

GRAMPS: A Programming Model for Graphics Pipelines

JEREMY SUGERMAN and KAYVON FATAHALIAN and SOLOMON BOULOS

Stanford University

and

KURT AKELEY

Microsoft Research

and

PAT HANRAHAN

Stanford University

We introduce GRAMPS, a programming model that generalizes concepts from modern real-time graphics pipelines by exposing a model of execution containing both fixed-function and application-programmable processing stages that exchange data via queues. GRAMPS allows the number, type, and connectivity of these processing stages to be defined by software, permitting arbitrary processing pipelines or even processing graphs. Applications achieve high performance using GRAMPS by expressing advanced rendering algorithms as custom pipelines, then using the pipeline as a rendering engine. We describe the design of GRAMPS, then evaluate it by implementing three pipelines—Direct3D, a ray tracer, and a hybridization of the two—and running them on emulations of two different GRAMPS implementations: a traditional GPU-like architecture; and a CPU-like multi-core architecture. In our tests, our GRAMPS schedulers run our pipelines with 500 to 1500 KB of queue usage at their peaks.

Categories and Subject Descriptors: I.3.1 [**Computer Graphics**]: Hardware Architecture—*Parallel Programming*

General Terms: graphics pipelines, many-core architectures, GPUs, stream computing, parallel programming

1. INTRODUCTION

Current GPUs are able to render complex, high-resolution scenes in real time using Z-buffer rasterization-based techniques. However, the real-time photorealistic rendering problem is not solved, and there remains interest in advanced rendering algorithms such as ray tracing, REYES, and combinations of these with the traditional graphics pipeline. Unfortunately these advanced rendering pipelines perform poorly when implemented on current GPUs.

While the earliest GPUs were simple, application-configurable engines, the history of high-performance graphics over the past three decades has been the co-evolution of a pipeline abstraction (the traditional

{yoel, kavyonf, boulos}@graphics.stanford.edu, kakeley@microsoft.com, hanrahan@graphics.stanford.edu

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0730-0301/20YY/0100-0001 \$5.00

graphics pipeline) and the corresponding driver/hardware devices (GPUs). In the recent past, the shading stages of the pipeline became software programmable. Prior to the transition, developers controlled shading by toggling and configuring an assortment of fixed options and parameters, but the widespread innovation in shading techniques led to an increasingly complex matrix of choices. In order to accommodate the trend towards more general shading, researchers and then graphics vendors added programmable shading to the graphics pipeline.

We see an analogy between the evolution from fixed to programmable shading and the current changes for enabling and configuring pipeline stages. After remaining static for a long time, there are a variety of new pipeline topologies available and under exploration. Direct3D 10 added new geometry and stream-output stages [Blythe 2006]. The Xbox 360 added a new stage for tessellation (future iterations of Direct3D will likely follow). We believe that future rendering techniques and increasing non-graphical usage will motivate more new pipeline stages and configuration options. As was true with pre-programmable shading, these new abilities are currently all delivered as predefined stage and pipeline options to be toggled and combined. Looking forward, we instead propose a programmably constructed graphics pipeline.

Our system, GRAMPS, is a programming model designed for future GPUs. It is motivated by the requirements of rendering applications, but provides a general set of abstractions for building parallel applications with both task and data-level parallelism. GRAMPS derives key ideas from OpenGL/Direct3D, but does not specify a pipeline with a fixed sequence of stages. Instead it allows applications to create custom pipelines. Pipelines can contain common fixed or programmable stages, but in arbitrary topologies. Thus, GRAMPS itself is not a rendering pipeline, but is a model and toolkit that allows rendering pipelines—and any applications than can be formulated as asynchronously communicating independent pipelines or state machines—to be programmably constructed and run.

The specific goals of the GRAMPS are:

- High performance.** An implementation of a traditional graphics pipeline, built as a layer above GRAMPS, should give up little performance over a native implementation. Advanced rendering pipelines should have high absolute performance, making efficient use of the underlying computation engine, special-function units, and memory resources.
- Large application scope.** It should be possible to express a wide range of advanced rendering algorithms using the GRAMPS abstraction. Developers should find GRAMPS more convenient and more effective than using roll-your-own approaches.
- Optimized implementations.** The GRAMPS model should provide sufficient opportunity (and clarity of intent) for implementations to be tuned in support of it.

While GRAMPS was conceived to fit future revisions of current GPU designs, we believe it is also a useful model for programming a very different more general purpose throughput-oriented ‘GPU’ like Intel’s Larrabee [Seiler et al. 2008]. As such, a further goal of GRAMPS is that it provide an effective abstraction for a range of alternate architectures, achieving the itemized goals above *while also affording* improved application portability.

Our primary contribution is the GRAMPS programming model with its central tenet of computation as a graph of stages operating asynchronously and exchanging data via queues. To demonstrate the plausibility and applicability of this approach, we evaluate it in several ways. First, in Section 4, we demonstrate application scope by implementing three rendering pipelines—Direct3D, a packet ray tracer, and a pipeline extending Direct3D to add ray traced shadow computations—using the GRAMPS abstraction. Second,

in Section 5, we demonstrate implementation scope by describing two GRAMPS implementations: one on a traditional GPU-like infrastructure modeled after NVIDIA’s 8-series architecture; and the other on an alternate architecture patterned after Intel’s Larrabee. Then, in Section 6, we measure and analyze the behavior of our renderers on our implementations to show how our initial work with GRAMPS progresses towards its goals. Of course, the ultimate validation of whether GRAMPS achieves its goals can come only if optimized systems inspired by its programming model, concepts, and constructs become widespread and successful.

2. BACKGROUND AND RELATED WORK

2.1 Throughput Architectures

An increasing number of architectures aim to deliver high performance to application domains, such as rendering, that benefit from parallel processing. These architectures omit hardware logic that maximizes single-threaded performance in favor of many simple processing cores that contain large numbers of functional units.

The most widely available, and most extreme, examples of such architectures are GPUs. NVIDIA’s 200-Series [Lindholm et al. 2008] and ATI’s HD 4800-Series [AMD 2008a] products are built around a pool of highly multi-threaded programmable cores tuned to sustain roughly a teraflop of performance when performing shading computations. GPUs provide additional computing capabilities via fixed-function units that perform tasks such as rasterization, texture filtering, and frame buffer blending.

Commodity high-throughput processing is no longer unique to GPUs. The CELL Broadband Engine [Pham et al. 2005], deployed commercially in the Playstation 3, couples a simplified PowerPC core with eight ALU-rich in-order cores. SUN’s UltraSPARC T2 processor [Kongetira et al. 2005] features eight multi-threaded cores that interact via a coherent shared address space. Intel has demonstrated a prototype 80-core “terascale” processor, and recently announced plans to productize Larrabee, a cache-coherent multi-core X86-based GPU [Seiler et al. 2008].

This landscape of high-performance processors presents interesting choices for future rendering system architects. GPUs constitute a simple to use, heavily-tuned platform for rasterization-based real-time rendering but offer only limited benefits for alternative graphics algorithms. In contrast, increasingly parallel CPU-based throughput architectures offer the flexibility of CPU programming, but implementing an advanced rendering system that leverages multi-core, multi-threaded, and SIMD processing is a daunting task.

