X10-specific Optimization of CPU-GPU Data transfer with Pinned Memory Management

Hideyuki Shamoto†, Tatsuhiro Chiba‡, Mikio Takeuchi‡

† Tokyo Institute of Technology
‡ IBM Research Tokyo
Programming for large-scale heterogeneous architecture

• Increase of the number of GPU-based heterogeneous architectures
e.g) Titan (18688 CPUs, 18688 GPUs),
   TSUBAME2.5 (2816 CPUs, 4224 GPUs)
   – High computing and memory performance of GPU
   – Hard to program on distributed system
   – Hard to program with GPU devices
     • X10 + CUDA
       • Large performance gap between X10+CUDA and CUDA
Performance gap between CUDA and X10

- **Operation**
  - CPU-GPU memcpy + kernel (add two rails into a rail)

- **Environment**
  - TSUBAME2.5

The performance of CPU-GPU memcpy of X10 is worse than that of CUDA
Performance of CPU-GPU memcpy

- Cost of CPU-GPU data transfer is larger than cost of running kernel on GPU in many applications
  - Sorting, Merging, etc.

- CPU-GPU memcpy mechanism of X10 degrades the memcpy performance compared to CUDA
  - A performance gap decrease the overall performance of X10CUDA applications
Approach & Contribution

• Approach
  – Reducing overheads of existing CPU-GPU memcpy of X10 with dynamic pinning memory
    • Not allocate pinned buffer preliminary, but pin memory each time when memcpy is called

• Contribution
  – Achieves runtime reduction of 27% for CPU-GPU memcpy compared to the current X10 implementation on TSUBAME2.5
Writing programs using GPU in X10

• How to write?
  – GPU is a remote place in X10
    • Naively do “async at (gpu) @CUDA”
  – Parallelizing operations with “async”

```plaintext
async at (gpu) @CUDA {
  finish for (block in 0n..239n) async {
    clocked finish for (thread in 0n..63n) clocked async {
      ...
    }
  }
}
```

240 × 64 threads work in parallel

Source: http://x10-lang.org/documentation/practical-x10-programming/x10-on-gpus.html
Pros and Cons of X10 using CUDA

• **Pros:**
  – Easier to program compared to CUDA
    • No need to handle pointer
    • No need to specify the #threads and #blocks
      – autoThreads(), autoBlocks()
    • etc.

• **Cons:**
  – Lower performance compared to CUDA
  – Lack of some features which we can use in CUDA
Source code level optimization

- Coalescing memory access
  - Neighbor threads access neighbor memory spaces
- Overlapping kernel and bidirectional memory copy
  - Using three CUDA streams
  - Current X10 implementation doesn’t support using over two streams

- Selecting optimal #threads and #blocks
  - autoBlocks()/autoThreads()
    - “Automatically choose enough blocks/threads to saturate the GPU”
  Source: [http://x10.sourceforge.net/x10doc/latest/](http://x10.sourceforge.net/x10doc/latest/)
- These optimizations are used in both X10 and CUDA
  → In order to reduce the performance gap, we do runtime level optimization
CPU-GPU Memcpy of X10

• Allocating two pinned buffers at the beginning

• Double buffering
  – Copy a part of data which will be transferred to GPU in the next step to one pinned buffer
  – Copy a part of data on the other pinned buffer to GPU
    • In these operations, the size of chunk is 1MB
    • Copy operations are managed by a queuing system
Analysis of the memcpy

Copy 1MB chunk from Host to Device
Accelerating CPU-GPU memcpy with dynamic pinning memory

• Instead of using buffers which are pinned at the beginning, we pin buffers each time when CPU-GPU memcpy is called
  – No need to allocate extra buffers
  – No need to copy data to pinned buffers

• Other modifications
  – Not splitting data into 1MB chunks and pinning data at once
  – Not iterating 1MB copy operations but copying data at once
    • Reduce the cost of managing queue
Experiment

• Evaluate the proposed memcpy mechanism
  – Iterating Rail.asyncCopy() 2 times
    • In order to easily check the overhead of memcpy in the profiling phase, we iterated asyncCopy

• TSUBAME2.5
  – CPU: Intel Xeon X5670 2.93 GHz (6 cores) x 2
  – GPU: NVIDIA Tesla K20X x 3
  – DRAM: 54GB
Default memcpy vs Optimized memcpy

Runtime reduction of 27%
Profiling overhead reduction

<table>
<thead>
<tr>
<th>Current X10 implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context 1 (CUDA)</td>
</tr>
<tr>
<td>Memcpy (HtoD)</td>
</tr>
<tr>
<td>Memcpy (DtoH)</td>
</tr>
<tr>
<td>Streams</td>
</tr>
<tr>
<td>Stream 13</td>
</tr>
<tr>
<td>Memcpy...</td>
</tr>
<tr>
<td>Memcpy...</td>
</tr>
<tr>
<td>Memcpy...</td>
</tr>
<tr>
<td>Memcpy...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimized implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context 1 (CUDA)</td>
</tr>
<tr>
<td>Memcpy (HtoD)</td>
</tr>
<tr>
<td>Compute</td>
</tr>
<tr>
<td>100.0% [1] Measurement</td>
</tr>
<tr>
<td>Streams</td>
</tr>
<tr>
<td>Stream 12</td>
</tr>
<tr>
<td>Stream 13</td>
</tr>
<tr>
<td>Memcpy HtoD [async]</td>
</tr>
<tr>
<td>Memcpy HtoD [async]</td>
</tr>
</tbody>
</table>

The ratio between memcpy and a overhead is reduced.
Performance Evaluation

25% longer runtime compared to CUDA
Conclusion

• Achieved runtime reduction of 27% for CPU-GPU memcpy compared to the original X10 implementation on TSUBAME2.5
  – Runtime of CPU-GPU data transfer of X10 is still approximately 25% larger than that of CUDA

• The overheads of memcpy haven’t been deleted completely
  – Applying double buffering technique to our method
Future Work

• Double buffering
  – Pinning memory
  – CPU-GPU memory copy

• Managing pinned memory spaces
  – Releasing pinned memory when the total pinned memory space grows
  – Not releasing pinned memory when the memory is used multiple times