Parallel Rendering In the GPU Era

Orion Sky Lawlor
UAF Computer Science Dept
lawlor@alaska.edu
2010-02-24

http://lawlor.cs.uaf.edu/
Parallel Rendering:
liveViz pixel transport
Basic model: LiveViz

- Serial 2D Client
- Parallel Charm++ Server
- Client connects
- Server sends client the current 2D image pixels (just pixels)
  - Can be from a 3D viewpoint (liveViz3D mode)
  - Can be color (RGB) or grayscale
  - Recently extended to support JPEG compressed network transport
    - Big win on slow networks!
**LiveViz on Fast Network**

<table>
<thead>
<tr>
<th>Window Size</th>
<th>No Compression</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>256x256</td>
<td>333 fps</td>
<td>25 fps</td>
</tr>
<tr>
<td>512x512</td>
<td>166 fps</td>
<td>24 fps</td>
</tr>
<tr>
<td>1024x1024</td>
<td>50 fps</td>
<td>15 fps</td>
</tr>
<tr>
<td>2048x2048</td>
<td>13 fps</td>
<td>4 fps</td>
</tr>
</tbody>
</table>

- On a gigabit network, JPEG compression is CPU-bound, and just slows us down!

- Compression hence **optional**
LiveViz on Slow Network

- On a slow 2MB/s wireless or lower 48 link, uncompressed liveViz is network bound

- Here, JPEG data transport is a big win!

<table>
<thead>
<tr>
<th>Window Size</th>
<th>No Compression</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>256x256</td>
<td>6 fps</td>
<td>22 fps</td>
</tr>
<tr>
<td>512x512</td>
<td>2 fps</td>
<td>15 fps</td>
</tr>
<tr>
<td>1024x1024</td>
<td>&lt; 1 fps</td>
<td>13 fps</td>
</tr>
<tr>
<td>2048x2048</td>
<td>&lt;&lt; 1 fps</td>
<td>4 fps</td>
</tr>
</tbody>
</table>
Parallel Rendering for Large Cosmology Datasets
Large Particle Dataset

- Large astrophysics simulation (Quinn et al)
  - $\geq 50M$ particles
  - $\geq 20$ bytes/particle
  - $\Rightarrow 1$ GB of data
Large Particle Rendering

- Rendering process (in principle)
  - For each pixel:
    - Find maximum mass along 3D ray
    - Look up mass in color table
Large Particle Rendering

Rendering process (in practice)

- For each particle:
  - Project 3D particle onto 2D screen
  - Keep maximum mass at each pixel
  - Ship image to client
  - Apply color table to 2D image at client
Large Particle Rendering (2D)
Rendering process (in practice)

- Project 3D particle onto 2D screen
- Keep maximum mass at each pixel
- Ship image to client
- Apply color table to 2D image at client

Large Particle Rendering (2D)
Particle Set to Volume Impostors
Shipping Volume Impostors

Slices of 3D Volume

Stack of 2D Slices
• Hey, that's just a 2D image!
• So we can use liveViz:
  Render slices in parallel
  Assemble slices across processors
  (Optionally) JPEG compress image
  Ship across network to (new) client
Volume Impostors Technique

- 2D impostors are flat, and can't rotate
- 3D voxel dataset can be rendered from any viewpoint on the client

Practical problem:

- Render voxels into a 2D image on the client by drawing slices with OpenGL
- Store maximum across all slices: `glBlendEquation(GL_MAX);`
- To look up (rendered) maximum in color table, render slices to texture and run a programmable shader
GLSL code to look up the rendered color in our color table texture:

```glsl
varying vec2 texcoords;
uniform sampler2D rendered, color_table;
void main() {
    vec4 rend = texture2D(rendered, texcoords);
    gl_FragColor = texture2D(color_table, vec2(rend.r + 0.5 / 255, 0));
}
```

But: color table index doesn't interpolate
  - Use GL_NEAREST, but it's pretty blocky!
Parallel Visualization:
MPIglut
MPIglut: Motivation

• All modern computing is parallel
  ▪ Multi-Core CPUs, Clusters
    • Athlon 64 X2, Intel Core2 Duo
  ▪ Multiple Multi-Unit GPUs
    • nVidia SLI, ATI CrossFire
  ▪ Multiple Displays, Disks, ...

