Towards a Usable Programming Model for GPGPU

Dr. Orion Sky Lawlor
lawlor@alaska.edu
U. Alaska Fairbanks

2011-04-19

http://lawlor.cs.uaf.edu/

Obligatory Introductory Quote

“He who controls the past,
controls the future.”

George Orwell, *1984*

In Parallel Programming...

"He who controls the writes,
controls performance."

~~"He who controls the past,~~
~~controls the future."~~

~~George Orwell, *1984*~~

Orion Lawlor, 2011

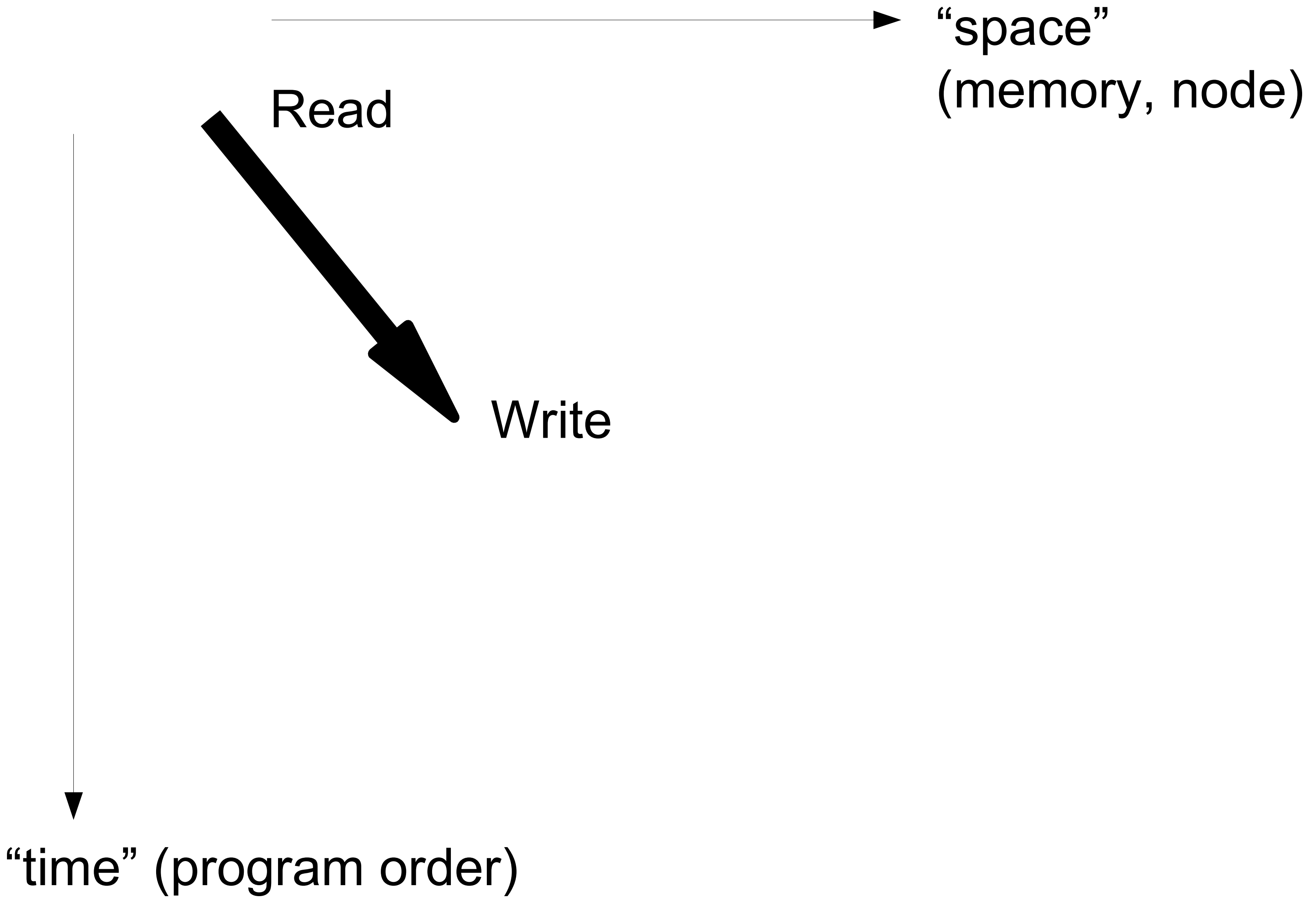
Talk Outline

- Existing parallel programming models
  - Who controls the writes?
- Charm++ and Charm--
  - Charm-style GPGPU
- Conclusions

