MIC-GPU: High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

GPU-Acceleration of Individual Components

Klaus Mueller
Computer Science
Center for Visual Computing
Stony Brook University

Overview

What to expect:
- details on the parallelization of various fundamental CT reconstruction algorithms
- details and fragment code for graphics-style GPU implementations of these
- insights on GPGPU-style implementations of these
- comparisons of results obtained with both

Decomposition

\[ P \cdot p_i = \sum_{j=0}^{N-1} (v_j \cdot w_j) \]

\[ B \cdot v_j = \sum_{i=0}^{M-1} (p_i \cdot w_j) \]

- FBP

\[ v_j = \sum_{p \in P_{b,a}} B \cdot w_j = \sum_{p \in P_{b,a}} S \cdot t \]

- Algebraic

\[ v_j = \frac{1}{\sum_{p \in P_{b,a}} w_j} \left( \sum_{p \in P_{b,a}} P \cdot w_j \right) \]

- OS-EM

\[ v_j = \frac{1}{\sum_{p \in P_{b,a}} w_j} \left( \sum_{p \in P_{b,a}} (p_i \cdot w_j) \right) \]

Kernel-Centric Reconstruction

Algebraic

EM

FBP

Projection
Backprojection
Correction
Update
Kernel-Centric Reconstruction

Algebraic

EM

FBP

P P B

P B U

P B U

C C B

C U U

C U U

compute intensive kernel

Projection Backprojection Correction Update

Backprojection

Sample in projection space, voxel-driven

Volume Representation

3D texture is not used (no write support)
2D texture stacks are used
Axis aligned (x, y, z), easy to compute and store

Transformation Matrix

A 4x4 matrix $M$ transforms 3D voxel coordinates to 2D pixel coordinates on the detector

Perform perspective divide if necessary (cone-beam)

Composition of the matrix from graphics point of view

- model-view matrix
- projection matrix
- translation / scaling matrix

\[
\begin{bmatrix}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & a_{12} & a_{13} \\
  a_{20} & a_{21} & a_{22} & a_{23} \\
  a_{30} & a_{31} & a_{32} & a_{33}
\end{bmatrix}
\begin{bmatrix}
  x_v \\
  y_v \\
  z_v \\
  1
\end{bmatrix}
= \begin{bmatrix}
  x_h \\
  y_h \\
  z_h \\
  w_h
\end{bmatrix}
\]

$P_v(U,V) = \left( \frac{x_h}{w_h}, \frac{y_h}{w_h} \right)$
Decomposition

Decomposition: Model/View

\[
\begin{bmatrix}
u_x & u_y & u_z & -\bar{u} \cdot \bar{s} \\
v_x & v_y & v_z & -\bar{v} \cdot \bar{s} \\
n_x & n_y & n_z & -\bar{n} \cdot \bar{s} \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x_v \\
y_v \\
z_v \\
1
\end{bmatrix}
\]

Decomposition: Projection

\[
\begin{bmatrix}
\frac{-n}{2} & 0 & 0 & 0 \\
0 & \frac{n}{2} & 0 & 0 \\
0 & 0 & \frac{f \pm n}{w + f} & \frac{2h}{w + f} \\
0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
u_x & u_y & u_z & -\bar{u} \cdot \bar{s} \\
v_x & v_y & v_z & -\bar{v} \cdot \bar{s} \\
n_x & n_y & n_z & -\bar{n} \cdot \bar{s} \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x_v \\
y_v \\
z_v \\
1
\end{bmatrix}
\]

