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 mimimize latency:
  - Caching (memory)
  - Speculation (branches, memory)
- To maximize bandwidth:
  - Clockrate (since 1970)
  - Pipelining (since 1985)
  - Parallelism (since 2004)
- For software, everything's transparent <u>except</u> 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 <u>eventually</u> "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 <u>cannot</u> 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**
- **<u>Copies</u> to/from host**
  - **>10us latency, <4GB/s bandwidth**

Quantitative
GPU Performance

CUDA: Memory Output Bandwidth

- **Build a trivial CUDA test harness:**

```
__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

Threads per Block	Number of Floats/Kernel				
	١K	١٠K	١٠٠K	١M	١٠M
١	٠.١	٠.٢	err	err	err
٢	٠.١	٠.٣	٠.٤	err	err
٤	٠.١	٠.٤	٠.٨	err	err
٨	٠.١	٠.٥	١.٤	err	err
١٦	٠.١	٠.٦	٢.٤	٣.٣	err
٣٢	٠.١	٠.٧	٣.٦	٦.٤	err
٦٤	٠.١	٠.٧	٤.٨	١١.٧	err
١٢٨	٠.١	٠.٧	٥.٤	١٦.٤	err
٢٥٦	٠.١	٠.٧	٥.٣	١٦.٤	٢٠.٧
٥١٢	٠.١	٠.٧	٥.٢	١٦.٣	٢٠.٧

Output Bandwidth (Gigafloats/second)

NVIDIA GeForce GTX 280

CUDA: Memory Output Bandwidth

Number of Floats/Kernel

Threads per Block	1K	10K	100K	1M	10M
1	0.1	0.2	err	err	err
2	0.1	0.3	0.4	err	err
4	0.1	0.4	0.8	err	err
8	0.1	0.5	1.4	err	err
16	0.1	0.6	2.4	3.3	err
32	0.1	0.7	3.6	6.4	err
64	0.1	0.7	4.8	11.7	err
128	0.1	0.7	5.4	16.4	err
256	0.1	0.7	5.3	16.4	20.7
512	0.1	0.7	5.2	16.3	20.7

Over 64K blocks needed

Output Bandwidth (Gigafloats/second)

NVIDIA GeForce GTX 280

CUDA: Memory Output Bandwidth

Kernel Startup Time Dominates

Number of Floats/Kernel

Threads per Block	1K	10K	100K	1M	10M
1	0.1	0.2	err	err	err
2	0.1	0.3	0.4	err	err
4	0.1	0.4	0.8	err	err
8	0.1	0.5	1.4	err	err
16	0.1	0.6	2.4	3.3	err
32	0.1	0.7	3.6	6.4	err