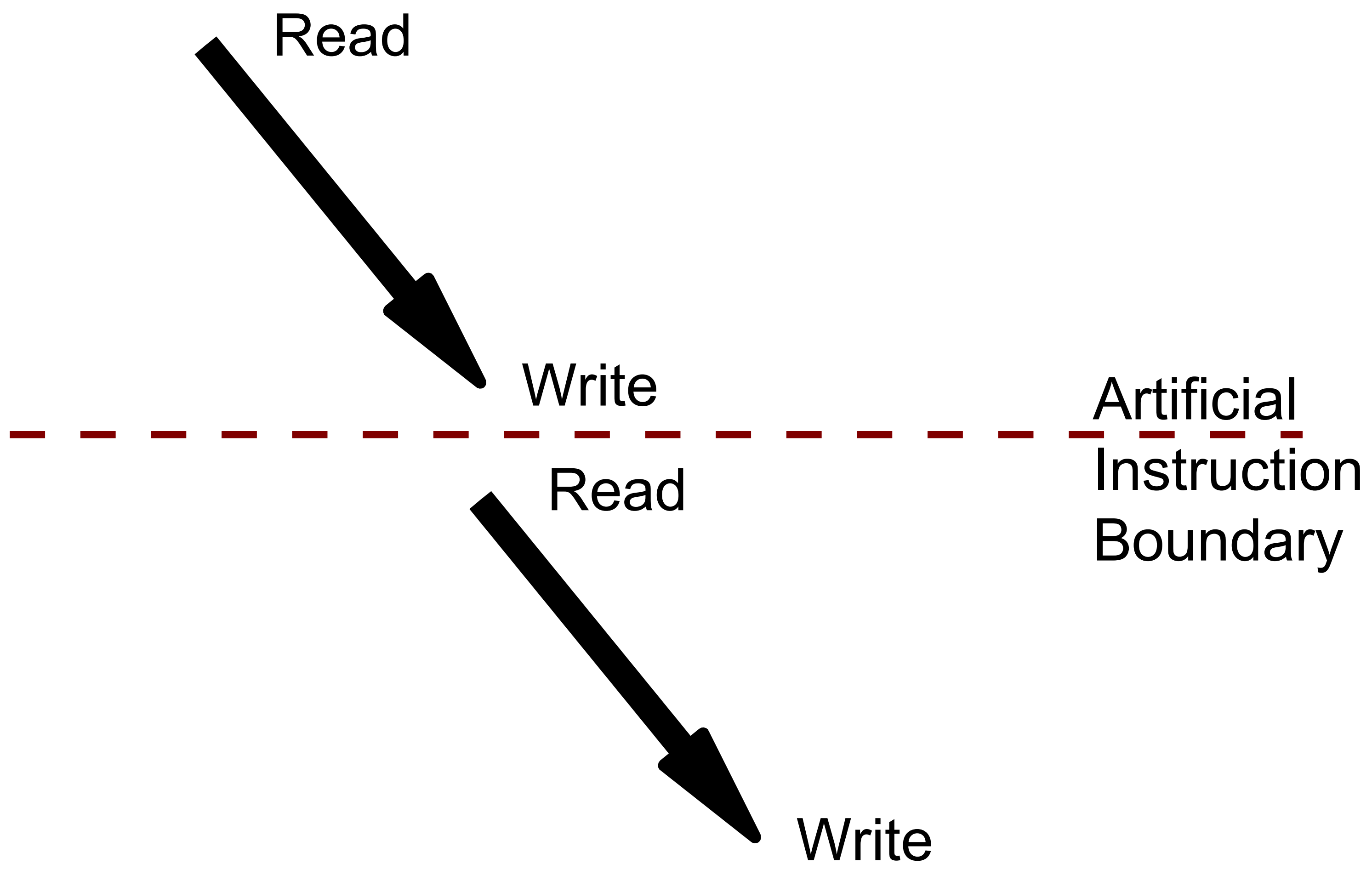
Existing Model: Superscalar

- **Hardware parallelization of a <u>sequential</u> programming model**
- **Fetch future instructions**
  - **Need good branch prediction**
- **Runtime Dependency Analysis**
  - **Load/store buffer for mem-carried**
  - **Rename away false dependencies**
  - **RAR, WAR, WAW, -> RAW <-**
- **Now “solved”: low future gain**

