CIS 665: GPU Programming and Architecture

Original Slides by: Suresh Venkatasubramanian
Updates by Joseph Kider

Administrivia

Instructor

- Joseph Kider
  (kiderj _at_ seas.upenn.edu)
- Office Hours: Tuesdays 3-5pm
- Office Location: Moore 103 - HMS Lab

Meeting Time

- Time: Tuesday and Thursday 6-9pm
- Location: Towne 319

Website

http://www.seas.upenn.edu/~cis665/

Administrivia

Course Description

This course will examine the architecture and capabilities of modern GPUs (graphics processing unit). The GPU has grown in power over recent years, to the point where many computations can be performed faster on the GPU than on a traditional CPU. GPUs have also become programmable, allowing them to be used for a diverse set of applications far removed from traditional graphics settings.

Topics covered will include architectural aspects of modern GPUs, with a special focus on their streaming parallel nature, writing programs on the GPU using high level languages like Cg, CUDA, SlabOps, and using the GPU for graphics and general purpose applications in the area of geometry modelling, physical simulation, scientific computing and games.

The course will be hands-on; there will be regular programming assignments, and students will also be expected to work on a project (most likely a larger programming endeavour, though more theoretical options will be considered).

- *NOTE: Students will be expected to have a basic understanding of computer architecture, graphics, and OpenGL.*
  - Course Survey
  - Time Change

Administrivia

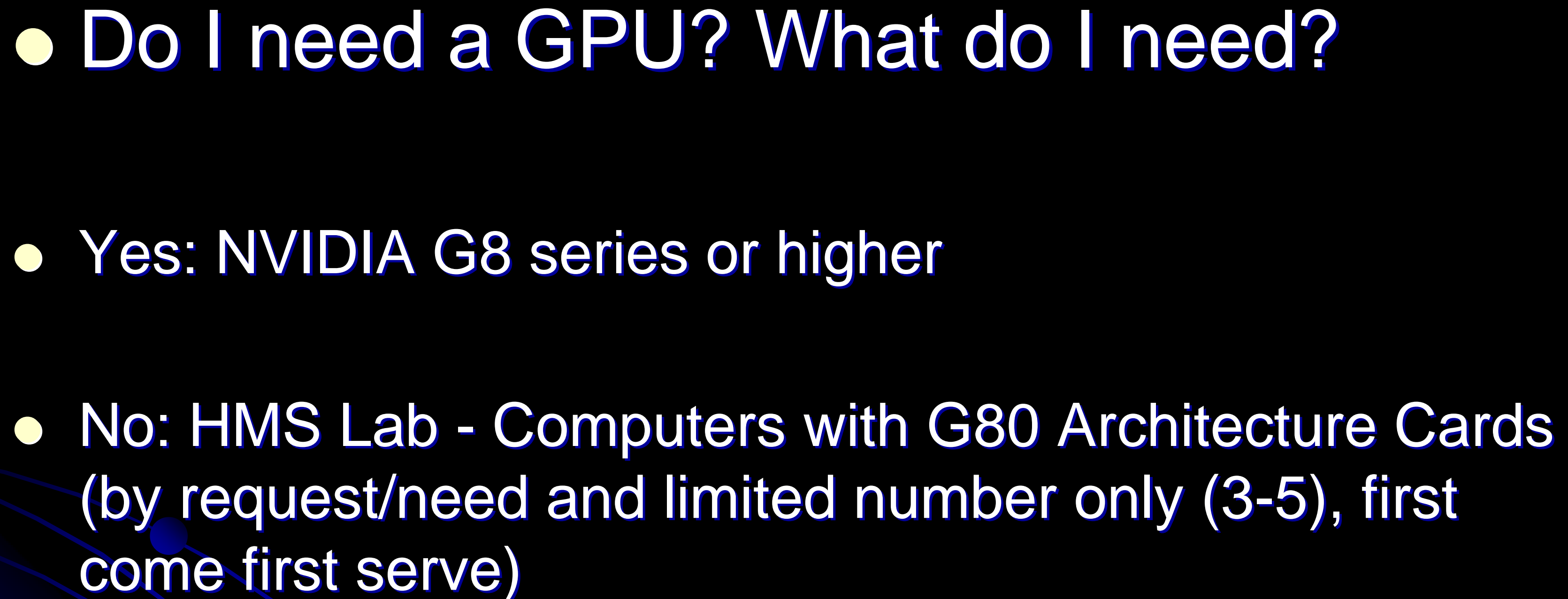
Grading

- Grading for this course is as follows: There is no final or mid-term exams. The grading will be based on homeworks, projects, and presentation. Detailed allocations are tentatively as follows:
- Homeworks (3-4) (75%): Each student will complete 3-4 programming assignments over the semester. These assignments start to fill the student's 'toolbox' of techniques and provide an understanding for the implementation of game rendering, animation, and general purpose algorithms being performed on GPUs. The last assignment will include an open area to choose a problem of your choice to solve.
- Paper Presentation (20%): Each student will present one or two papers on a topic that interests them based on a short list of important papers and subject areas relevant to the GPU literature.
- Quizzes and class participation (5%): A small portion to check if you're attending and paying attention in classes.

Bonus Days

- Each of you get three bonus days
  - A bonus day is a no-questions-asked one-day extension that can be used on most assignments
  - You can use multiple bonus days on the same thing
- Intended to cover illnesses, interview visits, just needing more time, etc.
- I have a strict late policy : if its not turned in on time 11:59pm of due date, 25% is deducted, 2 days late 50%, 3 days 75%, more 99% use your bonus days.
- Always add a readme, note if you use bonus days.

