Message Passing for GPGPU Clusters: cudaMPI

Orion Sky Lawlor [#1]

[#] *Department of Computer Science, University of Alaska Fairbanks*
202 Chapman Building, Fairbanks, Alaska 99712, USA
[1] lawlor@alaska.edu

***Abstract*—We present and analyze two new communication libraries, cudaMPI and glMPI, that provide an MPI-like message passing interface to communicate data stored on the graphics cards of a distributed-memory parallel computer. These libraries can help applications that perform general purpose computations on these networked GPU clusters. We explore how to efficiently support both point-to-point and collective communication for either contiguous or noncontiguous data on modern graphics cards. Our software design is informed by a detailed analysis of the actual performance of modern graphics hardware, for which we develop and test a simple but useful performance model.**

I. Introduction and Prior Work

In 1968 Myer and Sutherland [1] asked a surprisingly prescient question: "How much computing power should be included in the display processor?". Today, just as the decades-long exponential climb of serial processor performance ends, the parallelism available in modern GPUs can provide us with an enormous amount of computing power.

The modern Graphics Processing Unit (GPU) is a fully programmable parallel computer, with hardware capable of efficiently switching between thousands of threads. As such, interest is growing in the use of GPUs for non-graphics or "general purpose" problems (GPGPU). Excellent GPGPU results are becoming commonplace in fields from molecular dynamics to physics simulation [2]. For many real problems, GPUs have shown performance exceeding both the latest multicore processors and the Cell, not only in delivering more operations per second, but also more operations per second per dollar, and more operations per watt of power [3].

This performance potential, currently about one teraflop of single-precision floating point performance per GPU, is leading to increasing interest in using GPUs in networked supercomputers and clusters. This paper describes our new software libraries called cudaMPI[1] and glMPI for passing messages among a distributed-memory cluster of GPUs.

A. CPU Parallel Programming Interfaces

The Message Passing Interface, MPI, is an international standard programming interface for distributed-memory parallel CPU programming that actually achieved extremely widespread use. MPI is used by parallel applications on hardware ranging from multicore laptops through petaflop-and-megawatt supercomputers. Correspondingly, many implementations of MPI exist, from vendor-tuned high-performance implementations, to specialized research implementations, to open-source component-based implementations such as Open-MPI [4]. Back in 2003, we helped create an automated load-balancing implementation of MPI called Adaptive MPI or AMPI [5].

Our library cudaMPI uses CUDA to provide an MPI-like interface for GPU-to-GPU communication, as in Figure 1. Similarly, glMPI brings MPI-like networking to OpenGL.

B. Single GPU Programming Interfaces

There are two main classes of programming interface for the GPU: graphics interfaces, and general-purpose interfaces. Graphics interfaces such as OpenGL or Microsoft's DirectX are substantially older and more mature, and are now both portable and sophisticated interfaces supporting C++-like GPU programming, although designed around graphics concepts such as textures and pixels. The big limitation of graphics interfaces is that the only memory a pixel shader can change is its own pixel.

Newer general-purpose interfaces include NVIDIA's Compute Unified Device Interface (CUDA) and the new multi-vendor standard OpenCL. Several commercial languages also exist, including RapidMind and PeakStream. These interfaces provide a more natural programming environment, in particular allowing integer variables, pointer manipulation, and arbitrary memory reads and writes on the GPU cores. In this paper, we focus primarily on CUDA, which is emerging as the dominant interface for scientific GPGPU programming.

However, it is extremely important that we design software interfaces that can survive hardware changes, since high-performance hardware ceases to be high-performance in only a few years, yet parallel software often lives on for decades. OpenGL, despite its flaws, is approximately as old and well established as MPI, with dozens of stable implementations from major vendors, and hundreds of thousands of programs and

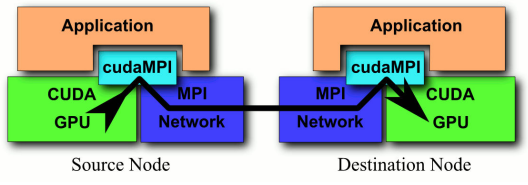

Fig. 1. Applications use cudaMPI to send GPU data across an MPI network.

[1]cudaMPI is not from NVIDIA, owner of the CUDA trademark.

programmers. For accessing programmable graphics hardware, OpenGL is currently the only choice that works with hardware from different vendors on all major operating systems, so in this paper we discuss OpenGL as well as CUDA.

C. Parallel GPU Programming

Programming parallel GPU machines is difficult, at least in part because both parallel and GPU programming are each difficult. However, a small class of GPU applications are easy to parallelize, because minimal or no communication is required during the application run. For example, we developed MPIglut [6], a parallelizing implementation of the sequential OpenGL windowing library glut which uses MPI internally to allow a serial GPU application to run across every node of a multi-screen multi-GPU powerwall. MPIglut manipulates the OpenGL viewport so each node's copy of the sequential application correctly renders its portion of the shared display, and MPIglut broadcasts user events across the parallel machine. Of course, more complicated applications require their own additional GPU-to-GPU communication.

