Goals and Expectations

• Understand fundamentals of GPU hardware
• Understand fundamentals of parallel programming
• Practical experience in implementing algorithms on highly-parallel heterogeneous systems
  • Heterogeneous computing integrates multiple types of specialized processors
  • Central processing units (CPUs) – Highly efficient single-thread execution using branch prediction, faster clock speeds, etc.
  • Graphics processing units (GPUs) – Highly efficient at when executing the same task on a lot of data but slow at single-thread execution
• Practical experience will give you confidence in interviews and reduce the time to productivity in an academic or industrial setting
Prerequisites

• ECE 3331 – Programming Applications in ECE
• MATH 3321 – Engineering Mathematics
• ECE 3340 – Numerical Methods

What this means in practice
• Linear algebra, differential equations
• C/C++
  • Know how to format and compile your own code
  • Understand how to dynamically allocate memory
  • Understand pointers
  • Be able to implement basic numerical algorithms (ex. Newton’s method)
Required Hardware and Software

• Writing and profiling code is an important part of this class
  • This requires access to a C\C++ compiler and an nVidia GPU
  • There is a training cluster in PGH 235 with the necessary hardware
    • You are welcome to use the lab anytime it is available (it is also a classroom)
    • Remote access will be granted to students of this class
  • Many desktop and laptop systems are equipped with the necessary hardware
    • The CUDA development kit is available from nVidia (https://developer.nvidia.com/cuda-zone)
      • Windows
        • Visual Studio is available for free through Microsoft Dreamspark (https://www.dreamspark.com/)
    • Mac and Linux
      • GCC/G++ is available through your package manager
Practical Recommendations

• If you expect to be doing GPU computing for your research, I recommend investing in a card you can use in a local system
  • GeForce cards, gaming and graphics ($50 – $800)
    • Specifications: http://www.geforce.com/hardware
  • Tesla cards, high performance computing (≈ $3000)

• I generally find code development to be easier in Visual Studio.
  • Excellent IDE and debugger

• Most of your production code will run on a Unix system
  • Develop, debug, and profile on Windows. Move to GCC later.
Textbooks

Recommended

• David Kirk and Wen-mei Hwe, *Programming Massively Parallel Processors*
  • Students can get from Amazon ≈$20 ($8 used)

  • open source and available on my lab website for the 3340 Numerical Methods class: stim.ee.uh.edu

• R.W. Hamming, *Numerical Methods for Scientists and Engineers*
  • Mathematical basis for numerical methods
  • Short and precise explanations for different aspects of numerical methods
  • ≈$14.00 on Amazon.com

• W.H. Press, et al., *Numerical Recipes*
  • Detailed discussions and code for numerical algorithms
  • This book may be helpful for certain projects and is very useful if you pursue a career in scientific programming
  • ≈$82 on Amazon.com
Grading

• Assignments
  • 5% Written homework – work together, turn in your own
  • 25% Programming assignments – individual work
  • 30% Mid-Terms 1 and 2 (20% each)
  • 40% Final Project

• Standard 10-point scale (90+ = A, 80-89 = B, etc.)
Advice/Expectations

• Homework assignments
  • Taken from class lecture (your notes) and similar math problems
  • You’re encouraged to work together, but don’t hand in copied work

• Programming Projects
  • You are expected to work independently
  • Document your code – average one line of comments / line of code
  • Explain “why” in the comments, not “what”

• Exams – Material will be taken from:
  • Homework assignments (often outside material, reading)
  • Lectures
  • Programming assignments (focus on algorithms, not code)
Advice/Expectations

• I expect most of your effort to be focused on programming
  • This includes programming assignments and projects
  • All of this work will be done individually
  • Rubrics will be made available with the assignment, detailing how grading will be done

• There will be two mid-term exams
  • Homework assignments will be representative of the exams
  • Exams will be individual work, but homework is collaborative

