PLASMA TURBULENCE SIMULATIONS: PORTING THE GYROKINETIC TOKAMAK SOLVER TO GPU USING CUDA

Praveen Narayanan
NVIDIA
FUSION ENERGY

▪ Fusion
  – Clean: produces no climate changing gases
  – Safe: no danger of runaway reactions or melt down
  – Abundant fuel source: sea water

▪ Need to ‘confine’ material to effect fusion reaction

▪ Two kinds of fusion: inertial and magnetic
PLASMA FUSION IN A TOKAMAK

- Plasmas contained in a toroidal reactor called tokamak
- Strong magnetic fields help contain the particles
NEED FOR NUMERICAL SIMULATIONS

- Experiments need prototyping with numerical simulations
- ITER: World’s largest tokamak in the making
  - A worldwide effort!
  - LOCATION: South of France
- Each ITER ‘shot’ or experiment estimated at $1M
WHAT IS THE GYROKINETIC TOKAMAK SOLVER (GTS)?

- GTS is a solver that simulates plasma magnetic fusion in a tokamak
- One of several solvers used to prototype experiments in plasma fusion
- Use a “Particle in Cell” (PIC) approach to solve equations
GTS SOLVER

- PIC code (due to Weixing Wang and Stephane Ethier from PPPL)
- Solves for particle distribution functions
- Guiding center approximation reduces 6D (3x, 3p) equations to 5D - effects a massive savings in computation time

\[
\frac{\partial f_\alpha}{\partial t} + \bar{u} \cdot \nabla f_\alpha + \frac{q_\alpha}{m_\alpha} \left( \bar{E} + \bar{u} \times \bar{B} \right) \cdot \nabla_{\bar{u}} f_\alpha = \left( \frac{\partial f_\alpha}{\partial t} \right)_c \quad f_\alpha(\bar{r}, \bar{u}, t)
\]

\[
\frac{\partial \bar{B}}{\partial t} = -\nabla \times \bar{E} \quad \nabla \cdot \bar{B} = 0 \quad \bar{J} = \sum_\alpha q_\alpha \int \bar{u} f_\alpha d\bar{u}
\]

\[
\frac{1}{c^2} \frac{\partial \bar{E}}{\partial t} = \nabla \times \bar{B} - \mu_0 \bar{J} \quad \nabla \cdot \bar{E} = \frac{\sigma}{\varepsilon_0} \quad \sigma = \sum_\alpha q_\alpha \int f_\alpha d\bar{u}
\]
GTS PROGRAMMING ARCHITECTURE

- Most of the code is written in Fortran 90
- F90 makes calls to C to gather particles (interpolates field quantities back to particle space)
  (this happens to be the most significant kernel in terms of execution time)
- Uses MPI, OpenMP
  - OpenMP used for loop level parallelization
- PETSc used for Poisson (field) solves
- For CUDA: map each MPI process to a GPU
  - Current implementation computes push and shift in GPU for electrons and ions
THE PARTICLE-IN-CELL METHOD

- Distribution function information contained in particles
- Obtain macroscopic quantities from distribution function
- Interactions via the grid, on which the potential is calculated (from deposited charges).

The PIC Steps

- “SCATTER”, or deposit charges on the grid (nearest neighbors)
- Solve Poisson equation (SOLVE)
- “GATHER” forces on each particle from potential
- Move particles (PUSH)
- Account for particles that enter and leave the domain (SHIFT)
- Repeat
DOMAIN DECOMPOSITION

- One dimensional decomposition
- Toroidal decomposition - long way round the torus (ntoroidal)
- Particle decomposition - within a toroidal domain, distribute particles in that domain among MPI ranks (npartdom)
- Total number of ranks = ntoroidal*npartdom
PIC SIMULATION WORKFLOW

- Gather+Push (Ion, electron)
- Shift (Ion, electron)
- Charge (Scatter) (Ion, electron)
- Field Solve (Poisson)
CHARGE DEPOSITION (SCATTER)

Charge Deposition Step (SCATTER operation)

- Classic PIC

4-Point Average GK (W.W. Lee)
SOLVE FOR FIELD EQUATIONS

- Poisson equation solved on (fixed) grid
  - This is a strong scaled problem
  - Currently done using PETSc for multigrid
  - Not computationally intensive at large number of processes
    - Fixed grid size
    - Runtime decreases as we increase concurrency (strong scaling)
GATHERING FIELD QUANTITIES BACK TO PARTICLE SPACE AND MOVING PARTICLES