64	0.1	0.7	4.8	11.7	err
128	0.1	0.7	5.4	16.4	err
256	0.1	0.7	5.3	16.4	20.7
512	0.1	0.7	5.2	16.3	20.7

Output Bandwidth (Gigafloats/second)

NVIDIA GeForce GTX 280

CUDA: Memory Output Bandwidth

Block Startup Time Dominates

Threads per Block	1K	10K	100K	1M	10M
1	0.1	0.2	err	err	err
2	0.1	0.3	0.4	err	err
4	0.1	0.4	0.8	err	err
8	0.1	0.5	1.4	err	err
16	0.1	0.6	2.4	3.3	err
32	0.1	0.7	3.6	6.4	err
64	0.1	0.7	4.8	11.7	err
128	0.1	0.7	5.4	16.4	err
256	0.1	0.7	5.3	16.4	20.7
512	0.1	0.7	5.2	16.3	20.7

Number of Floats/Kernel

Output Bandwidth (Gigafloats/second)

NVIDIA GeForce GTX 280

CUDA: Memory Output Bandwidth

Number of Floats/Kernel

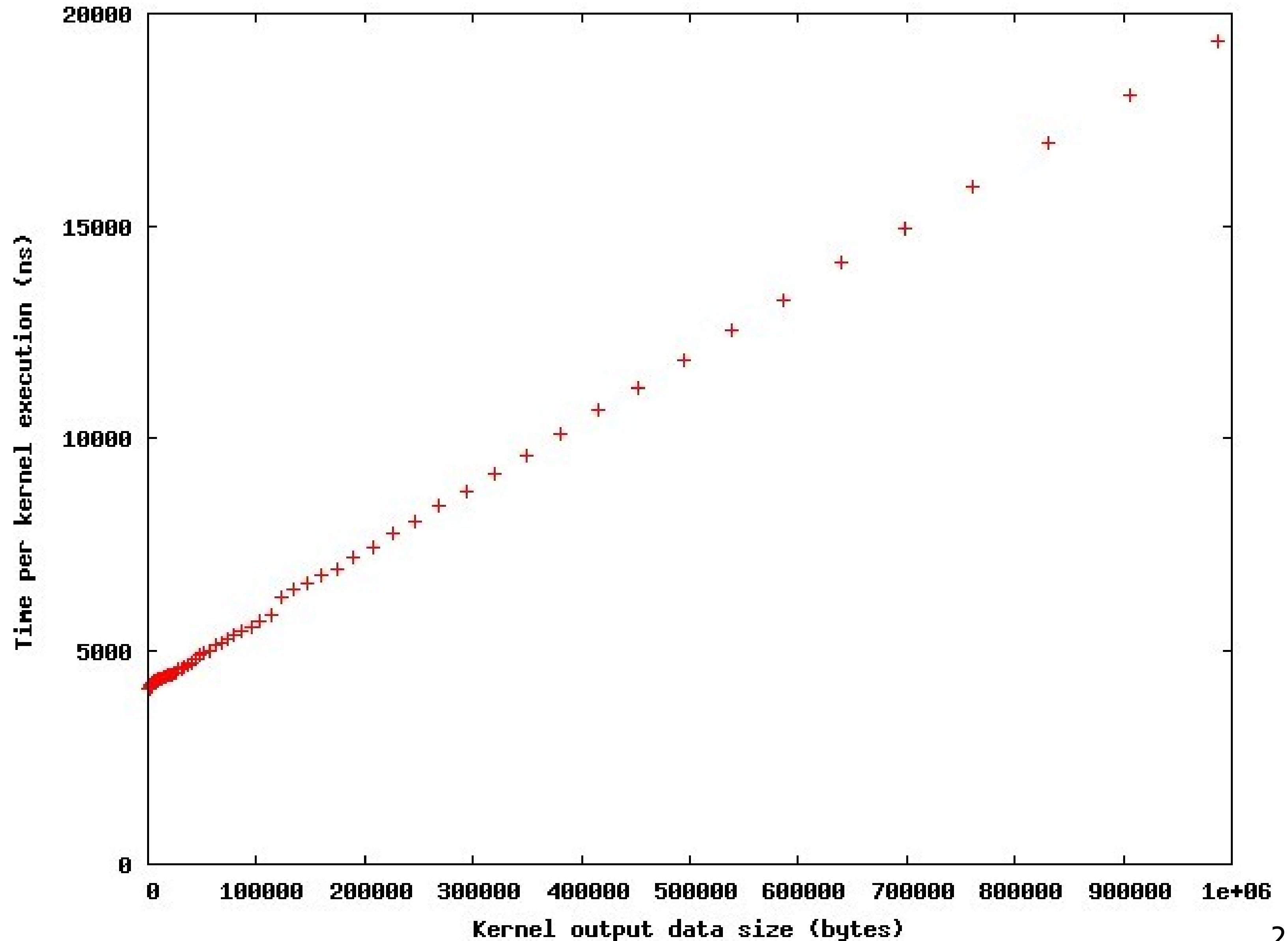
Threads per Block	1K	10K	100K	1M	10M
1	0.1	0.2	err	err	err
2	0.1	0.3	0.4	err	err
4	0.1	0.4	0.8	err	err
8	0.1	0.5	1.4	err	err
16	0.1	0.6	2.4	3.3	err
32	0.1	0.7	3.6	6.4	err
64	0.1	0.7	4.8	11.7	err
