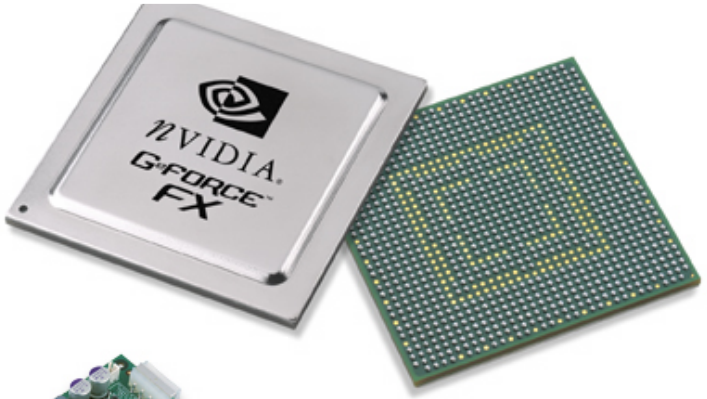
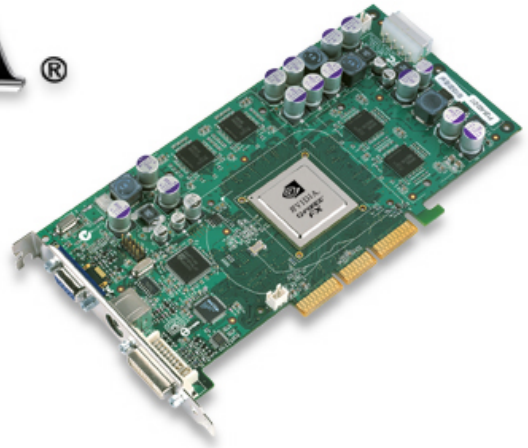
Implementing PDE Algorithms on the GPU





Gallagher Pryor

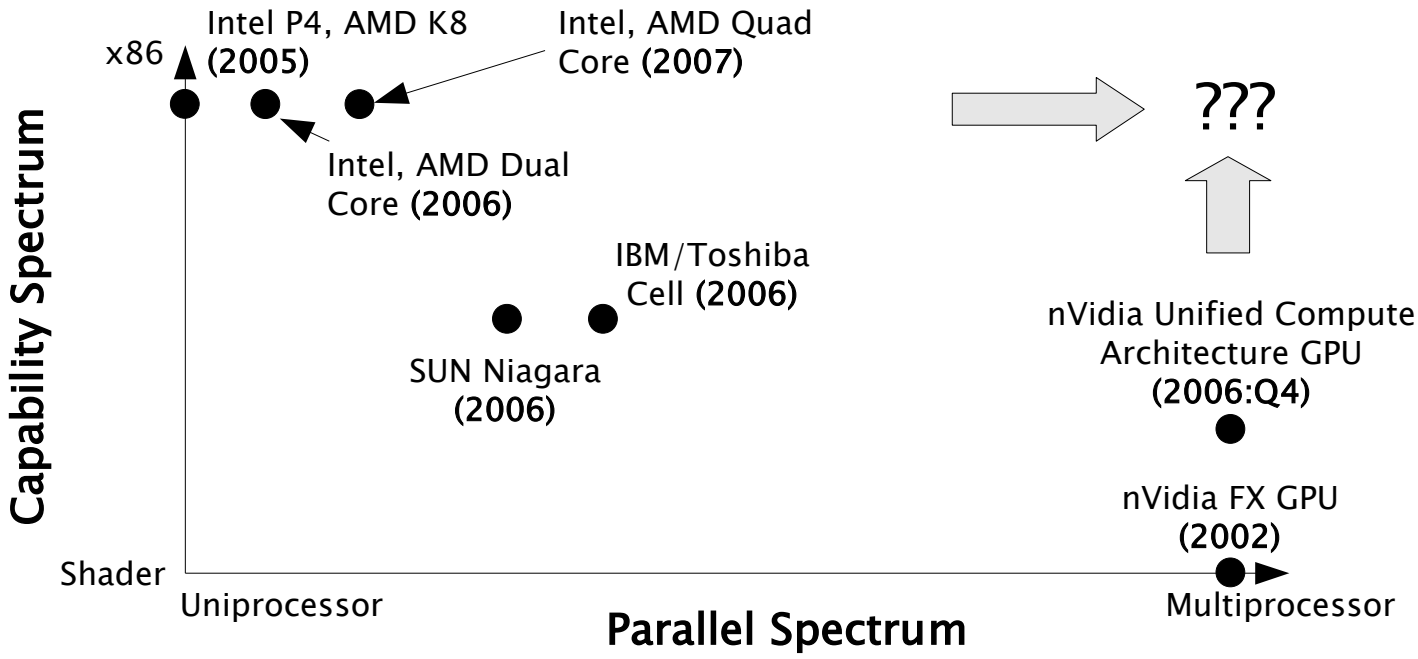
Tannenbaum Class Lecture

April 17th, 2007

Part I: Motivation

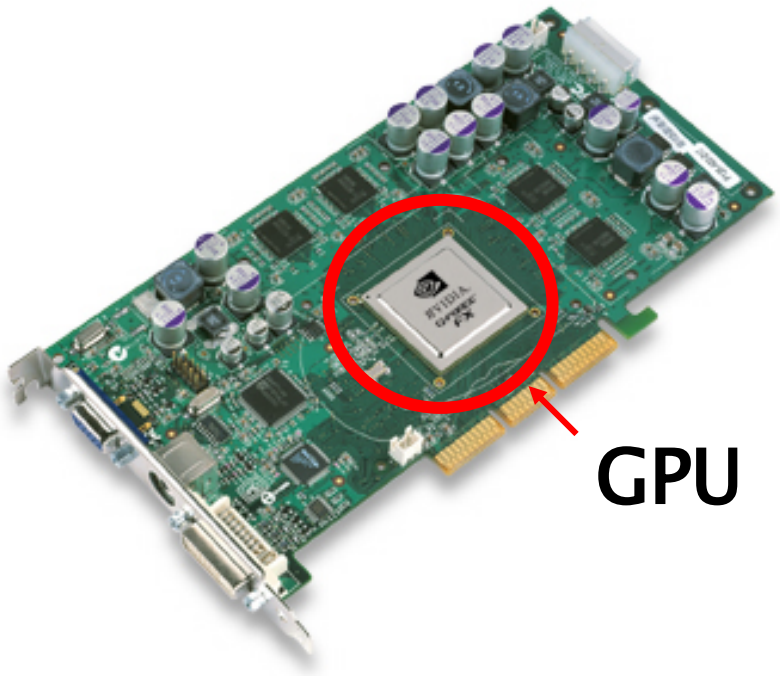
Computers are Going Parallel

- Two major threads in hardware development:
  - General uniprocessors are being integrated into parallel architectures at the package level.
  - Specialized parallel processors are being adapted to more general computation.



