Announcements

• HW#7 late being posted
• Hand back midterm
• Don’t forget project topics due Thursday
Notes on MPI

- Hard to think about. Running on different machine, so setting variables *does not* get set on all, like it does with OpenMP or pthreads

- On homework, want to load JPG data on rank0, send to rest

- Tricky: before you can send to rest, they have to know how big of an area to allocate to store it in. How will they know this?
• MPI does not give good error messages. OpenMPI worse than MPICH. Will often get segfault, hang forever, or weird stuff where it runs 4 single-threaded copies of program rather than one 4-threaded

• Many of the commands are a bit non-intuitive
Graphics Processing Units

- Retrospective on old graphics hardware
- Framebuffer is simple (though annoying pointer match like in sobel or worse). VGA Mode 13h, 0xa0000, 64kB
- Old video game systems didn’t even have that. Why? 1MB for a framebuffer was expensive. Only 64k RAM total.
- Atari 2600 only had 128B of RAM, total. 40-bit framebuffer. Racing the beam.
- Also could do sprites or tile based.
GPUs
Interfaces

- Originally each vendor had own 3D interface, SGI standardized
- OpenGL – SGI
- Direct3D – Microsoft
- Vulkan – new interface with less baggage
- Originally for HPC/CAD but gaming has brought down prices for everyone.
GPGPUS

• Interfaces needed, as GPU companies do not like to reveal what their chips do at the assembly level.
  – CUDA (Nvidia)
  – OpenCL (Everyone else) – can in theory take parallel code and map to CPU, GPU, FPGA, DSP, etc
Why GPUs?

• Old example:
  – 3GHz Pentium 4, 6 GFLOPS, 6GB/sec peak
  – GeForceFX 6800: 53GFLOPS, 34GB/sec peak

• Newer example
  – Raspberry Pi, 700MHz, 0.177 GFLOPS
  – On-board GPU: Video Core IV: 24 GFLOPS
GPGPU Key Ideas

- Using many slimmed down cores
- Have single instruction stream operate across many cores (SIMD)
- Avoid latency (slow textures, etc) by working on another group when one stalls
GPU Benefits

- Specialized hardware, concentrating on arithmetic. Transistors for ALUs not cache.
- Fast 32-bit floating point (16-bit?)
- Driven by commodity gaming, so much faster than would be if only HPC people using them.
- Accuracy? 64-bit floating point? 32-bit floating point? 16-bit floating point? Doesn’t matter as much if color slightly off for a frame in your video game.
- Highly parallel
GPU Problems

- Optimized for 3d-graphics, not always ideal for other things
- Need to port code, usually can’t just recompile cpu code.
- Companies secretive.
- Serial code with a lot of control flow runs poorly
- Off-chip memory transfers can be slow
Latency vs Throughput

- CPUs = Low latency, low throughput
- GPUs = high latency, high throughput
- CPUs optimized to try to get lowest latency (caches); with no parallelism have to get memory back as soon as possible
- GPUs optimized for throughput. Best throughput for all better than low-latency for one
Older / Traditional GPU Pipeline

- In old days, fixed pipeline (lots of triangles).
- Modern chips much more flexible, but the old pipeline can still be implemented in software via the fancier interface.
Older / Traditional GPU Pipeline

- CPU send list of vertices to GPU.
- Transform (vertex processor) (convert from world space to image space). 3d translation to 2d, calculate lighting. Operate on 4-wide vectors (x,y,z,w in projected space, r,g,b,a color space)
- Rasterizer – transform vertexes/vectors into a grid. Fragments. break up to pixels and anti-alias
• Shader (Fragment processor) compute color for each pixel. Use textures if necessary (texture memory, mostly read)

• Write out to framebuffer (mostly write)

• Z-buffer for depth/visibility
GPGPUs

• Started when the vertex and fragment processors became generically programmable (originally to allow more advanced shading and lighting calculations)

• By having generic use can adapt to different workloads, some having more vertex operations and some more fragment
# Graphics vs Programmable Use

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Vertex Processing</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygon</td>
<td>Polygon Setup</td>
<td>Lists</td>
</tr>
<tr>
<td>Fragment</td>
<td>Per-pixel math</td>
<td>Data</td>
</tr>
<tr>
<td>Texture</td>
<td>Data fetch, Blending</td>
<td>Data</td>
</tr>
<tr>
<td>Image</td>
<td>Z-buffer, anti-alias</td>
<td>Data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIMD processing</th>
<th>SIMD Rasterization</th>
<th>Programmable SIMD</th>
<th>Data Fetch</th>
<th>Predicated Write</th>
</tr>
</thead>
</table>
Example for Shader 3.0, came out DirectX9

They are up to Pixel Shader 5.0 now
Shader 3.0 Programming – Vertex Processor

• 512 static / 65536 dynamic instructions

• Up to 32 temporary registers

• Simple flow control

• Texturing – texture data can be fetched during vertex operations
• Can do a four-wide SIMD MAD (multiply ADD) and a scalar op per cycle:
  – EXP, EXPP, LIT, LOGP (exponential)
  – RCP, RSQ (reciprocal, r-square-root)
  – SIN, COS (trig)
Shader 3.0 Programming – Fragment Processor

- 65536 static / 65536 dynamic instructions (but can time out if takes too long)
- Supports conditional branches and loops
- fp32 and fp16 internal precision
- Can do 4-wide MAD and 4-wide DP4 (dot product)