Issues And Challenges In Compiling for Graphics Processors

Norm Rubin
Fellow
AMD Graphics Products Group

Overview

AMD Smarter Choice

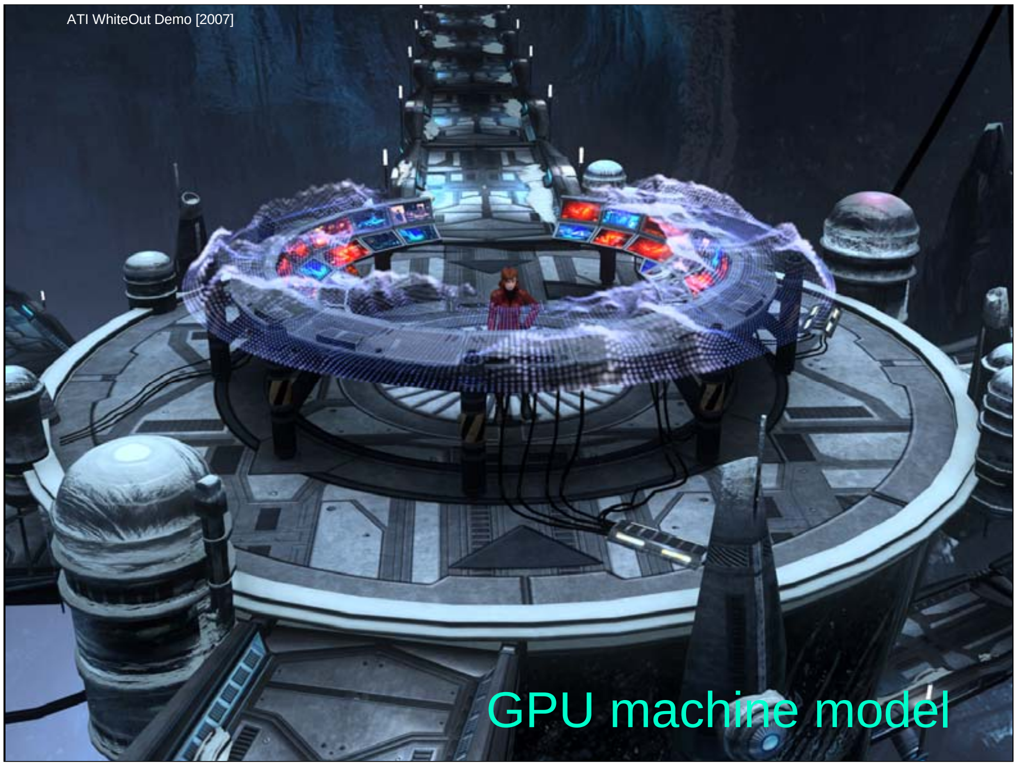
What is the difference between a GPU and a CPU?

What is the programming model for graphics

- How do 20,000 people easily write high performance parallel code?

Where does this compiler fit?

CGO 2008



All the images in this talk were rendered from real-time demos.

