

ECE4893A/CS4803MPG:  
**MULTICORE AND GPU**  
PROGRAMMING  
FOR VIDEO GAMES



## **“Classic” GPGPU**

Prof. Aaron Lanterman



School of Electrical and Computer Engineering  
Georgia Institute of Technology

# “Classic” vs. “Modern” GPGPU

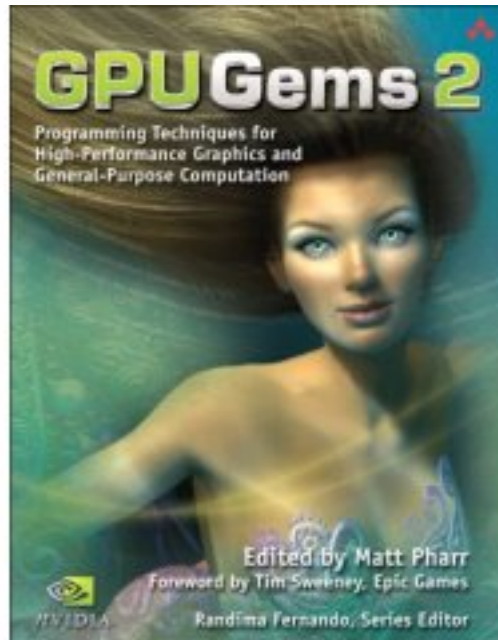
- “Classic GPGPU”
  - User must map their algorithm to a graphics framework not designed with general computation in mind
  - GPGPU implementations on the Playstation 3’s RSX and the Xbox 360’s Xenos will need to be programmed in this mindset
  - Many PC users will have older graphics cards
- “Modern GPGPU”
  - Runs on hardware (e.g., NVIDIA G80) specifically designed to be friendly to GPGPU computations
  - NVIDIA’s Compute Unified Device Architecture (CUDA) framework hides low-level details
  - OpenCL not tied to a specific vendor

# Typical classic GPGPU setup

- Load inputs into textures
  - GPU textures  $\leftrightarrow$  CPU arrays
  - Texture coordinates  $\leftrightarrow$  computational domain
- Draw a quadrangle
- Do computation in pixel shaders
  - Older GPUs typically have more pixel shader units than vertex shader units
  - Older GPUs can't do texture fetch in vertex shaders
  - GPU pixel shaders  $\leftrightarrow$  CPU inner loops
  - Texture sample (tex2D)  $\leftrightarrow$  CPU memory read
- Render output as pixels in the quadrangle
  - Render-to-texture  $\leftrightarrow$  CPU memory write
  - Vertex coordinates  $\leftrightarrow$  computational range

# Reference

- A lot of this discussion is inspired by various chapters in GPU Gems 2
  - Excellent reference for “classic GPGPU”
  - GPU Gems 3 focuses more on “modern GPGPU” solutions



# Scattering vs. gathering on a typical GPU

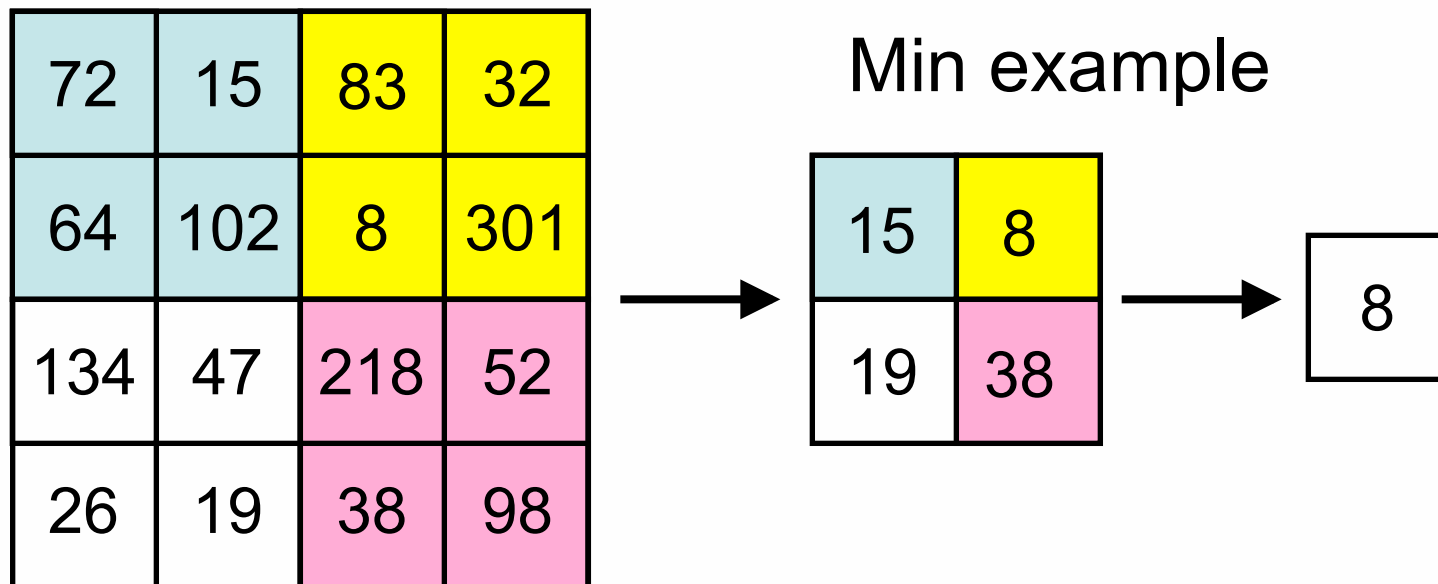
- Gather:
  - Indirect read
  - $a = x[i]$
- Texture fetch
- $\text{tex2D}(x, \dots)$
- Scatter:
  - Indirect write
  - $a[i] = x$
- No arbitrary texture writes on the GPU!
  - Can do “ordered writes” with “render to texture”

