The Way of the GPU
(based on GPGPU SIGGRAPH Course)

CS535

Daniel G. Aliaga
Department of Computer Science
Purdue University
Computer Graphics Pipeline

- **Geometry**
  - Modeling Transformation: Transform into 3D *world* coordinate system
  - Lighting: Simulate illumination and reflectance
  - Viewing Transformation: Transform into 3D *camera* coordinate system
  - Clipping: Clip primitives outside camera’s view
  - Projection: Transform into 2D camera coordinate system
  - Scan Conversion: Draw pixels (incl. texturing, hidden surface…)

*(this is really from 20 years ago…)*
Today, we have GPUs...

(GPU = graphical processing unit)
Motivation: Computational Power

- Why are GPUs 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
The Problem: Difficult To Use

• GPUs designed for & driven by video games
  – Programming model unusual
  – Programming idioms tied to computer graphics
  – Programming environment tightly constrained

• Underlying architectures are:
  – Inherently parallel
  – Rapidly evolving (even in basic feature set!)
  – Largely secret

• Can’t simply “port” CPU code!
nVIDIA GPU

• GTX Titan X
  – 3072 cores @ 1GHz (i.e., mini processors)
  – 192 gigatexels/sec fill rate
  – 966 Gflops in matrix multiplication
  – 5120x3200 pixels
  – 250W power
  – 91C max GPU temp
  – $999
Modern GPU has more ALU’s

Figure 1-2. The GPU Devotes More Transistors to Data Processing
GPU Pipeline: Transform

- Vertex/Geometry processor (multiple in parallel)
  - Transform from “world space” to “image space”
  - Compute per-primitive and per-vertex lighting
GPU Pipeline: Rasterize
(typically not programmable)

• Rasterizer
  – Convert geometric rep. (vertex) to image rep. (fragment)
    • Fragment = image fragment
      – Pixel + associated data: color, depth, stencil, etc.
  – Interpolate per-vertex quantities across pixels

![Diagram of a triangle being rasterized into pixels]
GPU Pipeline: Shade

• Fragment processors (multiple in parallel)
  – Compute a color for each pixel
  – Optionally read colors from textures (images)
GPU Programming Languages

• Many options!
  – A while ago: “Renderman”
  – cG (from NVIDIA)
  – GLSL (GL shading Language)
  – CUDA (more general than graphics)...

• Lets focus first on the concept, later on the language specifics...
Mapping Parallel Computational Concepts to GPUs
Mapping Parallel Computational Concepts to GPUs

- GPUs are designed for graphics
  - Highly parallel tasks
- GPUs process *independent* vertices & fragments
  - Temporary registers are zeroed
  - No shared or static data
  - No read-modify-write buffers
- Data-parallel processing
  - GPUs architecture is ALU-heavy
    - Multiple vertex & pixel pipelines, multiple ALUs per pipe
  - Hide memory latency (with more computation)
Example: Simulation Grid

• Common GPGPU computation style
  – Textures represent computational grids = streams

• Many computations map to grids
  – Matrix algebra
  – Image & Volume processing
  – Physically-based simulation
  – Global Illumination
    • ray tracing, photon mapping, radiosity

• Non-grid streams can be mapped to grids
Typical Stream Computation

• Grid Simulation algorithm
  – Made up of steps
  – Each step updates entire grid
  – Must complete before next step can begin

• Grid is a stream, steps are kernels
  – Kernel applied to each stream element
e.g.: Scatter vs. Gather

• Grid communication
  – Grid cells share information
Vertex Processor

- Fully programmable (SIMD / MIMD)
- Processes 4-vectors (RGBA / XYZW)
- Capable of scatter but not gather
  - Can change the location of current vertex
  - Cannot read info from other vertices
  - Can only read a small constant memory
- Latest GPUs: Vertex Texture Fetch
  - Random access memory for vertices
  - \(\approx\)Gather (But not from the vertex stream itself)
Fragment Processor

- Fully programmable (SIMD)
- Processes 4-component vectors (RGBA / XYZW)
- Random access memory read (textures)
- Capable of gather but not scatter
  - RAM read (texture fetch), but no RAM write
  - Output address fixed to a specific pixel
- Typically more useful than vertex processor
  - More fragment pipelines than vertex pipelines
  - Direct output (fragment processor is at end of pipeline)
GPU Simulation Overview