On parallel CPUs, shared-memory parallel programs are widely considered easier to develop than message passing.[2] Many researchers have sought to extend this shared-memory paradigm to GPUs. First, for a graphics interface, driver rendering tricks such as SLI or CrossFire can hide the existence of multiple GPUs, but this only works efficiently up to a few GPUs. A conceptual shared-memory model like Global Arrays, recently implemented for GPUs in ZippyGPU [7], provides a clean interface, but achieving high performance can be challenging. Finally, software distributed shared memory can be implemented by modifying each memory access to use a GPU page table, but a 2006 implementation by Moerschell and Owens [8] showed a 100-fold slowdown. However, despite extensive research effort on CPUs, currently distributed memory with explicit message passing is the only communication technology known to scale well beyond a few thousand nodes, so we expect message-passing will become the dominant GPU communication method, at least at large scale.

Eventually, a GPU-only communication model may become affordable, where 100% of the application's code runs on the GPU, and the CPU withers to a mere I/O processor. Stuart and Owens implemented DCGN [9] this way, but even dedicating CPUs to polling for GPU communication, this still requires hundreds of microseconds per message. Fundamentally, the networks' per-message cost dominates the time to send many tiny messages, such as the per-GPU-kernel-execution messages in DCGN. It is much faster to send the same bytes in fewer messages; yet combining messages from different GPU kernels is tricky without involving the CPU.

For this reason, cudaMPI's programming model is CPU-centric, with CPU code initiating each communication operation, and the GPU only providing the communicated data. This allows communication operations to be performed en masse on large blocks of data, to make more efficient use of both the CPU and network.

[2]Although synchronization bugs under shared memory can be quite subtle!

Operation	$\alpha =$ **setup time**	$\beta =$ **1/bandwidth**
GPU Kernel Execution t_K	4000 ns/kernel	0.01 ns/byte
GPU-CPU Memcpy t_M	10000 ns/copy	0.4 ns/byte
Network (Infiniband) t_N	1000 ns/message	1 ns/byte
Network (Gigabit) t_N	50000 ns/message	10 ns/byte

Fig. 2. Typical constants for $t = \alpha + \beta n$ performance model.

Initiating communication on the CPU is acceptable because real parallel applications are complex, having hundreds of thousands of lines of existing CPU code that are not likely to soon be ported to the GPU, for both technical and practical reasons. For the foreseeable future, applications will use a CPU-GPU hybrid approach, handling setup or rare cases with their large legacy CPU codebase, and using the GPU for the smaller performance-critical portion of their operations. Thus real application development will require clean, portable, high-productivity interfaces to exchange data and control between the CPU, GPU, and network.

II. GPU Hardware and Performance Model

This hardware is involved in parallel GPU computing:

- GPU shader cores, which run GPU kernels, are both parallel and deeply multithreaded to provide significant computational power, currently on the order of a teraflop per GPU.
- Graphics memory, which is directly accessible by GPU kernels, has a high clockrate and wide bus width to provide substantial bandwidth, currently about a hundred gigabytes per second.
- GPU interconnect, providing mainboard access to the GPU. This is typically PCI Express, and so delivers a few gigabytes per second of bandwidth.
- Mainboard RAM, which is directly accessible by CPU programs and the network.
- CPU cores, which are deeply pipelined and superscalar to provide good performance on sequential programs.
- Network hardware, which moves bytes between nodes.

We use the trivial latency plus bandwidth performance model to describe the time taken to operate on n bytes:

$t = \alpha + \beta n$

t: total time, in seconds, for the operation.

α: software overhead and hardware setup time, typically tens of thousands of nanoseconds. For networks, this is one aspect of latency.

β: time per byte, typically less than a nanosecond. For networks, this is the inverse of bandwidth.

n: number of bytes being computed or moved.

Since this model applies to many different parts of our parallel GPU hardware, we add subscripts to indicate what operation is being used. For example, the time to run a GPU kernel to compute n bytes is quite nearly $t_K = \alpha_K + \beta_K n$. In section III we use this performance model to describe the

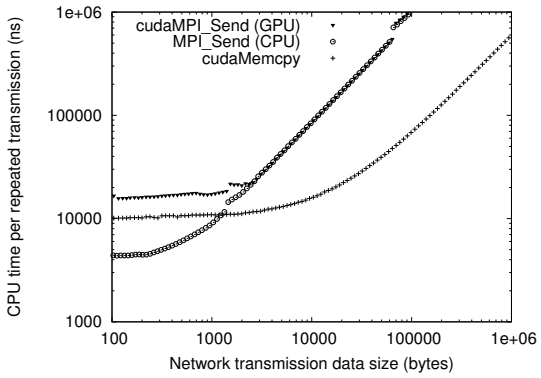

Fig. 4. CPU overhead for network operations on gigabit Ethernet.