Decomposition: Window

\[
\begin{bmatrix}
\frac{w}{2} & 0 & 0 & 0 \\
0 & \frac{w}{2} & 0 & 0 \\
0 & 0 & \frac{f \pm n}{w + f} & \frac{2h}{w + f} \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
u_x & u_y & u_z & -\bar{u} \cdot \bar{s} \\
v_x & v_y & v_z & -\bar{v} \cdot \bar{s} \\
n_x & n_y & n_z & -\bar{n} \cdot \bar{s} \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x_v \\
y_v \\
z_v \\
1
\end{bmatrix}
\]

n, f: near/far viewing frustrum extent

source
Full Transformation Matrix

Embedded transformation on graphics hardware (OpenGL)
4th coordinate (w) contains voxel depth value

\[
\begin{bmatrix}
\frac{2a}{\pi} & 0 & 0 & 0 \\
0 & \frac{2a}{\pi} & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
u_x \\ v_x \\ v_z \\ n_z \\
\end{bmatrix}
= 
\begin{bmatrix}
x_v \\ y_v \\ z_v \\ w_v \\
\end{bmatrix}
\]

\[w_h = n_x \cdot x_v + n_y \cdot y_v + n_z \cdot z_v - \bar{n} \cdot \bar{S} = \bar{n} \cdot \bar{V} - \bar{n} \cdot \bar{S}\]

\[w_h \Rightarrow \text{voxel depth}\]

Implementation: GPGPU

Acquire transformation matrix \(M\)
Pass into fragment shader
Orthographically render the volume slice to be reconstructed
In the fragment shader, compute detector-plane coordinates:
- generate homogenous location coordinates for each fragment
- multiply by \(M\)
Then, also in the fragment shader:
- perform perspective-divide \((x, y)\)
- extract \(w\) to compute depth weights
- interpolate densities (bilinear interpolation)

Fragment Code Sample

```cpp
float fVoxel = 0;
float4 vWorldPos = float4(wpos.x, y, wpos.y, 1); // voxel coordinates
for (int s = 0; s < iProjNo; s++) { // loop over projections
    // transform to detector positions, using 1st and 2nd matrix rows
    float2 vDetPos = float2(dot(vMatR0, vWorldPos), dot(vMatR1, vWorldPos));
    // compute depth values, using 3rd matrix row
    float fDepth = dot(vMatR3, vWorldPos);
    vDetPos /= fDepth; // perspective divide
    fWeight = pow(fSO/fDepth, 2); // compute depth weights
    // sample+accumulate
    fVoxel += fWeight* tex3D(texProj3D, float3(vDetPos.x, vDetPos.y, idx+0.5));
}
```
Implementation: Graphics

Vertex and fragment shader

Transformation in the vertex shader, much faster!
- acquire transformation matrix $M$
- pass into the vertex shader
- orthographic rendering of the slice to be reconstructed
- pass 3D coordinates of the vertices into the vertex shader
- in the vertex shader
  - multiply proxy polygon vertices coordinates with $M$

Fragments get rasterized

Only perform texture sampling in the fragment shader
- perspective-divide on each fragments, compute depth-weight
- interpolate densities (bilinear interpolation)

Code Sample

Fragment code:

```glsl
float fDepthRCP = 1/vTransPos.w;  // reciprocal of voxel depth
float2 vDetPos = vTransPos.xy*fDepthRCP;  // perspective division
float fWeight = pow(fSO*fDepthRCP, 2);  // compute voxel weights
float fOldVal = texRECT(texSlice, wpos);  // sample original value
float fNewVal = texRECT(texProj, vDetPos);  // sample incoming value
float fOutVal += fOldVal + fNewVal*fWeight;  // accumulate
return fOutVal;
```

GL code in host program:

```c
glBegin(GL_POLYGON);
glTexCoord3f(v00.x, v00.y, v00.z); glVertex3f(v00.x, v00.y, v00.z);
glTexCoord3f(v10.x, v10.y, v10.z); glVertex3f(v10.x, v10.y, v10.z);
glTexCoord3f(v11.x, v11.y, v11.z); glVertex3f(v11.x, v11.y, v11.z);
glTexCoord3f(v01.x, v01.y, v01.z); glVertex3f(v01.x, v01.y, v01.z);
glEnd();
```

