Does Your Software Scale?
Multi-GPU Scaling for Large Data Visualization
Thomas Ruge, NVIDIA
Agenda

- Multi-GPU Scaling for Large Data Visualization (Thomas Ruge, NVIDIA)
  - Multi GPU why?
  - Different Multi-GPU rendering methods, focus on database decomposition
  - System Analysis of depth compositing and its impact on multi-GPU rendering
  - Performance results of multi-GPU rendering with NVSG and OSG

- NVIDIA’s Multi-GPU SDK (Thomas Volk, NVIDIA)
  - How to use NVIDIA’s MGPU SDK to get scalability in your application

- NVSG-Scale (Subu Krishnamoorthy, NVIDIA)
  - Multi-GPU implemented in NVSG

- Technical Demos (with Horn Thorolf Tonjum, Stormfjord)

- Q&A
Multi-GPU: Why?

- Demand for Large Data visualization beyond capacity of 1 GPU
  - Boeing 777 (350 MTriangles) (between 4.2 - 29 GB of graphics data)
  - Visible Human 16 GB of 3D Textures

- Single GPU technology approaches cliff (like CPU’s do)
  - Power consumption is growing
  - Cooling problems with high heat output
  - High production costs per chip of dies with large footprints
    - G80: 681 M transistors, 90nm, 484 mm²
    - GT200: 1 bn transistors, 65 nm, 576 mm²

- Higher Performance than fastest single GPU on the market
  - MGPU can give you a time advantage, get tomorrow’s single GPU performance today with a multi-GPU system
Multi-GPU: Goal

- Goal: Get linear scalability with number of GPU’s in
  - Rendering performance
    - Triangle throughput
    - Fill rate
  - Data size
  - Image resolution (not topic of this presentation)

- Steps to get scalability:
  - Distribution of the workload/rendering tasks to all GPU’s
  - Collection of rendering results from all GPU’s
  - Assemble the rendering results in final image
Multi-GPU: Distributing the Load

- **Pixel Decomposition (e.g. SLI Antialiasing)**
  - Assigns different pixels or subpixels to each GPU
  - Helps with fill-rate bound apps, good approach for AA

- **Decomposition in time (e.g. SLI AFR)**
  - Different GPUs render different frames
  - Scales well for vertex-processing and fill rate bound applications
Multi-GPU: Screen Decompositing

- Sort-First or Screen Decomposition (e.g. SLI SFR or SLI Mosaic)
  - Different GPUs are rendering different portions of the screen
  - Fill rate bound applications scale well
  - Good method for Displays with very high resolutions (e.g. mersive Displays or Sony 4k)

Compositing Schemes:
- None, use (Tiled walls or multi-input Displays)
- Simple stitching (e.g. SLI SFR)
Multi-GPU: Database Decomposition

- **Sort-Last or Database Decomposition**
  - Different GPUs render different portions of the dataset
  - Scales well in Vertex processing, fill rate bound and graphics card memory bound applications

- **Compositing Schemes**
  - Depth or z-Buffer compositing (needs RGB and Depth Buffer from rendering GPU’s)
  - Alpha-compositing (needs only RGBA buffer)

=> Database decomposition is the method that gives us more graphics card memory
All Decomposition Schemes require Compositing to create the final image

This can be done:

- with special purpose hardware
  - very fast, typically very low additional latency and performance impact
  - Tight to specific graphics hardware (can’t mix/match different cards)
  - only subset of all possible composition schemes possible (e.g. SLI doesn’t support z- or alpha-compositing)
  - no general purpose compositing hardware available (but several attempts in the past)

- with commodity hardware and some software
  - Adds additional latency, performance impact depends on final image size and compositing mode
  - not very graphic card dependent (mix/match possible)
  - can cover all known compositing schemes
  - New HW developments (e.g. PCIe Gen 2) reduces the impact of SW-compositing
A System Analysis of Compositing with commodity hardware

System Analysis of database decomposition with depth compositing

- Goal:
  - Understand the impact of depth compositing on multi-GPU rendering
  - Means to estimate performance benefits
  - Valid on today’s HW

- Assumptions:
  - single shared memory system (no clusters)
  - only one displaying GPU (no powerwalls)
  - buffer transports go through host memory (no specific HW tricks)