128	0.1	0.7	5.4	16.4	err
256	0.1	0.7	5.3	16.4	20.7
512	0.1	0.7	5.2	16.3	20.7

Good Performance

Output Bandwidth (Gigafloats/second)

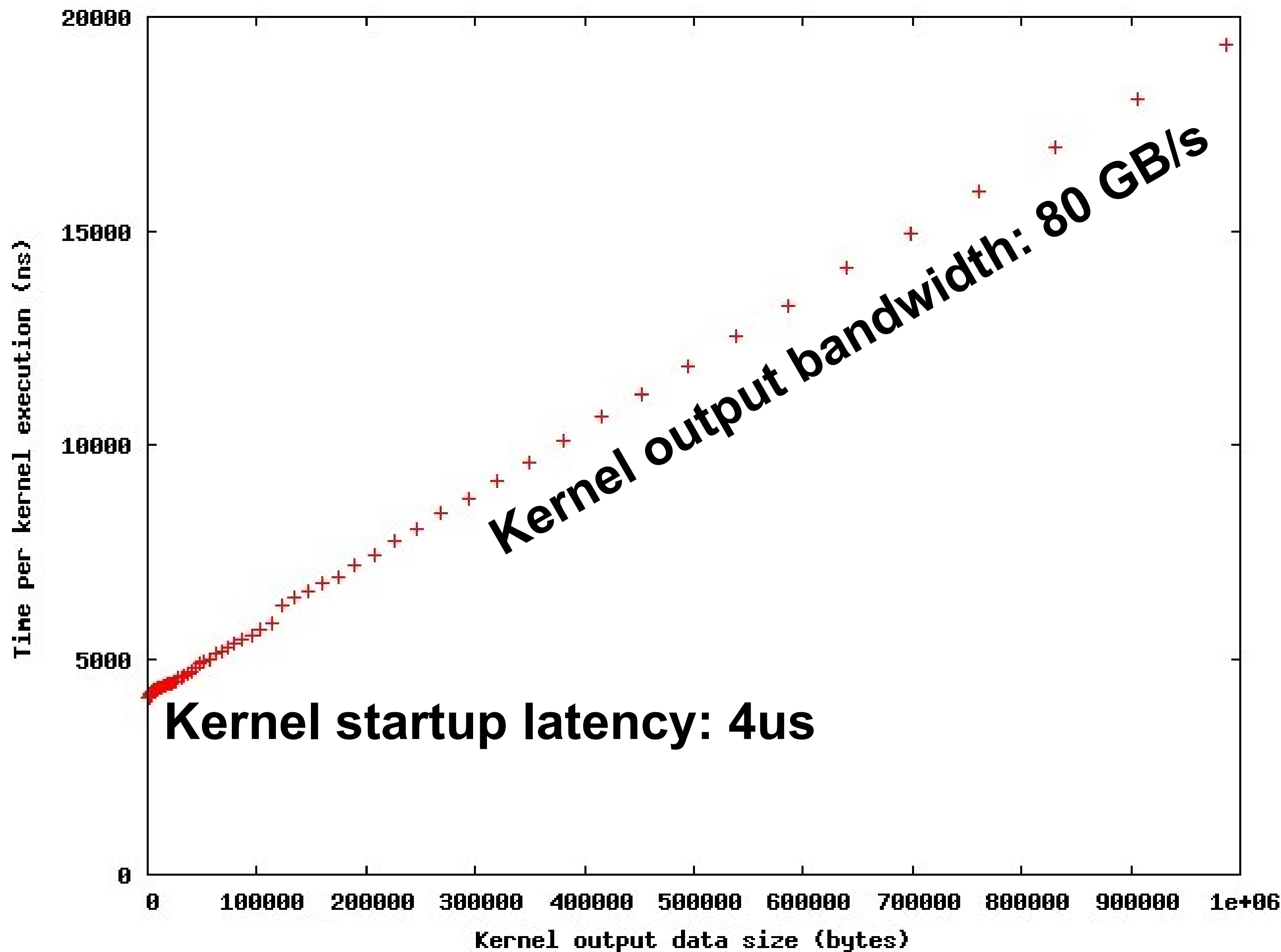
NVIDIA GeForce GTX 280

CUDA: Memory Output Bandwidth



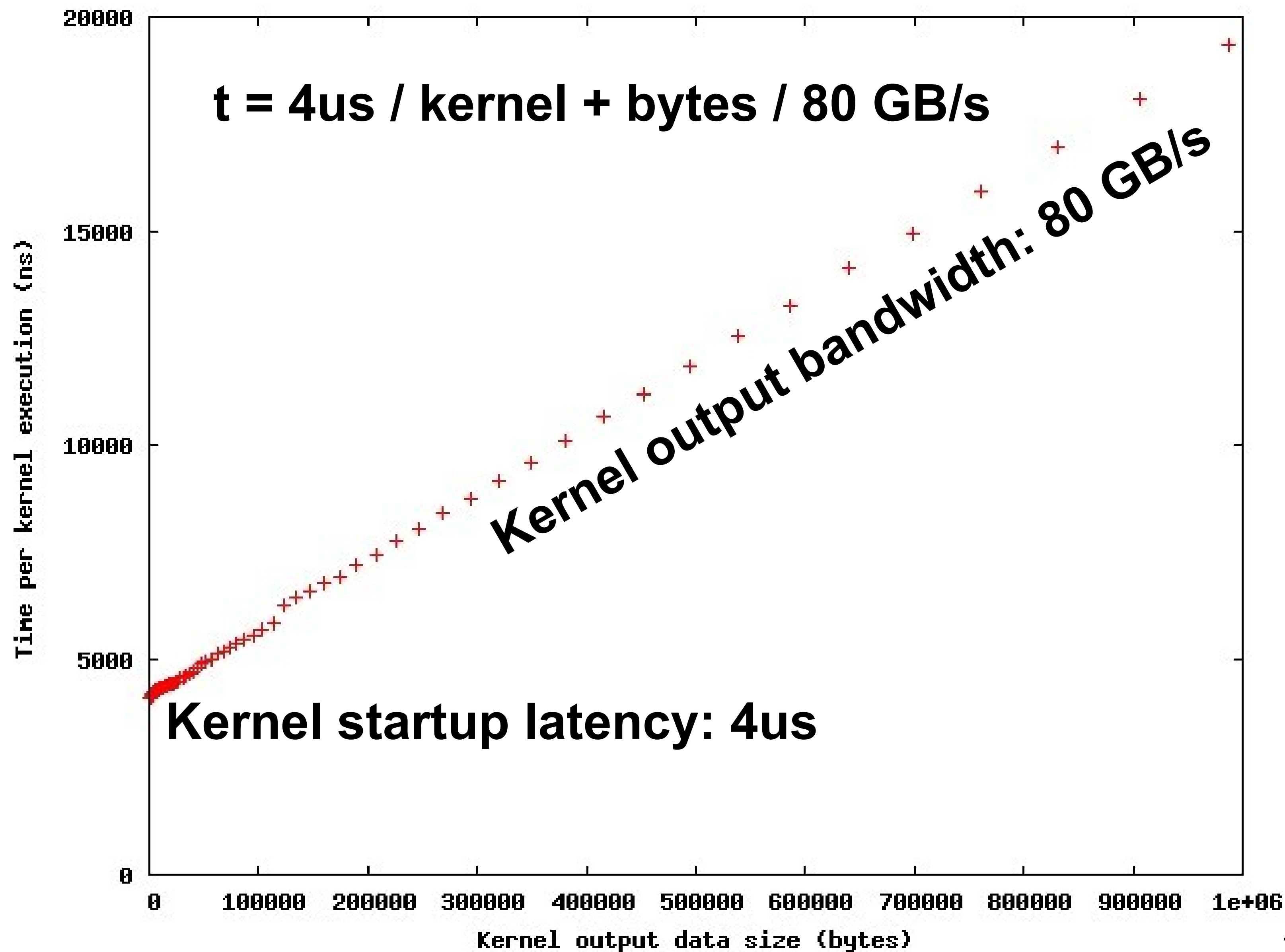
NVIDIA GeForce GTX 280, fixed 128 threads per block

CUDA: Memory Output Bandwidth



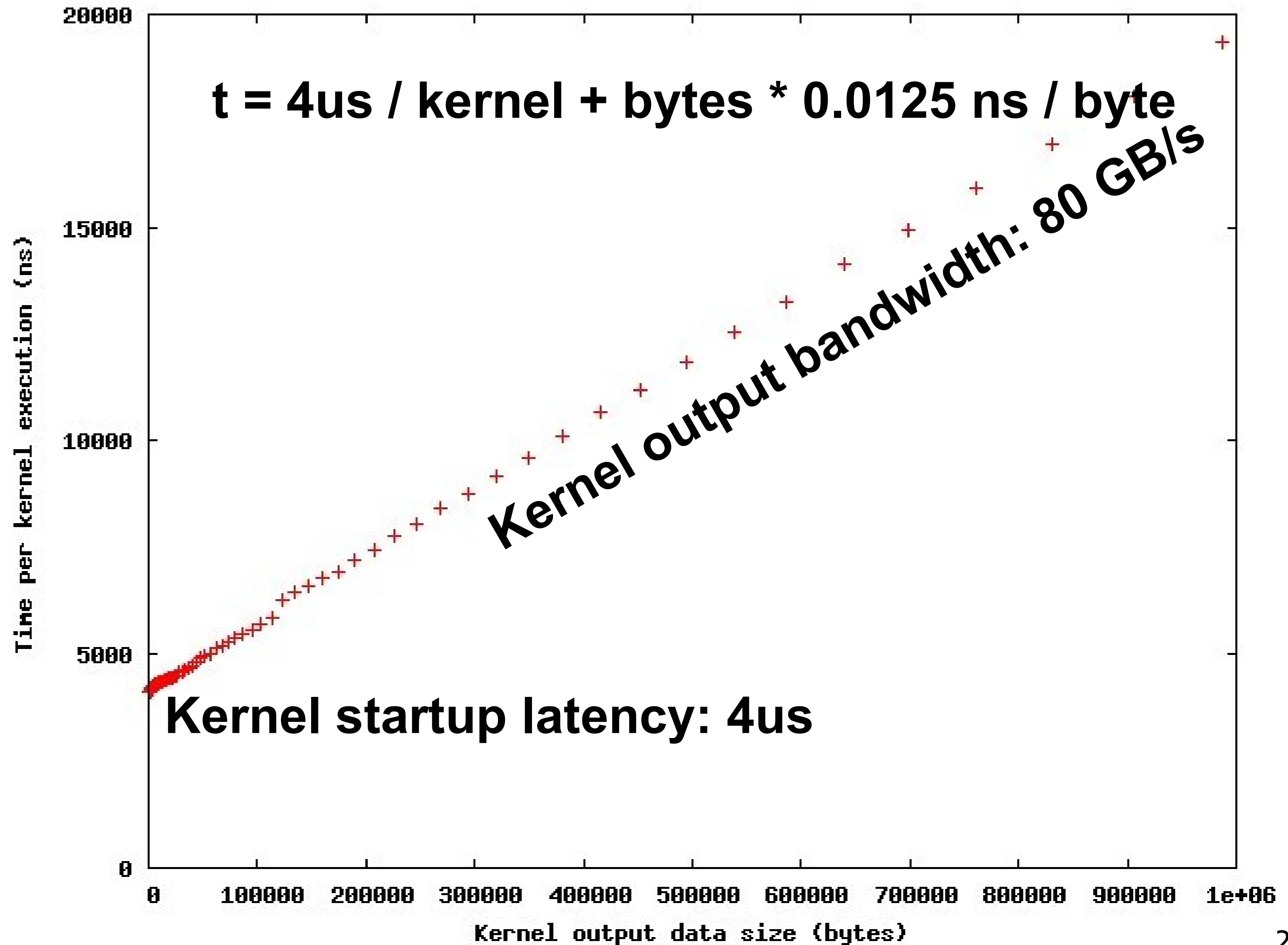
NVIDIA GeForce GTX 280, fixed 128 threads per block

CUDA: Memory Output Bandwidth



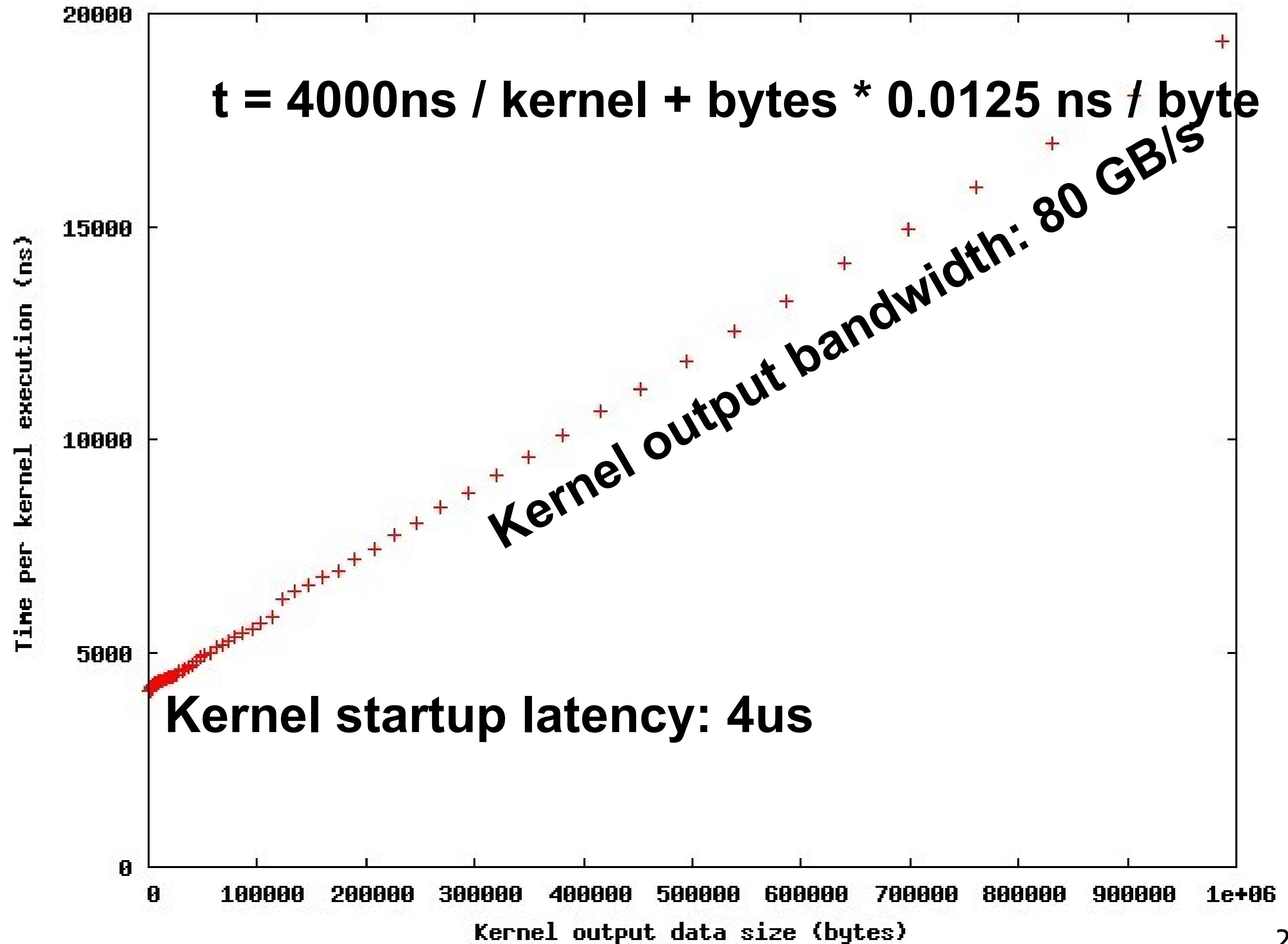
NVIDIA GeForce GTX 280, fixed 128 threads per block

CUDA: Memory Output Bandwidth



NVIDIA GeForce GTX 280, fixed 128 threads per block

CUDA: Memory Output Bandwidth



NVIDIA GeForce GTX 280, fixed 128 threads per block

CUDA: Memory Output Bandwidth

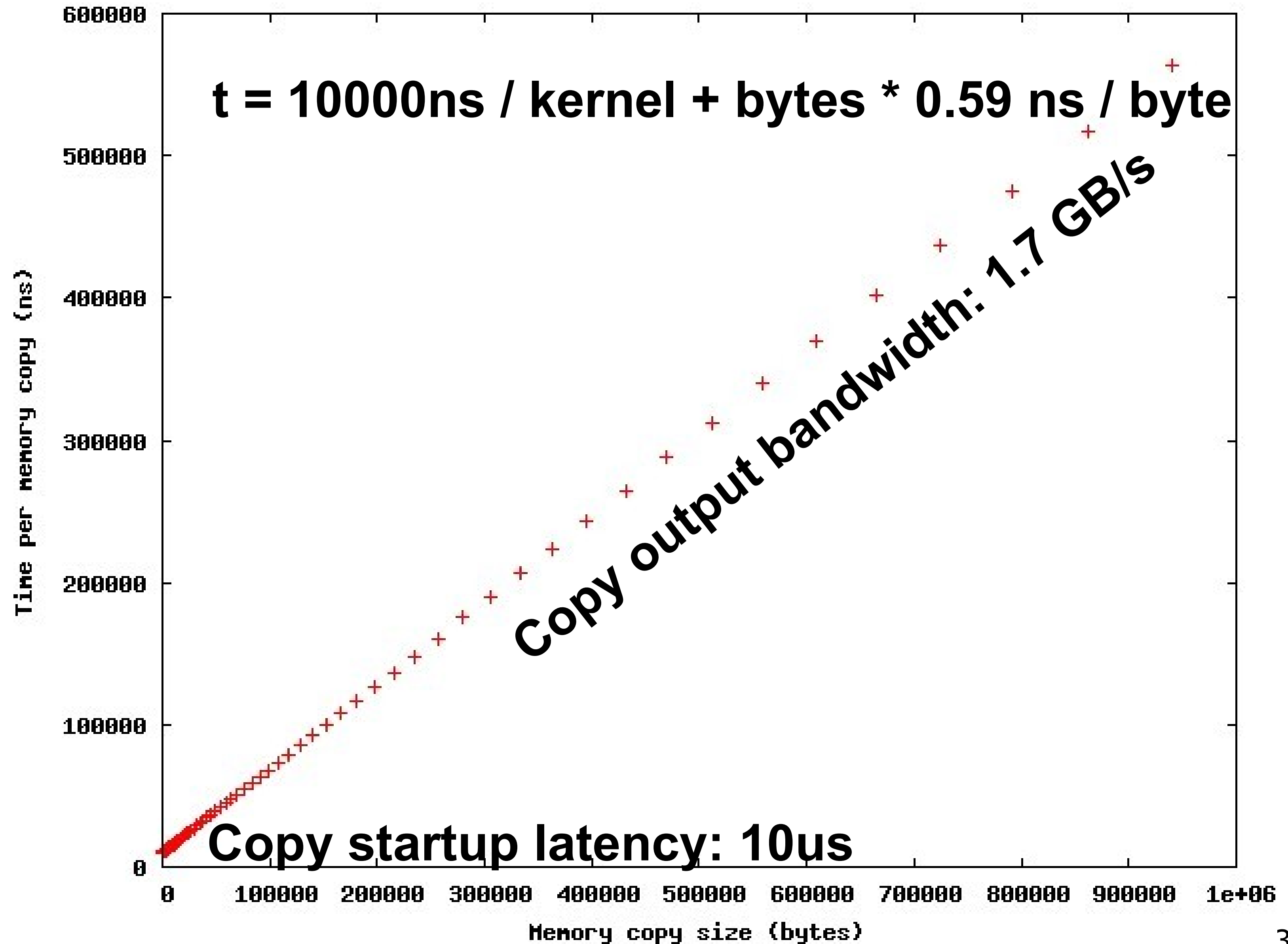
t = 4000ns / kernel + bytes * 0.0125 ns / byte

