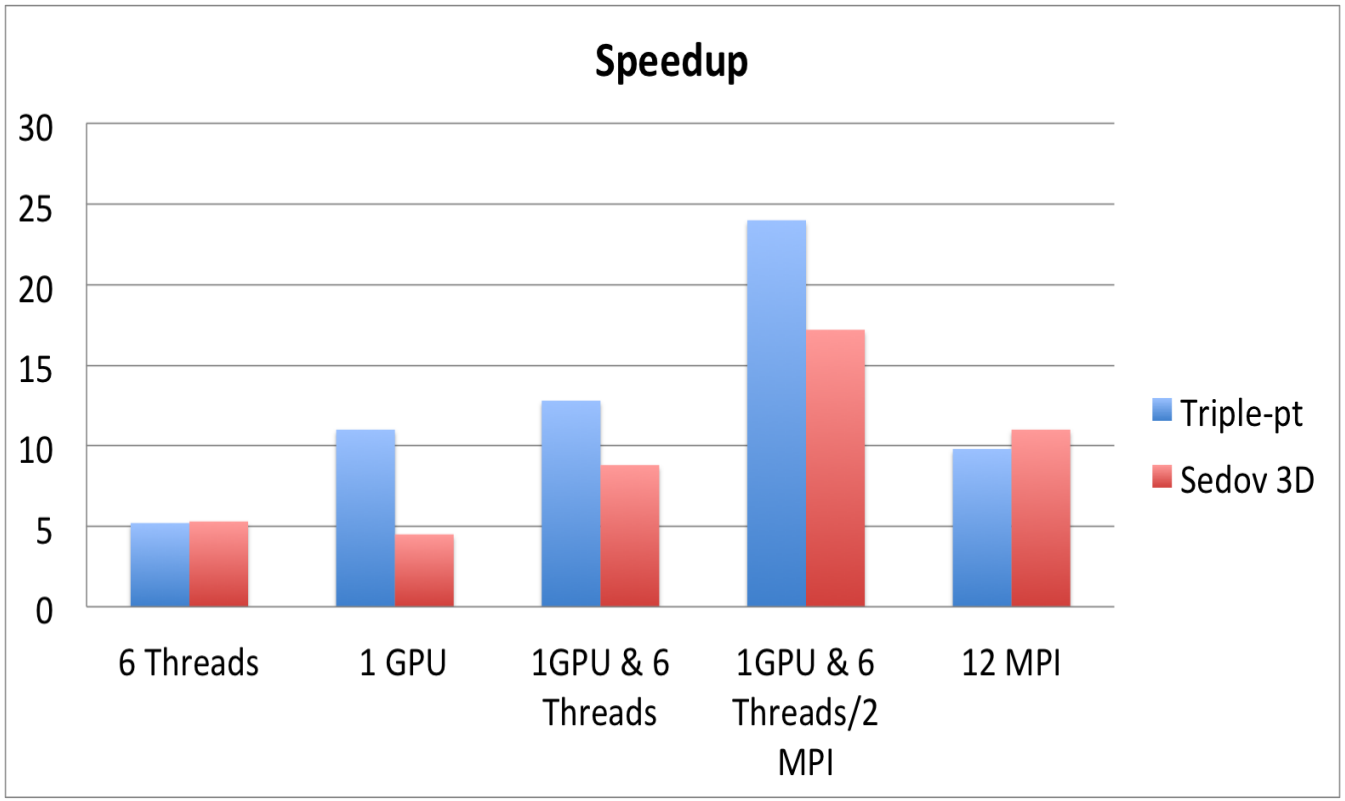


Figure: Speedup of corner force compared to serial code

Performance of CUDA-CG: 1 GPU vs 1 core

Problem	Method	MFEM PCG	CUDA PCG	Speedup
Triple-pt	Q3Q2	90.18	20.8	4.3
Sedov 3D	Q2Q1	27.81	10.55	2.6

Table: Speedup of CUDA-PCG compared to PCG in BLAST. Memory transfer overhead is counted in CUDA-PCG. Measurment is based on 1000 iterations. Time is in seconds.

Limitation: CUDA-CG only runs on 1 GPU until now.

Overall speedup

Overall testing consider every component, including CUDA corner force and CUDA-PCG and the unparallelized part. Tests are performed on a single GPU compared to a single CPU core.

