Overlapping Data Transfers with Computation on GPU with Tiles

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

Weiqun Zhang, Ann Almgren, John Shalf
Lawrence Berkeley National Laboratory, CA, USA
Motivation

• GPUs are attractive means for accelerating scientific and machine learning applications

• However, GPU programming requires substantial programming effort by the developer
  – Data transfers between host and device
  – Kernel code
  – Limited memory on GPU
  – PCI-e bandwidth
Prevalent Programming Models for GPUs

• **CUDA**
  – Programmer handles data transfers explicitly with functions
  – Programmer implements kernel code as a device functions

• **OpenACC**
  – OpenACC can handle data transfers itself, but it is necessary to handle them with directives for good performance
  – OpenACC generates kernel code with directives placed by programmer

• Neither of programming models provides a solution for limited memory on GPU and PCI-e bandwidth
Proposal: GPU Programming with Tiles

- Partition data into smaller tiles
- Transfer data between host and device
- Generate the kernel code
- Provide GPU execution for applications which use more memory than GPU has
- Overlap data transfers with computation

Burak Bastem, Koc University @ ICPP 2017
TiDA-acc

- Leverages TiDA, CUDA and OpenACC for uniform CPU and GPU execution
  - TiDA-acc extends the tile iterator of TiDA for GPUs
  - Data is partitioned to regions with TiDA
  - Regions are created and transferred with CUDA
  - Kernels are generated with OpenACC
TiDA: A Tiling Library

- Tiling is a well-known loop transformation
  - There are several compiler based solutions for tiling
  - Why not elevate it to the programming model?
- TiDA makes tiling part of the data structure declaration
  - Each array is extended with metadata
  - Metadata follows the array through the code
- TiDA comes with three simple abstractions

Burak Bastem, Koc University @ ICPP 2017
1. Regional Tiles

- Each structured grid is physically divided into regions
- Regions are mapped on to a different NUMA node or used as a transfer unit to GPU
- A programmer can set the region geometry using an env var or in the program
  - `export TiDA_REGIONS=x,y,z`
2. Logical Tiles

• Logically tiles regions
  – Different view of the data
  – No memory allocation is required
  – Only how the data traversed differs

• Designed for improving cache reuse within a NUMA node
3. Tile Iterator

- Tile iterator iterates over tiles (or regions) in an out-of-order fashion and manages parallelism.
  - Hides the details of execution policy
  - Applies the loop body or entire function on every tile
TiDA-acc Programming Interface
1. `TileArray` `ta = TileArray`(data_size, region_size, ghost_size);
2. `TileIter` `tIter = TileIter`(&`ta`, tile_size);
3. ...
4. `for(tIter.reset(GPU=true); tIter.isValid(); tIter.next())`
5. {
6.    `Tile` `tile = tIter.getTile`(`ta`);
7.    `TiDA::compute`(`tile`, [&](double* `data`, int `depth`,
8.                                             int `height`, int `width`, int `index`){
9.        `// compute`
10.       `data[index] = ...`
11.    }
12. });
13.}
**Simple User Interface**

```
TileArray ta = TileArray(data_size, region_size, ghost_size);
2. TileIter tIter = TileIter(&ta, tile_size);
3. ...
4. for(tIter.reset(GPU=true); tIter.isValid(); tIter.next())
5. {
6.   Tile tile = tIter.getTile(ta);
7.   TiDA::compute(tile, [&](double* data, int depth,
8.     int height, int width, int index){
9.     // compute
10.    data[index] = ...
11.  });
12. }
13.}
```

- At the beginning of the program, create a tile array by giving data, region and ghost size
Simple User Interface

1. `TileArray ta = TileArray(data_size, region_size, ghost_size);`
2. `TileIter tIter = TileIter(&ta, tile_size);`
3. ...
4. for(tIter.reset(GPU=true); tIter.isValid(); tIter.next())
5. {
6.   Tile tile = tIter.getTile(ta);
7.   TiDA::compute(tile, [&](double* data, int depth,
8.                     int height, int width, int index){
9.     // compute
10.    data[index] = ...
11.
12. });
13.}