WHAT!?

time/kernel = 320,000 x time/byte!?

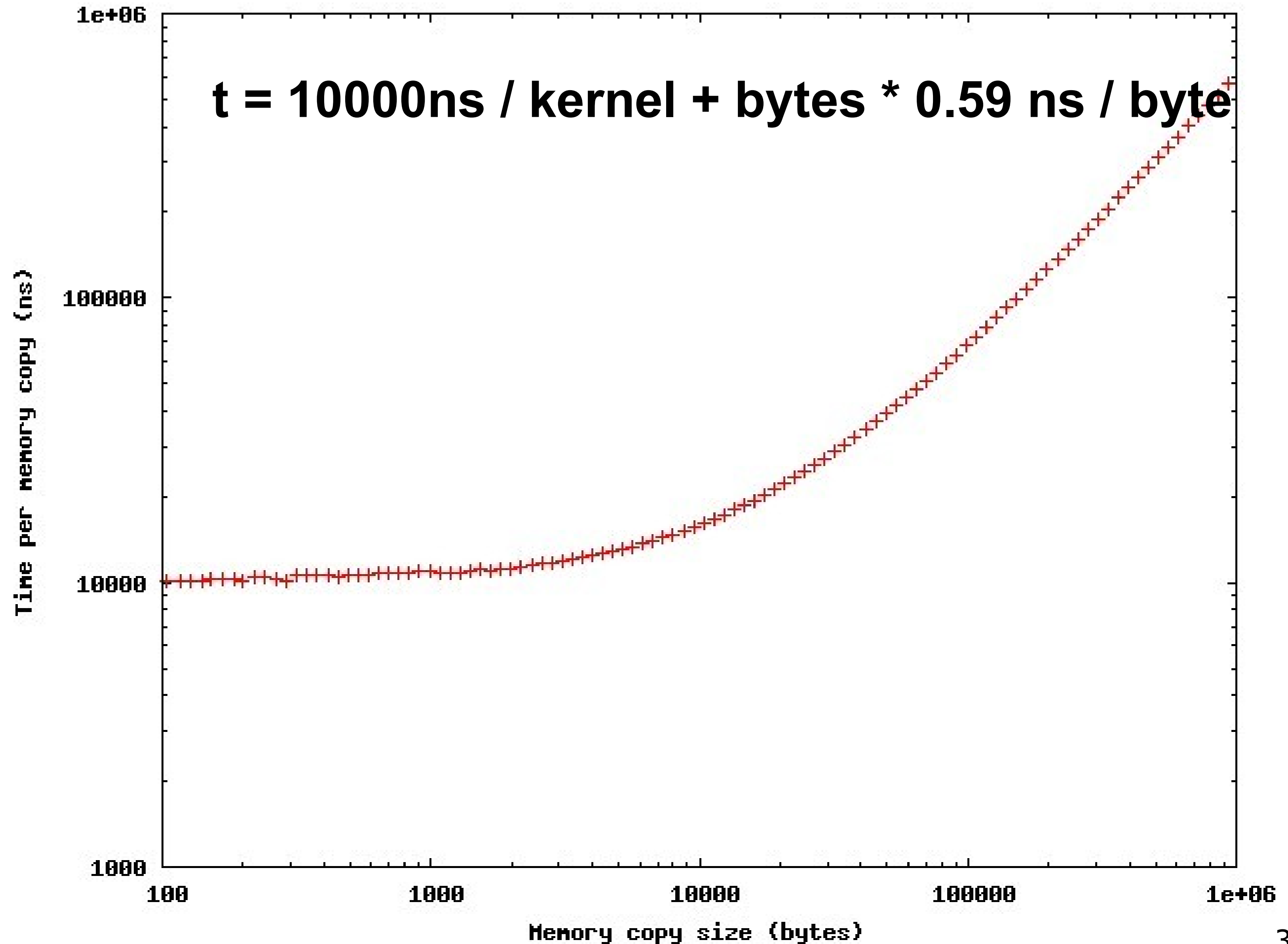
NVIDIA GeForce GTX 280, fixed 128 threads per block

CUDA: Device-to-Host Memcopy



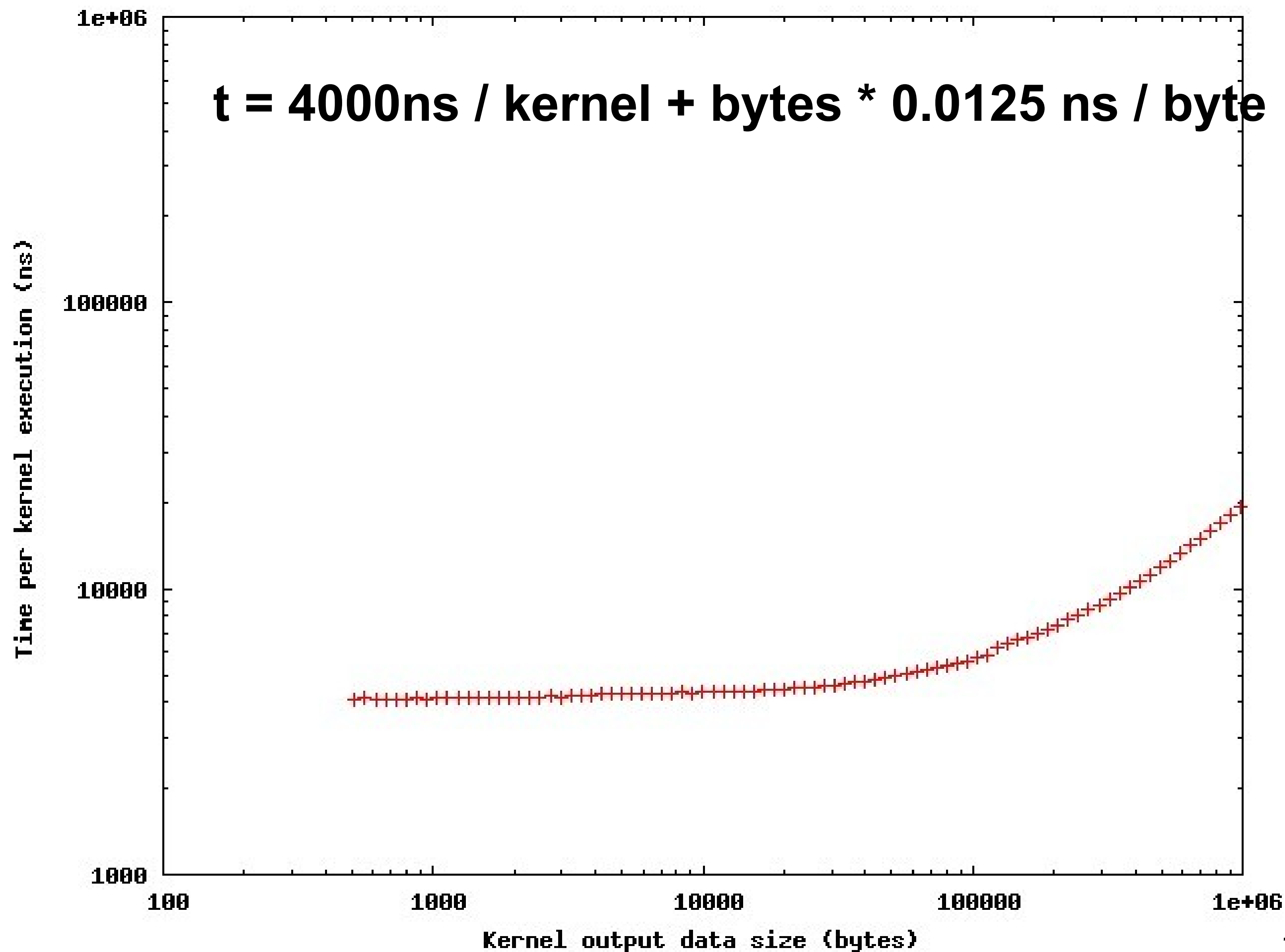
NVIDIA GeForce GTX 280, pinned memory

CUDA: Device-to-Host Memcopy



NVIDIA GeForce GTX 280, pinned memory

CUDA: Memory Output Bandwidth



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**

# 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%

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!**

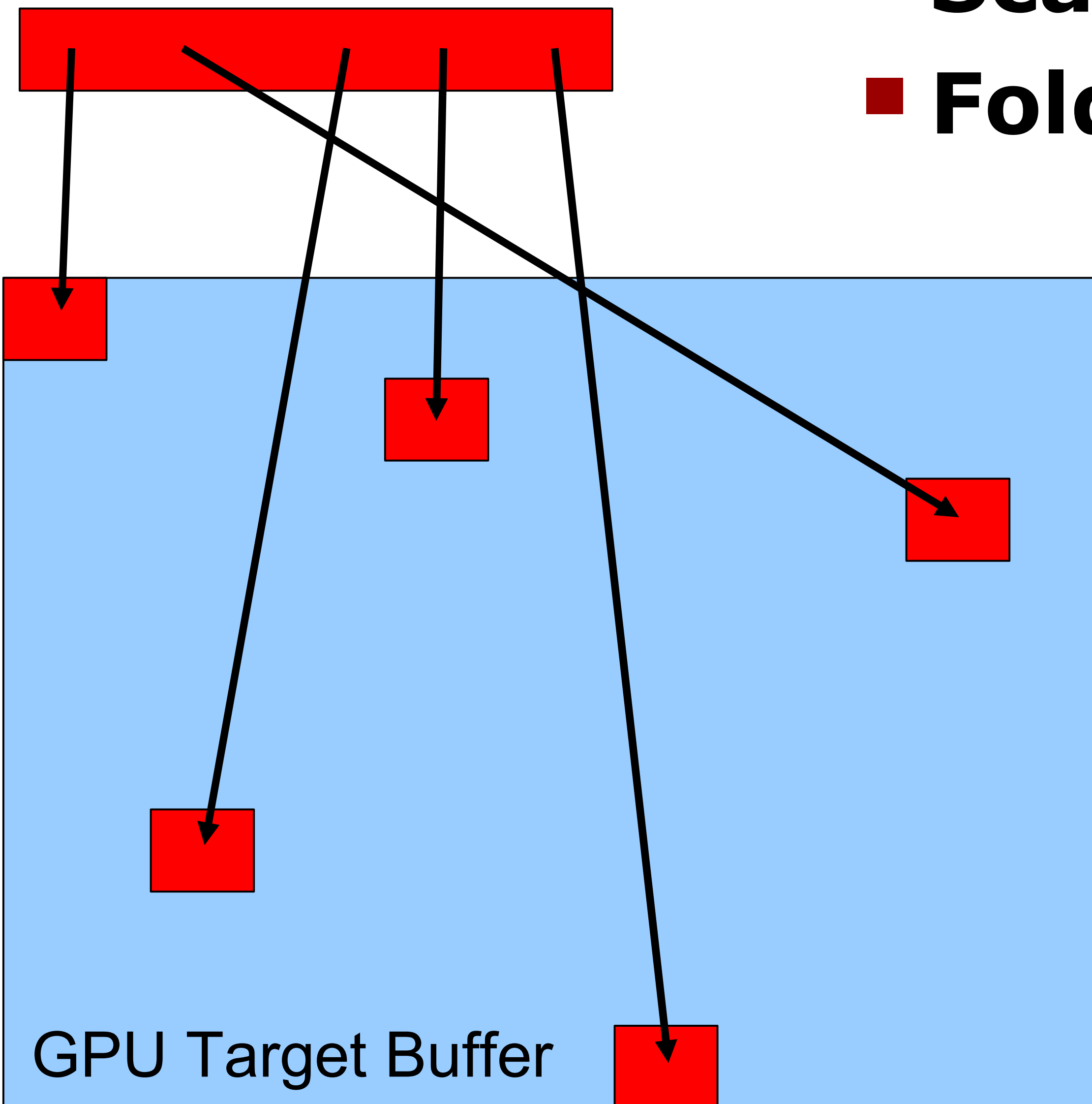
OpenGL Communication

- **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

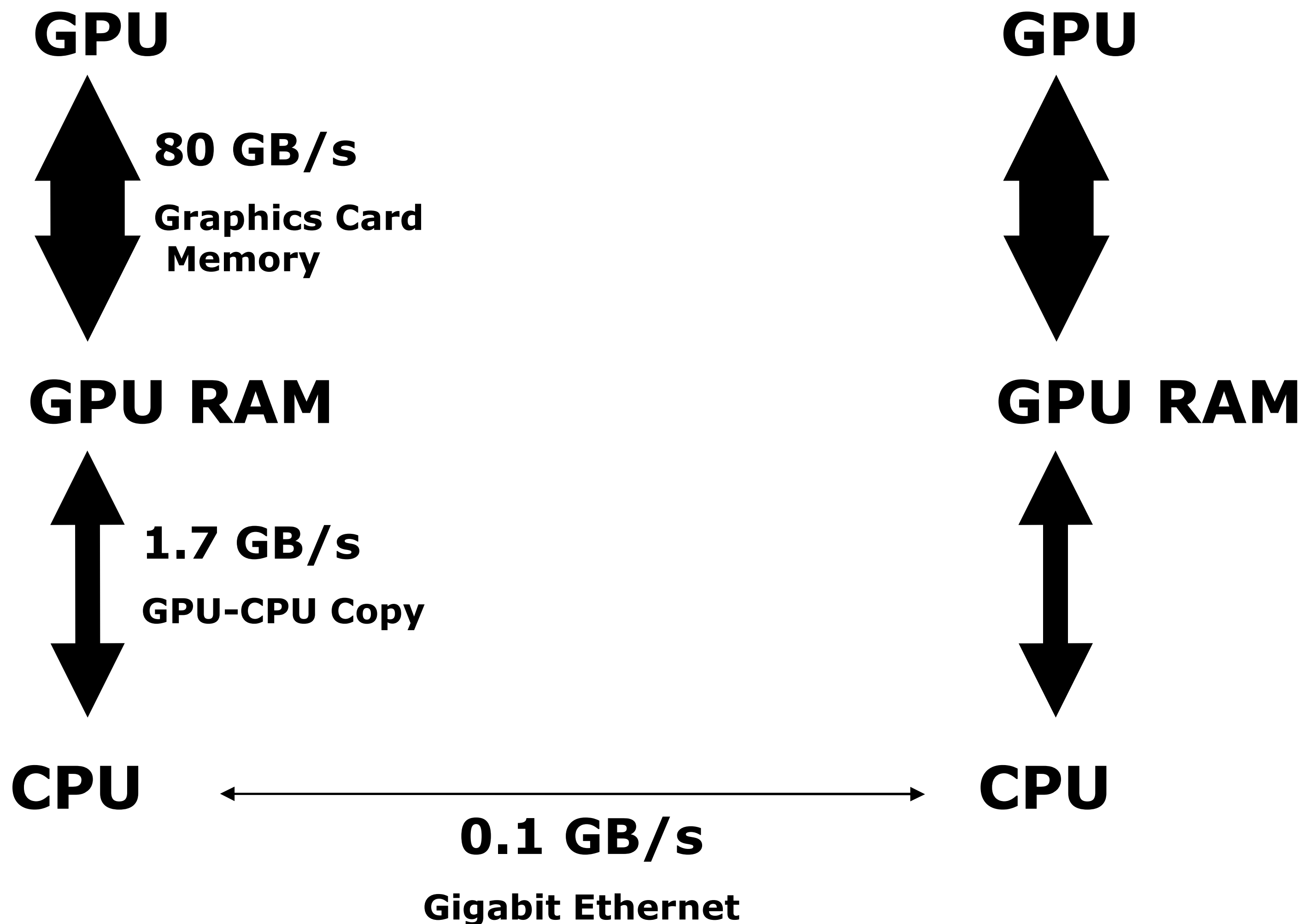


- **Scatter kernel?**
- **Fold into read?**

cudaMPI and glMPI
Performance

Parallel GPU Performance

- **Networks are <u>really</u> low bandwidth!**



Parallel GPU Performance: CUDA