2.2 Programming Models

Real-time Graphics Pipelines: OpenGL and Direct3D [Segal and Akeley 2006; Blythe 2006] provide developers a simple, vendor-agnostic interface for describing real-time graphics computations. More importantly, the graphics pipeline and programmable shading abstractions exported by these interfaces are backed by highly-tuned GPU-based implementations. By using rendering-specific abstractions (such as vertices, fragments, and pixels) OpenGL/Direct3D maintain high performance without introducing difficult concepts such as parallelism, threads, asynchronous processing, or synchronization. The drawback of these design decisions is limited flexibility. Applications must be restructured to conform to the pipeline that OpenGL/Direct3D present. A fixed pipeline makes it difficult to implement many advanced rendering techniques efficiently. Extending the graphics pipeline with domain-specific stages or data flows to provide new

functionality has been the subject of many proposals [Blythe 2006; Hasselgren and Akenine-Möller 2007; Bavoil et al. 2007].

Data-Parallel Programming on GPUs: General-purpose interfaces for driving GPU execution include low-level native frameworks such as AMD’s CAL [AMD 2008b], parallel programming languages such as NVIDIA’s CUDA [NVIDIA 2007], and third-party programming abstractions layered on top of native interfaces [Buck et al. 2004; McCool et al. 2004; Tarditi et al. 2006]. These systems share two key similarities that make them poor candidates for describing the mixture of both regular and highly dynamic algorithms that are present in advanced rendering systems. First, with the exception of CUDA’s support for filtered texture access, they expose only the GPU’s programmable shader execution engine (rasterization, compositing, and z-buffering units are not exposed). Second, to ensure high GPU utilization, these systems model computation as large data-parallel batches of work. Describing computation at large batch granularity makes it difficult to efficiently couple regular and dynamic execution.

Parallel CPU Programming: Basic threading libraries (such as POSIX threads) and vector instruction intrinsics are available for all modern CPU systems. They constitute fundamental building blocks for any parallel application, but place the entire burden of achieving good performance on application developers. Writing software using these primitives is known to be very difficult and a successful implementation for one machine often does not carry over to another. Due to these challenges, high-level parallel abstractions, such as Intel’s Thread Building Blocks [Intel 2008], which provides primitives such as work queues, pipelines, and threads, are becoming increasingly important. We highlight Carbon [Kumar et al. 2007] as an example of how generic high-level abstractions permit hardware acceleration of dynamic parallel computations.

2.3 Streaming

There is a wide range of work under the umbrella of generic “stream computing”—processors, architectures, programming models, and compilation techniques [Kapasi et al. 2002; Dally et al. 2003; Thies et al. 2002]. Streaming research seeks to build maximally-efficient throughput-oriented platforms by embracing principles such as data-parallel execution, high levels of (producer-consumer) memory locality, software management of the system memory hierarchy, and asynchronous bulk communication. In general, streaming research has focused on intensive static compiler analysis to perform key optimizations like data prefetching, blocking, and scheduling of asynchronous data transfers and kernel execution. Static analysis works best for regular programs that exhibit predictable data access and tightly bounded numbers of kernel inputs and outputs [Das et al. 2006]. Irregular computations are difficult to statically schedule because program behavior is not known at compile time. Unfortunately, graphics pipelines contain irregular components, and standard offline stream compilation techniques are insufficient for high performance.

GRAMPS embraces many of the same concepts and principles as streaming, but makes the fundamental assumption that applications are dynamic and irregular with unpredictable data-dependent execution. Thus GRAMPS inherently requires a model where data locality and efficient aggregate operations can be identified and synthesized at run time. GRAMPS’s stateful thread stages meet this need by enabling applications to explicitly aggregate and queue data dynamically. In addition, they are more pragmatically aligned with the capabilities of commodity processors and multi-core systems than traditional stream kernels.

We believe that GRAMPS and previous streaming work are complementary. A natural GRAMPS extension would permit applications to identify stages with predictable data flow during program initialization. In these cases GRAMPS could employ upfront streaming-style analysis and transformations that simplify or eliminate run-time logic.

2.3.1 *Streaming Rendering.* Prior research has explored using stream processors/streaming languages for rendering. Owens et al. implemented both REYES and OpenGL on Imagine [2002], Chen et al. implemented an OpenGL-like pipeline on Raw [2005], and Purcell introduced a streaming formulation of ray tracing [2004]. Each of these systems suffered from trying to constrain the dynamic irregularity of rendering in predictable streaming terms. Both the Imagine and Raw implementations redefined and recompiled their pipelines for each scene and frame they rendered. Additionally, they manually pre-rendered each frame to tune their implementations and offset the dynamic characteristics of rendering. Streaming ray tracing has always struggled with load-balancing [Foley and Sutherland 2005; Horn et al. 2007]. Initial multi-pass versions tried depth culling and occlusion queries with mixed success. Follow up single-pass techniques used branches, but suffered from divergent control flow and varying shader instance running times.

In the four to six years since those systems were first built, rendering algorithms and implementations have become significantly more dynamic: branching in shaders is routine as well as composing final frames from large numbers of off-screen rendering passes. With GRAMPS, we have set out to create a model whose run-time scheduling and on-demand instancing of data-parallel kernels can adaptively handle the variance in rendering workloads without manual programmer intervention or redefining the execution graph. Additionally, the aforementioned rendering systems considered only homogeneous hardware—custom Imagine and Raw processors and GPU shader cores. They would struggle to incorporate specialized rasterization units, for example, whereas the GRAMPS model consciously includes heterogeneity.

3. GRAMPS DESIGN

GRAMPS is a General Runtime/Architecture for Multicore Parallel Systems. It defines a programming model for expressing rendering pipelines and other parallel applications. It exposes a small, high-level set of primitives designed to be simple to use, to exhibit properties necessary for high-throughput processing, and to permit optimized hardware implementations. We intend for GRAMPS implementations to involve various combinations of software and underlying hardware support, similar to how OpenGL/Direct3D APIs permit flexibility in an implementation’s choice of driver and GPU hardware responsibilities. However, unlike OpenGL/Direct3D, we envision GRAMPS as a lower-level abstraction upon which graphics application toolkits and domain-specific abstractions (such as OpenGL or Direct3D) are built.

GRAMPS is organized around the basic concept of application-defined computation stages executing in parallel and communicating asynchronously via queues. We believe that this relatively simple producer-consumer parallelism is fundamental across a broad range of throughput applications. Unlike a GPU pipeline, where inter-stage queues specifically hold vertices, fragments, and primitives, GRAMPS graph execution is decoupled from detailed application-specific semantics. GRAMPS refines and extends this model with the abstractions of shader stages and queue sets to allow applications to further expose data-parallelism within a stage.

The following sections describe the primary abstractions used by GRAMPS computations: graphs, stages, queues, and data buffers. We highlight the role of each of these abstractions in building efficient graphics pipelines.

