Tiling-Based Programming Model for Structured Grids on GPU Clusters

Burak Bastem, Didem Unat,
Koç University, Istanbul, Turkey
Outline

➢ Motivation
  o Overview
  o Implementation
  o Performance
  o Related Work
  o Future Work
Motivation

• Moore’s Law is no longer applicable
• GPUs have massively parallel and power-efficient architectures that accelerate data-parallel applications
• As a result GPU-based heterogenous systems became popular
  – They constitute more than 25% of supercomputers on TOP500 list
    • The percentage keeps increasing
Motivation

- Programming a single-GPU system is a demanding task because it requires
  - Managing distinct address spaces
  - Implementing GPU-specific code (kernels)
• Having **multiple GPUs on a host**
  – Complicates address space management
  – Additionally requires distributing the work
Motivation

- Programming a **GPU cluster** is even more demanding since it additionally requires
  - handling communication between host
Motivation

• The interconnect linking GPUs to host has a lower bandwidth than host and device have
  – Transfers between hosts and devices need to be optimized
• Communication across cluster needs to be optimized for better scalability
✓ Motivation

Overview

- Implementation
- Performance
- Related Work
- Future Work
Tiling-Based Programming Model

(a) ghost cell updates between hosts

(b) ghost cell updates between hosts and devices

(c) ghost cell updates between devices
Tiling-Based Programming Model

- Manages distinct address spaces itself
- Automatically generates kernels
- Handles transfers between hosts and devices and communication across cluster
  - Also overlaps them with computation
    - Provides a solution for the interconnect bandwidth limit
Fundamental Data Structures

- Programming with Tiling Data Abstractions (TiDA)
  - **Tile**: Data partition
  - **Tile Array**: Responsible for partitioning, memory management, ghost-cell update
  - **Tile Iterator**: Iterates over tiles on CPUs and/or GPUs
1 TileArray ta(app_data, data_dim, tile_dim, ghost_cell_dim);
2 for(TileArray::Iterator i = ta.begin(gpu_exec_ratio); i != ta.end(); ++i) {
3   Tile t = *i;
4   compute(t, [](double* data, int depth, int height, int width, int index){
5     data[index] = ... // computation
6   });
7 }
8 ta.updateGhostCells();
Line 1 creates a tile array
Line 2 starts an iteration with for-loop
  - Syntax is the same as the syntax of iterating through a standard C++ list

• **gpu_exec_ratio** specifies the fraction of tiles that will be executed on GPUs
Interface

```cpp
1 TileArray ta(app_data, data_dim, tile_dim, ghost_cell_dim);
2 for(TileArray::Iterator i = ta.begin(gpu_exec_ratio); i != ta.end(); ++i) {
3   Tile t = *i;
4   compute(t, [](double* data, int depth, int height, int width, int index){
5     data[index] = ... // computation
6   });
7 }
8 ta.updateGhostCells();
```

- Line 3 gets a tile by dereferencing the iterator
compute is a function which takes the tile and a lambda expression

- Lambda contains the computation and hides kernel code generation
Interface

```c
1 TileArray ta(app_data, data_dim, tile_dim, ghost_cell_dim);
2 for(TileArray::Iterator i = ta.begin(gpu_exec_ratio); i != ta.end(); ++i) {
3 Tile t = *i;
4 compute(t, [](double* data, int depth, int height, int width, int index){
5 data[index] = ... //computation
6 });
7 }
8 ta.updateGhostCells();
```

- Line 8 updates ghost cells
✓ Motivation
✓ Overview
➢ Implementation
  o Performance
  o Related Work
  o Future Work
Implementation

• Implemented as a library
  – https://bitbucket.org/parcorelab/gpu-cluster-tiling-programming-model.git
• Uses multiple processes
  – Each process gets a GPU
• Partitions data into tiles with TiDA
• Manages memories with CUDA
• Leverages OpenACC for kernel generation
  – Hides directives with lambda expression
• Exploits CUDA streams for asynchronous GPU operations
• Uses non-blocking MPI routines to hide communication behind computation
Memory Management

Three different memory options:

- **Pageable**
  - During a memory transfer between host and device, pageable memory implicitly copied from/to pinned memory
  - Default memory allocations are pageable

- **Pinned**
  - Programmer can directly allocate pinned memory

- **Unified**
  - Host and device memory appears a single memory
Performance of Memory Types

