JÜLICH
FORSCHUNGSZENTRUM

Visualization of Complex Functions Using GPUs

September 26, 2012 | Khaldoon Ghanem | German Research School for Simulation Sciences

Outline

JÜLICH FORSCHUNGSZENTRUM

Why GPUs?

- Why to use them?
  - Cheaper cost per performance.
  - Energy efficient.

Device	GFlops	Price	TDP	GFlops/$	GFlops/Watt
Xeon(CPU)	210	2000$	135 Watt	0.105	1.56
GeForce(GPU)	3090	500$	195 Watt	6.180	15.84

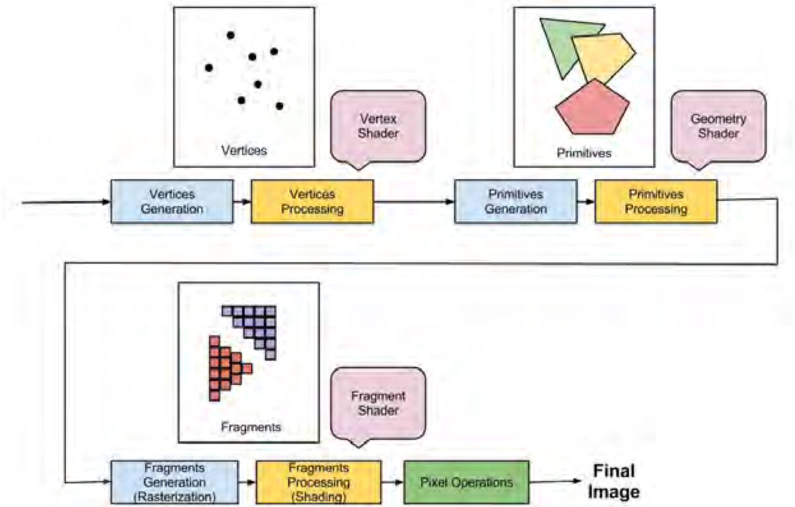
Xeon E5-2690 Vs. Nvidia GeForce GTX 680

- Why cheaper?
  - Different Architecture.
  - Development is driven by huge video game industry.

	2010	2015 (Expected)	
HPC	$26 billion	$36 billion	[HPC advisory council]
Video Game	$67 billion	$112 billion	[Research firm Gartner]

HPC Vs. Video Games Global Market

Graphics Pipeline



Mitglied der Helmholtz-Gemeinschaft

GPU Architecture - Simplified

- Too many ALUs .
- Very simplified control unit.
- Several ALUs (a Warp) share the same control unit.
  - They execute the same instruction.
  - Branching is possible but costly.
- Very small cache.
- Memory latency is hidden by computation.

Programming GPUs

Graphics Programming

- Common Shading Languages
  - HLSL by Microsoft.
  - GLSL by Khronos Group.
  - Cg by Nvidia.
- A program on GPU: Shader
- A collection of fragments is created and the fragment shader is executed for each fragment.
  There is also Vertex shader for vertices and Geometry shader for primitives.

General Programming

- Common Frameworks
  - CUDA by Nvidia.
  - OpenCL by Khronos Group.
- A program on CPU: Host
- A program on GPU: Kernel
- A collection of threads is created and the kernel is executed for each thread.

Outline

GPU in a Nutshell

Fractals - A Simple Fragment Shader

Domain Coloring

Visualizing Spherical Harmonics

Marching Tetrahedra

Summary

Visualizing Mandelbrot Set

Mandelbrot Set

Points c in complex plane for which the series $Z_{n+1} = Z_n^2 + c$ remains bounded.

- Embarrassingly parallel! Each point processed independently.
- Make rectangle covering screen.
- Rectangle's Fragments created by OpenGL and cover screen.
- No need for vertex and geometry shaders: primitives (rectangle) and vertices (its 4 corners) unchanged.
- Fragment Shader: Map each fragment to a point in complex plane and color it using escape time algorithm.