- Tasks:
  - Distribute workload to rendering GPUs
  - Download resulting 2D-buffers (from GPUs framebuffer) to host memory
  - Upload 2D-buffers to displaying GPU
  - composite 2D-buffers to final image
Compositing with 2 GPUs

Start Analysis with 2 GPUs

System description:
- 2 rendering GPUs, 1 displaying GPU
- Optimization: displaying GPU is also a rendering GPU
  => reduced number of buffer transports

Depth compositing needs RGB and Z-buffer

E.g. 1920x1200 pixels:
- $s_x = 1920$, $s_y = 1200$, $bpp = 8$ (BGRA+DEPTH_STENCIL)
- 18,432,000 Bytes per frame buffer
- 73,728,000 Bytes transported through system per frame
- @60 Hz = 4,423,680,000 > 4 GB per second!

=> High load on system Bus
Depth Compositing Performance on 790i Ultra

System: 790i Ultra, 2 FX 3700 (PCIe 2.0)

Specs relevant for Compositing:

- Buffer downloads: 2.8 GB/s ⇔ 360 ms/GB
- host memory copy: 4.1 GB/s ⇔ 240 ms/GB
- Buffer uploads: 3.7 GB/s ⇔ 270 ms/GB
- Execution time of the actual compositing (fragment shader) is negligible
## Depth Compositing Performance

### Time to depth-composite 1 MPixel (worst - best case)

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Sequential Execution</th>
<th>Parallel Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_x \cdot s_y \cdot bpp = 1024 \cdot 1024 \cdot 8 = 8 , \text{MByte}$</td>
<td>870 ms/GB $\Leftrightarrow$ 143 fps</td>
<td>360 ms/GB $\Leftrightarrow$ 350 fps</td>
</tr>
<tr>
<td>$t_{total} = t_{dn} + t_{mc} + t_{up}$</td>
<td>550 ms/GB $\Leftrightarrow$ 227 fps</td>
<td></td>
</tr>
<tr>
<td>(some concurrency)</td>
<td>(full concurrency)</td>
<td></td>
</tr>
</tbody>
</table>

- **Compositing performance for sequential execution:**
  - Worst case (no concurrency): $t_{total} = t_{dn} + t_{mc} + t_{up}$
  - Compositing performance for parallel execution:
    - Best case (full concurrency): $t_{total} = \max(t_{dn}, t_{mc}, t_{up})$

### Formulas

- $t_{dn} = \text{Time to download BGRA buffer [s/GB]}$
- $t_{dc} = \text{Time to download DEPTH_STENCIL buffer [s/GB]}$
- $t_{dn} = (t_{dc} + t_{ac})/2$
- $t_{mc} = \text{Time to copy from host mem to host mem [s/GB]}$
- $t_{up} = \text{Time to upload [s/GB]}$

### Diagram

- GPU 1 to GPU 2: 1.8 GB/s
- Host Memory
- CPU

© 2008 NVIDIA Corporation.
Single GPU Performance: Triangle throughput and data size

Rendering performance measured in triangles/sec depending on data size on graphics card with limited memory

\[ P(n) = \frac{n_0 \cdot t_in + (n - n_0) \cdot t_out}{n} \]

*\( P(n) \) = Triangle performance [1/s]*
*\( n \) = number of triangles*
*\( n_0 \) = number of triangles in memory*
*\( t_in \) = rendering time 1 triangle in memory [s]*
*\( t_out \) = rendering time 1 triangle out of memory [s]*

Example, Rendering performance on Quadro FX 5600:

bytes per triangle = 3 vertices (36 bytes) + 3 normals (36 bytes) = 72 bytes / triangle

NVSG-test program renders individual triangles in VBO’s:

- \( p_{in} = 143 \text{ Mtriangles/s for in-memory data} \Rightarrow t_{in} = 7 \text{ ns} \)
- \( p_{out} = 25 \text{ Mtriangles / s for out-of-memory data} \Rightarrow t_{out} = 40 \text{ ns} \)
- \( n_0 = 20 \text{ Mtriangles} \)
Multi GPU Rendering: 2 GPUs

- Compositing impact is known
- Rendering performance on single GPU is known

Let’s estimate what performance gain we can expect over a growing dataset.