# Mapping

- Apply some function to each element in parallel
- Store input values as texels in a texture
- Draw quadrangle with as many pixels as texels in input
- Pixel shader just computes the function
- Output values in pixels of rendered quadrangle
  - Use “render-to-texture” to use mapped values in another stage

# Reduction

- Associative operations
  - Ex: sums, products, min, max
- Do multiple passes, rendering to smaller textures with each pass



- Could do more than four per pixel
  - Depending on how many “texture fetches” are allowed
  - Less passes, but increased time per pass

# Flow control is tricky

- Most older GPUs don't "really" support branching
- Loops typically "unrolled" by the compiler
- Predicated branching:
  - Compute both parts of the branch
  - Use results from only one branch
- On some GPUs, vertex shaders support branching but pixel shaders don't
- Even newer GPUs that directly support branching may give you a significant performance hit
  - Parallel execution units may be restricted to executing only one branch at a time (locality is important)



# Static branch resolution

- Instead of branching in the pixel shader, execute different pixel shaders on different output quadrangles
- Ex: boundary conditions in PDE

# Skipping unnecessary work with Z-cull

- Setup
  - Write 0 to z-buffer of pixels where you want to skip computation
  - Write 1 to z-buffer of pixels where you do
- Feed pixel shader doing the computation z-buffer value of 0.5
- Ex: landlocked cells in fluid simulation
- Warning: GPU may do z-culling at coarser resolution than the pixels
  - Will skip shading only if all pixels in a region fail the depth test
- Can do similar tricks with alpha stenciling

# Applications tailor-made for GPGPU

- Applications with high compute-to-communication ratios
- Partial differential equations
- Cellular automata
  - Sim City!
- Finance
  - Options pricing
- Linear algebra
  - Chapter 44 of GPU Gems 2
  - There are clever ways to handle banded matrices, sparse matrices, etc.

# GPGPU applications in games

- Collision detection
- Physics
  - Rigid bodies
  - Fluids, clouds, smoke, cloth
- Particle systems
- Line of sight calculations for AI?
- Aside:
  - AGEIA marketed Physics Processing Unit (PPU) to accelerate their “PhysX” SDK
    - But now owned by NVIDIA
  - Havok FX can exploit NVIDIA and ATI GPUs

# General advice

- Quite often better to store long 1D arrays as “wrapped” 2D arrays
  - 1D arrays limited in length
  - GPUs seem to be faster at handling 2D textures than 1D
- Pre-compute constants on the CPU
- Pre-compute low-dimensional functions and store them as textures
  - GPU will naturally do an interpolated lookup