The Shader

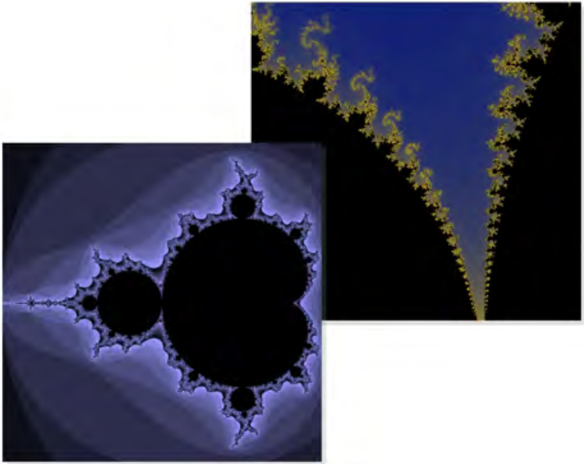
```
//Mandelbrot Set
#version 130
uniform mat3 windowToComplex;
uniform sampler1D tex;
out vec4 gl_FragColor;
void main(){

    vec3 c = windowToComplex*vec3(gl_FragCoord.x, gl_FragCoord.y,1);

    vec2 z = vec2(0, 0);
    int iter;
    int maxIter = 100;
    for (iter = 0; iter < maxIter; iter++){
        z = vec2((z.x*z.x - z.y*z.y)+c.x, (2.0*z.x *z.y)+c.y);
        if (length(z) > 2.0)
            break;
    }

    float colorIndx = float(iter) / float(maxIter);
    gl_FragColor = texture(tex, colorIndx);
}
```

Demo



Outline

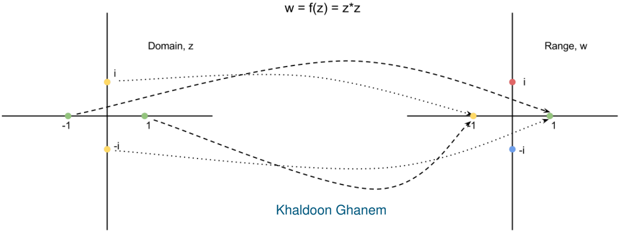
- GPU in a Nutshell
- Fractals - A Simple Fragment Shader
- Domain Coloring
- Visualizing Spherical Harmonics
- Marching Tetrahedra
- Summary

The Idea

Domain Coloring

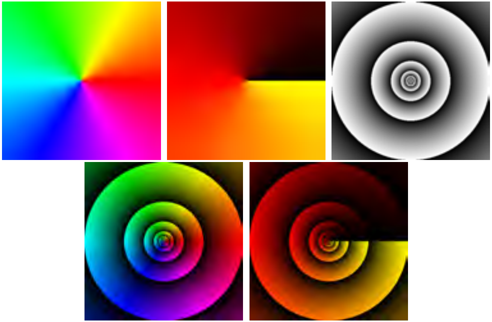
It is a method for visualizing complex functions of complex variables $f : \mathbb{C} \to \mathbb{C} : w = f(z)$

- Cover the range complex plane with some color map i.e. give a color to each w.
- For each point z of the domain complex plane , compute $f(z)$ then color z with the corresponding color of $w = f(z)$.

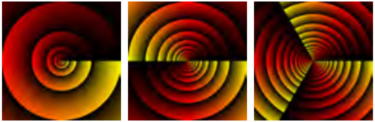


Choosing Color Map

- Using some image. Usually not easy to interpret!
- Using some mathematical formula:
  - Choose color hue according to the argument $arg(w)$ from a smooth color sequence (gradient).
  - Choose color brightness according to the fractional part of the $log_2|w|$



Reading the plot



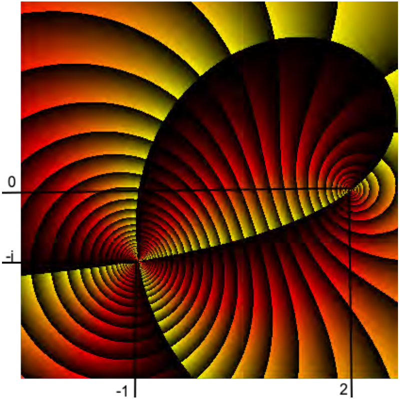
z z^2 z^3

z^k: ring accumulates around 0, color cycles k times. Useful for spotting zeros of kth order.