Assumptions:
- Execution of rendering and compositing happens sequentially (will be changed in the future);
- Rendering happens concurrently w/o overhead on both GPU’s;
- Fixed buffer size = 1k x 1k = 1Mpixel => \( t_{\text{compositing}} = 4.4 \text{ ms} \)

single GPU total time per frame:
- \( t_{\text{total,1 GPU}} = t_{\text{render}}(n) = n_0 \cdot t_{\text{in}} + (n-n_0) \cdot t_{\text{out}} \)

dual GPU total time per frame:
- \( t_{\text{total,2 GPU}} = t_{\text{render}}(n/2) + t_{\text{compositing}} \)

\[ \Rightarrow \text{performance gain} \quad s_{\text{2GPU}} = \frac{t_{\text{total,1 GPU}}}{t_{\text{total,2 GPU}}} \]

peak performance to be expected at \( n = 2 \cdot n_0 \):
\[ s(n=2\cdot n_0) = \frac{(n_0 \cdot t_{\text{in}} + n_0 \cdot t_{\text{out}})/(n_0 \cdot t_{\text{in}} + t_{\text{c}})}{(t_{\text{in}} + t_{\text{out}})/(t_{\text{in}} + t_{\text{c}}/n_0)} \]

improve peak performance by:
- reduce \( t_{\text{c}} \) (e.g., optimize buffer transports)
- increase \( n_0 \) (e.g., pack more triangles in card like tristrips instead individual triangles)

- Theoretical peak max \( s_{\text{max}} = (t_{\text{in}} + t_{\text{out}})/t_{\text{in}} \)
- applied to FX5600 \( s_{\text{max}} = (7\text{ns} + 40\text{ns})/7\text{ns} = 6.7 \)
Multi-GPU Rendering: 4 GPUs

Generalize 2-GPU formula to k GPUs:

Assumption:
buffer transports time grows linearly with number of GPUs

\[ k \text{ GPUs total time per frame:} \]
\[
t_{\text{total, } k \text{ GPU}} = t_{\text{render}}(n/k) + (k-1) \cdot t_{\text{compositing}}
\]

⇒ performance gain \( s_{k\text{GPU}} = t_{\text{total, }1 \text{ GPU}} / t_{\text{total, }k \text{ GPU}} \)

peak performance to be expected at \( n = k \cdot n_0 \):
\[
s(n=k\cdot n_0) = (n_0 \cdot t_{\text{in}} + (k-1) \cdot n_0 \cdot t_{\text{out}}) / (n_0 \cdot t_{\text{in}} + (k-1) \cdot t_{\text{c}})
\]

\[ s_{\text{max}} = (t_{\text{in}} + (k-1) \cdot t_{\text{out}}) / (t_{\text{in}} + (k-1) \cdot t_{\text{c}} / n_0) \]

- Theoretical peak max \( s_{\text{max}} = (t_{\text{in}} + (k-1) \cdot t_{\text{out}}) / t_{\text{in}} = 1 + (k-1) \cdot t_{\text{out}} / t_{\text{in}} \)

E.g. FX 5600
- \( k = 4 \text{ FX 5600} \) \( s_{\text{max}} = \frac{7\text{ns} + 3 \cdot 40\text{ns}}{7\text{ns}} = 18.1 \)
- \( k = 8 \text{ FX 5600} \) \( s_{\text{max}} = \frac{7\text{ns} + 7 \cdot 40\text{ns}}{7\text{ns}} = 41 \)

Rendering Performance

multi-GPU relative to 1 GPU

© 2008 NVIDIA Corporation.
Frame Rates

Application classes by frame rate

- \( \geq 60 \text{ Hz}, \) Visual Simulation (e.g. professional flight simulators)
- \( \geq 30 \text{ Hz}, < 60 \text{ Hz}, \) Entertainment (gaming)
- \( \geq 15 \text{ Hz}, < 30 \text{ Hz}, \) VR, Design Review
- \( \geq 5 \text{ Hz}, < 15 \text{ Hz}, \) Modeling, Large CAD Data Visualization, Seismic Interpretation,…

