Daino: A High-level Framework for Parallel and Efficient AMR on GPUs

Mohamed Wahib\textsuperscript{1}, Naoya Maruyama\textsuperscript{1,2}, Takayuki Aoki\textsuperscript{2}

\textsuperscript{1}RIKEN Advanced Institute for Computational Science, Kobe, Japan
\textsuperscript{2}Tokyo Institute of Technology, GSIC, Tokyo, Japan

11\textsuperscript{th} May 2017
GTC17
Motivation & Problem:
- “AMR is one of the paths to multi-scale exascale applications”
- Producing efficient AMR code is hard (especially for GPU)

Solution:
- A framework for producing efficient AMR code (for GPUs)
- Architecture-independent interface provided to the user
- A speedup model for quantifying the efficiency of AMR code

Key results: We evaluate three AMR applications
- Speedups & scalability comparable to hand-written code (~3,642 K20x GPUs)
Adaptive Mesh Refinement (AMR)

- For meshes in some simulations using PDEs:
  - We only require high resolution for areas of interest
  - Resolution changes dynamically during simulation
  - Achieving efficient AMR is challenging
    - Managing an adaptive mesh can be complicated
    - Balancing compute load and communication costs
Structured Tree-based AMR

- Many ways to represent the mesh
  - We focus on octree representation (quadtree in 2D)
  - Mesh divided into blocks, refine/coarsen if required

Octree-based meshes: (a) Adaptive mesh (b) Tree representation

Operations applied on tree are distributed
How AMR Works

- Initialize the Mesh
- **FOR** Simulation time **DO**
  - **IF** time to remesh
  - **ENDIF**
  - **IF** time to load balance
  - **ENDIF**
- **ENDFOR**

Computation

Remeshing

Load balancing

Reduced Computation (less data in mesh)

Overhead
AMR on GPUs

- Hard to achieve efficient AMR with GPUs

- Few existing AMR frameworks support GPU:
  1. User must provide code optimized for GPU
  2. Scalability problems due to CPU-GPU data movement
  3. No speedup-bound model

Contributions of our framework
Framework for Efficient AMR

- A compiler and runtime

**Input:**
- Serial code applying stencil on a uniform grid
- User adds directives to identify relevant data arrays
  - Architecture-neutral

**Output:**
- Executable binary for target architecture
- Code is parallel and optimized for GPU (MPI+CUDA)
#pragma daino

void 3D_alloy(..)
{
    #pragma daino
data (Nx,Ny,Nz)
    {p, u, dpt, no, o;}

    ... kernel code ...
}

---

**AMR frameworks**

CUDA code

```c
__global__ 3D_alloy(..)
{
    ... CUDA kernel code ...
}
```

OpenMP Code

```c
void 3D_alloy(..)
{
    #pragma omp for
    ... kernel code ...
}
```

**Our framework**

Uniform Mesh Serial C Code

```c
#pragma daino kernel
void 3D_alloy(..)
{
    #pragma daino data (Nx,Ny,Nz)
    {p, u, dpt, no, o;}
    ... kernel code ...
}
```

---

**Two benefits:**

- Productivity
- Ability to apply low-level GPU optimizations

---

Framework

GPU AMR Executable

CPU AMR Executable

Framework

GPU AMR Executable

CPU AMR Executable
Minimal example of using directives in our framework

```c
#pragma dno kernel
void func(float ***a, float ***b, ..) {
    #pragma dno data domName(i, j, k)
a, b;
    #pragma dno timeloop
    for(int t; t< TIME_MAX; t++) {
        for(int i; i<NX; i++)
            for(int j; i<NY; j++) {
                ... // comput. not related to a and b
                for(int k; k<NZ; k++) {
                    a[i][j][k] = c * (b[i-1][j][k]
                                      + b[i+1][j][k] + b[i][j][k]
                                      + b[i][j+1][k] + b[i][j-1][k]);
                }
            }
        }
    }
}
```