Administrivia



Course Goals

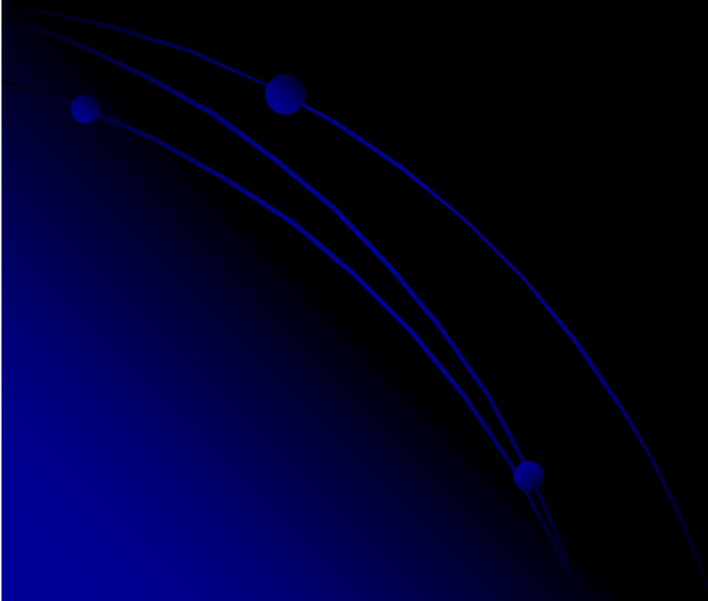
- Learn how to program massively parallel processors and achieve
  - high performance
  - functionality and maintainability
  - scalability across future generations
- Acquire technical knowledge required to achieve the above goals
  - principles and patterns of parallel programming
  - processor architecture features and constraints
  - programming API, tools and techniques

Academic Honesty

- You are allowed and encouraged to discuss assignments with other students in the class. Getting verbal advice/help from people who've already taken the course is also fine.
- Any reference to assignments from previous terms or web postings is unacceptable
- Any copying of non-trivial code is unacceptable
  - Non-trivial = more than a line or so
  - Includes reading someone else's code and then going off to write your own.

Academic Honesty (cont.)

- Penalties for academic dishonesty:
  - Zero on the assignment for the first occasion
  - Automatic failure of the course for repeat offenses

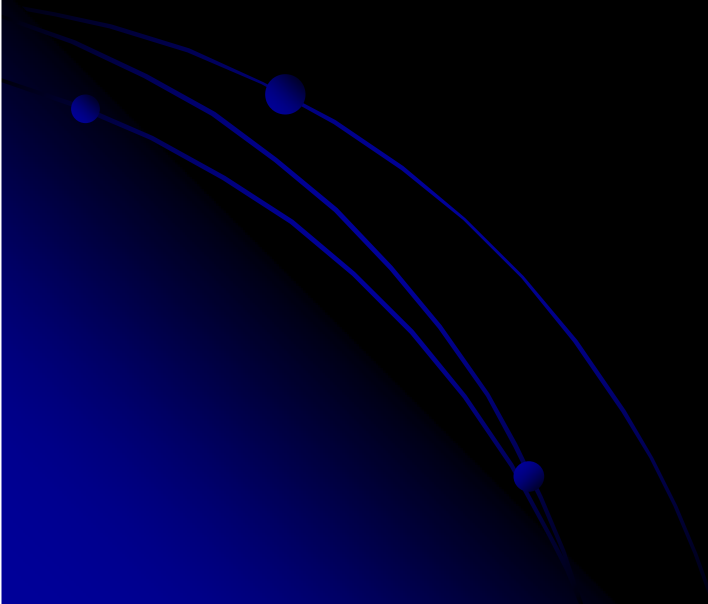


Text/Notes

1. No required text you have to buy.
2. GPU Gems 1 – 3 (1 and 2 online)
3. NVIDIA, *NVidia CUDA Programming Guide*, NVidia, 2007 (reference book)
4. T. Mattson, et al "Patterns for Parallel Programming," Addison Wesley, 2005 (recomm.)
5. The CG Tutorial (online)
6. Lecture notes will be posted at the web site