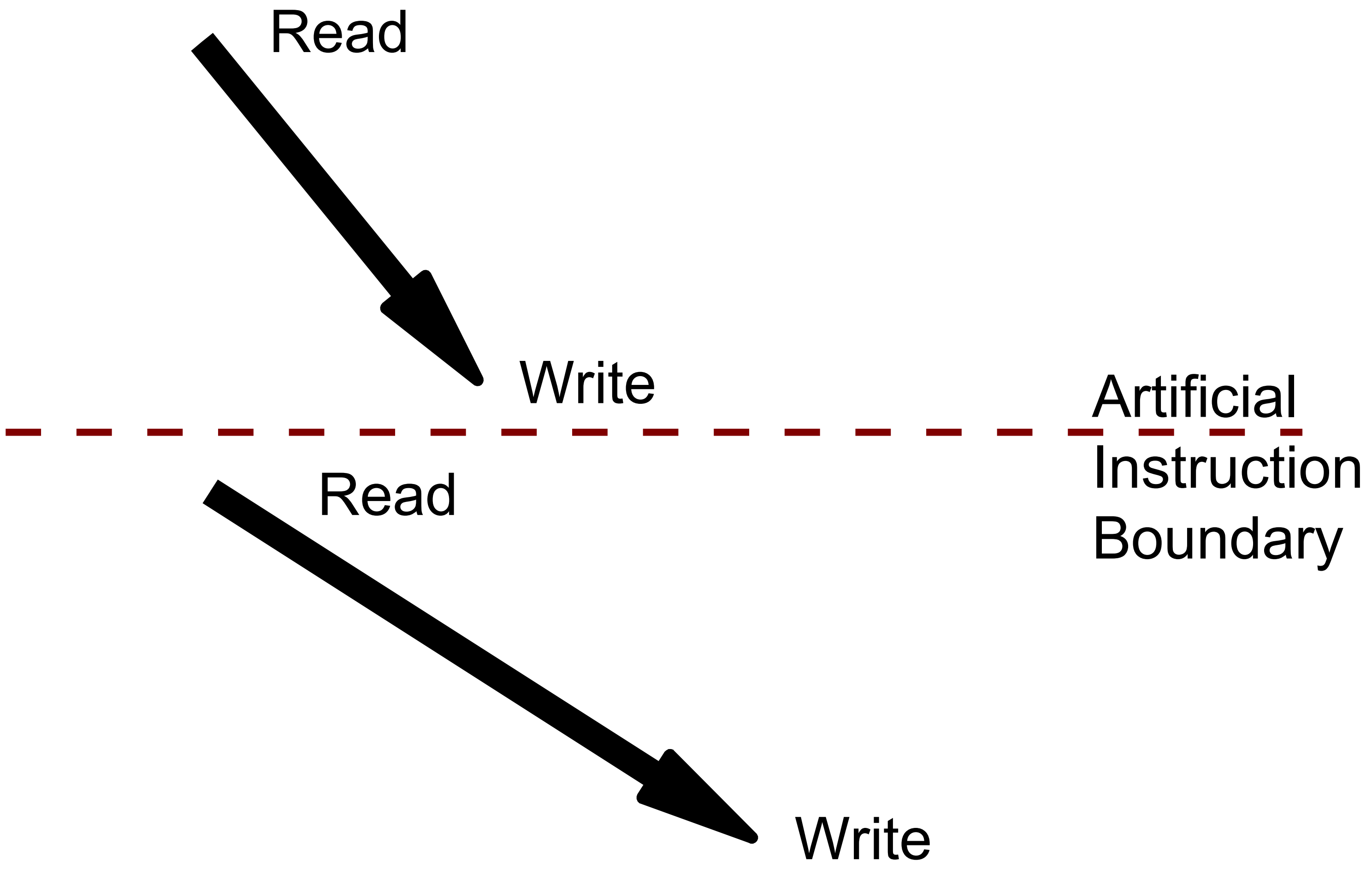
Spacetime Data Arrows



Read After Write Dependency



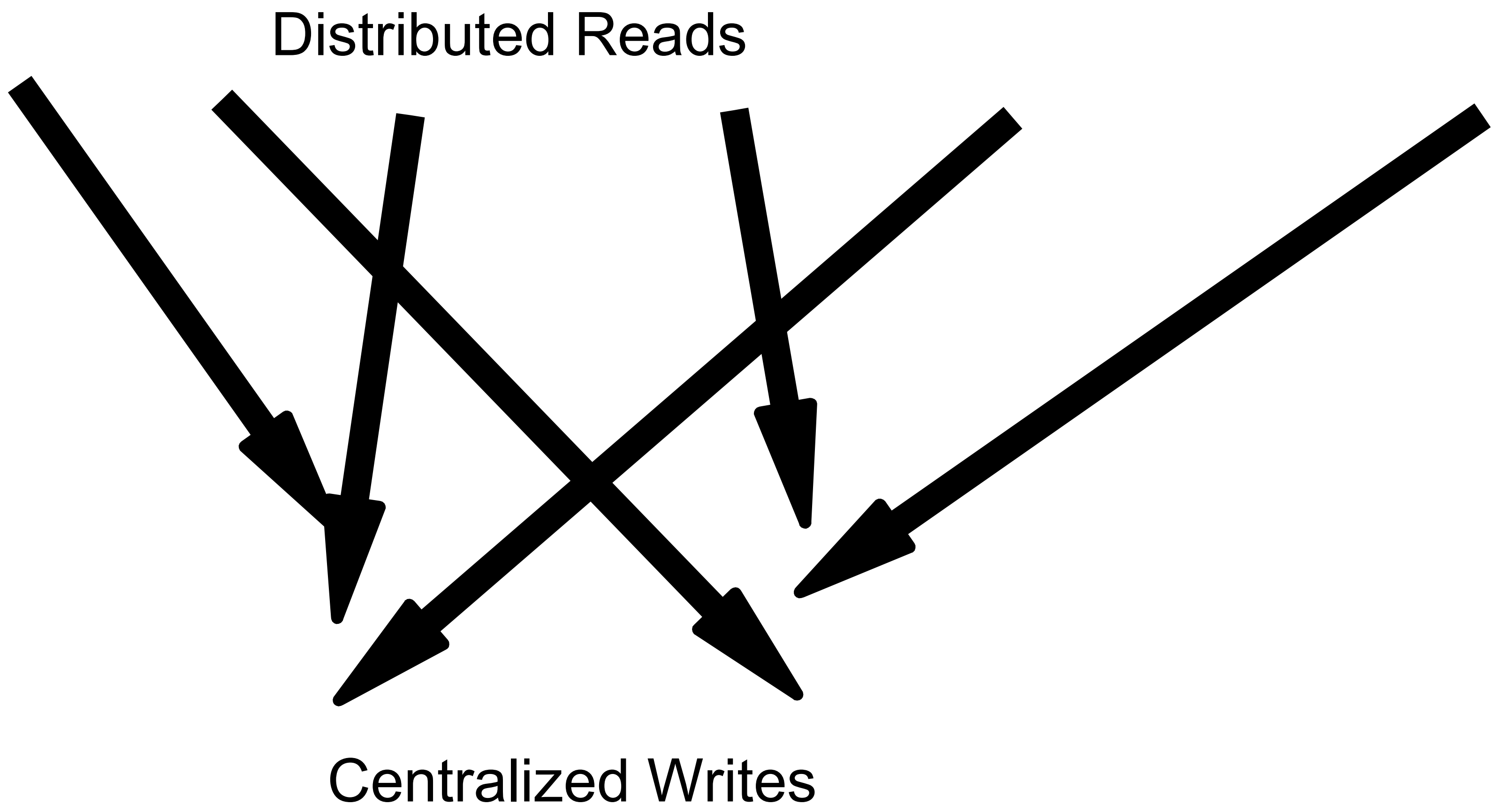
Read After Read: No Problem!



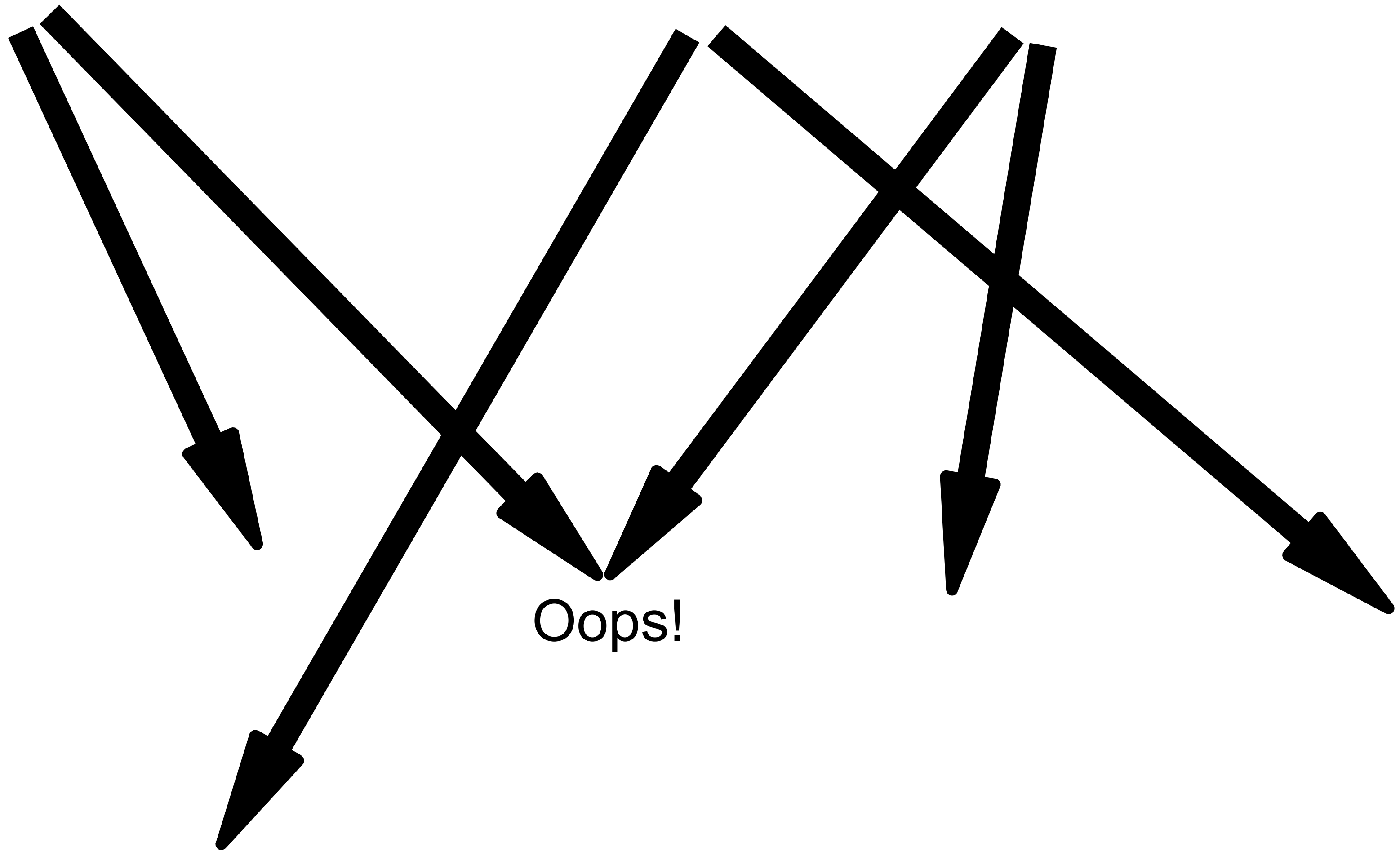
Existing Model: Shared Memory

- OpenMP, threads, shmem
- "Just" let different processors access each others' memory
  - HW: Cache coherence
    - false sharing, cache thrashing
  - SW: Synchronization
    - locks, semaphores, fences, ...
- Correctness is a <u>huge</u> issue
  - Weird race conditions abound
  - New bugs in 10+ year old code

Gather: Works Fine



Scatter: Tough to Synchronize!



