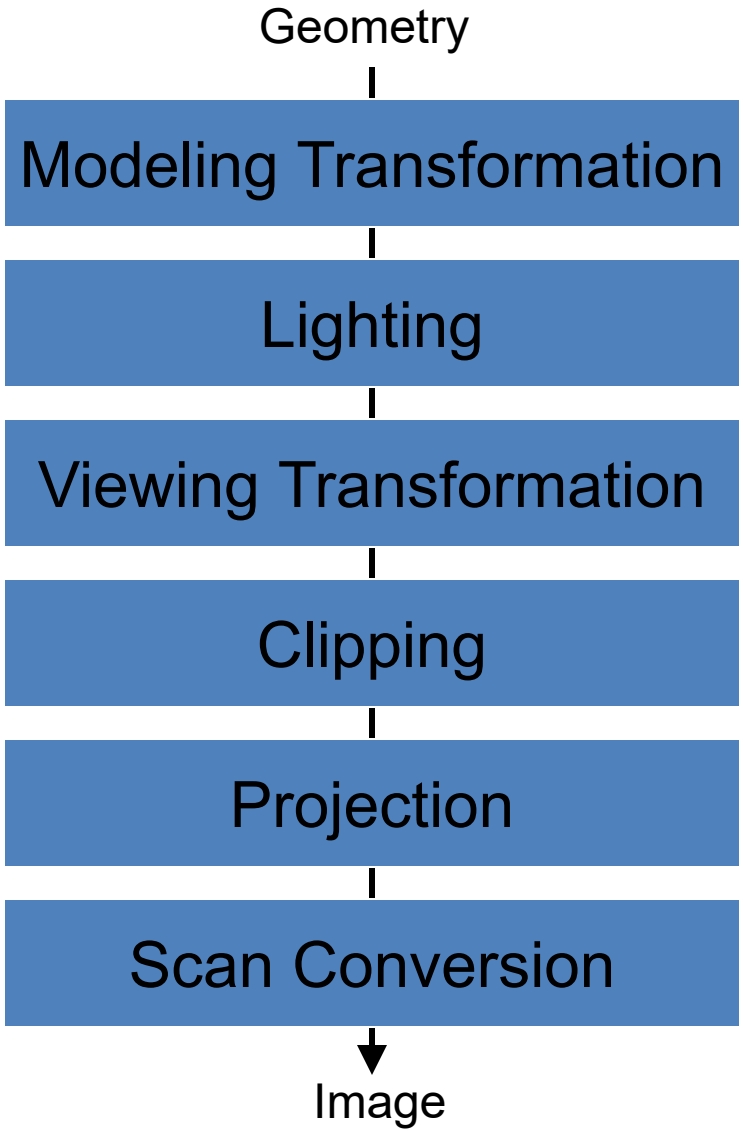
The Way of the GPU

(based on GPGPU SIGGRAPH Course)

CS535

Daniel G. Aliaga
Department of Computer Science
Purdue University

Computer Graphics Pipeline



Transform into 3D *world* coordinate system

Simulate illumination and reflectance

Transform into 3D *camera* coordinate system

Clip primitives outside camera's view

Transform into 2D camera coordinate system

Draw pixels (incl. texturing, hidden surface…)

