The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System

Qingyu Meng, Alan Humphrey, Martin Berzins

Thanks to: John Schmidt and J. Davison de St. Germain, SCI Institute
Justin Luitjens and Steve Parker, NVIDIA

DOE for funding the CSAFE project (97-10), DOE NETL, DOE NNSA
NSF for funding via SDCI and PetaApps
Current and Past Uintah Applications

- Explosions
- Sandstone Compaction
- Angiogenesis
- Gas mixing
- CPU pins
- Coal Boiler
- Shaped Charges
- Foam Compaction
- Virtual Soldier
- Fires
Uintah Data Parallelism

Uintah uses both data and task parallelism

- Structured Grid (Flows) + Particles System (Solids)
- Patch-based Domain Decomposition for Parallel Processing
- Adaptive Mesh Refinement
- Dynamic Load Balancing
  - Profiling + Forecasting Model
  - Parallel Space Filling Curves
  - Data Migration
Uintah Task Parallelism and Uintah Task Graph

Patch-based domain decomposition

User defines Uintah Tasks:
- **Serial** code (callback functions)
- **Input** and **output** variables
- **Distributed**: only creates tasks on local patches
- Framework analyzes task dependencies and creates TG
- **Automatic** MPI message generation
- **Dynamic** Task Execution (Data Driven Overlap)
Uintah Runtime System: How Uintah Runs Tasks

**Memory Manager:** Uintah Data Warehouse (DW)
- Variable dictionary (hashed map from: Variable Name, Patch ID, Material ID keys to memory)
- Provide interfaces for tasks to
  - Allocate variables
  - Put variables into DW
  - Get variables from DW

**Automatic Scrubbing** (garbage collection)

**Checkpointing & Restart** (data archiver)

**Task Manager:** Uintah schedulers
- Decides when and where to run tasks
- Decides when to process MPI
Thread/MPI Scheduler (De-centralized)

- **Memory saving**: reduce ghost copies and metadata
- **Work stealing** inside node: all threads directly pull tasks from task queues, no on-node MPI
- **Full Overlapping**: All threads process MPI sends/receives and execute tasks
- Use **lock-free** data structure (avoid locking overhead)
Scalability Improvement

Original Dynamic MPI-only Scheduler
De-centralized MPI/Thread Hybrid Scheduler (with Lock-free Data Warehouse)

- Achieve much better CPU Scalability
- 95% weak scaling efficiency on 256K cores (Jaguar XK6)
- Use GPUs to accelerate Uintah Components
First step to GPU

- **Profile** & find most time consuming task
- **Port** task’s serial CPU code to GPU
- Call CUDA API inside task code
- Framework **unaware** of GPU(s)
- **Result:** ~2x speedup (stencil code)
  - must hide PCIe latency
Uintah GPU Task Management

Framework manages all CUDA data movement (NOT inside task)

- Use **Asynchronous API**
- **Automatically** generate CUDA stream for each dependency
- **Concurrently** execute kernels and memcopies
- **Prefetching** data before task kernel execute
- **Multi-GPU** support

- Two call back functions for both CPU version and GPU version: **Compatible** for non-GPU nodes

---

Stages of GPU task in Uintah runtime

1. Pin this memory with `cudaHostRegister()`
2. Host Requires
3. Dev Requires
4. Computation
5. Host Computes
6. Free pinned host memory

---

Normal

Data Transfer | Kernel Execution | Data Transfer

GPU Task

---

Page-locked Memory

Data Transfer | Kernel Execution | Data Transfer

GPU Task
Multistage Task Queues

Architecture

Fully Overlap computation with PCIe transfers and MPI communication
GPU RMCRT Speedup Results
(Multi-Node)