• But languages and many existing applications are sequential
  ▪ Software problem: run existing serial code on a parallel machine
  ▪ Related: easily write parallel code
What is a “Powerwall”?

• A powerwall has:
  ▪ Several physical display devices
  ▪ One large virtual screen
  ▪ I.E. “parallel screens”

• UAF CS/Bioinformatics Powerwall
  ▪ Twenty LCD panels
  ▪ 9000 x 4500 pixels combined resolution
  ▪ 35+ Megapixels
#include <GL/glut.h>

void display(void) {
    glBegin(GL_TRIANGLES); ... glEnd();
    glutSwapBuffers();
}

void reshape(int x_size,int y_size) {
    glViewport(0,0,x_size,y_size);
    glLoadIdentity();
    gluLookAt(...);
}

...  

int main(int argc,char *argv[]) {
    glutInit(&argc,argv);
    glutCreateWindow("Ello!");
    glutMouseFunc(...);
    ...
    
    ...
}
MPIglut: Required Code Changes

```
#include <GL/mpi glut.h>

void display(void) {
  glBegin(GL_TRIANGLES); ... glEnd();
  glutSwapBuffers();
}

void reshape(int x_size, int y_size) {
  glViewport(0, 0, x_size, y_size);
  glLoadIdentity();
  gluLookAt(...);
}

int main(int argc, char *argv[]) {
  glutInit(&argc, argv);
  glutCreateWindow("Ello!");
  glutMouseFunc(...);
  ...
}
```

This is the **only** source change. Or, you can just copy mpiglut.h over your old glut.h header!
#include <GL/mpioglut.h>

void display(void) {
    glBegin(GL_TRIANGLES); ... glEnd();
    glutSwapBuffers();
}

void reshape(int x_size,int y_size) {
    glViewport(0,0,x_size,y_size);
    glLoadIdentity();
    gluLookAt(...);
}

...  

int main(int argc,char *argv[]) {
    glutInit(&argc,argv);
    glutCreateWindow("Ello!");
    glutMouseFunc(...);
    ...
}

MPIglut starts a separate copy of the program (a “backend”) to drive each powerwall screen
#include <GL/mpiglut.h>

void display(void) {
    glBegin(GL_TRIANGLES); ... glEnd();
    glutSwapBuffers();
}

void reshape(int x_size,int y_size) {
    glViewport(0,0,x_size,y_size);
    glLoadIdentity();
    gluLookAt(...);
}

... ...

int main(int argc,char *argv[]) {
    glutInit(&argc,argv);
    glutCreateWindow("Ello!");
    glutMouseFunc (...);
    ...
}

Mouse and other user input events are collected and sent across the network. Each backend gets identical user events (collective delivery)
MPIglut Runtime Changes: Sync

```c
#include <GL/mpiglut.h>

void display(void) {
    glBegin(GL_TRIANGLES); ... glEnd();
    glutSwapBuffers();
}

void reshape(int x_size, int y_size) {
    glViewport(0,0,x_size,y_size);
    glLoadIdentity();
    gluLookAt(...);
}

...  
int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    glutCreateWindow("Ello!");
    glutMouseFunc(...);
    ...
}
```

Frame display is (optionally) synchronized across the cluster
#include <GL/mpiGLUT.h>

void display(void) {
    glBegin(GL_TRIANGLES); ... glEnd();
    glutSwapBuffers();
}

void reshape(int x_size, int y_size) {
    glViewport(0, 0, x_size, y_size);
    glLoadIdentity();
    gluLookAt(...);
}

... User code works only in global coordinates, but MPIglut adjusts OpenGL's projection matrix to render only the local screen...
#include <GL/mpiglut.h>

void display(void) {
    glBegin(GL_TRIANGLES); ... glEnd();
    glutSwapBuffers();
}

void reshape(int x_size,int y_size) {
    glViewport(0,0,x_size,y_size);
    glLoadIdentity();
    gluLookAt(...);
}

...

int main(int argc,char *argv[]) {
    glutInit(&argc,argv);
    glutCreateWindow("Ello!");
    glutMouseFunc(...);
    ...
    return 0;
}
Benchmark Applications

UAF CS Bioinformatics Powerwall
Switched Gigabit Ethernet Interconnect
10 Dual-Core 2GB Linux Machines:
7 nVidia QuadroFX 3450
3 nVidia QuadroFX 1400

basic

mandel

soar

tex, tex_obj

vtx, vtx_obj
MPIglut Performance

Delivered Performance (fps)