Graphics Pipeline Revisited

Pipeline 1: GPU as a Programmable Multi-Processor (MP-GPU)

Fragments contain the (x,y,z) voxel coordinates
**Pipeline 2: GPU as a Programmable Graphics Processor (AG-GPU)**

![Diagram of GPU pipeline]

**Geometry Stage**
- Vertex Shader
- Transformation

**Fragment Stage**
- Fragment Shader
- Perspective divide
- Sample, weight
- Accumulate

**Graphics Pipeline Benefits**

Graphics-aware pipeline (AG-GPU) is considerably faster (~3×) than MP-GPU
- graphics facilities are hardwired!

There are further features that have their origins in graphics and come with GPUs:
- early fragment kill \(\Rightarrow\) eliminate fragments based on some condition before they even enter the fragment processor (AG-GPU+)
- hardwired 32-bit floating-point precision linear interpolations, matrix and vector arithmetic (+, -, *), frame-buffer blending and compositing
- RGBA parallelism


**RGBA Parallelism**

Exploit geometric mapping parallelism

**Example:** Feldkamp Cone-Beam Reconstruction

360 projections (1024\(^2\), general position), 512\(^3\) volume

![Graph showing performance comparison]

**Table:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Spec. time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>135</td>
</tr>
<tr>
<td>MP-GPU</td>
<td>24.8</td>
</tr>
<tr>
<td>AG-GPU</td>
<td>8.9</td>
</tr>
<tr>
<td>AG-GPU+</td>
<td>4.8</td>
</tr>
</tbody>
</table>

**Note:** Performance comparison for different methods.
Expressed in Projections/Sec.

360 projections, $512^3$ volume

Original

GPU-recon

GPU Enables Visual CT

High reconstruction frame rate enables injection of occasional volume rendering step

Also enables D²VR: real-time volume visualization directly from projection data


Next: Iterative Algorithms

SART, EM

All require a projection simulation step
  • should be as accurate as possible

Projection

Sampling in volume space, ray-driven

Raycasting methods [Krueger’03]
  • represent volume as a single 3D texture
  • rasterize the detector image, generate fragments
  • cast rays, sample and accumulate
  • no support for easy write
Slice-based method [Rezk-Salama'00]
- represent volume as a stack of 2D slices
- project each slice onto detector plane using texture mapping
- composite buffer to accumulate

3D transformation & texture mapping are intrinsic operations

Volume slice stacks
- select the slice most parallel to the detector
- non-uniform sampling, on-slice NN/BI
- perform as well as conventional sampling [Xu'06]
  - grid interpolated
  - box line integrated
  - etc.

Buffer = Buffer + Incoming

Avoid simultaneous read and write

Bounce back and forth between two buffers in each pass

Investigated various schemes in terms of accuracy:

It was shown that the convenient slice-interpolated scheme is qualitatively competitive to the more involved ones listed here.

Implementation

Create polygon proxies for each slice
Associate texture (slice content) with the polygon
Render polygons
  • 4 vertices carry voxel coordinates
  • 3D transformation is done on GPU, for ALL points in the polygon
  • volume slice gets interpolated in the fragment shader

Pixel-Wise Operation

Correction
  • algebraic: scanned projection – simulated projection
  • EM: scanned projection / simulated projection

Weighting, slice updating...

```
glMatrixMode(GL_MODELVIEW); glLoadIdentity();
glMatrixMode(GL_PROJECTION); glLoadIdentity();
gluOrtho2D(0, w, 0, h); glViewport(0, 0, w, h);
gBegin(GL_POLYGON);
gMultiTexCoord2iARB(GL_TEXTURE0_ARB, 0, 0);
gMultiTexCoord2iARB(GL_TEXTURE0_ARB, w, 0);
gMultiTexCoord2iARB(GL_TEXTURE0_ARB, w, h);
gMultiTexCoord2iARB(GL_TEXTURE0_ARB, 0, h);
....
gEnd();
```

```
float diff   = texRECT(tex0,texcoords0) - texRECT(tex1, texcoords1);
float weight = texRECT(tex2, texcoords2);
float corr = (weight == 0) ? 0: diff / weight;
```