A target kernel

Data arrays + iterators

Target loop
Scalable AMR: Data-centric Model (1 of 2)

- A data-centric approach
  - Each computing element specializes on its data
  - Blocks on GPU, octree data structure on CPU
  - Migrate all operations touching block data to GPU
  - CPU only processes octree data structure
Scalable AMR: data-centric Model (2 of 2)

- All kernels are data parallel (i.e. well-suited to GPU)

AMR promises reduced computation

- Problem → overhead in managing hierarchal mesh

Project speedup bound

- Informs framework designer of → efficiency of AMR code
  - Compare achieved speedup vs. projected upper-bound speedup

- Takes into account AMR overhead

- If projected speedup ⇐ far from → achieved speedup
  - Some AMR overhead(s) not properly accounted for
Figure 1: Overview of framework implementation

Apply translations and optimizations as passes
The Daino framework overview. Application C code is transformed to an optimized executable. Daino components enclosed in red dotted line.
Runtime Libraries

- AMR Management
  - Maintain the octree
  - Orchestration of work
  - Memory manager
    - Especially important with GPU

- Communication
  - MPI processes
  - Halo data exchange
    - Transparent access to blocks
  - Moving blocks (load balancing)
<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hydrodynamics Solver</td>
<td>A 2(^{nd}) order directionally split hyperbolic schemes to solve Euler equations. [RTVD scheme modified from GAMER(^{1})]</td>
</tr>
<tr>
<td>Shallow-water Solver</td>
<td>We model shallow water simulations by depth-averaging Navier–Stokes equations. [2(^{nd}) order Runge-Kutta method]</td>
</tr>
<tr>
<td>Phase-field Simulation</td>
<td>3D dendritic growth during binary alloy solidification(^{2}) [Time integration by Allen-Chan equation]</td>
</tr>
</tbody>
</table>

Results (1 of 4)

- We use TSUBAME2.5 supercomputer (TokyoTech)
- Up to 3,642 K20x GPUs
- TSUBAME Grand Challenge Category A (full machine)

Weak scaling of uniform mesh, hand-written and automated AMR (GAMER-generated AMR included in hydrodynamic)
Results (2 of 4)

- **Notes:**
  - Phase-field achieves 1.7x speedup
    - Original implementation is Gordon Bell 2011 winner
  - Daino is faster than GAMER AMR version
  - GAMER is a leading framework for AMR over GPUs

---

**Strong scaling of uniform mesh, hand-written and automated AMR (GAMER-generated AMR included in hydrodynamic)**
Results (3 of 4)

Overhead of the AMR framework (weak scaling):

<table>
<thead>
<tr>
<th>GPUs</th>
<th>Stencil (GPU)</th>
<th>AMR</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>88.7%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>87.2%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3600</td>
<td>84.3%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remeshing kernels are well-suited to GPU.
Results (4 of 4)

- Efficiency of transformation:
  - Achieved speedup > 86% of practical limit

Speedup: measured vs. projected. $M$ is measured, $P$ is the practical AMR speedup projection, and $T$ is the theoretical AMR speedup projection.
Summary

Problem:
- AMR is one of the paths to multi-scale exascale applications
- Producing efficient AMR code is hard (especially for GPU)

Solution:
- A framework for producing efficient AMR code (for GPUs)
- Architecture-independent interface provided to the user
- A speedup model for quantifying the efficiency of AMR code

Key results: We evaluate three AMR applications
- Speedups & scalability comparable to hand-written code (3,642 K20x GPUs)
Future Work

- Expand Daino
  - Incorporate Daino’s GPU backend in other AMR framework
  - Work-in-progress for porting new applications (CFD)

- Supporting user-specified boundary conditions, equations of state, and flux corrections

- Extend support for Intel Xeon Phi (KNL)
  - We already introduced experimental support for OpenMP (not fully optimized)

- Leverage the speedup model analysis
  - Auto-tuning

Daino will be publically released at: http://github.com/wahibium/Daino
Thank you for listening.

Questions?