Best scalability for Large Models (low impact by compositor)
- \( > 60 \text{ Hz} \)
- \( 30 \text{ – } 60 \text{ Hz} \)
- \( 15 \text{ – } 30 \text{ Hz} \)
- \( 5 \text{ – } 15 \text{ Hz} \)

Best scalability for Large Models (low impact by compositor)
- \( > 60 \text{ Hz} \)
- \( 30 \text{ – } 60 \text{ Hz} \)
- \( 15 \text{ – } 30 \text{ Hz} \)
- \( 5 \text{ – } 15 \text{ Hz} \)
Results System Analysis

System Analysis shows:

- Performance scales with number of GPUs
  => Higher framerate

- Graphics card memory scales linearly with number of GPUs
  => Larger models

- Small models don’t scale very well
  => Multi-GPU rendering (database decomposition) not useful for small models

- Peak scalability far beyond linear (Larger Cache effect)

Multi-GPU rendering for Large models works!
Results Multi GPU rendering: OSG

**OSG = Open Scene Graph**

Open Source Scene Graph widely used in Academia, Simulation, Oil&Gas and other industries.

We extended OSG to render in multiple threads (one thread per GPU) and added the MGPU SDK.

In this test OSG used around 140 bytes per triangle.
Results Multi GPU rendering: NVSG

NVSG = NVIDIA’s Scene Graph
Very efficient Scene Graph used in automotive, simulation and academia

NVSG used 72 bytes per triangle
MGPU - SDK Introduction

Thomas Volk
Agenda

• Scope of the SDK
• SDK features
• Programming Guide for the SDK
Scope of the MGPU SDK

- Multi-GPU set-up and addressing
  - Affinity context/screen per GPU
  - SDK utility class for context handling and off-screen rendering
- Application parallelism
  - Multi-threading of rendering loop: already solved in lots of scene graph implementations
  - Multi-process and out of core/cluster solutions: highly specialized solutions already available
- Load decomposition and balancing: needs to be addressed in SG for sort last (see NVSGScale)
  - Sample/reference implementations
  - Utility classes
- Image compositing with huge data transfer (5GB/s) and low latency: SDK compositor
SDK features

• OpenGL based
• Professional applications
• QuadroPlex only
  • With current systems up to 2-8 GPUs, e.g. 2xD2s, 2xD4s
  • Up to 16 GB of addressable video memory
• Platforms: win32/64, linux 64
• In comparison to SLI non-transparent to application
• Utility functions to create MGPU aware GL contexts and drawables
  • Platform independent
  • Stereo
  • AA
• Image compositor for sort first and sort last based applications
  • Easy to use abstract interface for compositing
  • Compositor implementation based on latest technologies, no migration effort for applications (next gen hardware will provide faster transport)
  • Multi-threaded, shared memory
  • Configurable: 1-1, n-1, hierarchical, hybrid
  • Screen tiling, alpha and depth based compositing
Compositor Types and Configurations

- Screen tiling
- Alpha compositing
- Depth compositing
- Hierarchical compositing
System Overview
SDK Classes

ICFactory
+createAlphaCompositor()
+createDepthCompositor()

ICCompositor
+setLayout()
+getSrcNode()
+getDstNode()

ICNode
+initialize()
+terminate()
+composite()

ICSrcNode

ICDstNode

_RenderArea_
+initialize()
+terminate()
+resize()
+makeCurrent()
+releaseCurrent()
+showFBContents()
+hideFBContents()
Code Example

```c
//init in control/main thread
ICDepthCompositor* c = CompositorFactory::getInstance()->getDepthCompositor();
c->setLayout(IC_ALLNODES, IC_RECT(width, height));
ICNode* myCompositorNode[MAX_NODES];
myCompositorNode[0] = c->getSrcNode(idx);

//init in every thread and context
//create affinity context and drawable;
RenderArea ra;
ra->initialize();
ra->makeCurrent();

//initialize GL resources of compositor
myCompositorNode[i]->initialize();

//in every render-thread
void drawFrame(int myID)
{
    drawPartialScene(myID);

    //trigger image composition
    myCompositorNode[myID]->composite();
}
```
Sample Sequence Diagram

