GRAMPS Overview and Design Decisions

Jeremy Sugerman
February 26, 2009
GCafe

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.

- Kavyon, Solomon, Pat, and Kurt were heavily involved in the GRAMPS 1.0 work, published in TOG.
- Now, it is largely just me, though a number of PPL participants like to kibitz.

Background

- Context: Commodity, heterogeneous, many-core
  - "Commodity": CPUs and GPUs. State of the art 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?

Ex: C run-time, GPU pipeline, GPGPU, MapReduce, …

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 efficient develop programs that efficiently map onto commodity, heterogeneous, many-core platforms.

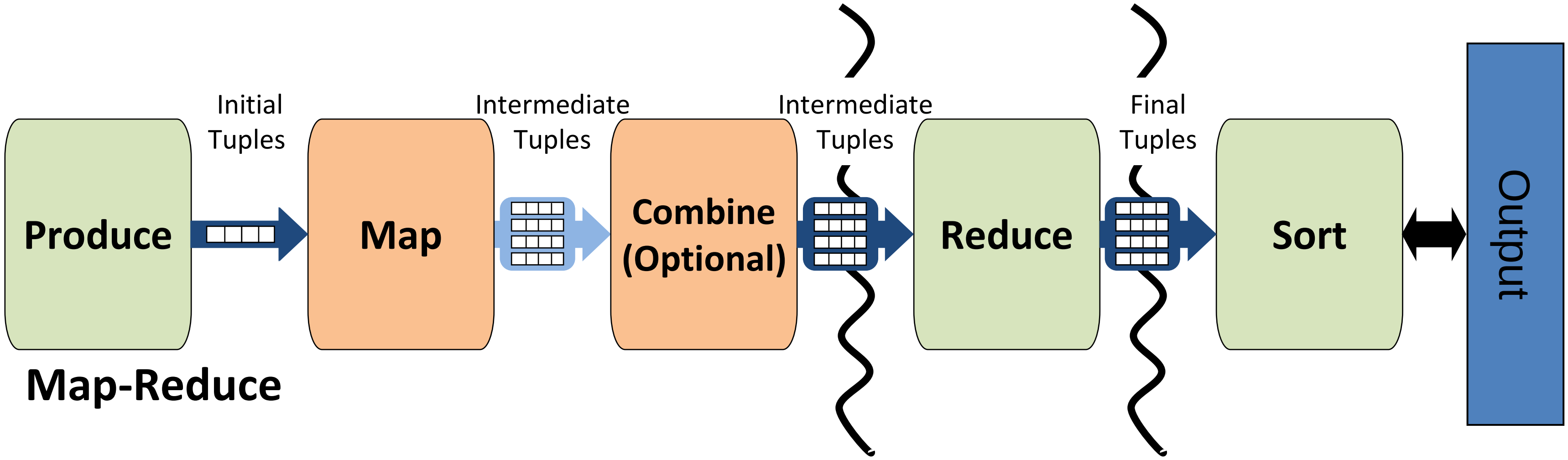
This Talk

1. What is that environment (i.e., GRAMPS)?
2. Why/how did we design it?

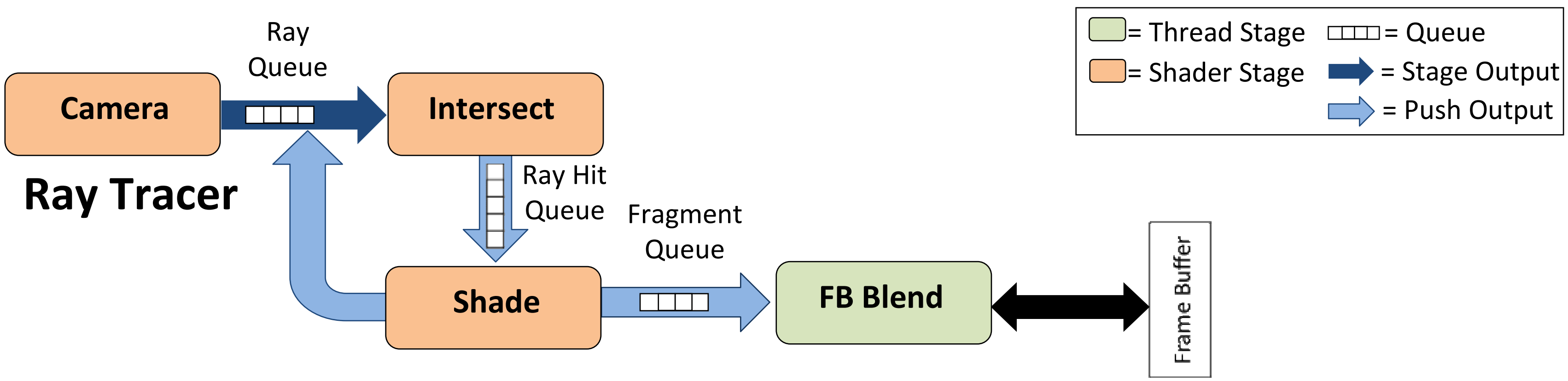
GRAMPS: Quick Introduction

- 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



Map-Reduce



Ray Tracer

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

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: 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

