GPUs in Supercomputing: Challenges and Opportunities

Matthew R. Norman

Scientific Computing Group
National Center for Atmospheric Research
Oak Ridge National Laboratory

2013 HPC Workshop, DOE CSGF Annual Program Review

Oak Ridge National Laboratory

OLCF
Oak Ridge Leadership Computing Facility

Outline

- **Motivation and data movement**
- Basic GPU Architecture
- Basic Programming in CUDA
- Other Programming Models
- Challenges
- Opportunities

Why Use GPUs?

- More performance, using less power
  - GPUs are one approach to addressing this
- Example: Jaguar XT5
  - 1.8 Petaflops used 7 Megawatts of power
  - That about 5,400 houses worth of electricity (2011 data)
- Titan XK7 (Nvidia K20x GPUs)
  - 18 Petaflops uses 8 Megawatts of power
  - Near order of magnitude improvement in power efficiency
- Linpack benchmark isn't a great proxy for scientific application performance

Massively Parallel Computers

- Architectured in a hierarchy

Massively Parallel Computers

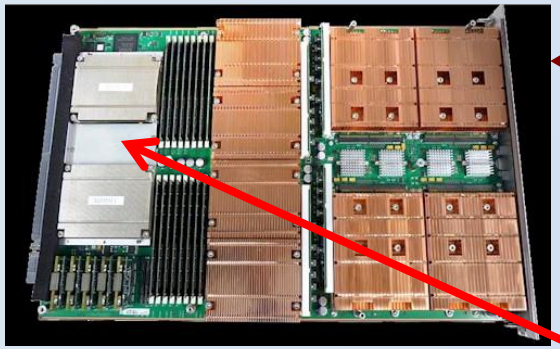
- Architectured in a hierarchy
- Collection of cabinets

Massively Parallel Computers

- Architectured in a hierarchy
- Collection of cabinets
- Each cabinet contains “blades”

Massively Parallel Computers

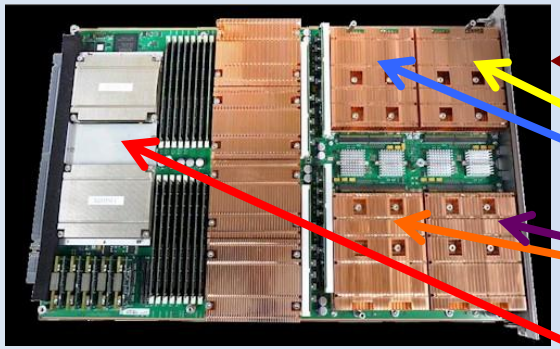
- Architectured in a hierarchy
- Collection of cabinets
- Each cabinet contains “blades”



- A blade has network interconnect

Massively Parallel Computers

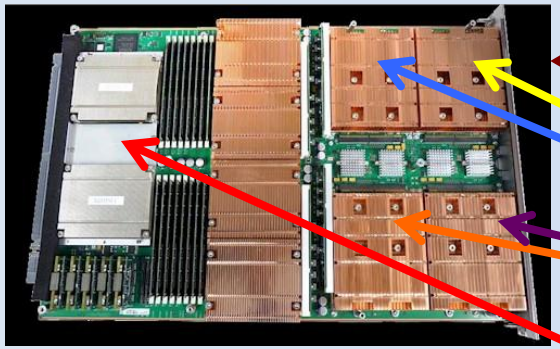
- Architectured in a hierarchy
- Collection of cabinets
- Each cabinet contains “blades”



- A blade has network interconnect and “nodes”

Massively Parallel Computers

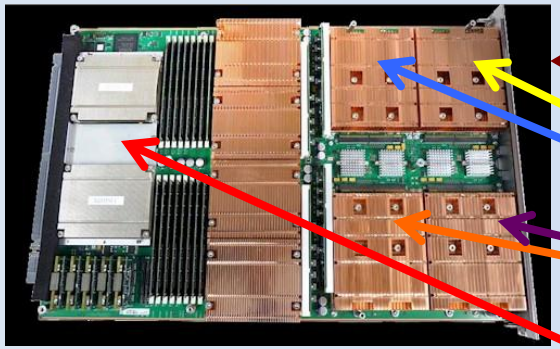
- Architectured in a hierarchy
- Collection of cabinets
- Each cabinet contains “blades”



- A blade has network interconnect and “nodes”
  - Each node has its own processors and memory (DRAM)

Massively Parallel Computers

- Architectured in a hierarchy
- Collection of cabinets
- Each cabinet contains “blades”



- A blade has network interconnect and “nodes”
  - Each node has its own processors and memory (DRAM)
- Nodes share data over fast, specialized networks

The Primary Difficulty Of Computing

- Peak Performance: Fictitious “perfect” world
  - [Cycles per second per core] * [FP Instructions per cycle] * [cores per processor] * [processors per node] * [# nodes]
  - Often has little bearing on science

The Primary Difficulty Of Computing

- Peak Performance: Fictitious “perfect” world
  - [Cycles per second per core] * [FP Instructions per cycle] * [cores per processor] * [processors per node] * [# nodes]
  - Often has little bearing on science
- Why?
  - Nothing * Nothing = Nothing (Don’t get too excited)

The Primary Difficulty Of Computing

- Peak Performance: Fictitious “perfect” world
  - [Cycles per second per core] * [FP Instructions per cycle] * [cores per processor] * [processors per node] * [# nodes]
  - Often has little bearing on science
- Why?
  - Nothing * Nothing = Nothing (Don’t get too excited)
  - Processors need useful data
  - Data movement significantly slower than processing

