cudaMPI and glMPI: Message Passing for GPGPU Clusters

Orion Sky Lawlor
lawlor@alaska.edu
U. Alaska Fairbanks
2009-08-31
http://lawlor.cs.uaf.edu/
Where is Fairbanks, Alaska?
Talk Outline

- Modern GPU Hardware
- Quantitative GPU Performance
- GPU-Network Interfacing
- cudaMPI and glMPI Performance
- Conclusions & Future Work
Modern GPU Hardware
Latency, or Bandwidth?

- time = latency + size / bandwidth

- To minimize latency:
  - Caching (memory)
  - Speculation (branches, memory)

- To maximize bandwidth:
  - Clockrate (since 1970)
  - Pipelining (since 1985)
  - Parallelism (since 2004)

- For software, everything's transparent except parallelism!
CPU vs GPU Design Targets

- **Typical CPU optimizes for instruction execution latency**
  - Millions of instructions / thread
  - Less than 100 software threads
  - Hardware caches, speculation

- **Typical GPU optimizes for instruction execution bandwidth**
  - Thousands of instructions / thread
  - Over 1 million software threads
  - Hardware pipelining, queuing, request merging
Modern GPUs: Good Things

- Fully programmable
  - Branches, loops, functions, etc
  - IEEE 32-bit Float Storage

- Huge parallelism in hardware
  - Typical card has 100 cores
  - Each core threaded 10-100 ways
  - Hardware thread switching (SMT)

- E.g., $250 GTX 280 has 240 cores, >900 gigaflops peak

- GPU is best in flops, flops/watt, flops/$ [Datta ... Yelik 2008]
Modern GPUs: Bad Things

- Programming is really difficult
  - Branch & load divergence penalties
  - Non-IEEE NaNs, 64-bit much slower

- Huge parallelism in software
  - Need millions of separate threads!
  - Usual problems: decomposition, synchronization, load balance

- E.g., $250 GTX 280 has *only* 1GB of storage, 4GB/s host RAM

- GPU is definitely a mixed blessing for real applications
Modern GPU Programming Model

- “Hybrid” GPU-CPU model
  - GPU cranks out the flops
  - CPU does everything else
    - Run non-GPU legacy or sequential code
    - Disk, network I/O
    - Interface with OS and user
- Supported by CUDA, OpenCL, OpenGL, DirectX, ...
- Likely to be the dominant model for the foreseeable future
  - CPU may eventually “wither away”
GPU Programming Interfaces

- **OpenCL (2008)**
  - Hardware independent (in theory)
  - SDK still difficult to find

- **NVIDIA's CUDA (2007)**
  - Mixed C++/GPU source files
  - Mature driver, widespread use

- **OpenGL GLSL, DirectX HLSL (2004)**
  - Supported on NVIDIA and ATI on wide range of hardware
  - Graphics-centric interface (pixels)
NVIDIA CUDA

- Fully programmable (loops, etc)
- Read and write memory anywhere
  - Memory accesses must be “coalesced” for high performance
  - Zero protection against multithreaded race conditions
  - “texture cache” can only be accessed via restrictive CUDA arrays
- Manual control over a small shared memory region near the ALUs
- CPU calls a GPU “kernel” with a “block” of threads
- Only runs on new NVIDIA hardware
 OpenGL: GL Shading Language

- Fully programmable (loops, etc)
- Can only read “textures”, only write to “framebuffer” (2D arrays)
  - Reads go through “texture cache”, so performance is good (iff locality)
  - Attach non-read texture to framebuffer
  - So cannot have synchronization bugs!
- Today, greyscale GL_LUMINANCE texture pixels store floats, not GL_RGBA 4-vectors (zero penalty!)
- Rich selection of texture filtering (array interpolation) modes
- GLSL runs on nearly every GPU
Modern GPU Fast Stuff

- Divide, sqrt, rsqrt, exp, pow, sin, cos, tan, dot all sub-nanosecond
  - Fast like arithmetic; not library call

- Branches are fast iff:
  - They're short (a few instructions)
  - Or they're coherent across threads

- GPU memory accesses fast iff:
  - They're satisfied by “texture cache”
  - Or they're coalesced across threads (contiguous and aligned)

GPU “coherent” ≈ MPI “collective”
Modern GPU Slow Stuff

- Not enough work per kernel
  - >4us latency for kernel startup
  - Need >>1m flops / kernel!