<table>
<thead>
<tr>
<th>Keeneland Initial Delivery System</th>
<th>Nodes</th>
<th>CPU(sec)</th>
<th>CPU+GPU (sec)</th>
<th>Speedup (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>319.769</td>
<td>8.49714</td>
<td>38 X</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>163.773</td>
<td>7.68571</td>
<td>21X</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>70.0943</td>
<td>4.80571</td>
<td>14X</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>39.5686</td>
<td>2.62857</td>
<td>15X</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>20.2414</td>
<td>3.52857</td>
<td>6X</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>18.8043</td>
<td>2.73857</td>
<td>6X</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>9.11571</td>
<td>2.95714</td>
<td>3X</td>
<td></td>
</tr>
</tbody>
</table>

CPU Core – (2) Intel Xeon 6-core X5660 (Westmere) @2.8GHz
GPU – (1) Nvidia M2090

* GPU implementation quickly runs out of work, scaling breaks down
Scaling Comparisons

- Uintah strong scaling results when using:
  - MPI-only
  - Multi-threaded MPI
  - Multi-threaded MPI w/ GPU

- Two Problems:
  - AMR MPMICE
    - Nearest neighbors communication
    - 3.62 billion particles
  - GPU-enabled ray tracer
    - All-to-all communication
    - 100 rays per cell 128^3 cells

Uintah Scaling Overview

- MPI only AMR MPMICE: N=6144 CPU cores; Largest = 98K CPU cores
- Thread/MPI AMR MPMICE: N=8192 CPU cores; Largest=256K CPU cores
- Thread/MPI RayTracing: N=16 CPU cores; Largest=1024 CPU cores
- Thread/MPI/GPU RayTracing: N=16 CPU and 1 GPU; Largest=1024 CPU and 64 GPU
Future Work

Scheduler – Infrastructure

- GPU affinity for multi socket/GPU nodes
- Support Intel MIC (Xeon Phi) offload-mode
- PETSc GPU interface utilization
- Mechanism to dynamically determine whether to run GPU or CPU version task

- Optimize GPU codes for Nvidia Kepler
  - CUDA 5.0 – Dynamic Parallelism
  - GPU sub-scheduler
Questions?

Alstom Clean Coal Boiler Simulation
Monte Carlo Ray Tracing on GPU, Flow simulation on CPU

Software Homepage  http://www.uintah.utah.edu/
Using GPUs in Energy Applications

- ARCHES Combustion Component
  - Alstom Clean Coal Boiler Problem
- Need to approximate radiation transfer equation
- Reverse Monte Carlo Ray Tracing (RMCRT)
  - Rays mutually exclusive - traced simultaneously
  - Ideal for SIMD parallelization of GPU
- Offload Ray Tracing and RNG to GPU(s)
  - CPU cores can perform other computation
## Performance Comparisons

### Master-Slave Model vs. Unified

#### Execution Times – CPU Only

<table>
<thead>
<tr>
<th>#Cores</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Slave</td>
<td>57.28</td>
<td>20.72</td>
<td>9.40</td>
<td>4.81</td>
<td>2.95</td>
</tr>
<tr>
<td>Unified</td>
<td>29.79</td>
<td>15.70</td>
<td>8.23</td>
<td>4.54</td>
<td>2.78</td>
</tr>
</tbody>
</table>

#### Execution Times – With GPU

<table>
<thead>
<tr>
<th>#Cores</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Slave</td>
<td>4.55</td>
<td>4.09</td>
<td>3.95</td>
<td>3.68</td>
<td>3.64</td>
<td>3.34</td>
</tr>
<tr>
<td>Unified</td>
<td>3.82</td>
<td>3.52</td>
<td>3.09</td>
<td>2.90</td>
<td>2.50</td>
<td>2.09</td>
</tr>
</tbody>
</table>

**CPU Problem:** Combined MPMICE problem using AMR  
Run on single Cray XE6 node with two 16-core AMD Opteron 6200 Series (Interlagos cores @2.6GHz) processors

**GPU Problem:** Reverse Monte Carlo Ray Tracer  
Run on a single 12-core heterogeneous node (two Intel Xeon X5650 processors each with Westmere 6-core @2.67GHz, (2) Nvidia Tesla C2070 GPUs and (1) Nvidia GeForce 570 GTX GPU)