3.1 Execution Graphs

The execution, or computation, graph is the GRAMPS analog of the GPU pipeline. It organizes the execution of shaders/kernels and threads into stages and limits data flow into and out of stages to access to first-class queues and buffers. In addition to specifying the basic information required for GRAMPS to initialize and

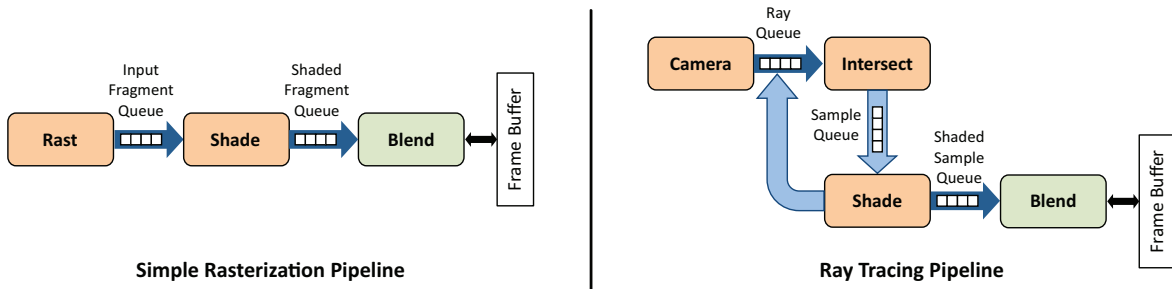


Fig. 1. Simplified GRAMPS graphs for a rasterization-based pipeline (left) and ray-tracer (right). The ray-tracing graph contains a loop. The early stages are automatically instantiated for parallelism while the Blend stages are both singletons to provide framebuffer synchronization.

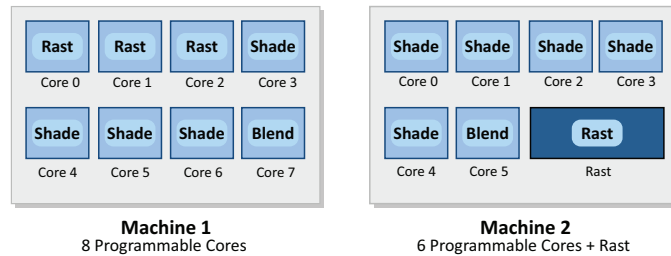


Fig. 2. Execution of the rasterization pipeline (top of Figure 1) on a machine with 8 cores and on a machine with 6 cores and a HW rasterizer. Instanced stage execution enables GRAMPS to utilize all machine resources. Each processing resource is labeled with the stage it is assigned to execute.

start the application, the graph provides valuable information about a computation that is essential to scheduling. An application specifies its graph to GRAMPS via a programmatic ‘driver’ interface that is modeled after Direct3D 10’s interface for creating and configuring shaders, textures, and buffer objects.

Figure 1 shows two examples of GRAMPS graphs excerpted from our actual renderers in Section 4. The first example illustrates part of a conventional 3D pipeline containing stages for rasterization, fragment shading, and frame buffer blending. The second example comes from a ray tracer and highlights that GRAMPS accepts full graphs, not just DAGs or pipelines.

GRAMPS supports general computation graphs to provide flexibility for a rich set of rendering algorithms. Graph cycles inherently make it possible to write applications that feedback endlessly through stages and amplify queued data beyond the ability of any system to manage. Thus, GRAMPS, unlike OpenGL/Direct3D, does not guarantee that all legal programs robustly make forward progress and execute to completion. Instead, we designed GRAMPS to encompass a larger set of applications that run well, at the cost of allowing some that do not.

Forbidding cycles would allow GRAMPS to guarantee forward progress—at any time it could stall a stage that was over-actively producing data until downstream stages could drain outstanding work from the system—at the expense of excluding some irregular workloads. For example, both the formulation of ray tracing and the proposed Direct3D extension described in Section 4 contain cycles in their graph structure. Sometimes cycles can be eliminated by “unrolling” a graph to reflect a maximum number of iterations, bounces, etc. However, not only is unrolling cumbersome for developers, it is awkward in irregular cases, such as when

different rays bounce different numbers of times according to local material properties. While handling cycles increases the scheduling burden for GRAMPS, it remains possible to effectively execute many graphs that contain them. We believe that the flexibility that graphs provide over pipelines and DAGs outweighs the cost of making applications take responsibility for ensuring they are well-behaved. The right strategy for notifying applications and allowing them to recover when their amplification swamps the system is an interesting avenue for future investigation.

3.2 Stages

GRAMPS stages correspond to nodes in the execution graph and are the analog of GPU pipeline stages. The fundamental reason to partition computation into stages is to increase performance. Stages operate asynchronously and therefore expose parallelism. More importantly, stages encapsulate phases of computation and indicate computations that exhibit similar execution or data access characteristics (typically SIMD processing or memory locality). Grouping these computations together yields opportunities for efficient processing. GRAMPS stages are useful when the benefits of coherent execution outweigh the costs of deferred processing.

A GRAMPS stage definition consists of:

- Type: Either shader, thread, or fixed-function.
- Program: Either program code for a shader/thread or configuration parameters for a fixed-function unit.
- Queues: Input, Output, and “Push” queue bindings.
- Buffers: Random-access, fixed-size data bindings.

We expect GRAMPS computations to run on platforms with significantly larger numbers of processing resources than computation phases. Thus, GRAMPS executes multiple copies of a stage’s program in parallel (each operating on different input queue data) to fill an entire machine. We refer to each executing copy of a stage program as an instance. Phases that require serial processing (initialization is a common example) execute as singleton stages. The diagram at left in Figure 2 illustrates execution of the three-stage rasterization pipeline on a machine with eight cores. Each core is labeled by the stage instance it executes. GRAMPS fills the machine with instances of Rast and Shade stage programs. In this simple example, Blend is serialized to preserve consistent and globally-ordered frame buffer update (the Blend stage executes as a singleton).

GRAMPS supports three types of stages that correspond to distinct sets of computational characteristics. A stage’s type serves as a hint facilitating work assignment, resource allocation, and computation scheduling. We strove for a minimal number of simple abstractions and concluded that fixed-function processing and GPU-style shader execution constituted two unique classes of processing. To support wider varieties of rendering techniques, we chose to add an additional, general purpose and stateful stage type rather than augment the existing shader concept with features that risked decreasing its simplicity and performance.

Shaders: Shader stages define short-lived, run-to-completion computations akin to traditional GPU shaders. They are designed as an efficient mechanism for running data-parallel regions of an application. Like GPU shaders, GRAMPS shader programs are written to operate per-element, which makes them stateless and enables multiple instances to run in parallel. GRAMPS manages queue inputs and outputs for shader instances automatically, which simplifies shader programs and allows the scheduler to guarantee they can run to completion without blocking. Unlike GPU shaders, GRAMPS shaders may use a special “push”

(Section 3.3) operation for conditional output. As a result of these properties, GRAMPS shader stages are suitable for large-scale automatic instancing and wide-SIMD processing for many of the same reasons as GPU shaders [Blythe 2006]. And, also like GPUs, GRAMPS actually creates and schedules shader instances in packets—many-instance groups, despite their element-wise programming model—in order to amortize overhead and better map to hardware.

Threads: Thread stages are best described as traditional CPU-style threads. They are designed for task-parallel, serial, and other regions of an application best suited to large per-element working sets or operations dependent on multiple elements at once (e.g., reductions or re-sorting of data). Unlike shaders, thread stages are stateful and thus must be manually parallelized and instanced by the application rather than automatically by GRAMPS. They explicitly manipulate queues and may block, either when input is not yet available or too much output has not yet been consumed. Thread stages are expected to most likely fill one of two roles: repacking data between shader stages, and processing bulk chunks of data where sharing/reuse or cross-communication make data-parallel shaders inefficient.

Fixed-function: GRAMPS allows stages to be implemented by fixed-function or specialized hardware units. Just like all other stages, fixed-function stages inter-operate with the rest of GRAMPS by exchanging data via queues. Applications configure these units via GRAMPS by providing hardware-specific configuration information at the time of stage specification.

