Dynamic Fine-Grain Scheduling of Pipeline Parallelism

Presented by: Ram Manohar Oruganti and Michael TeWinkle
Overview

• Introduction
• Motivation
• Scheduling Approaches
• GRAMPS scheduling method
• Evaluation
• Conclusion
Introduction

The presentation talks about:
• General Runtime Architecture for Multicore Parallel Systems (GRAMPS) programming model
• How a GRAMPS scheduler dynamically schedules pipelined computations effectively
• Its advantages over the existing scheduling approaches
Motivation

• Static scheduling techniques lead to a high load imbalance when the algorithm is irregular/unpredictable

• Task stealing can’t guarantee memory footprint bounds and can’t handle complex control flows efficiently

• Breadth-First techniques have high memory footprint and bandwidth requirements
Previous Scheduling Techniques

Static:

• Application is divided into a group of stages
• Analysis of dependencies leads to better locality of data and mapping of tasks
• Offline scheduling eliminates overheads at runtime
• Works well when all stages nodes have uniform work
Task-Stealing:

• Treats tasks as independent work units that can run concurrently
• Imposes low overheads on task creation and execution
• Data dependencies and priorities aren’t taken into consideration while scheduling
• Poor data locality
• The memory footprint bounds can’t be guaranteed and scale up easily with problem size
• No concrete guidelines on the implementation
Breadth-First:

• Application is divided into a group of data parallel stages
• Each stage runs a number of instances in parallel
• Only one stage can execute at a time
• Needs barrier synchronization for each stage
• Large execution contexts leads to spillage of intermediate outputs to main memory
• Poor performance when there are a lot of stages and limited parallelism per stage
Fig 1. (left) Stages of a fixed pipeline GPU

Fig 2. (right) Unified pipeline GPU architecture
Overview of GRAMPS

• A programming model for futuristic GPU Architectures
• Deals with complex rendering applications, like Ray Tracing, more efficiently
• Aims at optimizing the implementations of complex algorithms, offer a large application scope and effective abstract the underlying architecture to provide portability
GRAMPS Design

• An application is defined as a set of computational stages
• The stages execute in parallel and communicate asynchronously via queues
• Each stage is implemented on a Stream Multi-Processor (SMP)
• Assumes the number of stages in the computation to be much less than the number of processors in the system
Overview of the Constituents of a GRAMPS Design

• Execution Graph: A custom designed pipeline for the application specified by the program
• Stage: A part of the pipeline that performs a computation. It could be a shader, a thread or a fixed function
• Queues: Used between stages for communication. The minimum amount of data that can be communicated is called a work packet. Uses a reserve-commit protocol.
• Buffers: Bound to stages to support communication
Advantages of GRAMPS over other scheduling techniques

• As all the stages run concurrently, limited parallelism per stage wouldn’t impact the performance
• Concrete guidelines in terms of implementation of queues and use of techniques like backpressure to ensure strict memory footprint bounds
• Takes producer-consumer relationships into consideration while scheduling to exploit locality
• Exploits both data parallelism and functional parallelism to improve performance
Other Scheduling Approaches Implementation

- **Task-Stealing**: designed with a single LIFO task queue, no per stage queues, and the data queues are unbounded. When a task is stolen it is taken from the tail of the queue.

- **Breadth-First**: designed to execute one stage at a time, once all producers are done the stage can run. Achieves load balancing by using work-stealing dequeue that doesn’t experience overflow.

- **Static**: designed so that a graph is computed that minimizes the communication to computation ratio while at the same time balancing the computation within each partition. No load balancing is performed by the Static scheduler, but once a worker runs out of work they wait until they receive work because each worker has a producer-consumer queue.
Evaluation

• System setup:
  – 2-socket system with hexa-core 2.93 GHz Intel Xeon X5670 processors.
    • 2-way Simultaneous Multi-Threading
    • 12 total cores with 24 total threads
  – Memory
    • 256 KB L2 cache per core
    • 12 MB L3 cache per processor
    • 48 GB DDR3 with peak bandwidth of 21 GB/s
  – Communication through a 6.4 GB/s QPI interconnect.
Evaluation

<table>
<thead>
<tr>
<th>Workload</th>
<th>Origin</th>
<th>Graph Complexity</th>
<th>Shader Work</th>
<th>Pipeline Parallel</th>
<th>Regularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>raytracer</td>
<td>ORAMPS</td>
<td>Medium</td>
<td>99%</td>
<td>Yes</td>
<td>Irregular</td>
</tr>
<tr>
<td>hist-r/c</td>
<td>MapReduce</td>
<td>Small</td>
<td>97%</td>
<td>No</td>
<td>Irregular</td>
</tr>
<tr>
<td>fr-r/c</td>
<td>MapReduce</td>
<td>Small</td>
<td>99%</td>
<td>No</td>
<td>Regular</td>
</tr>
<tr>
<td>pca</td>
<td>MapReduce</td>
<td>Small</td>
<td>99%</td>
<td>No</td>
<td>Regular</td>
</tr>
<tr>
<td>mergesort</td>
<td>Cilk</td>
<td>Medium</td>
<td>99%</td>
<td>Yes</td>
<td>Irregular</td>
</tr>
<tr>
<td>fm</td>
<td>StreamIt</td>
<td>Large</td>
<td>43%</td>
<td>Yes</td>
<td>Regular</td>
</tr>
<tr>
<td>tde</td>
<td>StreamIt</td>
<td>Huge</td>
<td>8%</td>
<td>Yes</td>
<td>Regular</td>
</tr>
<tr>
<td>fft2</td>
<td>StreamIt</td>
<td>Medium</td>
<td>70%</td>
<td>Yes</td>
<td>Regular</td>
</tr>
<tr>
<td>serpent</td>
<td>StreamIt</td>
<td>Medium</td>
<td>86%</td>
<td>Yes</td>
<td>Regular</td>
</tr>
<tr>
<td>srad</td>
<td>CUDA</td>
<td>Small</td>
<td>99%</td>
<td>No</td>
<td>Regular</td>
</tr>
<tr>
<td>rg</td>
<td>CUDA</td>
<td>Small</td>
<td>99%</td>
<td>No</td>
<td>Regular</td>
</tr>
</tbody>
</table>
Evaluation
## Evaluation

