Beyond Programmable Shading 2011

The Real-Time Rendering Architectures talk shows how shaders can use large, coherent batches of work to achieve high throughput. I'm going to pop up a level to some of the essentials of how graphics pipelines keep these shaders fed.
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*.

Beyond Programmable Shading 2011

In this talk, I want to set up a simple framework for how to think about scheduling GPU-style pipelines. As part of this, I’m going to focus on 4 constraints which drive how we can schedule. I’m then going to show some examples of these concepts in a few real GPU architectures.

My goal is to help elucidate these so:

- you can better understand why GPUs and Graphics APIs impose the constraints they do.
- develop intuition for what GPUs can do well.
- and understand these patterns for one day building your own pipelines, in a world of computational graphics.

- This last point is really the biggest reason why you’ll all need to understand this as application developers. Mike’s talk about shader cores showed how usermode work creation is starting to appear in the fusion and sandybridge architectures. This is just the first step towards much more software control over rendering. It’s easy to take for granted many of the complexities hidden inside the graphics pipeline—the ways in which its doing a great deal of work for you to make your code run well. If you want more control to reach inside and take control over more and more of the graphics stack, you need to understand the responsibility that comes with that. It’s important to learn the lessons of the past, so you can push off in new directions rather than struggling with past mistakes.
First, some definitions

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

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

But first, I want to define what I mean by “scheduling,” since it’s an overloaded word.

- I’m defining it broadly as placing all the computations and data which make up a program’s execution, onto physical resources in space and time.

It isn’t just some unit or single algorithm responsible for choosing what instruction or thread to run next, it’s woven through systems all the way up to the order of their outermost loops, and even the choice of structure and nesting of those loops.

And with systems built around leveraging the enormous potential concurrency of data-parallelism for performance, this is the essential question. Because of the huge degree of latent parallelism in the graphics pipeline, we have many choices to make in how we map computations and data onto resources in space and in time.

- And throughout this talk I’m going to use **task** to describe the single, discrete units of work about which scheduling decisions are often made.
The workload: Direct3D

For this talk, the main workload we want to schedule is Direct3D.

The logical pipeline is a series of fixed-function and programmable shader stages, - taking in triangles at the top,
- transforming data through the pipeline
- and producing pixels at the bottom.

- From here on, I'm going to denote logical stages by these rounded rectangles, and distinguish fixed-function from programmable items by color.
The machine: a modern GPU

The machine we have to run that workload is a multicore GPU, like Mike introduced this morning.

It has a number of programmable shader cores, fixed-function logic, and fixed-function control for managing the execution of the pipeline, and movement of data through it.

I’m going to denote physical resources by sharp-cornered boxes, and use the same color scheme as for the logical pipeline stages to distinguish programmable from fixed-function resources.

...Given that setup, the first thing we need is a way to map the logical, sequential D3D pipeline onto this machine.
You should think about scheduling a draw call through the pipeline as mapping a series of tasks, onto the processor resources over time. These tasks correspond to the execution of logical stages over specific data items.

To see what that might look like in practice, I've spread some of the machine resources out horizontally. The vertical axis shows the progression of time.

- To process one draw call, first an Input Assembler unit runs the Input Assembly stage of the pipeline, loading a batch of vertices from memory.
- It passes these vertices to a Shader Core, which runs the Vertex Shader stage over them.
- The shaded vertices are fed to a Primitive Assembler unit which runs the Primitive Assembly stage to batch up triangles.
- ...which are then fed to the Rasterizer.
- The Rasterizer generates a stream of fragments, which may be fed to multiple Shader Cores for Pixel Shading.
In this view, one of our key goals is to pack this graph as tightly as possible.

A dense packing completes a given amount of work in the least amount of time, and it means we are using all our resources efficiently.
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.

Once we’ve turned our problem into a huge pool of tasks, the next question is which tasks to generate and run when, and where we want to run them. Our choices here are especially driven by four things:

- **Resource constraints**: Tasks can only execute when we can fit their computation and data into the same place, at the same time.
- **Coherence**: Control coherence—doing many similar things together—is essential to shader execution efficiency. Data coherence—using similar data close together in space and time—is essential to memory system efficiency.
- **Load balance**: Irregularity in execution creates bubbles and can dominate the time it takes to complete an entire set of tasks.
- **Ordering**: Graphics APIs (like D3D) define strict ordering semantics, which restrict the range of possible schedules.
To see how resource constraints impact scheduling, imagine we’ve done a good job. All 4 shader processors are busy with existing work, and all their storage is full—everything is fully utilized.