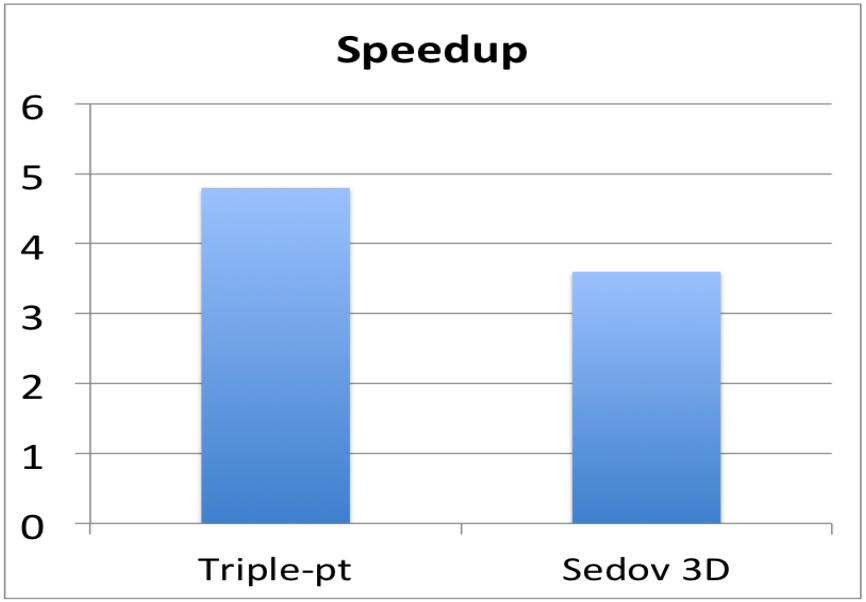


Figure: Overall speedup: including CUDA corner force, CUDA-PCG and unparallelized part

Method	Corner Force Matrix	CG Solver	Maximum Speedup
Triple-pt Q3Q2	70%	20%	10
Sedov Q2Q1 3D	54%	34%	8

Outline

- Introduction of BLAST
- Motivation
- Implementation
- Results
- Conclusion

Conclusion

- GPU computing is a good choice for CFD problems.
- We only parallelize part of code on GPU/OpenMP, the overall speedup is limited by Amdahl's law.
- On Kepler, MPI tasks "truely" share one GPU by Hyper-Q.

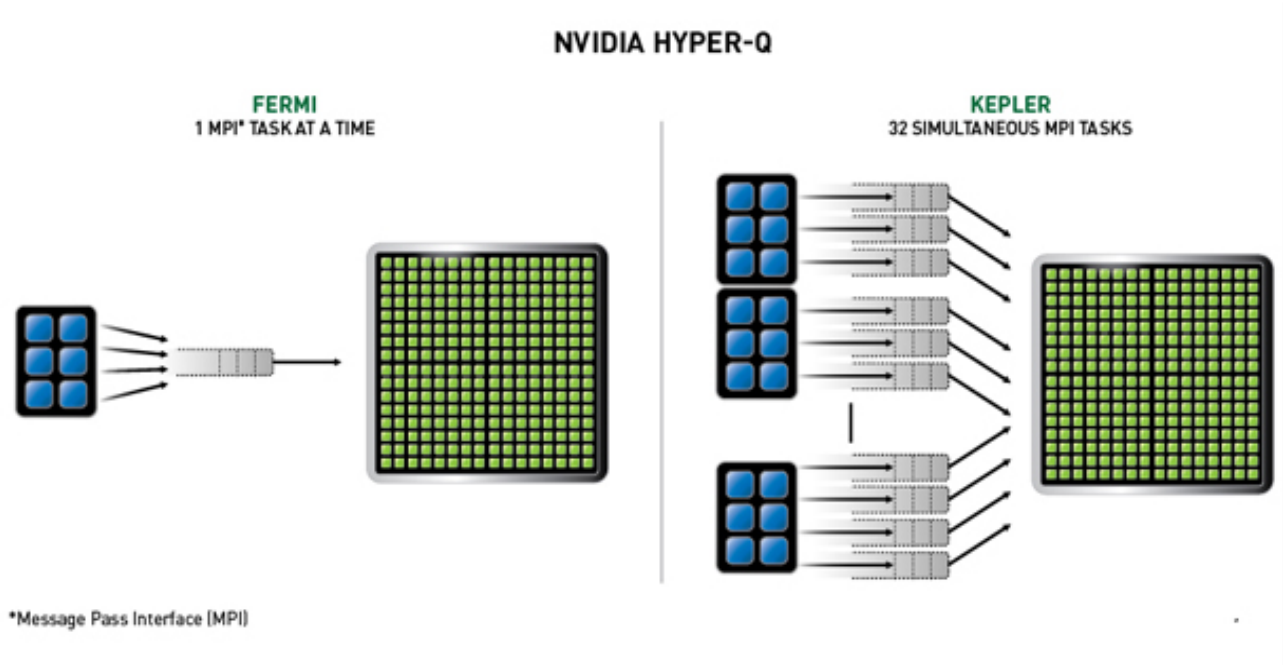


Figure: Kepler: Hyper-Q