• Start your final project early
  • Projects should be confirmed following the first exam
  • Written reports will be ≈5 pages long
    • Detail profiling, performance improvement, and techniques
  • Oral presentations will be on the last week of class and during the final exam session
About Me

• B.S. Computer Science – SWOSU (Oklahoma)
• Ph.D. Computer Science – Texas A&M University
• Beckman Fellowship – University of Illinois

• Background
  • Computer Graphics, Visualization
  • Biomedical Imaging, Microscopy, Spectroscopy
  • High-Performance Computing, GPU Computing

• Personal
  • video games, fencing, climbing
Research Interests

- **Biomedical imaging** – brain and cancer imaging
- **Microscopy** – large data sets
Research Interests

• **GPU Computing** – visualization and image processing

• **Physically-based modeling** – optical systems, fluid dynamics

• **Computer graphics** – interactive visualization and games
A Brief History of Graphics Processing Units

Part 1: Graphics before the GPU
Vector Graphics

- **polygon**: a series of line segments forming a closed planar shape
- **vertex**: a point joining two line segments
- **rotation**: 2D or 3D linear operation applied to a series of vertices

\[ R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \]

\[
\begin{align*}
  v'_1 &= Rv_1 = \begin{bmatrix} x'_1 \\ y'_1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \\
  v'_2 &= Rv_2 \\
  v'_3 &= Rv_3 \\
  v'_4 &= Rv_4 \\
  v'_5 &= Rv_5 \\
  v'_6 &= Rv_6
\end{align*}
\]
Vector Graphics

Rendering a virtual world

1) Describe the world as a series of vertices connected by lines
2) Create a virtual camera
3) Rotate all of the vertices to line the camera up with its position in the virtual world
4) Apply a projection to render the lines to a 2D plane (the screen)
Projections

• **orthogonal projection**: directly eliminate the z-axis component after rotating to the camera

\[
P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\]

\[
\vec{v}_1 = P \vec{v}'_1
\]

\[
\vec{v}_2 = P \vec{v}'_2
\]

\[
\vec{v}_3 = P \vec{v}'_3
\]

\[\ldots\]

\[
\vec{v}_{n-1} = P \vec{v}'_{n-1}
\]

\[
\vec{v}_n = P \vec{v}'_n
\]

• **perspective projection**: scale the \((x, y)\) coordinates as a function of \(z\) (distance from the camera)

• points must be represented as a 4D vector

\[
P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & D & 0 \end{bmatrix}
\]

\[
\vec{v}_n = \frac{P \vec{v}'_n}{w'_n}
\]

where \(\vec{v}'_n = \begin{bmatrix} x'_n \\ y'_n \\ z'_n \\ w'_n \end{bmatrix}\)
Vector Graphics

//calculate the view matrix R
//based on camera position
//calculate the projection matrix P
//based on FOV

ᵦ = vertex[2]

for each polygon p

    for each vertex i in p

        \[ \hat{v}[1] = P \times R \times v_{pi} \]

        if \( i \neq 0 \)

            draw_line(\( \hat{v}[0], \hat{v}[1] \))

        \( \hat{v}[0] = \hat{v}[1] \)

most of the work
Shading – light sources

• All polygons are triangles
• Given a triangle with color:
  \[ c = \begin{bmatrix} r \\ g \\ b \end{bmatrix} \]
• The lighted value is
  \[ \hat{c} = c \times (n \cdot l) \]
  • \( n \) is the surface normal
  • \( l \) is the direction to the light source
Shading