• Then, create a tile iterator with the tile array and a logical
  tile size
1. `TileArray ta = TileArray(data_size, region_size, ghost_size);`
2. `TileIter tIter = TileIter(&ta, tile_size);`
3. ...

```
for(tIter.reset(GPU=true); tIter.isValid(); tIter.next()) {
    Tile tile = tIter.getTile(ta);
    TiDA::compute(tile, [&](double* data, int depth,
                            int height, int width, int index) {
        // compute
        data[index] = ...;
    });
}
```

- Iterate data using tile iterator in for loop
Simple User Interface

1. `TileArray ta = TileArray(data_size, region_size, ghost_size);`
2. `TileIter tIter = TileIter(&ta, tile_size);`
3.
4. `for(tIter.reset(GPU=true); tIter.isValid(); tIter.next())`
5. {
6.     `Tile tile = tIter.getTile(ta);`
7.     `TlDA::compute(tile, [&](double* data, int depth,`
8.                `int height, int width, int index){`
9.         `// compute`
10.        `data[index] = ...`
11.    `});`
12. `}

• Iterate data using tile iterator in for loop

enable GPU execution by resetting tile iterator with ‘true’ boolean
1. `TileArray` `ta = TileArray`(data_size, region_size, ghost_size);
2. `TileIter` `tIter = TileIter`(&`ta`, tile_size);
3. ...
4. for(tIter.reset(GPU=true); tIter.isValid(); tIter.next())
5. {
   
   `Tile` `tile = tIter.getTile`(`ta`);
   
   `TiDA::compute`(`tile`, [&](double* `data`, int `depth`,
                   int `height`, int `width`, int `index`){
   // compute
   `data[index] = ...`
   
   `});

   `}

   - Request a tile from iterator in the loop
     - library checks if region is already in GPU; if not, it copies
       region to GPU and returns its device pointer
Simple User Interface

1. `TileArray` `ta = TileArray`(data_size, region_size, ghost_size);
2. `TileIter` `tIter = TileIter`(&`ta`, tile_size);
3. ...
4. for(`tIter`.reset(GPU=true); `tIter`.isValid(); `tIter`.next())
5. {
6.   `Tile` `tile = tIter`.getTile`(`ta`);
7.   TiDA::compute`(`tile`, [&](double* `data`, int `depth`, int `height`, int `width`, int `index`){
     // compute
     `data`[`index`] = ...
   });
8. }  

- Call compute function to do computation on tile
Simple User Interface

1. `TileArray ta = TileArray(data_size, region_size, ghost_size);`
2. `TileIter tIter = TileIter(&ta, tile_size);`
3. ...
4. `for(tIter.reset(GPU=true); tIter.isValid(); tIter.next())`
5. {
6.   `Tile tile = tIter.getTile(ta);`
7.   `TiDA::compute(tile, [&](double* data, int depth, int height, int width, int index){`
8.     `// compute`
9.     `data[index] = ...`
10.    `});`
11. }
12. `Compute function takes two parameters:`
13. `  - tile to iterate over`
14. `  - a lambda function in which user implements desired computation`
Simple User Interface

1. ...
2. ...
3. ...
4. ...
5. ...
6. ...

