Scheduling the Graphics Pipeline

Jonathan Ragan-Kelley, MIT CSAIL
9 August 2011
This talk

How to think about scheduling GPU-style pipelines
Four constraints which drive scheduling decisions

Examples of these concepts in real GPU designs

Goals
Know why GPUs, APIs impose the constraints they do.
Develop intuition for what they can do well.
Understand key patterns for building your own pipelines.
This talk

How to think about scheduling GPU-style pipelines
Four constraints which drive scheduling decisions

Examples of these concepts in real GPU designs

Goals
Know why GPUs, APIs impose the constraints they do.
Develop intuition for what they can do well.
Understand key patterns for building your own pipelines.
First, some definitions

**Scheduling** [n.]:
A single, discrete unit of work.

**Task** [n.]:
A single, discrete unit of work.
First, some definitions

Scheduling [n.]:
Assigning computations and data to resources in space and time.

Task [n.]:
A single, discrete unit of work.
First, some definitions

**Scheduling** [n.]: Assigning **computations** and **data** to resources in **space** and **time**.

**Task** [n.]: A single, discrete unit of work.
The workload: Direct3D

- IA
- VS
- PA
- HS
- Tess
- DS
- PA
- GS
- Rast
- PS
- Blend
The workload: Direct3D
The workload: Direct3D

Beyond Programmable Shading 2011
The workload: Direct3D

IA
VS
PA
HS
Tess
DS
PA
GS
Rast
PS
Blend

data flow
The workload: Direct3D

 Logical pipeline
- Fixed-function stage
- Programmable stage

Data flow:
- IA
- VS
- PA
- HS
- Tess
- DS
- PA
- GS
- Rast
- PS
- Blend

Beyond Programmable Shading 2011
The machine: a modern GPU
Scheduling a draw call as a series of tasks

- Shader Core
- Shader Core
- Shader Core
- Shader Core
- Input Assembler
- Primitive Assembler
- Rasterizer
- Output Blend

(time)
Scheduling a draw call as a series of tasks

Beyond Programmable Shading 2011
Scheduling a draw call as a series of tasks
Scheduling a draw call as a series of tasks

Shader Core → Shader Core → Shader Core → Shader Core → Input Assembler → Primitive Assembler → Rasterizer → Output Blend

IA → VS → PA
Scheduling a draw call as a series of tasks

- Shader Core
- Shader Core
- Shader Core
- Shader Core
- Input Assembler
- Primitive Assembler
- Rasterizer
- Output Blend

VS -> IA -> PA -> Rast

time
Scheduling a draw call as a series of tasks
Scheduling a draw call as a series of tasks

- Shader Core
- Shader Core
- Shader Core
- Shader Core
- Input Assembler
- Primitive Assembler
- Rasterizer
- Output Blend

- IA
- VS
- PA
- Rast
- PS
- PS
- PS
- Blend
- Blend
- Blend

Beyond Programmable Shading 2011
An efficient schedule keeps hardware busy

Beyond Programmable Shading 2011
Choosing which tasks to run when (and where)

**Resource constraints**
Tasks can only execute when there are sufficient resources for their computation and their data.

**Coherence**
Control coherence is essential to shader core efficiency. Data coherence is essential to memory and communication efficiency.

**Load balance**
Irregularity in execution time create bubbles in the pipeline schedule.

**Ordering**
Graphics APIs define strict ordering semantics, which restrict possible schedules.
Resource constraints limit scheduling options
Resource constraints limit scheduling options

Beyond Programmable Shading 2011
Resource constraints limit scheduling options

Beyond Programmable Shading 2011
Resource constraints limit scheduling options

Beyond Programmable Shading 2011
Resource constraints limit scheduling options

Shader Core  Shader Core  Shader Core  Shader Core  Input Assembler  Primitive Assembler  Rasterizer  Output Blend

time

PS  PS  ???  PS

Deadlock

Beyond Programmable Shading 2011
Resource constraints limit scheduling options

**Key concept:** Preallocation of resources helps guarantee forward progress.
Coherence is a balancing act

Intrinsic tension between:

**Horizontal** (control, fetch) coherence and **Vertical** (producer-consumer) locality.

**Locality** and **Load Balance**.
Graphics workloads are irregular
Graphics workloads are irregular
Graphics workloads are irregular
Graphics workloads are irregular
Graphics workloads are irregular

But: Shaders are optimized for regular, self-similar work. Imbalanced work creates bubbles in the task schedule.
Graphics workloads are irregular

But: Shaders are optimized for regular, self-similar work. Imbalanced work creates bubbles in the task schedule.