GRAMPS Design: Queues

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

- Queues operate at a “packet” granularity
  - Let the system amortize work and developer group related objects when possible
  - An effective packet size of 1 is always possible, just not a good 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

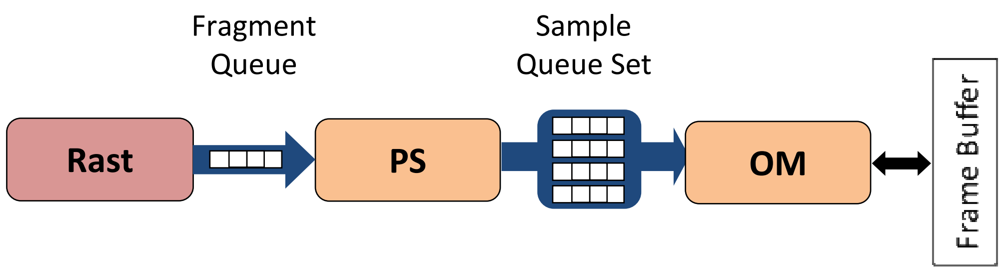
- Operate on Opaque packets
- No/limited automatic instancing
- Pre-emptible, expected to be stateful and long-lived
- Manipulate queues in-place via reserve/commit

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
- A well-defined way to instance Thread stages safely

GRAMPS Design: Queue Set Example

Checkboarded / tiled sort-last renderer:



- Rasterizer tags pixels with their tile
- Pixel shading happens completely data-parallel
- Blend / output merging is screen space subdivided and serialized within each tile

Analysis & Metrics

- Reminder of Principles/Goals
  - Broad Application Scope
  - Multi-Platform
  - High Application Performance
  - Tunable
  - Optimized Implementations