- Gather - reverse of scatter operations (accumulate quantities from grid to particle locations)
- Push particles (Newton’s law)
- Memory bandwidth intensive
SUMMARY OF A TIMESTEPPING LOOP

- Loop:
  - For each electron, ion
    - Do charge deposition on to grid (CPU) “Scatter”
    - Solve field equations on fixed grid (CPU) “Solve”
    - Gather: Interpolate field quantities to particle space and push (GPU) “Push”
    - Move particles leaving toroidal domain and account (GPU) “Shift”

- The ‘gather’ portion is the most computationally intensive at large particle count
PORTING APPROACH

- Use PGI’s CUDA FORTRAN bindings to manage device data and call kernels
  - Pinned memory - add keyword ‘pinned’
  - Device memory pool
  - Function calls (call CUDA C kernels from Fortran code)

- Port ion and electron pushers (actually contains “Gather” and “Push”), written in CUDA C
  - Use textures/LDG to coalesce memory

- Port ion and electron shifters - use existing implementation from Peng Wang
PORTING APPROACH

- Implementation of a particle memory pool
- Allocate a pool of memory as a multiple of the max number of particles (= (large multiple of) * memax)
- Apportion memory using existing templates in multiples of memax as needed for particle arrays
  - eg. We have pool of 100*memax available
    Need 3*memax for particle working array
    Use 3*memax, and do bookkeeping to increment pool pointer location by 3*memax for next allocation
- Do not free CUDA memory - (free() and cudaFree() are potentially expensive and dependent on machine)
PORTING OF “GATHER” STEP TO CUDA

- Ported pushers and shifters for electron and ion
- Call CUDA C from F90
- Perform the gather, and push steps in GPU kernels
- Push:

  ```
  zeon[7*(tid-1)] = zeon0[7*(tid-1)] + dttime*da[tid-1];
  zeon[1+7*(tid-1)] = zeon0[1+7*(tid-1)] + dttime*dgq[tid-1];
  zeon[2+7*(tid-1)] = zeon0[2+7*(tid-1)] + dttime*dgf[tid-1];
  # Need quantities in red to be interpolated from grid after linear solve
  ...
  ```

- Quantities in red are to be ‘gathered’ from grid and interpolated to particle locations
PORTING OF “GATHER” STEP TO CUDA CONT.

- Gathering field quantities
- Bandwidth intensive kernel

- Difficult to coalesce in CPU
- Quantities at red stored in textures and read

```c
f0 = XF*sr[j00] + XF*sr[j10] + XD*sra[j00] + XD*sra[j10];
f1 = XF*sr[j01] + XF*sr[j11] + XD*sra[j01] + XD*sra[j11];
fq0 = XF*srq[j00] + XF*srq[j10] + XD*sraq[j00] + XD*sraq[j10];
fq1 = XF*srq[j01] + XF*srq[j11] + XD*sraq[j01] + XD*sraq[j11];
r = YF*f0 + yF*f1 + YD*fq0 + yD*fq1;
```

Quantities in red are constants calculated at startup, but with irregular access patterns (indices j00, j10, etc.)
PORTING APPROACH

- CUDA textures and LDG

```c
/*
 // z,z'_a,z'_q
 f0  =XF*sz[j00]+xF*sz[j10]+XD*sza[j00]+xD*sza[j10];
 f1  =XF*sz[j01]+xF*sz[j11]+XD*sza[j01]+xD*sza[j11];
 fq0 =XF*szq[j00]+xF*szq[j10]+XD*szaq[j00]+xD*szaq[j10];
 fq1 =XF*szq[j01]+xF*szq[j11]+XD*szaq[j01]+xD*szaq[j11];
 zq  =dY*(f1-f0)+dYD*fq0+dyD*fq1;
 f0  =dX*(sz[j10]-sz[j00])+dXd*sza[j00]+dxD*sza[j10];
 f1  =dX*(sz[j11]-sz[j01])+dXd*sza[j01]+dxD*sza[j11];
 fq0 =dX*(szq[j10]-szq[j00])+dXd*szaq[j00]+dxD*szaq[j10];
 fq1 =dX*(szq[j11]-szq[j01])+dXd*szaq[j01]+dxD*szaq[j11];
 za  =YF*f0+yF*f1+YD*fq0+yD*fq1;
*/

double4 tmp1, tmp2, tmp3, tmp4;

j_array = make_int4(j00,j10,j01,j11);
fetch_double(sz, &tmp1, j_array);
fetch_double(sz+i_sza, &tmp2, j_array);
fetch_double(sz+i_szq, &tmp3, j_array);
fetch_double(sz+i_szq, &tmp4, j_array);

//Handler for double precision textures - extern "C"
static __inline__ __device__ void
fetch_double(double *t, double4 *u, int4 i)
{
    int j00, j01, j10, j11;
    j00 = i.x;
    j10 = i.y;
    j01 = i.z;
    j11 = i.w;
    u->x = __ldg(t+j00);
    u->y = __ldg(t+j10);
    u->z = __ldg(t+j01);
    u->w = __ldg(t+j11);
}
```
SHIFT KERNEL

