GRAMPS: A Programming Model for Graphics Pipelines and Heterogeneous Parallelism

Jeremy Sugerman
March 5, 2009
EEC277
History

- GRAMPS grew from, among other things, our GPGPU and Cell processor work, especially ray tracing.
- We took a step back to pose the question of what we would like to see when “GPU” and “CPU” cores both became normal entities on a multi-core processor.

- GRAMPS 1.0 Collaborators: Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, Pat Hanrahan
- Published in TOG, January 2009.
Background

- Context: Commodity, heterogeneous, many-core
  - “Commodity”: CPUs and GPUs. Modern out of order CPUs, Niagara and Larrabee-like simple cores, GPU-like shader cores.
  - “Heterogeneous”: Above, plus fixed function
  - “Many-core”: Scale out is a central necessity

Problem: How the heck do people harness such complex systems?

Status Quo: C run-time, GPU pipeline, GPGPU, ...
Our Focus

- Bottom up
  - Emphasize simple/transparent building blocks that can be run well.
  - Eliminate the rote, encourage good practices
  - Expect an informed developer, not a casual one

- Design an environment for systems-savvy developers that lets them efficiently develop programs that efficiently map onto commodity, heterogeneous, many-core platforms.
This Talk

1. What is GRAMPS?
2. Case Study: Rendering
3. Lessons Learned
4. (Bonus: Current Thoughts, Efforts)
GRAMPS: Quick Introduction

- Applications are graphs of stages and queues
- Producer-consumer inter-stage parallelism
- Thread and data intra-stage parallelism
- GRAMPS (“the system”) handles scheduling, instancing, data-flow, synchronization
GRAMPS: Examples
Evolving a GPU Pipeline

- “Graphics Pipeline” becomes an app!
  - Policy (topology) in app, execution in GRAMPS/hw
- Analogous to fixed → programmable shading
  - Pipeline undergoing massive shake up
  - Diversity of new parameters and use cases

- Not (unthinkably) radical even just for ‘graphics’
  - More flexible, not as portable
  - No domain specific knowledge
Evolving Streaming (1)

- Sounds like streaming:
  Execution graphs, kernels, data-parallelism

- Streaming: “squeeze out every FLOP”
  - Goals: bulk transfer, arithmetic intensity
  - Intensive static analysis, custom chips (mostly)
  - Bounded space, data access, execution time
Evolving Streaming (2)

- GRAMPS: “interesting apps are irregular”
  - Goals: Dynamic, data-dependent code
  - Aggregate work at run-time
  - Heterogeneous commodity platforms

- Streaming techniques fit naturally when applicable
  - Predictable subgraphs can be statically transformed and schedule.
Digression: Parallelism
Parallelism How-To

- Break work into separable pieces (dynamically or statically)
  - Optimize each piece (intra-)
  - Optimize the interaction between pieces (inter-)
- Ex: Threaded web server, shader, GPU pipeline
- Terminology: I use “kernel” to mean any kind of independent piece / thread / program.
- Terminology: I think of parallel programs as graphs of their kernels / kernel instances.
Intra-Kernel Organization, Parallelism

- Theoretically it is a continuum.
- In practice there are sweet spots.
  - Goal: span the space with a minimal basis

- Thread/Task (divide) and Data (conquer)
- Two?! What about the zero-one-infinity rule?
  - Applies to type compatible entities / concepts
  - Reminder: trying to span a complex space
Inter-kernel Connectivity

- Input dependencies / barriers
  - Often simplified to a DAG, built on the fly
  - Input data / communication only at instance creation
  - Instances are ephemeral, data is long-lived

- Producer-consumer / pipelines
  - Topology often effective static with dynamic instancing
  - Input data / communication happens ongoing
  - Instances may be long lived and stateful
  - Data is ephemeral and prohibitive to spill (bandwidth or raw size)
Here endeth the digression
GRAMPS Design
Criteria, Principles, Goals

- Broad Application Scope: preferable to roll-your-own
- Multi-platform: suits a variety of many-core configs
- High Application Performance: competitive with roll-your-own
- Tunable: expert users can optimize their apps
- Optimized Implementations: is informed by, and informs, hardware
GRAMPS Design: Setup

- Build Execution Graph
- Define programs, stages, inputs, outputs, buffers

- GRAMPS supports graphs with cycles
  - This admits pathological cases.
  - It is worth it to enable the well behaved uses
  - Reminder: target systems-savvy developers
  - Failure/overflow handling? (See Shaders)
GRAMPS Design: Queues

- GRAMPS can optionally enforce ordering
  - Basic requirement for some workloads
  - Brings complexity and storage overheads

- Queues operate at a “packet” granularity
  - “Large bundles of coherent work”
  - A packet size of 1 is always possible, just a bad common case.
  - Packet layout is largely up to the application
GRAMPS Design: Stages

Two* kinds of stages (or kernels)
- Shader (think: pixel shader plus push-to-queue)
- Thread (think: POSIX thread)
- Fixed Function (think: Thread that happens to be implemented in hardware)

✘ What about other data-parallel primitives: scan, reduce, etc.?
GRAMPS Design: Shaders

- Operate on ‘elements’ in a Collection packet
- Instanced automatically, non-preemptible

- Fixed inputs, outputs preallocated before launch
- Variable outputs are coalesced by GRAMPS
  - Worst case, this can stall or deadlock/overflow
  - It’s worth it.
  - Alternatives: return failure to the shader (bad), return failure to a thread stage or host (plausible)
GRAMPS Design: Threads