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

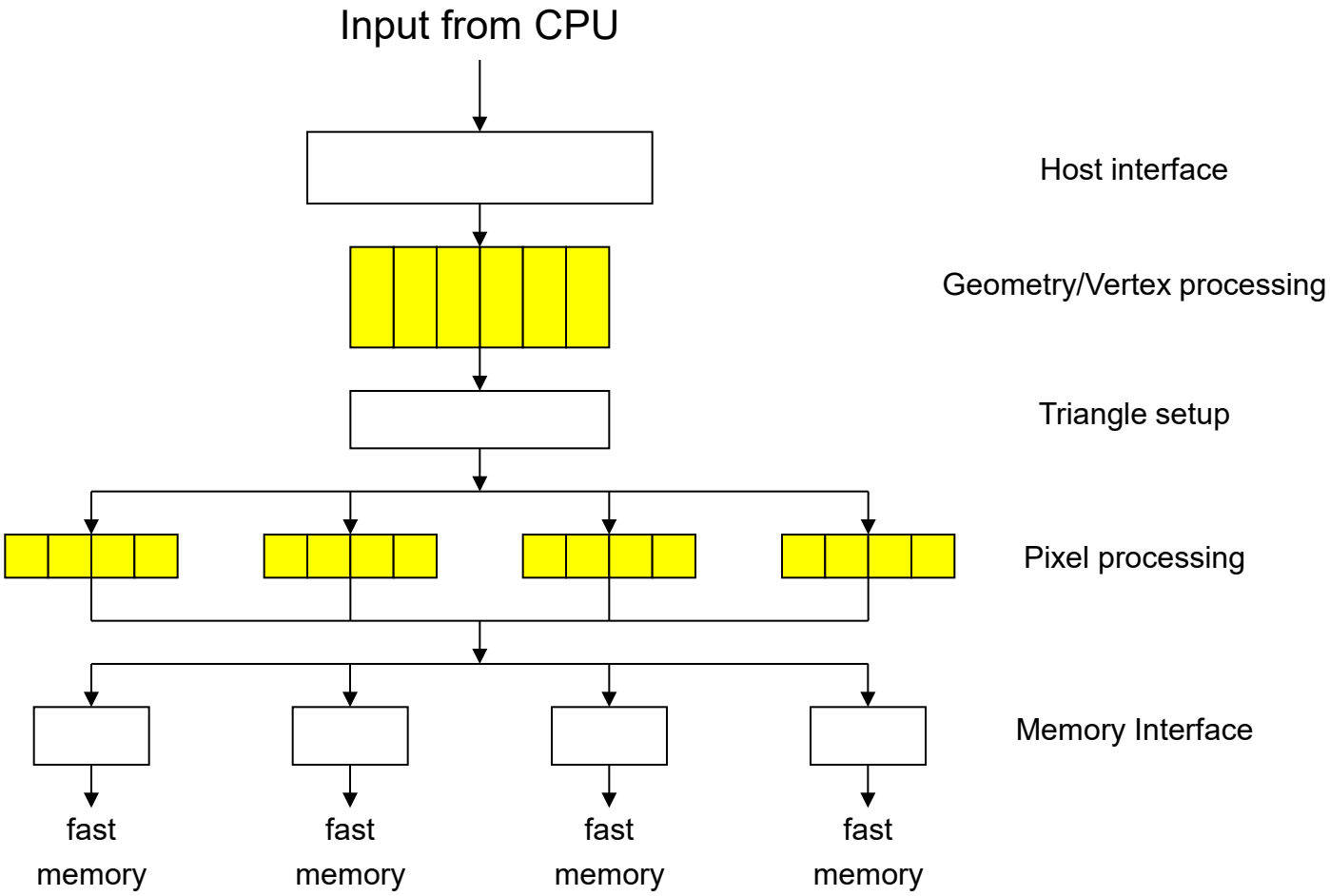
Motivation: Flexible and Precise

- *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!

Diagram of a Modern GPU



nVIDIA GPU

- GTX 4090
  - 16,384 cores (i.e., mini processors)
  - FLOPS: 83-191 TFLOPS
  - Temp: 90 C (water boils at 100 C)
  - Power: 850 Watts (~a microwave oven)

- 1 Datacenter ~32,000 GPUs
  - ~27 Megawatts
  - (watts needed to for about 2,700 average US homes)

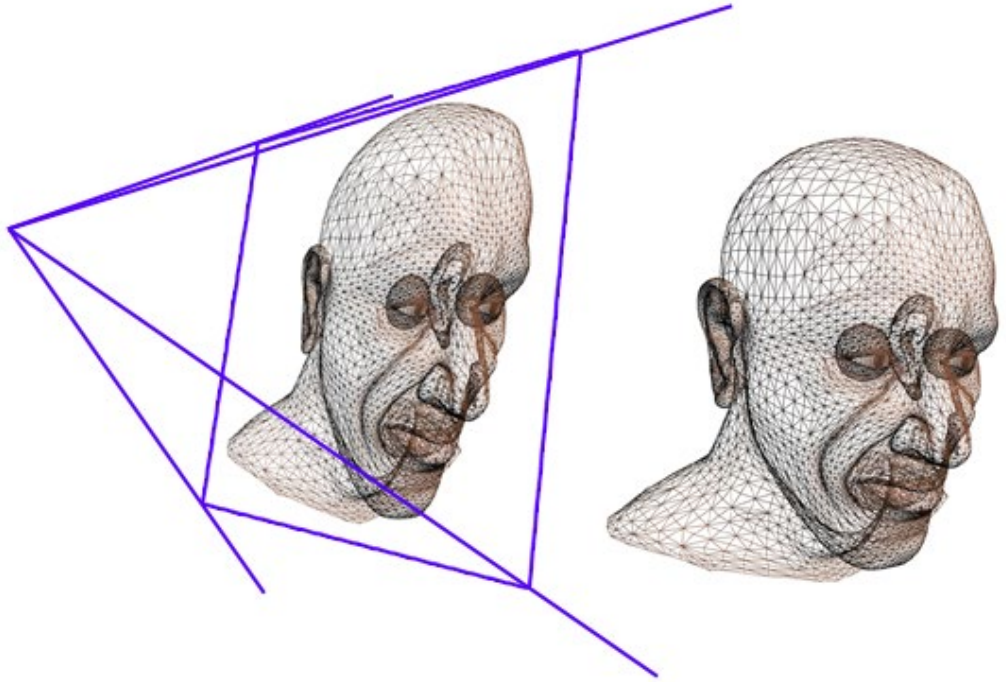
- For all datacenters (~11,000 datacenters globally)
  - Power of about ~30,000,000 average US homes