Chip Design Focus Point

AMD Smarter Choice

CPU

Lots of instructions little data
- Out of order exec
- Branch prediction

Reuse and locality

Task parallel

Needs OS

Complex sync

Latency machines

GPU

Few instructions lots of data
- SIMD
- Hardware threading

Little reuse

Data parallel

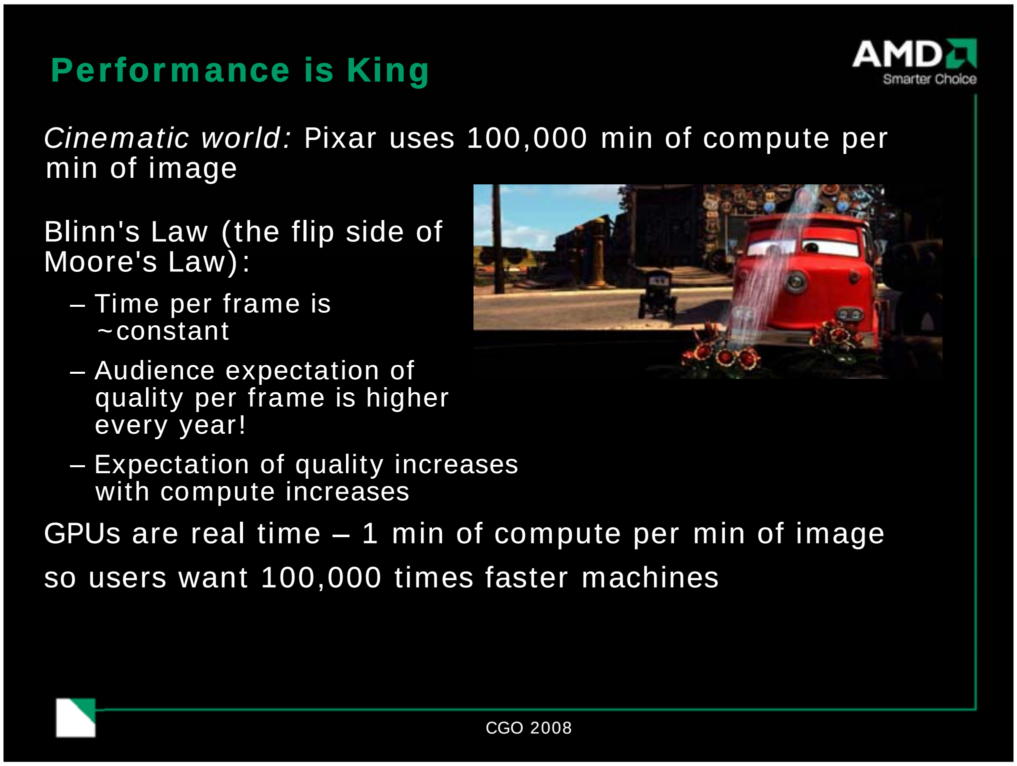
No OS

Simple sync

Throughput machines

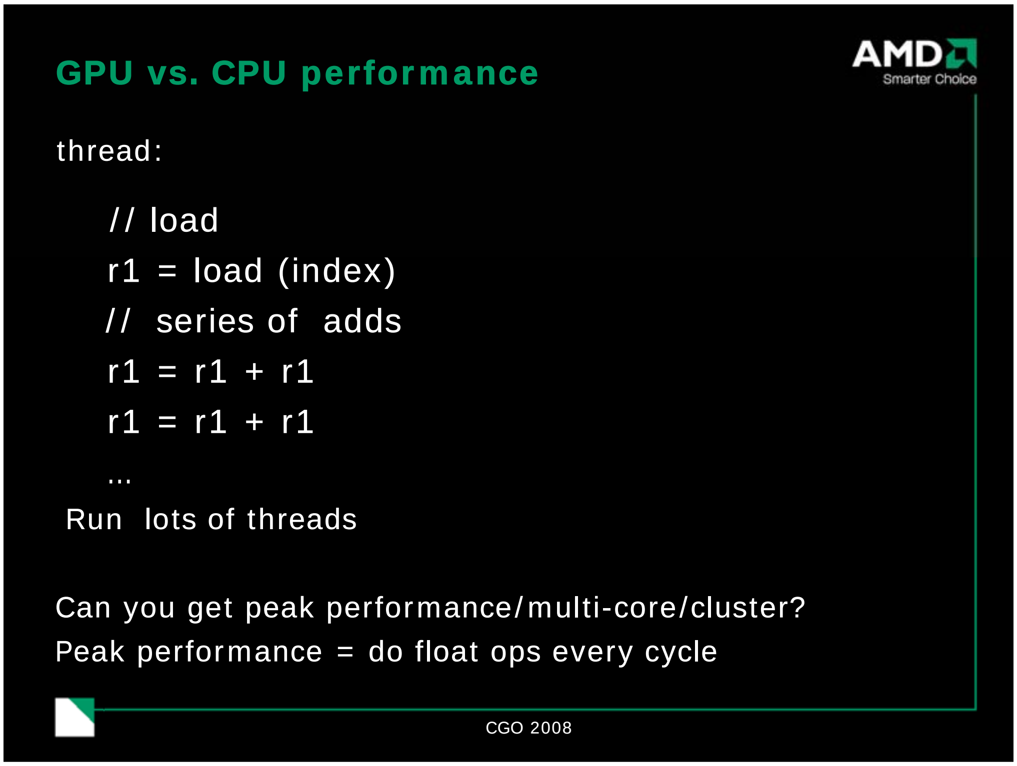
CGO 2008

The main difference is that gpu's use multi-threading to tolerate latency, each time you wait for a read, just start another thread, This works if there are lots of threads

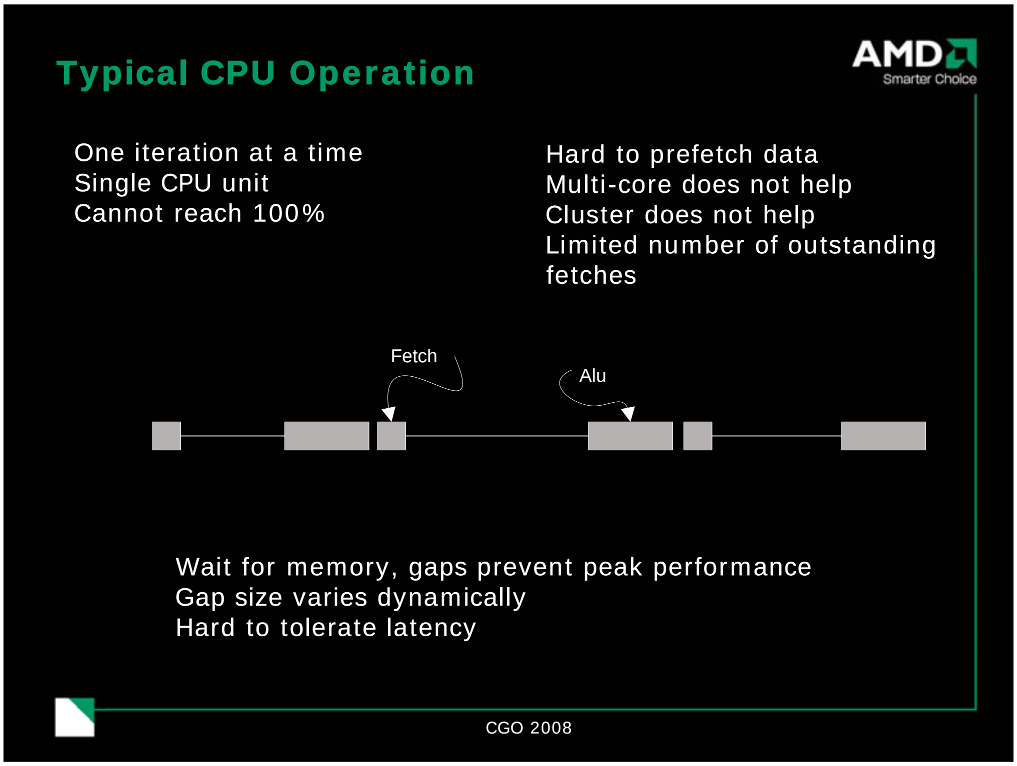


100,000 times faster for current pixar results, more needed next year

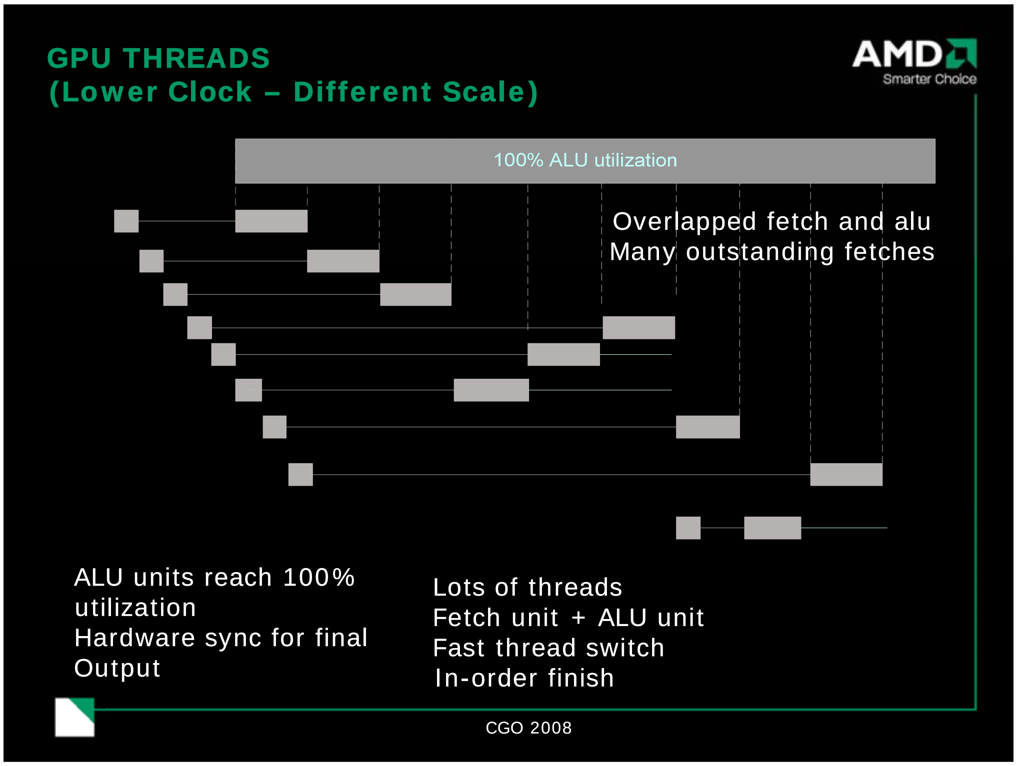
In entertainment-related computer graphics business, the amount of time that it takes to compute one frame is constant over time. The reason is that audience expectation increases at the same rate as computer power.