Existing Model: Message Passing

- MPI, sockets
- Explicit control of parallel reads (send) and writes (recv)
  - Far fewer race conditions
- Programmability is an issue
  - Raw byte-based interface (C style)
  - High per-message cost (alpha)
  - Synchronization issues: when does MPI_Send block?

Existing Model: SIMD

- SSE, AVX, and GPU
- Single Instruction, Multiple Data
  - Far fewer fetches & decodes
  - Far higher arithmetic intensity
- CPU: Programmability N/A
  - Assembly language (hello, 1984!)
  - mmintrin.h wrappers: _mm_add_ps
  - Or pray for automagic compiler!
- GPU: Programmability OK
  - Graphics side: GLSL, HLSL, Cg
  - GPGPU: CUDA, OpenCL, DX CS

NVIDIA CUDA

- CPU calls a GPU “kernel” with a “block” of threads
  - Now fully programmable (throw/catch, virtual methods, recursion, etc)
- Read and write memory anywhere
  - Zero protection against multithreaded race conditions
- Manual control over a small __shared__ memory region
- Only runs on NVIDIA hardware (OpenCL is portable... sorta)

OpenGL: GL Shading Language

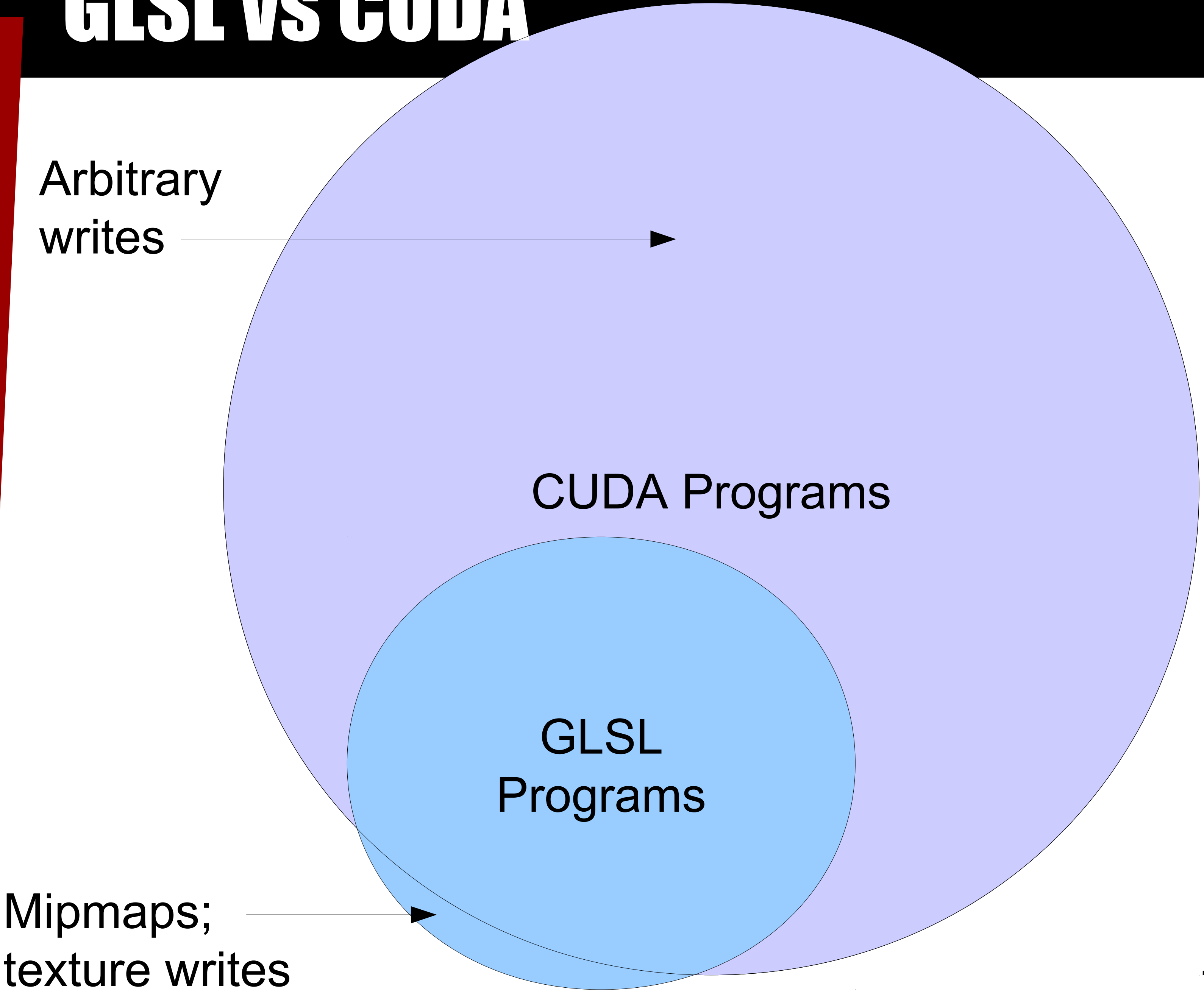
- Mostly programmable (loops, etc)
- Can read anywhere in “textures”, only write to “framebuffer” (2D/3D arrays)
  - Reads go through “texture cache”, so performance is good (iff locality)
  - Writes are on space-filling curve
  - Writes are controlled by the graphics driver
  - So <u>cannot</u> have synchronization bugs!
- Rich selection of texture filtering (array interpolation) modes
  - Includes mipmaps, for multigrid
- GLSL can run OK on every modern GPU

  (well, except Intel…)