# Limitations of GPGPU

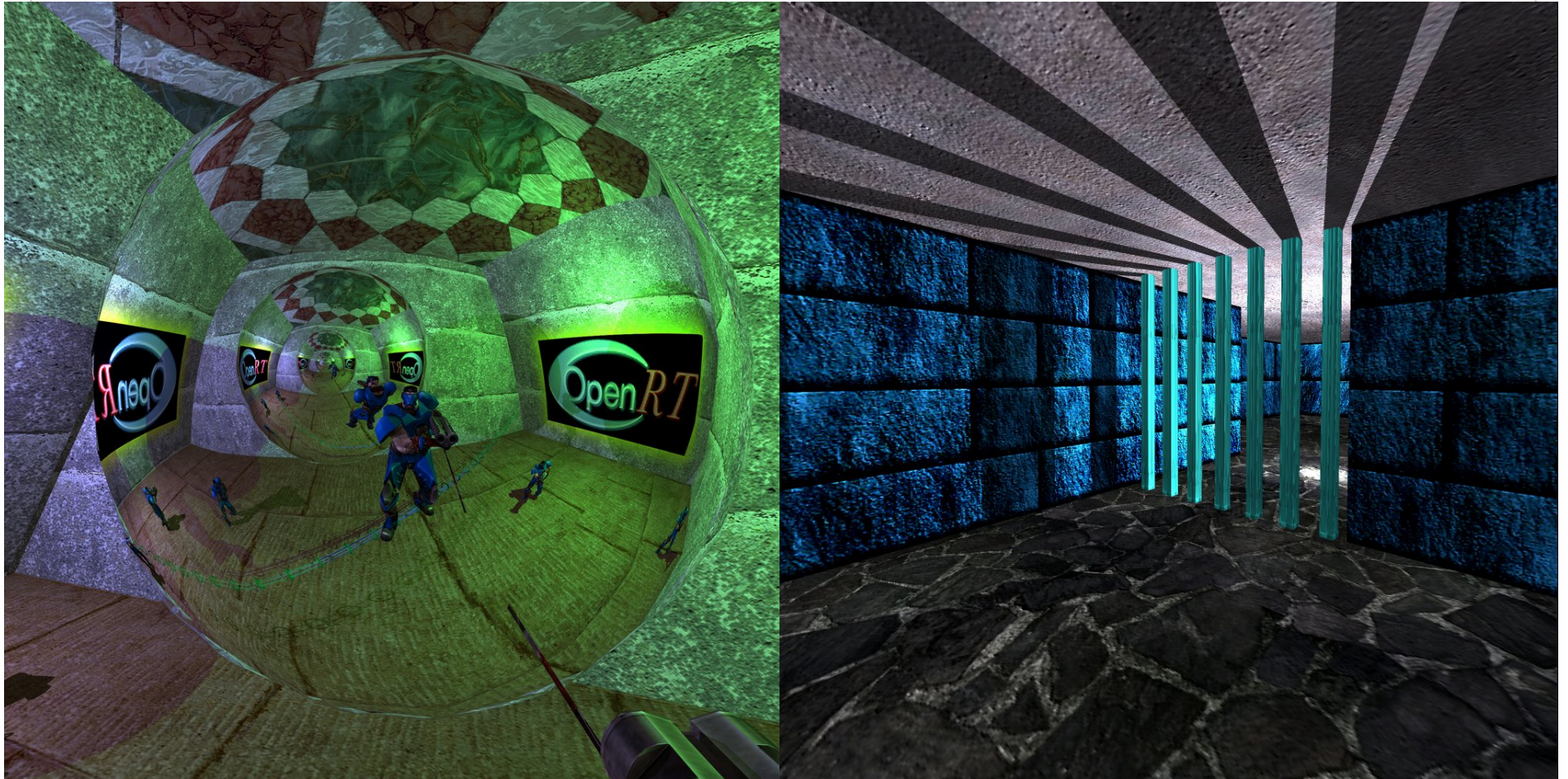
- Usually 32-bit floats (used to be worse!)
  - Some newer high-end cards support 64-bit floats
- May not have integers
  - Can be a problem with precise texture accesses
- May not have bitwise operations
- Not-so-great with dynamic data structures (queues, stacks, trees, etc.)
  - But lots of clever people have come up with tricks
- Think outside the box: strange algorithms that might be silly on a CPU might map well to a GPU

# Some successful GPGPU applications

- Computed Tomography (CT)
- MRI (application of FFTs)
- Phase unwrapping for Synthetic Aperture Radar (SAR)
  - 35x speedup obtained by Peter Karasev, Dan Campbell, and Mark Richards
- Data mining
- Raytracing
- GPU-accelerated Photoshop!



# Quake 3: Raytraced



<http://www.youtube.com/watch?v=bpNZt3yDXno>

Images from [graphics.cs.uni-sb.de/~sidapohl/egoshooter](http://graphics.cs.uni-sb.de/~sidapohl/egoshooter)