Number of Screens and CPUs (MPIglut)

- basic
- tex_obj
- vtx_obj
- tex
- vtx
- soar
- mandel
Load Balancing a Powerwall

• Problem: Sky really easy, Terrain really hard

• Solution: Move the rendering for load balance, but you've got to move the finished pixels back for display!
GPGPU & cudaMPI
CUDA: Memory Output Bandwidth

Kernel startup latency: 4us

\[ t = \frac{4000\text{ns}}{\text{kernel}} + \text{bytes} \times 0.0125 \text{ ns / byte} \]

Kernel output bandwidth: 80 GB/s

NVIDIA GeForce GTX 280, fixed 128 threads per block
CUDA: Memory Output Bandwidth

\[ t = \frac{4000\text{ns}}{\text{kernel}} + \text{bytes} \times 0.0125\text{ ns/byte} \]

WHAT!??

\[ \text{time/kernel} = 320,000 \times \text{time/byte}!? \]
GPU Kernels: high latency, and high bandwidth

GPU-CPU copy: high latency, but low bandwidth

High latency means use big batches (not many small ones)

Low copy bandwidth means you MUST leave most of your data on the GPU (can't copy it all out)

Similar lessons as network tuning! (Message combining, local storage.)
Parallel GPU Application Interface

- On the CPU, for message passing the Message Passing Interface (MPI) is standard and good
  - Design by committee means if you need it, it's already in there!
    - E.g., noncontiguous derived datatypes, communicator splitting, buffer mgmt

- So let's steal that!
  - Explicit send and receive calls to communicate GPU data on network
Parallel GPU Send and Receives

- `cudaMPI_Send` is called on CPU with a pointer to GPU memory
  - `cudaMemcpy` into (pinned) CPU communication buffer
  - `MPI_Send` communication buffer
- `cudaMPI_Recv` is similar
  - `MPI_Recv` to (pinned) CPU buffer
  - `cudaMemcpy` off to GPU memory
- `cudaMPI_Bcast`, `Reduce`, `Allreduce`, etc all similar
  - Copy data to CPU, call MPI (easy!)
OpenGL Communication

■ It's easy to write corresponding glMPI_Send and glMPI_Recv
  ■ glMPI_Send: glReadPixels, MPI_Send
  ■ glMPI_Recv: MPI_Recv, glDrawPixels

■ Sadly, glDrawPixels is very slow: only 250MB/s!
  ■ glReadPixels is over 1GB/s
  ■ glTexSubImage2D from a pixel buffer object is much faster (>1GB/s)
<table>
<thead>
<tr>
<th># GPUs</th>
<th>Output Rate</th>
<th>Computation</th>
<th>Network</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>20.4 Gpix/s</td>
<td>2.01 ms</td>
<td>0.34 ms</td>
<td>85%</td>
</tr>
<tr>
<td>3</td>
<td>27.8 Gpix/s</td>
<td>1.36 ms</td>
<td>0.36 ms</td>
<td>78%</td>
</tr>
<tr>
<td>4</td>
<td>34.2 Gpix/s</td>
<td>1.06 ms</td>
<td>0.35 ms</td>
<td>72%</td>
</tr>
<tr>
<td>5</td>
<td>39.2 Gpix/s</td>
<td>0.85 ms</td>
<td>0.37 ms</td>
<td>66%</td>
</tr>
<tr>
<td>6</td>
<td>43.7 Gpix/s</td>
<td>0.72 ms</td>
<td>0.38 ms</td>
<td>61%</td>
</tr>
<tr>
<td>7</td>
<td>47.9 Gpix/s</td>
<td>0.63 ms</td>
<td>0.37 ms</td>
<td>57%</td>
</tr>
<tr>
<td>8</td>
<td>51.3 Gpix/s</td>
<td>0.55 ms</td>
<td>0.38 ms</td>
<td>54%</td>
</tr>
<tr>
<td>9</td>
<td>53.3 Gpix/s</td>
<td>0.51 ms</td>
<td>0.39 ms</td>
<td>50%</td>
</tr>
<tr>
<td>10</td>
<td>55.3 Gpix/s</td>
<td>0.47 ms</td>
<td>0.40 ms</td>
<td>46%</td>
</tr>
</tbody>
</table>

Ten GTX 280 GPUs over gigabit Ethernet; OpenGL