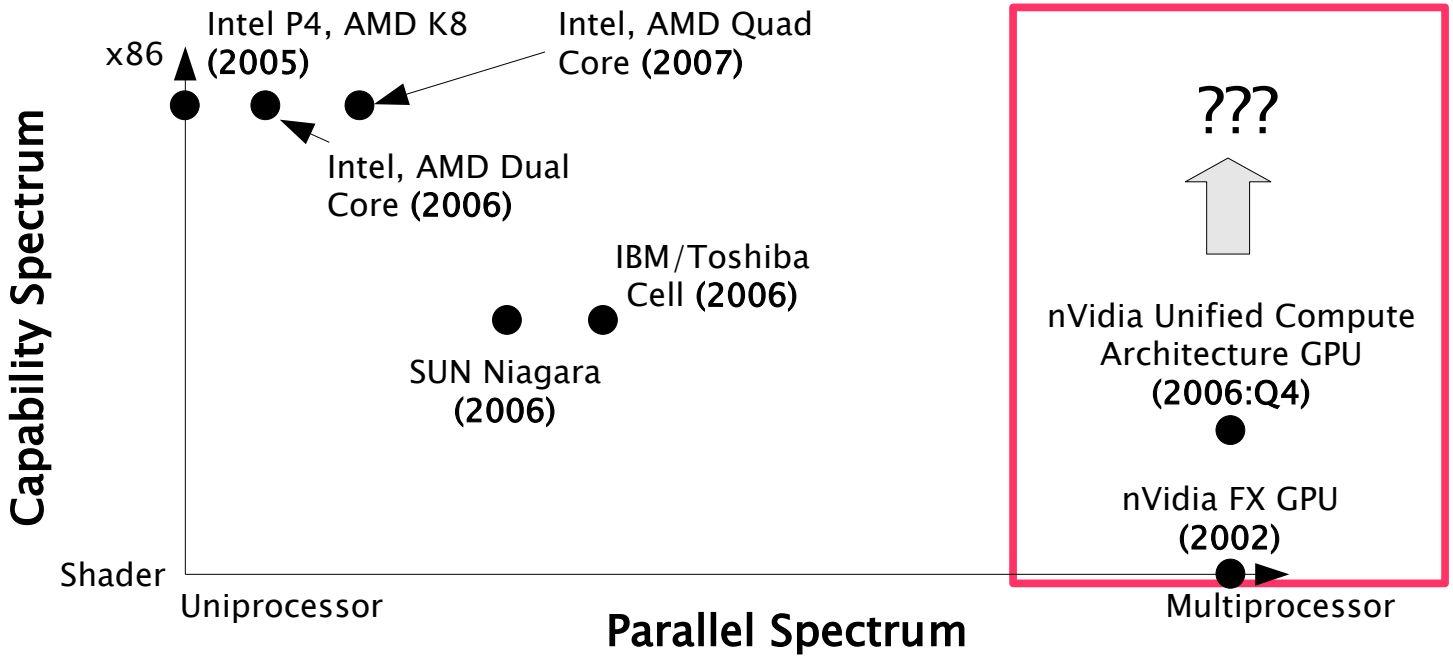
GPU Definition

- GPU: Graphics Processing Unit
- Two major components:
  - **Legacy Hardware** for traditional video display.
  - **Graphics Processing Unit (GPU)**: Contains acceleration hardware for fast 2D and 3D image synthesis.
- Major Vendors / Brands:
  - nVidia & ATI
- Parallel
  - Up to 128 processors on board.



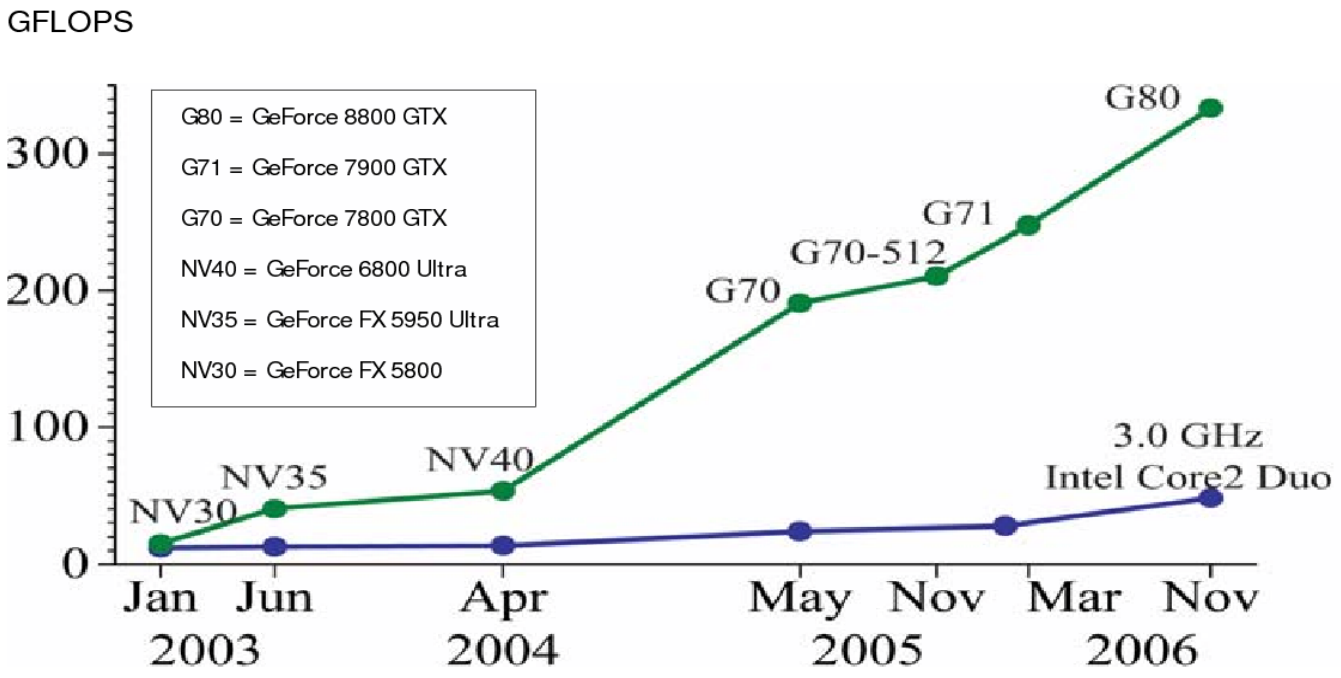
GPU > x86 in the Parallel World

- The GPU claims a unique spot among computing architectures:
  - First ubiquitous parallel architecture
  - Simplest computing model
  - Most massively parallel