- Operate on Opaque packets
- No* (limited) automatic instancing
- Pre-emptible, expected to be stateful and long-lived
- Manipulate queues in-place via reserve/commit
GRAMPS Design: Queue sets

- Queue sets enable binning-style algorithms
- A queue with multiple lanes (or bins)
- One consumer at a time per lane
  - Many lanes with data allows many consumers
- Lanes can be created at setup or dynamically

- Bonus: A well-defined way to instance Thread stages safely
Queue Set Example

Checkboarded / tiled sort-last renderer:

- Rasterizer tags pixels based on screen space tile.
- Pixel shading is completely data-parallel.
- Blend / output merging is screen space subdivided and serialized within each tile.
Case Study: Rendering
Reminder of Principles/Goals

- Broad Application Scope
- Multi-Platform
- High Application Performance
- Tunable
- Optimized Implementations
Broad Application Scope

Direct3D Pipeline (with Ray-tracing Extension)

Ray-tracing Graph

[Diagram of the pipeline and ray-tracing graph]

Legend:
- Thread Stage
- Queue
- Shader Stage
- Stage Output
- Fixed-func
- Push Output
Multi-Platform: CPU-like & GPU-like

[Diagram showing multi-platform architecture with sections for CPU-like and GPU-like configurations.]
High Application Performance

- Priority #1: Show scale out parallelism (GRAMPS can fill the machine, capture the exposed parallelism, ...)
- Priority #2: Show ‘reasonable’ bandwidth / storage capacity required for the queues
- Discussion: Justify that the scheduling overheads are not unreasonable (migration costs, contention and compute for scheduling)

× Currently static scheduling priorities
× No serious modeling of texture or bandwidth
### Renderer Performance Data

- Queues are small (< 600 KB CPU, < 1.5 MB GPU)
- Parallelism is good (at least 80%, all but one 95+%)

<table>
<thead>
<tr>
<th></th>
<th>CPU-like Configuration</th>
<th>GPU-like Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fat Core Occup (%)</td>
<td>Micro Core Occup (%)</td>
</tr>
<tr>
<td></td>
<td>Peak Queue Size (KB)</td>
<td></td>
</tr>
<tr>
<td>Teapot</td>
<td>D3D</td>
<td>87.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>510</td>
</tr>
<tr>
<td></td>
<td>Ext. D3D</td>
<td>90.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>582</td>
</tr>
<tr>
<td></td>
<td>Ray Tracer</td>
<td>99.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>156</td>
</tr>
<tr>
<td>Courtyard</td>
<td>D3D</td>
<td>88.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>544</td>
</tr>
<tr>
<td></td>
<td>Ext. D3D</td>
<td>94.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>586</td>
</tr>
<tr>
<td></td>
<td>Ray Tracer</td>
<td>99.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>176</td>
</tr>
<tr>
<td>Fairy</td>
<td>D3D</td>
<td>77.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>561</td>
</tr>
<tr>
<td></td>
<td>Ext. D3D</td>
<td>92.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>605</td>
</tr>
<tr>
<td></td>
<td>Ray Tracer</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>205</td>
</tr>
</tbody>
</table>

**Table 2:** Simulation results: Core thread slot occupancy and peak memory footprint of all graph queues.
Tunability

- **Tools:**
  - Raw counters, statistics, logs
  - Grampsviz

- **Knobs:**
  - Graph topology: e.g., sort-last vs. sort-middle
  - Queue watermarks: e.g., 10x impact on ray tracing
  - Packet sizes: Match SIMD widths, data sharing
Tunability: GRAMPSViz
Optimized Implementations

- Model for impedance matching heterogeneity
- Room to optimize parallel queues
- Room to optimize hardware thread scheduling
  - Shader core or threaded CPU core
Conclusion, Lessons Learned
Summary I: Design Principles

- Make application details opaque to the system
- App: policy (control), system: execution (data)
- Push back against every feature, variant, and special case.
- Only include features which can be run well*
- *Admit some pathological cases when they enable natural expressiveness of desirable cases
Summary II: Key Traits

- Focus on inter-stage connectivity
  - But facilitate standard intra-stage parallelism
- Producer-consumer >> only dependencies / barriers
- Queues impedance match many boundaries
  - Asynchronous (independent) execution
  - Fixed function units, fat – micro core dataflow
- Threads and Shaders (and only those two)
Summary III: Critical Details

- Order is powerful and useful, but optional
- Queue sets: finer grained synchronization and thread instancing without violating the model
- User specified queue depth watermarks as scheduling hints
- Grampsviz and the right (user meaningful) statistics
That’s All

- Thank you, any questions?

- TOG Paper:
  http://graphics.stanford.edu/papers/gramps-tog/

- Funding agencies:
  Stanford PPL, Department of the Army Research, Intel
  Rambus SGF, Intel PhD Fellowship, NSF Fellowship
Bonus Material
Broad Application Scope

Two new apps!

- Cloth Simulation (Collision detection, particle systems)
- A MapReduce App (Enables many things)
Application Scope: Cloth Sim

- Update is not producer-consumer!
- Broad Phase will actually be either a (weird) shader or multiple thread instances.
- Fast Recollide details are TBD.
Application Scope: MapReduce

- **Dynamically** instanced thread stages and queue sets.
- Combine might motivate a formal **reduction shader**.