• A Simulation:
  – Its algorithm steps are fragment programs
    • Called Computational *kernels*
  – Current state is stored in textures
  – Feedback via “render to texture”

• Question:
  – How do we invoke computation?
Invoking Computation

• Must invoke computation at each pixel
  – Just draw geometry!
  – Most common GPGPU invocation is a full-screen quad

• Other Useful Analogies
  – Rasterization = Kernel Invocation
  – Texture Coordinates = Computational Domain
  – Vertex Coordinates = Computational Range
Typical “Grid” Computation

• Initialize “view” (so that pixels:texels::1:1)
  
  ```gl
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0, 1, 0, 1, 0, 1);
  glViewport(0, 0, outTexResX, outTexResY);
  ```

• For each algorithm step:
  
  – Activate render-to-texture
  – Setup input textures, fragment program
  – Draw a full-screen quad (1x1)
Example: N-Body Simulation

- Brute force 😞
- \( N = 8192 \) bodies
- \( N^2 \) gravity computations

- 64M force comps. / frame
- \( \sim 25 \) flops per force
- 10.5 fps

- 17+ GFLOPs sustained in this example
Computing Gravitational Forces

• Each body attracts all other bodies
  – $N$ bodies, so $N^2$ forces

• Draw into an $N \times N$ buffer
  – Pixel $(i,j)$ computes force between bodies $i$ and $j$
  – Very simple fragment program
    • More than $N=2048$ bodies is tricky
    • Why?
Computing Gravitational Forces

N-body force Texture

Body Position Texture

\[ F(i,j) = \frac{gM_i M_j}{r(i,j)^2}, \]

\[ r(i,j) = |\text{pos}(i) - \text{pos}(j)| \]

Force is proportional to the inverse square of the distance between bodies
Computing Gravitational Forces

```cpp
float4 force(float2 ij : WPOS,
             uniform sampler2D pos) : COLOR0
{
    // Pos texture is 2D, not 1D, so we need to
    // convert body index into 2D coords for pos tex
    float4 iCoords = getBodyCoords(ij);
    float4 iPosMass = texture2D(pos, iCoords.xy);
    float4 jPosMass = texture2D(pos, iCoords.zw);
    float3 dir = iPos.xyz - jPos.xyz;
    float r2 = dot(dir, dir);
    dir = normalize(dir);
    return dir * g * iPosMass.w * jPosMass.w / r2;
}
```
Computing Total Force

- Have: array of \((i, j)\) forces
- Need: total force on each particle \(i\)
  - Sum of each column of the force array
- Can do all \(N\) columns in parallel

This is called a *Parallel Reduction*
Parallel Reductions

1D parallel reduction:

sum N columns or rows in parallel

add two halves together repeatedly...

Until we’re left with a single row of texels

\[ N \times N + N \times \frac{N}{2} + N \times \frac{N}{4} + N \times 1 \]

Requires \( \log_2 N \) steps
Update Positions and Velocities

• Now we have a 1-D array of total forces
  – One per body

• Update Velocity
  – \( u(i, t+dt) = u(i, t) + F_{total}(i) * dt \)
  – Simple pixel shader reads previous velocity and force textures, creates new velocity texture

• Update Position
  – \( x(i, t+dt) = x(i, t) + u(i, t) * dt \)
  – Simple pixel shader reads previous position and velocity textures, creates new position texture
Linear Algebra on GPUs

• Use linear algebra for VR, education, simulation, games and much more!
Representation

Vector representation

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

Dense Matrix representation

- treat a dense matrix as a set of column vectors
- again, store these vectors as 2D textures
Representation (cont.)
Banded Sparse Matrix representation
– treat a banded matrix as a set of **diagonal** vectors
Representation (cont.)

Banded Sparse Matrix representation

- combine opposing vectors to save space
Operations

• Vector-Vector Operations
  – Reduced to 2D texture operations
  – Coded in vertex/fragment shaders

• Example: Vector1 + Vector2 → Vector3
Operations (cont.)

• Vector-Vector Operations
  – Reduce operation for scalar products

![Diagram showing reduce operation for scalar products in a vector-vector context, with an original texture and two passes.](image)
The “single float” on GPUs

Some operations generate single float values e.g. reduce

Read-back to main-mem is slow

→ Keep single floats on the GPU as 1x1 textures
Operations (cont.)

In depth example: Vector / Banded-Matrix Multiplication

\[ A \times b = x \]
Example (cont.)