//calculate the view matrix R
//calculate the projection matrix P

\[ v = \text{vertex}[N][3] \] //all vertices
\[ n = \text{vector}[N] \] //all normals
\[ l = \text{vertex}[L] \] //all light positions

for each triangle \( i \)

\[ t[0] = P \times R \times v[i][0] \]
\[ t[1] = P \times R \times v[i][1] \]
\[ t[2] = P \times R \times v[i][2] \]

\[ c = \text{get_color}(t) \]

for each pixel \( p \) in \( t \)

for each light \( j \)

\[ l_{\text{sum}} = l_{\text{sum}} + \cos(n \cdot l[j]) \]

draw_pixel\((p, l_{\text{sum}})\)
Texture Mapping

- *texture mapping*: mapping a 2D image onto the polygons of a 3D object

- provides additional color and shading details w/o extra vertices

- GeForce2 Ti, Grove, 2000
- GeForce2 MX, Creature, 2001
- GeForce2 Go, Toy Soldier, 2002
Texture Mapping

//calculate the view matrix R
//calculate the projection matrix P
v = vertex[N][3]  //all vertices
n = vector[N]  //all normals
l = vertex[L]  //all light positions

for each triangle i
    t[0] = P * R * v[t][0]
    t[1] = P * R * v[t][1]
    t[2] = P * R * v[t][2]

for each pixel p in t
    c = get_texture(p)

    for each light j
        l_sum = l_sum + cos(n \cdot l[j])

    draw_pixel(c, l_sum)
A Brief History of Graphics Processing Units

Part 2: Evolution of the Programmable GPU
Programmable GPUs

1998, Ocarina of Time

1999, System Shock II

2000, Baldur's Gate II

2002, Warcraft III

2004, Fable

DirectX 6
multitexture

DirectX 7

DirectX 8
shader model 1.x

DirectX 9
shader model 2.0 - 3.0

Geforce 256
nVidia
hardware T&L

Geforce 2

Geforce 3
programmable
shading used in XBox

Geforce hardware T&L

Cg shader
language

Radeon 9700
software
floating point

GeForce FX
practical GPGPU

GLSL
shading
language

Voodoo
3dfx Interactive
Texture Mapping

Geforce 2

Geforce 3
programmable
shading used in XBox

Geforce
hardware T&L

Geforce 2

Geforce 3
programmable
shading used in XBox

Geforce hardware
T&L

Texture Mapping

1998, Ocarina of Time

1999, System Shock II

2000, Baldur's Gate II

2002, Warcraft III

2004, Fable
Programmable GPUs

• GeForce 3
  • first programmable shader model (assembly)

• GeForce FX – first practical nVidia card for GPGPU
  • floating point precision
  • SM 2.0 DirectX support
  • Cg – C for Computer Graphics
    • DirectX and OpenGL shading language
GPGPU – dark days/fond memories

- *general purpose GPU programming* (GPGPU): using the shader framework to do general calculations

- Calculate $C = A + B$ where $A$ and $B$ are large matrices
  1) Copy $A$ and $B$ into main memory
  2) Send $A$ and $B$ to the GPU as texture maps
  3) Render a square to the framebuffer (screen memory), assigning both textures as a “material”
  4) Write a simple fragment shader (executed for each pixel):
     1) calculate the current pixel position in the matrix
     2) retrieve the values for $A$ and $B$ using texture fetches
     3) add the values
     4) output the sum as a “color”
  5) Read the data from the framebuffer to main memory
  6) Use the result!
GPGPU – cats/dogs living together

• Debugging was extremely difficult – scale values so that they could be displayed on the screen

• Texture coordinates go from [0 1], so you had to map this to your arrays
  • what’s the texture coordinate for (123, 67) in a 1024x1024 matrix?

• You could get a PhD implementing reasonably complex algorithms using shading languages and rendering rectangles

• Financial companies would hire game programmers to do market analysis
CUDA

• GeForce 8800 – 2006
  • Tesla architecture
  • first CUDA compatible card

• CUDA 1.0 SDK – 2007
  • 8800+ CUDA compatible
  • constantly evolving
    • compute capability
    • current: 6.1

• Parallelism model still based on GPUs
  • data parallel
  • single-instruction multiple data (similar to shader programs)
  • Commonly used in graphics applications