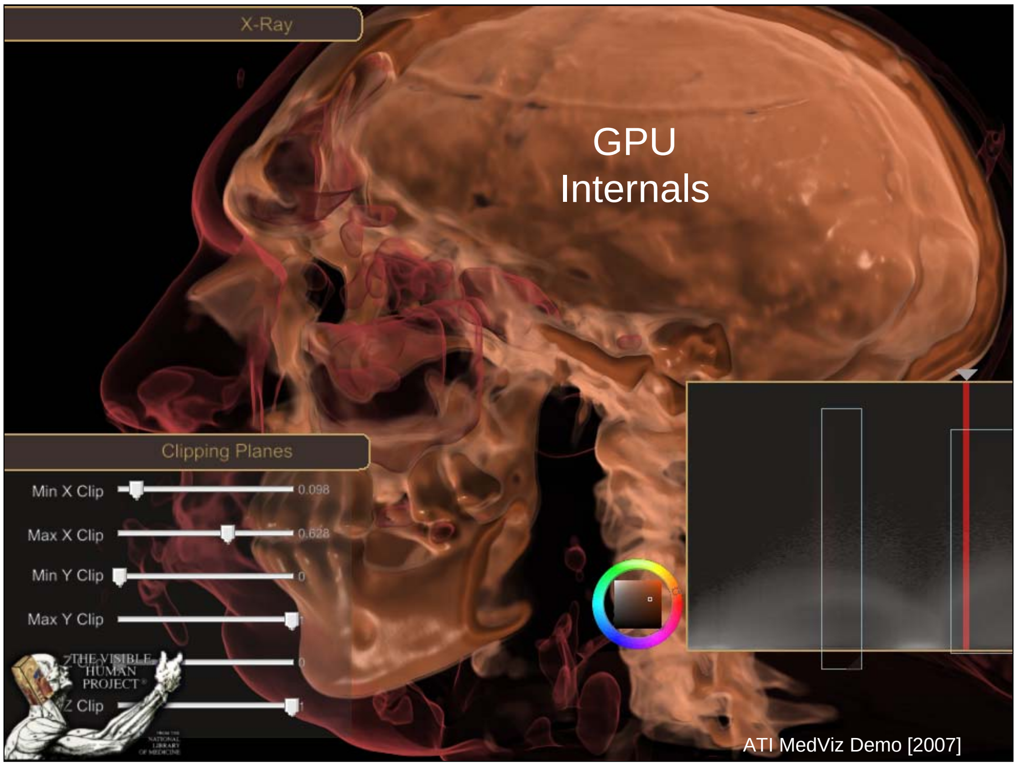
This simple program is supposed to show a case where the gpu is much better then the cpu

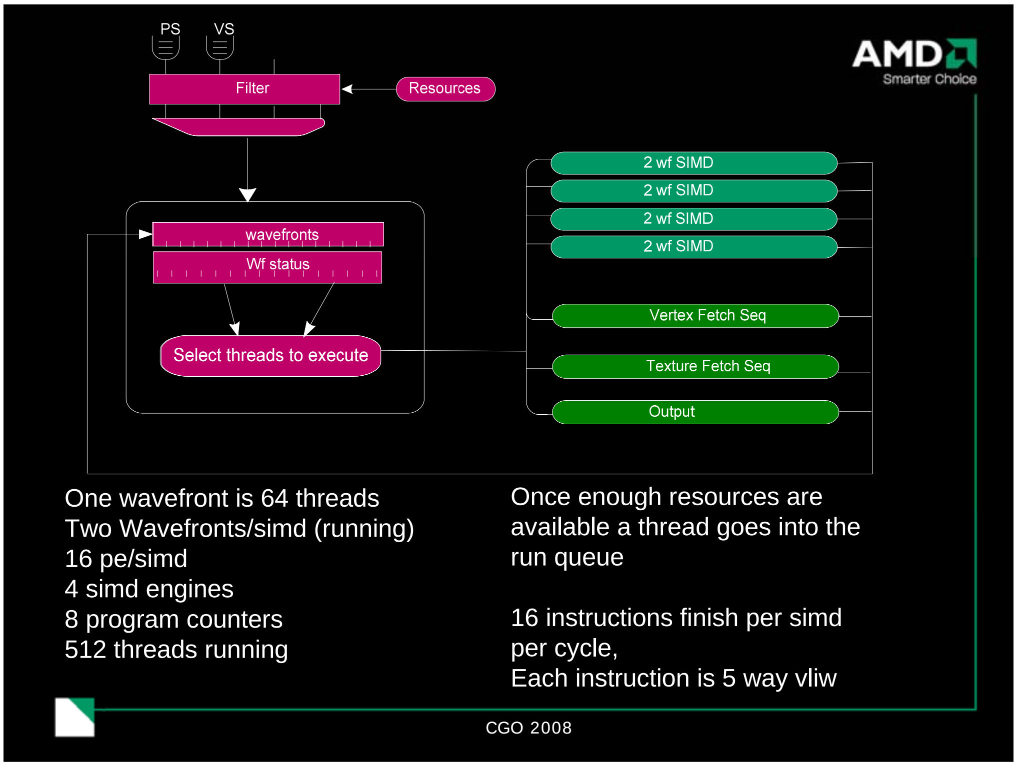


The gap between fetch and the alu is the latency



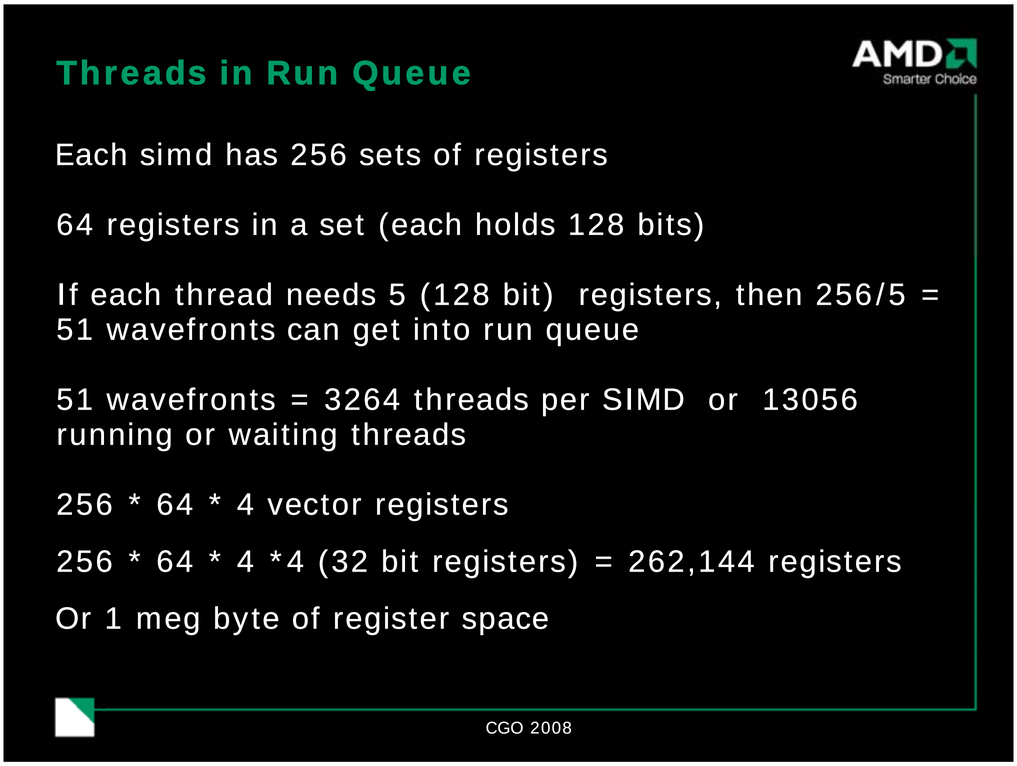
The big bar at the top shows when the float units are running. It is 100% active if there are enough threads