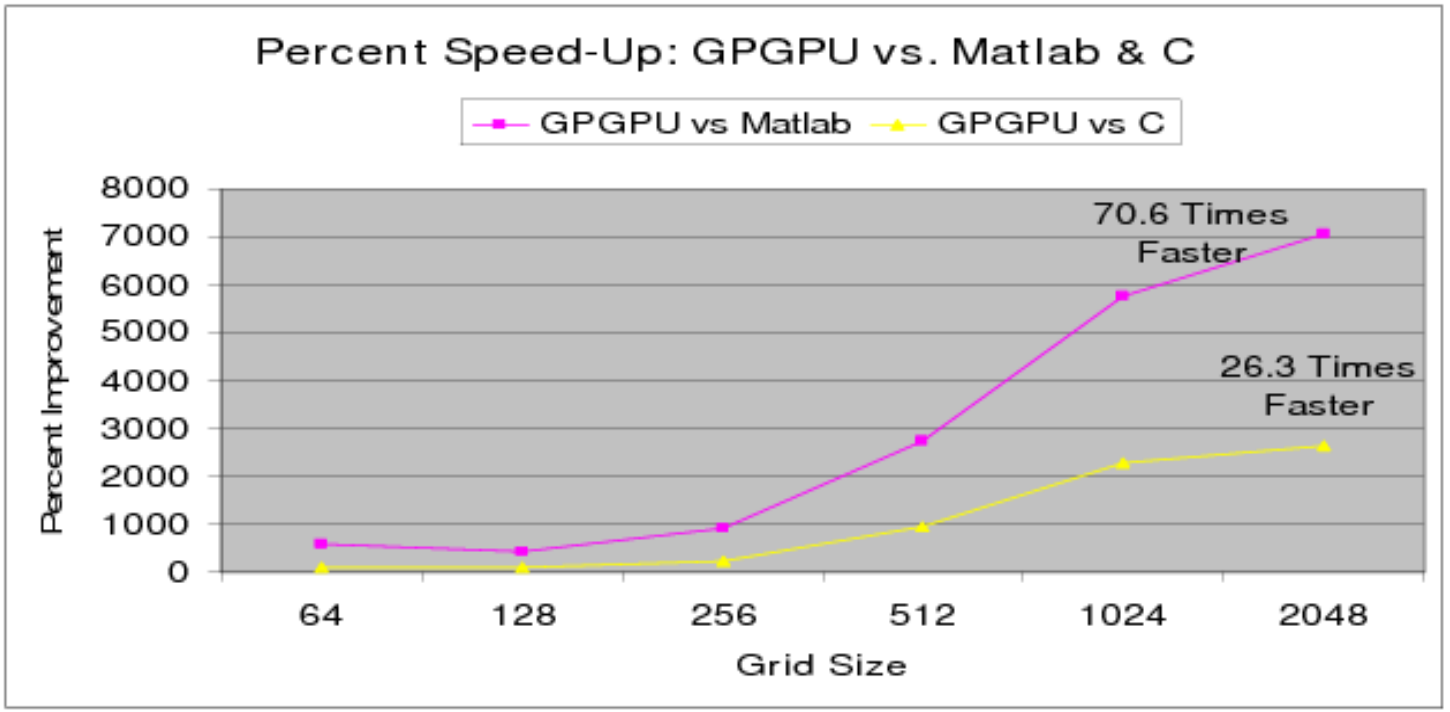
GPU > x86 in the Parallel World

- New GPU hardware now capable of ~10x more GFLOPS than most recent x86 offerings [1].
- Recent results show even greater performance margin [2].



GPU > x86 in the Parallel World

- New GPU hardware now capable of ~10x more GFLOPS than most recent x86 offerings [1].
- Recent results show even greater performance margin [2].