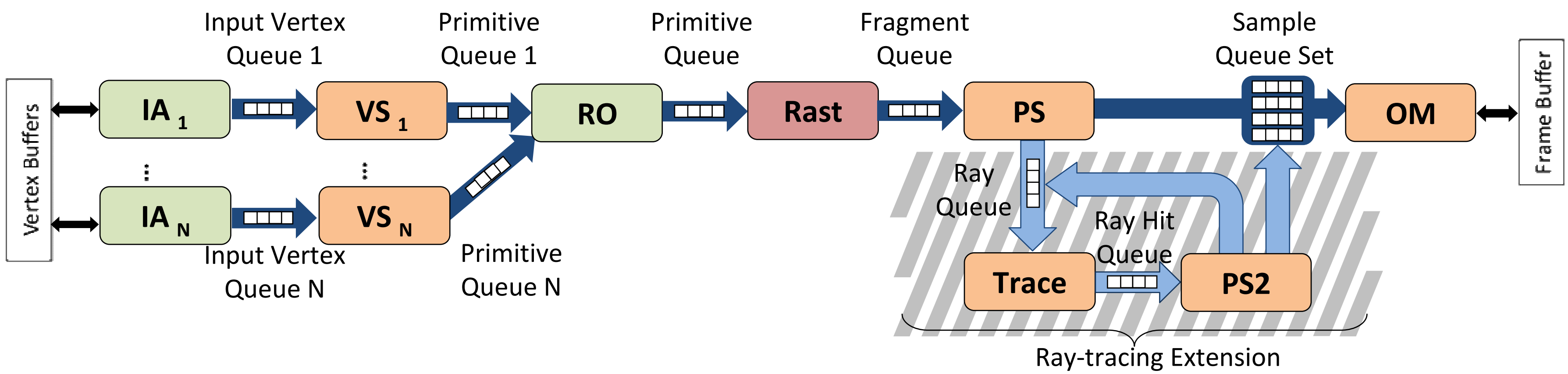
Metrics: Broad Application Scope

- Renderers: Direct3D plus Push extension; Ray Tracer
  - Hopefully a micropolygon renderer
- Cloth Simulation (Collision detection, particle systems)
- A MapReduce App (Enables many things)

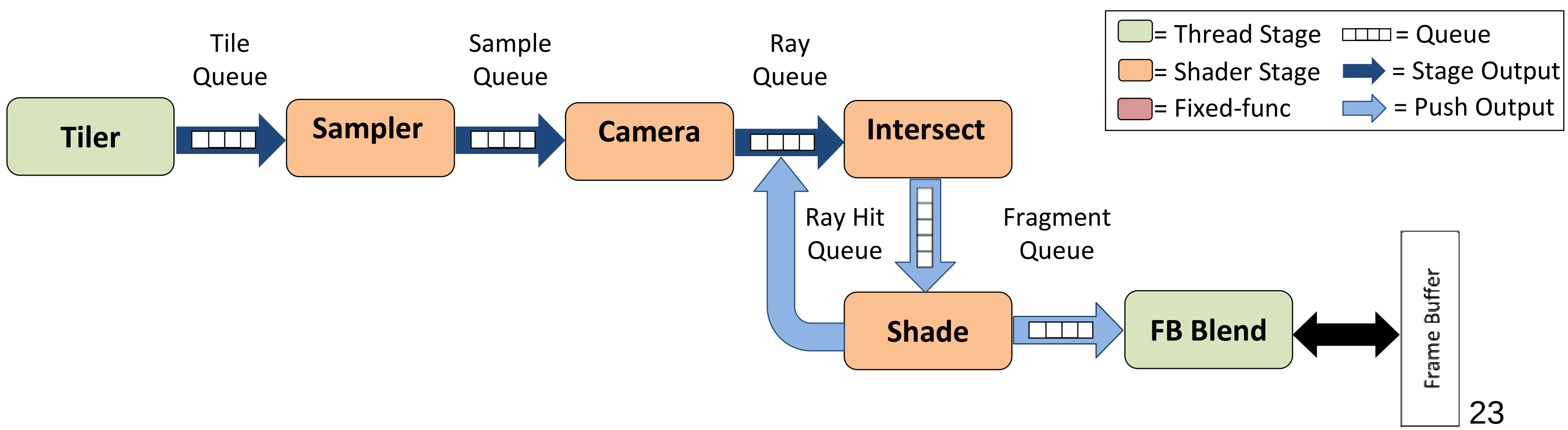
- Convinced? Do you have a challenge? Obvious app?