Modern GPU has more ALU's

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

GPU Pipeline: Transform

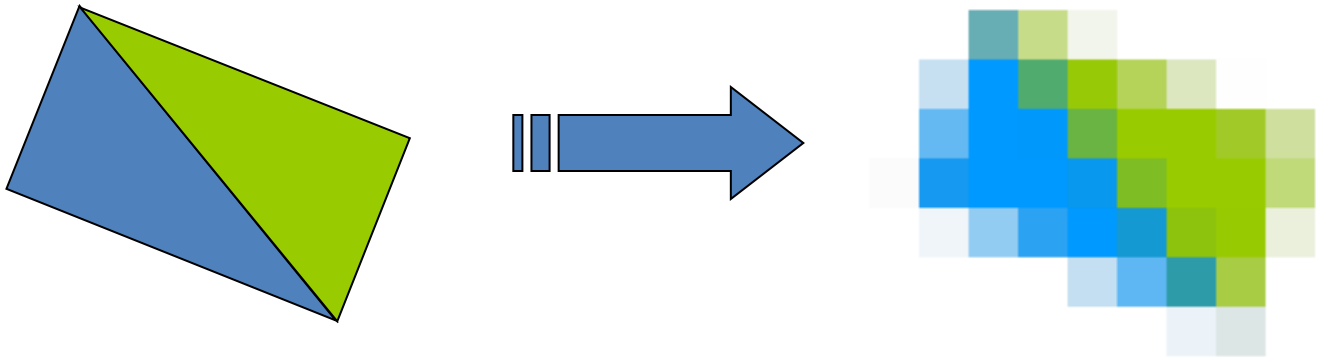
- Vertex/Geometry processor (multiple in parallel)
  - Transform from “world space” to “image space”
  - Compute per-primitive and per-vertex lighting



GPU Pipeline: Rasterize

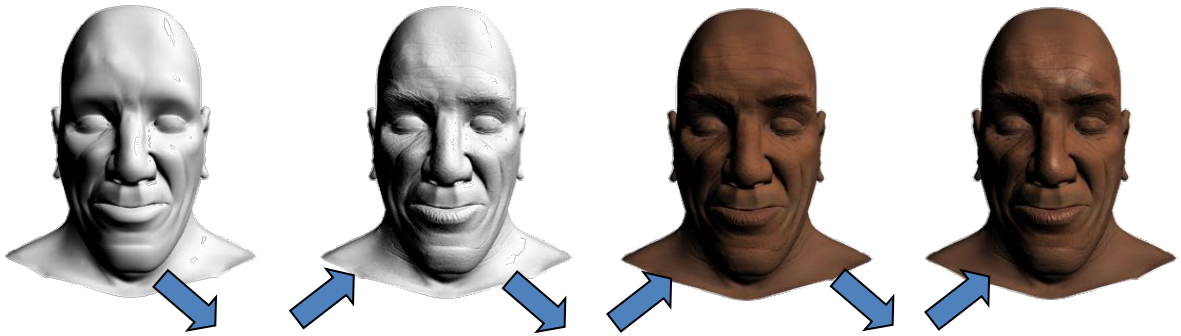
(typically not programmable)

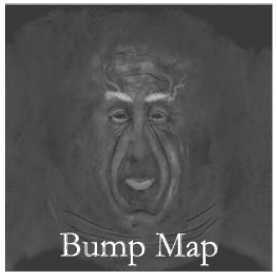
- Rasterizer
  - Convert geometric rep. (vertex) to image rep. (fragment)
    - Fragment = image fragment
      - Pixel + associated data: color, depth, stencil, etc.
  - Interpolate per-vertex quantities across pixels



