EXPOSING PARTICLE PARALLELISM IN THE XGC PIC CODE BY EXPLOITING GPU MEMORY HIERARCHY

Stephen Abbott, March 26 2018
ACKNOWLEDGEMENTS

Collaborators:

Oak Ridge Nation Laboratory - Ed D’Azevedo

NVIDIA - Peng Wang


And many others in the XGC team!

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.
OUTLINE

1. A very brief review of the Particle-In-Cell cycle
2. Overview of the XGC1 Approach
3. Optimized Particle Pushing with CUDA Fortran/C
THE PIC CYCLE

1. SCATTER: particle attributes to a mesh/grid
2. PUSH: each particle
3. SOLVE: for the fields on the mesh/grid
4. GATHER: the fields to each particle
THE XGC1 ALGORITHM

XGC is a full-f gyrokinetic particle-in-cell/finite element code that solves the 5D Gyrokinetic equation on a field-following triangular mesh.

A time-step is (minimally)

1) Collect charge density to mesh
2) Solve the GK Poisson equation
3) Compute $E$ and derivatives
4) Calculate diagnostics
5) **Push particles**
6) Calculate collisions and source
THE XGC1 ELECTRON PUSHER

- Electrons stepped $O(40)$ times per ion step
- 4th order Runge-Kutta per step reduces numerical heating
- Electron sub-cycling (PUSHE) done without communication
- Even after initial CUDA Fortran port to GPUs, was a major cost
- Deep call tree makes optimization difficult
- Kernel is limited by memory access, not FLOPS

To make the kernel faster: understand data structures, how particles move across the grid, and map to GPU memory regions
THE XGC1 ELECTRON PUSHER
DATA STRUCTURES

Unstructured Mesh Data: $\phi / E$

**Data characteristics**
- $\phi / E$ field lives on field-following triangular mesh
- $O(10^6)$ triangles for ITER mesh
- Pre-computed node-to-node field following between $O(30)$ toroidal $\varphi$-planes, with node number for lookup.

2D Uniform Grids: $\psi^0(\tau, z)$

**Data characteristics**
- Equilibrium flux function on uniform 2D grid read in from EQDSK file
- Interpolation to particle positions using PSPLINE bicubic splines
- Extract spline coefficients for GPU

1D Uniform Grids: $I(\psi^0)$

**Data characteristics**
- Equilibrium current read in from EQDSK file
- Uniform 1D grid in $\psi^0$ space
- Interpolate $\psi^0$ from particle position, then $I(\psi^0)$ using PSPLINE
- Extract spline coefficients for GPU

Particles

The only read-write data structures for the electron sub-cycling.
- CPU storage is A.o.S.
- Per particle:
  - 6 R/W phase variables (8B each)
  - 3 RO constants (8B each)
  - 1 RO integer uid (8B)
THE XGC1 ELECTRON PUSHER
IDEAL GLOBAL TRANSACTIONS

Assumptions
Best to read once, write once (or not at all)
Particle access are perfectly coalesced (128B lines)
1.57 \times 10^6 particles (a medium-sized test)

Minimum Necessary
1.72 \times 10^6 load transactions
6.37 \times 10^5 store transactions

‘Un-optimized’ Profile
(has algorithm improvements)
2.58 \times 10^8 load transactions
3.20 \times 10^6 store transactions

Initial profiling shows \sim 100X loads and \sim 5X stores
Per particle data is small, but there’s a lot of it. Two important considerations:
1. The default Array-of-Structs storage on CPU is un-coalesced on GPUs.
2. Particle data is used throughout the pushe call tree.

AoS on host

SoA in GPU
  global memory

Pack into thread
  private struct
K20X has a 48KB Texture/Constant cache with specialized access paths

<table>
<thead>
<tr>
<th>CUDA Fortran</th>
<th>CUDA C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>texture, pointer</code></td>
<td>Texture Objects and References, <code>__ldg</code></td>
</tr>
<tr>
<td><code>intent(in), compiler analysis</code></td>
<td><code>const __restrict__</code></td>
</tr>
</tbody>
</table>

1D grids can be bound in linear memory in CUF

2D grids benefit >15% from proper CUDA textures, but HW filtering is too low precision.
The electric field array is large

- 3 components
- 2 directions (along B field)
- \( N_{\text{phi}} \) (4-32) planes (toroidal discretization)
- \( N_{\text{nodes}} \) (50K – 1M on mesh)

Touched non-uniformly 4x per particle

**Extreme register, L1/L2, and DRAM pressure!**

Can over-run \( 2^{27} \) element limit on linear textures

**Solution**

Must decompose. Basic splitting works for now, not sustainable for larger grids

Must still optimize access pattern (bonus material)
Coalesced loads and cache reuse fundamental for good performance

Frequent sorting helps

Sorting to a coarse Cartesian grid ignores true dimensionality of the underlying data structures (the triangular mesh)

Instead sort by the last know triangle number.