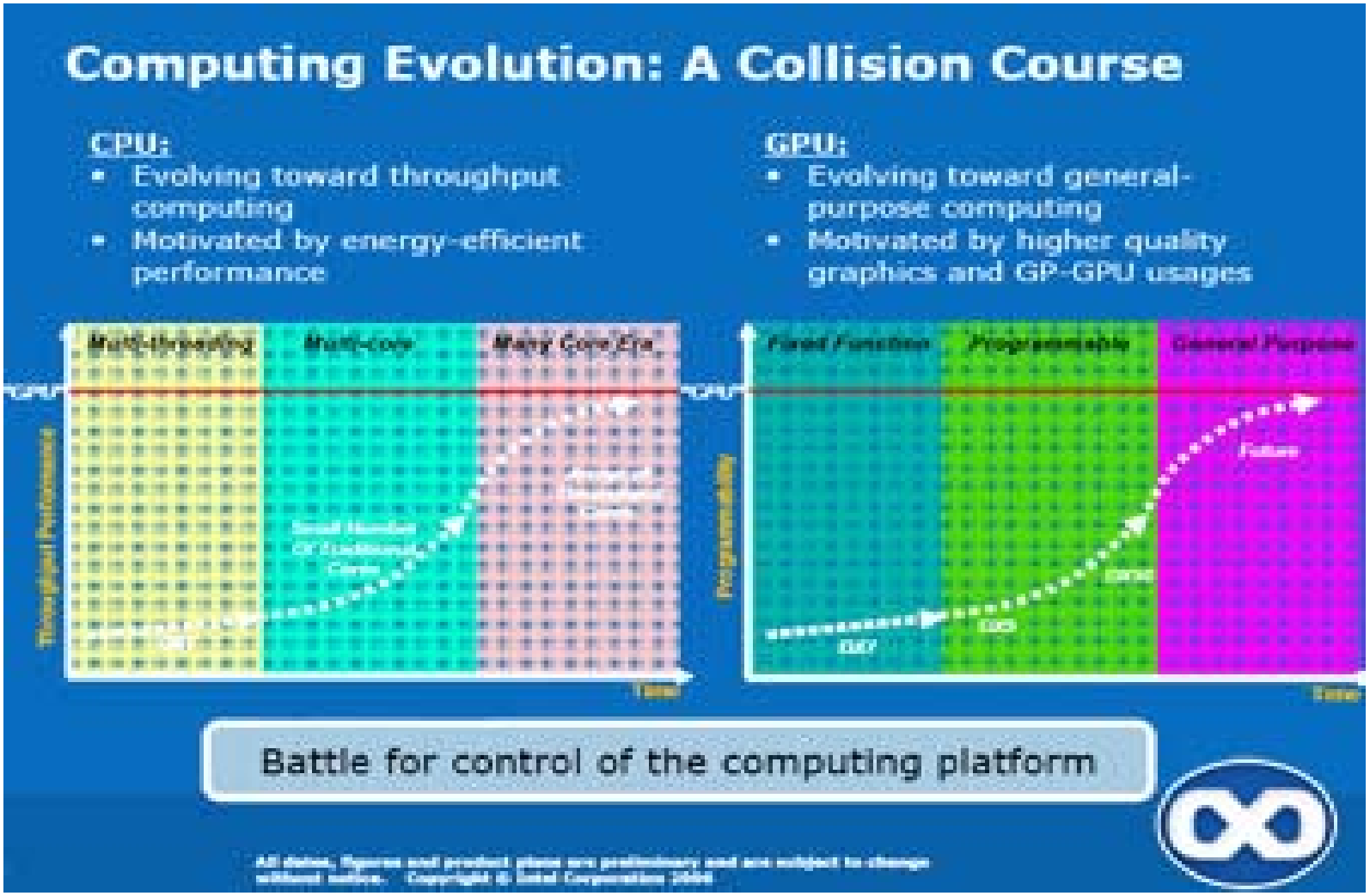
GPU's Simplicity Yields Performance

- GPU's simplicity enforces the following performance-enhancing features:
  - No operating system on board
  - Dedicated local, high-speed memory
  - Implicit thread barrier in hardware
  - (PDE Specific) No cache issues
- At first seemingly restrictive, but ultimately beneficial.
- GPUs have inadvertently hit a computational “sweet spot” with this configuration.

GPU May Become Coprocessor

- Industry rumors and musings of GPU+CPU integration efforts:
  - AMD/ATI have announced a project code-named “Fusion” integrating a GPU and CPU on a single die.
  - Intel reportedly claims that their upcoming “Nehalem” core will sport an integrated GPU [3].
  - **THIS JUST IN:** Intel reports on Larrabee core.
- Either way, the competition is there and one should expect such coupling.

GPU May Become Coprocessor



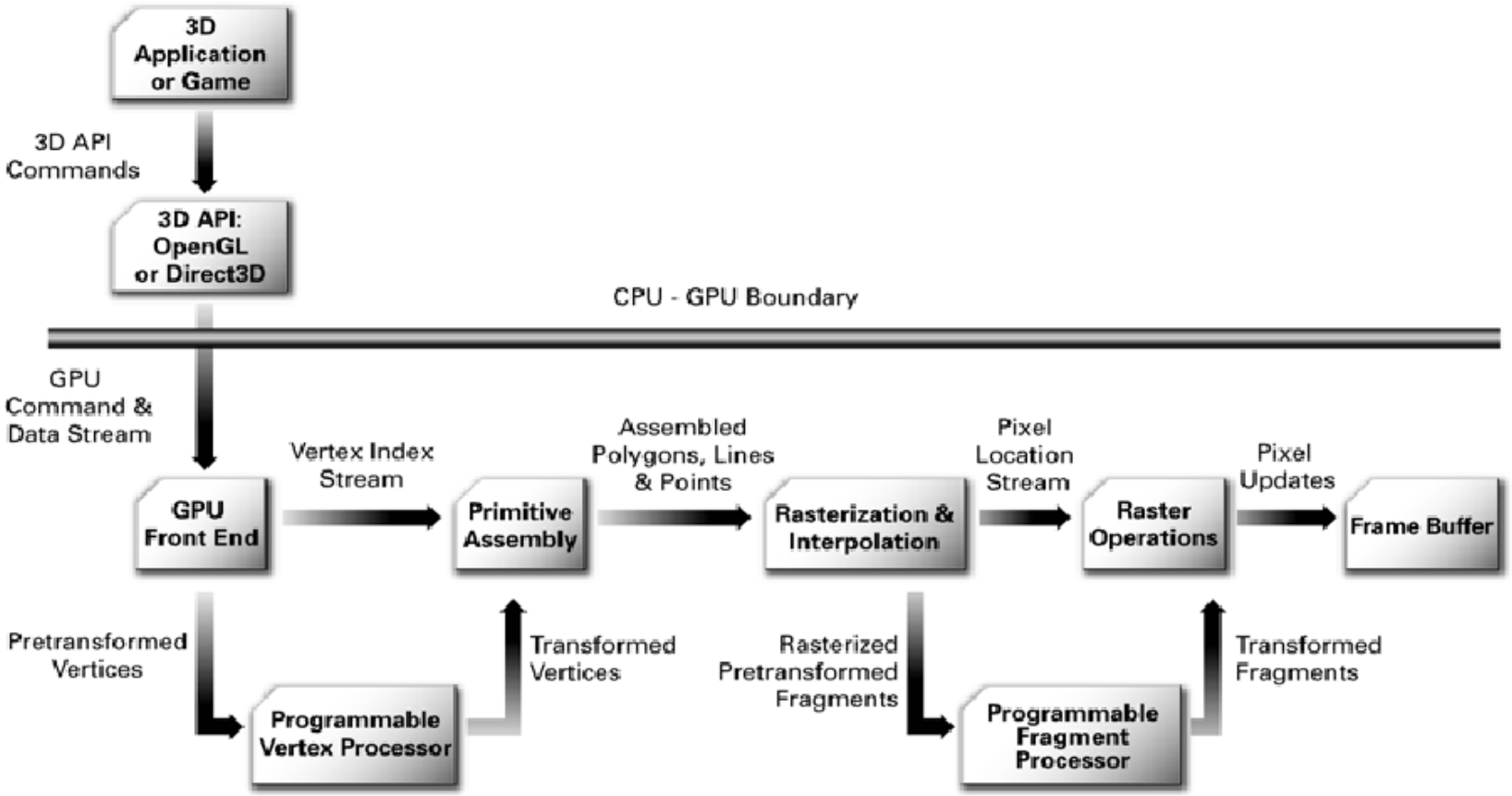
Slide from Douglas Carmean's (Chief Architect; Intel Visual Computing Group) presentation: “Future CPU Architectures -- The Shift from Traditional Models.” [4]