- Divergence: costs up to 97%!
  - Noncoalesced memory accesses
    - Adjacent threads fetching distant regions of memory

- Divergent branches
  - Adjacent threads running different long batches of code

- Copies to/from host
  - >10us latency, <4GB/s bandwidth
Quantitative GPU Performance
CUDA: Memory Output Bandwidth

- Build a trivial CUDA test harness:

  ```c
  __global__ void write_arr(float *arr) {
    arr[threadIdx.x+blockDim.x*blockIdx.x]=0.0;
  }
  int main() { ... start timer
    write_arr<<<nblocks,threadsPerBlock>>>(arr);
    ... stop timer
  }
  ```

- Measure speed for various array sizes and number of threads per block
### CUDA: Memory Output Bandwidth

<table>
<thead>
<tr>
<th>Threads per Block</th>
<th>Number of Floats/Kernel</th>
<th>1K</th>
<th>10K</th>
<th>100K</th>
<th>1M</th>
<th>10M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
</tbody>
</table>

Output Bandwidth (Gigafloats/second)

NVIDIA GeForce GTX 280
## CUDA: Memory Output Bandwidth

<table>
<thead>
<tr>
<th>Threads per Block</th>
<th>Number of Floats/Kernel</th>
<th>1K</th>
<th>10K</th>
<th>100K</th>
<th>1M</th>
<th>10M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.2</td>
<td>err</td>
<td>err</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>0.3</td>
<td>0.4</td>
<td>err</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>0.4</td>
<td>0.8</td>
<td>err</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>8</td>
<td>0.1</td>
<td>0.5</td>
<td>1.4</td>
<td>err</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>16</td>
<td>0.1</td>
<td>0.6</td>
<td>2.4</td>
<td>3.3</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>32</td>
<td>0.1</td>
<td>0.7</td>
<td>3.6</td>
<td>6.4</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>64</td>
<td>0.1</td>
<td>0.7</td>
<td>4.8</td>
<td>11.7</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>128</td>
<td>0.1</td>
<td>0.7</td>
<td>5.4</td>
<td>16.4</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>256</td>
<td>0.1</td>
<td>0.7</td>
<td>5.3</td>
<td>16.4</td>
<td>20.7</td>
<td>20.7</td>
</tr>
<tr>
<td>512</td>
<td>0.1</td>
<td>0.7</td>
<td>5.2</td>
<td>16.3</td>
<td>20.7</td>
<td>20.7</td>
</tr>
</tbody>
</table>

Over 64K blocks needed

Output Bandwidth (Gigafloats/second)

NVIDIA GeForce GTX 280
## CUDA: Memory Output Bandwidth

<table>
<thead>
<tr>
<th>Number of Floats/Kernel</th>
<th>1K</th>
<th>10K</th>
<th>100K</th>
<th>1M</th>
<th>10M</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Threads per Block</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.1</td>
<td>0.6</td>
<td>2.4</td>
<td>3.3</td>
<td>err</td>
</tr>
<tr>
<td>32</td>
<td>0.1</td>
<td>0.7</td>
<td>3.6</td>
<td>6.4</td>
<td>err</td>
</tr>
<tr>
<td>64</td>
<td>0.1</td>
<td>0.7</td>
<td>4.8</td>
<td>11.7</td>
<td>err</td>
</tr>
<tr>
<td>128</td>
<td>0.1</td>
<td>0.7</td>
<td>5.4</td>
<td>16.4</td>
<td>err</td>
</tr>
<tr>
<td>256</td>
<td>0.1</td>
<td>0.7</td>
<td>5.3</td>
<td>16.4</td>
<td>20.7</td>
</tr>
<tr>
<td>512</td>
<td>0.1</td>
<td>0.7</td>
<td>5.2</td>
<td>16.3</td>
<td>20.7</td>
</tr>
<tr>
<td><strong>Kernel Startup Time</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.2</td>
<td>err</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>0.3</td>
<td>0.4</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>0.4</td>
<td>0.8</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>8</td>
<td>0.1</td>
<td>0.5</td>
<td>1.4</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>16</td>
<td>0.1</td>
<td>0.6</td>
<td>2.4</td>
<td>3.3</td>
<td>err</td>
</tr>
<tr>
<td>32</td>
<td>0.1</td>
<td>0.7</td>
<td>3.6</td>
<td>6.4</td>
<td>err</td>
</tr>
<tr>
<td>64</td>
<td>0.1</td>
<td>0.7</td>
<td>4.8</td>
<td>11.7</td>
<td>err</td>
</tr>
<tr>
<td>128</td>
<td>0.1</td>
<td>0.7</td>
<td>5.4</td>
<td>16.4</td>
<td>err</td>
</tr>
<tr>
<td>256</td>
<td>0.1</td>
<td>0.7</td>
<td>5.3</td>
<td>16.4</td>
<td>20.7</td>
</tr>
<tr>
<td>512</td>
<td>0.1</td>
<td>0.7</td>
<td>5.2</td>
<td>16.3</td>
<td>20.7</td>
</tr>
</tbody>
</table>