Attenuation Modeling

Two sliced volumes
  • attenuation A + emission C (under reconstruction)
  • first normalize A to [0…1]
  • then compute T = 1 - A

Composition
  • front-to-back: emission + transparency buffer
  • back-to-front: emission buffer (simpler)

Attenuation Modeling: Projection

```
attenuation volume
    \[ T_s \]
```

```
emission volume
    \[ C_s \]
```

```
emission buffer C

C = C \cdot T_s + C_s
```
Attenuation Modeling: Projection

\[ C = C \cdot T_s + C_s \]

Attenuation Modeling: Backprojection

\[ C = C \cdot T_s \]

Scattering Effects

Recursive convolution using a Gaussian filter [Bai'00][Zeng'00]

For slice-based projection
- attenuation adjusted kernels
- distance adjusted kernels
- direction adjusted kernels

Projection: Back-to-front order
- adjusted Gaussian blurring
  - scatter energy
- multiply with (1 - attenuation volume)
  - attenuate energy
- add to the slice in the front
  - accumulate energy

Backprojection: Front-to-back
Scattering: Projection

\[ C = C \otimes G(T) + C_s \]

Scattering: Backprojection

\[ C = C \otimes T_s \]
Attenuation + Scattering: Projection

\[ \text{attenuation volume} \xrightarrow{T_s} G \otimes x \xrightarrow{+} C \]

\[ C = C \otimes G(T) \cdot T_s + C_s \]

Attenuation + Scattering: Backproj

\[ \text{attenuation volume} \xrightarrow{T_s} G \otimes x \xrightarrow{-} C \]

\[ C = C \otimes G(T) \cdot T_s + C_s \]
We may store transparency and emission volumes in 3D textures. However, 3D textures do not provide efficient write-back capabilities. 

- solution: perform forward projection with 3D textures and back-projection with 2D textures
- however, 3D/2D texture switching is expensive

Therefore, we use a 2D texture scheme for all operations.

- using an axis-aligned scheme
- but, can this be as accurate than a true 3D texture scheme?
Using 2D Textures

3D texture  2D texture  2D texture
- scale kernel width by 1/cosα

Δs=1
Δs=1/cosα

- scale kernel width by 1/cosα

- adjust kernel width for perspective distortion

3D texture  2D texture  2D texture
- scale kernel width by 1/cosα

Δs=1
Δs=1/cosα

- scale kernel width by 1/cosα

V₁  V₂

V₁  V₂

MIC-GPU
Results: Texture Schemes

Detector-aligned  Axis-aligned  Difference  Line profile

0°  20°  40°

RMS error within 1-2%

Results: Simulations

Scattering creates substantially more blur

Attenuation weakens the projections of emissions traversing highly attenuating material

- both with and without scattering.

Results: Performance

Adding attenuation and scattering to projector adds little overhead

- compensated back-projector 5 more expensive
- thus, unmatched projector-backprojector significantly more efficient
- will be used for reconstruction

Adding attenuation and scattering to projector adds little overhead

- compensated back-projector 5 more expensive
- thus, unmatched projector-backprojector significantly more efficient
- will be used for reconstruction
Results: Reconstructions

No Compensation  With Compensation  Line Profile

E+A  

E+S  

E+S+A  

10 iterations

Course Schedule

1:30 – 2:00: Introduction
2:00 – 2:30: GPU architecture, programming model, and programming facilities
2:30 – 3:00: GPU programming examples (image processing)
   
   Coffee Break

3:30 – 4:00: CT reconstruction pipeline components
4:00 – 4:30: GPU-acceleration of individual components

4:30 – 5:00: Various CT reconstruction pipelines, load balancing and load estimation

5:00 – 5:30: Reconstruction visualization and final remarks