Part II: The GPU In Detail

Taking Apart the GPU

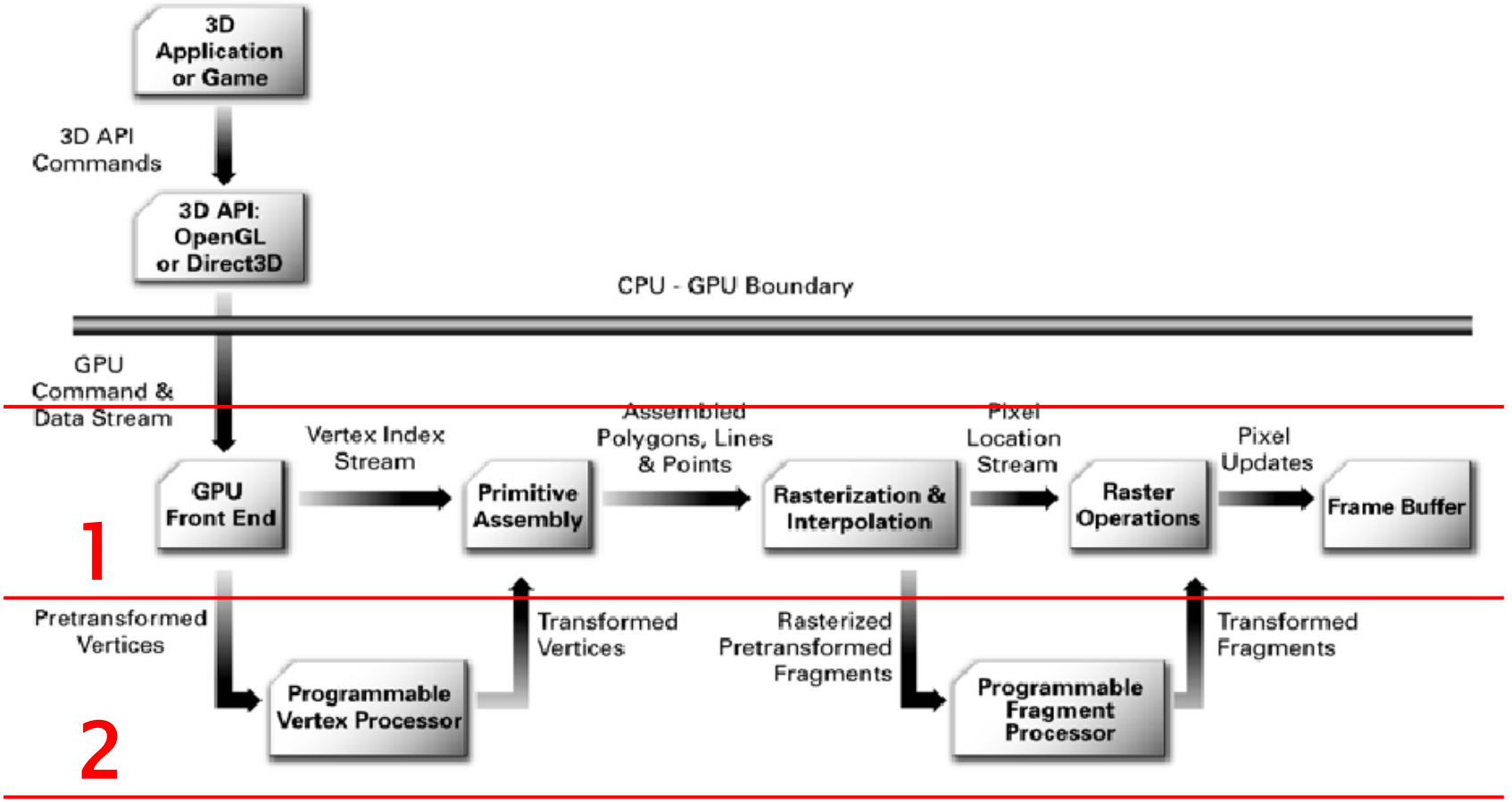
- The GPU is first and foremost a graphics renderer:
  - Input: Scene Description **(vertices,textures,instructions)**
  - Output: 2D Image
- Processing steps are always:
  - **Primitive Assembly**: Vertices become squares, triangles meshes.
  - **Rasterization/Interpolation**: Projection, antialiasing.
  - **Pixel Rasterization**: Surface and texture rendering
  - **Dump to Screen**: Final result hits the framebuffer
- Hardware Required:
  - Highly specialized vectorized floating point unit
  - On-board memory into the gigabyte range
  - So called “stream architecture” (Parallel FIFO Pipeline)

Taking Apart the GPU: The Pipeline



Rendering a Picture: The Modern GPU Rendering Pipeline

Taking Apart the GPU: The Pipeline



- Old generation GPUs had hardware for only row 1
- New generation GPUs provide new hardware for row 2

Vertex Programmability Example

- Vertex processor takes as input a static geometric model and outputs new, dynamically moving model [5].
- Operates on each incoming vertex *in parallel*.