GPU Pipeline: Shade

- Fragment processors (multiple in parallel)
  - Compute a color for each pixel
  - Optionally read colors from textures (images)





GPU Programming Languages

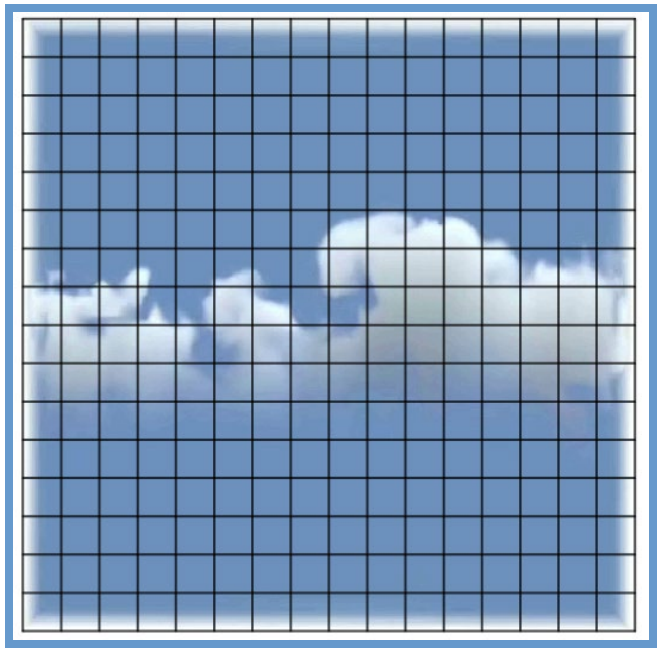
- Many options!
  - A while ago: “Renderman”
  - cG (from NVIDIA)
  - GLSL (GL shading Language)
  - CUDA (more general that graphics)…
- Lets focus first on the concept, later on the language specifics…

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

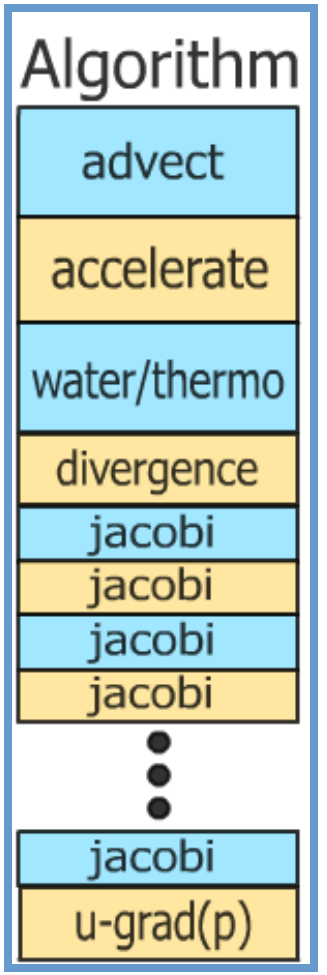
- Common GPGPU computation style
  - Textures represent computational grids = streams
- Many computations map to grids
  - Matrix algebra
  - Image & Volume processing
  - Physically-based simulation
  - Global Illumination
    - ray tracing, photon mapping, radiosity
- Non-grid streams can be mapped to grids



Typical Stream Computation

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

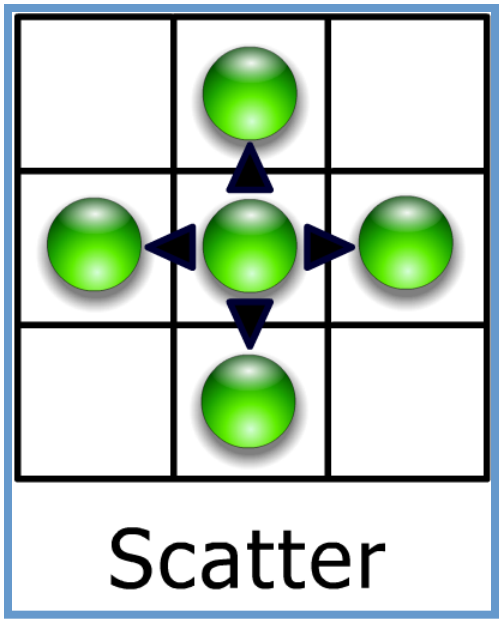
- Grid is a stream, steps are kernels
  - Kernel applied to each stream element