X-Ray
GPU
Internals
Clipping Planes
Min X Clip
0.098
Max X Clip
Min Y Clip
0
Max Y Clip
THE VISIBLE
HUMAN
PROJECT
Z Clip
ATI MedViz Demo [2007]



Wavefronts are 64 thread units, they are also called warps

All resources are allocated at start, so no deadlock is possible



A thread is one pc/one group of registers, a wavefront is 64 threads

Implications

CPU: Loads determine performance

- Compiler works hard to
  - Minimize ALU code
  - Reduce memory overhead
  - Try to use prefetch and other magic to reduce the amount of time waiting for memory

GPU: Threads determine performance

- Compiler works hard to
  - Minimize ALU code
  - Maximize threads
  - Try to reorder instructions to reduce synchronization and other magic to reduce the amount of time waiting for threads

Graphics Programming Model

CPU part

```
for each frame  (sequential)  {
  build vertex buffer
  set uniform inputs
  draw
}
```

This is producer consumer parallelism

Internal queue of pending draw commands (often hundreds)

Programming Model – GPU Part

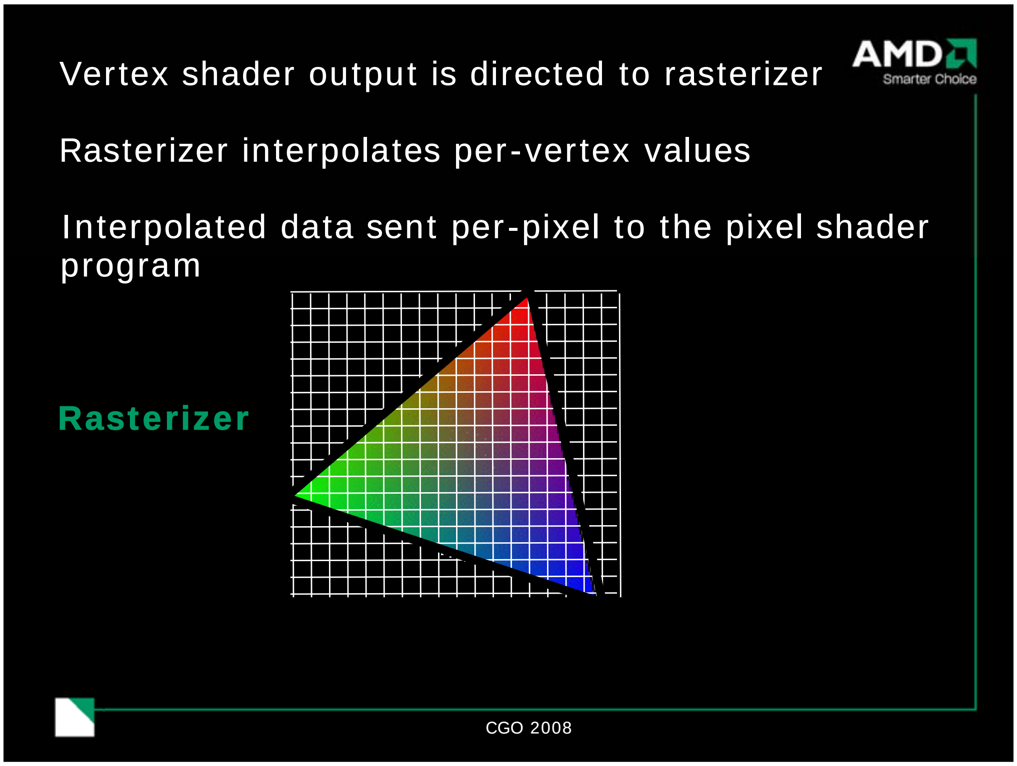
```
foreach vertex in buffer (parallel) {
  call vertex kernel/shader }

foreach set of 3 vertex outputs (a triangle) (seq) {
  fixed function rasterize
  foreach pixel (parallel) {
    call pixel kernel/shader
  }}
```

Nothing about number of cores

Nothing about sync

Developer just writes kernels (in RED)

CGO 2008



Each box in the grid gets its own thread, thread count is determined by hardware not by app, bigger screen means more threads

The whole system scales with bigger screen or more processors, without change

Pixel Shader

```
float4 ambient;
float4 diffuse;
float4 specular;
float Ka, Ks, Kd, N;
float4 main( float4 Diff  : COLOR0,     float3 Normal: TEXCOORD0,
             float3 Light : TEXCOORD1, float3 View  : TEXCOORD2 )
    : COLOR
{
   // Compute the reflection vector:
   float3 vReflect = normalize(2*dot(Normal, Light)*Normal -
                     Light);

   // Final color is composed of ambient, diffuse and specular
   // contributions:
   float4 FinalColor = Ka * ambient +
                       Kd * diffuse * dot( Normal, Light ) +
                       Ks * specular * pow( max( dot( vReflect,
                                                   View), 0), N ) ;
   return FinalColor;
}
20 statements in byte code
```


What is the
parallelism model?

Programming model

Vertex and pixel kernels (shaders)