Vector / Banded-Matrix Multiplication

\[ \mathbf{A} \mathbf{x} = \mathbf{b} \]
Example (cont.)

Compute the result in 2 Passes

Pass 1:

- **A**
- **b**
- **X**

[Diagram showing matrix multiplication process]

```
multiply
```
Example (cont.)

Compute the result in 2 Passes

Pass 2:

A

b

\[ \begin{array}{ccc}
  \text{multiply} \\
  \text{shift} \\
  \end{array} \]

b'

x

\[ \begin{array}{ccc}
multiply \\
\end{array} \]
Random sparse matrix representation

- Textures do not work
  - Splitting yields highly fragmented textures
  - Difficult to find optimal partitions
- Idea: encode only non-zero entries in vertex arrays

\[
\begin{align*}
\text{Matrix} \cdot \text{Vector} &= \text{Result} \\
\text{Row Index as Position} & \rightarrow \text{Values as TexCoord 0} \\
\text{Column as TexCoord 1-4} & \rightarrow \text{Vertex Array 1} & \rightarrow \text{Vertex Array 2} & \rightarrow \text{Vector} \\
& \rightarrow \text{Result}
\end{align*}
\]
Sorting

• Given an unordered list of elements, produce list ordered by key value
  – Kernel: compare and swap
• GPUs constrained programming environment limits viable algorithms
  – Based on “Bitonic merge sort” [Batcher 68]
  – Periodic balanced sorting networks [Dowd 89]
Bitonic Merge Sort Overview

• Repeatedly build bitonic lists and then sort them
  – Bitonic list is two monotonic lists concatenated together, one increasing and one decreasing.
    • List A: (3, 4, 7, 8)  monotonically increasing
    • List B: (6, 5, 2, 1)  monotonically decreasing
    • List AB: (3, 4, 7, 8, 6, 5, 2, 1)  bitonic
Bitonic Merge Sort

3
7
4
8
6
2
1
5

8x monotonic lists:  (3) (7) (4) (8) (6) (2) (1) (5)
4x bitonic lists:   (3,7) (4,8) (6,2) (1,5)
Bitonic Merge Sort

Sort the bitonic lists
Bitonic Merge Sort

4x monotonic lists: (3,7) (8,4) (2,6) (5,1)
2x bitonic lists: (3,7,8,4) (2,6,5,1)
Bitonic Merge Sort

Sort the bitonic lists
Bitonic Merge Sort

Sort the bitonic lists
Bitonic Merge Sort

Sort the bitonic lists
Bitonic Merge Sort

2x monotonic lists: (3,4,7,8) (6,5,2,1)
1x bitonic list: (3,4,7,8, 6,5,2,1)
Sort the bitonic list
Bitonic Merge Sort

Sort the bitonic list
Sort the bitonic list
Bitonic Merge Sort

Sort the bitonic list
Sort the bitonic list
Bitonic Merge Sort

```
3 3 3 3 3 3 2 1
7 7 4 4 4 1 2
4 8 8 7 2 3 3
8 4 7 8 1 4 4
6 2 5 6 6 6 5
2 6 6 5 5 5 6
1 5 2 2 7 7 7
5 1 1 1 8 8 8
Done!
```
Bitonic Merge Sort Summary

• Separate rendering pass for each set of swaps
  – $O(\log^2 n)$ passes
  – Each pass performs $n$ compare/swaps
  – Total compare/swaps: $O(n \log^2 n)$
    • Limitations of GPU cost us factor of $\log n$ over best CPU-based sorting algorithms
Database and Streaming on GPUs

- Utilize graphics processors for fast computation of common database operations
  - Conjunctive selections
  - Aggregations
  - Semi-linear queries

- Essential components
Basic DB Operations

Basic SQL query

» Select  A
» From T
» Where C

A= attributes or aggregations (SUM, COUNT, MAX etc)
T=relational table
C= Boolean Combination of Predicates (using operators AND, OR, NOT)
Database Operations

• Predicates
  – $a_i \text{ op } \text{constant or } a_i \text{ op } a_j$
  – op: $<,>,<=,>=,!=, =, \text{TRUE, FALSE}$

• Boolean combinations
  – Conjunctive Normal Form (CNF)

