Acceleration of BLAST Hydra Code on GPU

Tingxing(Tim) Dong

Lawrence Livermore National Laboratory

September 9th, 2011

Mentors: Tzanio Kolev, Robert Rieben
Outline

- Introduction of BLAST
- Motivation
- Details
- Optimization and Restriction
- Examples and Results
- Conclusion
Outline

- Introduction of BLAST
- Motivation
- Details
- Optimization and Restriction
- Examples and 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

Q1Q0, Q2Q1, and Q3Q2 cases

reference

random

velocity, position
density, energy, pressure
Euler’s Equations in a Lagrangian Frame

**Euler’s Equations**

**Momentum Conservation:**
\[ \rho \frac{d\vec{v}}{dt} = \nabla \cdot \sigma \]

**Mass Conservation:**
\[ \frac{1}{\rho} \frac{d\rho}{dt} = -\nabla \cdot \vec{v} \]

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

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

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

- Introduction of BLAST
- Motivation
- Details
- Optimization and Restriction
- Examples and Results
- Conclusion
Consider three cases of Lagrange hydro problems with different computational workloads:

- 2D q2q1: least expensive, most common
- 2D q3q2: more expensive, greater robustness and accuracy
- 3D q2q1: most expensive

---

**Google Profiler of BLAST**

### 2D q2q1

![Diagram of 2D q2q1 performance profile]

**blast**
- Total samples: 29855
- Focusing on: 29855
- Dropped nodes with <= 149 abs(samples)
- Dropped edges with <= 29 samples

**HydroStatePU**
- **ComputeCornerForces**
  - 2918 (9.8%) of 16502 (55.3%)

**MultiAbt**
- 4548 (15.2%)

**DenseMatrix**
- **CalcEigenvalues**
  - 1589 (5.3%)
- **CalcSingularvalue**
  - 815 (2.7%)
- **FNorm**
  - 1128 (3.8%)

**Vector**
- **GetSubVector**
  - 187 (0.6%)

**Hypre_CSRMatrixMatvec**
- 7515 (25.2%)

---

T. Dong (LLNL)

Acceleration of BLAST Hydra Code on GPU

Seminar 7 / 34
Google Profiler of BLAST

2D q3q2

blast
Total samples: 255527
Focusing on: 255527
Dropped nodes with <= 1277 abs(samples)
Dropped edges with <= 255 samples
blast
Total samples: 195012
Focusing on: 195012
Dropped nodes with <= 975 abs(samples)
Dropped edges with <= 195 samples
My background before this work

- Fairly familiar with Finite Difference Method (FDM)
- Limited understanding about Finite Element Method (FEM)
- Starts from June 1st ends at August 13th  2.5months
Outline

- Introduction of BLAST
- Motivation
- Details
- Optimization and Restriction
- Examples and Results
- Conclusion
Generalized corner forces 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: \[ \frac{dx}{dt} = v \]

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

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

Pre-computed constant values that can go in constant memory

varies with basis functions, dimension, etc. F can be arbitrarily expensive.
Generalized corner forces 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 1: Loop over quadrature points. Compute part of \( F \) based on \( v, e, x \) (transferred from CPU) and allocated work space (on GPU)

CUDA kernel 2: Loop over zones. each zone does a Matrix Matrix Transpose Multiplication and assemble the \( F \) (stay on GPU)

CUDA kernel 3 (in Momentum Equation): 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 (in Energy Equation): Compute \( F^T \cdot v \) based on \( v \) (results stay on GPU)
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 (in Momentum Equation): Custom CG solver (Provided by Stan) for $M_v^{-1}F \cdot 1$ based on CUBLAS/CUSPARSE, with a diagonal preconditioner (We add later).