Parallel loops are implicit

Performance aware code does not know how many cores or how many threads

All sorts of queues maintained under covers

All kinds of sync done implicitly

Programs are very small

Parallelism Model

All parallel operations are hidden via domain specific API calls

Developers write sequential code + kernels

Kernel operate on one vertex or pixel

Developers never deal with parallelism directly

No need for auto parallel compilers

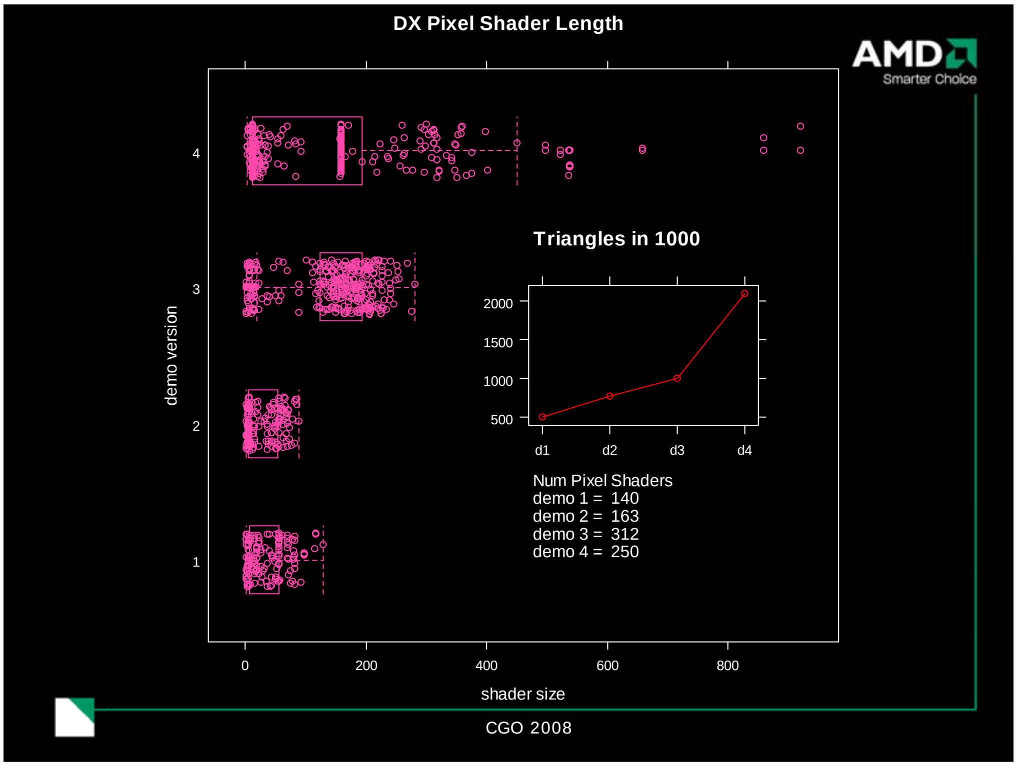
Ruby Demo Series

Four versions – each done by experts to show of features of the chip as well as develop novel forward-looking graphics techniques

First 3 written in DirectX9, fourth in DirectX10

All four demos have specific names, for a graphics talk I'd use the acutal names

Either the demo or movie goes here



Box and whisker plot of shader length, box is ½ std div around mean, line is 1 and 1/2 , outliers after this, size of max shader is growing, more control flow,

Inside graph is the max triangles in 1000 triangle units so the highest value is 2 million, number of shaders is the count, time or chip version is going up

d1 d2 d3 d4 are the demo numbers

We see a double exponent in growth, triangles and shader size, count does down because of more control flow

The hor scale is in asm lines so an 800 asm line shader is a big one

Shader Compiler (SC)

- Developers ship games in byte code
  - Each time a game starts the shader is compiled
- Compiler is hidden in driver
  - No user gets to set options or flags
- Compiler updates with new driver (once a month)
- Compile done each time game is run
- Like a JIT but we care about performance
- SC runs on consoles/phones/laptops/desktops etc

Relations to Std CPU Compiler

About ½ code is traditional compiler, all the usual stuff

SSA form

Graph coloring register allocator

Instruction scheduler

But there is a lot of special stuff!

Some Odd Features

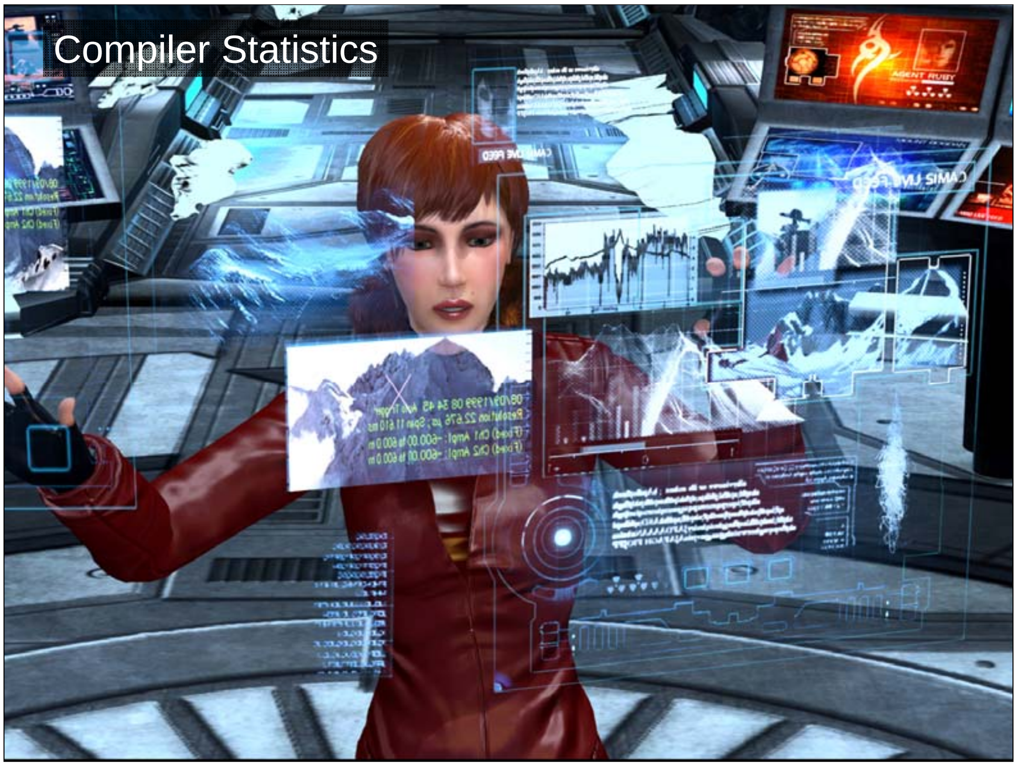
HLSL compiler is written by Microsoft and has its own idea of how to optimize a program

- Each compiler fights the other, so SC undoes the ms optimizations

Hardware updates frequently so

- SC supports large number of targets, internally (picture gcc done in c++ classes)
- One version of the compiler supports all chips

Compiler Statistics



Register Allocation

A local allocator run as part of instruction scheduling

A global – Graph coloring allocator

(even though this is a JIT)