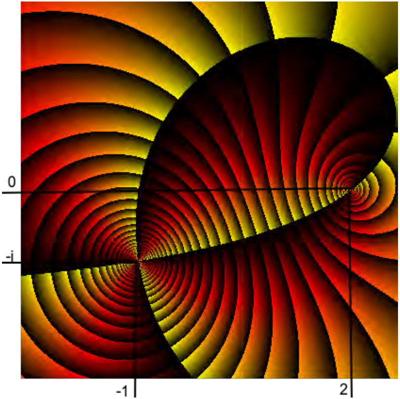
$1/z$ $1/z^2$ $1/z^3$

$1/z^k$: rings diverge from 0, color cycles k times in the opposite direction. Useful for spotting poles of kth order.

What is this function?



What is this function?



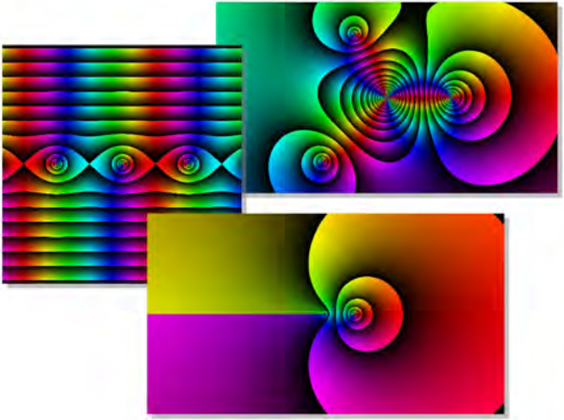
$$\frac{(z-2)^2}{(z+1+i)^4}$$

Using GPU

The basic idea is the same as for Mandelbrot set.

- Make a rectangle covering the screen.
- Rectangle's Fragments will be created by OpenGL and cover the screen.
- No need for vertex and geometry shaders: primitives (rectangle) and vertices (its 4 corners) are left unchanged.
- Fragment Shader:
  - Map each fragment to a point in complex plane z.
  - Compute w = f(z).
  - Get the color of w according to the used color map.
  - Color the fragment.

Demo



Outline

Visualizing Spherical Harmonics

Spherical Harmonics

Complex functions on the unit sphere.

$Y_\ell^m : [0,\pi]x[0,2\pi) \to \mathbb{C} : Y_\ell^m(\theta,\varphi) = \sqrt{\frac{(2\ell+1)}{4\pi}\frac{(\ell-m)!}{(\ell+m)!}}\, P_\ell^m(\cos\theta)\, e^{im\varphi}$

where P_ℓ^m are the associated Legendre polynomials.

How to plot them?

- Plot 3D surface defined by the following implicit relation: $r = |Y_\ell^m(\theta,\varphi)|$
- Color the surface according to $arg(Y_\ell^m(\theta,\varphi))$ using some smooth color sequence as we did in Domain Coloring Method

Lighting Spherical Harmonics

For lighting,we need Normals!

- Think of the surface as an isosurface of the scalar field: $F(r,\theta,\varphi)=\sqrt{Y_\ell^m(\theta,\varphi)\overline{Y_\ell^m}(\theta,\varphi)}-r$ with isovalue 0.
- Then the gradient of the field $\nabla F(r,\theta,\varphi)$ is normal its isosurfaces.
- $\nabla F=\frac{\partial F}{\partial r}\hat{\boldsymbol{r}}+\frac{1}{r}\frac{\partial F}{\partial\theta}\hat{\boldsymbol{\theta}}+\frac{1}{r\sin\theta}\frac{\partial F}{\partial\varphi}\hat{\boldsymbol{\varphi}}$
- $\hat{\boldsymbol{r}}=\sin\theta\cos\phi\hat{\boldsymbol{\imath}}+\sin\theta\sin\phi\hat{\boldsymbol{\jmath}}+\cos\theta\hat{\boldsymbol{k}}$
- $\hat{\boldsymbol{\theta}}=\cos\theta\cos\phi\hat{\boldsymbol{\imath}}+\cos\theta\sin\phi\hat{\boldsymbol{\jmath}}-\sin\theta\hat{\boldsymbol{k}}$
- $\hat{\boldsymbol{\varphi}}=-\sin\phi\hat{\boldsymbol{\imath}}+\cos\phi\hat{\boldsymbol{\jmath}}$
- $\frac{\partial F}{\partial r}=-1$
- $\frac{\partial F}{\partial\theta}=\frac{\frac{\partial\overline{Y}}{\partial\theta}Y+\frac{\partial Y}{\partial\theta}\overline{Y}}{2\sqrt{Y\overline{Y}}}=\frac{Re[\frac{\partial\overline{Y}}{\partial\theta}Y]}{|Y|}$
- $\frac{\partial F}{\partial\varphi}=\frac{Re[\frac{\partial\overline{Y}}{\partial\varphi}Y]}{|Y|}$