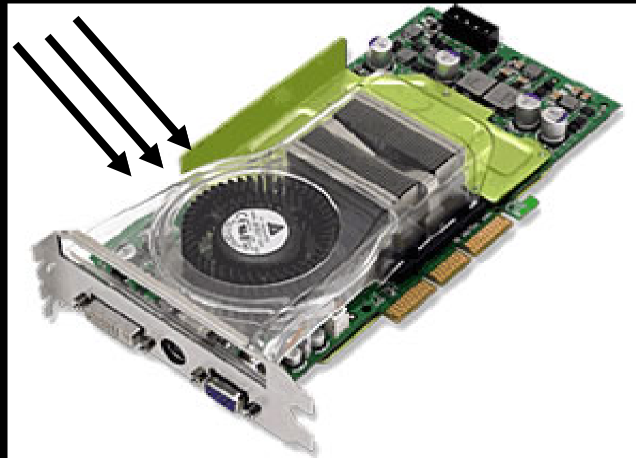
Tentative Schedule

- Review Syllabus
- Talk about paper topic choice



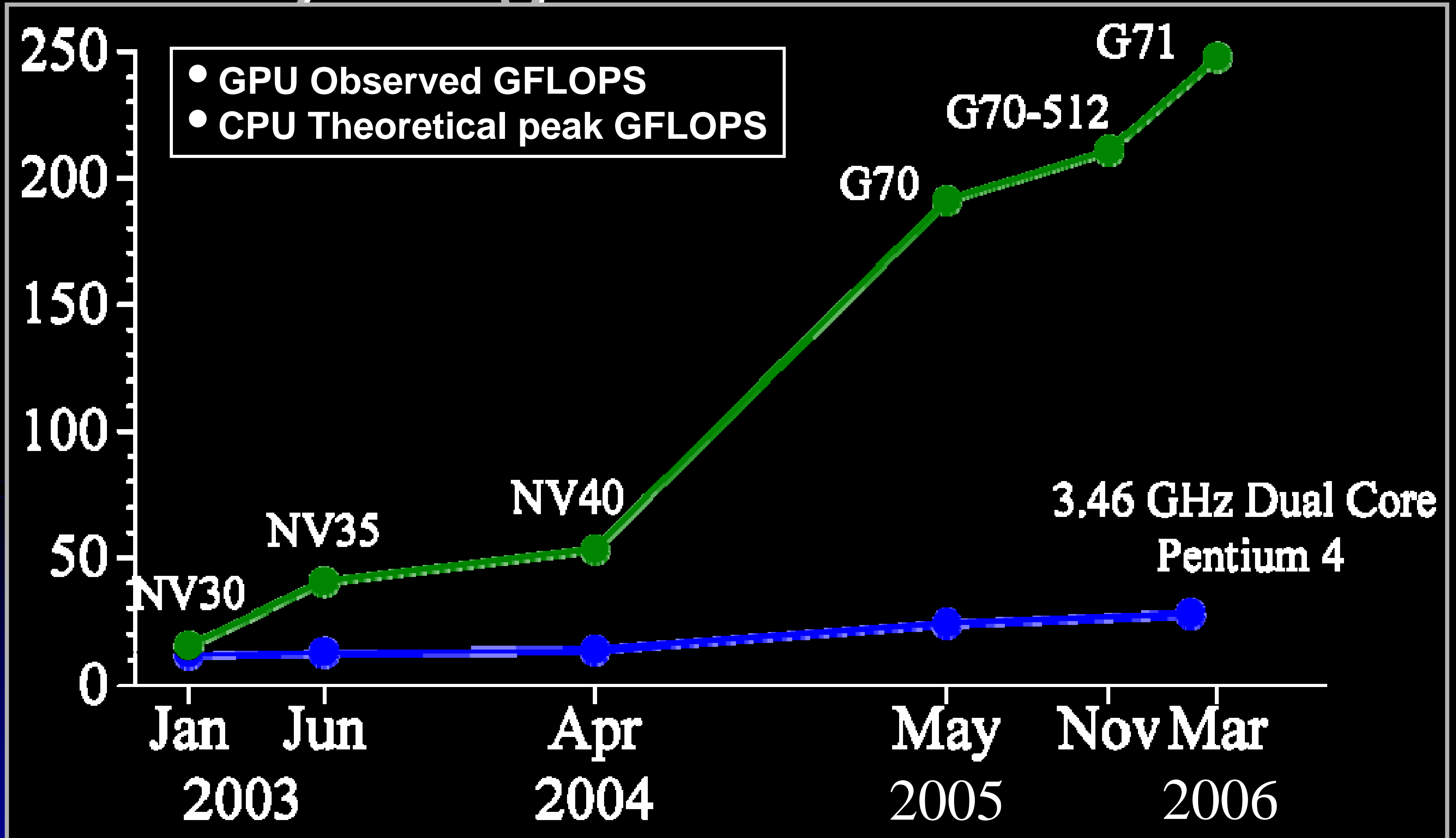
What is a GPU?

- GPU stands for Graphics Processing Unit
- Simply – It is the processor that resides on your graphics card.



- GPUs allow us to achieve the unprecedented graphics capabilities now available in games (ATI Demo)

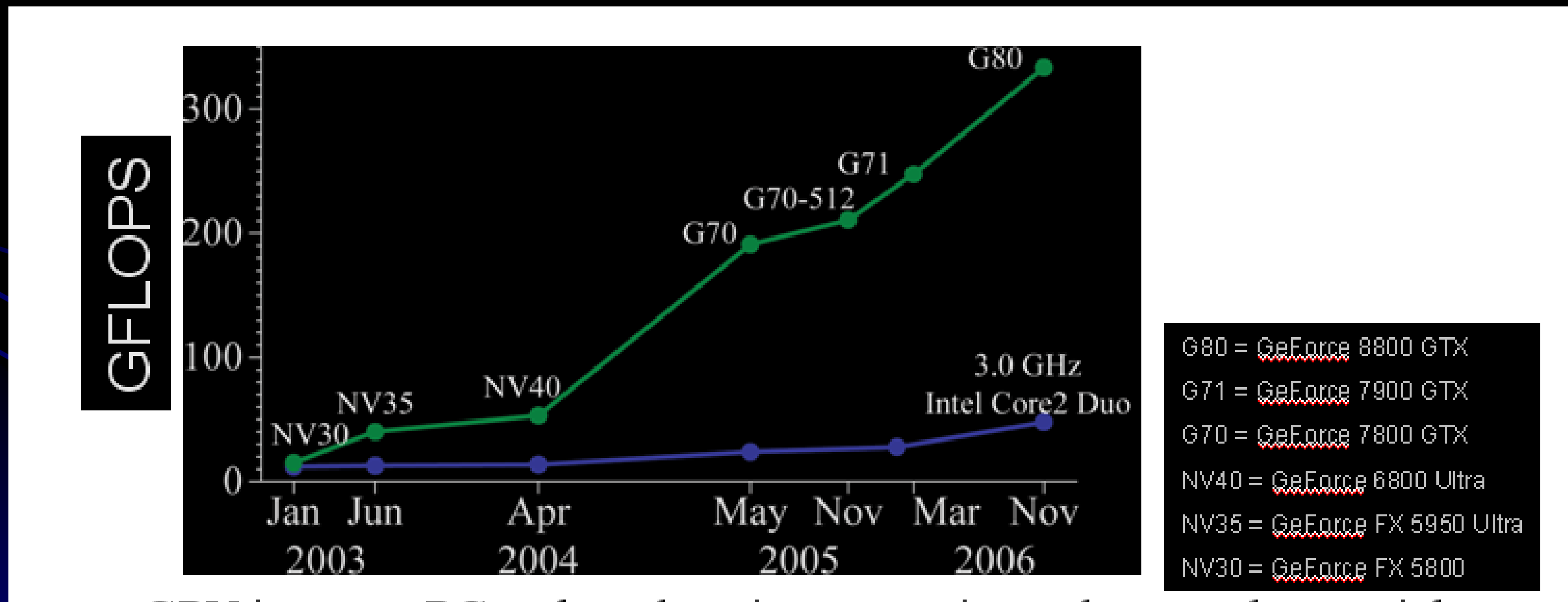
Why Program on the GPU ?