- the input assembler is ready to fetch the next chunk of primitives.
- but where should it push the vertex shading tasks it will generate?

Clearly in this case we have no choice—it needs to stall until there is space available.

This is the simplest way resource constraints drive scheduling.

With a feed-forward pipeline, this stall is fine...
Resource constraints limit scheduling options

Key concept: Preallocation of resources helps guarantee forward progress.

But what if the completion of the shaders on which the Input Assembler is stalled, depended on freeing resources held by that Input Assembler?

- This can cause deadlock.

- So another key concept in scheduling is that static preallocation of all necessary resources before starting a task is often necessary to avoid this condition and 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

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

Coherence and Load Balance are also at odds with the fact that graphics workloads are irregular. The most obvious example is rasterization.

- one triangle might generate only a few fragments,
- while another might cover the whole screen.
- What’s more, one might be running a shader which takes thousands of cycles for every fragment, while the other is only one instruction long.

This creates several problems.
First, if you remember from Mike’s talk, shaders are optimized to around the assumption of fairly regular, self-similar work.
Second, imbalanced work creates bubbles in the pipeline schedule, destroying efficiency.

- The solution...
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.

Beyond Programmable Shading 2011

- The solution is to design the logical pipeline to split stages at points of irregularity, and dynamically generate a new, dense set of tasks for the next stage. These tasks can then be redistributed to restore load balance.

You should think of the fixed-function pieces of the graphics pipeline first and foremost as an efficient task-generation and aggregation system which keeps the Shader Cores busy running your code.
Beyond Programmable Shading 2011

Redistribution after irregular amplification

- Careful placement of redistribution after likely amplification points is a key concept in designing and scheduling graphics pipelines.
**Key concept:** Managing irregularity by dynamically generating, aggregating, and redistributing tasks.

...back in the task schedule, our example had one primitive batch assembled from one vertex shading task on one core,

- but rasterization generated multiple Pixel Shading tasks, which were spread across multiple cores.

Rasterization has created, aggregated, and redistributed new pixel shading tasks based on its irregular output stream.

- Careful placement of redistribution after likely amplification points is a key concept in designing and scheduling graphics pipelines.
Ordering

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**.

...but there’s a natural tension between **redistribution for load balance and coherence** and our last constraint: **ordering**.

**APIs like D3D** specify **strict ordering semantics**.

They mandate that **all framebuffer updates** must appear as though all triangles were drawn in **strict sequential order**.

- This obviously **heavily impacts** the **ways** in which **tasks** can be **redistributed**, and is another **key concept** in designing **pipeline scheduling strategies**.
Building a real pipeline

Given that background for thinking about scheduling the graphics pipeline, let’s look at a few real examples.
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 2011

The simplest thing you could imagine doing is recognizing that most of the amplification is in pixel processing, and statically assigning tiles of the screen to multiple Pixel Processors.

This is how, for example, the ARM Mali GPU works.

I consider this scheduling not because some dedicated unit called a scheduler is necessarily involved, but because it’s a strategy for mapping pipeline computations to a parallel machine.
If we imagine we have a simple division of the screen into 4 quadrants, one assigned to each of 4 pixel processors.

- A single triangle comes in.
Static tile scheduling

Exemplar: ARM Mali 400

Beyond Programmable Shading 2011

The Vertex Processor transforms it into the screen.
The tiles are **statically** assigned to processors.

- Each processor performs **pixel processing** on the **parts** of any **primitives** which **land** in its **tile**.
This actually works pretty well.

- The tiles naturally **capture significant locality** at each processor.

- Since all processing is in-order, and all pixel processors process a fixed portion of the screen at a time, **resource management** is simple.

- And most significantly, since all **parallelism is statically partitioned** over different **locations** on the screen, and there’s a **single front-end** processor, the **apparent ordering** of **updates** to any **individual pixel** is as **strictly ordered**.
The problem: load imbalance

only one task creation point.

no dynamic task redistribution.

...but because the allocation of tiles to pixel processors is static, hotspots can create load imbalance.

- if too much of the work lands on the tiles assigned to one processor,
- ...that processor maxes out, while the others sit idle.

The problem is that tasks are only distributed statically, not according to dynamic load.

Exemplar:
ARM Mali 400
Sort-last fragment shading

Redistribution restores fragment load balance.
But how can we maintain order?