More keys means slower $O(n+k)$ count sort, but extra cost is worth it

Similarly, particles move slow enough per time-step that it’s worth checking last know triangle before conducting a lookup table search
## PERFORMANCE IMPACT

<table>
<thead>
<tr>
<th></th>
<th>3mm ITER</th>
<th>12mm ITER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>38s (100%)</td>
<td>23s (100%)</td>
</tr>
<tr>
<td>Ptl Mapping</td>
<td>28s (73%)</td>
<td>16s (70%)</td>
</tr>
<tr>
<td>Tr. check</td>
<td>15s (39%)</td>
<td>12s (52%)</td>
</tr>
<tr>
<td>Triangle sort</td>
<td>13s (34%)</td>
<td>11s (48%)</td>
</tr>
<tr>
<td>2D Textures</td>
<td>9.2s (24%)</td>
<td>7.7s (33%)</td>
</tr>
<tr>
<td>Mesh Tex.</td>
<td>7.8s (21%)</td>
<td>7.2s (31%)</td>
</tr>
<tr>
<td>Compiler Options</td>
<td>7.7s (20%)</td>
<td>7.0s (30%)</td>
</tr>
<tr>
<td>Current</td>
<td>14s (37%)</td>
<td>12s (52%)</td>
</tr>
<tr>
<td>Best with OpenACC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Graph:**

- **X-axis:** Unoptimized, Stage 1, Stage 2, Fully Optimized
- **Y-axis:** DRAM Global Transactions
- **Legend:**
  - Blue: Loads
  - Green: Stores

**Legend:**

- **Ideal Load**
- **Ideal Store**
SUMMARY

Reducing the impact of mesh and cell data is critical to performance in XGC

CUDA Fortran has most of the tools needed, CUDA C/C++ has the rest

OpenACC compilers can get close, but not close enough

Testing is necessary, and alleviates portability impact
BONUS COMPILER OPTIMIZATION FLAGS FOR PGI

For CUDA Fortran with –Mcuda=….  
7.5,cc35 (on K20X), 8.0,cc60 (on Pascal), 9.1,cc70 (on Volta) to optimize better  
madconst to put module arrays descriptors in cmem  
maxrregcount=N to restrict register usage  
ptxinfo,keepgpu,keeptx to give info on generated source

For OpenACC with –acc –ta=tesla:  
Same as above, but no madconst
## USING OPTIMIZED CUDA WITH OPENACC

### Texture Cache Usage (1.58 x 10^6 particles)

<table>
<thead>
<tr>
<th></th>
<th>CUDA Fortran</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Explicit &quot;texture, pointer&quot;</td>
<td>Use <code>intent(in)</code> to hint <code>__ldg</code></td>
</tr>
<tr>
<td></td>
<td><code>intent(in)</code> hints to use <code>__ldg</code></td>
<td><code>!$acc copyin(..)</code></td>
</tr>
<tr>
<td>Un-cached Global Loads</td>
<td>2.0 x 10^6</td>
<td>2.5 x 10^8 (6.43 x 10^7)</td>
</tr>
<tr>
<td>Texture Cache Hit Rate</td>
<td>76%</td>
<td>99.9% (73%)</td>
</tr>
<tr>
<td>L2 Texture Reads</td>
<td>1.1 x 10^8</td>
<td>500 (9.1 x 10^7)</td>
</tr>
<tr>
<td>L2 Texture Read Hit Rate</td>
<td>94%</td>
<td>99.6% (93%)</td>
</tr>
</tbody>
</table>

Deep call trees are hard for the compiler to tune

Value safety over performance

Developers have finite patience

```fortran
subroutine cuda_bi2(..)
!$acc routine seq nohost
(wrapper for CUDA C)
end subroutine cuda_bi2

subroutine bicub_interpol2(..)
!$acc routine seq bind(cuda_bi2)
(host-side routine)
end subroutine bicub_interpol2
```