Motivation: Computational Power

- GPUs are fast...
  - 3 GHz Pentium4 theoretical: 6 GFLOPS, 5.96 GB/sec peak
  - GeForceFX 5900 observed: 20 GFLOPs, 25.3 GB/sec peak
- GPUs are getting faster, faster
  - CPUs: annual growth $\times 1.5 \rightarrow$ decade growth $\times 60$
  - GPUs: annual growth $\times 2.0 \rightarrow$ decade growth $\times 1000$

An Aside: Computational Power

- Why are GPUs getting faster so fast?
  - Arithmetic intensity: the specialized nature of GPUs makes it easier to use additional transistors for computation not cache
  - Economics: multi-billion dollar video game market is a pressure cooker that drives innovation

Motivation: Flexible and precise

- Modern GPUs are deeply programmable
  - Programmable pixel, vertex, video engines
  - Solidifying high-level language support
- Modern GPUs support high precision
  - 32 bit floating point throughout the pipeline
  - High enough for many (not all) applications
Motivation: The Potential of GPGPU
- The power and flexibility of GPUs makes them an attractive platform for general-purpose computation
- Example applications range from in-game physics simulation to conventional computational science
- Goal: make the inexpensive power of the GPU available to developers as a sort of computational coprocessor

The Problem: Difficult To Use
- GPUs designed for and driven by video games
  - Programming model is unusual & tied to computer graphics
  - Programming environment is tightly constrained
- Underlying architectures are:
  - Inherently parallel
  - Rapidly evolving (even in basic feature set!)
  - Largely secret
- Can’t simply “port” code written for the CPU!

High-level Shading Languages
- Cg, HLSL, & GLSLang
  - Cg:
  - HLSL:
  - CUDA:

CGC & FXC
- HLSL and Cg are semantically 99% identical
  - Same language, two names
- Command line compilers
  - Microsoft’s FXC.exe
    - Included in DX9 SDK
    - Compiles to DirectX vertex and pixel shader assembly only
    - Generates ATI preferred assembly
      - nvidia /Tps_2_0 myshader.cg
  - NVIDIA’s CGC.exe
    - Compiles to everything
    - Generates NV preferred assembly by default
      - nvidia /profile ps_2_0 myshader.hlsl
    - Can generate very different assembly!
  - Typically FXC does better for ATI, CGC for NVIDIA

GPGPU Languages
- Why do we want them?
  - Make programming GPUs easier!
    - Don’t need to know OpenGL, DirectX, or ATI/NV extensions
    - Simplify common operations
    - Focus on the algorithm, not on the implementation
- Sh: University of Waterloo
  - http://sh.blameforthis.net
  - http://www.cgl.uwaterloo.ca
- Brook: Stanford University
- CUDA
  - Nvidia
Sh Features

- Implemented as C++ library
  - Use C++ modularity, type, and scope constructs
  - Use C++ to metaprogram shaders and kernels
  - Use C++ to sequence stream operations

- Operations can run on
  - GPU in JIT compiled mode
  - GPU in immediate mode
  - CPU in JIT compiled mode

- Can be used
  - To define shaders
  - To define stream kernels

- No glue code
  - To set up a parameter, just declare it and use it
  - To set up a texture, just declare it and use it

- Memory management
  - Automatically uses pbuffers and/or uberbuffers
  - Texture and stream manipulation
  - Programs can use data texture data

- Program manipulation
  - Interception
  - Source rewriting/conversion
  - Program specialization
  - Program composition
  - Program serialization
  - Interface adaptation

Free and Open Source

http://libsh.sourceforge.net

Brook: general purpose streaming language

- stream programming model
  - enforce data parallel computing
  - streams
  - encourage arithmetic intensity
    - kernels

- C with stream extensions
- GPU = streaming coprocessor

Linear Algebra on GPUs

Jens Krüger
Technische Universität München

Why LA on GPUs?

1. Why should we care about Linear Algebra at all?

   Use LA to solve PDEs
   - solving PDEs can increase realism for VR, Education, Simulations, Games, ...

Why Linear Algebra on GPUs?

2. ... and why do it on the GPU?

   a) The GPU is a fast streaming processor
      - LA operations are easily "streamable"

   b) The result of the computation is already on the GPU and ready for display

Representation

- Vector representation
  - 2D textures best we can do
  - Per-fragment vs. per-vertex operations
  - High texture memory bandwidth
  - Read-write access, dependent fetches

.png
Representation (cont.)

Dense Matrix representation
- treat a dense matrix as a set of column vectors
- again, store these vectors as 2D textures

Operations (cont.)

- Vector-Vector Operations
  - Reduce operation for scalar products
  - Reduce m x n region in fragment shader

High-Level Algorithms

- Conjugate Gradient
  - Solve large linear systems
- Navier-Stokes
  - Fluid simulations

Simulating the world

- Simulate a wide variety of phenomena on GPUs
  - Anything we can describe with discrete PDEs or approximations of PDEs
  - Examples: boiling, reaction-diffusion, fluids, and clouds

Example: Boiling

- State = Temperature
- Three operations:
  - Diffusion, buoyancy, & latent heat
  - 7 passes in 2D, 9 per 3D slice (GeForce 4)
- Based on [Yanagita 1992]
Textures: C++ Arrays (CPU)

- Creating arrays on the CPU
- One option to hold data for GPGPU calculations

```c
float* data = malloc(sizeof(float) * texSize * texSize);
float* data = malloc(sizeof(float) * texSize * texSize);
```

Another option for rendering is to draw geometry and use that as the input data to the textures used more for advanced rendering effects.