**Output Bandwidth (Gigafloats/second)**

**NVIDIA GeForce GTX 280**
## CUDA: Memory Output Bandwidth

<table>
<thead>
<tr>
<th>Block Startup Time Dominates</th>
<th>Number of Floats/Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per Block</td>
<td>1K</td>
</tr>
<tr>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
</tr>
<tr>
<td>8</td>
<td>0.1</td>
</tr>
<tr>
<td>16</td>
<td>0.1</td>
</tr>
<tr>
<td>32</td>
<td>0.1</td>
</tr>
<tr>
<td>64</td>
<td>0.1</td>
</tr>
<tr>
<td>128</td>
<td>0.1</td>
</tr>
<tr>
<td>256</td>
<td>0.1</td>
</tr>
<tr>
<td>512</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Output Bandwidth (Gigafloats/second)
## CUDA: Memory Output Bandwidth

<table>
<thead>
<tr>
<th>Threads per Block</th>
<th>Number of Floats/Kernel</th>
<th>1K</th>
<th>10K</th>
<th>100K</th>
<th>1M</th>
<th>10M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.2</td>
<td>err</td>
<td>err</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>0.3</td>
<td>0.4</td>
<td>err</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>0.4</td>
<td>0.8</td>
<td>err</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>8</td>
<td>0.1</td>
<td>0.5</td>
<td>1.4</td>
<td>err</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>16</td>
<td>0.1</td>
<td>0.6</td>
<td>2.4</td>
<td>3.3</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>32</td>
<td>0.1</td>
<td>0.7</td>
<td>3.6</td>
<td>6.4</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>64</td>
<td>0.1</td>
<td>0.7</td>
<td>4.8</td>
<td>11.7</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>128</td>
<td>0.1</td>
<td>0.7</td>
<td>5.4</td>
<td>16.4</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>256</td>
<td>0.1</td>
<td>0.7</td>
<td>5.3</td>
<td>16.4</td>
<td>20.7</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>0.1</td>
<td>0.7</td>
<td>5.2</td>
<td>16.3</td>
<td>20.7</td>
<td></td>
</tr>
</tbody>
</table>

Output Bandwidth (Gigafloats/second)

*Good Performance*
CUDA: Memory Output Bandwidth

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

Kernel startup latency: 4us

Kernel output bandwidth: 80 GB/s

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

Kernel startup latency: 4us

\[ t = \frac{4 \text{us}}{\text{kernel}} + \frac{\text{bytes}}{80 \text{ GB/s}} \]

Kernel output bandwidth: 80 GB/s

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

Kernel output bandwidth: 80 GB/s

Kernel startup latency: 4us

\[ t = \frac{4\text{us}}{\text{kernel}} + \text{bytes} \times 0.0125\text{ ns/byte} \]
CUDA: Memory Output Bandwidth

Kernel output bandwidth: 80 GB/s

Kernel startup latency: 4us

$\text{Kernel output bandwidth: } 80 \text{ GB/s}$

$\text{Kernel startup latency: } 4\text{us}$

$t = 4000\text{ns} / \text{kernel} + \text{bytes} \times 0.0125 \text{ ns} / \text{byte}$
CUDA: Memory Output Bandwidth

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

WHAT!?!