Solution:
Dynamically generating and aggregating tasks isolates irregularity and recaptures coherence. Redistributing tasks restores load balance.
Redistribution after irregular amplification

Beyond Programmable Shading 2011
Redistribution after irregular amplification

Shader Core  Shader Core  Shader Core  Shader Core  Input Assembler  Primitive Assembler  Rasterizer  Output Blend

time

PS  PS  PS

VS

IA

PA

Rast

Blend

Blend

Blend

Beyond Programmable Shading 2011
Redistribution after irregular amplification

Key concept: Managing irregularity by dynamically **generating**, **aggregating**, and **redistributing** tasks
Ordering

Rule:
All framebuffer updates must appear as though all triangles were drawn in strict sequential order.
Rule:
All framebuffer updates must appear as though all triangles were drawn in strict sequential order.

Key concept: Carefully structuring task redistribution to maintain API ordering.
Building a real pipeline
Static tile scheduling

The simplest thing that could possibly work.

Multiple cores:
  1 front-end
  $n$ back-end

Exemplar:
ARM Mali 400
Static tile scheduling

Exemplar: ARM Mali 400
Static tile scheduling

Exemplar: ARM Mali 400
Static tile scheduling

Exemplar: ARM Mali 400
Static tile scheduling

Exemplar:

ARM Mali 400
Static tile scheduling

Exemplar: ARM Mali 400
Static tile scheduling

Locality
captured within tiles

Resource
constraints
static = simple

Ordering
single front-end,
sequential processing
within each tile

Exemplar:
ARM Mali 400
The problem: load imbalance

- only one *task creation* point.
- no *dynamic task redistribution*.

Exemplar: *ARM Mali 400*
Static tile scheduling

The problem: load imbalance

- only one *task creation* point.
- no *dynamic task redistribution*.

Exemplar: ARM Mali 400

Beyond Programmable Shading 2011
The problem: load imbalance

only one task creation point.

no dynamic task redistribution.

Exemplar: ARM Mali 400
Sort-last fragment shading

Exemplars:
NVIDIA G80, ATI RV770
Sort-last fragment shading

Redistribution restores fragment load balance.

But how can we maintain order?

Exemplars:
NVIDIA G80, ATI RV770
Sort-last fragment shading

Exemplars: NVIDIA G80, ATI RV770

Preallocate outputs in FIFO order
Sort-last fragment shading

Exemplars:
- NVIDIA G80
- ATI RV770

Complete shading asynchronously
Sort-last fragment shading

Exemplars:
NVIDIA G80, ATI RV770
Unified shaders

Solve load balance by time-multiplexing different stages onto shared processors according to load

Exemplars:
NVIDIA G80, ATI RV770

Beyond Programmable Shading 2011
Unified Shaders: time-multiplexing cores

Exemplars:
NVIDIA G80, ATI RV770
Unified Shaders: time-multiplexing cores

Exemplars:
NVIDIA G80, ATI RV770
Prioritizing the logical pipeline

- IA
- VS
- PA
- Rast
- PS
- Blend
Prioritizing the logical pipeline

- **IA** (5)
- **VS** (4)
- **PA** (3)
- **Rast** (2)
- **PS** (1)
- **Blend** (0)
Prioritizing the logical pipeline