- Used Peng Wang’s implementation
- Uses thrust and sorting of electrons
- Shifter performance is an impediment to scaling at large concurrency due to communication of large particle buffers
PLATFORMS

- **Machines:**
  - CRAY ‘nano’: 16 core AMD interlagos (based on Titan@ORNL) Kepler K20X
  - Titan: 16 core AMD interlagos, Kepler K20X
  - Perflab cluster: 12 core, dual socket Intel Ivy bridge, Kepler K20X, K40
GTS EXECUTION PROFILE

CPU only run with in 2 nodes MPI+OpenMP

PUSH as a % of overall particle time

- 84% PUSH
- 16% Others

Kernel to focus on is PUSH
WHY THE PORT TO CUDA?

![Diagram showing performance improvements for various tasks with CUDA.]
OVERALL PARTICLE WORK

- More particles => better GPU perf

<table>
<thead>
<tr>
<th>Particles per cell</th>
<th>GPU (100 P/cell)</th>
<th>CPU (10 P/cell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100P</td>
<td>30</td>
<td>112</td>
</tr>
<tr>
<td>10P</td>
<td>4</td>
<td>11.7</td>
</tr>
</tbody>
</table>

Speedup: 3.8 for 100 P/cell, 2.9 for 10 P/cell
PARTICLE VS GRID WORK

- Grid work (Poisson solve) is constant for problem
WEAK SCALING RUNS AT TITAN GIVE 3X CODE SPEEDUP

- Good weak scaling performance at 8000 nodes

AMD Interlagos (16 cores per node)
32 G RAM per node
K20X (1 GPU per node)
Runs use 16 OMP threads per node
Each MPI process is mapped to a GPU
GRID WORK BECOMES NEGLIGIBLE AT HIGHER CONCURRENCY

- Contribution from grid work negligible for large runs
- Particle work dominant

AMD Interlagos (16 cores per node)
32 G RAM per node
K20X (1 GPU per node)
Runs use 16 OMP threads per node
Each MPI process is mapped to a GPU
CODE IS MEMCPY BOUND

Contribution of "PUSH" to walltime

- PUSH: 42%
- Others: 58%

"PUSH" kernel

- Memcpy: 76%
- Others: 24%

Memcpy optimizations are key
RELATIVE PUSH PERFORMANCE OF K40 TO K20X

**cudaMemcpy**
- K20X: 8 seconds
- K40: 4 seconds
- 40% improvement

**"PUSH"**
- K20X: 10 seconds
- K40: 7 seconds
- 30% improvement

**Wall**
- K20X: 25 seconds
- K40: 20 seconds
- 20% improvement
K40 ADVANTAGE ON PUSH

- **Reason** - K40 uses PCI Gen 3, and K20X uses PCI Gen 2
  - Faster transfer of data across the PCI bus (cudaMemcpy)
- Currently, GTS spends most (~50%) of its time in transferring data to and from the device
- Improved memory capacity (run a larger problem, increase programmer happiness by allowing freer use or arrays)
- **Speculation**: K40 might help scale better
  - Run on 1 MPI rank instead of 2 MPI ranks: shifter perf at large concurrency
CONCLUSIONS

- Port gives speedup of 3-4x for code, 5-10x for push kernels
- K40 performs 20% better than K20X due to PCI Gen3 in K40 vs PCI Gen2 in K20X
- Good weak scaling seen up to 8000 nodes in Titan
- Potential future work
  - Charge deposition (plausible due to improvements in atomics in kepler)
  - Investigate shifter performance at large concurrency, and how it maps with K40 performance