• Aggregations
  – COUNT, SUM, MAX, MEDIAN, AVG
Predicate Evaluation

• $a_i \text{ op constant (d)}$
  – Copy the attribute values $a_i$ into depth buffer
  – Specify the comparison operation used in the depth test
  – Draw a screen filling quad at depth $d$ and perform the depth test
If \( a_i \ op \ d \) pass fragment
Else reject fragment
Predicate Evaluation

Relational Query

Time (in msec)

<table>
<thead>
<tr>
<th>NV40</th>
<th>NV35</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>90K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>160K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>250K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>360K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>490K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>640K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>810K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1M</td>
</tr>
</tbody>
</table>

Number of Records
Predicate Evaluation

- $a_i \text{ op } a_j$
  - Equivalent to $(a_i - a_j) \text{ op } 0$

- Semi-linear queries
  - Defined as linear combination of attribute values compared against a constant
  - Linear combination is computed as a dot product of two vectors
    \[ \sum s_i \cdot a_i \]
  - Utilize the vector processing capabilities of GPUs
Semi-linear Query

Semi-linear Queries

Time (in msec)

NV40
NV35
CPU Time

Number of Records

90K 160K 250K 360K 490K 640K 810K 1M
Boolean Combination

• CNF:
  – \((A_1 \text{ AND } A_2 \text{ AND } \ldots \text{ AND } A_k)\) where
    \[A_i = (B_{i1} \text{ OR } B_{i2} \text{ OR } \ldots \text{ OR } B_{imi})\]

• Performed using stencil test recursively
  – \(C_1 = (\text{TRUE AND } A_1) = A_1\)
  – \(C_i = (A_1 \text{ AND } A_2 \text{ AND } \ldots \text{ AND } A_i) = (C_{i-1} \text{ AND } A_i)\)

• Different stencil values are used to code the outcome of \(C_i\)
  – Positive stencil values — pass predicate evaluation
  – Zero — fail predicate evaluation
A_1 \text{ AND } A_2
$A_1$ AND $A_2$
A₁ AND A₂

Stencil value = 0

Stencil value = 1

A₁
A₁ AND A₂
$A_1 \text{ AND } A_2$
$A_1 \text{ AND } A_2$
Aggregations

- COUNT, MAX, MIN, SUM, AVG
  - These can be efficiently implemented as well!
Geometric Computations

• Well studied
  – Computer graphics, computational geometry etc.

• Widely used in games, simulations, virtual reality applications
GPUs: Geometric Computations

• Used for geometric applications
  – Minkowski sums [Kim et al. 02]
  – CSG rendering [Goldfeather et al. 89, Rossignac et al. 90]
  – Voronoi computation [Hoff et al. 01, 02, Sud et al. 04]
  – Isosurface computation [Pascucci 04]
  – Map simplification [Mustafa et al. 01]
Collision Detection Pointers:

• Overview
• Collision Detection: CULLIDE
• Inter- and Intra-Object Collision Detection: Quick-CULLIDE
• Reliable Collision Detection: FAR
• Analysis
Geometry processing on GPUs

• so far: GPGPU limited to texture output

• new APIs allow geometry generation on GPU
Examples

Fluid Simulation

3D Smoke & Fire

Particles

Water Simulation

Grid displacement

3D Water Surfaces
Examples

Fluid Simulation

Particles

3D Smoke & Fire

Grid displacement

3D Water Surfaces

Point Decompression

Point Rendering

Water Simulation

Point Compression
High Level Shading Languages

- Cg, HLSL, & OpenGL Shading Language
  - Cg:
  - HLSL:
  - OpenGL Shading Language:
GPGPU Languages

• Why do 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
Compilers: CGC & FXC

- HLSL and Cg are syntactically almost identical
  - Exception: Cg 1.3 allows shader “interfaces”, unsized arrays
- Command line compilers
  - Microsoft’s FXC.exe
    - Compiles to DirectX vertex and pixel shader assembly only
    - fxc /Tps_2_0 myshader.hlsl
  - NVIDIA’s CGC.exe
    - Compiles to everything
    - cgc -profile ps_2_0 myshader.cg
  - Can generate very different assembly!
    - Driver will recompile code
  - Compliance may vary
Babelshader