From 2006 GDC Presentation Nvidia

Why Massively Parallel Processor

- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until last couple years, programmed through graphics API



  - GPU in every PC and workstation – massive volume and potential impact

How has this come about ?

- Game design has become ever more sophisticated.
- Fast GPUs are used to implement complex shader and rendering operations for real-time effects.
- In turn, the demand for speed has led to ever-increasing innovation in card design.
- The NV40 architecture has 225 million transistors, compared to about 175 million for the Pentium 4 EE 3.2 Ghz chip.
- The gaming industry has overtaken the defense, finance, oil and healthcare industries as the main driving factor for high performance processors.

GPU = Fast co-processor ?

- GPU speed increasing at cubed-Moore's Law.
- This is a consequence of the data-parallel streaming aspects of the GPU.
- GPUs are cheap ! Put a couple together, and you can get a super-computer.

So can we use the GPU for general-purpose computing ?

NYT May 26, 2003: TECHNOLOGY; From PlayStation to Supercomputer for $50,000:

National Center for Supercomputing Applications at University of Illinois at Urbana-Champaign builds supercomputer using 70 individual Sony Playstation 2 machines; project required no hardware engineering other than mounting Playstations in a rack and connecting them with high-speed network switch

Future Apps Reflect a Concurrent World

- Exciting applications in future mass computing market have been traditionally considered "supercomputing applications"
  - Molecular dynamics simulation, Video and audio coding and manipulation, 3D imaging and visualization, Consumer game physics, and virtual reality products
  - These "Super-apps" represent and model physical, concurrent world
- Various granularities of parallelism exist, but…
  - programming model must not hinder parallel implementation
  - data delivery needs careful management

Yes ! Wealth of applications

Data Analysis

Motion Planning

Voronoi Diagrams

Particle Systems

Geometric Optimization

Force-field simulation

Molecular Dynamics

Graph Drawing

Physical Simulation

Matrix Multiplication

Database queries

Conjugate Gradient

Sorting and Searching

Range queries

Image Processing

Signal Processing

Finance

Radar, Sonar, Oil Exploration

... and graphics too !!

Optimization

Planning

When does "GPU=fast co-processor" work ?

Real-time visualization of complex phenomena

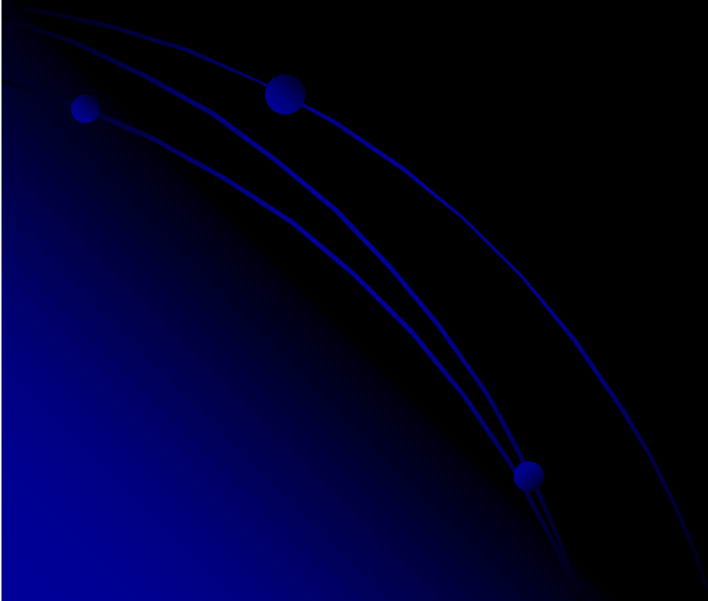
The GPU (like a fast parallel processor) can simulate physical processes like fluid flow, n-body systems, molecular dynamics

In general: Massively Parallel Tasks

When does "GPU=fast co-processor" work ?

Interactive data analysis

For effective visualization of data, interactivity is key



When does “GPU=fast co-processor” work ?

Rendering complex scenes (like the ATI demo)

Procedural shaders can offload much of the expensive rendering work to the GPU. Still not the Holy Grail of “80 million triangles at 30 frames/sec*”, but it helps.

* Alvy Ray Smith, Pixar.

Note: The GeForce 8800 has an effective 36.8 billion texel/second fill rate

General-purpose Programming on the GPU:

What do you need ?

In the abstract:

- A model of the processor
- A high level language

In practical terms:

- Programming tools (compiler/debugger/optimizer/)
- Benchmarking

Follow the language

- Some GPU architecture details hidden, unlike CPUs (Less now than previously).
- OpenGL (or DirectX) provides a *state machine* that represents the rendering pipeline.
- Early GPU programs used properties of the state machine to “program” the GPU.
- Recent GPUs provide high level programming languages to work with the GPU as a general purpose processor

Programming using OpenGL state

- One "programmed" in OpenGL using state variables like blend functions, depth tests and stencil tests

```
glEnable( GL_BLEND ) ;
glBlendEquationEXT ( GL_MIN_EXT ) ;
glBlendFunc(  GL_ONE, GL_ONE ) ;
```

Follow the language

- As the rendering pipeline became more complex, new functionality was added to the state machine (via extensions)

- With the introduction of *vertex* and *fragment programs*, full programmability was introduced to the pipeline.

Follow the language

- With fragment programs, one could write general programs at each fragment

```
MUL     tmp, fragment.texcoord[0], size.x;
FLR     intg, tmp;
FRC     frac, tmp;
SUB     frac_1, frac, 1.0;
```

But writing (pseudo)-assembly code is clumsy and error-prone.

