Issues And Challenges In Compiling for Graphics Processors

Norm Rubin
Fellow
AMD Graphics Products Group
Overview

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?
All the images in this talk were rendered from real-time demos.
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 than the CPU.
Typical CPU Operation

One iteration at a time
Single CPU unit
Cannot reach 100%

Hard to prefetch data
Multi-core does not help
Cluster does not help
Limited number of outstanding fetches

Wait for memory, gaps prevent peak performance
Gap size varies dynamically
Hard to tolerate latency

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.
Wavefronts are 64 thread units, they are also called warps
All resources are allocated at start, so no deadlock is possible
Threads in Run Queue

Each simd has 256 sets of registers

64 registers in a set (each holds 128 bits)

If each thread needs 5 (128 bit) registers, then 256/5 = 51 wavefronts can get into run queue

51 wavefronts = 3264 threads per SIMD or 13056 running or waiting threads

256 * 64 * 4 vector registers

256 * 64 * 4 *4 (32 bit registers) = 262,144 registers

Or 1 meg byte of register space

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)
Vertex shader output is directed to rasterizer

Rasterizer interpolates per-vertex values

Interpolated data sent per-pixel to the pixel shader program

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
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
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 actual 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 $\frac{1}{2}$ 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
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
Open Problem

Path aware allocation

If (run-time-const) {
    call s1;
} else {
    call s2;
}

Can we allocate high numbered regs to s2?

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

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
Here we have the same 3k shaders hlsli thinks 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

### 5 way VLIW Packing Rate

<table>
<thead>
<tr>
<th>Game</th>
<th>Packing Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bioshock</td>
<td>2290 ps</td>
</tr>
<tr>
<td>Call of Juarez</td>
<td>594 ps</td>
</tr>
<tr>
<td>Crysis</td>
<td>284 ps</td>
</tr>
<tr>
<td>LostPlanet</td>
<td>83 ps</td>
</tr>
</tbody>
</table>

**Best would be 5.0**
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 for this case. I think this is clearly a hand unrolled loop, can we do some fast graph analysis to figure out the structure?
Bug Report

Graphics
  Shadow acne
Compiler
  Round off error

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

norman.rubin @ amd.com

Trademark Attribution

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.

©2006 Advanced Micro Devices, Inc. All rights reserved.