Using the GPU

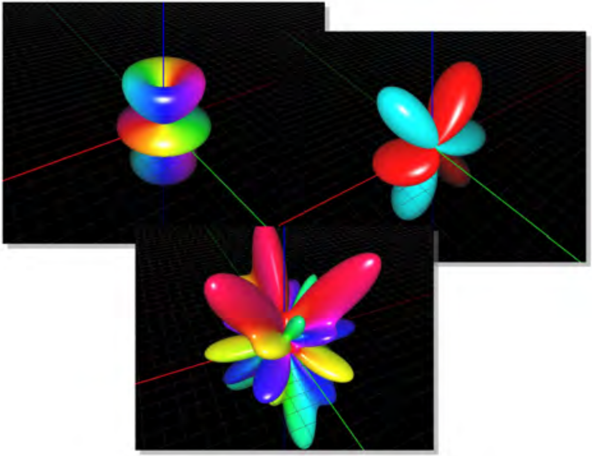
- Generate a unit sphere (vertices and triangles).
- Vertex shader: For each vertex
  - Retrieve vertex's angles (θ, φ)
  - Compute spherical harmonic $Y_\ell^m(\theta, \varphi)$
  - Modify vertex coordinates such that $r = |Y_\ell^m(\theta, \varphi)|$
  - Modify vertex color according to $arg(Y_\ell^m(\theta, \varphi))$ using some smooth color sequence (gradient).
  - Compute partial derivatives of the spherical harmonic and use them to compute the gradient vector.
  - Modify vertex normal such that it points in the direction of the gradient.
- No need for geometry and fragment shaders: triangles and their shading are left alone.

Linear Combination

Easily extendible to a linear combination of spherical harmonics

- Scalar field becomes $F(r, \theta, \varphi) = |\sum_\ell \sum_m c_{\ell,m} Y_\ell^m(\theta, \varphi)| - r$
- compute the value and partial derivative of each spherical harmonic
- For computing partial derivatives, derivative of a linear combination is the linear combination of the derivatives.
- This can be used for visualizing real spherical harmonics.
- Spherical harmonics forms a complete set of orthonormal functions thus any any square-integrable function complex function on a unit sphere can be expanded as linear combination of them.

Demo



Outline

Visualizing 3D complex functions

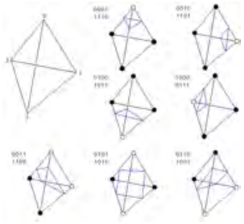
We often need to visualize discrete complex function define in 3D space $f : \mathbb{R}^3 \rightarrow \mathbb{C}$ like wave functions resulting form quantum calculations.

- Specify one absolute value to visualize.
- Calculate isosurface of the absolute value using marching tetrahedra.
- Color the isosurface according to the argument using some smooth color sequence.
- If necessary, change isovalue and repeat process to gain more info.

Marching Tetrahedra

A method for extracting isosurfaces of real scalar fields.

- Function values are given on structured mesh points.
- Divide each mesh cell into six tetrahedra.
- For each tetrahedron:
  - Classify each tetrahedron's vertex as below or above isosurface (white or black).
  - Isosurface passes only through edges with opposite colored vertices.
    Determine intersection point through interpolation.
  - 4 vertices with 2 states lead to 16 different tetrahedron's states.
    Use tetrahedron's state index to generate triangles.



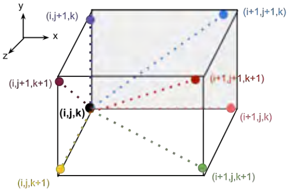
Avoid Duplicated Vertices

In the previous algorithm, isosurface vertices shared between adjacent tetrahedra are duplicated.

