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
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
Outline

- Introduction of BLAST
- Motivation
- Implementation
- Results
- Conclusion
Target Cluster: Edge

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
<th>Memory/Node</th>
<th>Switch</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x 6Core X5660</td>
<td>2 M2050</td>
<td>96GB</td>
<td>IB QDR</td>
<td>216</td>
</tr>
</tbody>
</table>

**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:** \( \rho \frac{d\vec{v}}{dt} = \nabla \cdot \sigma \)
- **Mass Conservation:** \( \frac{1}{\rho} \frac{d\rho}{dt} = -\nabla \cdot \vec{\rho} \)
- **Energy Conservation:** \( \rho \frac{de}{dt} = \sigma : \nabla \vec{v} \)
- **Equation of State:** \( p = EOS(e, \rho) \)
- **Equation of Motion:** \( \frac{d\vec{x}}{dt} = \vec{v} \)

**Semi-discrete finite element method in BLAST**

- **Momentum Conservation:** \( M_v \cdot \frac{dv}{dt} = -F \cdot 1 \)
- **Energy Conservation:** \( \frac{de}{dt} = M_e^{-1}F^T \cdot v \)
- **Equation of Motion:** \( \frac{dx}{dt} = v \)
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 absolutely independently

$$
(F_z)_{ij} = \int_{\Omega_z(t)} (\sigma : \nabla \hat{w}_i) \phi_j = \sum_k \alpha_k \hat{\sigma}(\hat{q}_k) : J_z^{-1}(\hat{q}_k) \nabla \hat{w}_i(\hat{q}_k) \hat{\phi}_j(\hat{q}_k) |J_z(\hat{q}_k)|
$$

have to solve eigen vectors/values, SVD.
Profile of BLAST

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Corner Force Matrix</th>
<th>CG Solver</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q4Q3</td>
<td>198.6</td>
<td>53.6</td>
<td>262.7</td>
</tr>
<tr>
<td>Q3Q2</td>
<td>72.6</td>
<td>26.2</td>
<td>103.7</td>
</tr>
<tr>
<td>Q2Q1 3D</td>
<td>90</td>
<td>56.7</td>
<td>164</td>
</tr>
</tbody>
</table>
Hybrid programming model

two layers of parallelism

- MPI-based parallel domain-partition and communication 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 computation of corner force
- OpenMP and GPU are done; Synchronization of CPU and GPU.
- Host thread resumes
- Auto balance to maintain load balance between OpenMP and GPU
### Semi-discrete finite element method in BLAST

<table>
<thead>
<tr>
<th>Equation</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Momentum Conservation:</td>
<td>$M_v \cdot \frac{dv}{dt} = -F \cdot 1$</td>
</tr>
<tr>
<td>Energy Conservation:</td>
<td>$\frac{de}{dt} = M_e^{-1} F^T \cdot v$</td>
</tr>
<tr>
<td>Equation of Motion:</td>
<td>$\frac{dx}{dt} = v$</td>
</tr>
</tbody>
</table>

CUDA kernel 1: Loop over quadrature 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 $F \cdot 1$ and either return result to the CPU or keep on the GPU depending on the CG solver settings.

CUDA kernel 4: Compute $F^T \cdot v$
Mass matrix solve on the GPU

Semi-discrete finite element method in BLAST

Momentum Conservation: \[ \frac{dv}{dt} = -M_v^{-1} F \cdot 1 \]

Energy Conservation: \[ \frac{de}{dt} = M_e^{-1} F^T \cdot v \]

Equation of Motion: \[ \frac{dx}{dt} = v \]

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

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

Notice:

\( M_v \) and \( M_e^{-1} \) are computed once and read only thereafter (stay on GPU)

\( M_v^{-1} \) is dense, so we did not use it directly

\( M_e \) is a diagonal local dense matrix, so \( M_e^{-1} \) 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
Table: Running time of different versions of kernel 2. In kernel 2.3 the bandwidth is highly improved by taking advantage of the memory hierarchy. For comparison, time of Kernel 1 is also presented. Kernel time is measured with CUDA event in milliseconds.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Kernel 2.1</th>
<th>Kernel 2.2</th>
<th>Kernel 2.3</th>
<th>CUBLAS</th>
<th>Kernel 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sedov 2D</td>
<td>0.63</td>
<td>0.42</td>
<td>0.34</td>
<td>27.2</td>
<td>2.06</td>
</tr>
<tr>
<td>Triple-pt</td>
<td>0.14</td>
<td>0.095</td>
<td>0.08</td>
<td>5.7</td>
<td>0.48</td>
</tr>
</tbody>
</table>

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.
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 computational component;
- First, workload (zones) is distributed among CPU and GPU in an arbitrary ratio, say half to half.
- In each iteration, GPU and CPU are timed separately.
- If the ratio of their running time exceeds 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 extended to other tuning parameters.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Optimal ratio</th>
<th>Convergence period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sedov 3D</td>
<td>0.45</td>
<td>3</td>
</tr>
<tr>
<td>Sedov 2D</td>
<td>0.75</td>
<td>14</td>
</tr>
<tr>
<td>Triple-pt</td>
<td>0.77</td>
<td>12</td>
</tr>
</tbody>
</table>

**Table:** The optimal ratio refers to the percentage of zones distributed 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 iterations.
Outline

- Introduction of BLAST
- Motivation
- Implementation
- Results
- Conclusion
Validation: Energy preservation

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.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Time out</th>
<th>Kinetic</th>
<th>Internal</th>
<th>Total</th>
<th>Total Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU/GPU</td>
<td>Initial</td>
<td>0.0000000e+00</td>
<td>1.0050000e+01</td>
<td>1.005e+01</td>
<td>N/A</td>
</tr>
<tr>
<td>CPU</td>
<td>0.6</td>
<td>5.0423596e-01</td>
<td>9.5457640e+00</td>
<td>1.005e+01</td>
<td>-9.21929199648e-13</td>
</tr>
<tr>
<td>GPU</td>
<td>0.6</td>
<td>5.0418618e-01</td>
<td>9.5458131e+00</td>
<td>1.005e+01</td>
<td>-4.93827201353e-13</td>
</tr>
</tbody>
</table>
Validation

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

Figure: 12 MPI CPU tasks
Figure: Speedup of corner force compared to serial code
Performance of CUDA-CG: 1 GPU vs 1 core

<table>
<thead>
<tr>
<th>Problem</th>
<th>Method</th>
<th>MFEM PCG</th>
<th>CUDA PCG</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triple-pt</td>
<td>Q3Q2</td>
<td>90.18</td>
<td>20.8</td>
<td>4.3</td>
</tr>
<tr>
<td>Sedov 3D</td>
<td>Q2Q1</td>
<td>27.81</td>
<td>10.55</td>
<td>2.6</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Corner Force Matrix</th>
<th>CG Solver</th>
<th>Maximum Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triple-pt Q3Q2</td>
<td>70%</td>
<td>20%</td>
<td>10</td>
</tr>
<tr>
<td>Sedov Q2Q1 3D</td>
<td>54%</td>
<td>34%</td>
<td>8</td>
</tr>
</tbody>
</table>
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