IA 5
VS 4
PA 3
Rast 2
PS 1
Blend 0

Beyond Programmable Shading 2011
Prioritizing the logical pipeline

fixed-size queue storage

IA 5
VS 4
PA 3
Rast 2
PS 1
Blend 0

Beyond Programmable Shading 2011
Scheduling the pipeline

Beyond Programmable Shading 2011
Scheduling the pipeline
Scheduling the pipeline

Beyond Programmable Shading 2011
Scheduling the pipeline

- **Shaders Core**
  - **VS**
  - **PS**

- **IA**
  - **VS**
  - **PA**
  - **Rast**
  - **PS**
  - **Blend**

- Time

**High priority, but stalled on output**

**Lower priority, but ready to run**

Beyond Programmable Shading 2011
Scheduling the pipeline

Beyond Programmable Shading 2011
Queue sizes and backpressure provide a natural knob for balancing horizontal batch coherence and producer-consumer locality.
A real computational graphics pipeline: OptiX
A real computational graphics pipeline: OptiX

Pipeline abstraction for ray tracing
A real computational graphics pipeline: OptiX

Pipeline abstraction for ray tracing

Application functionality in shader-style programs

Intersecting primitives

Shading surfaces, firing rays

Host

Entry Points

Ray Generation Program
Exception Program

Buffers

Texture Samplers
Variables

Trace

Intersection Program
Any Hit Program
Selector Visit Program

Ray Shading

Closest Hit Program
Miss Program

Beyond Programmable Shading 2011
A real computational graphics pipeline: OptiX

Pipeline abstraction for ray tracing
Application functionality in shader-style programs
Pipeline structure from API

- Host
- Entry Points: Ray Generation Program, Exception Program
- Trace
- Ray Shading: Closest Hit Program, Miss Program
- Buffers
- Texture Samplers
- Variables
- Traversal: Intersection Program, Any Hit Program, Selector Visit Program

Intersecting primitives
Shading surfaces, firing rays
Traversing, acceleration structures
Order of execution
Resource management

Beyond Programmable Shading 2011
Issues in scheduling a ray tracer

Breadth first or depth first traversal?

Wide execution aggregates more potentially coherent work. Depth-first execution reduces footprint needed.
Issues in scheduling a ray tracer

Breadth first or depth first traversal?
Wide execution aggregates more potentially coherent work. Depth-first execution reduces footprint needed.

OptiX: as wide as the the machine, but no wider.
Issues in scheduling a ray tracer

Breadth first or depth first traversal?
Wide execution aggregates more potentially coherent work. Depth-first execution reduces footprint needed.

OptiX: as wide as the the machine, but no wider.

Extracting SIMD coherence
Shader core requires SIMD batches for efficiency. Rays may diverge.
Ray tracing on a SIMD machine

Scalar ray tracing

traverse

intersect?

traverse

intersect!
Ray tracing on a SIMD machine

Scalar ray tracing

Packet tracing
Breaking packets: SIMT ray tracing

[Aila & Laine 2009]

Allow *data* divergence
Different rays traverse, intersect different parts of the scene

Maintain *control* (SIMD) coherence
All rays in bundle either *traverse* or *intersect* together
Breaking packets: SIMT ray tracing

[Alia & Laine 2009]

Allow *data* divergence
Different rays traverse, intersect different parts of the scene

Maintain *control* (SIMD) coherence
All rays in bundle either *traverse* or *intersect* together

```cpp
while(state != done) {
    if (state == traverse) traverse();
    if (state == intersect) intersect();
}
```
A pipeline program as a state machine

while(myState != DONE) {
    nextState = scheduler();
    if (myState == nextState)
        switch(myState) {
            case 0: myState = traverse(); break;
            case 1: myState = intersector1(); break;
            case 2: myState = intersector2(); break;
            case 3: myState = shader1(); break;
            case 4: myState = shader2(); break;
            ...
        }
}
A pipeline program as a state machine