- Split the algorithm into two stages: Generating Vertices and Generating Triangles.
- Generating Vertices:
  - For each edge, check whether it is cut by the isosurface.
  - If so, calculate the intersection point(Isosurface Vertex).
  - Store the vertex in hash table indexed by edge.
- Generating Triangles:
  - For each mesh cell, process all six tetrahedra.
  - For each tetrahedra, calculate state index.
  - Using tetrahedron's state index, get its triangulation.
  - Get vertices' pointers by looking up edge-vertex hash table.
  - Generate triangles using pointers to vertices (not vertices directly).

Identifying Edges

- Previous algorithm requires a unique ID for each edge.
- Each mesh point (i,j,k) can be associated uniquely with seven edges.
  - 1: (i,j,k)-(i, j, k+1)
  - 2: (i,j,k)-(i, j+1, k)
  - 3: (i,j,k)-(i, j+1, k+1)
  - 4: (i,j,k)-(i+1, j, k)
  - 5: (i,j,k)-(i+1, j, k+1)
  - 6: (i,j,k)-(i+1, j+1, k)
  - 7: (i,j,k)-(i+1, j+1, k+1)



- So an edge can be identified by a tuple of the associated mesh point id and a number between 1 and 7 identifying the edge for the mesh point.

Implementing on the GPU

- Much more convenient to be implemented using GPGPU (OpenCL or CUDA).
- Previous algorithm won't work out of the box.
  - No dynamic allocation of memory on GPU.
  - Memory allocation and deallocation is done before and after computation but not during.
- Algorithm Outline
  - Count the number of generated vertices beforehand.
  - Allocate vertex array.
  - Know where to store and retrieve each vertex.
  - Generate vertices.
  - Count the number of generated triangles beforehand.
  - Allocate triangle array.
  - Know where to store each triangle.
  - Generate triangles.

Implementing on the GPU

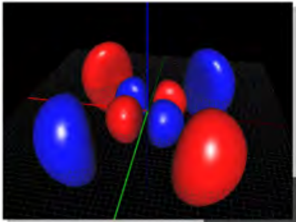
- Some Details:
  - Run a kernel '*CountVerts*' for each mesh point.
    it should compute the number of vertices associated with each mesh point and store them in array **vertsNum**.
    vertex-mesh point association is the same as edge-mesh point association.
  - Apply a prefix-scan on **vertsNum** and store the result in array **vertBaseAddress**.
  - Total number of vertices equal the sum of last entry in **vertsNum** and last entry in **vertAddressBase**.
  - In host program, allocate memory on GPU necessary for vertex array.
  - Run kernel '*GenerateVerts*' for each mesh point. it compute the vertices associated the mesh point and store then in the vertex array starting from entry **vertAddressBase[mp]**
  - Similar for triangles.

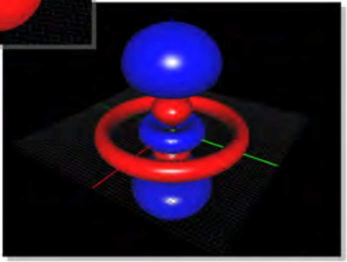
Notes

- Considerable speed-up can be obtained by considering only mesh points that actually has a non-zero number of associated vertices.
  - requires building yet another array for knowing which mesh points are ’active’.
- For lighting, we need normals.
  - Compute normals at mesh points using central differences.
  - Then interpolate to get the normals at isosurface vertices.

JÜLICH FORSCHUNGSZENTRUM

Demo





Mitglied der Helmholtz-Gemeinschaft

Outline

Summary

- GPUs are really cheaper than CPUs,if used appropriately!
- Complex function of complex variable: $\mathbb{C} \rightarrow \mathbb{C}$
  - Domain Coloring Method
  - Fragment Shader
- Complex function on real unit sphere: $[0, \pi]x[0, 2\pi) \rightarrow \mathbb{C}$
  - Encode absolute value in the radius, encode argument in color.
  - requires normals:Gradient
  - Vertex Shader
- Discretized complex function in 3D space: $\mathbb{R}^3 \rightarrow \mathbb{C}$
  - visualize isosurfaces of the absolute value.
  - encode argument in color.
  - Marching Tetrahedra Method.
  - requires normals:central differences on voxels then interpolation on isosurface vertices.
  - OpenCL or CUDA kernel

Thanks for Listening...
Questions?