\[ \text{time/kernel} = 320,000 \times \text{time/byte}!? \]
CUDA: Device-to-Host Memcopy

Copy startup latency: 10μs

Copy output bandwidth: 1.7 GB/s

t = 10000ns / kernel + bytes * 0.59 ns / byte
CUDA: Device-to-Host Memcopy

\[ t = 10000\text{ns} / \text{kernel} + \text{bytes} \times 0.59 \text{ ns} / \text{byte} \]
CUDA: Memory Output Bandwidth

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

NVIDIA GeForce GTX 280, fixed 128 threads per block
Performance Analysis Take-Home

- 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.)
Why GPGPU clusters?
Parallel GPU Advantages

- **Multiple GPUs: more compute & storage bandwidth**

<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>6.1 Gpix/s</td>
<td>7.81 ms</td>
<td>- ms</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>11.3 Gpix/s</td>
<td>3.91 ms</td>
<td>0.35 ms</td>
<td>92%</td>
</tr>
<tr>
<td>3</td>
<td>16.2 Gpix/s</td>
<td>2.62 ms</td>
<td>0.35 ms</td>
<td>88%</td>
</tr>
<tr>
<td>4</td>
<td>20.7 Gpix/s</td>
<td>1.97 ms</td>
<td>0.36 ms</td>
<td>84%</td>
</tr>
<tr>
<td>5</td>
<td>25.0 Gpix/s</td>
<td>1.58 ms</td>
<td>0.35 ms</td>
<td>81%</td>
</tr>
<tr>
<td>6</td>
<td>28.5 Gpix/s</td>
<td>1.32 ms</td>
<td>0.37 ms</td>
<td>77%</td>
</tr>
<tr>
<td>7</td>
<td>32.4 Gpix/s</td>
<td>1.13 ms</td>
<td>0.36 ms</td>
<td>75%</td>
</tr>
<tr>
<td>8</td>
<td>35.3 Gpix/s</td>
<td>1.00 ms</td>
<td>0.37 ms</td>
<td>71%</td>
</tr>
<tr>
<td>9</td>
<td>38.2 Gpix/s</td>
<td>0.91 ms</td>
<td>0.39 ms</td>
<td>67%</td>
</tr>
<tr>
<td>10</td>
<td>40.7 Gpix/s</td>
<td>0.80 ms</td>
<td>0.39 ms</td>
<td>66%</td>
</tr>
</tbody>
</table>
Parallel GPU Communication

- Shared memory, or messages?
  - History: on the CPU, shared memory won for small machines; messages won for big machines
  - Shared memory GPUs: Tesla
  - Message-passing GPUs: clusters
    - Easy to make in small department
      - E.g., $2,500 for ten GTX 280's (8 TF!)
  - Or both, like NCSA Lincoln

- Messages are lowest common denominator: run easily on shared memory (not vice versa!)
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 Implementation

- Should we send and receive from GPU code, or CPU code?
  - GPU code seems more natural for GPGPU (data is on GPU already)
  - CPU code makes it easy to call MPI
- Network and GPU-CPU data copy have high latency, so message combining is crucial!
  - Hard to combine efficiently on GPU (atomic list addition?) [Owens]
  - But easy to combine data on CPU
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!)
Asynchronous Send and Receives

- cudaMemcpy, MPI_Send, and MPI_Recv all block the CPU
- cudaMemcpyAsync, MPI_Isend, and MPI_Irecv don't
  - They return a handle you occasionally poll instead
- Sadly, cudaMemcpyAsync has much higher latency than blocking version (40us vs 10us)
- cudaMPI_Isend and Irecv exist
  - But usually are not worth using!
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)
CUDA or OpenGL?

- cudaMemcpy requires contiguous memory buffer
- glReadPixels takes an arbitrary rectangle of texture data
  - Rows, columns, squares all equally fast!
- Still, there's no way to simulate MPI's rich noncontiguous derived data types
Noncontiguous Communication

Network Data Buffer

- Scatter kernel?
- Fold into read?

GPU Target Buffer
cudaMPI and glMPI
Performance
Parallel GPU Performance

- Networks are really low bandwidth!