Two Concepts For Data Transfer

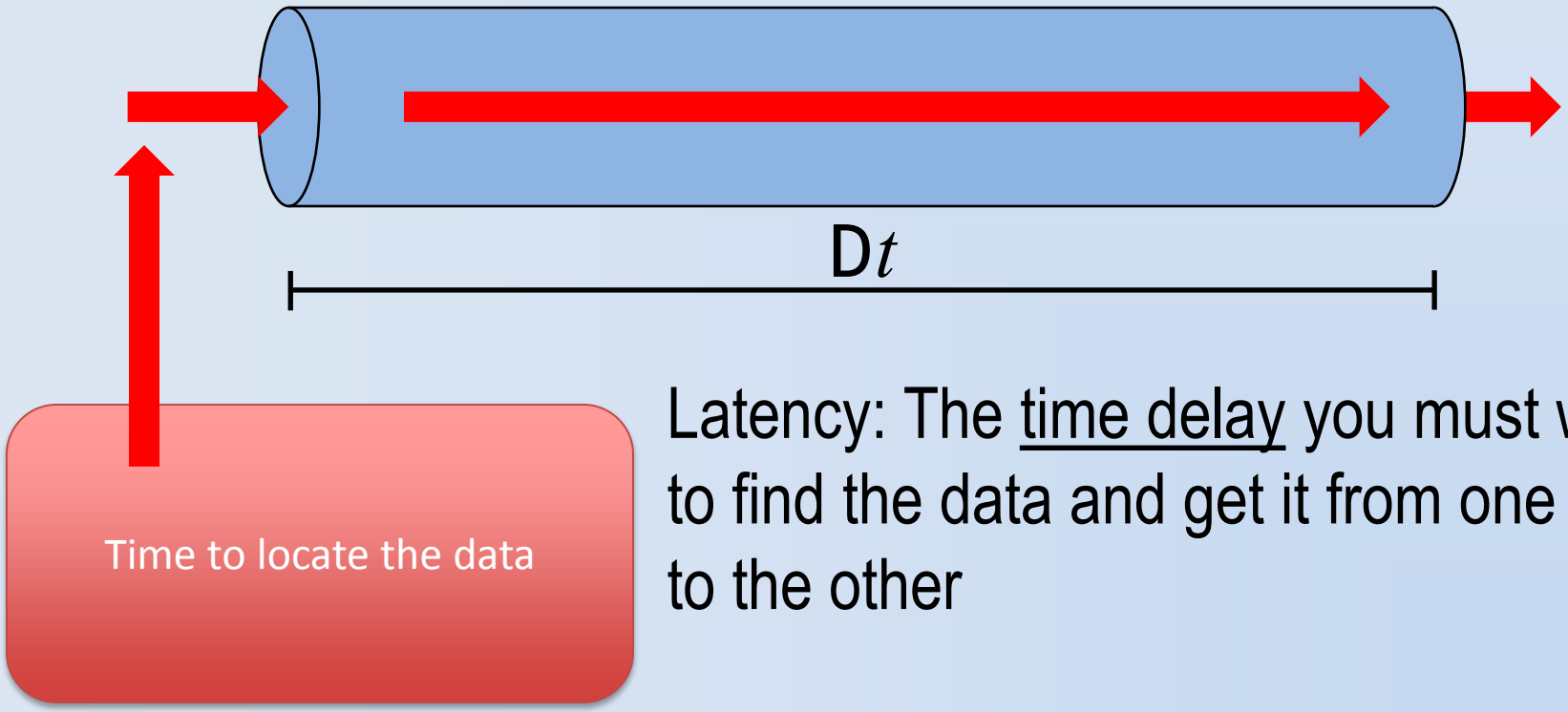
Data is transferred over wires

Two main properties: Latency & Bandwidth

Two Concepts For Data Transfer

Data is transferred over wires

Two main properties: Latency & Bandwidth



Latency: The <u>time delay</u> you must wait to find the data and get it from one end to the other

OLCF
Oak Ridge Leadership Computing Facility

Two Concepts For Data Transfer

Data is transferred over wires

Two main properties: Latency & Bandwidth

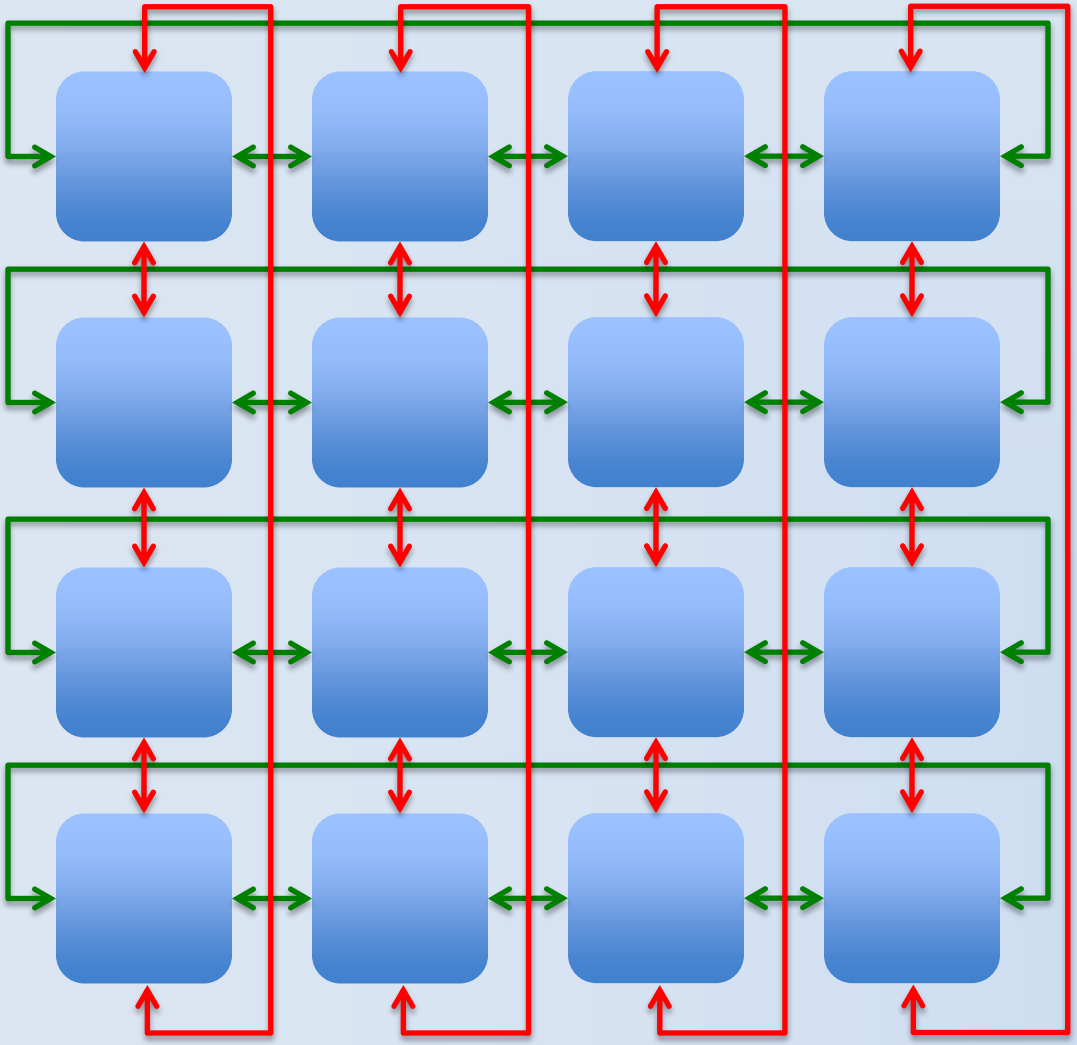
Bandwidth: The <u>amount</u> of data you can pass through <u>per second</u>