3.3 Queues

GRAMPS stages communicate and exchange data via queues that are built up of work packets. Stages asynchronously produce and consume packets using GRAMPS intrinsics. Each queue in a GRAMPS graph also specifies its capacity in packets. As alluded to in the discussion of graphs with cycles, there are two possible strategies for queue growth: enforce a preset maximum capacity and report errors on overflow, or grow without bounds (at least until all available memory is exhausted). Our current implementations treat capacity as a hard limit, but we are also interested in treating it as a scheduling hint in conjunction with an overflow mechanism for handling spilling.

To support applications with ordering requirements (such as OpenGL/Direct3D), GRAMPS queues are strictly FIFO by default. Maintaining FIFO order limits parallelism within instanced stages and incurs costs associated with tracking and buffering out-of-order packets. GRAMPS permits execution graphs to tag any queue as unordered when the application does not require FIFO ordering.

3.3.1 Packets. GRAMPS queues contain homogeneous collections of data packets that adhere to one of two formats. A queue’s packet format is defined when the queue is created.

- Opaque:** Opaque packets are for bundles of work/data that GRAMPS has no need to interpret. The application graph specifies only the size of Opaque packets so they can be enqueued and dequeued by GRAMPS. The layout of an Opaque packet’s contents is entirely defined and interpreted by the logic of stages that produce and consume it.
- Collection:** Collection packets are for queues with at least one end that is bound to a shader stage. Although GRAMPS shader instances operate individually on data elements, GRAMPS dispatches groups of shader instances simultaneously. Together, a group of shader instances process all the elements in a Collection packet. Collection packets contain a set of independent elements plus a shared header. The application graph specifies sizes for the overall packet, the header, and the elements. GRAMPS defines the layout of system-interpreted fields in the packet header (specifically, the first word is a count of valid

elements in the packet). The remainder of the header and internal layout of elements are application defined and opaque to GRAMPS.

The inclusion of thread stages influenced our decision to compose queues using packets rather than adopt an element-based abstraction. Thread stages produce and consume data in units of packets that may contain data in any form (not just collections of elements). Additionally, by giving threads a view on queue data that spans multiple elements, GRAMPS provides a mechanism for threads to produce Collection-format packets and use the header for inter-element sharing. We commonly size packets to multiples of a machine’s SIMD width and store data aligned for vector hardware.

GRAMPS provides three intrinsics for queue manipulation: `reserve`, `commit`, and `push`. `reserve` and `commit` operate on packets while `push` provides a method for shaders to enqueue individual elements and have GRAMPS coalesce them into complete packets.

3.3.2 Queue Manipulation: Thread/Fixed Stages. Thread and fixed-function stages always operate on queues via `reserve` and `commit`, which operate in-place. `reserve` returns the caller a “window” that is a reference to one or more contiguous packets. GRAMPS guarantees the caller exclusive access to this region of the queue until it receives a corresponding `commit` notification. An input queue `commit` indicates that packet(s) have been consumed and can be reclaimed. Output queue `commit` operations indicate the packet(s) are now available for downstream stages.

The queue `reserve-commit` protocol allows stages to perform in-place operations on queue data and allows GRAMPS to manage queue access and underlying storage. Queue windows permit various strategies for queue implementation and add a degree of indirection that enables customized implementations for systems with distributed address spaces, explicit prefetch, or local store capabilities.

3.3.3 Queue Manipulation: Shader Stages. Shaders do not explicitly perform these operations, but GRAMPS transparently arranges for shader inputs and outputs to be manipulated in-place using the same underlying mechanisms. As shader input packets arrive in a queue, GRAMPS internally obtains corresponding output packet reservations. Once the requisite reservations are obtained, GRAMPS runs the packet’s worth of shader instances. Each instance receives a reference to the shared packet header and to one element in each of the pre-reserved input and output packets. When all of the instances have completed, GRAMPS internally commits the inputs and outputs. By pre-reserving shader packets, GRAMPS guarantees that the commit order of output packets corresponds exactly to the order of input packets, and preserves order across stages.

Input and output pre-reservation reflects GRAMPS shaders’ antecedents in GPUs, but GRAMPS offers shaders one operation that significantly extends their flexibility. In addition to fixed inputs and outputs, shaders can dynamically insert elements into *unordered* output queues using `push`. GRAMPS accumulates pushed elements into Collection packets. It coalesces as full a packet as possible, sets the element count in the header, and enqueues the packet for downstream consumption.

`push` allows heavily instanced independent shader programs to build dense packets despite variable, conditional, and potentially sparse output from any single instance. As opposed to in-place `reserve` and `commit`, `push` copies its data into a temporary buffer maintained by GRAMPS. Not only is packet coalescing simplified by the copy, it also allows a shader instance to `push` in a single atomic operation. Also, `push` is a simpler operation than `reserve` and `commit` which is consistent with the lightweight, simple nature of shader execution.

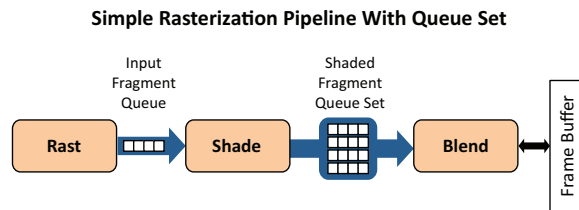


Fig. 3. Replacing the Blend input queue from Figure 1 with a queue set enables parallel instancing.

3.4 Queue Sets

The rasterization pipeline example from Figure 1 serializes frame buffer updates through a singleton Blend stage. Performant renderers typically parallelize frame buffer operations using checker-boarding or tiling to subdivide the screen into disjoint regions. Non-overlapping regions modify different frame buffer pixels and are thus free of relative ordering constraints. Processing within each region can be serialized as a simple method to ensure correct fragment ordering while different regions can be run in parallel. In the example, Blend’s single input queue makes it impossible for GRAMPS to distinguish which elements (fragments) can be processed in parallel. The application author could potentially increase parallelism by creating separate queues and Blend threads per screen region. Manual replication of queues and stages would not only be tedious, but also preclude the automatic instancing and queue management benefits of shader stages.

GRAMPS provides queue sets to enable this idiom for applications using shaders. A queue set functions like N subqueues bundled together as a single logical queue. GRAMPS instances shaders to process different subqueues independently, but ensures that at most one packet per subqueue is consumed at a time.

Stages add data into an output queue set (via `reserve` or `push`) by explicitly specifying the subqueue to manipulate. On an input queue set, `reserve` takes only the logical queue. GRAMPS is free to select any subqueue as a source of packets to satisfy the `reserve`. Figure 3 shows the example rendering pipeline recast with a queue set between its Shade and Blend stages. With this modification, Shade now inserts fragments into subqueues based on pixel location. Multiple instances of Blend process different subqueues and update the frame buffer in parallel.

3.5 Buffers

GRAMPS buffers are untyped random-access memory objects that are bound to stages and fill a similar roll to GPU constant buffers, unfiltered textures, and render buffers. The application sizes and optionally initializes buffer contents during graph setup in a manner similar to creating and binding Direct3D 10 pipeline resources. A stage buffer binding specifies one of the following permissions: read-only, write-only, read-write (local), or read-write (coherent). Read-write (local) access provides no guarantee when (or if) modifications will become visible to any other instance or stage. Read-write (coherent) is reserved for full shared memory, but is currently unsupported and unimplemented by GRAMPS. Like queues, programs access buffers via `reserve/commit` windows. Buffer windows make it possible to implement GRAMPS on machines without a unified memory model and provide GRAMPS with explicit notifications of active memory regions. An implementation may leverage these notifications to trigger optimizations such as bulk pre-fetch on `reserve` or cache flushes on `commit`.