Exemplars:
NVIDIA G80, ATI RV770

The obvious solution is not to statically tile the screen, but to turn it all into a big soup of fragments which get shaded by a dynamically assigned pool of pixel shaders.

- This smoothly load balances across pixel shaders by dynamically redistributing tasks, but because shaders now process fragments in parallel and asynchronously, the key question is how can we maintain order?
The key strategy to making this work is to pre-allocate output storage in FIFO order.
Sort-last fragment shading

Exemplars: NVIDIA G80, ATI RV770

Beyond Programmable Shading 2011

The Pixel Shaders can then complete asynchronously, writing their outputs to the pre-reserved location associated with the item they are shading.
Sort-last fragment shading

The backend then blends the fragments in FIFO order, producing the same apparent order as if the fragments were processed serially, in the order of the input primitives.

This approach is called sort-last fragment shading, because it allows out-of-order fragment shading, but restores order at the end of the pipeline.

This type of architecture has been used in most modern NVIDIA or ATI GPUs.

But what I've shown so far can still suffer from load imbalance between vertex and pixel shading.
Unified shaders

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

Exemplars: NVIDIA G80, ATI RV770

Modern GPUs—starting with G80 and Xenos in the Xbox—also introduced a unified shader architecture, where the same physical processors are used to run all shader stages.

This solves the vertex vs. pixel shading load balance problem by time-multiplexing different stages onto the same shared processors.
If we go back to our original simple example,
- notice that we execute both vertex and pixel shading on the same processor at different points in time.

But the key question now is how to choose what stage to run at any given time.
Moving back to a simple logical D3D9 pipeline, the first thing we can do is assign unique priorities to each stage.

One logical choice is to prioritize the top of the pipeline over the bottom. This helps ensure that there is always work in the pipeline, keeping any stage from starving.

But now what’s to stop the system from running wild, assembling and processing all vertices before consuming any, generating a huge stream of intermediate data?
The next thing we can do is introduce fixed-sized queues in between the stages. Fixed queues, combined with our top-down priority order, create backpressure:

- **Early stages** run until they run out of queue space, at which point **later stages**—which consume those queue items—are the highest-priority stages available to run.
Switching back to a few cores on our actual processor, let’s imagine the vertex stage’s input queue is full. It’s the highest-priority stage with available work, so at the next chance, each of the shader cores dequeues and begins processing a batch of vertex shading.
Beyond Programmable Shading 2011

Switching back to a few cores on our actual processor, let's imagine the vertex stage's input queue is full. It's the highest-priority stage with available work, so at the next chance, each of the shader cores dequeues and begins processing a batch of vertex shading.
The output of these fills up the Primitive Assembly queue.
Beyond Programmable Shading 2011

Scheduling the pipeline

...So now, even though the vertex queue has work and is higher priority, the two shaders run pixel shaders next, because the logical vertex stage is stalled on backpressure from its fixed output queue.
Beyond Programmable Shading 2011

Scheduling the pipeline

...and so on until the entire pipeline is drained.
Queue sizes and backpressure provide a natural knob for balancing horizontal batch coherence and producer-consumer locality.
As the last example, I want to **switch gears** quickly to talk about a real example of **computational graphics**: a whole new rendering pipeline and architecture, built in software on these same machines.

- NVIDIA’s OptiX ray tracing system presents a **pipeline abstraction** in the spirit of Direct3D and OpenGL, but for ray tracing.

- **Application functionality** is specified in **shader-style programs**, where you can specify how to **intersect rays** against your primitive types—almost like *programmable rasterization*—and what to do when rays hit a surface, to compute its color or other results.

- The system is responsible for the overall **pipeline execution structure**, including
  - the **acceleration structures** used,
  - how they are **traversed**, and
  - the **order of execution**, and
- managing the **resources** needed by this whole process.
Two issues come up in scheduling a ray tracing pipeline like this on the GPUs we’ve talked about.

- First, like in the graphics pipeline, is whether to traverse rays in a breadth-first or depth-first fashion. Wider, breadth-first execution lets you aggregate more work which is potentially coherent to execute as large batches, but it requires storage for all the tasks you want to aggregate.

- Depth-first execution sacrifices some potential parallelism for reduced storage footprint.

- OptiX makes the choice of going as wide as the machine needs, but no wider. Filling all the useful physical thread contexts on the GPU with rays, but then recursively traversing and completing those rays before aggregating more.

- Second, we’ve seen how the shader cores require coherent batches of work to execute in a SIMD fashion to run fast. But rays may diverge: they may
  - traverse different objects,
  - hit different surfaces,
  - or otherwise do different things from their neighbors.