Application Scope: Renderers

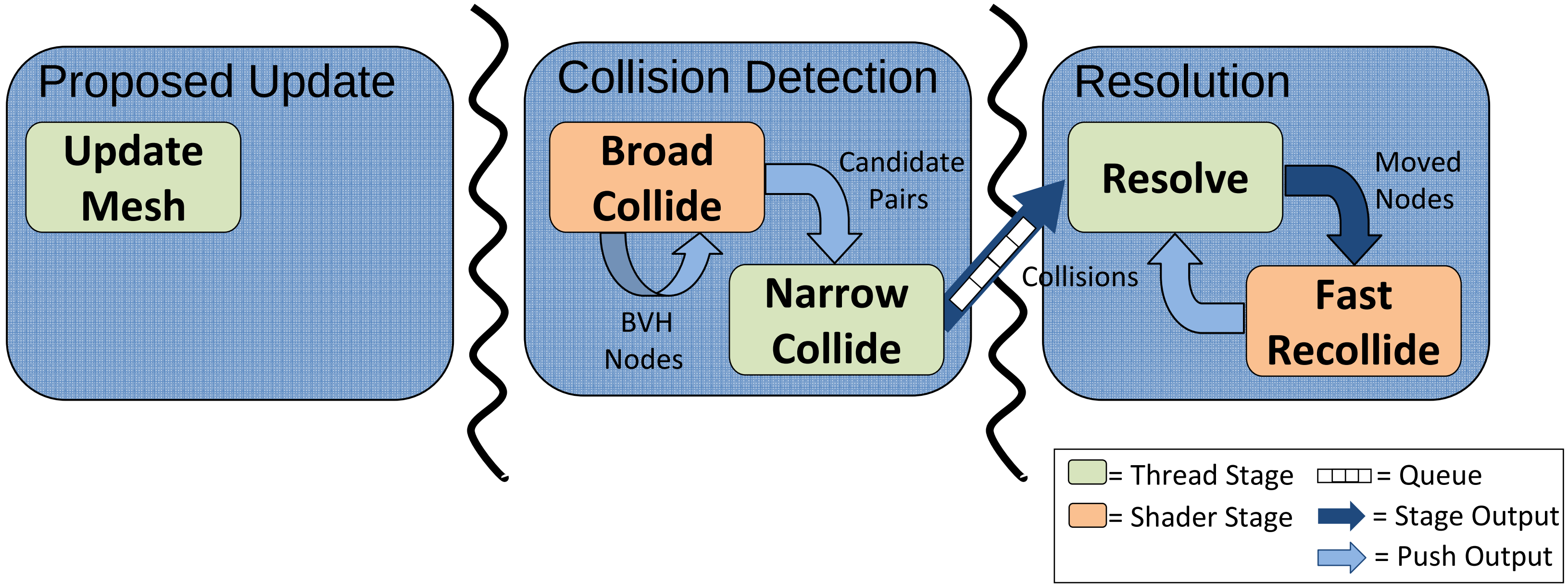
Direct3D Pipeline (with Ray-tracing Extension)