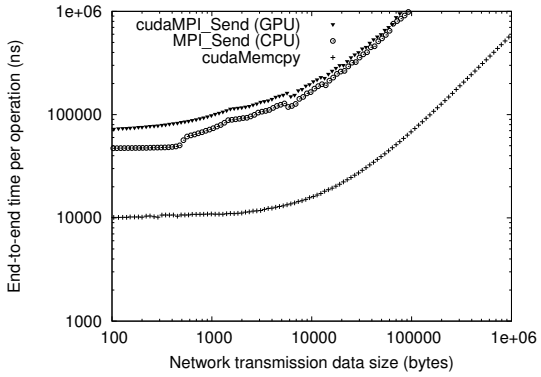

Fig. 5. End-to-end time for network operations on gigabit Ethernet.

time to do a CPU-GPU memory copy $t_M = \alpha_M + \beta_M n$ or network communication $t_N = \alpha_N + \beta_N n$. Figure 2 shows typical values for these constants.

The surprising fact on modern hardware is that $\alpha >> \beta$: the per-message cost is much larger than the per-byte cost, so small messages are expensive. Patterson[10] points out that the α term is both large and increasing, yet widely ignored both in marketing (e.g., gigabit ethernet, 3Gb/s SATA) and by many designers. We have found many real parallel GPU applications that operate on small n problems are completely dominated by α_K or α_M startup costs.

III. Message Passing on the GPU with CUDA

Unfortunately, there is no direct connection between our network device and GPU memory. Thus to send GPU data across the network, we must copy the send-side GPU data to CPU memory, send across the network using a standard CPU interface such as MPI, and finally copy the received data from CPU memory into GPU memory.

The simplest possible message-passing interface is to send a contiguous run of bytes or GPU pixels across the network. Contiguous messages work nicely with simple data structures such as arrays, dense matrices, or regular grids. Other applications, such as those sending sparse matrices or irregular meshes, send and receive from noncontiguous regions of memory.

A. Naive GPU Message Passing in CUDA

In CUDA, `cudaMemcpy` can copy data in either direction between GPU and CPU memory. This call runs at over a gigabyte per second between GPU memory and *pinned* CPU memory (unpageable memory allocated with `cudaMallocHost`). Ordinary unpinned CPU memory copies run at about half this bandwidth. One conceptual limitation with `cudaMemcpy` is that both GPU and CPU buffers must be contiguous groups of bytes; no stride or derived datatypes are allowed.

In CUDA, the per-memory-copy setup cost α_M is substantial, around 10 microseconds per call, which is faster than Ethernet but much worse than most high-performance networks. Per-pixel time β_M is much better, mostly limited by the PCI Express bus to about a gigabyte per second, and so GPU-CPU memory copy bandwidth is competitive with modern networks.

A naive blocking implementation for contiguous GPU-to-GPU data transmission is thus:

```
// Sending CPU code:
 cudaMemcpy(cpuBuffer,gpuBuffer,n,...
 MPI_Send(cpuBuffer,n,MPI_FLOAT,...
// Receiving CPU code:
 MPI_Recv(cpuBuffer,n,MPI_FLOAT,...
 cudaMemcpy(gpuBuffer,cpuBuffer,n,...
```

We expect this implementation to take CPU time $t = 2\,t_M + t_N$, and experimentally, it indeed does as shown in Figure 4 for CPU overhead (repeated send time) and Figure 5 for end-to-end time (pingpong time). The only minor detail to note is that we should be sure to have previously allocated `cpuBuffer` as pinned CPU memory, for higher copy bandwidth. It would be best to manage this pinned memory in a single location in the code, especially since pinned memory allocations currently take absurdly long, millions of nanoseconds, as shown in Figure 3.

Hence in cudaMPI we provide the following blocking interface for GPU-to-GPU data transmission, which simply moves

Function	Performance Model	Tolerance
CUDA Kernel writing to GPU memory	4.9×10^3ns + n * 0.012 ns/byte	+0%/-13%
CUDA Kernel writing to mapped CPU RAM	5.3×10^3ns + n * 0.5 ns/byte	+0%/-20%
C++ new and delete (unpinned CPU RAM)	100 ns + n * 0.7×10^{-3}ns/byte	+94%/-58%
cudaMallocHost (pinned CPU RAM)	3.5×10^6ns + n * 0.7 ns/byte	+0%/-23%
cudaMalloc (GPU memory)	if ($n < 2.05\times10^3$ bytes) then 1.49×10^3ns else 196.67×10^3ns + n * 0.13 ns/byte	+0%/-8%

Fig. 3. Measured performance of CUDA functions operating on n bytes. GPU memory is faster than mapped CPU RAM, and pinned allocation is expensive.

the data copy call and related `cpuBuffer` management inside the cudaMPI library. The result is exactly the MPI interface, but with the data passed in as a GPU memory pointer:

```
// Sending CPU code:
 cudaMPI_Send(gpuBuffer,n,MPI_FLOAT,...
// Receiving CPU code:
 cudaMPI_Recv(gpuBuffer,n,MPI_FLOAT,...
```

Similar copy-communicate-copy wrappers can identically be constructed around `MPI_Bcast`, `MPI_Reduce` and the other collective MPI calls, whose performance we list in Figure 6 for cudaMPI and plain MPI on the same hardware. Note that even though we are communicating GPU data buffers, the GPU is not at all involved in the communication at this point, and hence can be busy doing other work.

B. Asynchronous GPU Message Passing in CUDA

Because both `MPI_Recv` and `cudaMemcpy` are blocking functions, that do not return until their data transfer is complete, with the blocking `cudaMPI_Recv` interface above, the CPU cannot accomplish useful work during the transfer.

