Scaling Large Data Computations on Multi-GPU Accelerators

Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann

School of ECE, Purdue University
Outline

- Motivation
- Computation Splitting (COSP)
- Pipelining CPU-GPU copies
- Multi-GPU Code Generation
- Adaptive Runtime Tuning
- Evaluation
- Conclusion
Motivation

• GPGPUs – Accelerators of choice
• Many supercomputers use them
• Ideal for large parallel computations

However, large computations involve
• Large data sets → GPU device memories are smaller, lack good virtualization support
• Data has to be copied in and out of the GPU card → Memory copy overhead

Programmability and performance tuning have remained to be a major issues in GPGPU computing
Goals

- Automatically handle out-of-card computations
- Design a pipelining system that overlaps the kernel computations with data communications
- Implement above techniques on top of OpenMPC (OpenMP to CUDA translator)
- Automatically port OpenMP codes to multi-GPUs attached to a node
- Provide an online-tuning mechanism to choose the performance-optimal pipeline size
GPU Device Memory Requirement

What factors impact the GPU device memory requirement and how?

• *Shared* data: a single storage is required
• *Private* data: storage is required *per thread* → GPU grid size makes an impact

_Some optimizations increase the memory requirement_

• Prefetching (Early copy-in and late copy-out)
• Pipelining
Computation Splitting (COSP)

- **Split a large problem into smaller sub-problems** → memory requirement reduced
- Are there side-effects?
  - YES, splitting is not always perfect
    - Segregate data-types
- Data required by **every** sub-problem: **MemFused**
- Data required **only** by a sub-problem: **MemSplittable**

B, C are **MemSplittable**, A is **MemFused**
COSP - Code Example – Scalar Product

```c
#pragma omp parallel for shared(D,E,F) private(vec, pos, sum)
for (vec = 0; vec < NUM_VECTORS; vec++) {
    sum = 0;
    for(pos = 0; pos < NUM_ELEMENTS; pos++) {
        sum += D[NUM_ELEMENTS*vec + pos]* E[NUM_ELEMENTS*vec + pos];
    }
    F[vec] = sum;
}

for (split = 0; split < NUM_VECTORS/SplitSize; split++) {
    #pragma omp parallel for shared(D, E, F) private(vec, pos, sum) shared(split, SplitSize)
    for (vec = 0; vec < SplitSize; vec++) {
        sum = 0;
        for(pos = 0; pos < NUM_ELEMENTS; pos++) {
            sum += D[(NUM_ELEMENTS * (vec + split*SplitSize)))+ pos] * E[(NUM_ELEMENTS * (vec + split*SplitSize)))+ pos];
        }
        F[(vec + split*SplitSize)] = sum;
    }
```
Pipelining

- Computation Splitting creates pipelining opportunities
- Resources to pipeline:
  - Copy-in channel
  - GPU cores
  - Copy-out channel
- Maximum speedup – 3*

* Considering different copy-in and copy-out channels (engines)
Pipelining – Achievable Speedup

• Computation time: \( t_{\text{compute}} \)

• Time required to copy *MemFused* data: \( t_{\text{memFused}} \) (in and out of the GPU)

• For *MemSplittable* data,
  - \( t_{ci} \) (data copy-in time)
  - \( t_{co} \) (data copy-out time)

• Achievable Speedup

\[
\frac{t_{\text{memFused}} + t_{ci} + t_{co} + t_{\text{compute}}}{t_{\text{memFused}} + \max(t_{ci}, t_{co}, t_{\text{compute}})}
\]
Pipelining - Implementation

- `cudaStream` : A CUDA abstraction of instruction queues
- `cudaStreams` act independently
- Memory copy requests across `cudaStreams` get serialized
- Kernel executions across `cudaStreams` overlap

For pipelining:
- Use 2 `cudaStreams`
- Create 2 device buffers per `MemPrivate` Data
- Create single device buffer per `MemShared` Data
- Place memory copy operations for the `MemShared` Data out of the **Split Loop**
- Schedule alternate `Splits` on each `cudaStream`
Extending to Two GPUs

cudaStream 0 – Device 0
Copy-in
Split 1 - Device 0
Kernel – Split 1 - Device 0
Copy-out - Split 1 – Device 0

cudaStream 1 – Device 0
Copy-in
Split 2 - Device 0
Kernel – Split 2 - Device 0
Copy-out - Split 2 – Device 0

cudaStream 2 – Device 1
Copy-in
Split 3 - Device 1
Kernel – Split 3 - Device 1
Copy-out - Split 3 – Device 1

cudaStream 3 – Device 1
Copy-in
Split 4 - Device 1
Kernel – Split 4 - Device 1
Copy-in – Split 4 - Device 1

Split Loop

Time
Compiler Structure

* Darker boxes are inherited from OpenMPC
Tuning Objective

Pipelining benefits  Equality of pipeline stage sizes

However, static scheme to determine best stage size is hard since:

• GPU systems: Intricate architecture (PCI version, #cores, GPU memory BW)
• Kernel overlaps → Difficult to model

We propose an adaptive runtime tuning system to choose the optimal SplitSize
For Compute Intensive programs, optimal SplitSize $\leq$ MaxThreads

For Memory Copy-Intensive programs, optimal SplitSize $\geq$ MaxThreads

MaxThreads = No. of threads that can simultaneously coexist on a GPU
Heuristic Algorithm

\[ t_{ci} = \text{Copy-in Time} \quad t_{co} = \text{Copy-out Time} \quad t_k = \text{Kernel Time} \]