7. TileAcc::compute(tile, [&](double* data, int depth, int height, int width, int index){
8. // compute
9. data[index] = ...
10. });

11. TiDA::compute(Dimension(0, 0, 3), Dimension(15, 15, 12),
12. tile, [&](double* data, int depth, int height, int width, int index){
13. // compute
14. data[index] = ...
15. });

• For a specific iteration, desired lower and higher bounds is given to compute function
To use more than one tile in computation, desired tiles are given to compute function

- Their data pointers are also given respectively in lambda function
Simple User Interface

1. for(int i=lo(0); i<=hi(0); i++){
2.   for(int j=lo(1); j<=hi(1); j++){
3.     for(int k=lo(2); k<=hi(2); k++){
4.       // compute
5.       data[i][j][k] = ...
6.     }
7.   }
8. }

1. for(tIter.reset(GPU=true); tIter.isValid(); tIter.next()) {
2.   Tile tile = tIter.getTile(ta);
3.   TiDA::compute(tile, [&](double* data, int depth, int height,
4.             int width, int index){
5.     // compute
6.     data[index] = ...
7.   });
8. }

Original

TiDA-acc
Implementation of TiDA-acc Library
Implementation of TiDA-acc

• TiDA-acc is responsible for
  1. Memory Allocation/Deallocation
  2. Kernel Code Generation
  3. Data Transfers
  4. Caching for unnecessary data transfers
  5. Overlapping transfer with computation thru Streams
  6. Ghost Cell Update in GPU
1. Memory Management

- Allocates memory on GPU for regions
- Assigns CPU regions to GPU regions
  - One-to-one if there is available memory
  - Regions share GPU memories, if there is no available memory

**Enough Memory Case Assignment**

- Host Memory
  - Region 0
  - Region 1
  - Region 2
  - Region 3

- GPU Memory
  - GPU Region 0
  - GPU Region 1
  - GPU Region 2
  - GPU Region 3

**Limited Memory Case Assignment**

- Host Memory
  - Region 0
  - Region 1
  - Region 2
  - Region 3

- GPU Memory
  - GPU Region 0
  - GPU Region 1
  - GPU Region 2
  - GPU Region 3
CUDA provides three types of memory for programmer in terms of host-device transfers:

- **Pageable Memory**
- **Pinned Memory**
- **Unified Memory**
• All implementations run with data of size $384^3$ for 100 iterations on a NVIDIA Tesla K40m.
## Types of Memory: Performance

- Pinned memory outperforms others
- Pinned memory with CUDA performs best

### 3D HEAT SOLVER EXECUTION TIMES (sec)

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenACC Pinned</td>
<td>1.61</td>
</tr>
<tr>
<td>OpenACC Pageable (Default)</td>
<td>2.35</td>
</tr>
<tr>
<td>OpenACC Managed</td>
<td>2.86</td>
</tr>
<tr>
<td>CUDA Pinned</td>
<td>1.34</td>
</tr>
<tr>
<td>CUDA Pageable (Default)</td>
<td>1.67</td>
</tr>
<tr>
<td>CUDA Unified Memory</td>
<td>2.12</td>
</tr>
</tbody>
</table>
2. Kernel Code Generation

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

• Using OpenACC for kernel code generation
  – Directives are easy to use,
  – Directives can be hidden in library
  – Supported and maintained by Nvidia
Type of Memory: Performance

HEAT SOLVER EXECUTION TIMES (sec)

- OpenACC Pinned: 1.61 sec
- OpenACC Pageable (Default): 2.35 sec
- OpenACC Managed: 2.86 sec
- OpenACC + CUDA Pinned: 1.62 sec
- OpenACC + CUDA Pageable: 1.94 sec
- OpenACC + CUDA Unified Memory: 2.42 sec
- CUDA Pinned: 1.34 sec
- CUDA Pageable (Default): 1.67 sec
- CUDA Unified Memory: 2.12 sec

- Use CUDA for data transfers and OpenACC for kernel code generation
3. Data Transfers

- It uses asynchronous CUDA functions for memory transfers.
  - Operations in the GPU are queued within streams, so no synchronization after host to device transfers.
  - After host to device transfers, streams of corresponding regions are synchronized.