MPI provides nonblocking point-to-point communication functions called `MPI_Isend` and `MPI_Irecv`, which return an `MPI_Request` object that must subsequently be polled to make communication progress. In CUDA, pinned CPU RAM can be copied to or from the GPU asynchronously using `cudaMemcpyAsync`, chained to other copies or kernel executions in a CUDA "Stream", and finally probed for completion with `cudaStreamQuery`.

In principle, the code for an asynchronous CUDA send would look like:

```
cudaMemcpyAsync(cpuBuffer,gpuBuffer,n,...
while (!cudaStreamQuery(s)) doWork();
MPI_Isend(cpuBuffer,n,MPI_FLOAT,...
while (!MPI_Test(&request,...)) doWork();
```

In practice, we invert this "doWork()" call structure. The code actually getting work done issues a `cudaMPI_Isend` to start the communication, then periodically calls a new `cudaMPI_Test` function, which internally calls the appropriate communication or query function.

But for an application, choosing between asynchronous and synchronous messaging is more subtle than it might appear. For short messages, issuing `cudaMemcpy` costs α_M, around 10,000ns; while issuing `cudaMemcpyAsync` costs much less, around 1,000ns. The CPU can do useful application work from that point on, yet overall `cudaMemcpyAsync` takes much longer to complete, about 40,000ns. Issuing a `cudaStreamQuery` costs 10,000ns of CPU time if the transfer is still ongoing, but is nearly instant once the transfer is finished. The most efficient approach, then, is to start the transfer, do useful CPU work until the performance model indicates the transfer should be finished, then issue one query to verify that the transmission is complete. Asynchronous transmission is hence only beneficial if the CPU's additional work during the transmission is worth the loss in end-to-end communication latency.

C. Mapping CPU RAM for Faster CUDA Copies

The significant drawback of both the naive and nonblocking message passing implementations described above is the memory copy latency α_M. Each explicit memory copy operation into and out of graphics memory must go through the CPU kernel driver, as well as make several high-latency transactions across the PCI Express bus. Worse, each GPU to GPU data transmission requires two such copies, one on each end.

On the latest cards, CUDA as of version 2.2 supports directly mapping specially allocated CPU RAM into GPU address space [11], so the GPU can directly read or write mapped CPU RAM over the PCI Express bus. Bandwidth, and hence β_M, is nearly identical to a direct memcpy. But because mapped memory accesses are serviced in hardware, they have a near-zero per-transaction α_M cost. Still, Figure 3 shows the bandwidth of mapped access is dozens of times slower than that of normal graphics memory, so it is not affordable to simply run all GPU computations directly in mapped CPU RAM. And unfortunately the reverse mapping

Function	Performance Model	Tolerance	Discussion
cudaMemcpy (CPU-GPU)	11.20×10^3ns + n * 0.50 ns/byte	+0%/-3%	This α_M cost gets added to each cudaMPI communication.
MPI_Send (end-to-end)	47×10^3ns + n * 8.5 ns/byte	+31%/-1%	Half a round trip: counts CPU and network time.
cudaMPI_Send (end-to-end)	72×10^3ns + n * 9.5 ns/byte	+24%/-0%	Must pay one $\alpha_M + n\,\beta_M$ on each end.
MPI_Send (oneway)	4.3×10^3ns + n * 8.5 ns/byte	+22%/-43%	Repeated messages: counts CPU overhead only.
cudaMPI_Send (oneway)	17×10^3ns + n * 9.5 ns/byte	+16%/-41%	Must pay cudaMemcpy cost.
cudaMemcpyAsync (CPU-GPU)	58×10^3ns + n * 0.5 ns/byte	+29%/-2%	Async copy has an even higher α_M cost.
MPI_Isend (oneway)	4.2×10^3ns + n * 8.5 ns/byte	+23%/-37%	Same cost as MPI_Send.
cudaMPI_Isend (oneway)	54×10^3ns + n * 9.5 ns/byte	+15%/-43%	Must pay higher cudaMemcpyAsync cost.
MPI_Bcast (10 nodes)	16×10^3ns + n * 11.0 ns/byte	+81%/-13%	Similar cost to point-to-point message.
cudaMPI_Bcast (10 nodes)	32×10^3ns + n * 13.2 ns/byte	+56%/-19%	Must pay memory copy costs.
MPI_Reduce (10 nodes)	3.9×10^3ns + n * 13.1 ns/byte	+61%/-30%	β cost is higher due to MPI_OP.
cudaMPI_Reduce (10 nodes)	31×10^3ns + n * 14.2 ns/byte	+38%/-42%	α cost is due to copy in and out.
MPI_Allreduce (10 nodes)	225×10^3ns + n * 20 ns/byte	+74%/-0%	Requires several network roundtrips.
cudaMPI_Allreduce (10 nodes)	232×10^3ns + n * 21 ns/byte	+72%/-0%	Small additional cost.

Fig. 6. Measured performance for CUDA communication operations for varying numbers of bytes n over gigabit Ethernet.

is not yet supported, so one cannot access fast GPU memory directly from the CPU.