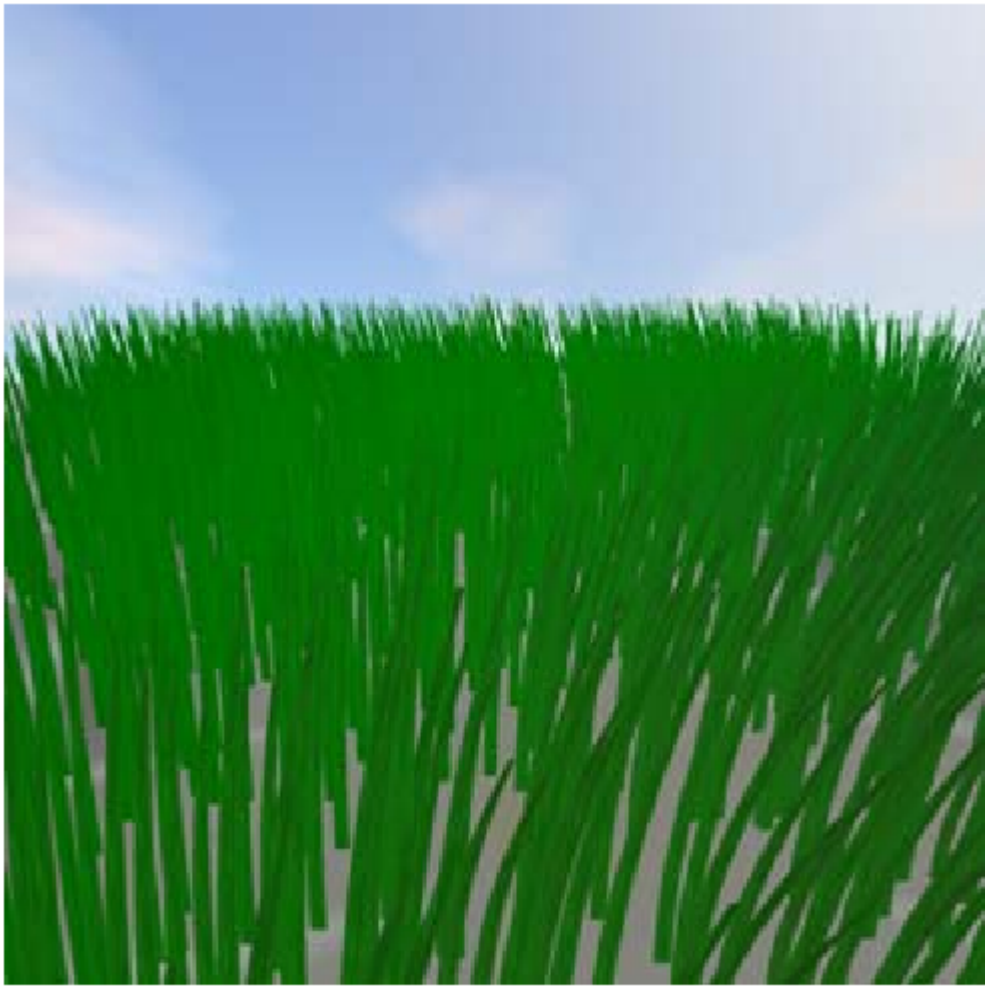


Figure 17 Example of Grass

Fragment Programmability Example

- Shader processor takes as input a 3D position and outputs a color [5].
- Operates on each drawable pixel on the screen *in parallel*.

Figure 18 Example of Refraction

Custom Vertex/Fragment Processing

- Having programmable control over the way vertices and fragments (pixels) are processed is **powerful**:
  - VIDEO: Rendering before and after.
- **This is the feature we will use for PDE solver implementation.**

Simple GPGPU Programming Overview

- Computation is achieved via the following:
  - Set up the GPU to render a single polygon that fills the rendering screen s.t. each pixel of the polygon occupies one pixel of screen.
  - Load data to be processed onto the video card as textures (2D arrays).
  - Associate a *fragment shader* with the polygon. (You write this to do computer vision).
  - Render scene to memory.
- Result is that a chosen operation is performed on each element of input data and output as a result *in parallel*.

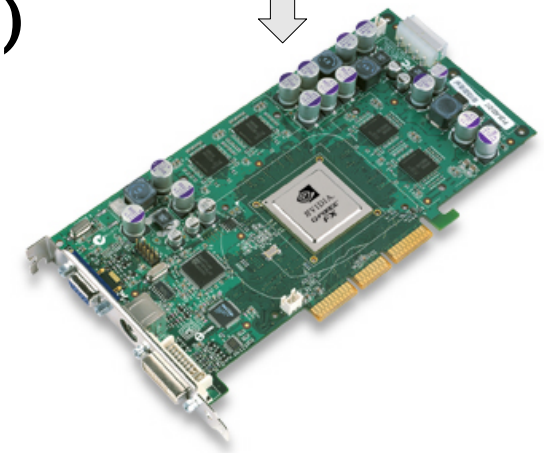
Simple GPGPU Programming Overview

(1)

Polygon defined in scene with 1-1 pixel to output element correspondence.

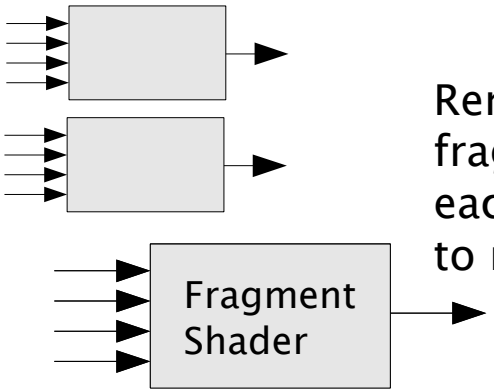
0101101010110
1011010101...

(2)



Input data loaded into 2D arrays, or textures in video memory.

(3)



Render runs fragment shader for each pixel *in parallel* to memory.

GPU Programming Concepts

- CPU = GPU Concept Mappings:
  - Texture = Array. Extra work needed for >2D data.
  - Function = Fragment Shader. Executed on entire texture at one time on per-element basis.
  - Thread = 1 Pixel Evaluation.
  - Program Execution = Render Pass. Implicit thread barrier.
- Math = GPU Concept Mappings:
  - Function = Texture
  - Arbitrary Kernel = Fragment Shader

GPU Limitations and Advantages

- GPU Limitations:
  - Fragment Shaders may not communicate during render. Therefore not suited for:
    - Large Sums
    - Texture-wide Counting
  - No Random-Access Output. Each shader outputs to exactly one, fixed, element of an output texture.
  - Limited branching support (selector method).
  - Limited iterator support (unrolled).
  - Program Length Restricted.
  - GPU-CPU Memory Bandwidth is an issue. Avoid moving data across the bus.