![Diagram of data transfers]

No synchronization after host to device transfer

Synchronization after on device to host transfer

Burak Bastem, Koc University @ ICPP 2017
4. Caching

- TiDA-acc has a caching mechanism to avoid unnecessary data transfers
- Initially, it uses a cache list whose elements are set to -1 indicating all regions are in the host

![Limited Memory Case Assignment Diagram]

- Host Memory
  - Region 0
  - Region 1
  - Region 2
  - Region 3

- GPU Memory
  - GPU Region 0
  - GPU Region 1

Cache List

- 0: -1
- 1: -1

Burak Bastem, Koc University @ ICPP 2017
4. Caching

- Programmer requests “Region 3” in GPU enabled for loop

![Limited Memory Case Assignment Diagram]

![Cache List Diagram]
4. Caching

- TiDA-acc takes $3 \% 2 = 1$
4. Caching

- TiDA-acc takes $3 \% 2 = 1$

**Limited Memory Case Assignment**

- **Host Memory**
  - Region 0
  - Region 1
  - Region 2
  - Region 3

- **GPU Memory**
  - GPU Region 0
  - GPU Region 1

**Cache List**

- 0: -1
- 1: -1
4. Caching

- TiDA-acc takes $3 \% 2 = 1$

**Limited Memory Case Assignment**

- **Host Memory**
  - Region 0
  - Region 1
  - Region 2
  - Region 3

- **GPU Memory**
  - GPU Region 0
  - GPU Region 1

**Cache List**

- 0: -1
- 1: -1
4. Caching

• TiDA-acc takes $3 \mod 2 = 1$

**Limited Memory Case Assignment**

<table>
<thead>
<tr>
<th>Host Memory</th>
<th>GPU Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region 0</td>
<td>GPU Region 0</td>
</tr>
<tr>
<td>Region 1</td>
<td>GPU Region 1</td>
</tr>
<tr>
<td>Region 2</td>
<td></td>
</tr>
<tr>
<td>Region 3</td>
<td></td>
</tr>
</tbody>
</table>

**Cache List**

<table>
<thead>
<tr>
<th>Index of cache list</th>
<th>Requested region ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>
4. Caching

- Checks if index 1 in cache list has the ID of requested tile
4. Caching

- Checks if index 1 in cache list has the ID of requested tile
4. Caching

- Copies “Region 3” to “GPU Region 1”
- Places region ID to corresponding index in cache list
5. Streams

• Streams are sequences of device operations.
  – Different device operations assigned to different streams overlap each other.

• TiDA-acc creates and assigns a stream for each separate GPU memory.
  – Device operations for each region are assigned to a specific stream.
  – Operations for different regions overlap with each other.
5. Streams

- GPU has a limited memory for the applications
  - Regions share GPU memories and streams
- While a kernel executes region 2 at stream 2, data transfers take place at stream 1
  - region 1 is transferred from device to host
  - region 3 is transferred from host to device

execution profile of an application that cannot fit into GPU

<table>
<thead>
<tr>
<th>stream 1</th>
<th>stream 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2H: R1</td>
<td>Execution: R2</td>
</tr>
<tr>
<td>H2D: R3</td>
<td>D2H: R2</td>
</tr>
<tr>
<td>Execution: R3</td>
<td>H2D: R4</td>
</tr>
<tr>
<td>D2H: R3</td>
<td>Execution: R4</td>
</tr>
<tr>
<td>H2D: R5</td>
<td>D2H: R4</td>
</tr>
<tr>
<td>Execution: R5</td>
<td>H2D: R6</td>
</tr>
</tbody>
</table>

Burak Bastem, Koc University @ ICPP 2017
6. Ghost Cell Update in GPU

- Source and ghost cell indices of regions are calculated on CPU while update of ghost cells is done on GPU.
  - Calculation of indices involves branching in which GPUs are unutilized. By calculating indices on CPU, both CPU and GPU are utilized.