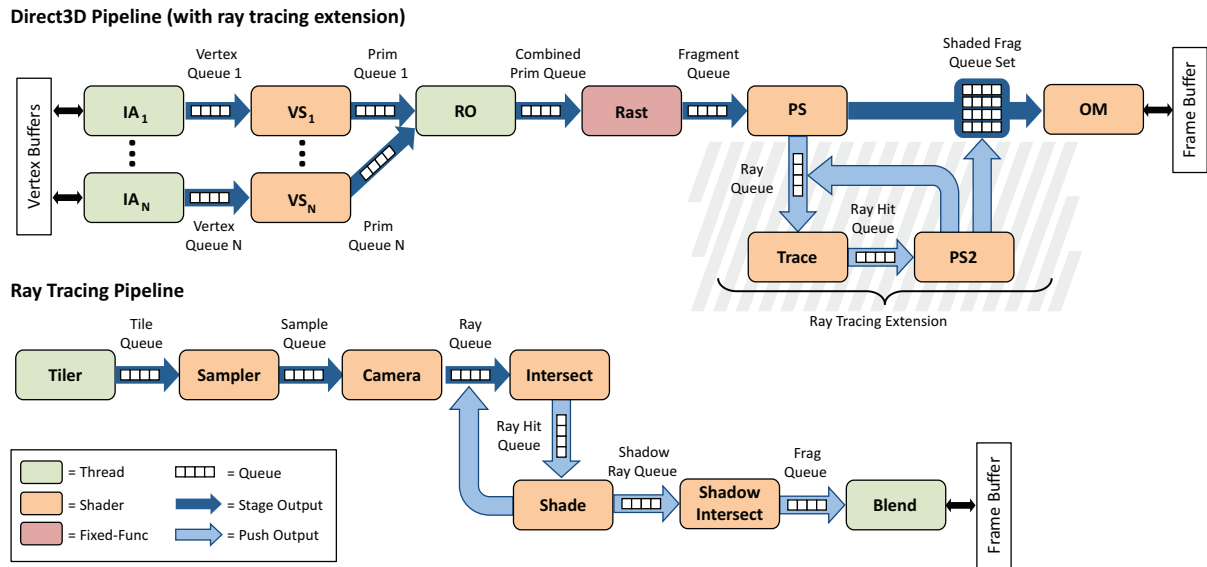


Fig. 4. GRAMPS graphs corresponding to Direct3D and ray tracing rendering pipelines. The two stages added to the Direct3D graph (labeled “Ray Tracing Extension”) provide the ability to cast and trace rays from fragment shader programs.

3.6 Summary

We summarize the process of describing and running a GRAMPS computation into three key steps. First, an application creates GRAMPS stages, queues, and buffers. Next, the queues and buffers are bound to stages forming a computation graph. Last, GRAMPS executes the computation defined by this graph to completion.

The abstractions available for graph creation embody key GRAMPS ideas. Most importantly, computation graphs are fully programmable, not a configuration of pre-defined stages and data flows. GRAMPS permits arbitrary graph topologies by allowing for variable numbers of stages and allowing these stages to be programmatically wired together via explicitly named queues.

Second, GRAMPS embraces the need to dynamically aggregate work at runtime to achieve high performance in the face of irregularity. GRAMPS queues consist of packets, not individual data elements. Mechanisms like `push` and direct thread-stage packet manipulation allow for dynamic work aggregation into packets sized for high-throughput processing.

Last, GRAMPS embraces both workload and system implementation heterogeneity. Shader stages and Collection packet queues permit specialization for the case of data-parallel execution. In addition, fixed-function stages allow GRAMPS computations to leverage special-purpose hardware when present.

4. GRAPHICS PIPELINES USING GRAMPS

In this section we describe three example rendering systems framed in terms of GRAMPS: a simplified Direct3D pipeline, a packet-based ray tracer, and a hybrid that augments the simplified Direct3D pipeline

with additional stages used for ray tracing.

4.1 Direct3D

The GRAMPS graph corresponding to our sort-last formulation of a simplified Direct3D pipeline is shown at the top of Figure 4. A major challenge of a Direct3D implementation is exposing high levels of parallelism while preserving Direct3D fragment ordering semantics.

The pipeline’s front-end consists of several groups of Input Assembly (IA) and Vertex Shading (VS) stages that operate in parallel on disjoint regions of the input vertex set. Currently, we manually create these groups, built-in instancing of subgraphs is a potentially useful future addition to GRAMPS. Each IA/VS group produces an ordered stream of post transform primitives. Each input is assigned a sequence number so that these streams can be collected and totally-ordered by a singleton Reorder (RO) stage before being delivered to the fixed-function rasterizer (Rast).

The pipeline back-end starts with a Pixel Shader (PS) stage that processes fragments. After shading, fragment packets are routed to the appropriate subqueue in the output queue set based on their screen space position, much like described in Section 3.4. The queue set lets GRAMPS instance the Output Merger while still guaranteeing that fragments are blended into the frame buffer atomically and in the correct order. Note that Rast facilitates this structure by scanning out packets of fragments that never cross the screen space routing boundaries.

Notice that the Direct3D graph contains no stages that correspond to fixed-function texture filtering. While a GRAMPS implementation is free to provide dedicated texturing support (as modern GPUs do through special instructions), special-purpose operations that occur within a stage are considered part of its internal operation, not part of the GRAMPS programming abstraction or any GRAMPS graph.

4.2 Ray Tracer

Our implementation of a packet-based ray tracer maps natural components of ray tracing to GRAMPS stages (bottom of Figure 4). With the exception of Tiler and Blend, whose performance needs are satisfied by singleton thread stages, all graph stages are instanced shader stages. All queues in the packet tracer graph are unordered.

A ray tracer performs two computationally expensive operations: ray-scene intersection and surface hit point shading. Considered separately, each of these operations is amenable to wide SIMD processing and exhibits favorable memory access characteristics. Because recursive rays are conditionally traced, SIMD utilization can drop severely if shading directly invokes intersection [Boulos et al. 2007].

Our implementation decouples these operations by making Intersect, Shadow Intersect, and Shade separate graph stages. Thus, each of the three operations executes efficiently on batches of inputs from their respective queues. To produce these batches of work, the ray tracer leverages the GRAMPS queue `push` operation. When shading yields too few secondary rays to form a complete packet, execution of Intersect (or Shadow Intersect) is delayed until more work is available. Similarly, if too few rays from Intersect need shading, they won’t be shaded until a sufficiently sized batch is available. This strategy could be extended further using more complex GRAMPS graphs. For example, separating Intersect into a full subgraph could allow for rays to be binned at individual BVH nodes during traversal.