Parallel Disk I/O

- Slowest component of data transfer
  - Magnitude of disk I/O is application dependent
- Needed for initialization, history files, restart files
- Needed for analysis and post-processing

Nodes Communicating Over A Network

You will need data from other nodes



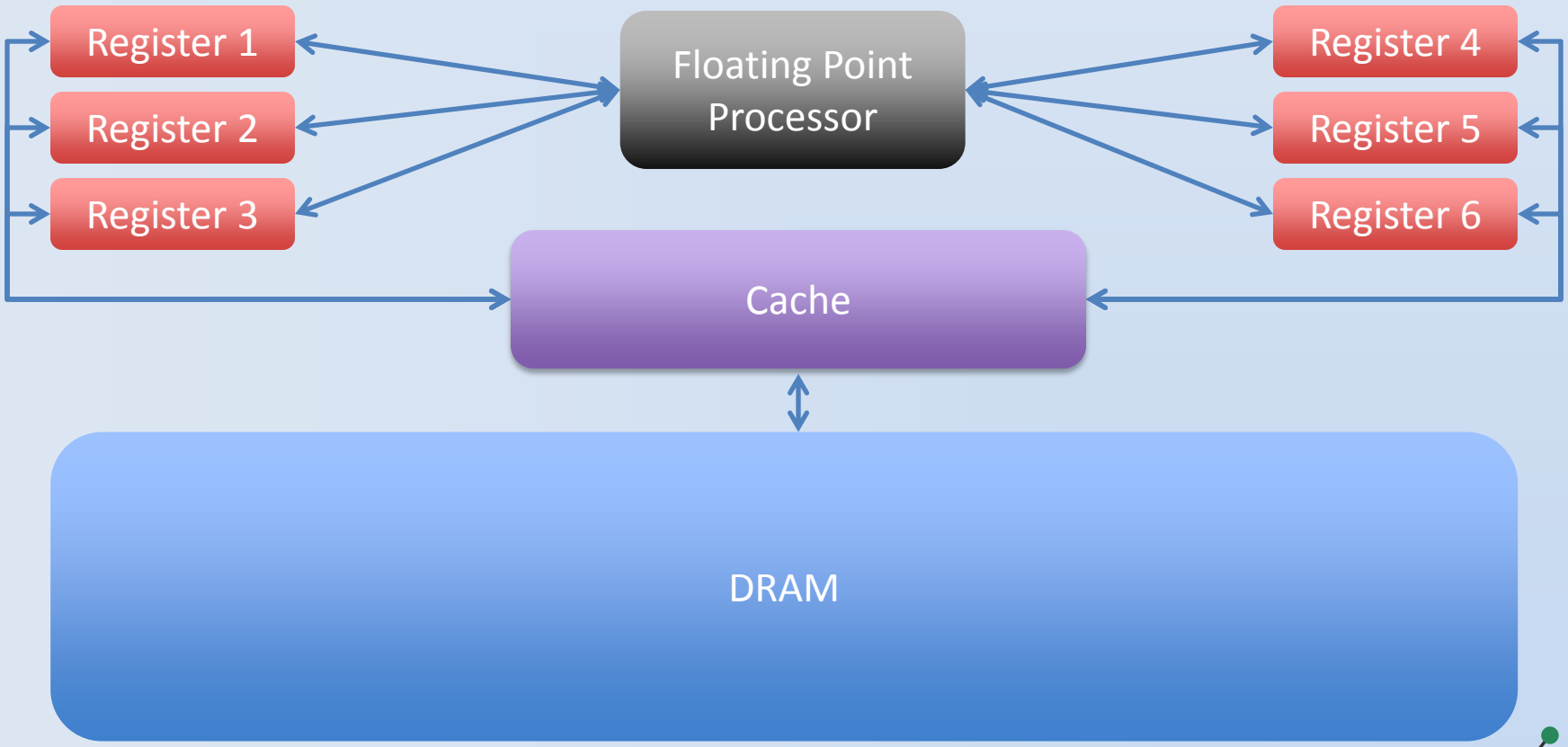
This is the second slowest mode of data transfer

Done with MPI or global arrays

Different modes of transfer: all-to-all, point-to-point, reductions, etc.