- **CUDA Performance**
  - **Graphics Card Memory:** 80 GB/s
  - **GPU-CPU Copy:** 1.7 GB/s

- **CPU-Peer Network Bandwidth:** 0.1 GB/s (Gigabit Ethernet)
### Parallel GPU Performance: CUDA

<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>6.1 Gpix/s</td>
<td>7.81 ms</td>
<td>- ms</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>11.3 Gpix/s</td>
<td>3.91 ms</td>
<td>0.35 ms</td>
<td>92%</td>
</tr>
<tr>
<td>3</td>
<td>16.2 Gpix/s</td>
<td>2.62 ms</td>
<td>0.35 ms</td>
<td>88%</td>
</tr>
<tr>
<td>4</td>
<td>20.7 Gpix/s</td>
<td>1.97 ms</td>
<td>0.36 ms</td>
<td>84%</td>
</tr>
<tr>
<td>5</td>
<td>25.0 Gpix/s</td>
<td>1.58 ms</td>
<td>0.35 ms</td>
<td>81%</td>
</tr>
<tr>
<td>6</td>
<td>28.5 Gpix/s</td>
<td>1.32 ms</td>
<td>0.37 ms</td>
<td>77%</td>
</tr>
<tr>
<td>7</td>
<td>32.4 Gpix/s</td>
<td>1.13 ms</td>
<td>0.36 ms</td>
<td>75%</td>
</tr>
<tr>
<td>8</td>
<td>35.3 Gpix/s</td>
<td>1.00 ms</td>
<td>0.37 ms</td>
<td>71%</td>
</tr>
<tr>
<td>9</td>
<td>38.2 Gpix/s</td>
<td>0.91 ms</td>
<td>0.39 ms</td>
<td>67%</td>
</tr>
<tr>
<td>10</td>
<td>40.7 Gpix/s</td>
<td>0.80 ms</td>
<td>0.39 ms</td>
<td>66%</td>
</tr>
</tbody>
</table>

Ten GTX 280 GPUs over gigabit Ethernet; CUDA
Parallel GPU Performance: OpenGL

<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
Conclusions
Conclusions

- Consider OpenGL or DirectX instead of CUDA or OpenCL
- Watch out for latency on GPU!
  - Combine data for kernels & copies
- Use cudaMPI and glMPI for free:
  - http://www.cs.uaf.edu/sw/
- Future work:
  - More real applications!
  - GPU-JPEG network compression
  - Load balancing by migrating pixels
  - DirectXMPI? cudaSMP?
Related work: 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
MPIglut: The basic idea

- Users compile their OpenGL/glut application using MPIglut, and it “just works” on the powerwall.
- MPIglut's version of glutInit runs a separate copy of the application for each powerwall screen.
- MPIglut intercepts glutInit, glViewport, and broadcasts user events over the network.
- MPIglut's glViewport shifts to render only the local screen.
#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

```c
#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!
MPIglut Runtime Changes: Init

```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(...);
    ...  
}
```

**MPIglut** starts a separate copy of the program (a "backend") to drive each powerwall screen.
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(...);
}

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

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

```c
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(...);
}
```

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

MPIglut does **NOT** intercept or interfere with rendering calls, so programmable shaders, vertex buffer objects, framebuffer objects, etc all run at full performance.
Each backend app must be able to render its part of its screen

- Does not automatically imply a replicated database, if application uses matrix-based view culling

Backend GUI events (redraws, window changes) are collective

- All backends must stay in synch
- Automatic for applications that are deterministic function of events
  - Non-synchronized: files, network, time
MPIglut: Bottom Line

• Tiny source code change
• Parallelism hidden inside MPIglut
  □ Application still “feels” sequential
• Fairly major runtime changes
  □ Serial code now runs in parallel (!)
  □ Multiple synchronized backends running in parallel
  □ User input events go across network
  □ OpenGL rendering coordinate system adjusted per-backend
□ But rendering calls are left alone
• 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!
Future Work: Load Balancing

• **AMPILut:** principle of persistence should still apply
• But need cheap way to ship back finished pixels every frame
• Exploring GPU JPEG compression
  - DCT + quantize: really easy
  - Huffman/entropy: really hard
  - Probably need a CPU/GPU split
    - 10000+ MB/s inside GPU
    - 1000+ MB/s on CPU
    - 100+ MB/s on network