However, because GPU-mapped CPU RAM can be accessed directly by both CPU and GPU, this space can be used as a temporary buffer for low-latency communication:

```
// Send side GPU kernel fills buffer
cpuBuffer[i]=...
// Sending CPU code:
MPI_Send(cpuBuffer,n,...
// Receiving CPU code:
MPI_Recv(cpuBuffer,n,...
// Receive size GPU kernel extracts data
... = cpuBuffer[i];
```

Ignoring the GPU kernel startup time, this should and does take time $t = 2\,n\,\beta_M + t_N$, which is missing the $2\,\alpha_M$ term (approximately 20,000ns) taken by a naive copy. Even if a dedicated GPU kernel needs to be run for each copy, the corresponding α_K startup costs are still lower than the cost for a memory copy. Finally, many real applications can fold the message buffer copy into the preceding GPU computation, as we examine in Section V.

D. Noncontiguous GPU Message Passing in CUDA

Some applications need to communicate only a small noncontiguous subset of their GPU data. For example, a finite element application with a 1 million node mesh sitting on the GPU might only need to communicate forces for the 20,000 boundary nodes. We can handle noncontiguous communication by:

- **Renumber** the mesh's nodes to force these boundary nodes to sit together in one contiguous group on the GPU, hence taking time $t = 2\,t_M + t_N$. This works nicely, but adds application-level complexity, and may hurt performance for applications that had numbered their nodes for better access locality.
- **CPU copy** the noncontiguous data by making many separate contiguous memory copy calls, taking time $t = 2\,s\,\alpha_M + t_N$ for s separate small sections. Unfortunately, the memory copy startup overhead α_M is far too big to make this feasible. For example, copying out s =20,000 nodes at 10 microseconds each would take 0.2 seconds!
- **GPU copy** the noncontiguous data into a contiguous buffer by running a special GPU kernel that gathers up the data on the send side, use the CPU to copy out and deliver the resulting buffer, and finally scatter the data back out with a receive-side GPU kernel. The communication time is hence $t = 2\,t_K + 2\,t_M + t_N$. The main cost here is the extra GPU kernel invocation time α_K, unless this can be folded into the previous and next kernels.
- **Map** noncontiguous buffers into the CPU, which can then gather the data needed into a contiguous buffer. This relies on the PCI Express bus hardware to reduce the per-copy overhead. Currently CUDA mapping as described in Section III-C can only go the other way, providing GPU access to CPU buffers.

Currently, the best approach for noncontiguous communication is to use a GPU kernel to copy the noncontiguous data into a contiguous and GPU mapped portion of CPU RAM.

IV. GPU Communication in OpenGL

OpenGL applications store their application data in *textures*, which are simply 2D or 3D arrays of pixels. Each pixel can store a single float (`GL_LUMINANCE32F_ARB`), four floats (`GL_RGBA32F_ARB`), or a variety of lower-precision data formats. Textures can be read freely, taking the place of arrays (and all other data structures) in GPU programs called pixel shaders. New textures are created on the GPU by running a pixel shader in an output device called an OpenGL *framebuffer object*. A pixel shader's only side effects are the new pixels written into the current output texture, and the same texture cannot be used for reading and writing at the same time. Though this lack of random write capability makes data access race conditions impossible, it also makes some algorithms difficult to express efficiently.

In OpenGL, `glReadPixels` copies GPU framebuffer pixels to CPU memory, and `glDrawPixels` and `glTexSubImage2D` copy CPU memory into GPU framebuffer or texture pixels. All three calls work with any 2D rectangle of GPU pixels, although they require a contiguous CPU buffer. These functions also synchronize with the GPU, waiting until any pending commands are complete, and then block the CPU until the copy is complete. An asynchronous data copy can be achieved using *pixel buffer objects*, OpenGL's abstraction for memory areas. OpenGL supports a command called `glMapBuffer` to give the CPU direct access to an OpenGL buffer, but unlike CUDA host-mapped memory this command seems to normally be implemented via a whole-buffer copy from GPU to CPU memory, which limits performance.

Function	Performance Model	Tolerance	Discussion
glReadPixels	31×10^3ns + n * 1.0 ns/byte	+0%/-28%	High α, but bandwidth is good.
glDrawPixels	22×10^3ns + n * 3.8 ns/byte	+94%/-10%	Poor bandwidth, especially for below-64KB transfers.
glTexSubImage2D	40×10^6ns	+30%/-0%	Internally copies the entire texture (not just the "SubImage").
glBufferData (PBO)	382 ns + n * 0.5 ns/byte	+2%/-39%	Pixel Buffer Object (PBO) gives excellent latency and good bandwidth.
glTexSubImage2D (PBO)	3.14×10^3ns + n * 0.38 ns/byte	+3%/-1%	This function is much faster loading data from a Pixel Buffer Object.
glMPI_Send (oneway)	40×10^3ns + n * 10.7 ns/byte	+4%/-48%	α is mostly by the receive-side glReadPixels, β mostly the network.
glMPI_Send (end-to-end)	91×10^3ns + n * 10.7 ns/byte	+19%/-4%	Includes network latency.