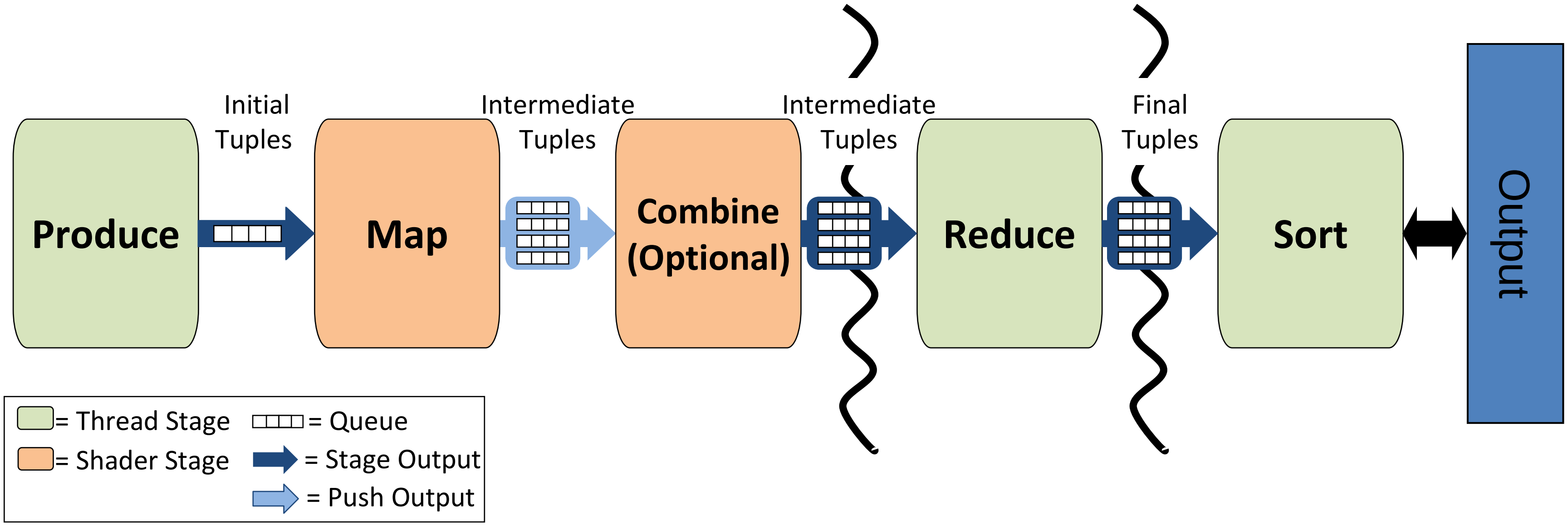
Ray-tracing Graph



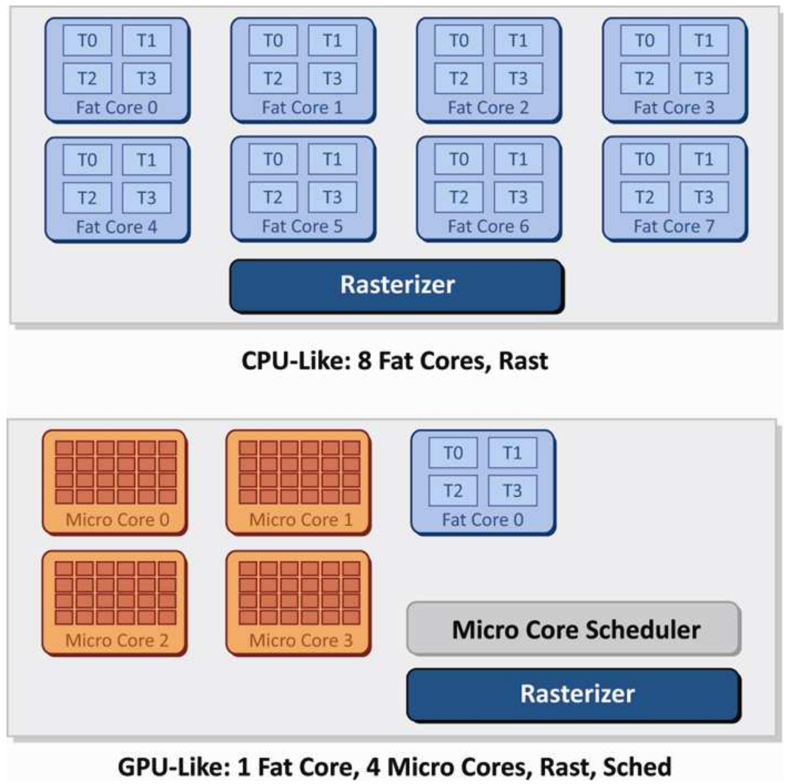
Application Scope: Cloth Sim



Application Scope: MapReduce



Metrics: Multi-Platform



- Convinced? Low hanging / credibility critical additional heterogeneity?