<table>
<thead>
<tr>
<th>Memory Configuration</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenACC Pinned Memory</td>
<td>1.61</td>
</tr>
<tr>
<td>OpenACC Pageable Memory</td>
<td>2.35</td>
</tr>
<tr>
<td>OpenACC Managed Memory</td>
<td>2.86</td>
</tr>
<tr>
<td>CUDA Pinned Memory + OpenACC</td>
<td>1.62</td>
</tr>
<tr>
<td>CUDA Pageable Memory + OpenACC</td>
<td>1.94</td>
</tr>
<tr>
<td>CUDA Unified Memory + OpenACC</td>
<td>2.42</td>
</tr>
<tr>
<td>CUDA Pinned Memory</td>
<td>1.34</td>
</tr>
<tr>
<td>CUDA Pageable Memory</td>
<td>1.67</td>
</tr>
<tr>
<td>CUDA Unified Memory</td>
<td>2.12</td>
</tr>
</tbody>
</table>

- **GPU**: NVIDIA K40
- CUDA + OpenACC versions use CUDA for memory management and OpenACC for kernel generation
- The library uses CUDA pinned memory for memory management

15.01.2020
Burak Bastem @ HPC Asia 2020
Kernel Generation

• Using CUDA
  – Forces programmer to implement kernels
  – Or requires in-house compiler implementation which needs support for longevity

✓ Using OpenACC
  – Directives are easy to use
  – Directives can be hidden in library
  – Supported and maintained by NVIDIA
Overlapping Memory Transfers

- Employs CUDA streams to overlap device operations
  - Stream is a sequence of device operations
  - GPUs can execute different streams concurrently
- Assigns a CUDA stream to each tile
  - Operations of a tile overlaps with operations of other tiles
  - In case a GPU has limited memory, some tiles share streams
    • Overlapping prevents performance penalty

- Avoids unnecessary memory transfers with
  - Cache mechanism
  - Lazy initialization
• Utilizes non-blocking MPI routines
• Employs packing-unpacking for ghost cells
• Supports GPUDirect with CUDA-aware MPI
  – GPUDirect Peer-to-Peer (P2P) Transfer
• Supports GPUDirect with CUDA-aware MPI
  – GPUDirect Remote Direct Memory Access (RDMA)
• Phase I: push stream events to each stream of region
• Phase II: pack and send
  – Initiate non-blocking MPI receives
  – Pack ghost cells to buffers (initiate the ones on the device)*
  – Send the ones on the host with non-blocking MPI send
• Phase III: send
  – Sync packing of ghost cells on the device
  – Send them with non-blocking MPI send
• Phase IV: transfer
  – Initiate packing ghost cells on the device that needs to be sent to the host
  – Initiate transfer of these buffers
  – Push transfer event to their stream
  – Pack ghost cells on the host that needs to be sent to the device
  – Initiate transfer of these buffers
• Phase V: exchange cells of the regions on the same device and process
  – Sync streams to stream events
• Phase VI: Unpack buffers sent with MPI*
  – MPITest while all packages are received
• Phase VII: transfer
  – Initiate unpacking of buffer received on device
  – Sync stream to transfer event for the buffer sent to host
  – Unpack the buffer sent to host
Outline

- Motivation
- Overview
- Implementation
  - Performance
    - Related Work
    - Future Work
Applications

• **Heat simulation**
  – Computes heat transfer equation
  – For each point in 3D space performs 7-point stencil at each timestep
    • 100 timesteps
  – Memory intensive

• **Cardiac modeling**
  – Simulates the propagation of electrical signals in the cardiac tissue
  – In 2D space uses the Aliev-Panfilov model
    • 1350 timesteps
  – Compute intensive
Performance Study

• All speedups are reported against a baseline
  – Manages memory manually with CUDA
  – Uses pinned memory for host allocations
  – Generates GPU kernels with OpenACC annotations

• In all experiments, the measured time includes
  – Computation
  – And the time required for data transfers between hosts and devices.
GPU Cluster

Summitdev:

- 54 nodes
  - 3 racks each with 18 nodes
  - Connected in a full fat-tree via EDR InfiniBand

- 108 IBM POWER8 CPUs
  - Each node has 2 CPUs
  - 10 core per CPU

- 216 NVIDIA P100 GPUs
  - Each node has 4 GPUs
  - Connected via NVLink 1.0
### Single GPU Performance