Fig. 7. Measured performance of OpenGL operations for varying numbers of bytes n over gigabit Ethernet. glTexSubImage2D from a PBO is fast.

Rendered pixels can be extracted from an OpenGL framebuffer object onto the CPU using `glReadPixels`, which has reasonable latency and obtains near-peak bandwidth. No corresponding pixel extraction call exists for textures, though a texture can be attached to a framebuffer and then read with `glReadPixels`. This appears to be the best way to send data from a GPU in OpenGL.

Receiving data into an OpenGL texture efficiently is much trickier. The `glTexSubImage2D` interface works, but when called from ordinary CPU data to modify a texture that is also mapped as a framebuffer object, its performance appears to become proportional to the total size of the texture, not the size of the changed region. For a typical application that communicates only a few kilobytes from a hundred-megabyte texture, this call is hence extremely slow. For these large textures `glDrawPixels` has a much smaller α_M startup cost, but only provides a fraction of the bandwidth we might expect. Pixel buffer objects provide similarly poor bandwidth when used with `glDrawPixels`.

Substantially higher performance can be achieved this way:

1) Allocate a pixel buffer object of the appropriate size.
2) Map the pixel buffer object into CPU memory using `glMapBuffer`.
3) `MPI_Recv` the network data directly into the pixel buffer object.
4) Unmap and `glTexSubImage2D` the pixel buffer object into the desired region of the texture.

The blocking glMPI implementation is hence:

```
//Send pixels from the current framebuffer
int glMPI_Send(int X, int Y, int W,int H,
  GLenum F,int P, int T, MPI_Comm C)
{
  glMPI_Buf buf(W,H,F); // CPU buffer
  glReadPixels(X,Y,W,H,
    buf.glformat,buf.gltype,buf.data);
  return MPI_Send(buf.data,buf.count,...);
}

//Receive pixels into the current texture
int glMPI_Recv(int X, int Y, int W,int H,
  GLenum F,int P, int T, MPI_Comm C,...)
{
  glMPI_BufSize buf(W,H,F);
  GLuint pbo=...pbo recycling omitted...
  GLenum pt=GL_PIXEL_UNPACK_BUFFER_ARB;
  glBindBuffer(pt, pbo);
  // Receive directly into CPU-mapped PBO
  void *V=glMapBuffer(pt,GL_WRITE_ONLY);
  MPI_Recv(V,buf.count,...);
  glUnmapBuffer(pt);
  // Upload received data to GPU texture
  glTexSubImage2D(GL_TEXTURE_2D,0,
    X,Y,W,H,buf.glformat,buf.gltype,0);
  glBindBuffer(pt, 0); /* restore PBO */
}
```

Fig. 8. Powerwall used to benchmark cudaMPI and glMPI.

This results in quite reasonable latency and bandwidth for glMPI communication, as shown in Figure 7. One significant advantage of the glMPI communication interface over that of cudaMPI is the ability to send arbitrary rectangles of pixels. Also, nearly identical performance is obtained whether sending wide flat rows or tall skinny columns of pixels—this is due to the GPU storing textures not in row or column major form, but in *swizzled* layout [12] following a space-filling curve.

V. Applied Performance Analysis

We recently upgraded our 20-screen, 10-node powerwall display and compute cluster[3] with ten NVIDIA GeForce GTX 280 graphics cards. Figure 8 shows this cluster running a GPGPU wave simulation application built using glMPI and our powerwall display library MPIglut [6].

We used both cudaMPI and glMPI to parallelize a distributed-memory GPU cluster port of an existing trivial MPI 5-point stencil 2D simulation application with 1D decomposition. The simulation stores one floating-point temperature at each pixel on a 2D grid, and at each step averages the temperatures from the left, right, top, and bottom neighbors, which costs just four floating-point operations per output pixel.

A. Stencil Performance in cudaMPI

Stencil computations like this are normally memory bound. Stencils are also known to be somewhat tricky to tune in CUDA [13] because the stencil's memory reads cannot all be aligned in memory (e.g., if the left access is aligned, then the right cannot be), and hence the GPU cannot coalesce the stencil's memory accesses. First fetching input data using

[3]Ten nodes with Intel Core2 Duo 6300 CPUs, CUDA 2.2, NVIDIA 185.18 driver, Linux 2.6.24, OpenMPI 1.3, switched gigabit ethernet.

# GPUs	Output Rate	Computation	Network	Efficiency
1	6.1 Gpix/s	7.81 ms	- ms	100%
2	11.3 Gpix/s	3.91 ms	0.35 ms	92%
3	16.2 Gpix/s	2.62 ms	0.35 ms	88%
4	20.7 Gpix/s	1.97 ms	0.36 ms	84%
5	25.0 Gpix/s	1.58 ms	0.35 ms	81%
6	28.5 Gpix/s	1.32 ms	0.37 ms	77%
7	32.4 Gpix/s	1.13 ms	0.36 ms	75%
8	35.3 Gpix/s	1.00 ms	0.37 ms	71%
9	38.2 Gpix/s	0.91 ms	0.39 ms	67%
10	40.7 Gpix/s	0.80 ms	0.39 ms	66%

Fig. 9. cudaMPI time per step for 48 Mpix problem on gigabit Ethernet.