http://graphics.stanford.edu/~danielrh/babelshader.html

- Converts between DirectX pixel shaders and OpenGL shaders
- Allows OpenGL programs to use DirectX HLSL compilers to compile programs into ARB or fp30 assembly.
- Enables fair benchmarking competition between the HLSL compiler and the Cg compiler on the same platform with the same demo and driver.

```plaintext
texld r9, t5, s4 ; fetch
dp2add r0.a, r0, r0, c0.z
rsq r0.a, r0.a
rcp r0.b, r0.a
mad r3, r9, c0.w, -c0.z
mad r6, r3, c4.r, r0
mad r3, r9, c0.w, -c0.z
mad r7, r3, c4.g, r0
mad r1.a, r5.r, c3.x, c3.y
dp3 r4.a, t4, t4
rsq r4.a, r4.a ;1/view
mul r4, t4, r4.a ;norm ; reflection vector
mul r2.rgb, r0.x, t1
mad r2.rgb, r0.y, t2, r2
mad r2.rgb, r0.z, t3, r2 ;
transform bump map normal
```

```plaintext
TEX r9, t5, texture[4], 2D;
MAD r0.w, R0.x, R0.y, -c0.z;
MAD R0.a, R0.y, R0.y, R0.w;
RSQ R0.a, R0.a;
RCP R0.b, R0.a;
MAD r3, r9, c0.w, -c0.z;
MAD r6, r3, c4.r, r0;
MAD r3, r9, c0.w, -c0.z;
MAD r7, r3, c4.g, r0;
MAD r1.a, r5.r, c3.x, c3.y;
DP3 r4.a, t4, t4;
RSQ r4.a, r4.a;
; 1/ view
MUL r4, t4, r4.a;
MUL r2.rgb, r0.x, t1;
MAD r2.rgb, r0.y, t2, r2;
MAD r2.rgb, r0.z, t3, r2;
```
‘printf’ Debugging

• MOV suspect register to output
  – Comment out anything else writing to output
  – Scale and bias as needed
• Recompile
• Display/readback frame buffer
• Check values
• Repeat until error is (hopefully) found
‘printf’ Debugging Examples
‘printf’ Debugging Examples
‘printf’ Debugging Examples
‘printf’ Debugging Examples
Ideal Fragment Program Debugger

• Automate ‘printf’ debugging
• Intuitive and easy to use interface
• Features over performance
  – Debuggers don’t need to be fast
    • Should still be interactive
• Easy to add to existing apps
  – And remove
• Touch as little GPU state as possible
• Report actual hardware values
  – No software simulations!
Data Parallel Computing

- Instruction-Level Parallelism
- Data-Level Parallelism
A really naïve shader

frag2frame Smooth(vert2frag IN, uniform samplerRECT Source : texunit0, uniform samplerRECT Operator : texunit1, uniform samplerRECT Boundary : texunit2, uniform float4 params)
{
    frag2frame OUT;

    float2 center = IN.TexCoord.xy;
    float4 U = f4texRECT(Source, center);

    // Calculate Red-Black (odd-even) masks
    float2 intpart;
    float2 place = floor(1.0f - modf(round(center + float2(0.5f, 0.5f)) / 2.0f, intpart));
    float2 mask = float2((1.0f-place.x) * (1.0f-place.y), place.x * place.y);

    if (((mask.x + mask.y) && params.y) || !(mask.x + mask.y) && !params.y))
    {
        float2 offset = float2(params.x*center.x - 0.5f*(params.x-1.0f), params.x*center.y - 0.5f*(params.x-1.0f));
        ...
        float4 neighbor = float4(center.x - 1.0f, center.x + 1.0f, center.y - 1.0f, center.y + 1.0f);
        float central = -2.0f*(O.x + O.y);

        float poisson = ((params.x*params.x)*U.z + (-O.x * f1texRECT(Source, float2(neighbor.x, center.y)) +
                      -O.y * f1texRECT(Source, float2(neighbor.y, center.x)) +
                      -O.z * f1texRECT(Source, float2(neighbor.w, center.x))) / O.w;