Two allocators for speed

New Issues

Less registers are better, because of threading

Load/store multiple takes same time as load/store so spill costs are different

Registers hold 4 element vectors

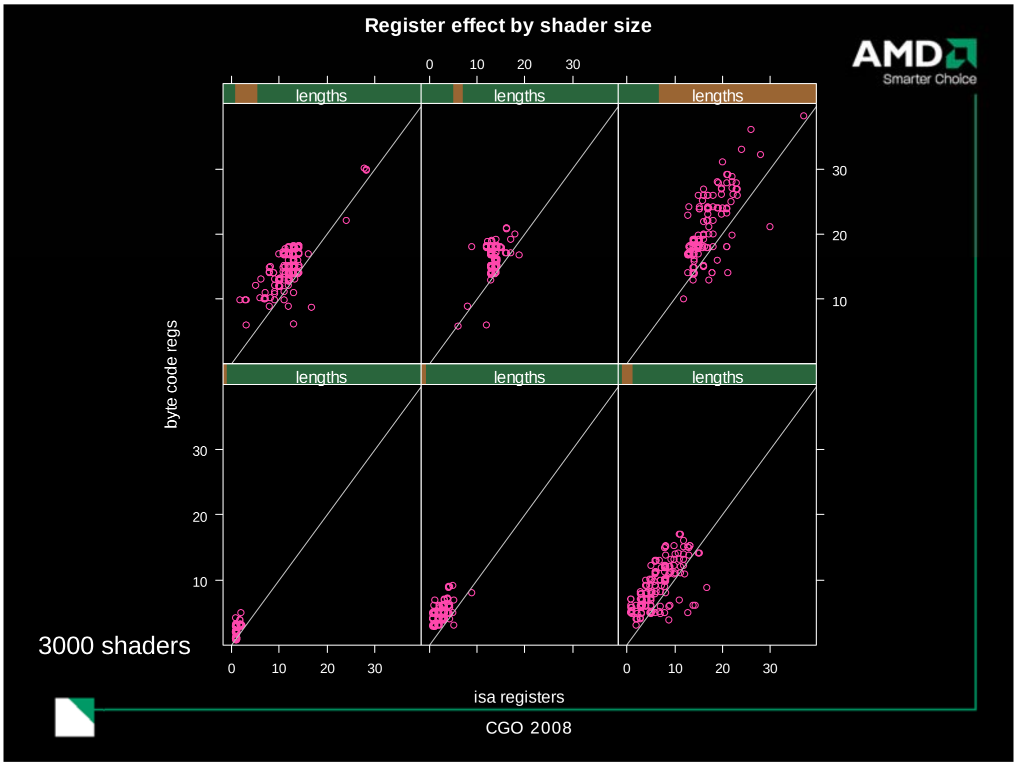
Register Allocation

Two register allocators – hlsl/manual vs SC

SC, which knows the machine usually needs less registers

Register delta is not related to program size

But bigger programs need more registers

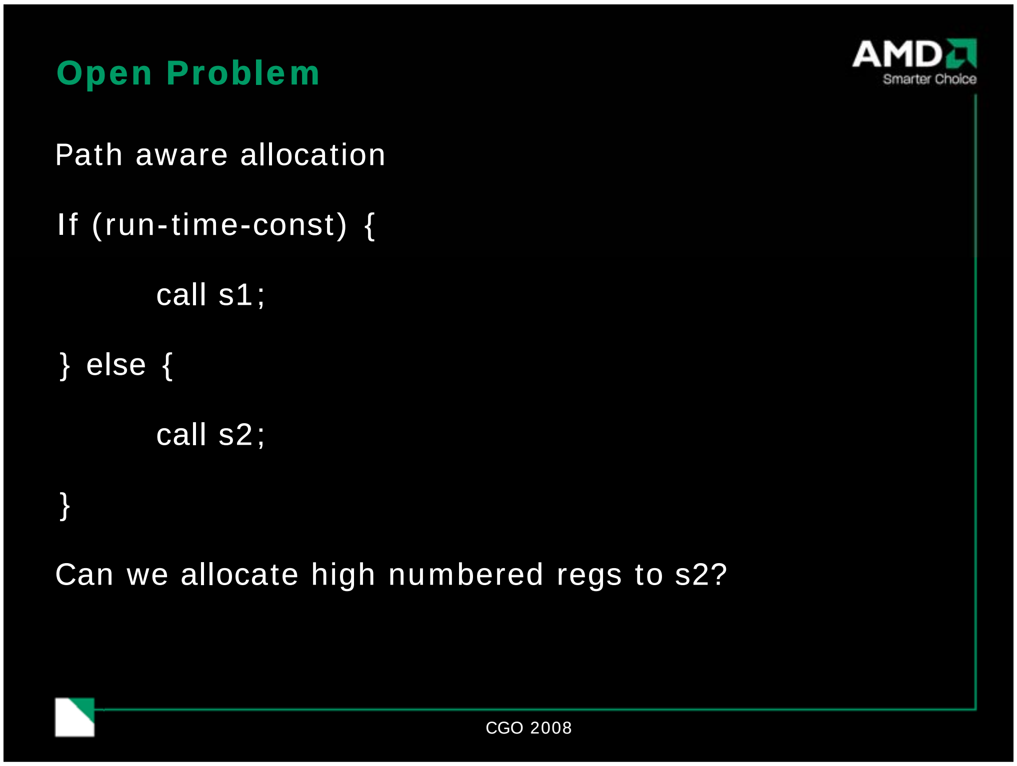


Shows the delta in registers, hlsl thinks the machine has vector registers and it does, so hlsl does an ok job, I split the 3000 shaders into 6 groups by length

Smallest are lower left, biggest are upper right (lots of small one not so many large ones)

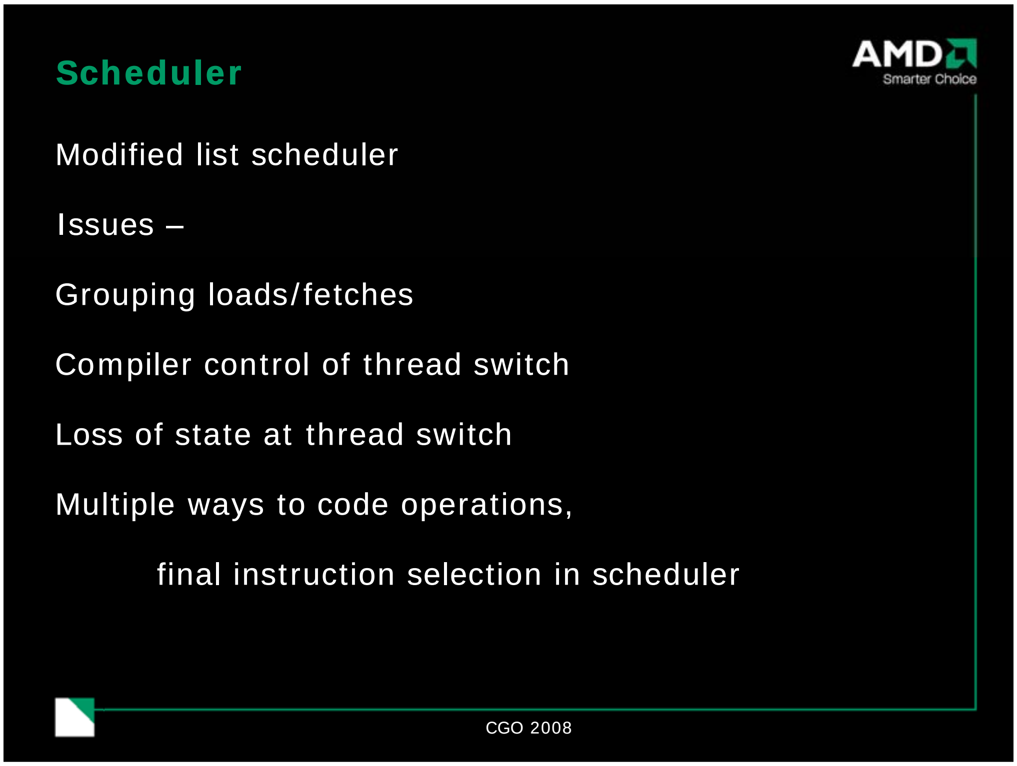
A shader on the diag line means hlsl and sc used the same number of registers