Lastly, the ability to cast ray tracing as a graph with loops, rather than a feed-forward pipeline allows for an easy implementation of both max-depth ray termination and also ray tree attenuation termination by

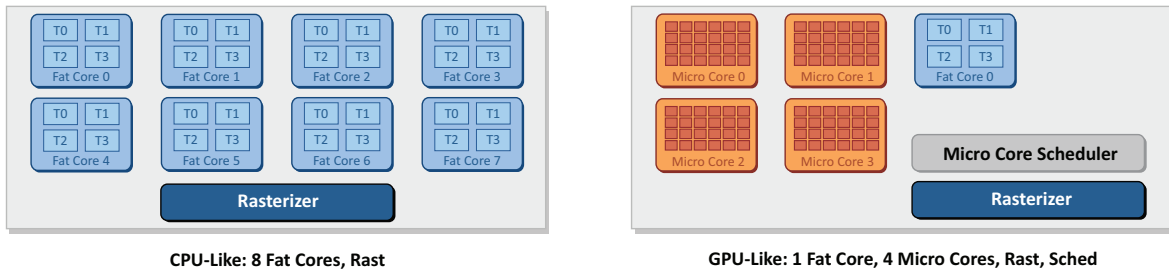


Fig. 5. The CPU-like and GPU-like simulator configurations: different mixtures of XPU fat (blue) and micro (orange) cores plus a fixed function rasterizer. Boxes within the cores represent hardware thread slots.

tracking depth/attenuation with each ray [Hall and Greenberg 1983]. While reflections to a fixed maximal depth could also be modeled with a statically unrolled pipeline, this is an awkward implementation strategy and does not permit ray tree attenuation.

4.3 Extended Direct3D

By formally constructing execution graphs that are decoupled from hardware, GRAMPS creates an opportunity for specialized pipelines. Our third renderer extends the Direct3D pipeline to form a new graph that adds ray traced effects (top of Figure 4, including the shaded portion). We insert two additional stages, Trace and PS2, between PS and OM and allow Extended Direct3D Pixel Shaders to **push** rays in addition to performing standard local shading. Trace performs packetized ray-scene intersection and pushes the results to a second shading stage (PS2). Like PS, PS2 is permitted to send its shaded output to OM, or generate additional rays for Trace (the Extended Direct3D graph contains a loop). We introduced PS2 as a distinct stage to retain the ability to specialize PS shading computations for the case of high coherence (fragments in a packet from Rast all originate from the same input triangle) and to separate tracing from shading as explained above.

There are two other important characteristics of our Extended Direct3D renderer. Our implementation uses a pre-initialized early-Z buffer from a prior Z-only pass to avoid unnecessary ray-scene queries. In addition, early-Z testing is required to generate correct images because pixel contributions from the PS2 stage can arrive out of triangle draw order (input to PS2 is an unordered **push** queue).

Note that while this example uses **push** only for the purposes of building ray and shading packets, other natural uses include handling fragment repacking when coherence patterns change, or as a mechanism for efficiently handling a constrained form of data amplification or compaction.

5. IMPLEMENTATION

We developed a machine simulator to serve as the basis for our preliminary evaluation of GRAMPS. The simulation environment provides two types of programmable cores (referred to as XPU) and a monolithic fixed-function rasterizer. All XPU cores have a MIPS64 architecture [MIPS Technologies Inc. 2005] extended with a vector instruction set. XPU *micro cores* are intended to resemble current GPU shader cores and execute efficiently under the load of many lightweight threads. Each micro core supports up to 24 independent hardware thread execution slots and features 16-wide SIMD vector units. XPU *fat cores* are general-purpose cores optimized for throughput. They are in-order, four-threaded processors with the same 16-wide vector

units as the micro cores. Both cores can execute one thread per clock. Compiled XPU shader program binaries utilize vector instructions to simultaneously process multiples of 16 elements within a single thread. Thus, each XPU micro core is capable of hardware-interleaved execution of 384 shader instances.

We run our simulation environment in two different hardware configurations (Figure 5). The *GPU-like* configuration contains one fat core, four micro cores, and a fixed-function rasterizer and is envisioned as an evolution of current GPUs. The *CPU-like* configuration consists of the rasterizer plus eight fat cores, mimicking a more general purpose many-core implementation. This choice of machine configurations allows us to explore two GRAMPS scheduler implementations employing different levels of hardware support.

5.1 Scheduling

The goal of the GRAMPS scheduler is to maximize machine utilization. Specifically, it seeks to synthesize at run-time what streaming systems arrange during up-front compilation: aggregated batches of parallel work with strong data locality. Recall that the queues of a GRAMPS computation graph are intended to delineate such coherency groupings. The GRAMPS scheduler then balances the need to accumulate enough work to fill all available cores against the storage overheads of piling up undispached packets and the computational overhead of making frequent scheduling decisions. GRAMPS’s generality creates a significant scheduling disadvantage compared to a native GPU or other single pipeline-specific scheduler: GRAMPS lacks semantic knowledge of—and any scheduling heuristics based on—stage internals and the data types passed between them. The GRAMPS abstractions are designed to give an implementer two primary hints to partially compensate: the topology of the execution graph, and the capacity of each queue.

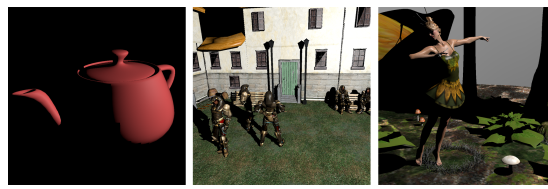
Our current scheduling algorithm assigns each stage a static priority based upon its proximity in the graph to sink nodes (stages with no output queues) and distance from source nodes (stages with no input queues). In a pipeline or DAG, this gives the start(s) lowest weight and the end(s) highest weight which predisposes the scheduler towards draining work out of the system and keeping queue depths small. In graphs with cycles, the top node (stage closest to an input) has higher priority than the bottom to prevent the backwards looping queue starving and growing deep. Additionally, the scheduler maintains an ‘inspect’ bitvector containing a field for each graph stage. A stage’s inspectable bit is set whenever a new input packet is available or output packet is consumed (in case it was blocked on a full output queue). Its bit is cleared whenever the scheduler next inspects the stage and either dispatches all available input or determines the stage is not truly runnable.

Our scheduler is organized hierarchically in ‘tiers’. The top tier (Tier-N) has system-wide responsibilities for notifying idle cores and fixed-function units when they should look for work. The cores themselves then handle the lower levels of scheduling.

5.1.1 Fat Core Scheduling. Since we intended fat cores to resemble CPUs and be suitable for arbitrary threads, the GRAMPS fat-core scheduling logic is implemented directly in software. It is organized as a fast, simple Tier-0 that manages a single thread slot and a more sophisticated Tier-1 that is shared per-core.

Tier-1 updates and maintains a prioritized work list of runnable instances based upon the inspect bitvector. It also groups shader instances for launch and issues the implicit `reserve` and `commit` operations on their behalf. Tier-1 runs asynchronously at a parameterized period (one million cycles in our experiments). However, if a Tier-0 scheduler finds the list of runnable instances empty (there is no work to dispatch), Tier-0 will invoke Tier-1 before idling its thread slot.

Tier-0 carries out the work of loading, unloading, and preempting instances on individual thread slots. It makes no “scheduling” decisions other than comparing the current thread’s priority to the front of the work



	Triangles	Fragments/Tri
Teapot	6,230	67.1
Courtyard	31,375	145.8
Fairy	174,117	36.8

Table I. The Teapot, Courtyard, and Fairy test scenes. Courtyard uses character models from Unreal Tournament 3. Fairy is a complex scene designed to stress modern ray tracers.

list at potential preemption points. For thread stages, preemption points include queue manipulations and terminations. For shader stages, preemption is possible only between instance invocations.

