Dynamic Fine-Grain Scheduling of Pipeline Parallelism

Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, Christos Kozyrakis
Stanford University

PACT-20, October 11th 2011
Executive Summary

- Pipeline-parallel applications are hard to schedule
  - Existing techniques either ignore pipeline parallelism, cannot handle its dependences, or suffer from load imbalance

- Contributions:
  - Design a runtime that dynamically schedules pipeline-parallel applications efficiently
  - Show it outperforms typical scheduling techniques from multicore, GPGPU and Streaming programming models
Outline

- Introduction
- GRAMPS Programming Model
- GRAMPS Runtime
- Evaluation
High-Level Programming Models

- High-level parallel programming models provide:
  - Simple, safe constructs to express parallelism
  - Automatic resource management and scheduling

- Many aspects; we focus on scheduling
  - Model, scheduler and architecture often intimately related

- In terms of scheduling, three main types of models:
  - Task-parallel models, typical in multicore (Cilk, X10)
  - Data-parallel models, typical in GPU (CUDA, OpenCL)
  - Streaming models, typical in streaming architectures (StreamIt, StreamC)
Some models (e.g. streaming) define applications as a graph of stages that communicate explicitly through queues.
- Each stage can be sequential or data-parallel.
- Arbitrary graphs allowed (multiple inputs/outputs, loops).

- Well suited to many algorithms.
- Producer-consumer communication is explicit → Easier to exploit to improve locality.
- Traditional scheduling techniques have issues dynamically scheduling pipeline-parallel applications.
Task-Parallel – Task-Stealing

- Model: Task-parallel with fork-join dependences or independent tasks (Cilk, X10, TBB, OpenMP, …)

- Task-Stealing Scheduler:
  - Worker threads enqueue/dequeue tasks from local queue
  - Steal from another queue if out of tasks

- Efficient load-balancing
- × Unable to handle dependences of pipeline-parallel programs
- Model: Sequence of data-parallel kernels (CUDA, OpenCL)
- Breadth-First Scheduler: Execute one stage at a time in breadth-first order (source to sink)

Very simple model
- Ignores pipeline parallelism → works poorly with sequential stages, worst-case memory footprint
Model: Graph of *stages* communicating through *streams*

Static Scheduler:
- Assume app and architecture are regular, known in advance
- Use sophisticated compile-time analysis and scheduling to minimize inter-core communication and memory footprint

- Very efficient if application and architecture are regular
- **Load imbalance** with irregular applications or non-predictable architectures (DVFS, multi-threading …)
## Summary of Scheduling Techniques

<table>
<thead>
<tr>
<th></th>
<th>Supports pipeline-parallel apps</th>
<th>Supports irregular apps/archs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task-Stealing</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Breadth-First</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Static</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>GRAMPS</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Outline

- Introduction
- GRAMPS Programming Model
- GRAMPS Runtime
- Evaluation
Programming model for dynamic scheduling of irregular pipeline-parallel workloads

- Brief overview here, details in [Sugerman 2010]
- Shader (data-parallel) and Thread (sequential) stages
- Stages send packets through fixed-size data queues
  - Queues can be ordered or unordered
  - Can enqueue full packets or push elements (coalesced by runtime)
GRAMPS: Threads vs Shaders

- Threads are stateful, instanced by the programmer
  - Arbitrary number of input and output queues
  - Blocks on empty input/full output queue
  - Can be preempted by the scheduler

- Shaders are stateless, automatically instanced
  - Single input queue, one or more outputs
  - Each instance processes an input packet
  - Does not block
Similar model to Streaming, but features ease dynamic scheduling of irregular applications:
- Packet granularity $\rightarrow$ reduce scheduling overheads
- Stages can produce variable output (e.g., push queues)
- Data parallel stages, queue ordering are explicit

Static requires applications to have a steady state; GRAMPS can schedule apps with no steady state

GRAMPS was evaluated with an idealized scheduler when proposed; we implement a real multicore runtime
Outline

- Introduction
- GRAMPS Programming Model
- GRAMPS Runtime
- Evaluation
GRAMPS Runtime Overview

- Runtime = Scheduler + Buffer Manager

- Scheduler: Decide what to run where
  - Dynamic, low-overhead, keeps bounded footprint
  - Based on task-stealing with multiple task queues/thread

- Buffer Manager: Provide dynamic allocation of packets
  - Generic memory allocators are too slow for communication-intensive applications
  - Low-overhead solution, based on packet-stealing
Scheduler organization