GPU Limitations and Advantages

- GPU Advantages:
  - Simplicity leads to speed and ease of coding.
  - Overcomes many shortcomings of many other more complex parallel machines.
- Several limitations are disappearing (CUDA):
  - Communication between fragments
  - Random Access Output

Part III: Practicum

OpenGL GPGPU Tool Stack

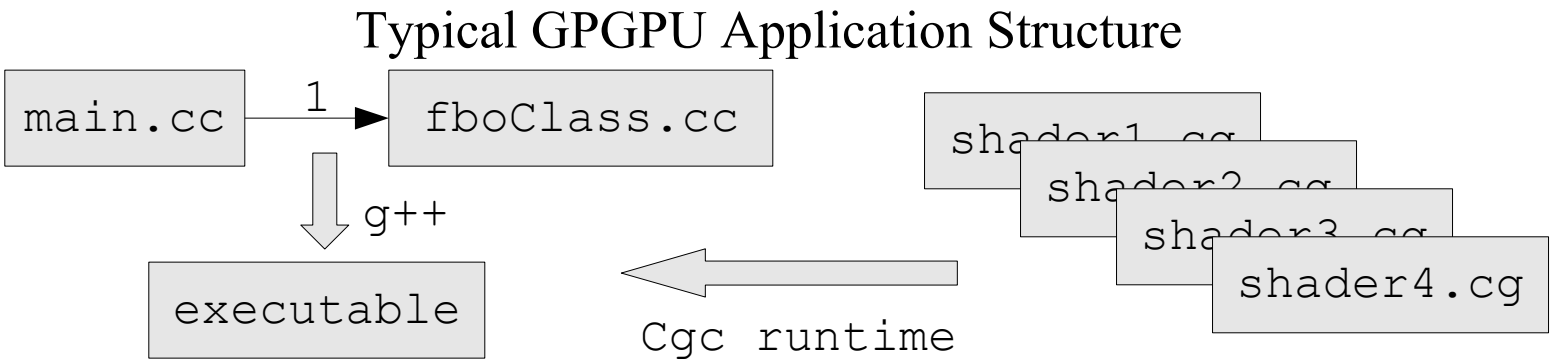
- Required Libraries/Toolset:
  - OpenGL
    http://berkelium.com/OpenGL/sgi-download.html
  - GLUT: GL Utility Toobox
    http://freeglut.sourceforge.net/
  - GLEW: GL Extension Wrangler
    http://glew.sourceforge.net/
  - Cg
    http://developer.nvidia.com/page/cg_main.html
- Application Architecture
  - C Application Calls OpenGL
  - Loads Cg programs (text files) as shaders
  - OpenGL Directs Rendering/ GPU Computation

Steps for GPGPU Processing

- Initialize OpenGL & friends.
- Construct scene as described previously.
- Set a FrameBuffer Object as a render target
  - Instead of the screen
  - Renders to GPU memory
- Initialize Cg and load Cg program as a fragment shader.
- Load input data into textures
- Do renders.
- Push and Pull data to/from GPU memory

Steps for GPGPU Processing

- The FBO Class by Aaron Lefohn implements these steps nicely [7].
  - Not state of the art; one can achieve much higher performance.
  - We have a very fast toolbox in Prof Tannenbaum's lab that took >1 year to create.
- All you need to do is define shaders in the Cg language and apply them in meaningful steps.



Learn By Doing: Solving the Heat Eq.

- The Heat Equation is given by:
  - $\frac{\partial \phi}{\partial t} = \nabla^2 \phi$ on the unit square Ω
- Iteratively solved by a difference equation:
  - $\phi^{t+1} = \phi^t + \Delta t \nabla^2 \phi^t$
    with some initial condition ϕ^0
- Where $\nabla^2 \phi = \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{\Delta x^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{\Delta y^2}$

Learn By Doing: Solving the Heat Eq.

- GPGPU Implementation Outline (main.cc):
  - GPGPU framework initialized via fboClass.
  - Cg shader loaded which implements $\phi^{t+1} = \phi^t + \Delta t \nabla^2 \phi^t$
  - Texture A loaded with init. cond ϕ^0
  - Texture B created.
  - For i=1..n iterations
    - Render from Texture A -> Texture B w/ Cg shader.
    - Swap Texture A with Texture B
  - Pull answer out of appropriate texture to CPU memory.

Heat Equation Cg Shader

- Cg Shader code is as follows (shader1.cg):

```
void FragmentProgram(in float2 c           : TEXCOORD0,
                     out float b           : COLOR0,
                     uniform float dt,
                     uniform samplerRECT a : TEXUNIT0) {
   left = texRECT(a, c + float2(-1,0));
   right = texRECT(a, c + float2(1,0));
   center = texRECT(a, c);
   up = texRECT(a, c + float2(0,-1));
   down = texRECT(a, c + float2(0,1));
   b = dt * ( right - 2*center + left +
              down  - 2*center + up );
}
```