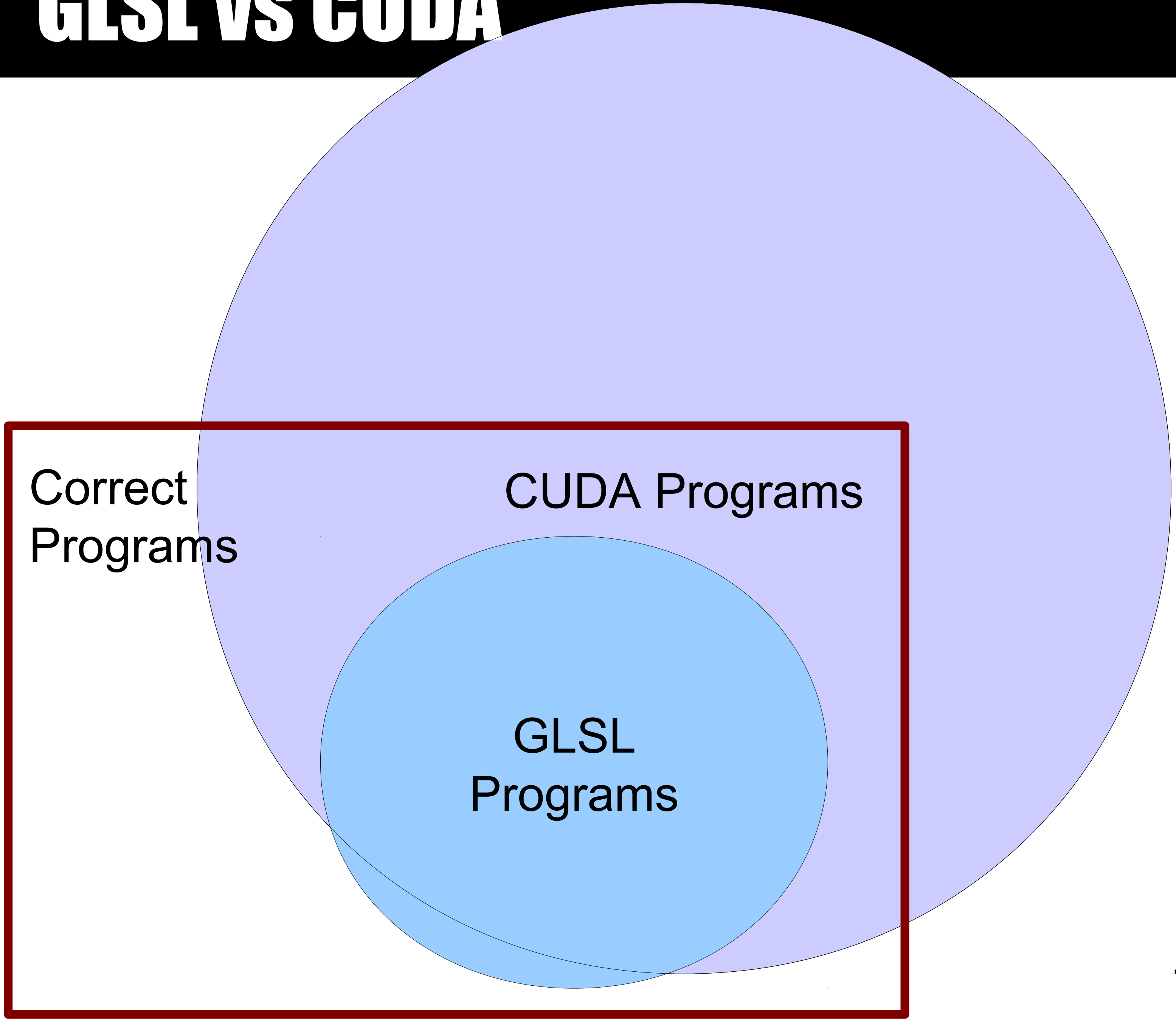
GLSL vs CUDA

GLSL
Programs

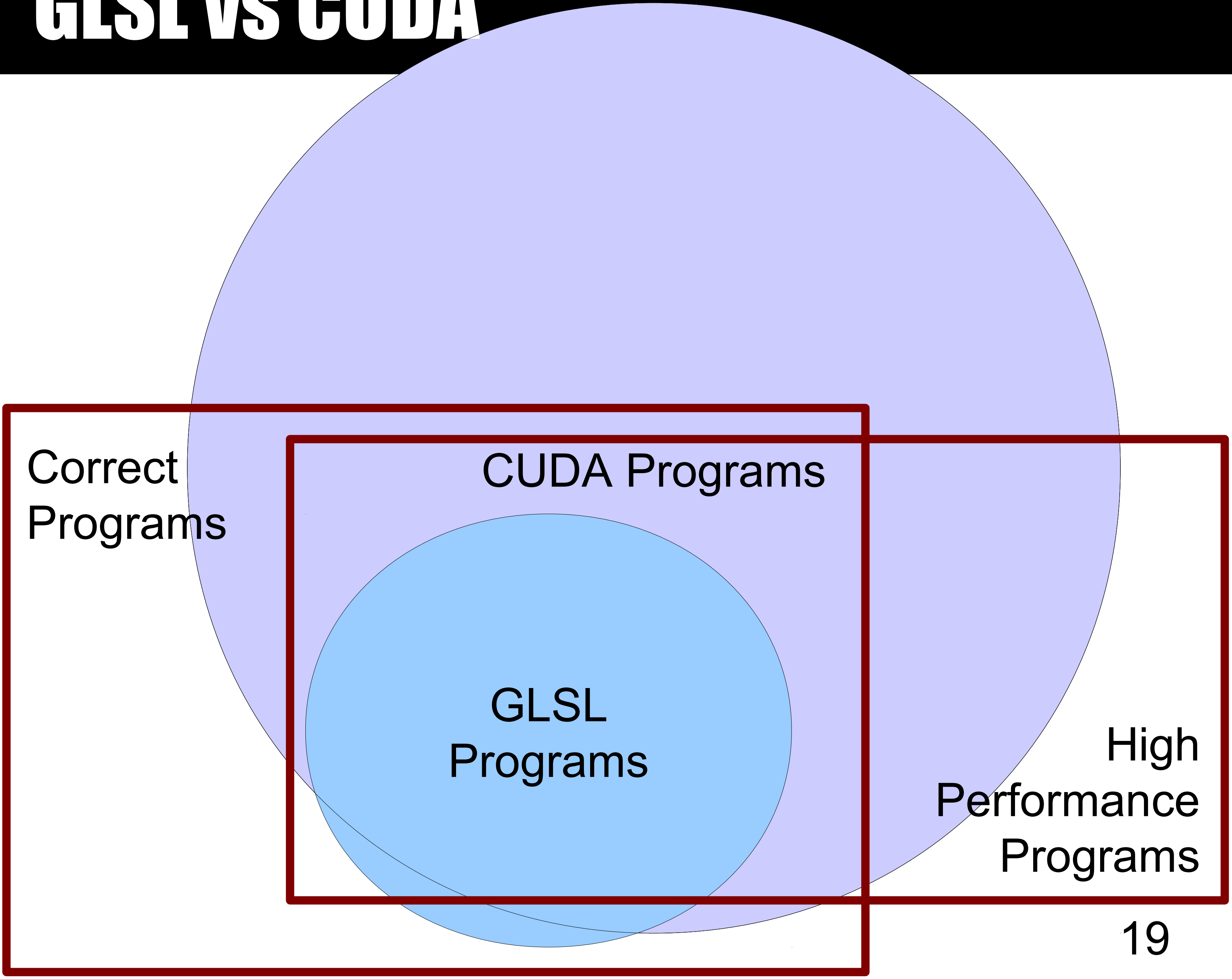
GLSL vs CUDA


Arbitrary writes
CUDA Programs
GLSL Programs
Mipmaps; texture writes

GLSL vs CUDA



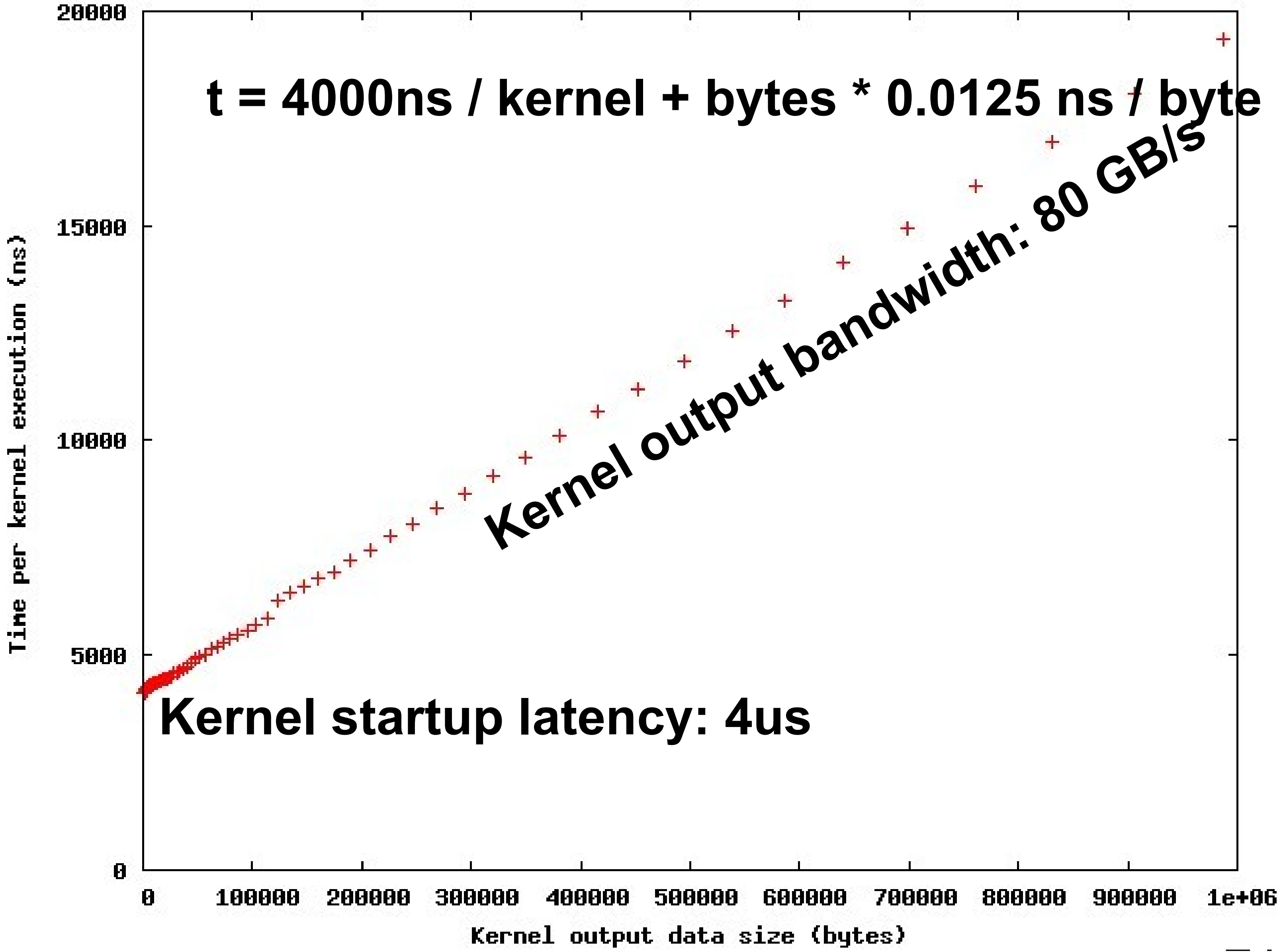
GLSL vs CUDA


Correct Programs
CUDA Programs
GLSL Programs
High Performance Programs

GPU/CPU Convergence

- GPU, per socket:
  - SIMD: 16-32 way