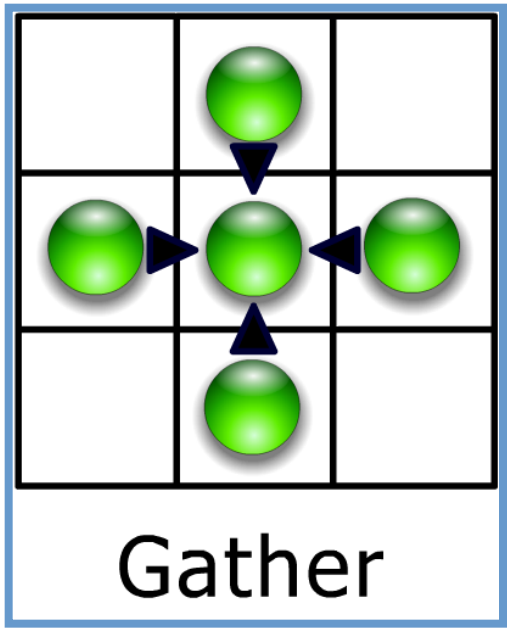


Cloud simulation algorithm

e.g.: Scatter vs. Gather

- Grid communication
  - Grid cells share information





Vertex Processor

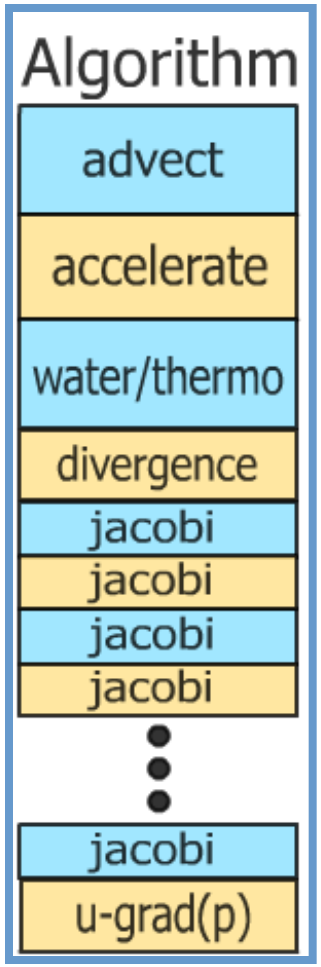
- Fully programmable (SIMD / MIMD)
- Processes 4-vectors (RGBA / XYZW)
- Capable of scatter but not gather
  - Can change the location of current vertex
  - Cannot read info from other vertices
  - Can only read a small constant memory
- Latest GPUs: Vertex Texture Fetch
  - Random access memory for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