# 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%

Ten GTX 280 GPUs over gigabit Ethernet; CUDA

Parallel GPU Performance: OpenGL

# 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%

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?**

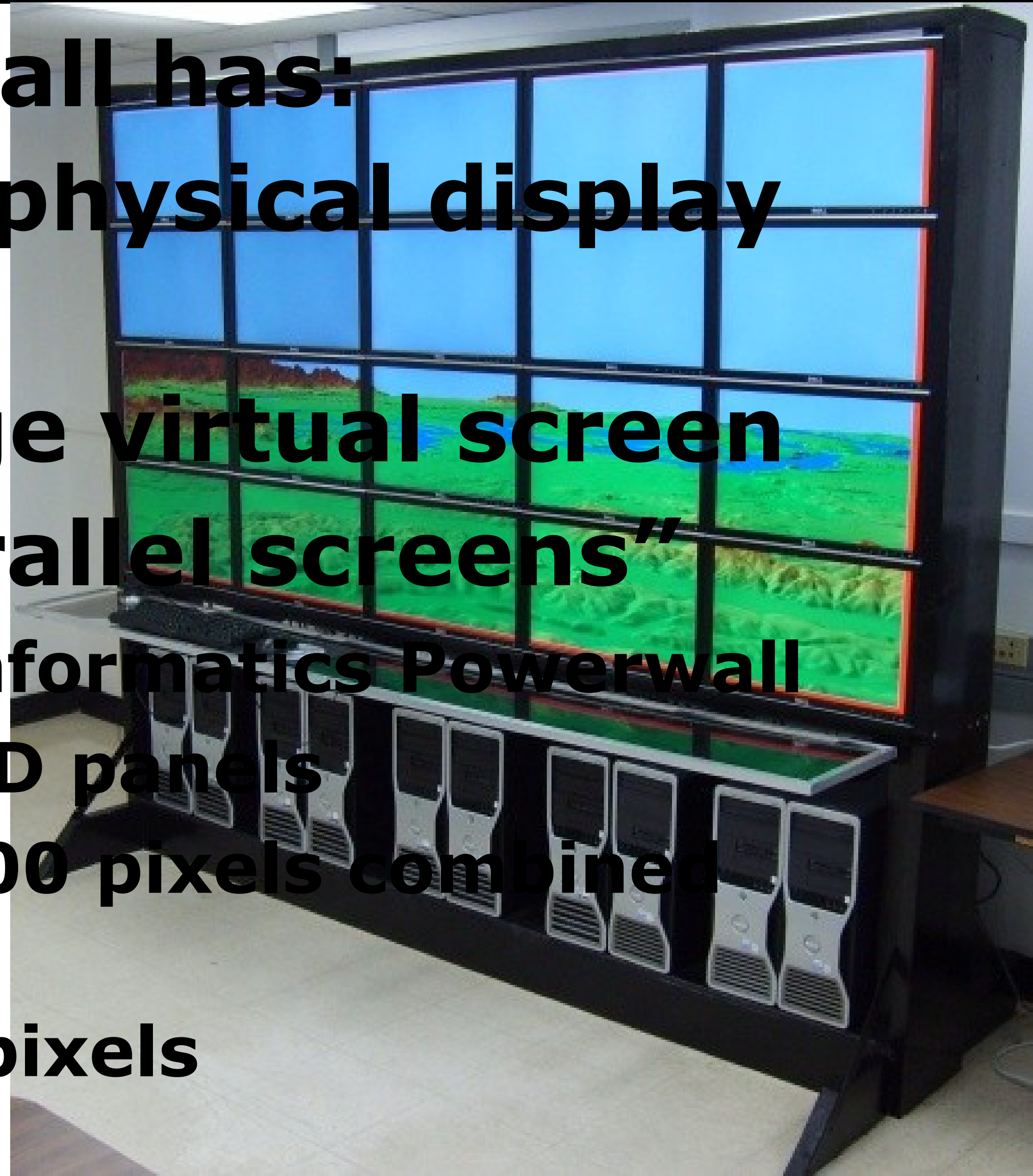
Related work:
MPIglut

MPIglut: Motivation

- **All modern computing is <u>parallel</u>**
  - **Multi-Core CPUs, Clusters**
    - **Athlon 64 X<u>2</u>, Intel Core2 <u>Duo</u>**
  - **Multiple Multi-Unit GPUs**
    - **nVidia SLI, ATI CrossFire**
  - **Multiple Displays, Disks, ...**
- **But languages and many existing applications are <u>sequential</u>**
  - **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



DELL
Backspace

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 <u>intercepts</u> glutInit, glViewport, and broadcasts user events over the network**