<table>
<thead>
<tr>
<th>App</th>
<th>GRAMPS</th>
<th>Task-Stealing</th>
<th>Breadth-First</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Max</td>
<td>Avg</td>
<td>Max</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Avg</td>
<td>Max</td>
<td>Avg</td>
</tr>
<tr>
<td>ray-0</td>
<td>215.9 KB</td>
<td>287.8 KB</td>
<td>191.7 KB</td>
<td>276.5 KB</td>
</tr>
<tr>
<td>ray-1</td>
<td>361.1 KB</td>
<td>447.4 KB</td>
<td>327.5 KB</td>
<td>424.6 KB</td>
</tr>
<tr>
<td>lr-r</td>
<td>22.9 KB</td>
<td>40.7 KB</td>
<td>22.9 KB</td>
<td>40.9 KB</td>
</tr>
<tr>
<td>lr-c</td>
<td>8.1 KB</td>
<td>8.8 KB</td>
<td>7.9 KB</td>
<td>8.2 KB</td>
</tr>
<tr>
<td>hist-r</td>
<td>4921.6 KB</td>
<td>9658.1 KB</td>
<td>4925.0 KB</td>
<td>9654.1 KB</td>
</tr>
<tr>
<td>hist-c</td>
<td>1860.3 KB</td>
<td>3096.8 KB</td>
<td>1864.5 KB</td>
<td>3096.3 KB</td>
</tr>
<tr>
<td>pca</td>
<td>0.7 KB</td>
<td>1.3 KB</td>
<td>0.4 KB</td>
<td>0.4 KB</td>
</tr>
<tr>
<td>msort</td>
<td>2.0 KB</td>
<td>7.0 KB</td>
<td>1.4 KB</td>
<td>4.6 KB</td>
</tr>
<tr>
<td>fm</td>
<td>1672.3 KB</td>
<td>4276.8 KB</td>
<td>2245.2 KB</td>
<td>3691.1 KB</td>
</tr>
<tr>
<td>tde</td>
<td>3989.0 KB</td>
<td>6662.2 KB</td>
<td>18398.7 KB</td>
<td>36231.2 KB</td>
</tr>
<tr>
<td>fft2</td>
<td>261.3 KB</td>
<td>376.0 KB</td>
<td>149.8 KB</td>
<td>211.0 KB</td>
</tr>
<tr>
<td>serpent</td>
<td>79.1 KB</td>
<td>88.0 KB</td>
<td>68.7 KB</td>
<td>73.2 KB</td>
</tr>
<tr>
<td>srad</td>
<td>0.9 KB</td>
<td>1.6 KB</td>
<td>0.4 KB</td>
<td>0.5 KB</td>
</tr>
<tr>
<td>rg</td>
<td>0.7 KB</td>
<td>1.4 KB</td>
<td>0.4 KB</td>
<td>0.5 KB</td>
</tr>
</tbody>
</table>
GRAMPS Scheduler Performance

- The speedup grows close to linearly up to 12 threads.
- Provides the most effective scheduler because of the minimal time without work over all applications.
- Both the buffer manager and scheduler provide very low overheads.
Task-Stealing Scheduler Performance

- Performs the best when the application contains a simple graph.
- Fm, tde and fft2 contain complex graphs and therefore perform worse than in GRAMPS.
  - This causes long stall time for these applications because the simple LIFO has trouble reorganizing incoming packets.
Breadth-First Scheduler Performance

• The only applications that perform similar to the GRAMPS scheduler are the histogram, lr, pca, srad and rg applications.
  – They contain no pipeline parallelism.

• Footprints created are fairly large in all applications which in turn creates pressure on the memory system and buffer manager.
Static Scheduler Performance

• Performs the worst when the application is irregular.
  – Mergesort performs the worst because of its highly irregular packer rates.

• All application speedups experience drop offs after 12 threads because the SMT is used in some cores and causes the threads to run at different speeds.
  – Caused because of poor load balancing.

• Optimized locality causes the application execution time to be lower than GRAMPS but poor load balancing negates these results.
Buffer Management Results

• The best approach is the packet-stealing buffer manager.
  – Small overheads as well as good locality.
• Per-queue approach has lower overheads than dynamic memory but can still be much larger than packet-stealing.
  – Locality is also worse in this scheme.
• Dynamic memory scheme is the worst of the three because of the frequent calls to malloc/free bottleneck at the memory allocator.
Conclusion

- The GRAMPS scheduler out performed all the other schedulers throughout the variety of applications and showed that it was very scalable to an increasing number of threads.
- It was able to dynamically load balance efficiently when the programs were irregular.
- GRAMPS was able to preserve locality and bounded memory footprints even if the application contained a complex graph.
Questions?