Realizing Out-of-Core Stencil Computations using Multi-Tier Memory Hierarchy on GPGPU Clusters

~ Towards Extremely Big & Fast Simulations ~

Toshio Endo
GSIC, Tokyo Institute of Technology (東京工業大学)
Stencil Computations

Important kernels for various simulations (CFD, material...)

ASUCA weather simulator

Phase-Field computation (2011 Gordon Bell)

Air flow simulation

Stencil computations are “memory intensive” →

On GPU clusters, Highly successful *in speed*
But not *in scale*
Issues on Typical Stencil Implementations on GPUs

In typical stencil implementations on GPUs, array sizes are configured as < (aggregated) GPU memory ➔ Prohibits extremely Big & Fast simulation

Using multiple GPUs is a solution
• But we are still limited by “GPU memory capacity × #GPUs”
• Larger capacity of lower memory hierarchy is not utilized
Stencil Code Example on GPU

- **Copy domain**: Host → Device
- **Temporal Loop**
- **MPI comm. of boundary**
- **Compute Grid points**
- **Copy domain**: Device → Host

Double buffering

- Device Memory capacity

Speeds of 7-point stencil on K40

- Fast, but not Big

Problem Size (GiB)

Speed (GFlops)

- Normal

Double buffering < 12GB

Faster

Bigger

Copy domain Host → Device

Faster
Goals of This Work

When we have existing apps, we want to realize followings

Large Scale
High Performance
High Productivity

Using memory swapping of the HHRT library
Locality improvement with Temporal Blocking

Co-design approach that spans Algorithm layer, Runtime layer, Architecture layer
Contents

• Step 1: using HHRT library
  – Expands available memory capacity by data swapping
  – Supporting multi-tier memory hierarchy
• Step 2: using Temporal blocking (briefly)
  – Optimizations of stencils for locality improvement
The HHRT Runtime Library for GPU Memory Swapping

• HHRT supports applications written in CUDA and MPI
  – HHRT is as a wrapper library of CUDA/MPI
  – Original CUDA and MPI are not modified
  – Not only for stencil applications

[Diagram showing the comparison between w/o HHRT and With HHRT]

github.com/toshioendo/hhrt
T. Endo and Guanghao Jin. Software technologies coping with memory hierarchy of GPGPU clusters for stencil computations. IEEE CLUSTER2014
# Functions of HHRT

1. HHRT supports **overprovisioning** of MPI processes on each GPU
   - Each GPU is shared by $m$ MPI processes

2. HHRT executes implicitly **memory swapping** between device memory and host memory
   - "process-wise" swapping
   - OS-like "page-wise" swapping is currently hard, without modifying original CUDA device/runtime
Execution model of HHRT

w/o HHRT (typically)

```markdown
- Node
- Device memory
- Process's data
- Lower memory
```

With HHRT

```markdown
- Node
- Device memory
- Process's data
- Lower memory
```

m MPI processes share a single GPU
In this case, m=6
We suppose $s < \text{Device-memory-capacity} < m \times s$:

- $s$: Size of data that each process allocates on device memory
- $m$: The number of processes sharing a GPU

→ We can support larger data size than device memory in total

- We cannot keep all of $m$ processes running

→ HHRT makes some processes “sleep” forcibly and implicitly

- Blocking MPI calls are “yield” points
State Transition of Each Process

A process is **blocked** due to MPI operation (MPI_Recv, MPI_Wait...)

All data on upper (cudaMalloc’ed) are evacuated to lower memory

Swapping finished

Swapping finished

All data are restored to device

There is **enough space** on upper memory

MPI operation is now **unblocked** (cf. message arrived)
Executions on HHRT

6 processes are time-sharing a GPU
Two-tier (Device/Host) is used

MPI is called
Proc is restarted
MPI is finished
Swapping out
Swapping in

Time
 Processes
 0 1 2 3 4 5
 40 45 50 55 60 (sec)

RUNNING  Runnable  BLOCKED  D2H  H2F  F2H  H2D

Swapping out
Swapping in
What HHRT does NOT

• It does NOT automate data transfer (cudaMemcpy) → It is not OpenACC
  – Supports (traditional) CUDA programming
  – Instead, it implicitly swaps out data on device memory to lower hierarchy

• It does NOT swap in page-wise style like OS → It is NOT NVIDIA Unified Memory
  – In stencil, page-wise swapping tends to be slow
  – Instead, it adopts process-wise swapping

• It does NOT extend memory for a single process
  – Instead, our focus is to extend the aggregate capacity for multiple processes
Swapping Data in Multi-tier Memory Hierarchy

[What data are swapped]

Following data allocated by user processes
- On device memory (cudaMalloc)
- On host memory (malloc)

For this purpose, cudaMalloc, malloc... are wrapped by HHRT

Exceptionally, buffers just used for MPI communications must be remained on upper

[Where data are swapped out]

- Host memory first
- And then Flash SSD

For swapping, HHRT internally uses
- cudaMemcpy() for device ⇔ host
- read(), write() for host ⇔ Flash SSD
### Evaluation Environment