- **MPIglut's glViewport shifts to render <u>only</u> the local screen**

MPIglut Conversion: Original Code

```
#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/mpiglut.h>
void display(void) {
   glBegin(GL_TRIANGLES); ... glEnd();
   glutSwapBuffers();
}
void
   glV
   glL
   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

```
#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);
   glLoa
   gluLo
}
...
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

MPIglut Runtime Changes: Events

```
#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);
   glLo
   gluL
}
...
int ma
   glut
   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

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

Frame display is (optionally) synchronized across the cluster

MPIglut Runtime Changes: Coords

```
#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 ma
   glu
   glu
   glutMouseFunc(...);
   ...
}
```

User code works only in <u>global</u> coordinates, but MPIglut adjusts OpenGL's projection matrix to render only the <u>local</u> screen

MPIglut Runtime Non-Changes

```
#include <GL/mpiglut.h>
void display(void) {
   glBegin(GL_TRIANGLES); ... glEnd();
   glutSwapBuffers();
}
void re
   glVi
   glLo
   gluL
}
...
int ma
   glut
   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

MPIglut Assumptions/Limitations

- **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 <u>events</u>**
    - **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

Load Balancing a Powerwall

- **Problem:**

- **Solution: Move the rendering for load balance, but you've got to move the finished pixels back for display!**

Future Work: Load Balancing

- **AMPIglut: 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**