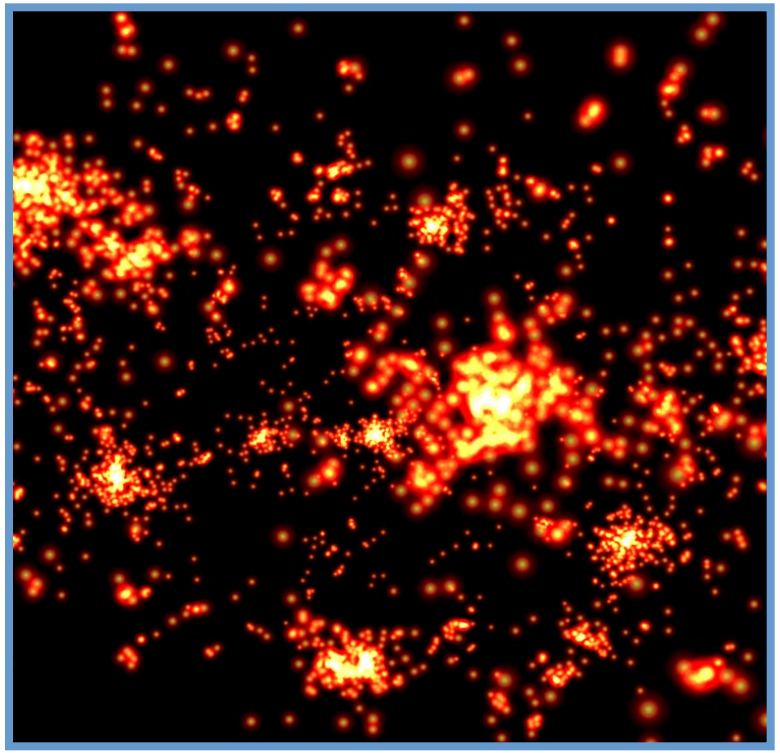
- Initialize “view” (so that pixels:texels::1:1)

```
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



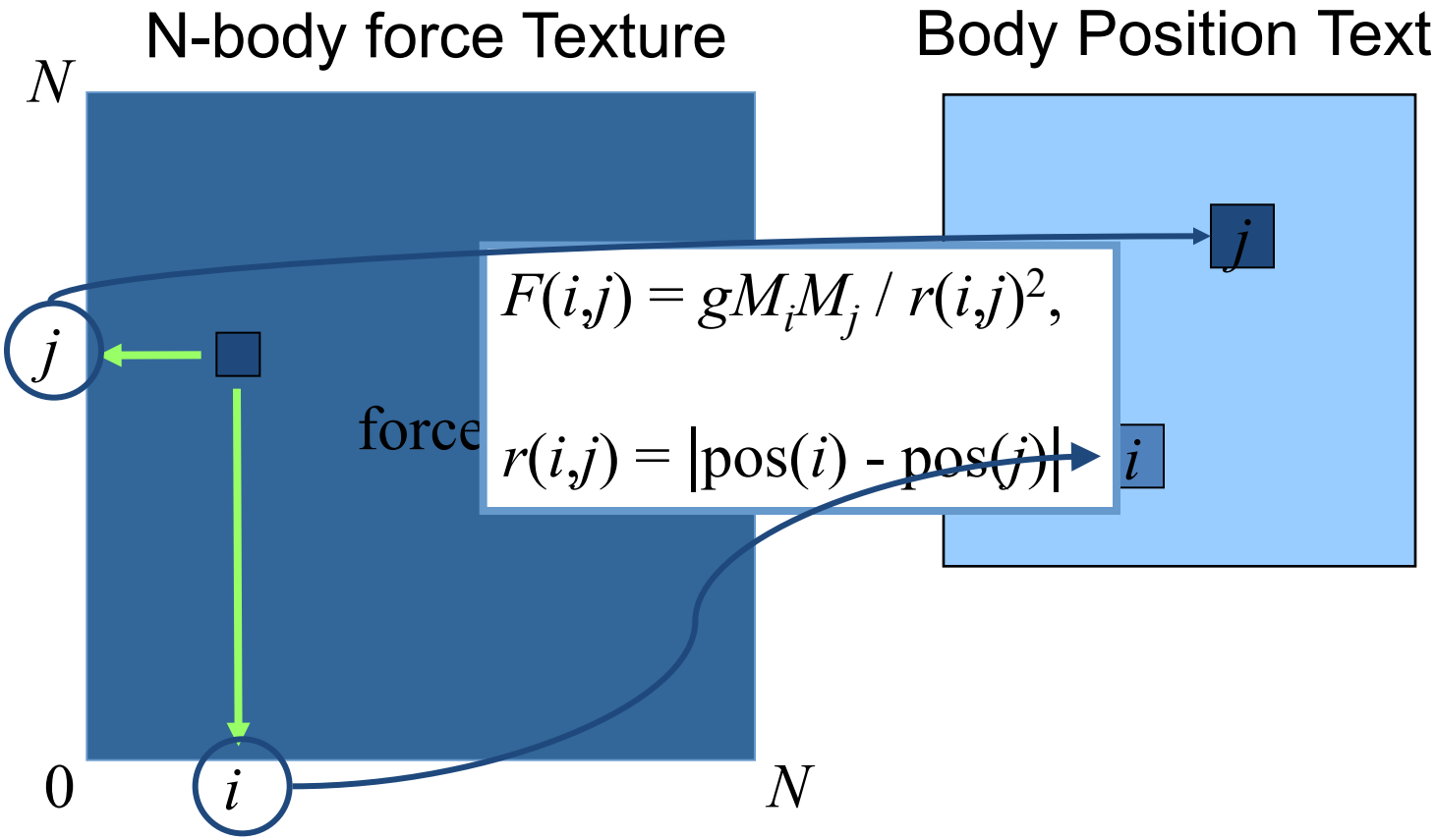
- 64M force comps. / frame
- ~25 flops per force
- 10.5 fps

- 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*x*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



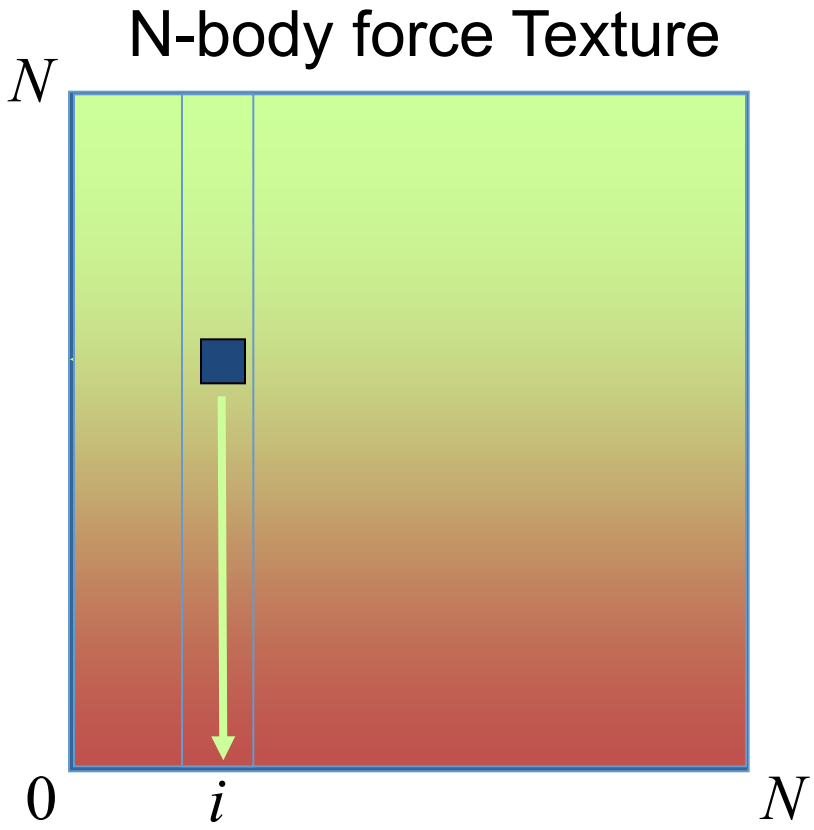
Force is proportional to the inverse square of the distance between bodies

Computing Gravitational Forces