While blue ghost cells of region 2 is updated with a kernel on GPU, CPU calculates indices of red ghost cells of region 2 and their correspondences in region 3.
Performance of TiDA-acc
Performance

• Performance is evaluated with two kernels which are data transfer intensive and compute intensive.
• Data used in both kernels has size of $512^3$.
• The setup has Intel Xeon E5-2695 v2 processors as the host and a NVIDIA Tesla K40m as the device.
• PGI compiler version 17.1.0 for OpenACC and NVCC compiler version 7.5.17 for CUDA is used.

<table>
<thead>
<tr>
<th>Tesla K40m CUDA Bandwidth Test (GB/s)</th>
<th>Pageable Memory Transfers</th>
<th>Pinned Memory Transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host to Device</td>
<td>3.19</td>
<td>6.06</td>
</tr>
<tr>
<td>Device to Host</td>
<td>2.62</td>
<td>6.53</td>
</tr>
<tr>
<td>Device to Device</td>
<td>183.32</td>
<td>183.56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intel(R) Xeon(R) E5-2695 v2 Stream Bandwidth Test (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 Threads</td>
</tr>
</tbody>
</table>
Test Kernels

• Data Transfer Intensive

```c
data_next[cell] = data[cell] - alpha * (data[front_cell] +
data[back_cell] + data[top_cell] +
data[bottom_cell] + data[left_cell] +
data[right_cell]);
```

• Compute Intensive

```c
double s, c;
for(double i=0; i<kernel_iteration; i++){
   s = sin(data[index]);
   c = cos(data[index]);
   data[index] = data[index] + sqrt(s*s+c*c);
}
```
Performance: Data Transfer Intensive Kernel

Data Transfer Intensive Kernel Speedups over CUDA Pageable

- In lower iterations, TiDA-acc achieves better performance because of overlapping.
- As number of iterations increase, computation dominates and memory transfers become negligible, so TiDA-acc shows similar performance with CUDA.

Burak Bastem, Koc University @ ICPP 2017
Performance: Compute Intensive Kernel

- OpenACC math function kernels run faster
- Shows similar results with data transfer intensive kernel
- Data transfers are completely overlapped with kernel executions, so it shows better performance especially in lower iterations

Burak Bastem, Koc University @ ICPP 2017
Limited Memory Case

- TiDA-acc executes the application even there is not enough memory on GPU without performance loss because of overlapping
- CUDA or OpenACC does not provide such a mechanism
## Related Work

<table>
<thead>
<tr>
<th>Related Work</th>
<th>TiDA-acc</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mint [2], OpenMPC [3], Daino [14]</strong></td>
<td>requires directives and uses in-house compiler to generate CUDA code</td>
</tr>
<tr>
<td>RAJA [5], Kokkos [6], Thrust [8], SYCL [9], C++ AMP [10]</td>
<td>uses C++ lambda functions to hide kernel generation, but does not completely hide memory management</td>
</tr>
<tr>
<td>PACXX [11]</td>
<td>hides kernel generation and data transfers by extending specific C++ data structures</td>
</tr>
<tr>
<td>CuMAS [12]</td>
<td>focuses on scheduling multiple applications on GPU and overlaps their data transfers and kernel executions,</td>
</tr>
<tr>
<td>dCUDA [13]</td>
<td>overlaps computation with inter-node communication on a multi-GPU environment and relies on the programmer to implement the CUDA kernels</td>
</tr>
</tbody>
</table>
Conclusion

• TiDA-acc
  – is easy to use
  – provides GPU acceleration without sacrificing performance
  – Executes applications that cannot fit into device
  – provides a mechanism for PCIe bandwidth problem

• Right now, implementation of a real world application, combustion simulation (SMC), is in progress.

• As future work, TiDA-acc will be extended to multi-GPU.
Acknowledgements

• Authors from Koç University are supported by the Turkish Science and Technology Research Centre Grant No: 215E185.
• Dr. Unat is supported by the Marie Sklodowska Curie Reintegration Grant 655965 by the European Commission.
• Authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231.

Thank you!


TiDA: Region
TiDA: Tile

Tile (2 x 2 x 2)
TiDA: Tile Iterator