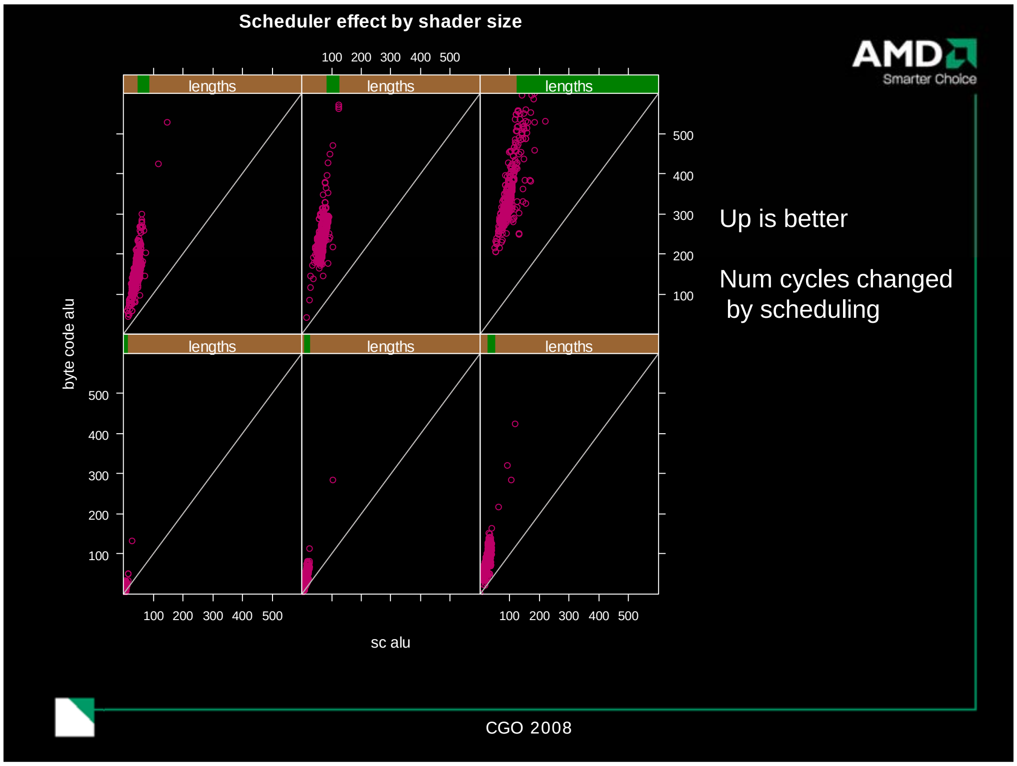
A dot can be a lot a shaders if they overlap



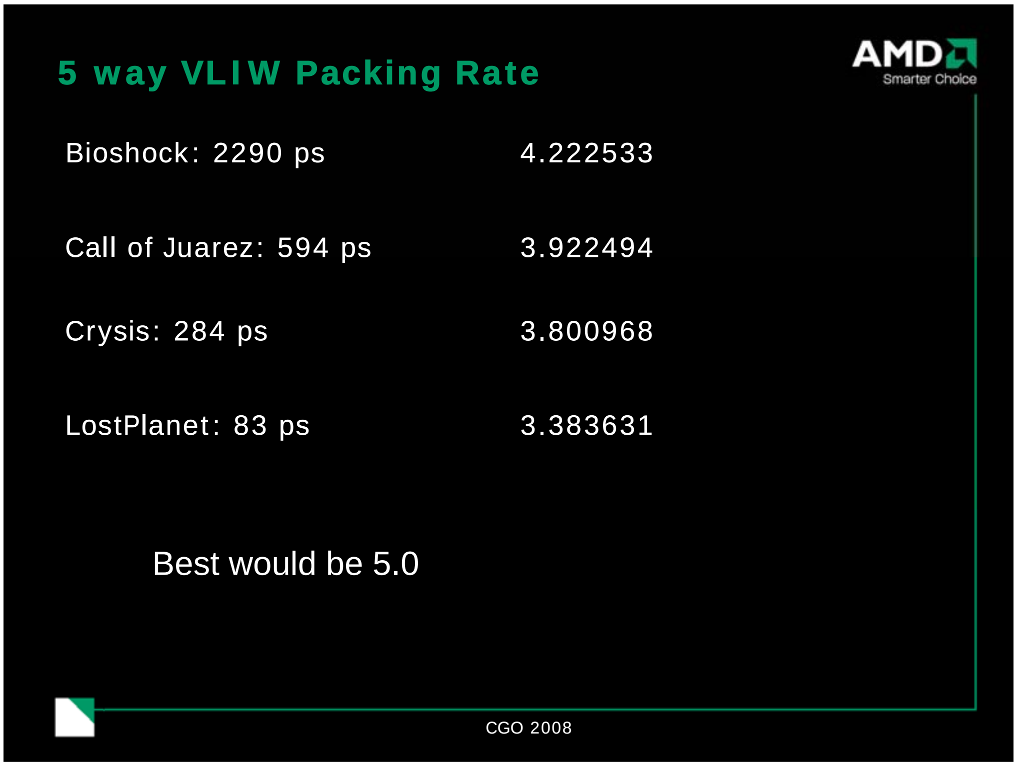
Problem is to allocate registers and then at run tim, if we know that s1 will always be called just say the shader needs less registers and so it gets more threads