```
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

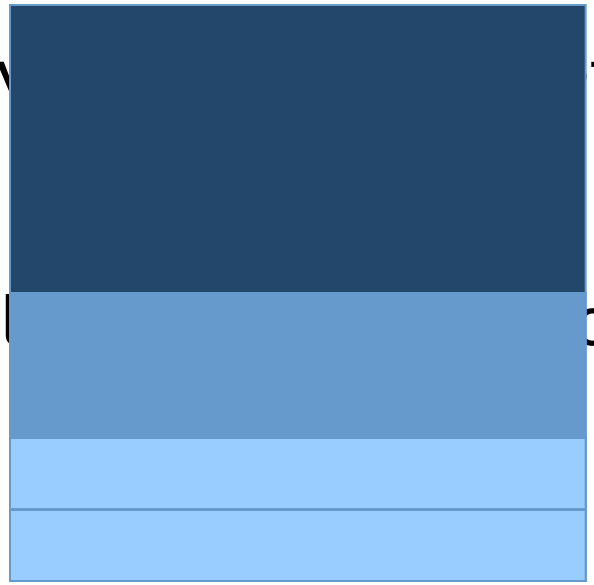
1D parallel reduction:

sum N columns or rows in parallel

add two halv ther

repeatedly..

Until we're l ow of texels



Requires $\log_2 N$ steps

Update Positions and Velocities

- Now we have a 1-D array of total forces
  - One per body
- Update Velocity
  - $\boldsymbol{u}(i,t+dt) = \boldsymbol{u}(i,t) + \boldsymbol{F}_{total}(i) * dt$
  - Simple pixel shader reads previous velocity and force textures, creates new velocity texture
- Update Position
  - $x(i, t+dt) = \boldsymbol{x}(i,t) + \boldsymbol{u}(i,t) * dt$
  - 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!