Textures: OpenGL

- This gets complicated fast...
- Look at `glTexImage2D`
  - Texture target (next slide)
  - 0: don’t use any mipmap levels for this texture
  - Internal format (next slide)
  - `0`: turns off texture
  - Width and height of the texture
  - `GL_Float`: chooses the number of channels
  - GL_FLOAT / Float texture: using OpenGL’s internal representation of the values
  - Don’t need to specify texture data right now...
Textures: Formats

- On the GPU, we use floating point textures to store the data
- A variety of different so-called texture targets available

<table>
<thead>
<tr>
<th>Texture Target</th>
<th>Texture Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal texture format</td>
<td>Internal texture format</td>
</tr>
<tr>
<td>GPUs allow for the simultaneous processing of scalars, touples, triplets or four-touples of data</td>
<td></td>
</tr>
<tr>
<td>Internal texture format</td>
<td>Internal texture format</td>
</tr>
</tbody>
</table>

Mapping textures

- Later we update our data stored in textures by a rendering operation.
- To be able to control exactly which data elements we compute or access from texture memory, we need to choose a special projection that maps from the 3D world (world or model coordinate space) to the 2D screen (screen or display coordinate space), and additionally a 1:1 mapping between pixels (which we want to render to) and texels (which we access data from).

- The key to success here is to choose an orthogonal projection and a proper viewport that will enable a one to one mapping between geometry coordinates.

- ATI warning ... here is where you need to specify ATI extensions

Using Textures as Render Targets

- The traditional end point of every rendering operation is the frame buffer, a special chunk of graphics memory from which the image that appears on the display is read.
- Problem: the data will always be clamped to the range of [0/255; 255/255] once it reaches the framebuffer.

- What to do?

- Cumbersome arithmetic that maps the sign-mantissa-exponent data format of an IEEE 32-bit floating point value into the four 8-bit channels ???
- OpenGL extension called EXT_framebuffer_object allows us to use an offscreen buffer as the target for rendering operations such as our vector calculations, providing full precision and preventing all the unwanted clamping issues. The community used abbreviation is FBO, short for framebuffer object.

- The key to success here is to choose an orthogonal projection and a proper viewport that will enable a one to one mapping between geometry coordinates.

- ATI warning ... here is where you need to specify ATI extensions

Frame Buffer Objects: (FBO)

- To use this extension and to turn off the traditional framebuffer and use an offscreen buffer (FBO) for our calculations, a few lines of code suffice. Note that binding FBO number 0 will restore the window-system specific framebuffer at any time.

- ATI warning ... here is where you need to specify ATI extensions

- The framebuffer object extension provides a very narrow interface to render to a texture. To use a texture as render target, we have to attach the texture to the FBO.

- Drawback: Textures are either read-only or write-only.

- To use this extension and to turn off the traditional framebuffer and use an offscreen buffer (FBO) for our calculations, a few lines of code suffice. Note that binding FBO number 0 will restore the window-system specific framebuffer at any time.

- ATI warning ... here is where you need to specify ATI extensions

- Drawback: Textures are either read-only or write-only.

Transferring data from CPU arrays to GPU textures

- To transfer data (like the two vectors dataX and dataY we created previously) to a texture, we have to bind the texture to a texture target and schedule the data for transfer with an OpenGL command.

- Again, not only method if you want to render other data.

Transferring data from GPU textures to CPU arrays

- Many times you want the actual values that you calculated back, there are 2 ways to do this...

- ATI warning ... here is where you need to specify ATI extensions

- Drawback: Textures are either read-only or write-only.
Transferring data from GPU textures to QUADS

- Other time you really just want to see the mess you created on the screen
- To do this you have to render a QUAD

Preparing the computational kernel setting up input textures/arrays

- Setting output arrays / textures
  - Defining the output array (the left side of the equation) is essentially the same operation like the one we discussed to transfer data to a texture already attached to our FBO. Simple pointer manipulation by means of GL calls is all we need. In other words, we simply redirect the output. If we did not do so yet, we attach the target texture to our FBO and use standard GL calls to use it as the render target.

Performing a computation

- Let us briefly recall what we did so far:
  - We enabled a 1:1 mapping between the target pixels, the texture coordinates and the geometry we are about to draw.
  - We also prepared a fragment shader we want to execute for each fragment.
  - All that remains to be done is: Render a “suitable geometry” that ensures that our fragment shader is executed for each data element we stored in the target texture.
  - In other words, we make sure that each data item is transformed uniquely into a fragment.
  - Given our projection and viewport settings, this is embarrassingly easy: All we need is a filled quad

Multiple rendering passes

- In a proper application, the result is typically used as input for a subsequent computation.
- On the GPU, this means we perform another rendering pass and bind different input and output textures, eventually a different kernel etc.
- The most important ingredient for this kind of multipass rendering is the ping pong technique.

The ping pong technique

- Ping pong is a technique to alternately use the output of a given rendering pass as input in the next one.
- This means that we swap the role of the two textures \( y_{\text{new}} \) and \( y_{\text{old}} \), since we do not need the values in \( y_{\text{old}} \) any more once the new values have been computed.
- There are three possible ways to implement this kind of data reuse (take a look at Simon Green’s FBO slides for additional material on this):

The ping pong technique

- During the computation, all we need to do now is to pass the correct value from these two tuples to the corresponding OpenGL calls, and to swap the two index variables after each pass:

```c
// code snippet
```