DRAM and Caching

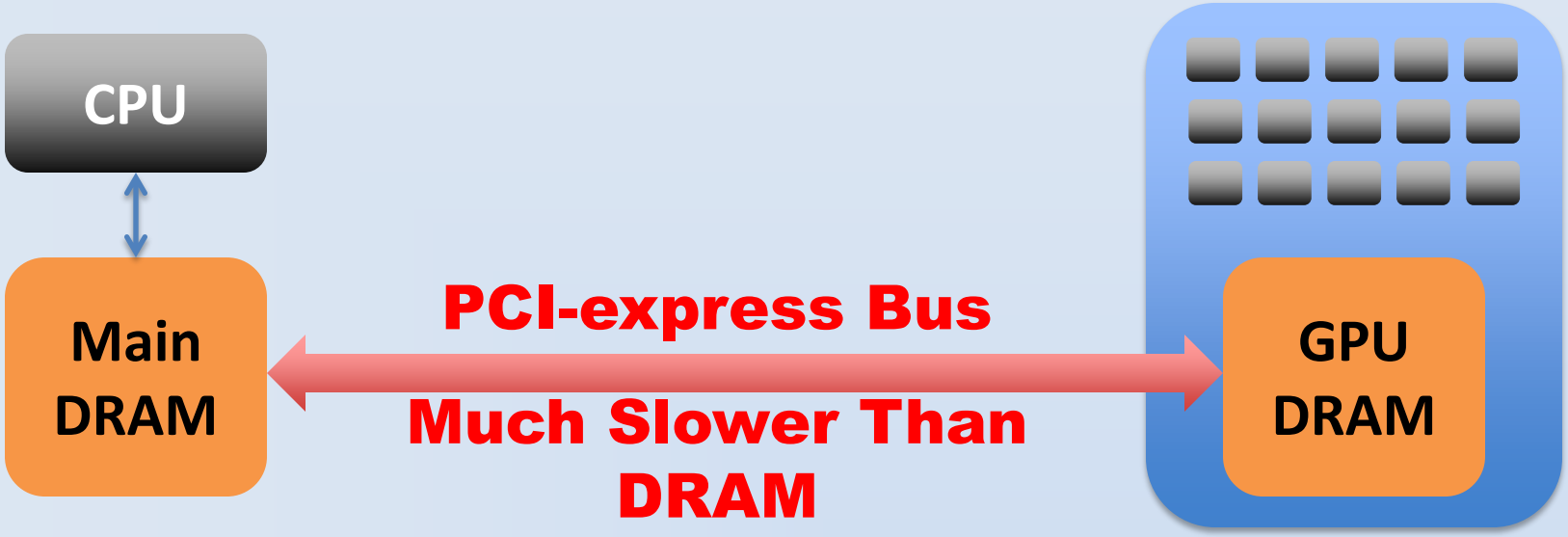
Problem: DRAM latency is very high, bandwidth ≈100x too slow
Solution: Smaller size, lower latency, higher bandwidth "cache"



Outline

- Motivation and data movement
- **Basic GPU Architecture**
- Basic Programming in CUDA
- Other Programming Models
- Challenges
- Opportunities

The PCI-e Bus



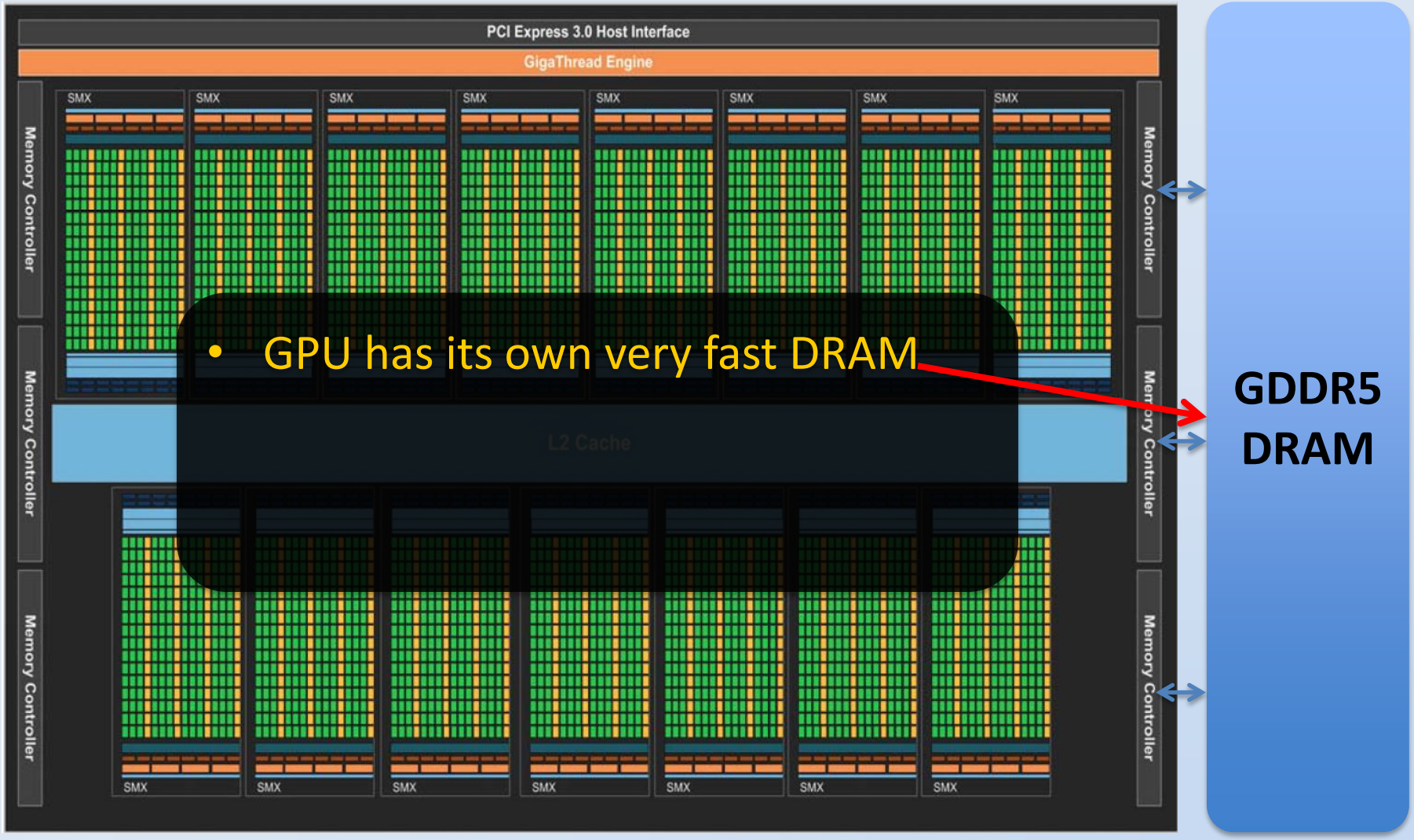
- GPU has its own DRAM, higher bandwidth than main DRAM
- BUT, a PCI-express bus connects main DRAM & GPU DRAM
- PCI-e has painfully low bandwidth, painfully high latency
  - PCI-e bandwidth 864 times lower than peak DP GPU flops
- Latency is 1,000x higher than MPI on Titan