5.1.2 Micro Core Scheduling. In the GPU-like configuration, all shader work is run on micro cores. Similar to current GPU designs, micro cores rely on a hardware-based scheduling unit to manage their numerous simple thread contexts (see Figure 5). This unit is functionally similar to combined fat-core Tier-1 and Tier-0’s with two significant differences: a single hardware Tier-1 is shared across all micro cores, and it is invoked on demand at every shader instance termination rather than asynchronously.

When data is committed to shader queues, the micro-core scheduler identifies (in order of stage priority) input queues with sufficient work, then pre-reserves space in the corresponding stage’s input and output queues. It associates this data with new shader instances and assigns the instances to the first unused thread slot in the least-loaded micro core. When shader instances complete, the scheduler commits their input and output data, then attempts to schedule a new shader instances to fill the available thread slot. The micro-core scheduler also takes care of coalescing elements generated via `push` into packets.

6. EVALUATION

We conducted a preliminary evaluation of GRAMPS with respect to our stated design goals. We exercised the CPU- and GPU-like GRAMPS implementations using the three rendering pipelines described in Section 4. Each pipeline was used to render the three scenes described in Table I at 1024×1024 resolution. The ray tracer casts shadow rays and one bounce of reflection rays off all surfaces. Extended Direct3D casts only shadow rays. The scenes vary in both overall complexity and distribution of triangle size, requiring GRAMPS to dynamically balance load across graph stages.

As explained in Section 5.1, our primary focus was our implementations’ ability to find parallelism and to manage queues (especially in the context of loops and the use of `push`). Specifically, we measure the extent to which GRAMPS keeps core thread execution slots occupied with active threads, and the depth of queues during graph execution.

Our measurements are conducted with two simplifying assumptions: First, although our implementations seek to minimize the frequency at which scheduling operations occur, we assign no cost to the execution of the GRAMPS scheduler or for possible contention in access to shared queues. Second, we incorporate only a simple memory system model—a fixed access time of four cycles for fat cores and 100 cycles for micro cores. Given these assumptions, we use thread execution-slot occupancy as our performance metric

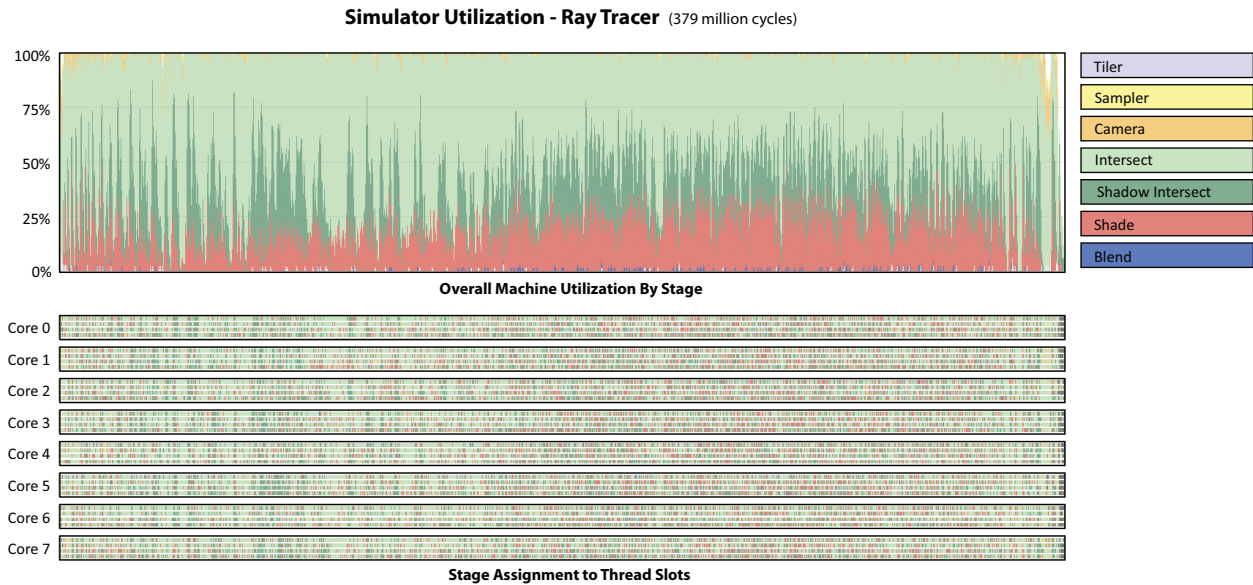


Fig. 6. Ray tracing the Teapot scene on the CPU-like simulator. As work is enqueued, the GRAMPS scheduler dynamically assigns stage instances to hardware thread slots as shown on the bottom.

rather than ALU utilization (ALUs may be underutilized due to memory stalls or low SIMD efficiency even if a slot is filled). Slot occupancy is convenient because it directly reflects the scheduler’s ability to recognize opportunities for parallelism. At the same time, it is less dependent on the degree of optimization of thread/shader programs—which is not a need unique to GRAMPS nor a focus in our prototype system. We agree that a future detailed evaluation of a highly optimized GRAMPS implementation would require these factors to be better approximated.

Figure 6 is a visualization from the simulator while running the CPU-like configuration of the ray tracer. As expected, the cost of the computation is dominated by ray-scene intersection and shading. Note that the mapping of instances onto cores is highly dynamic as data flows through the queues. Table II summarizes the overall simulation statistics. Note that on GPU-like configurations, we focus on the occupancy of the micro cores that run shader work (the fat core in the GPU-like configuration is rarely used as our graphs perform a majority of computation in shaders).

Both GRAMPS implementations maintained high thread-slot occupancy with all of the renderers. With the exception of rendering the Fairy using Direct3D, the GRAMPS scheduler produced occupancy above 87% (small triangles in the Fairy scene bottle-neck the pipeline in RO limiting available parallelism—see GPU-like fat core occupancy in Table II).

Our emulations maintained high occupancy while keeping worst-case queue footprint low. In all experiments queue sizes remained small enough to be contained within the on-chip memories of modern processors. The ray tracer, despite a graph loop for reflection rays and heavy use of `push`, had by far the smallest queue footprint. This was the direct result of using entirely unordered queues. With ordered queues, when instances complete out of order, as happens from time to time, GRAMPS cannot make their output available downstream or reclaim it until the missing stragglers arrive.

		CPU-like Configuration		GPU-like Configuration		
		Fat Core	Peak Queue	Fat Core	Micro Core	Peak Queue
		Occup (%)	Size (KB)	Occup (%)	Occup (%)	Size (KB)
Teapot	D3D	87.8	510	13.0	95.9	1,329
	Ext. D3D	90.2	582	0.5	98.8	1,264
	Ray Tracer	99.8	156	3.2	99.9	392
Courtyard	D3D	88.5	544	9.2	95.0	1,301
	Ext. D3D	94.2	586	0.2	99.8	1,272
	Ray Tracer	99.9	176	1.2	99.9	456
Fairy	D3D	77.2	561	20.5	81.5	1,423
	Ext. D3D	92.0	605	0.8	99.8	1,195
	Ray Tracer	100.0	205	0.8	99.9	537

Table II. Simulation results: Core thread-slot occupancy and peak memory footprint of all graph queues.

While the GRAMPS graphs we present and evaluate perform well, our experiences proved that choosing a good graph for a rendering pipeline can make a major difference. The GRAMPS concepts permit graphs that do not run well, and even good graphs profit from considerable tuning. For example, our initial Direct3D graph—which used a single shader stage to handle both PS and OM—exhibited large queue memory consumption.