**Speedup over *Baseline on a Single GPU**

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Tiles</td>
<td>1,80</td>
</tr>
<tr>
<td>Single Tile</td>
<td>1,12</td>
</tr>
<tr>
<td>*CUDA Pinned Memory + OpenACC</td>
<td>1,00</td>
</tr>
<tr>
<td>OpenACC Managed Memory</td>
<td>0,31</td>
</tr>
<tr>
<td>OpenACC Pinned Memory</td>
<td>1,00</td>
</tr>
<tr>
<td>OpenACC Pageable Memory</td>
<td>0,71</td>
</tr>
<tr>
<td>CUDA Unified Memory</td>
<td>0,64</td>
</tr>
<tr>
<td>CUDA Pinned Memory</td>
<td>1,24</td>
</tr>
<tr>
<td>CUDA Pageable Memory</td>
<td>1,08</td>
</tr>
</tbody>
</table>

#### Heat Simulation

- 96% of GPU memory is used
- Multiple tiles: 128
- 80% performance increase with overlapping
%96 of GPU memory is used
Multiple tiles: 81
89% performance increase with overlapping
Cluster Performance

Heat Simulation - Strong Scaling

Number of GPUs

0 1 2 4 8 16 32 48 64 96 128 192

Speedup

0 10 20 30 40 50 60 70

97% 48% 24% 12% 6% 3% 0.5%

Compact  Scatter  GPU Memory Consumption

Burak Bastem @ HPC Asia 2020
Cluster Performance

Cardiac Simulation - Strong Scaling

Speedup vs. Number of GPUs

- Compact
- Scatter
- GPU Memory Consumption

Burak Bastem @ HPC Asia 2020
Outline

✓ Motivation
✓ Overview
✓ Implementation
✓ Performance
➢ Related Work
○ Future Work
Related Work

• Well-known programming models
  – **CUDA, OpenCL**: Memory management with functions and low-level kernel implementation
  – **OpenACC, OpenMP**:Pragma-based programming models
  – No support for GPU Clusters
    • **CUDASA, CudaMPI, dCUDA, SnuCL** and **dOpenCL** introduce communication primitives for CUDA and OpenCL
    • I) Wu et al., 2016, II) Aji et al., 2016, etc. modify MPI to optimize communication from GPU memory
    • I) Komoda et al., 2013 and II) Matsumura et al., 2018 extend OpenACC for single-host-multi-GPU systems
Related Work

• Pragma-based other programming models with their in-house compilers
  – I) Unat et al., 2011 and II) Lee and Eigenmann, 2010
  – Difficult to maintain in-house compilers

• Instead of hiding memory management, some works provide high-level means
  – Thrust, Kokkos, C++ AMP, etc.

• Task-based programming models
  – I) Agullo et al., 2018, II) Grasso et al., 2014, etc.
  – Programmer is responsible from kernel implementations and task scheduling

• Skeleton-based programming models
  – Muesli, Cluster-SkePU, etc.
  – Skeletons arguably reduce programming flexibility

! Most of the studies do not consider the low interconnect bandwidth
Outline

✓ Motivation
✓ Overview
✓ Implementation
✓ Performance
✓ Related Work
➢ Future Work
• Auto tuning of tile size
• Performance study and auto tuning of hybrid (CPU-GPU) execution
• Supported by the Turkish Science and Technology Research Centre Grant No: 215E185.

• Utilized resources from Lawrence Berkeley National Laboratory (LBNL), Oak Ridge National Laboratory (ORNL) and Swiss National Supercomputing Center (CSCS).

• Received conference and travel support from Yapı Kredi Teknoloji.

Thank you for listening!
APPENDIX
Summitdev

• Had an issue on GPU Inter-Process Communication
  – Prevented us from conducting experiments with direct communication across the cluster
  – Instead library employed indirect communication
• Configured as exclusive-process, which allows only one process per GPU
  – Also prevents having multiple processes per host if they use pinned host memory
  – To employ multiple processes, CUDA multi-process service (MPS) should be enabled
• Jobs are submitted with resource sets
  – Each GPU can have its own resource set
  – or share the resource set with the other GPUs on the host
Single Host Performance - 4 GPUs

- MPS has an overhead
- Separate resource sets for GPUs yield to a better performance
• Heat shows a slight increase with direct transfers on V100 workstation
  – Employs high-bandwidth NVLink 2.0 between GPUs