- As many worker pthreads as hardware threads
- Work is represented with tasks

- Shader stages are function calls (stateless, non-preemptive)
  - One task per runnable shader instance

- Thread stages are user-level threads (stateful, preemptive)
  - User-level threads enable fast context-switching (100 cycles)
  - One task per runnable thread
Scheduler: Task Queues

- Load-balancing with task stealing
  - Each thread has one LIFO task queue per stage
  - Stages sorted by breadth-first order (higher priority to consumers)
  - Dequeue from high-priority first, steal low-priority first
    - Higher priority tasks drain the pipeline, improve locality
    - Lower priority tasks produce more work (less stealing)
Scheduler: Data Queues

- Thread input queues maintained as linked lists
- Shader input queues implicitly maintained in task queues
  - Each shader task includes a pointer to its input packet
- Queue occupancy tracked for all queues
- **Backpressure**: When a queue fills up, disable dequeues and steals from queue producers
  - Producers remain stalled until packets are consumed, workers shift to other stages
  - Queues never exceed capacity → bounded footprint
- Queues are optionally ordered (see paper for details)
Example

Thread 1 \rightarrow Shader 2 \rightarrow Thread 3

Queue 1 occupancy
9/20

Queue 2 occupancy
1/10
Queue 2 full $\Rightarrow$ disable dequeues and steals from Stage 2
Packet-Stealing Buffer Manager

- Packets pre-allocated to a set of pools
  - Each pool has packets of a specific size

- Each worker thread maintains a LIFO queue per pool
  - Release used input packets to local queue
  - Allocate new output packets from local queue, if empty, steal
  - Due to bounded queue sizes, no need to dynamically allocate packets
  - LIFO policy results in high locality and reuse
Outline

- Introduction
- GRAMPS Programming Model
- GRAMPS Runtime
- Evaluation
Methodology

- Test system: 2-socket, 12-core, 24-thread Westmere
  - 32KB L1I+D, 256KB private L2, 12MB per-socket L3
  - 48GB 1333MHz DDR3 memory, 21GB/s peak BW

- Benchmarks from different programming models:
  - GRAMPS: raytracer
  - MapReduce: histogram, lr, pca
  - Cilk: mergesort
  - StreamIt: fm, tde, fft2, serpent
  - CUDA: srad, recursiveGaussian
Alternative Schedulers

- GRAMPS scheduler can be substituted with other implementations to compare scheduling approaches
- **Task-Stealing**: Single LIFO task queue per thread, no backpressure
- **Breadth-First**: One stage at a time, may do multiple passes due to loops, no backpressure
- **Static**: Application is profiled first, then partitioned using METIS, and scheduled using a min-latency schedule, using per-thread data queues
- All applications scale well
- Knee at 12 threads due to HW multithreading
- Sublinear scaling due to memory bandwidth (hist, CUDA)
Performance Comparison

GRAMPS  MapReduce  Cilk  StreamIt  CUDA

Gramps  Task-Scheduling  Static  Breadth-First  Gramps
Dynamic runtime overheads are small in GRAMPS

Task-Stealing performs worse on complex graphs (fm, tde, fft2)

Breadth-First does poorly when parallelism comes from pipelining

Static has no overheads and better locality, but higher stalled time due to load imbalance
- Task-Stealing fails to keep footprint bounded (tde)
- Breadth-First has worst-case footprints → much higher footprint, memory bandwidth requirements
Buffer Manager Performance

- **Dynamic**: Allocate packets using `malloc/free` (tcmalloc)
- **Per-Queue**: Use per-queue, shared packet buffers
Buffer Manager Performance

- Generic dynamic memory allocator causes up to 6x slowdown on buffer-intensive applications
- Per-queue allocator degrades locality, performance with lots of stages (tde)
- Packet-stealing has low overheads, maintains locality
Conclusions

- Traditional scheduling techniques have problems with pipeline-parallel applications
  - Task-Stealing: fails on complex graphs, ordered queues
  - Breadth-First: no pipeline overlap, terrible footprints
  - Static: load imbalance with any irregularity

- GRAMPS runtime performs dynamic fine-grain scheduling of pipeline-parallel applications efficiently
  - Low scheduler and buffer manager overheads
  - Good locality
THANK YOU FOR YOUR ATTENTION QUESTIONS?