<table>
<thead>
<tr>
<th></th>
<th>TSUBAME2.5 (K20X GPU)</th>
<th>TSUBAME-KFC (K80 GPU)</th>
<th>PC server with m.2 SSD (K40 GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Device memory</strong></td>
<td>6GB • 250GB/s</td>
<td>12GB • 240GB/s</td>
<td>12GB • 288GB/s</td>
</tr>
<tr>
<td><strong>Host memory</strong></td>
<td>54GB • 8GB/s</td>
<td>64GB • 16GB/s</td>
<td>64GB • 16GB/s</td>
</tr>
<tr>
<td>(Speeds are via PCIe)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Flash SSD</strong></td>
<td>120GB • R 0.2GB/s</td>
<td>960GB • R 1GB/s (with two SSDs)</td>
<td>512GB • R 2GB/s</td>
</tr>
</tbody>
</table>

In our context, both of speed and capacity are insufficient (SSDs installed in 2010)
Result of Step 1: Exceeding Memory Capacity Wall

Certainly we exceed capacity wall for scale, however, the performance is seriously bad!
Issues in Step1: Too low GPU utilization

In the case of 96GB problem
• 32 processes on a GPU

Runs only for 40msec after sleeping >60secs
→ Too low GPU utilization
Why is GPU Utilization Too Low?

• Each process can suffer from heavy memory swapping costs every iteration
  – It incurs transfer of the entire process’es sub-domain between memory hierarchy

• This is done automatically, but too heavy to hide

• This is due to lack of locality of stencil computations
  – Array data are swapped out every iteration

• We need optimizations to improve locality as step 2!!
Step 2: Temporal Blocking (TB) for Locality Improvement

**Temporal blocking (in our context):**
Larger halo region, with width of k, is introduced per process
After a process receives halo with MPI, we do **k-step update at once** without MPI

* k is “temporal block size”

**Frequency of MPI comm (yielding points on HHRT) is reduced to 1/k**
Appropriate Temporal Block Sizes ($k$)

- If $k$ is too small, we suffer from swapping costs (if swap occurs)
- If $k$ is too large, we suffer from redundant computation costs for larger halo

<table>
<thead>
<tr>
<th>Problem Sizes</th>
<th>PC server with an m.2 SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>k=1</td>
</tr>
<tr>
<td>6(GiB)</td>
<td>149</td>
</tr>
<tr>
<td>8</td>
<td>140</td>
</tr>
<tr>
<td>12</td>
<td>8.72</td>
</tr>
<tr>
<td>16</td>
<td>9.39</td>
</tr>
<tr>
<td>24</td>
<td>9.37</td>
</tr>
<tr>
<td>32</td>
<td>9.79</td>
</tr>
<tr>
<td>48</td>
<td>8.12</td>
</tr>
<tr>
<td>64</td>
<td>3.23</td>
</tr>
<tr>
<td>96</td>
<td>2.68</td>
</tr>
<tr>
<td>128</td>
<td>2.67</td>
</tr>
<tr>
<td>192</td>
<td>2.55</td>
</tr>
</tbody>
</table>
Results of Step 2: Performance Improvement

- With high-speed with ~2GB/s Read, we obtain ~55% performance with 1.5x larger problem than host memory
  - We observe performance difference of SSDs
  - We still see significant slow down with > 100GB sizes
Current Limitations on Performance and Discussion

• Even with swapping facility, there is still memory pressure for:
  – MPI communication buffers
    • Both on user space and on MPI internally
  – CUDA’s internal device memory consumption
    • \(~75\text{MB (per proc)} \times 80\text{ proc} = 6\text{GB} \implies \sim50\% \text{ of GPU memory}!!\)

<table>
<thead>
<tr>
<th>Problem Sizes</th>
<th>Device memory</th>
<th>Host memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>6(GiB)</td>
<td>149</td>
<td>149</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8.72</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>9.39</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>9.37</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>9.79</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>8.12</td>
</tr>
<tr>
<td>48</td>
<td>48</td>
<td>3.25</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>2.68</td>
</tr>
<tr>
<td>96</td>
<td>96</td>
<td>2.67</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>2.55</td>
</tr>
<tr>
<td>192</td>
<td>192</td>
<td>2.55</td>
</tr>
</tbody>
</table>

Execution failure due to out-of-memory limits us. Why?
Weak Scalability on Multi GPU/Node

The TSUBAME-KFC Cluster
(1 K80 GPU + 2 SSDs) per node are used

Fairly good weak scalability,
But costs of SSDs are still heavy
Future Work

• More performance
  – We still suffer from memory pressure
    • Dozens of processes share MPI/CUDA
    • Scalable MPI/CUDA multiplexor will be the key

• More scale
  – Using burst buffers?

• More productivity
  – Integrating DSL (Exastencil, Physis..)
  – Integrating Polyhedral compilers
Summary

Out-of-core stencil computations on 3-tier memory hierarchy has been described

- **Architecture level:**
  - High performance (>GB/s) Flash SSDs

- **Middleware level:**
  - HHRT library for data swapping

- **App. Algorithm level:**
  - Temporal blocking for locality improvement

Co-design is the key