Follow the language

- Finally, with the advent of high level languages like HLSL, Cg, GLSL, CUDA, CTM, BrookGPU, and Sh, general purpose programming has become easy:

```
float4 main(      in float2 texcoords : TEXCOORD0,
                  in float2 wpos : WPOS,
         uniform samplerRECT pbuffer,
                  uniform sampler2D nvlogo) : COLOR
{
   float4 currentColor = texRECT(pbuffer, wpos);
   float4 logo = tex2D(nvlogo, texcoords);
   return currentColor + (logo * 0.0003);
}
```

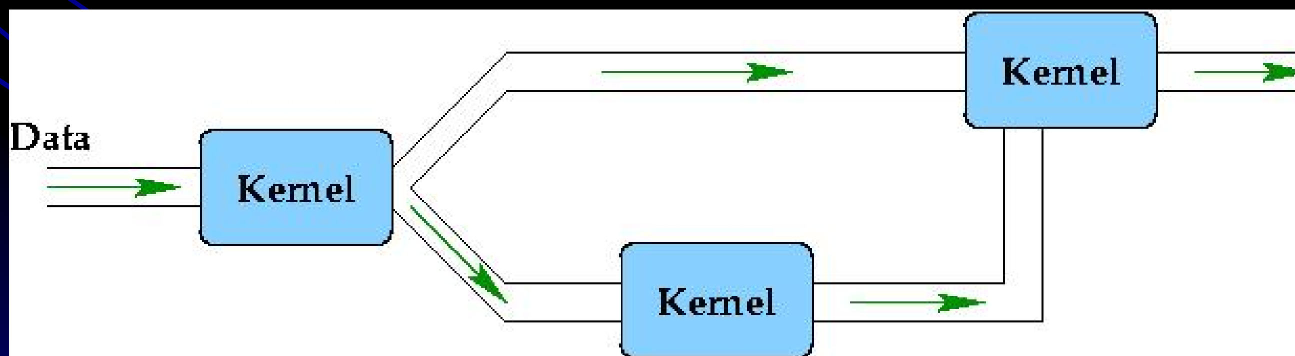
A Unifying theme: Streaming

All the graphics language models share basic properties:

1. They view the frame buffer as an array of “pixel computers”, with the same program running at each pixel (SIMD)
2. Fragments are streamed to each pixel computer
3. The pixel programs have limited state.

What is stream programming?

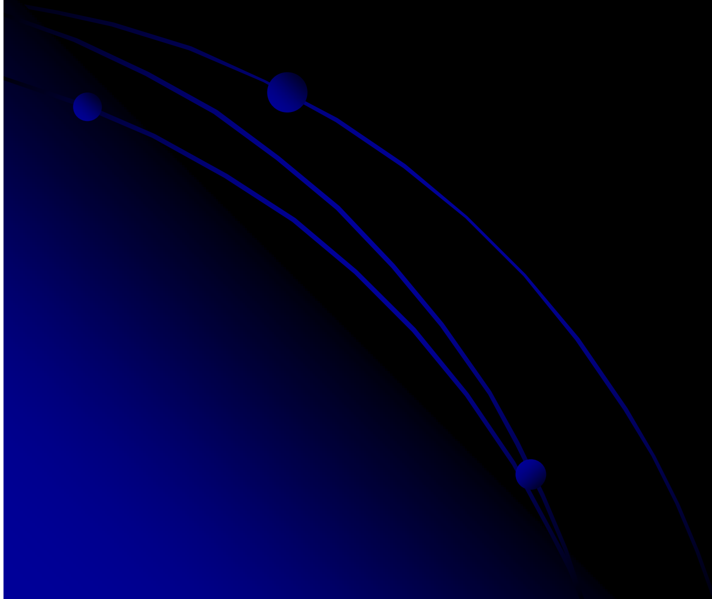
- A stream is a sequence of data (could be numbers, colors, RGBA vectors,…)
- A kernel is a (fragment) program that runs on each element of a stream, generating an output stream (pixel buffer).



Stream Program => GPU

- Kernel = vertex/fragment program
- Input stream = stream of fragments or vertices or texture data
- Output stream = frame buffer or pixel buffer or texture.
- Multiple kernels = multi-pass rendering sequence on the GPU.

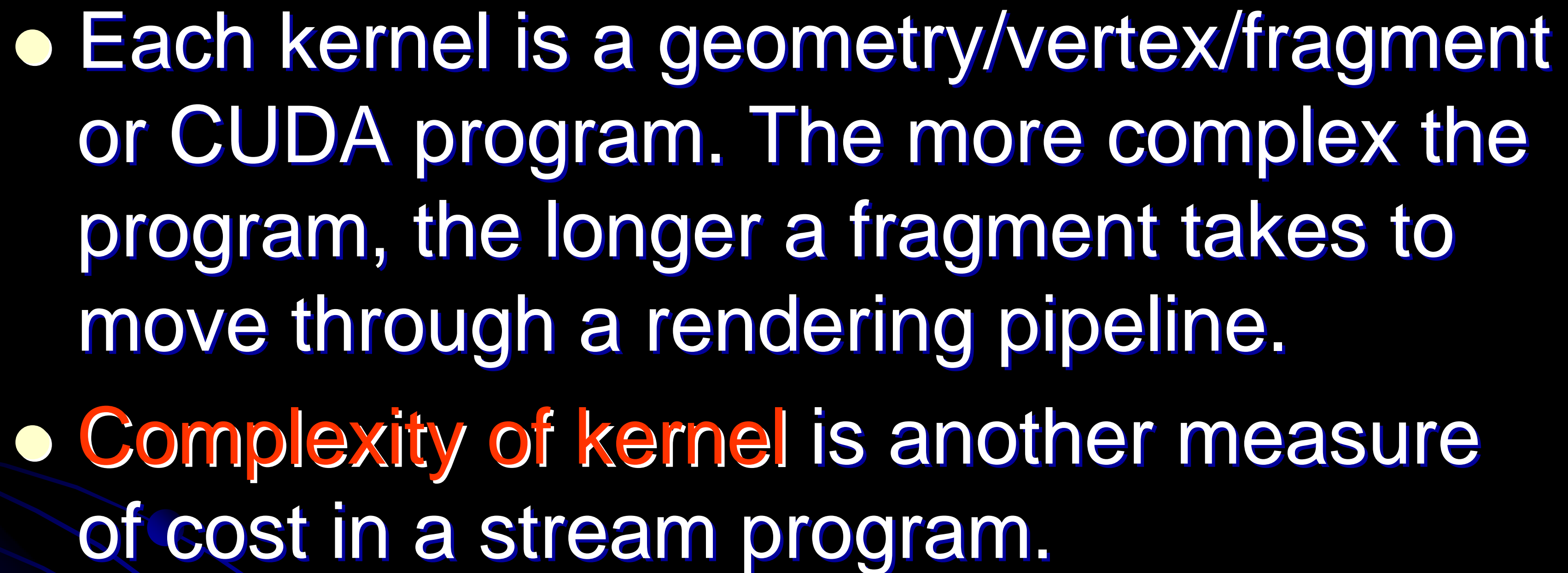
To program the GPU, one must think of it as a (parallel) stream processor.