  - SMT: 2-50 way (register limited)
  - SMP: 4-36 way
- CPUs will get there, soon!
  - SIMD: 8 way AVX (64-way SWAR)
  - SMT: 2 way Intel; 4 way IBM
  - SMP: 6-8 way/socket already
    - Intel has shown 48 way chips
- Biggest difference: CPU has branch prediction & superscalar!

CUDA: Memory Output Bandwidth



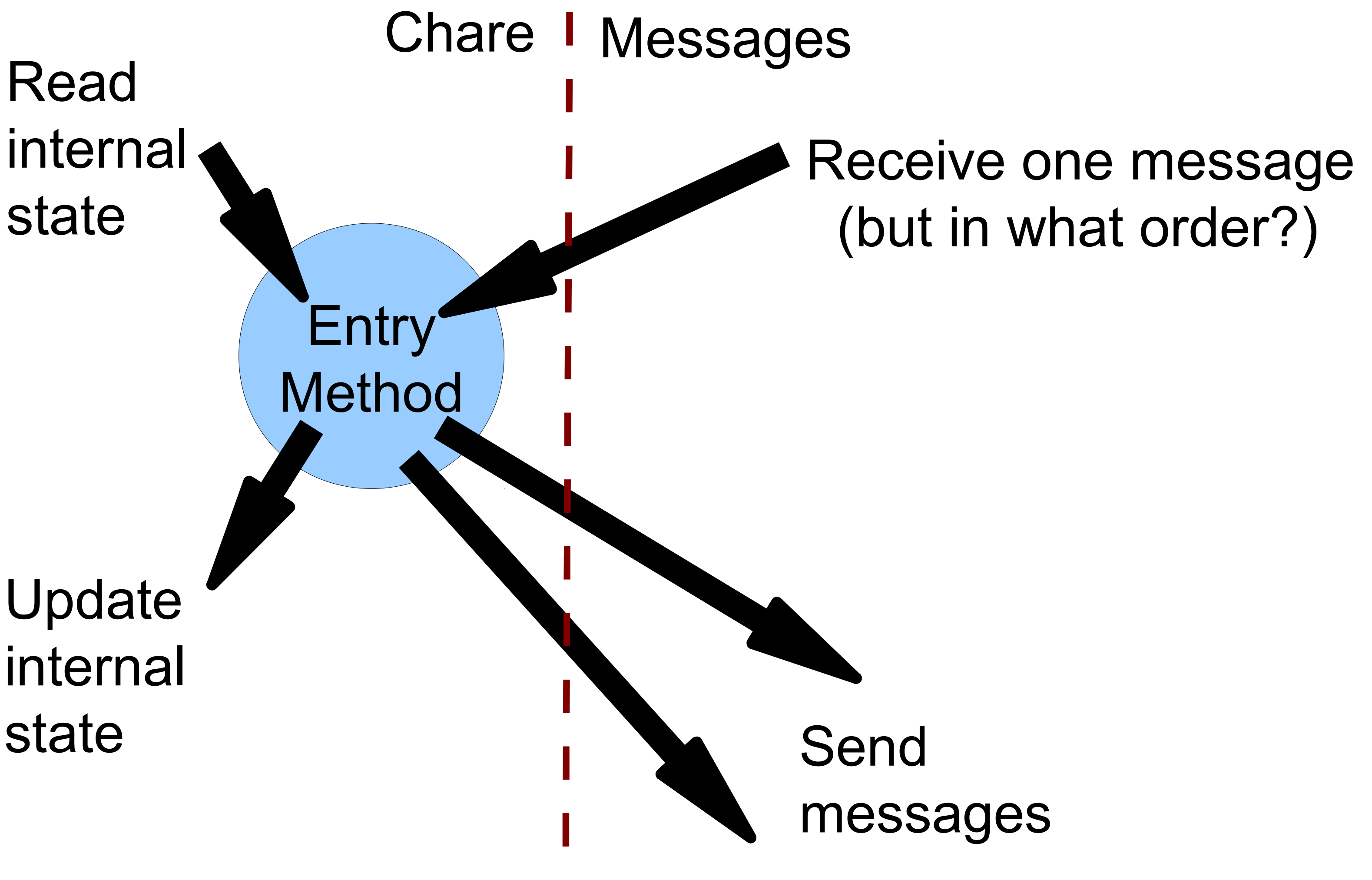
NVIDIA GeForce GTX 280, fixed 128 threads per block