CUDA kernel 6 (in Energy Equation): Sparse Matrix(CSR) Multiplication 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
Map to CUDA Thread Hierachy

CUDA kernel 1: loop points
Each thread < -- > one quadrature point
Each thread block < -- > one or more zones (tunable)
In fact more flexible: one zone can be split into two thread blocks

CUDA kernel 2: loop zone
Each thread block < -- > one zone
Each block(zone) do MMtMult(ABt=C)
Each thread < -- > one row of Matrix C

CUDA kernel 3, 4
Each thread < -- > one zone
Each thread block is composed of 32 or 64 threads (tunable)

CUDA kernel 5, 6
Call CUBLAS/CUSPARSE/MAGMA library routines
Kernel2: $ABt = C$

- Each thread block (zone) does a $ABt = C$
- $A, B$ are not big and generally can be fitted in shared and constant memory on Fermi
- $A$ is varying to each thread block and updated in each iteration; $B$ is read only and the same to every block
- Matrix are stored in column major
- Accessing global memory is coalesced
Technical details: Memory management

- CUDA code can be integrated into the previous C++ code very well
- Malloc GPU memory in C++ Constructor
- Free GPU memory in C++ Destructor
- Add a new method CUDA ComputerCornerForce

### Constructor and Destructor

```
// in .hpp files
// declare the variables
  double *d_vec;

// in .cu files
HydroState::CUDA_Constructor//called by HydroState()
{
  // malloc variables on GPU and copy initialized
  // and read only data from CPU to GPU
  cudaMalloc(&d_vec);
  cudaMemcpy(ToDevice);
}

HydroState::CUDA_Destructor//called by ~HydroState()
{
  cudaMemcpy(ToDevice);
  cudaFree(d_vec);
}
```

### Corner Force Method

```
// still in .cu files,
HydroState::CUDA_CornerForce
{
  // copy updated hydro states(v,e,x) to GPU
  cudaMemcpy(ToDevice);
  // compute on GPU
  kernel<<< , >>>(d_vec, ...);
  // copy outputs to CPU
  cudaMemcpy(ToHost);
}
```
Technical details: GPU Class

- Porting code, not developing algorithms
- Maximize use of previous C++ code to avoid developing new code
- In BLAST, almost everything is class
- CUDA4.0 support C++ class in device code (although not fully)

**Class on CPU**

```cpp
class Vector
{
private:
    int size;
    double *data;
public:
    Vector(int a)
    double *GetData()
    {return data;}
    void Operation()
    Vector()
}
```

**Class on GPU**

```cpp
class Vector_GPU
{
private:
    int size;
    double *data;
public:
    __device__ Vector_GPU(int a)
    __device__ double *GetData()
    {return data;}
    __device__ void Operation()
    __device__ Vector_GPU()
}
```

T. Dong (LLNL)
An example of using class in GPU kernel

```c
#define num_threads 32
#define vector_length 4

__global__ void kernel(double* d_data)
{
    Vector_GPU vec(vector_length);
    // threadIdx.x is the thread's id, from 0-31, each thread grab its own data via pointer cast
    vec.GetData() = d_data + threadIdx.x * vector_length;
    vec.Operation();
}

int main()
{
    double *d_data;
    // malloc a space on device memory
    cudaMalloc(&d_data, sizeof(double) * num_threads * vector_length);
    // in the kernel, each thread grab its own portion of data and execute parallelly
    kernel<<<1, num_threads,>>>(d_data);
    cudaFree(d_data);
    return 1;
}
```
Technical details: Transfer of arguments

- hydro state $e$, $v$, $x$ (in the form of class objects) needs to be transferred into CUDA kernel
- cannot transfer C++ class objects directly like scalar
- grab data (pointer) from class objects and stored into double array or structs (see programming guide 4.0 page)

**define a struct**

typedef struct
{
    int height;
    int width;
    int chunk;
    int size;
    double *data;
} d_Matrix;

**transfer struct as argument**

void configureMatrix(d_Matrix &dm);
{
    // malloc memory
    // grab data from C++ class objects and copy to dm.data
    // initialize height, width, chunk, size
}

int main()
{
    d_Matrix dm;
    configureMatrix(dm,..);
    kernel<<<, >>> (d_Matirx dm);
}
Outline

- Introduction of BLAST
- Motivation
- Details
- Optimization and Restriction
- Examples and Results
- Conclusion
Optimization

- Use cuda profiler to identify the hot spot (pics of profiler shown later)
- Constant memory store static read only coefficients, like basis function, weight parameters, etc (in kernel 1-4)
- Use shared (store A) and constant (store B) memory to accelerate in cuda kernel 2 $AB^T=C$ (memory refer $O(n^3)$), hinted by profiler
- Implement PCG Solver which uses CUBLAS, CUSPARSE routines instead of coding myself
- Hand code Eigenvalue/vector, SVD (by Veselin) (for very small matrix 2*2(2D) or 3*3(3D), just specific to this application) in kernel 1, as can't call LAPACK as used in C++ BLAST
Restriction