What is the cost of a program ?

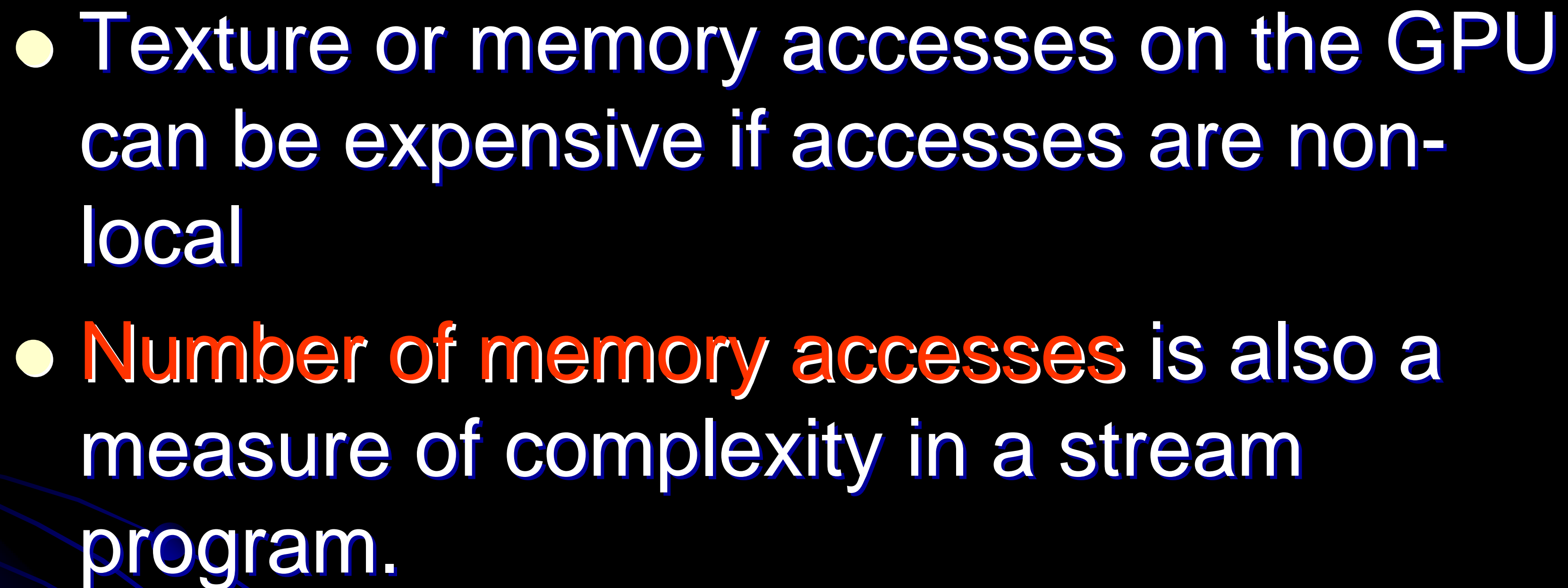
- Each kernel represents one pass of a multi-pass computation on the GPU.
- Readbacks from the GPU to main memory are expensive, and so is transferring data to the GPU.
- Thus, the number of kernels in a stream program is one measure of how expensive a computation is.

What is the cost of a program ?



- Each kernel is a geometry/vertex/fragment or CUDA program. The more complex the program, the longer a fragment takes to move through a rendering pipeline.
- Complexity of kernel is another measure of cost in a stream program.

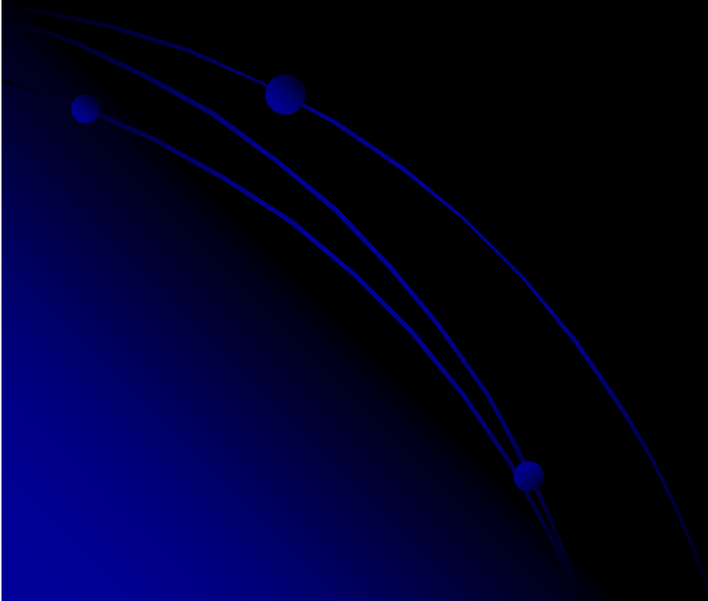
What is the cost of a program ?