So how can we deal with that?
The first thing you …scalar

Packet tracing: trace bundles of rays as a group.

Keep coherent items together
- follow same code paths together
- access same geometry, acceleration structures as a group

When some die off or do different things, mask them out and keep going with the rest of the packet.

Problem is low packet occupancy once they diverge
Timo Aila and Samuli Laine came up with an approach to improve SIMD utilization in wide SIMD machines. Their new idea was to break up the packets and allow data divergence, while keeping SIMD program execution. Where all rays in a packet had to traverse the acceleration structure together, and intersect the same primitives at the same time, they let different rays can traverse and intersect different objects.

But all rays are still either traversing or intersecting at any one time, so you can still execute in a SIMD fashion. In code, it looks like this.

```c
while(state != done) {
    if (state == traverse) traverse();
    if (state == intersect) intersect();
}
```

The thing to realize is this lets rays re-converge. Each time around the loop, if any ray is ready to traverse, it does, as part of the batch. This significantly increases SIMD utilization. But in the case of OptiX, you have multiple shaders for different intersection and surface shading tasks.
A pipeline program as a state machine

```java
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;
            ...
        }
}
```

This state machine idea generalizes to multiple stages. Here, we have:
- the traversal stage
- multiple intersectors for different primitive types
- and multiple shaders, for different surface types.
- We can also add a scheduler which makes context-aware decisions about what stages to run when

This is what OptiX does. But notice that some of these stages are your programs, while others are defined by the pipeline.

In practice, this is all generated by a JIT compiler.
Once you bind programs to all the different stages and objects in your scene, OptiX fuses and transforms them with pipeline-defined code and scheduling logic, into one large, self-scheduling program for the whole pipeline, which runs directly on the shader cores as a single CUDA program.

...Steve Parker will talk about OptiX more later this afternoon in this course, and on Thursday in the compilers for graphics course.
To pop back up for a little review,
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.

First, you should think of *scheduling the pipeline as mapping tasks onto cores.* This is a natural model within which to think about everything else.

When scheduling a task, it is often easiest to *preallocate all resources it might need to complete.* This helps *ensure forward progress* and *prevent deadlock.*

This suggests a few 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.**

*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.

And this is also a key reason the first geometry shader implementations were slow:

Geometry Shaders allow programmable amplification in shaders.

This leads to a dilemma in how to cope with the resulting resource allocation.

You can assume the worst-case, and always stream the outputs through very large storage, like DRAM. This is what ATI's first implementation did, and it behaved relatively well at large very amplification, but had high overhead for small amplification which should be kept on-chip.

You can also just bound the worst case and then scale down the degree of parallelism in your scheduler until you know you can fit on chip. This is fast for small amplification, but leaves the shaders underutilized when larger amplification requires reduced parallelism.
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.

This leads to another lesson:

an obvious reason today’s GPUs don’t have programmable rasterization is that it's computationally intensive.

but it’s also a key point of highly irregular data amplification.

It’s responsible for quickly carving up triangles, generating an irregular stream of fragments, and then re-aggregating that irregular stream into coherent batches of fragment shading tasks.

And it has to help redistribute these in a way that still maintains ordering semantics.
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.

And finally, **order matters.** Any scheduling system for today’s graphics pipelines needs to **deeply consider** how it will **maintain API ordering**, especially when doing **task redistribution.**
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?

A few questions for the future:

As we’ve seen in many examples, order is the enemy of load balance. It is the biggest shackle around flexibility in scheduling. It would be interesting to see how we might relax this constraint, and how much it might enable scheduling to improve efficiency.

Another question is whether it would be possible to build a completely generic scheduler for any application-defined pipeline, and what application-specific hints it might need to work well.

This is a key question if we want to move towards a world of programmable graphics pipelines.
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

Here are a few starting points to learn more, and sources for some of the ideas I included here.

I’d especially encourage you to look at the *Pomegranate* architecture, which is similar to the parallel primitive processing added to NVIDIA’s *Fermi GPU*. Maintaining order gets even more challenging with parallel primitive processing and rasterization.

And also NVIDIA’s *OptiX* ray tracing system, which defines another pipeline-style graphics API, but with recursion, and all scheduled in software on current GPUs. These same ideas apply there.
Steve Parker will be talking about OptiX from a programming perspective in this course this afternoon, and also from a scheduling and compiler perspective—in more detail than I have here—on Thursday in the compilation for graphics course.
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