# GPUs	Output Rate	Computation	Network	Efficiency
1	-	-	-	-
2	20.4 Gpix/s	2.01 ms	0.34 ms	85%
3	27.8 Gpix/s	1.36 ms	0.36 ms	78%
4	34.2 Gpix/s	1.06 ms	0.35 ms	72%
5	39.2 Gpix/s	0.85 ms	0.37 ms	66%
6	43.7 Gpix/s	0.72 ms	0.38 ms	61%
7	47.9 Gpix/s	0.63 ms	0.37 ms	57%
8	51.3 Gpix/s	0.55 ms	0.38 ms	54%
9	53.3 Gpix/s	0.51 ms	0.39 ms	50%
10	55.3 Gpix/s	0.47 ms	0.40 ms	46%

Fig. 10. glMPI time per step, same problem on gigabit Ethernet. The problem exceeds OpenGL's texture size limit on one GPU.

coalesced reads, and then using a CUDA `__shared__` buffer for neighboring values, results in about a fourfold performance improvement on older GeForce 8000-series hardware, although the benefit is only about 40% on the new GTX series cards with their lower penalty for noncoalesced memory accesses. With our tuned GPGPU simulation each GPU produces over six billion floating-point pixels per second, over twenty times faster than our original CPU-based simulation. This pixel rate appears to be similar to that achieved by an advanced machine-tuned 3D stencil implementation running on the same hardware by Datta et al [3].

Though the bulk of the simulation's data is permanently stored on the GPU, it took only a few lines of code to exchange the simulation boundary data with adjacent distributed-memory GPU compute nodes using `cudaMPI_Send` and `cudaMPI_Recv`. Though easy, these blocking calls do not allow any overlapping of computation and communication.

In general it is difficult to obtain high parallel efficiency when combining a fast GPU like the GeForce GTX 280 with a relatively slow network like gigabit Ethernet.[4] However, since both fast GPUs and commodity network clusters are increasingly common, we feel this combination of hardware is a useful research target. Figure 9 shows cudaMPI's delivered performance per step on our stencil application for a fixed-size 3,000 by 16,000 pixel problem, the largest such problem that would fit on a single node. At each step, from each GPU cudaMPI sends and receives two 12KB messages, which takes 0.35ms, or a throughput of around 65MB per second per node per direction, acceptable for gigabit ethernet. Parallel efficiency is impacted by this non-overlapped network time, which would be reduced when using a faster network. Also, for a more computation-intensive nontrivial problem, this small and fixed communication cost would represent a smaller fraction of the total runtime.

Adding nonblocking communication only improves this version's efficiency by a few percent, mostly because there is no useful work for the CPU to do during the communication. Similarly, though CUDA's GPU-mapped CPU RAM allows us to fold the memory copy cost into the GPU kernel, this transformation adds substantial complexity, while the bottom line performance improvement is quite small.

[4]Parallel GPU clusters are the polar opposite of 1997's beloved ASCI Red, which used slow 200MHz Pentium CPUs on a fast 800MB/s interconnect.

B. Stencil Performance in glMPI

OpenGL's support for texture access is very good. In particular, unlike CUDA global memory reads, OpenGL texture accesses are cached, so no special code transformations are needed to decrease global memory bandwidth usage. We obtained surprisingly good performance from an extremely simple OpenGL pixel shader, which implements the same 4-point stencil that required 35 lines to optimize in CUDA. OpenGL's sequential performance on this code is almost double that of CUDA, which implies our CUDA implementation could be improved, perhaps by using CUDA *arrays* (the CUDA interface to the GPU's texture hardware).

Our OpenGL implementation's timeloop looks like this:

```
for(t=0;t<niterations;t++) {
// Exchange boundaries with GPU neighbors
  SRC->bind();
  glMPI_Send(1,1,      WIDTH,1, type, ...
  glMPI_Send(1,DEPTH,  WIDTH,1, type, ...
  glMPI_Recv(1,DEPTH+1,WIDTH,1, type, ...
  glMPI_Recv(1,0,      WIDTH,1, type, ...

// Run stencil computation on our data
  GPU_RUN(*DEST,
" float l=texture2D(SRC,P+vec2(-S,0.0));"
" float r=texture2D(SRC,P+vec2(+S,0.0));"
" float t=texture2D(SRC,P+vec2(0.0,-S));"
" float b=texture2D(SRC,P+vec2(0.0,+S));"
" gl_FragColor=0.25*(l+r+t+b);"
  )
  SRC->swapwith(*DEST); //pingpong
}
```

In the end, glMPI's communication performance is very similar to cudaMPI, which is reassuring since both interfaces use the same hardware. Figure 10 shows glMPI's delivered performance on our GL stencil application for the same fixed-size 3,000 by 16,000 pixel problem. Unfortunately, the dimensions of this problem exceed our OpenGL driver's fixed

8192x8192 pixel texture size limit, so performance measurements begin with 2 GPUs. We calculated parallel efficiency by assuming one GPU's computation time would be double the two GPU computation time (i.e., perfect speedup of the computation portion).