- Cg semantics
- DEMO

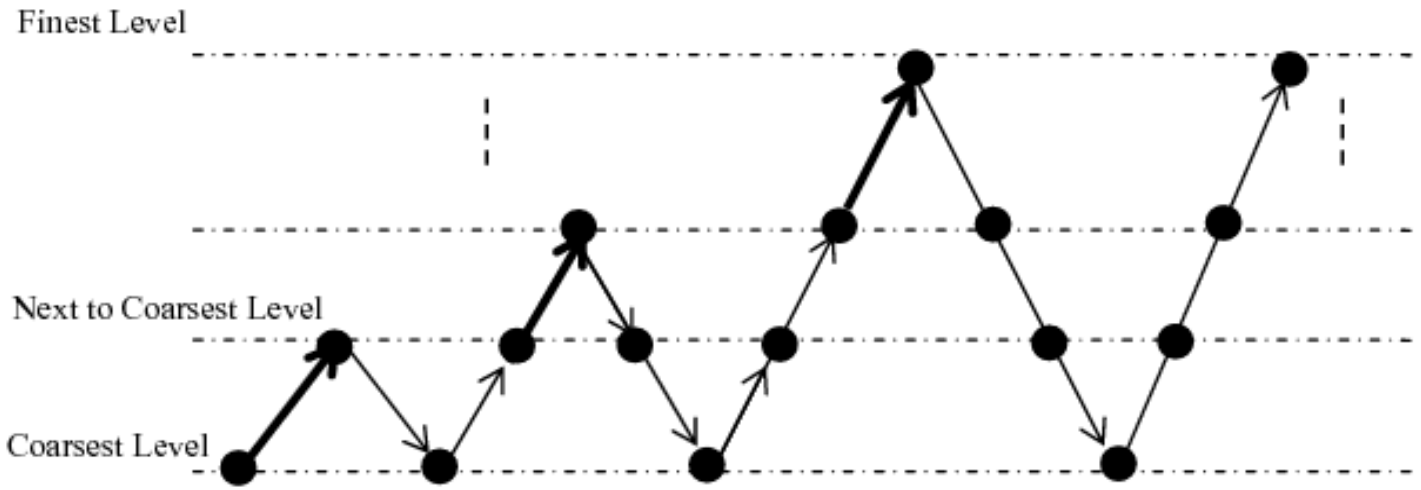
Cg Language Features

- Cg language supports:
  - Predicates
  - Iterators
  - Structures / Objects
  - Inheritance

```
fixed FragmentProgram( float2 texcoord : TEXCOORD0,
                       uniform samplerRECT Ixsq,
                       uniform samplerRECT Iysq,
                       uniform samplerRECT u_avg,
                       uniform samplerRECT IxIy,
                       uniform samplerRECT v_avg,
                       uniform samplerRECT IxIt,
                       uniform float lambda,
                       uniform float texwidth,
                       uniform float texheight ) : COLOR
{

    float result;

    if( floor( texcoord.x ) == 0 || floor( texcoord.y ) == 0 ||
        ceil( texcoord.x ) == ceil( texwidth ) ||
        ceil( texcoord.y ) == ceil( texheight ) ) {
        result = 0.0;
    }
    else {
        result = texRECT( u_avg, texcoord ) -
            ( texRECT( Ixsq, texcoord ) *
              texRECT( u_avg, texcoord ) +
              texRECT( IxIy, texcoord ) *
              texRECT( v_avg, texcoord ) +
              texRECT( IxIt, texcoord ) ) /
            ( 1.0 + lambda*lambda*( texRECT( Ixsq, texcoord ) +
                                    texRECT( Iysq, texcoord )
 ) );
    }

    return result;
}
```

Very Fast PDE Implementation

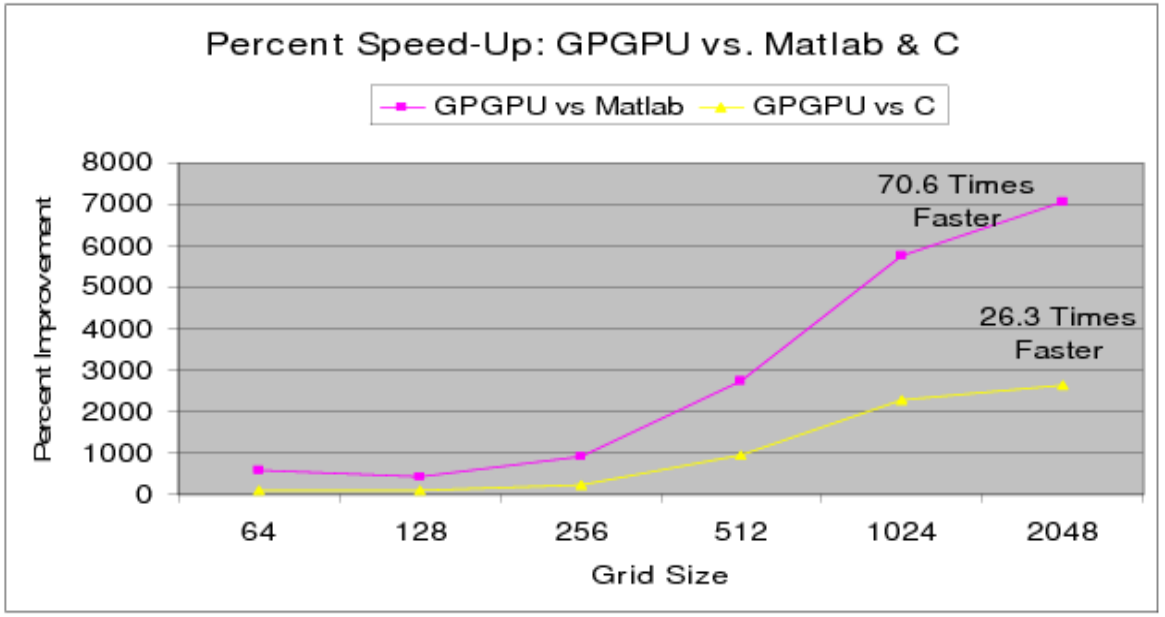
- In our experience multigrid methods along with the GPU provide the best performance.
- Multigrid solves PDE's on grids at many resolutions:



- Orders of magnitude convergence improvements are often realized.

Demos

- Real-time optical flow
- Optimal Mass Transport (2D)
- Optimal Mass Transport (3D)



Biological Similarity

- PDE's are simple to implement:
  - Homogenous
  - Involve communication between few grid points
  - Involve simple computations
- GPU is an array of simple processors with sparse interconnects.
- Visual cortex has been shown to be made up of connected layers of neurons, or simple processors, communicating with one another.
- Visual cortex has been shown to operate at multiple resolutions. Multigrid?

Conclusions

- Computing is about to undergo an amazing revolution.
- Parallelizable problems such as the solution of PDE's will leverage this the most.
- A very exciting time for both computer vision and computer science!
- I hope you enjoyed the talk!

References

[1] nVidia, *The nVidia CUDA Programming Guide.* http://developer.nvidia.com/object/cuda.html

[2] Rehman, Pryor, Tannenbaum. *GPU Enhanced Multigrid Optimal Mass Transport for Image Morphing,* ICIP 2007 (In Submission)

[3] J. Stokes. "Intel Aims Nehalem at AMD's Fusion: Integrated Graphics on-die Memory Controller, SMT," Ars Technica, March 28th, 2007.

[4] "Intel presentation reveals the future of the CPU-GPU war," Beyond 3D, April 11th, 2007

[5] nVidia, *Cg Users Manual*. ftp://download.nvidia.com/developer/cg/Cg_1.2.1/Docs/Cg.

[6] A. Vance, "Intel Confirms Programmable, multi-core chip," The Register, April 17th, 2007; http://www.theregister.com/2007/04/17/intel_larrabee_gpgpu/

References

[7] gpgpu.org, “GPGPU Developer Resources”. http://www.gpgpu.org/developer/