Handle this without recompiling

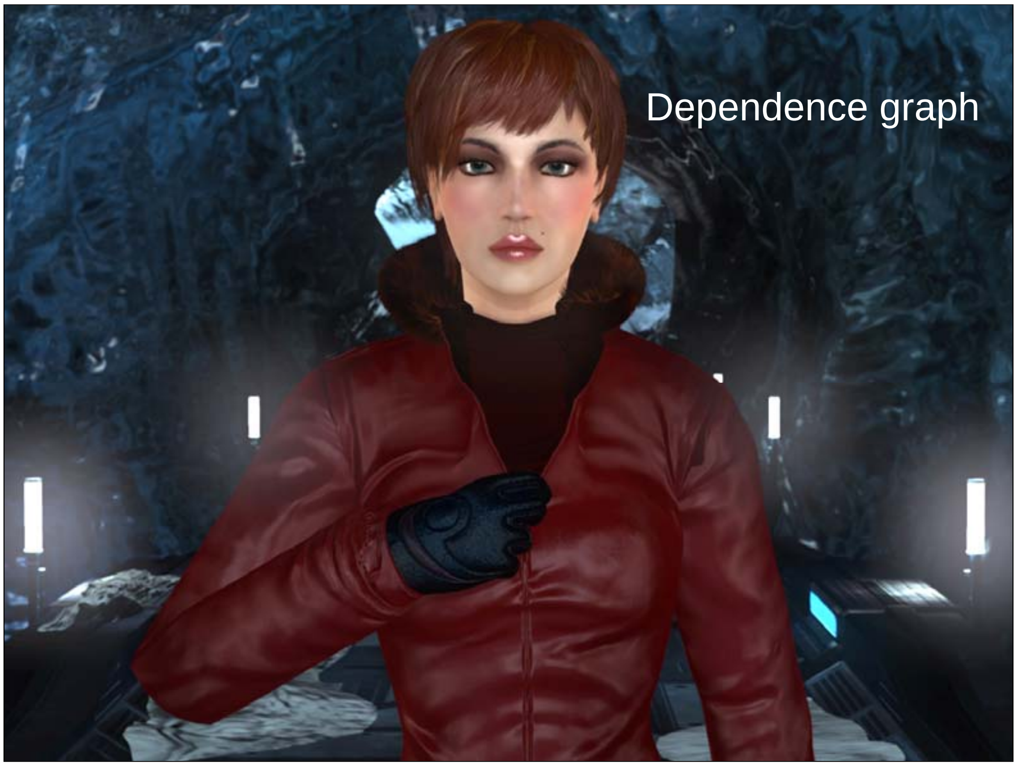

Scheduler
AMD
Smarter Choice
Modified list scheduler
Issues –
Grouping loads/fetches
Compiler control of thread switch
Loss of state at thread switch
Multiple ways to code operations,
final instruction selection in scheduler
CGO 2008

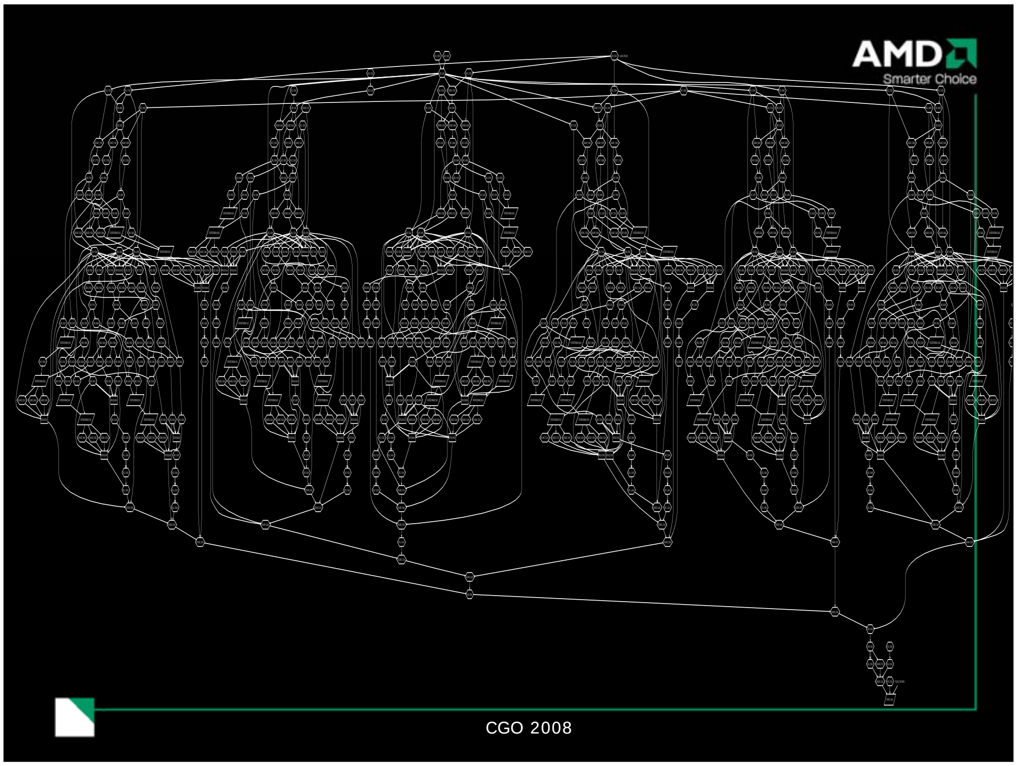


Here we have the same 3k shaders hlsl thiinks it is vector machine but is actually 5 way vliw, so the vector assignment does not work well

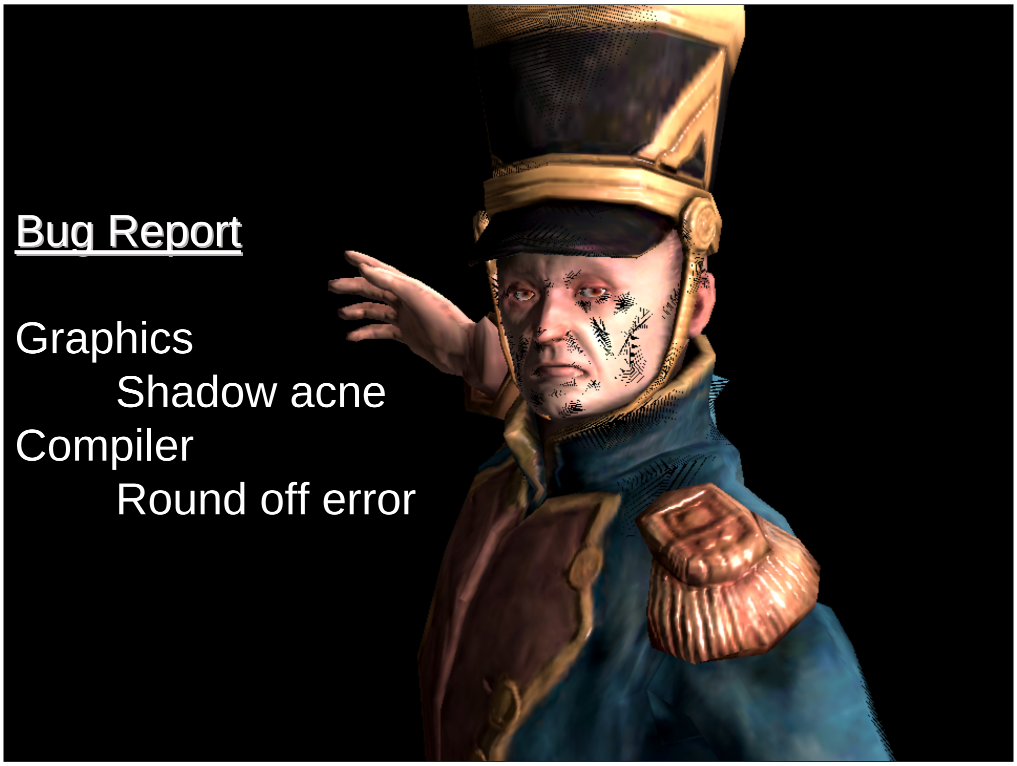


These are 5 current dx games, number of pixel shader and average packing in 5 way vliw issue


Dependence graph



This is an actual graph generated by sc for a single basic block in a shader computing perlin noise, the greedy list scheduler should have left some holes in the schedule fror this case, I think this is clearly a hand or hlsl unrolled loop, can we do some fast graph analysis to figure out the structure?



This was a real bug report, I listed the two names for the error

Questions?

norman.rubin @ amd.com

Trademark Attribution