VI. Conclusions and Future Work

As general-purpose graphics processing units are adopted more and more widely, application developers will need well designed and high performance libraries to use on this new hardware. We have presented and benchmarked cudaMPI and glMPI, message passing libraries for distributed-memory GPU clusters. cudaMPI extends the popular parallel programming interface MPI to work with data stored on the GPU using the CUDA programming interface. glMPI does the same for OpenGL GPU programs. We have carefully examined the performance of both implementations, and find them useful for real applications.

As is good software design, a single application can arbitrarily mix functionality from the libraries CUDA, MPI, cudaMPI, OpenGL, and glMPI. Because the machine's native MPI library is used in a straightforward way, it is even possible to send floating-point data from CUDA with `cudaMPI_Send` and receive it into OpenGL pixels with `glMPI_Recv`, or send from OpenGL and receive on the CPU with `MPI_Recv`. This sort of communication orthogonality is useful both for debugging simple applications and for decoupling the components of more complex applications.

Still, both cudaMPI and glMPI are incomplete in several senses. First, neither implements the full range of features supported by MPI. Some missing features, such as derived datatypes with noncontiguous storage, would be very difficult to implement efficiently on the GPU with currently available GPU interfaces, which usually assume contiguous CPU-side storage. Other missing features, such as MPI communicator splitting, can be handled adequately by the underlying native MPI implementation. Both cudaMPI and glMPI require frequent CPU attention even in non-blocking mode; a more fully asynchronous interface such as Wesolowski's offload API [14] would allow less CPU-GPU coupling. It is also likely that new GPU-specific communication primitives could be added, such as a collective communication operation to build higher texture mipmap levels from distributed texture pieces, which would be very useful in multigrid applications.

We have not addressed the interesting topic of load balancing, which can be even more difficult on a hybrid SMP-SLI CPU-GPU cluster than a conventional CPU-based parallel machine. We believe that an MPI library capable of internode process migration, such as our Adaptive MPI [5], could be a very useful tool for solving these dynamic load balancing problems. For today's ubiquitous multicore machines, cudaMPI is designed to be threadsafe, although like MPI cudaMPI could provide much better specialized multicore support.

In the future, it would be also useful to create message-passing interfaces for other GPU programming interfaces, such as Microsoft's DirectX, which supports some new graphics operations better than OpenGL. Also, as implementations of the multivendor OpenCL interface become available, it would be useful to create a corresponding implementation of MPI to communicate OpenCL data.

We welcome readers to download [15], use, and extend cudaMPI and glMPI!

Acknowledgment

This research was supported in part by the NASA Applied Information Systems Research and NSF GK-12 programs.

References

[1] T. H. Myer and I. E. Sutherland, "On the design of display processors," *Commun. ACM*, vol. 11, no. 6, pp. 410–414, 1968.

[2] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008. [Online]. Available: http://graphics.idav.ucdavis.edu/publications/print_pub?pub_id=936

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12. [Online]. Available: http://www.eecs.berkeley.edu/~kdatta/pubs/sc08.pdf

[4] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104. [Online]. Available: http://www.open-mpi.org/papers/euro-pvmmpi-2004-overview/

[5] C. Huang, O. S. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003. [Online]. Available: http://charm.cs.uiuc.edu/research/ampi/

[6] O. S. Lawlor, M. Page, and J. Genetti, "MPIglut: Powerwall programming made easier," *Journal of WSCG*, pp. 130–137, 2008. [Online]. Available: http://www.cs.uaf.edu/sw/mpiglut

[7] Z. Fan, F. Qiu, and A. Kaufman, "ZippyGPU: Programming toolkit for general-purpose computation on GPU clusters," in *Poster at Supercomputing '06 Workshop General-Purpose GPU Computing: Practice And Experience*. ACM, 2006. [Online]. Available: http://gpgpu.org/static/sc2006/workshop/SBU_ZippyGPU_Abstract.pdf

[8] A. Moerschell and J. D. Owens, "Distributed texture memory in a multi-gpu environment," in *Graphics Hardware*, 2006. [Online]. Available: http://graphics.idav.ucdavis.edu/publications/print_pub?pub_id=886

[9] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, March 2009. [Online]. Available: http://graphics.idav.ucdavis.edu/publications/print_pub?pub_id=959

[10] D. A. Patterson, "Latency lags bandwith," *Commun. ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[11] R. Farber, "CUDA, supercomputing for the masses: Part 12," *Dr. Dobb's Journal*, May 2009. [Online]. Available: http://www.ddj.com/cpp/217500110

[12] B. Carter, *The Game Asset Pipeline*. Charles River Media, 2004. [Online]. Available: http://books.google.com/books?id=Sr2HKC46ImcC&pg=PA113

[13] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 79–84.

[14] L. Wesolowski, "An application programming interface for general purpose graphics processing units in an asynchronous runtime system," Master's thesis, Dept. of Computer Science, University of Illinois, 2008. [Online]. Available: http://charm.cs.uiuc.edu/papers/08-12

[15] O. S. Lawlor. (2009) cudaMPI software homepage. [Online]. Available: http://www.cs.uaf.edu/sw/cudaMPI