while(myState != DONE) {
    nextState = scheduler();
    if (myState == nextState)
        switch(myState) {
            case 0: myState = traverse(); break;
            case 1: myState = intersector1(); break;
            case 2: myState = intersector2(); break;
            case 3: myState = shader1(); break;
            case 4: myState = shader2(); break;
            ...
        }
}
A pipeline program as a state machine

```c
while(myState != DONE) {
    nextState = scheduler();
    if (myState == nextState)
        switch(myState) {
        case 0: myState = traverse(); break;
        case 1: myState = intersector1(); break;
        case 2: myState = intersector2(); break;
        case 3: myState = shader1(); break;
        case 4: myState = shader2(); break;
        ...
        }
}
```
A pipeline program as a state machine

while(myState != DONE) {
    nextState = scheduler();
    if (myState == nextState)
        switch(myState) {
            case 0: myState = traverse(); break;
            case 1: myState = intersector1(); break;
            case 2: myState = intersector2(); break;
            case 3: myState = shader1(); break;
            case 4: myState = shader2(); break;
            ...
        }
}
A pipeline program as a state machine

while(myState != DONE) {
  nextState = scheduler();
  if (myState == nextState)
    switch(myState) {
      case 0: myState = traverse(); break;
      case 1: myState = intersector1(); break;
      case 2: myState = intersector2(); break;
      case 3: myState = shader1(); break;
      case 4: myState = shader2(); break;
      ...
    }
}
A pipeline program as a state machine

while(myState != DONE) {
    nextState = scheduler();
    if (myState == nextState)
        switch(myState) {
        case 0: myState = traverse(); break;
        case 1: myState = intersector1(); break;
        case 2: myState = intersector2(); break;
        case 3: myState = shader1(); break;
        case 4: myState = shader2(); break;
        ...
        }
}
Summary
Key concepts

Think of scheduling the pipeline as mapping tasks onto cores.

Preallocate resources before launching a task.
Preallocation helps ensure forward progress and prevent deadlock.

Graphics is irregular.
Dynamically generating, aggregating and redistributing tasks at irregular amplification points regains coherence and load balance.

Order matters.
Carefully structure task redistribution to maintain ordering.
Why don’t we have dynamic resource allocation? e.g. recursion, malloc() in shaders

Static preallocation of resources guarantees forward progress.

Tasks which outgrow available resources can stall, causing **deadlock**.
Geometry Shaders are slow because they allow dynamic amplification in shaders.

Pick your poison:

**Always stream through DRAM.**

*exemplar: ATI R600*
Smooth falloff for large amplification, but very slow for small amplification (DRAM latency).

**Scale down parallelism to fit.**

*exemplar: NVIDIA G80*
Fast for small amplification, poor shader throughput (no parallelism) for large amplification.
Key concepts

Think of **scheduling the pipeline as mapping tasks onto cores.**

**Preallocate resources before launching a task.**
Preallocation helps ensure forward progress and prevent deadlock.

**Graphics is irregular.**
Dynamically **generating, aggregating** and **redistributing tasks** at irregular amplification points regains **coherence** and **load balance.**

**Order matters.**
Carefully structure **task redistribution** to maintain ordering.
Why isn’t rasterization programmable?

Yes, partly because it is computationally intensive, but also:

It is *highly irregular*.

It must generate and aggregate *regular output*.

It must integrate with an *order-preserving task redistribution mechanism*.
Key concepts

Think of scheduling the pipeline as mapping tasks onto cores.

Preallocate resources before launching a task.
Preallocation helps ensure forward progress and prevent deadlock.

Graphics is irregular.
Dynamically generating, aggregating and redistributing tasks at irregular amplification points regains coherence and load balance.

Order matters.
Carefully structure task redistribution to maintain ordering.
Questions for the future

Can we relax the strict ordering requirements?

Can you build a generic scheduler for application-defined pipelines?

What application-specific information would a generic scheduler need to work well?
Starting points to learn more

The next step: parallel primitive processing


Scheduling cyclic graphs, in software, on current GPUs

Details of the ARM Mali design
Thank you

Special thanks:
Tim Purcell, Steve Molnar, Henry Moreton, Steve Parker, Austin Robison - NVIDIA
Jeremy Sugerman - Stanford
Mike Houston - AMD
Mike Doggett - Lund University
Tom Olson - ARM