Although our first Direct3D graph used a queue set to respect OM ordering requirements while still enabling parallel processing of distinct screen space image tiles, this formulation caused all PS work for a single tile to be serialized. Thus, the graph suffered from load imbalance (and corresponding queue backup) when one tile—and thus one subqueue—had a disproportionate number of fragments. Separating PS and OM into unique graph stages and connecting these stages using a queue set allowed shading of all fragments— independent of screen location—to be performed in parallel. This modification reduced queue footprints by over two orders of magnitude. Similarly, in the ray tracer, limiting the maximum depth of the queue between the sampler and the camera while leaving the others effectively unbounded reduced the overall footprint by more than an order of magnitude.

In the same vein, although the general graph structure is the same across our two simulation configurations, we made slight tuning customizations as a function of how many machine thread slots were available. In the GPU-like configuration of Direct3D/Extended Direct3D, we increased the number of OM input subqueues to enable additional parallelism. We also set the capacities on several critical Direct3D queues and, as mentioned above, the ray tracer’s sample queue based on the maximum number of machine threads.

7. FUTURE WORK

This paper introduces the GRAMPS programming model: an abstraction for expressing advanced rendering pipelines. It articulates three design goals for GRAMPS: high performance, large application scope, and optimized implementation flexibility.

We have demonstrated that pipelines with variable numbers of outputs and with cycles can be efficiently implemented using GRAMPS. These abstractions are very powerful and we used them to build and combine rasterization and ray-tracing based renderers.

Our prototype implementation of GRAMPS suggests that implementations can be optimized for different

hardware configurations. The performance and resource consumption of our prototypes is encouraging. With well designed execution graphs, the hardware utilization is high and the queue storage overhead is low.

We restate that GRAMPS itself does not provide a graphics pipeline abstraction or a specific machine organization. Rather, it is a programming model that permits a large class of renderers to run on a variety of high-performance many-core chips. In this context there are three clear avenues for future work. First, the GRAMPS abstractions must continue to be refined and the graph scheduling problem should be studied in detail. For example, we have no experience with scheduling complicated graph structures such as those containing nested loops. Second, evolving a modern graphics pipeline abstraction like Direct3D to incorporate software-defined stages and data flows, without sacrificing its graphics-domain specific benefits, (e.g., namely convenience and abstraction portability) remains unsolved. Last, the ideas in GRAMPS can be used to build applications beyond rendering. GPU and CPU architects are struggling to find a point of convergence that best combines the traits of both architectures. GRAMPS seems to provide a natural example for informing the design of such systems.

8. ACKNOWLEDGEMENTS

The Fairy Forest scene is from the Utah 3D Animation Repository and the Courtyard scene is courtesy of University of Texas at Austin’s Computer Graphics Group. This research was supported by the Stanford Pervasive Parallelism Lab, the Department of the Army Research (grant W911NF-07-2-0027), and a visual computing research grant from Intel Corporation. Additionally, Jeremy Sugerman was supported by the Rambus Stanford Graduate Fellowship, Kayvon Fatahalian by the Intel PhD Fellowship Program, and Solomon Boulos by an NSF Graduate Research Fellowship.

REFERENCES

- AMD. 2008a. AMD Radeon HD 4800 Product Documentation. <http://ati.amd.com/products/radeonhd4800>.
- AMD. 2008b. ATI Stream Computing Website. <http://ati.amd.com/technology/streamcomputing/>.
- BAVOIL, L., CALLAHAN, S. P., LEFOHN, A., COMBA, J. L. D., AND SILVA, C. T. 2007. Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*. ACM, New York, NY, USA, 97–104.
- BLYTHE, D. 2006. The Direct3D 10 system. *ACM Transactions on Graphics* 25, 3 (July), 724–734.
- BOULOS, S., EDWARDS, D., LACEWELL, J., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2007. Packet-based Whitted and distribution ray tracing. *Proceedings of Graphics Interface 2007*, 177–184.
- BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics* 23, 3, 777–786.
- CHEN, J., GORDON, M. I., THIES, W., ZWICKER, M., PULLI, K., AND DURAND, F. 2005. A reconfigurable architecture for load-balanced rendering. In *Workshop on Graphics Hardware*. ACM, New York, NY, USA, 71–80.
- DALLY, W. J., HANRAHAN, P., EREZ, M., KNIGHT, T. J., LABONTE, F., A., J.-H., JAYASENA, N., KAPASI, U. J., DAS, A., GUMMARAJU, J., AND BUCK, I. 2003. Merrimac: Supercomputing with streams. In *SC’03*. Phoenix, Arizona.
- DAS, A., DALLY, W. J., AND MATTSON, P. 2006. Compiling for stream processing. In *Proceedings of PACT 2006*. 33–42.
- FOLEY, T. AND SUGERMAN, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Workshop on Graphics Hardware*. ACM, New York, NY, USA, 15–22.
- HALL, R. AND GREENBERG, D. 1983. A testbed for realistic image synthesis. *IEEE Comput. Graph. Appl.* 3, 8, 10–20.
- HASSELGREN, J. AND AKENINE-MÖLLER, T. 2007. PCU: the programmable culling unit. *ACM Transactions on Graphics* 26, 3, 92.
- HORN, D., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-D Tree GPU Raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*. ACM, New York, NY, USA.
- INTEL. 2008. Intel thread building blocks product documentation. <http://www.intel.com/cd/software/products/asm-na/eng/294797.htm>.

- KAPASI, U., DALLY, W. J., RIXNER, S., OWENS, J. D., AND KHAILANY, B. 2002. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*. 282–288.
- KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. 2005. Niagara: a 32-way multithreaded SPARC processor. *IEEE Micro* 25, 2, 21–29.
- KUMAR, S., HUGHES, C., AND NGUYEN, A. 2007. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. *Proceedings of the 34th annual international conference on computer architecture*, 162–173.
- LINDHOLM, E., NICKOLLS, J., OBERMANAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A Graphics and Computing Architecture. *IEEE Micro* 28, 2, 39–55.
- MCCOOL, M., TOIT, S. D., POPA, T., CHAN, B., AND MOULE, K. 2004. Shader algebra. In *Proceedings of SIGGRAPH 2004*. ACM, New York, NY, USA, 787–795.
- MIPS TECHNOLOGIES INC. 2005. MIPS64 architecture. <http://mips.com/products/architectures/mips64/>.
- NVIDIA. 2007. NVIDIA CUDA programming guide. http://developer.download.nvidia.com/compute/cuda/1.1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.
- OWENS, J. D., KHAILANY, B., TOWLES, B., AND DALLY, W. J. 2002. Comparing Reyes and OpenGL on a stream architecture. In *Workshop on Graphics Hardware*. 47–56.
- PHAM, D., ASANO, S., BOLLIGER, M., DAY, M., HOFSTEE, H., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI, Y., ET AL. 2005. The design and implementation of a first-generation CELL processor. *ISSCC. 2005 IEEE International*, 184–186.
- PURCELL, T. J. 2004. Ray Tracing on a Stream Processor. Ph.D. thesis, Stanford University.
- SEGAL, M. AND AKELEY, K. 2006. The OpenGL 2.1 specification. <http://www.opengl.org/registry/doc/glspec21.20061201.pdf>.
- SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics* 27, 3.
- TARDITI, D., PURI, S., AND OGLESBY, J. 2006. Accelerator: using data parallelism to program GPUs for general-purpose uses. *SIGOPS Operating Systems Review* 40, 5, 325–335.
- THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. 2002. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*. Grenoble, France.