(Number in superscript is the \textit{cudaStream} number)

\begin{itemize}
  \item For Type 1 kernels: Larger \textit{SplitSizes} work better due to the higher bandwidth usage.
  \item For Type 2 kernels: \textit{SplitSizes} smaller than MaxThreads are the candidates
  \item For Type 3 kernels: Candidate set is much larger. Type 3 is uncommon.
\end{itemize}

\begin{itemize}
  \item Highly Memory Copy-Intensive
  \item Highly Compute-Intensive
\end{itemize}

On a single Split,

- Determine the Type of the kernel
- For Type 1 kernels: Larger \textit{SplitSizes} work better due to the higher bandwidth usage.
- For Type 2 kernels: \textit{SplitSizes} smaller than MaxThreads are the candidates
- Generate a set of candidate \textit{SplitSizes}, run each to find the best
- For Type 3 kernels: Candidate set is much larger. Type 3 is uncommon.

\[ t_{ci} > t_k + t_{co} \quad \text{OR} \quad t_{co} > t_k + t_{ci} \]

\[ t_k > t_{ci} + t_{co} \]

\[ t_{ci} > t_{k} + t_{co} \quad \text{OR} \quad t_{co} > t_{k} + t_{ci} \]

Tuning requires extra runs, but only on a single \textit{Split} \rightarrow \textit{Tuning overhead is negligible}
Evaluation

Setup
• GPU - Tesla M2090 GPUs (4), 6GB memory, x16 PCIe link
• CPU – AMD Opteron Processor 6282, 16 cores, 2.6 GHz, 64 GB RAM

Benchmarks
• Kernels – Black Scholes, Monte Carlo, DCT, Filterbank, Vector Add, Scalar Product, FFT
• Applications – CFD, SRAD
<table>
<thead>
<tr>
<th>Bench mark</th>
<th>DataSize (Iteration Space)</th>
<th>CUDA Time (s)</th>
<th>OpenMPC Base Time (s)</th>
<th>%Copy-in Time(s)</th>
<th>%Copy-out Time(s)</th>
<th>%Kernel Time(s)</th>
<th>OpenMPC Pipelined Time (s)</th>
<th>Speedup Ideal</th>
<th>Speedup Achieved</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalar Product</strong></td>
<td>1024 x 1024</td>
<td>0.633</td>
<td>0.88807</td>
<td>52.19789</td>
<td>0.22676</td>
<td>47.57535</td>
<td>0.46297</td>
<td>1.91579</td>
<td>1.91822</td>
</tr>
<tr>
<td></td>
<td>1024 x1024 x2</td>
<td>1.267</td>
<td>1.7723</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.73067</td>
</tr>
<tr>
<td></td>
<td>1024 x1024 x4</td>
<td>-----</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monte Carlo</strong></td>
<td>1024 x32</td>
<td>0.00537</td>
<td>0.00454</td>
<td>21.55109</td>
<td>13.71958</td>
<td>64.72933</td>
<td>0.0037</td>
<td>1.54489</td>
<td>1.22733</td>
</tr>
<tr>
<td></td>
<td>1024 x1024 x32</td>
<td>***</td>
<td>1.79902</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.44268</td>
</tr>
<tr>
<td></td>
<td>1024 x1024 x64</td>
<td>***</td>
<td>3.5924</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.42859</td>
</tr>
<tr>
<td></td>
<td>1024 x1024 x128</td>
<td>-----</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Black Scholes</strong></td>
<td>1024 x16</td>
<td>0.00105</td>
<td>0.00158</td>
<td>46.64777</td>
<td>41.9736</td>
<td>11.37863</td>
<td>0.00164</td>
<td>2.14373</td>
<td>0.96344</td>
</tr>
<tr>
<td></td>
<td>1024 x1024 x128</td>
<td>1.8598</td>
<td>1.22591</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.39786</td>
</tr>
<tr>
<td></td>
<td>1024 x1024 x256</td>
<td>-----</td>
<td>2.457</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.41219</td>
</tr>
<tr>
<td></td>
<td>1024 x1024 x384</td>
<td>-----</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`***` represent failure of the code due to larger-than-allowed grid sizes used. `-----` represent code failure due to out-of-memory data size errors.
Tuning Performance

Naïve scheme – Use 1024 Splits (Heuristically found to be effective)

A static scheme to select SplitSize can not be efficient
Comprehensive Results

Compute-Intensive

- Filterbank
- DCT
- Monte Carlo
- FFT

Memory Copy-Intensive

- Scalar Product
- BlackScholes
- Vector Add
- SRAD
- CFD

- Compute-intensive benchmarks show better scalability on multi-GPU systems.
- PCIe link forms a bottleneck on memory-copy intensive programs.
Related Work

Out-of-card computations
• Single device image for multi-GPUs

Pipelining/Memory related
• Prefetch
• Redundant memory transfer removal
• Asynchronous computations

Execution Models
• StreamIt based approaches
Conclusions

We presented

• An automatic computation splitting mechanism, COSP, that handles out-of-card computations
• A mechanism to effectively pipeline the slow CPU-GPU data copy channels with GPU computation
• An automatic adaptive runtime tuning system to select optimal pipeline stage size
• A porting scheme to run OpenMP applications on multi-GPU systems
Future Work

• Better strategy to deal with irregular applications

• Smart virtualization of the GPU address space – exploiting prediction to move data back and forth between CPU and GPU memories
Thank You!