- Texture or memory accesses on the GPU can be expensive if accesses are non-local
- Number of memory accesses is also a measure of complexity in a stream program.

What is the cost of a program ?

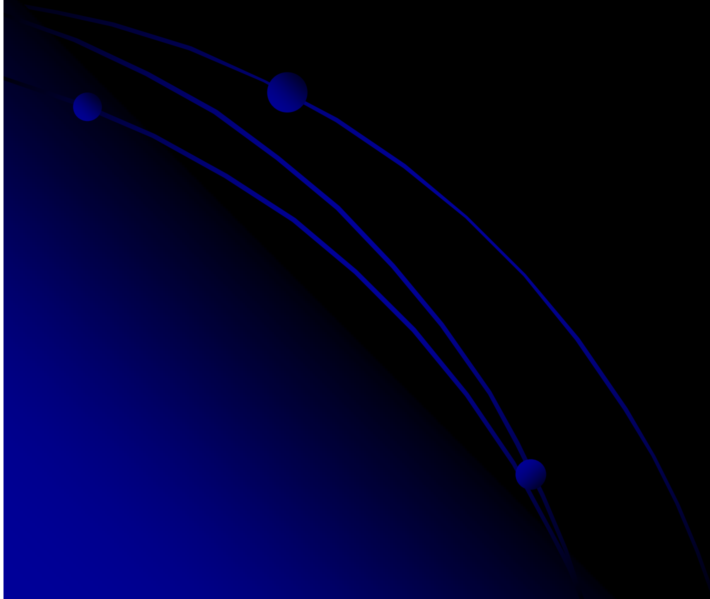
- Conditional Statements do not work well on streaming processors
- Fragmentation of code is also a measure of complexity in a stream program.



The GPGPU Challenge

- Be cognizant of the stream nature of the GPU.
- Design algorithms that minimize cost under streaming measures of complexity rather than traditional measures.
- Implement these algorithms efficiently on the GPU, keeping in mind the limited resources (memory, program length) and various bottlenecks (conditionals) on the card.

What will this course cover ?



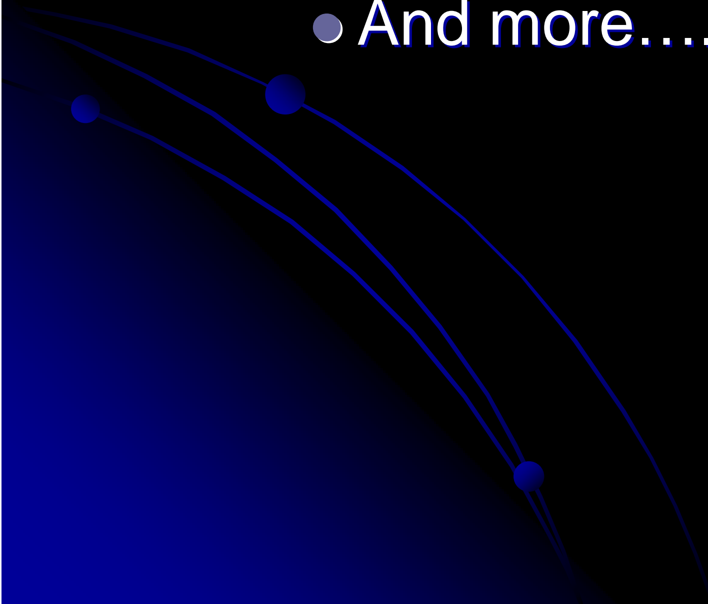
1. Stream Programming Principles

- OpenGL, the fixed-function pipeline and the programmable pipeline
- The principles of stream hardware
- Viewing the GPU as a realization of a stream programming abstraction
- How do we program with streams ?

How should one think in terms of streams ?

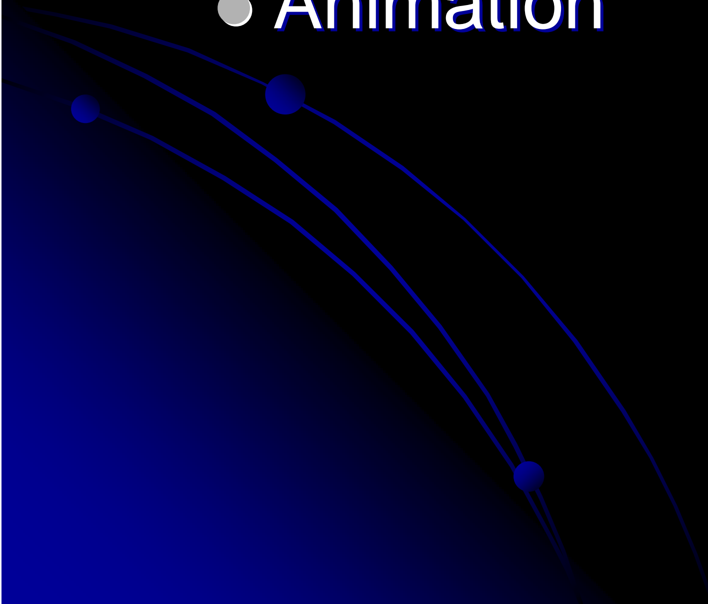
2. Basic Shaders

- How do we compute complex effects found in today's games?
  - Parallax Mapping
  - Reflections
  - Skin and Hair
  - And more….