Diagram Of The Latest GPU: Kepler

Image Source:
http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

OLCF
Oak Ridge Leadership Computing Facility

Diagram Of The Latest GPU: Kepler

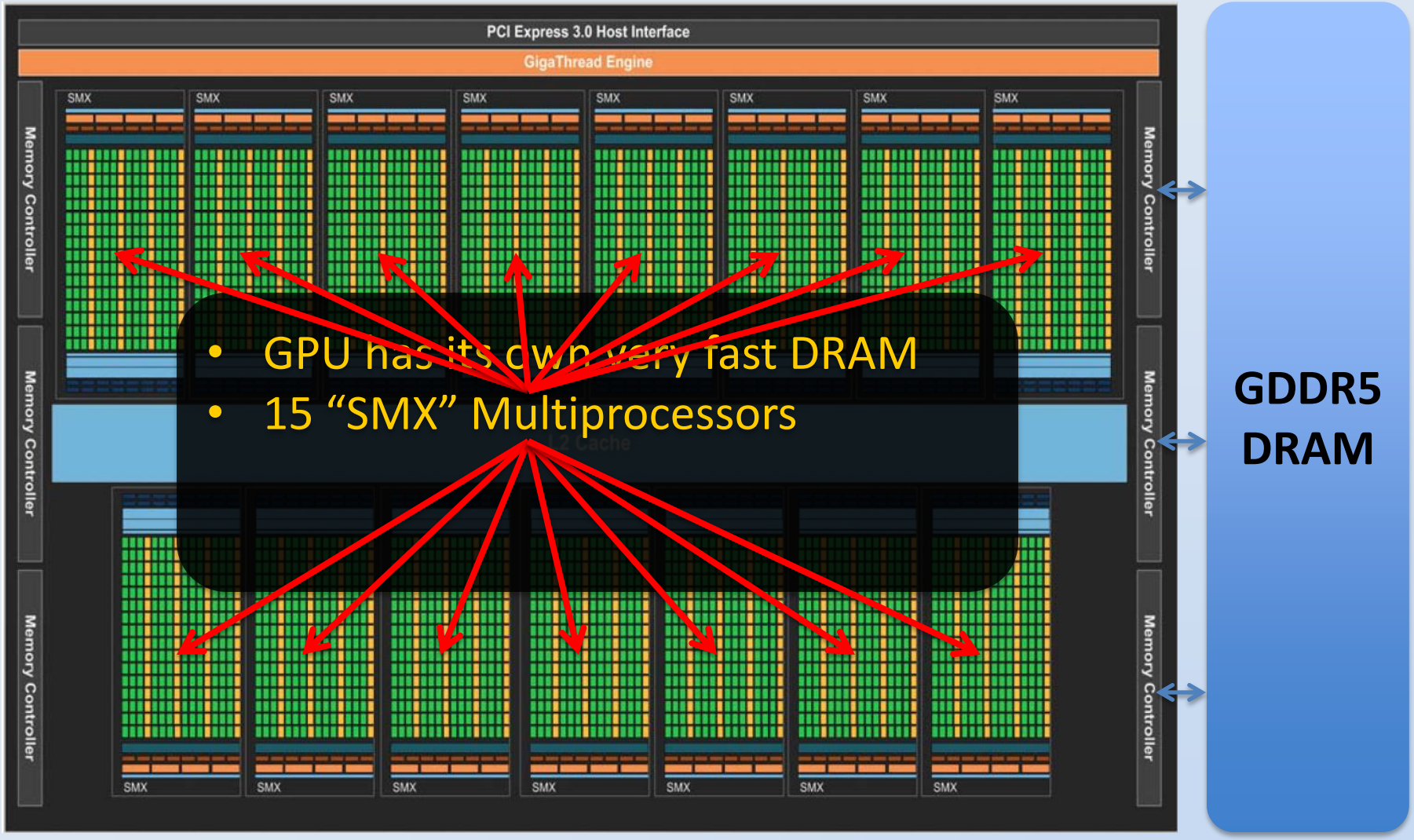


- GPU has its own very fast DRAM

Image Source:
http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

OLCF
Oak Ridge Leadership Computing Facility

Diagram Of The Latest GPU: Kepler

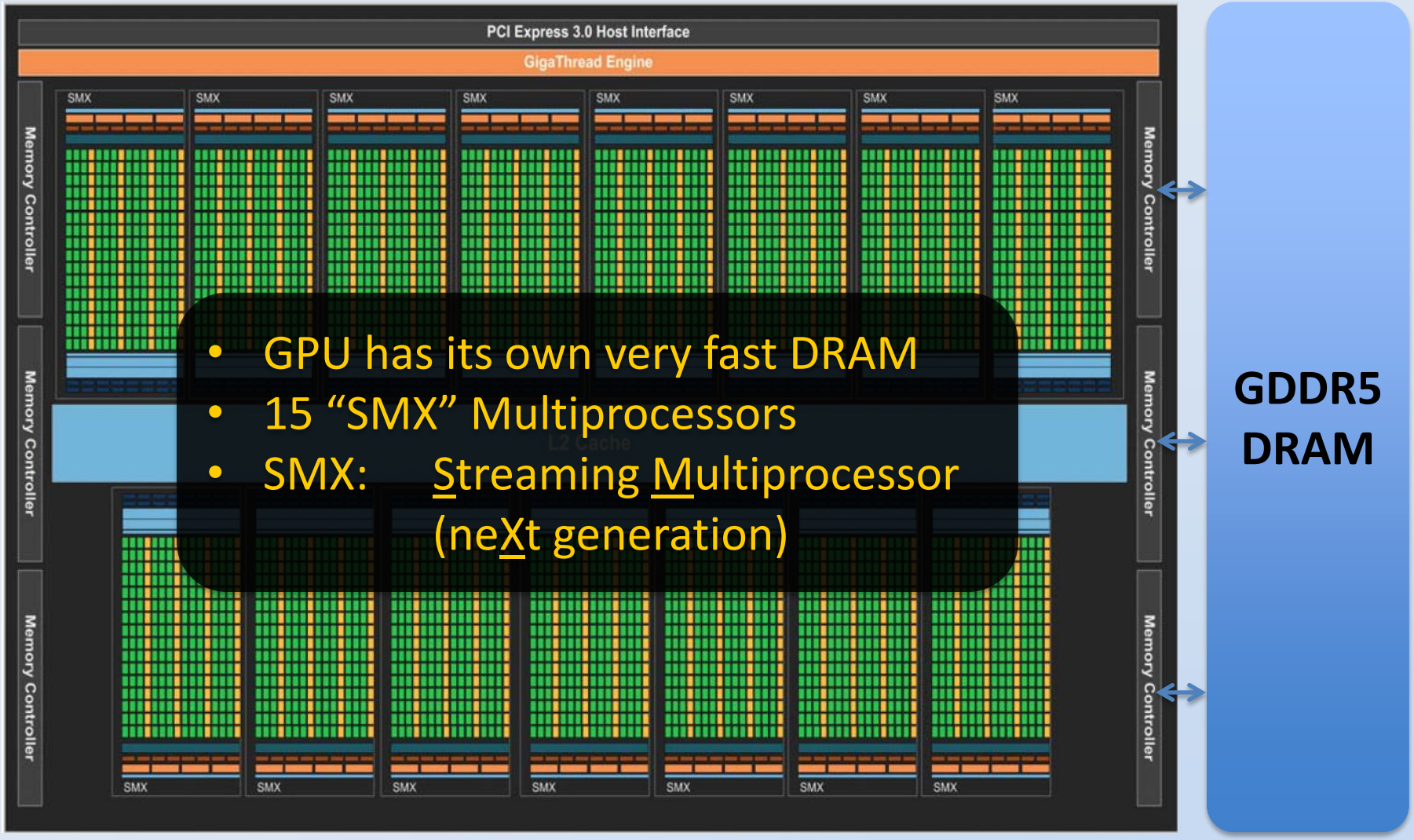


- GPU has its own very fast DRAM
- 15 “SMX” Multiprocessors

Image Source:
http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

OLCF
Oak Ridge Leadership Computing Facility

Diagram Of The Latest GPU: Kepler

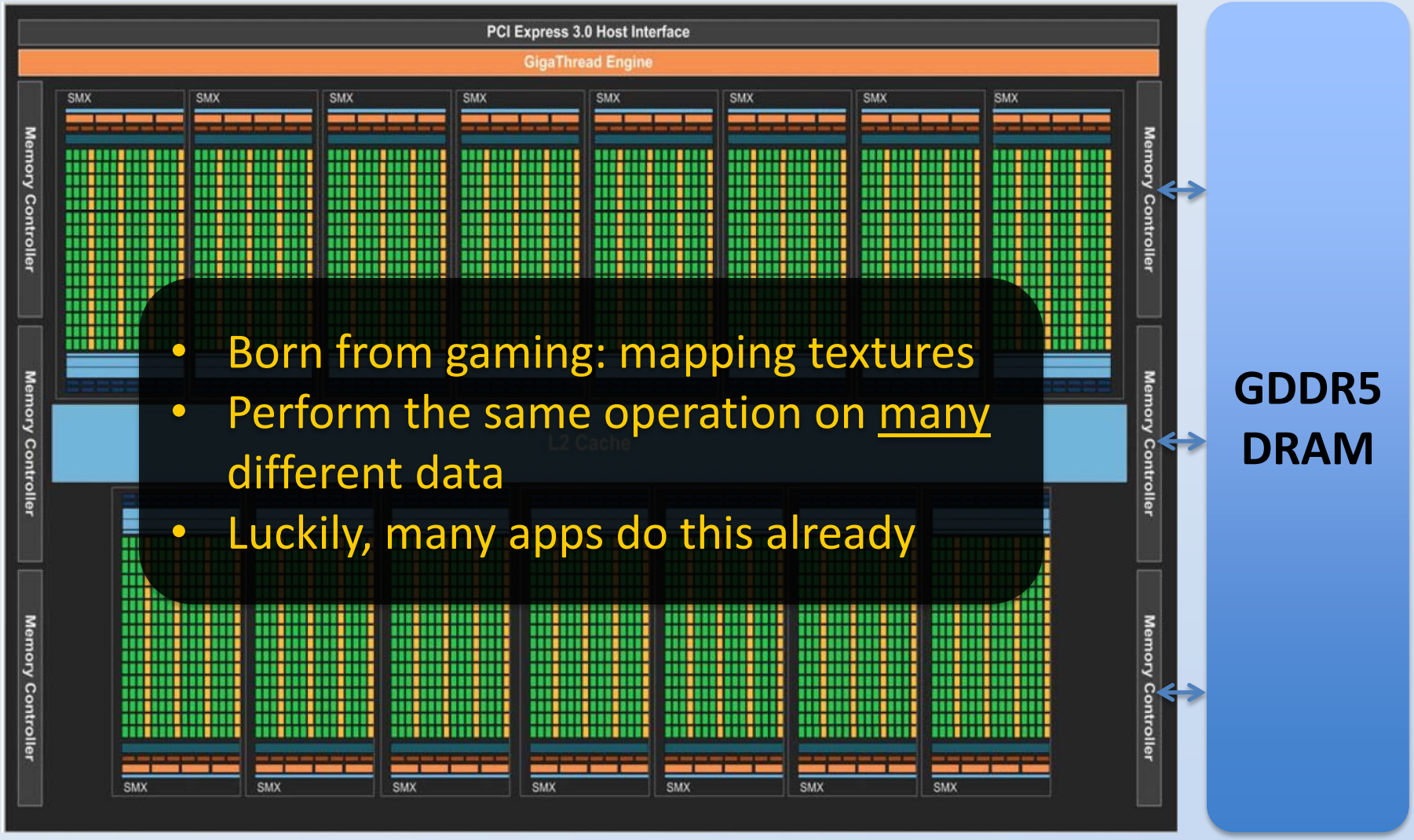


- GPU has its own very fast DRAM
- 15 "SMX" Multiprocessors
- SMX: Streaming Multiprocessor (neXt generation)

Image Source:
http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

OLCF
Oak Ridge Leadership Computing Facility

Diagram Of The Latest GPU: Kepler



- Born from gaming: mapping textures
- Perform the same operation on <u>many</u> different data
- Luckily, many apps do this already