AppThread ➔ Compositor ➔ RenderThread ➔ SrcNode ➔ RenderThread ➔ DstNode

- Init
- Paint
- DrawFrame
- Composite
- Layout
- Resize
MGPU Threading

App thread

Render src thread

Render dst thread

App thread

Render src thread

Render dst thread
Wrap up

• Current status
  • Beta release next week
  • No stereo and AA, linux only for beta
  • Freely available
  • Release mid Oct

• Future features
  • Better hardware support
  • Access to frame-buffers
  • Utility classes for load balancing
  • Performance feedback
  • Extensible shader compositor
  • Additional color formats
Multi-GPU Rendering with NVIDIA Scene Graph

The World’s Fastest Scene Graph On Steroids

Subu Krishnamoorthy
Distributing Scene Objects

- Audience - specifies which GPUs are responsible for processing a scene graph object
- DistributionTraverser - assigns Audiences to balance load on each GPU
- GLDistributedTraverser - respects Audiences and prunes traversal when its GPU is excluded
Distribution Scheme

• Opaque objects
  Least loaded GPU included, others excluded

• Translucent and overlay objects
  Primary GPU included, others excluded

• Implies use of Depth Compositor
  Image composited before translucent and overlay objects are drawn
The Components

- GLAffinityArea creates a thread that drives an associated GLDistributedTraverser
- GLDistributedRenderArea manages the collection of N-1 GLAffinityAreas
Parallel Rendering
Ease of Use

• Derive client RenderArea from GLDistributedRenderArea (instead of nvui::RenderArea)

```cpp
class wxGLDistributedRenderArea : public wxGLCanvas,
                                    public nvui::GLDistributedRenderArea
{
    ...
};
```

• Override GLDistributedRenderArea::init()
  • Create Primary GL context (affinity context) and make it current
  • Invoke base implementation

```cpp
bool wxGLDistributedRenderArea::init(nvui::RenderArea *shareArea)
{
    ...
    m_glContext = new wxGLAffinityContext(this, shareArea);
    wxGLCanvas::SetCurrent(*m_glContext);
    ...
    GLDistributedRenderArea::init(shareArea);
    ...
}
Opaque Object Distribution
Depth Composited Image
Spatial Distribution

- Audience assigned based object bounding box
- Implies use of Alpha Compositor
Summary

- Powerful new feature of NVSG
- Visualize large models at interactive frame rates
The Industrial GPU Revolution
Visual Computing In The Oil Sector, Horn Thorolf Tonjum
Dinosaurs in the Museum

Model characteristics:
- Model designed in Maya, AutoDesk
- 217 Million Triangles

Rendering Hardware:
- HP xw8600 with 32 GB System Memory
- 2xD2’s (4xGT200 with 4 GB FB memory) with 16 GB total FB memory

Rendering Performance:
1 GPU = 0.7 fps
2 GPU’s = 3 ½ fps = 5 times faster than 1 GPU
4 GPU’s = 6 fps = 8 ½ times faster than 1 GPU
=> Total max. Triangle throughput = 1.3 GTri/s
Multi-GPU rendering of Kristin, a detailed off-shore platform

Model characteristics:
• Kristin is an off-shore platform in the North Sea
• Model designed in MicroStation, Bentley
• 230 Million Triangles (individual triangles, no triangle strips)

Rendering Hardware:
• HP xw8600 with 32 GB System Memory
• 2xD2’s (4xGT200 with 4 GB FB memory) with 16 GB total FB memory

Rendering Performance:
1 GPU = 0.4 fps
2 GPU’s = 2 ½ fps = 6 times faster than 1 GPU
4 GPU’s = 4 ½ fps = 11 times faster than 1 GPU
=> Total max. Triangle throughput = 1 GTri/s

With the friendly permission of stormfjord
Summary

• System analysis of multi-GPU rendering

• How to do multi-GPU rendering with NVIDIA’s Multi-GPU SDK

• Ease of use of multi-GPU rendering with NVSG

• Large CAD data visualization in the Oil&Gas industry

=> Database decomposition together with depth compositing is a strong tool to give you tomorrows graphics performance today
The End

Questions ?