Charm++ and “Charm--”

Existing Model: Charm++

- Chares send each other messages
- Runtime system does delivery
  - <u>Scheduling!</u>
  - Migration with efficient forwarding
  - Cheap broadcasts
- Runtime system schedules Chares
  - Overlap comm and compute
- Programmability still an issue
  - Per-message overhead, even with message combining library
  - Collect up your messages (SDAG?)
  - Cheap SMP reads? SIMD? GPU?

One Charm++ Method Invocation



Between send and receive: migration, checkpointing, ...

The Future: SIMD

- AVX, SSE, AltiVec, GPU, etc
- Thought experiment
  - Imagine a block of 8 chares living in one SIMD register
    - Deliver 8 messages at once (!)
  - Or imagine 100K chares living in GPU RAM
- Locality (mapping) is important!
  - Branch divergence penalty
  - Struct-of-Arrays member storage
    - xxxxxxxx yyyyyyyy zzzzzzzz
    - Members of 8 separate chares!

Vision: Charm-- Stencil

```
array [2D] stencil {
public:
   float data;
   [entry] void average(
        float nbors[4]=fetchnbors())
   {
     data=0.25*( nbors[0]+
        nbors[1]+            nbors[2]+
                    nbors[3]);
   }
};
```

Vision: Charm-- Explained

```
array [2D] stencil {
public:
   float data;
   [entry] void average(
        float nbors[4]=fetchnbors())
   {
     data=0.25*( nbors[0]+
        nbors[1]+            nbors[2]+
                    nbors[3]);
   }
};
```

Assembled into
GPU arrays or SSE vectors

Vision: Charm-- Explained

```
array [2D] stencil {
public:
   float data;
   [entry] void average(
        float nbors[4]=fetchnbors())
   {
     data=0.25*( nbors[0]+
         nbors[1]+            nbors[2]+
                      nbors[3]);
   }
};
```

Broadcast out to blocks of array elements

Vision: Charm-- Explained

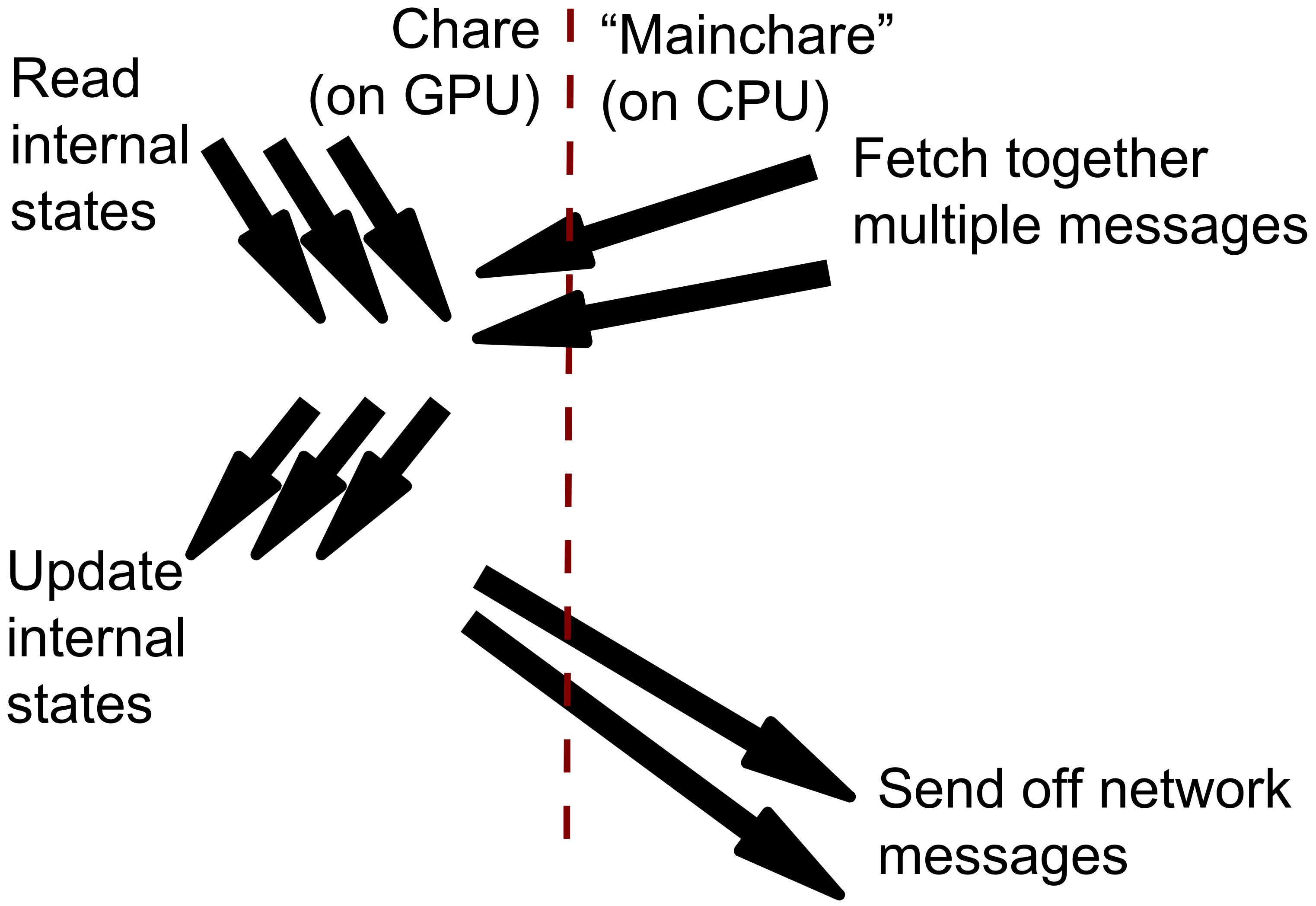
```
array [2D] stencil {
public:
  float data;
  [entry] void average(
      float nbors[4]=fetchnbors())
  {
    data=0.25*( nbors[0]+
      nbors[1]+          nbors[2]+
                nbors[3]);
  }
};
```

