Keeping Many Cores Busy: Scheduling the Graphics Pipeline

Jonathan Ragan-Kelley, MIT CSAIL

29 July 2010
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, a definition

Scheduling [n.]:

Beyond Programmable Shading 2010
First, a definition

Scheduling [n.]: Assigning computations and data to resources in space and time.
The workload: Direct3D
The workload: Direct3D
The workload: Direct3D

IA
VS
PA
HS
Tess
DS
PA
GS
Rast
PS
Blend

data flow

Beyond Programmable Shading 2010
Beyond Programmable Shading 2010

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:
- Beyond Programmable Shading 2010
The machine: a modern GPU
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

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

Shading Core
Shading Core
Shading Core
Shading Core
Input Assembler
Primitive Assembler
Rasterizer
Output Blend

IA
VS
PA
Rast

Beyond Programmable Shading 2010
Scheduling a draw call as a series of tasks

Beyond Programmable Shading 2010
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
- Blend
- Blend
- Blend

Beyond Programmable Shading 2010
An efficient schedule keeps hardware busy

```
<table>
<thead>
<tr>
<th>Shader Core</th>
<th>Shader Core</th>
<th>Shader Core</th>
<th>Shader Core</th>
<th>Input Assembler</th>
<th>Primitive Assembler</th>
<th>Rasterizer</th>
<th>Output Blend</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS</td>
<td>PS</td>
<td>PS</td>
<td>VS</td>
<td>IA</td>
<td>PA</td>
<td>Rast</td>
<td>Blend</td>
</tr>
<tr>
<td>VS</td>
<td>PS</td>
<td>PS</td>
<td>VS</td>
<td>IA</td>
<td>PA</td>
<td>Rast</td>
<td>Blend</td>
</tr>
<tr>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>IA</td>
<td>PA</td>
<td>Rast</td>
<td>Blend</td>
</tr>
<tr>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>IA</td>
<td>PA</td>
<td>Rast</td>
<td>Blend</td>
</tr>
<tr>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>IA</td>
<td>PA</td>
<td>Rast</td>
<td>Blend</td>
</tr>
<tr>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>IA</td>
<td>PA</td>
<td>Rast</td>
<td>Blend</td>
</tr>
<tr>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>IA</td>
<td>PA</td>
<td>Rast</td>
<td>Blend</td>
</tr>
<tr>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>PS</td>
<td>IA</td>
<td>PA</td>
<td>Rast</td>
<td>Blend</td>
</tr>
</tbody>
</table>
```
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

Beyond Programmable Shading 2010
Resource constraints limit scheduling options
Resource constraints limit scheduling options

Beyond Programmable Shading 2010
Resource constraints limit scheduling options

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

.time

- PS
- PS
- ???
- PS
- VS
- ???
Resource constraints limit scheduling options

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

Right arrow: time

PS | PS | ??? | PS

Deadlock
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

Beyond Programmable Shading 2010
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 2010
Redistribution after irregular amplification
Key concept: Managing irregularity by dynamically generating, aggregating, and redistributing tasks.
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

Beyond Programmable Shading 2010
Static tile scheduling

Exemplar: ARM Mali 400
Static tile scheduling

Exemplar:
ARM Mali 400
Static tile scheduling

Vertex

Pixel

Pixel

Pixel

Exemplar: ARM Mali 400

Beyond Programmable Shading 2010
Static tile scheduling

Exemplar: ARM Mali 400

Beyond Programmable Shading 2010
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 \textit{task creation} point.

no \textit{dynamic task redistribution}.

Exemplar: ARM Mali 400
The problem: load imbalance

only one task creation point.

no dynamic task redistribution.

Exemplar: ARM Mali 400
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

Vertex → Rasterizer → Pixel → Pixel → Pixel → Pixel
Sort-last fragment shading

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

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

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

Exemplars:
NVIDIA G80, ATI RV770
Prioritizing the logical pipeline

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

Beyond Programmable Shading 2010
Prioritizing the logical pipeline

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

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

priority
Prioritizing the logical pipeline

fixed-size queue storage

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

Beyond Programmable Shading 2010
Scheduling the pipeline

- Shader Core
- Shader Core
- IA
- VS
- PA
- Rast
- PS
- Blend

Beyond Programmable Shading 2010
Scheduling the pipeline

Beyond Programmable Shading 2010
Scheduling the pipeline

- Shader Core
- Shader Core
- VS
- VS
- IA
- VS
- PA
- Rast
- PS
- Blend

 Beyond Programmable Shading 2010
Beyond Programmable Shading 2010

Scheduling the pipeline

- High priority, but stalled on output
- Lower priority, but ready to run

IA → VS → PS → Rast → Blend → Shader Core

Shader Core

VS

PS

Vienna
Scheduling the pipeline

Beyond Programmable Shading 2010
Scheduling the pipeline

Queue sizes and backpressure provide a natural knob for balancing horizontal batch coherence and producer-consumer locality.
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
Some Lessons
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.
\textit{exemplar: ATI R600}
Smooth falloff for large amplification, but very slow for small amplification (DRAM latency).

Scale down parallelism to fit.
\textit{exemplar: NVIDIA G80}
Fast for small amplification, poor shader throughput (no parallelism) for large amplification.
Why isn’t rasterization programmable?

(Yes, it is *computationally intensive.*)

It is *highly irregular.*

It must generate and aggregate *regular output.*

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