3. Special Effects

- How do we interact
  - Particle Systems
  - Deformable Mesh
  - Morphing
  - Animation



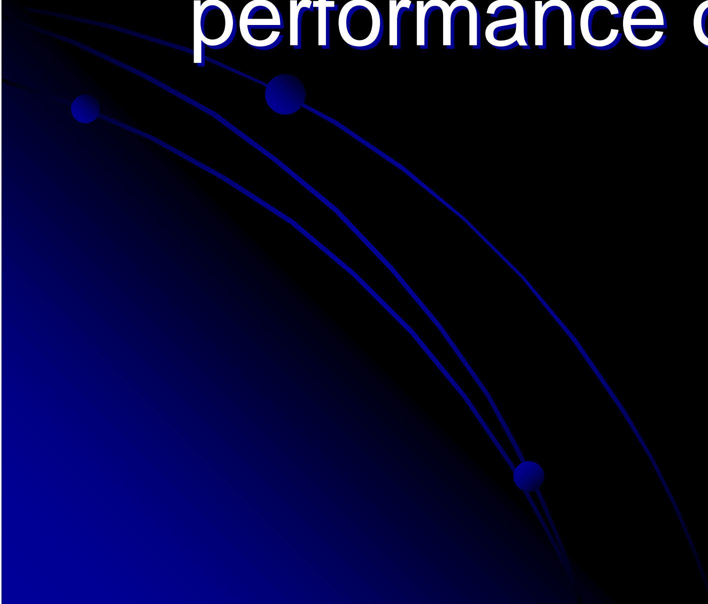
4. GPGPU

- How do we use the GPU as a fast co-processor?
  - GPGPU Languages such as CUDA
  - High Performance Computing
  - Numerical methods and linear algebra:
    - Inner products
    - Matrix-vector operations
    - Matrix-Matrix operations
    - Sorting
    - Fluid Simulations
    - Fast Fourier Transforms
    - Graph Algorithms
    - And More…
  - At what point does the GPU become faster than the CPU for matrix operations ? For other operations ?

(This will be about half the course)

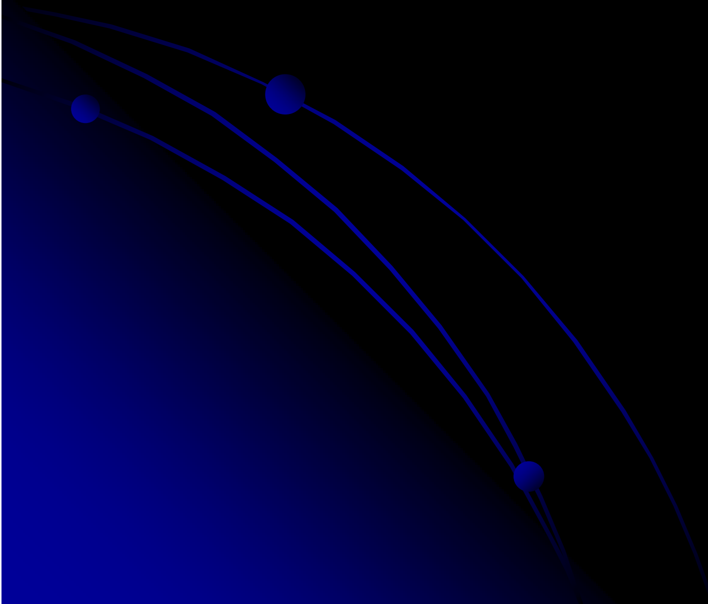
5. Optimizations

- How do we use the full potential of the GPU?
- What makes the GPU fast?
- What tools are there to analyze the performance of our algorithms?



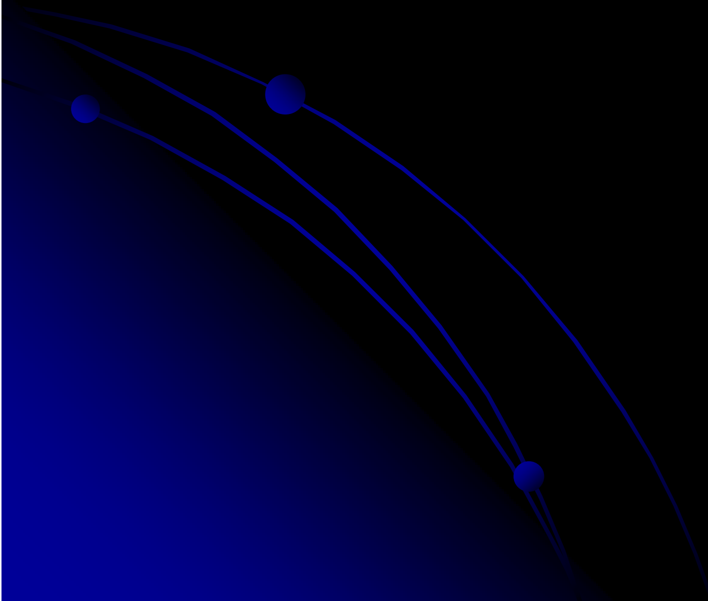
6. Physics, The future of the GPU?

- Physical Simulation
- Collision Detection



7. Artificial Intelligence (The next future of GPU)

- Massive Simulations
- Flocking Algorithms
- Conjugant Gradient



What we want you to get out of this course!

1. Understanding of the GPU as a graphics pipeline
2. Understanding of the GPU as a high performance compute device
3. Understanding of GPU architectures
4. Programming in CG and CUDA
5. Exposure to many core graphics effects performed on GPUs
6. Exposure to many core parallel algorithms performed on GPUs

Main Languages we will use

- OpenGL and CG
  - Graphics languages for understanding visual effects.
  - You should already have an understanding of OpenGL, please see myself or Joe after class if this is not the case
  - We will NOT be using DirectX or HLSL because of the high learning curve
- CUDA
  - GPGPU language. This will be used for any general purpose algorithms. (Only works on NVIDIA cards)
  - We will NOT be using CTM because it is a lower level language than CUDA.

Class URLs

- Blackboard site:
  - Check here for assignments and announcements.
- Course Website

  www.seas.upenn.edu/~cis665
  - Check here for lectures and related articles
  - READ the related articles! (They're good)