Hides local synchronized reads, network, and domain boundaries

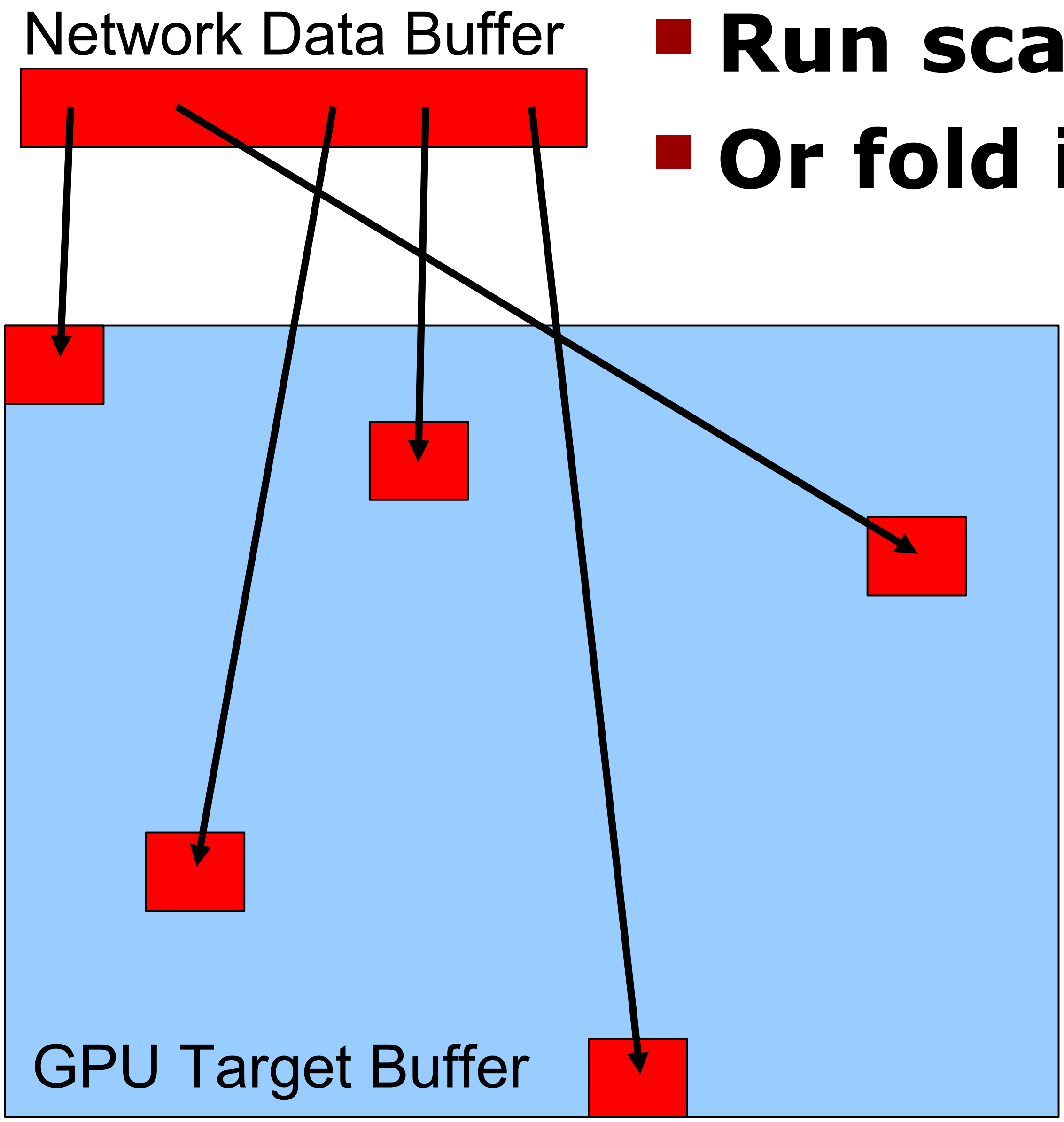
Vision: Charm-- Springs

```
array [1D] sim_spring {
public:
  float restlength;
  [entry] void netforce(
       sim_vertex ends[2]=fetch_ends())
  {
     vec3 along=ends[1].pos-ends[0].pos;
     float f=-k*(length(along)-restlength);
     vec3 F=f*normalize(along);
     ends[0].netforce+=F;
     ends[1].netforce-=F;
  }
};
```

One Charm-- Method Invocation



Noncontiguous Communication



- **Run scatter kernel**
- **Or fold into fetch**

Key Charm-- Design Features

- Multiple chares receive message at once
  - Runtime block-allocates incoming and outgoing message storage
  - Critical for SIMD, GPU, SMP
- Receive multiple messages in one entry point
  - Minimize roundtrip to GPU
- Explicit support for timesteps
  - E.g., double-buffer message storage

Charm-- Not Shown

- Lots of work in “mainchare”
  - Controls decomposition & comms
  - Set up “fetch”
- Still lots of network work
  - Gather & send off messages
  - Distribute incoming messages
- Division of labor?
  - Application scientist writes Chare
  - Computer scientist writes Mainchare

Related Work

- Charm++ Accelerator API [Wesolowski]
  - Pipeline CUDA copy, queue kernels
  - Good backend for Charm--
- Intel ArBB: SIMD from kernel
  - Based on RapidMind
  - But GPU support?
- My "GPGPU" library
  - Based on GLSL

The Future

The Future: Memory Bandwidth

- Today: 1TF/s, but only 0.1TB/s
- Don't communicate, <u>recompute</u>
  - multistep stencil methods
  - “fetch” gets even more complex!
- 64-bit -> 32-bit -> 16-bit -> 8?
  - Spend flops scaling the data
  - Split solution + residual storage
    - Most flops use fewer bits, in residual
  - Fight roundoff with stochastic rounding
    - Add noise to improve precision

Conclusions

- C++ is dead. Long live C++!
- CPU and GPU on collision course
  - SIMD+SMT+SMP+network
- Software is the bottleneck
  - Exciting time to build software!
- Charm-- model
  - Support ultra-low grainsize chares
    - Combine into SIMD blocks at runtime
  - Simplify programmer's life
  - Add flexibility for runtime system
  - BUT must scale to real applications!