High Level Shading Languages

- Cg, HLSL, & OpenGL Shading Language
  - Cg:
    - http://www.nvidia.com/cg
  - HLSL:
    - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/highlevellanguageshaders.asp
  - OpenGL Shading Language:
    - http://www.3dlabs.com/support/developer/ogl2/whitepapers/index.html

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

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.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.y)) +
                                                    -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;
}
```

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.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.y)) +
                                                    -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;
}
```

Computational Frequency

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

```
...
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
    assembledtriangle = setup_tri(v1, v2, v3);

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

    // 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

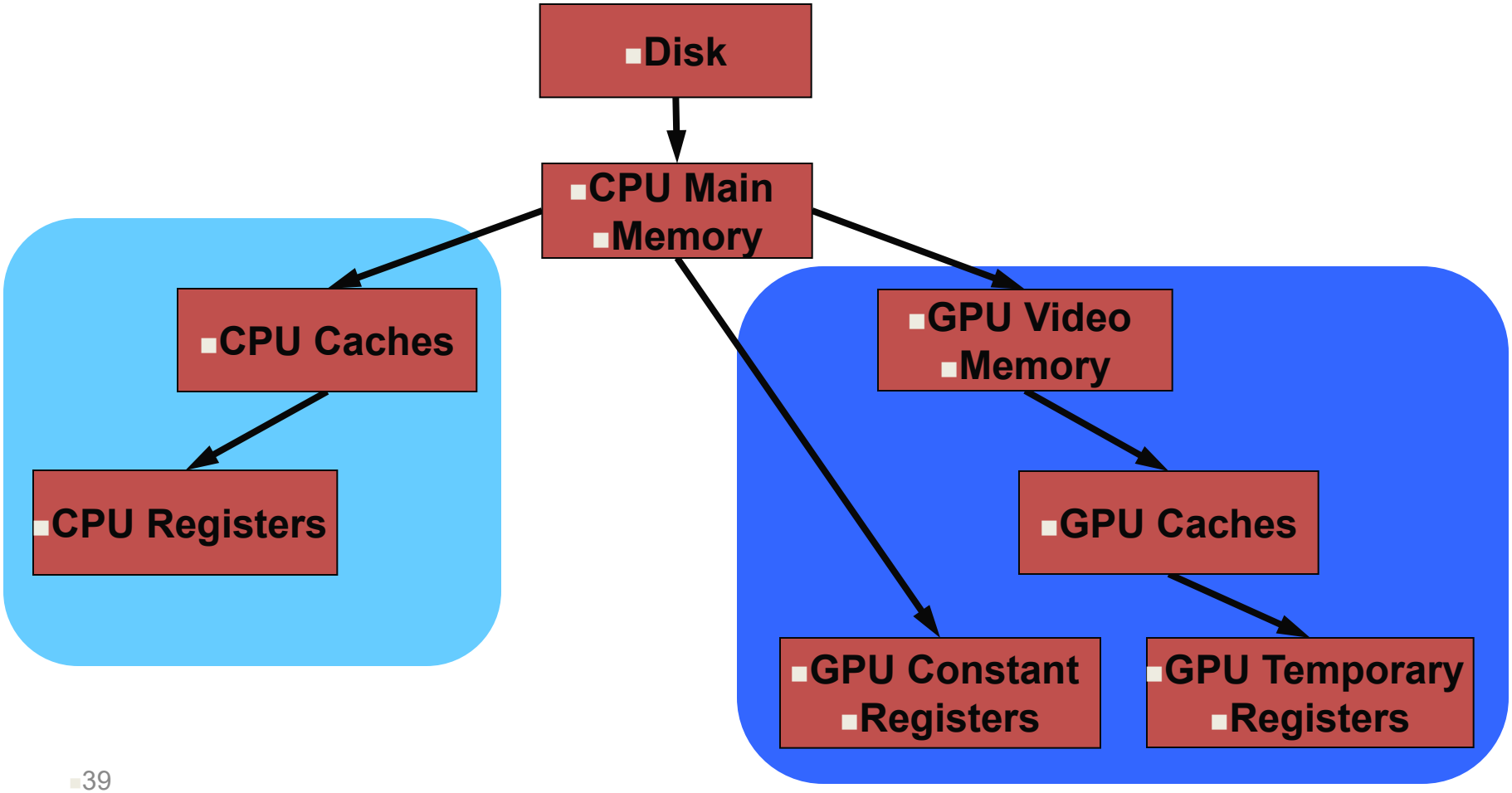
- Static branch resolution
  - write several variants of each fragment program to handle boundary cases
  - eliminates conditionals in the fragment program
  - equivalent to avoiding CPU inner-loop branching

Computational Frequency

- Dynamic branching
  - Use only when needed
    - Good perf requires spatial coherence in branching

Memory Hierarchy

- CPU and GPU Memory Hierarchy



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