- Code is developed on Telsa C1060 (my local PC), tests run on Fermi.
- Tesla (1.3): not support dynamic malloc and free inside kernel, memory has to be pre-allocated outside kernel, even temporary variables.
- Fermi (2.0): supports dynamic malloc and free inside kernel.
- Tesla: do not support virtual function, so some codes has to be rewritten.
- PCG solver and kernel 6 only work on one processor at present.
- Fairly tuned, but not fully optimized for Fermi.
- Kernel 1 (also the most complicated and expensive one) reading global memory is uncoalesced: each thread (quadrature points) access small matrix (2*2, 3*3) although consecutively.
Outline

- Introduction of BLAST
- Motivation
- Details
- Optimization and Restriction
- Examples and Results
- Conclusion
<table>
<thead>
<tr>
<th>Test Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>q2q1 2D Triple-pt</td>
<td>9 * 2 velocity unknowns dofs (degree of freedoms). 4 energy dofs (zones) 16 points per zones</td>
</tr>
<tr>
<td>q3q2 2D Triple-pt</td>
<td>16 * 2 velocity unknowns dofs. 9 energy dofs (zones) 36 points per zones</td>
</tr>
<tr>
<td>q2q1 3D Sedov wave</td>
<td>27 * 3 velocity unknowns dofs. 8 energy dofs (zones) 64 points per zones</td>
</tr>
</tbody>
</table>
Performance

2550 lines of CUDA code + 2 months

**Tesla C1060**
CPU: Xeon E5520 at 2.27GHz

**Tesla C2050**
CPU: Xeon Westmere-EP X5660 (on Edge)

**Quadro 5000**
CPU: Xeon Westmere-EP X5660
Performance

On Edge
2D q2q1 triple-pt problem

MPI+CUDA

Result of 4 MPI

Result of 4 MPI+CUDA

T. Dong (LLNL)
CPU vs GPU results

2D q3q2 triple-pt on CPU vs GPU

total energy = kinetic energy + internal energy
CPU energy change: -4.04832e-12 with iteration step 38910 at t=2.5
GPU energy change: 2.99486e-09 with iteration step 38748 at t=2.5 (below, 3x on C2050)
CPU vs GPU results

3D q2q1 Sedov on CPU

total energy = kinetic energy + internal energy
CPU: energy change: +1.89570e-13 with iteration step 848 at t = 0.3
GPU: energy change: +1.26013e-11 with iteration step 848 at t = 0.3 (right, 4x on C2050)
CUDA Profiler

Profiler of 2D q3q2 triple-pt

Gpu Time Summary Plot

- kernel_loop_quadrature_point
- kernel_loop_zones
- kernel0_loop_quadrature_point
- kernel_diag_preconditioner_matrix
- csmv_kernel_0
- kernel_loop_zones_dv_dft
- memcpyHtoD
- nnt2_kernel
- dot_kernel
- asy_kernel_val
- kernel_diag_preconditioner_b
- scal_kernel_val
- copy_kernel
- memcpyDtoH
- reduce_1block_kernel
- memcpy32_aligned1D

- Kernel time = 35.08 % of total GPU time
- Memory copy time = 2.4 % of total GPU time
- Kernel taking maximum time = csmv_kernel_0 (15.7 % of total GPU time)
- Memory copy taking maximum time = memcpyHtoD (2.8 % of total GPU time)
- There is no time overlap between memory copies and kernels on GPU

Hint(s)
- Double click on the kernel name in the Summary Table to analyze the kernel
- Consider using page-locked memory to attain higher bandwidth between host and device memory. Overuse of pinned memory should be avoided as it may reduce overall system performance.

Refer to the "Pinned Host Memory" section in the "CUDA C Best Practices" chapter of the CUDA C Programming Guide for more details.
Outline

- Introduction of BLAST
- Motivation
- Details
- Optimization and Restriction
- Examples and Results
- Conclusion
Conclusion

- GPU is well suited for computational heavy kernel
- Floating points operation should accommodate the penalty of transferring data between CPU and GPU
- Optimization is a procedure of discovering
- Profiler does help to identify the bottleneck
- Use existing library instead of coding yourself (not necessarily to be the best but the most stable)
Thanks