        OUT.COL.x = poisson;
    }
    ...
    return OUT;
}
A really naïve shader

```cshar
frag2frame Smooth(vert2frag IN, uniform samplerRECT Source : texunit0, uniform samplerRECT Operator : texunit1, uniform samplerRECT Boundary : texunit2, uniform float4 params)
{
    frag2frame OUT;

    float2 center = IN.TexCoord0.xy;
    float4 U = f4texRECT(Source, center);

    // Calculate Red-Black (odd-even) masks
    float2 intpart;
    float2 place = floor(1.0f - modf(round(center + float2(0.5f, 0.5f)) / 2.0f, intpart));
    float2 mask = float2((1.0f - place.x) * (1.0f - place.y), place.x * place.y);

    if (((mask.x + mask.y) && params.y) || (!(mask.x + mask.y) && !params.y))
    {
        float2 offset = float2(params.x*center.x - 0.5f*(params.x-1.0f), params.x*center.y - 0.5f*(params.x-1.0f));
        ...
        float4 neighbor = float4(center.x - 1.0f, center.x + 1.0f, center.y - 1.0f, center.y + 1.0f);
        float central = -2.0f*(O.x + O.y);

        float poisson = ((params.x*params.x)*U.z + (-O.x * f1texRECT(Source, float2(neighbor.x, center.y)) +
        -O.x * f1texRECT(Source, float2(neighbor.y, center.x)) +
        -O.y * f1texRECT(Source, float2(center.x, neighbor.z)) +
        -O.z * f1texRECT(Source, float2(center.x, neighbor.w)))) / O.w;

        OUT.COL.x = poisson;
    }
    ...
    return OUT;
}
```
float2 offset = float2\(\text{params}.x*\text{center}.x - 0.5f*(\text{params}.x-1.0f),\)
\text{params}.x*\text{center}.y - 0.5f*(\text{params}.x-1.0f)\);

float4 neighbor = float4(\text{center}.x - 1.0f,\)
\text{center}.x + 1.0f,\)
\text{center}.y - 1.0f,\)
\text{center}.y + 1.0f);
float2 offset = center.xy - 0.5f;
offset = offset * params.xx + 0.5f;  // MADR is cool too - one
  // cycle, two flops

float4 neighbor = center.xxyy + float4(-1.0f,1.0f,-1.0f,1.0f);
Data-Level Parallelism

• Pack scalar data into RGBA in texture memory
Computational Frequency

• Think of your CPU program and your vertex and fragment programs as different levels of nested looping.

```plaintext
... 
foreach tri in triangles {
    // run the vertex program on each vertex
    v1 = process_vertex(tri.vertex1);
    v2 = process_vertex(tri.vertex2);
    v3 = process_vertex(tri.vertex2);

    // assemble the vertices into a triangle
    assembled_triangle = setup_tri(v1, v2, v3);

    // rasterize the assembled triangle into [0..many] fragments
    fragments = rasterize(assembled_triangle);

    // run the fragment program on each fragment
    foreach frag in fragments {
        outbuffer[frag.position] = process_fragment(frag);
    }
}
...
Computational Frequency

• Branches
  – Avoid these, especially in the inner loop – i.e., the fragment program.
Computational Frequency

• Static branch resolution
  – write several variants of each fragment program to handle boundary cases
  – eliminates conditionals in the fragment program
  – equivalent to avoiding CPU inner-loop branching
Computational Frequency

• Dynamic branching
  – Use only when needed
    • Good perf requires spatial coherence in branching
Computational Frequency

- Precompute
- Precompute
- Precompute
Computational Frequency

• Precompute texture coordinates
  – Take advantage of under-utilized hardware
    • vertex processor
    • rasterizer
  – Reduce instruction count at the per-fragment level
  – Avoid lookups being treated as texture indirections
Memory Hierarchy

- CPU and GPU Memory Hierarchy

- Disk
- CPU Main Memory
- CPU Caches
- CPU Registers
- GPU Video Memory
- GPU Caches
- GPU Constant Registers
- GPU Temporary Registers
CPU Memory Model

- At any program point
  - Allocate/free local or global memory
  - Random memory access
    - Registers
      - Read/write
    - Local memory
      - Read/write to stack
    - Global memory
      - Read/write to heap
    - Disk
      - Read/write to disk
GPU Memory Model

• Much more restricted memory access
  – Allocate/free memory only before computation
  – Limited memory access during computation (kernel)
    • Registers
      – Read/write
    • Local memory
      – Does not exist
    • Global memory
      – Read-only during computation
      – Write-only at end of computation (pre-computed address)
• Disk access
GPU Memory Model

- Where is GPU Data Stored?
  - Vertex buffer
  - Frame buffer
  - Texture
Glift: http://graphics.cs.ucdavis.edu/~lefohn/work/glift/

• Goal
  – A template for simple creation and efficient use of random-access GPU data structures for graphics and GPGPU programming

• Contributions
  – Abstraction for GPU data structures
  – The template library