Metrics: High App Performance

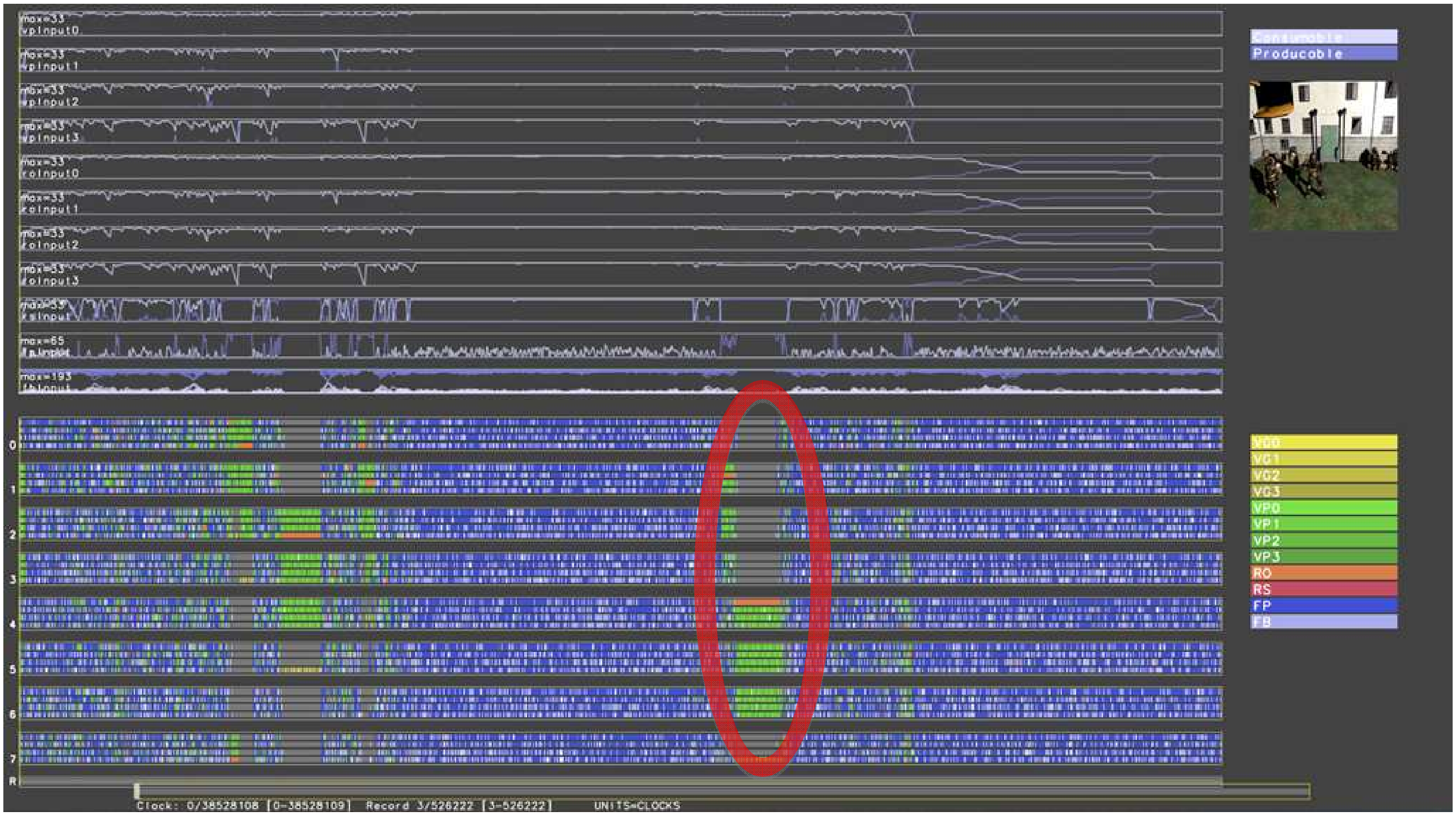
- Priority #1: Show scale out parallelism (GRAMPS can fill the machine, capture the exposed parallelism, …)
- Priority #2: Show 'reasonable' bandwidth / storage capacity required for the queues
- Discussion: Justify that the scheduling overheads are not unreasonable (migration costs, contention and compute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

- ✘ What about bandwidth aware co-scheduling?
- ✘ What about a comparison against native apps?

Metrics: 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



Metrics: Optimized Implementations

- Primarily a qualitative / discussion area
  - Discipline / model for supporting fixed function
  - Ideas for efficient parallel queues
  - Ideas for microcore scheduling
  - Perhaps primitives to facilitate software scheduling

- Other natural hardware vendor takeaways / questions?

Summary I: Design Principles

- Make application details opaque to the system
- 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 with out violating the model
- User specified queue depth watermarks as scheduling hints
- Grampsviz and the right (user meaningful) statistics

That’s All

- Thank you.
- Questions?

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

http://ppl.stanford.edu/internal/display/Projects/GRAMPS