Image Source:
http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

OLCF
Oak Ridge Leadership Computing Facility

Diagram Of The Latest GPU: Kepler

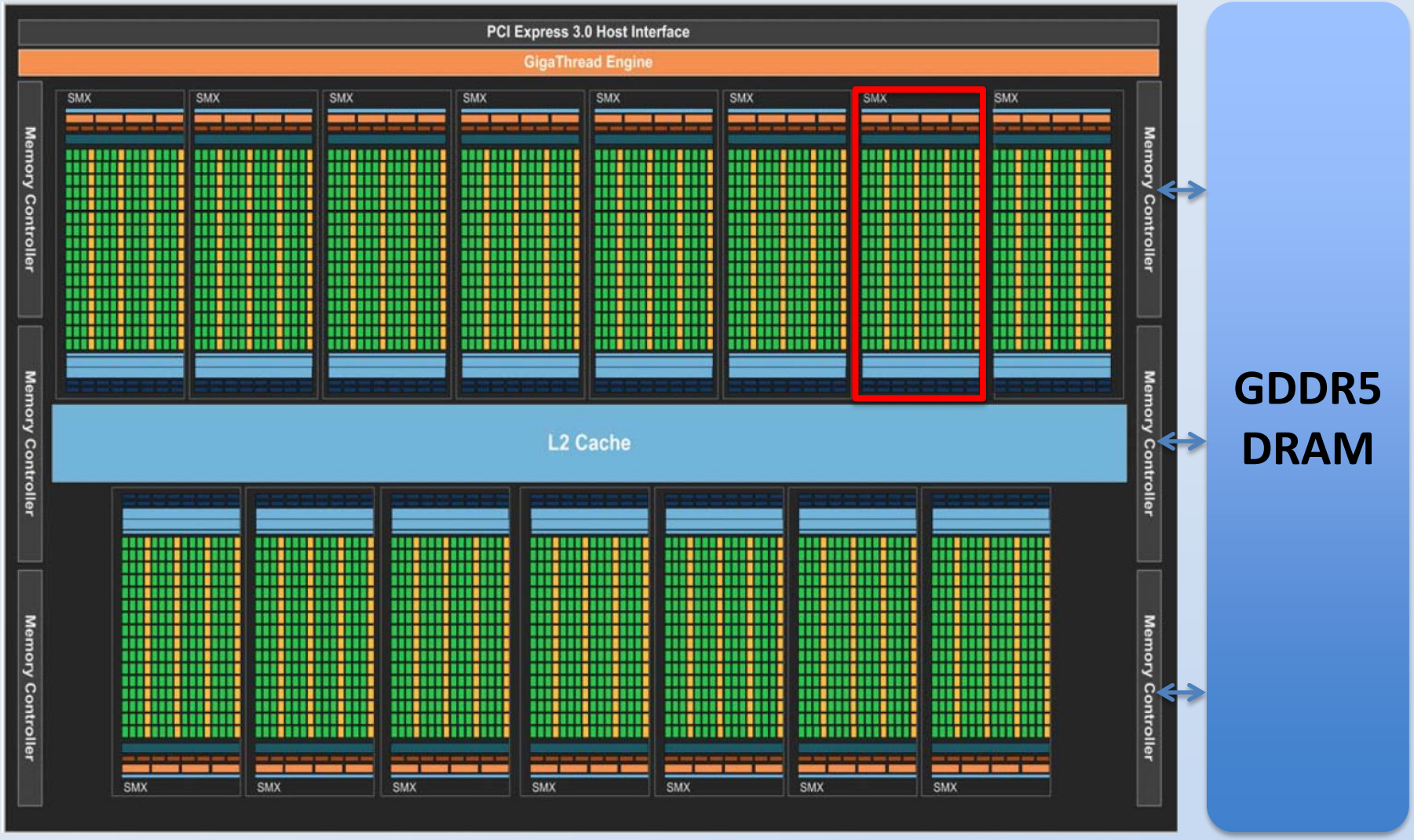


Image Source:
http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

OLCF
Oak Ridge Leadership Computing Facility

Diagram Of An SMX Multiprocessor

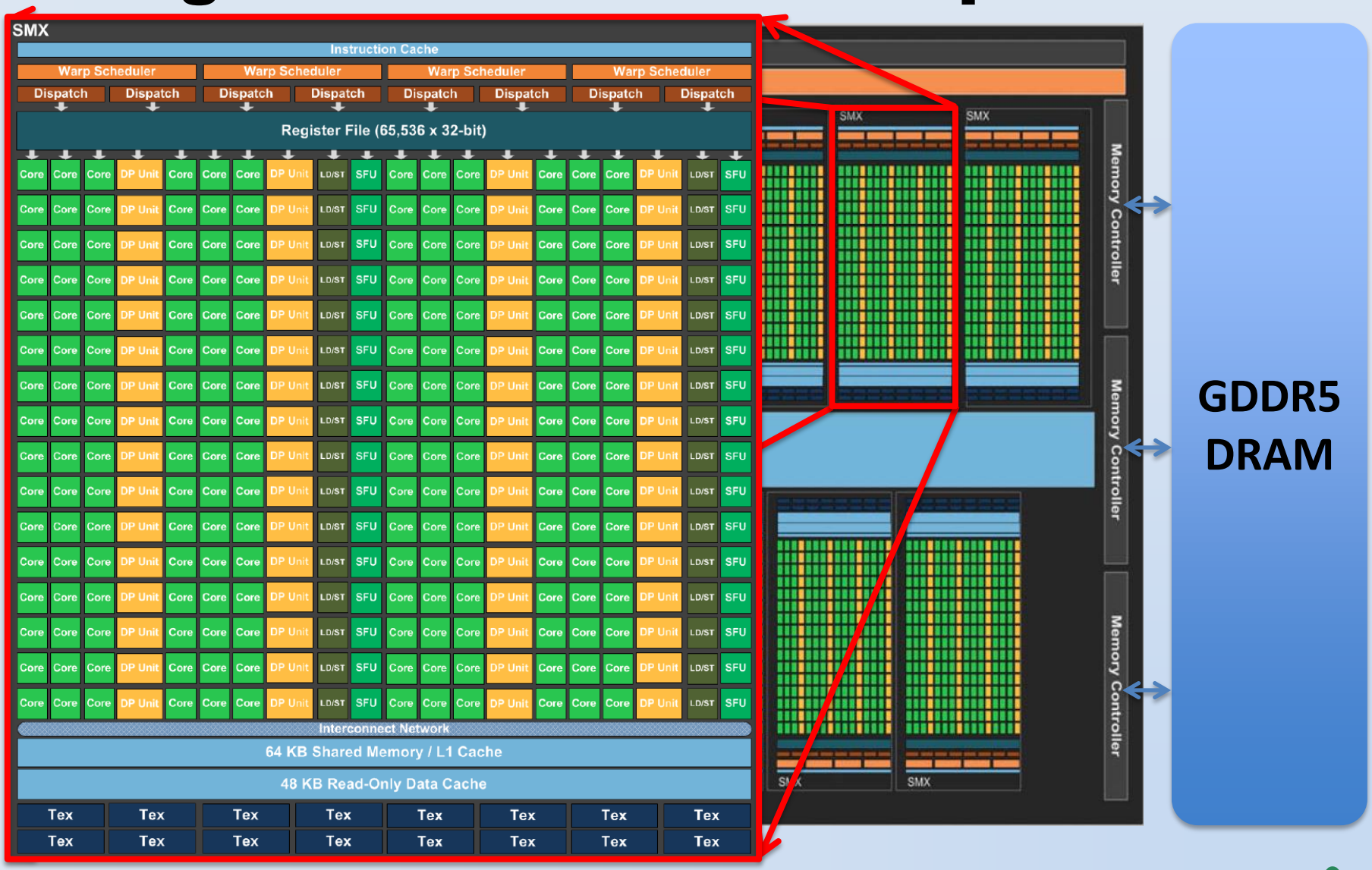
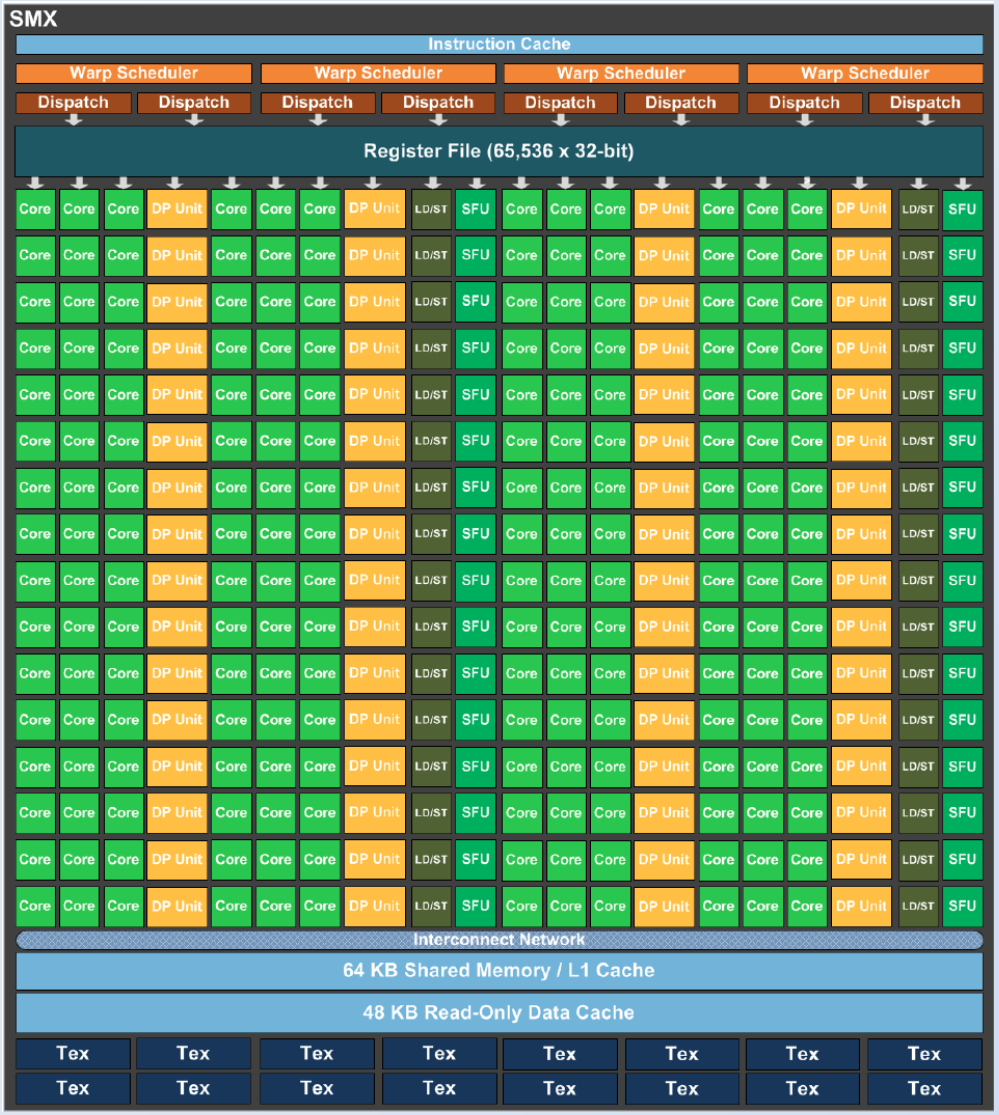


Image Source:
http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Diagram Of An SMX Multiprocessor



- **192** single-precision FP cores
- **64** double-precision FP cores
- These are dumb, lean cores

- Only 64KB general cache
  - Less than 1 MB per GPU
- 64K 4-byte registers
  - Can hold 32K doubles
- Threads launched over cores
  - May have >1 thread per core
  - Threads synchronize and share registers & cache only <u>within</u> SMX not <u>between</u> them

Image Source: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

OLCF
Oak Ridge Leadership Computing Facility

Outline

- Motivation and data movement
- Basic GPU Architecture
- **Basic Programming in CUDA**
- Other Programming Models
- Challenges
- Opportunities

Do You Need / Can You Use GPUs?

- Before you do anything, profile your code!
  - Is there enough user-function runtime to warrant porting?
  - Where are your runtime hotspots?
  - Can you expose massive data-independent threading?
    - Operations inside loops should not be dependent
    - Triangular loop structures are often difficult
    - You need preferably 10,000s threads per GPU
  - Is there divergent branching logic in the kernel?
    - Each SM has only one instruction scheduler
    - Threads execute the exact same instruction
    - Each group of 32 threads must take the same if-then branch
  - Is code restructuring necessary?

Prototypical Loop Transformation

CPU Code

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) / dp(i,j,k,ie)
```

GPU Code

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) / dp(i,j,k,ie)
```

Prototypical Loop Transformation

CPU Code

Outermost loop indexed as "blocks"

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) / dp(i,j,k,ie)
```

GPU Code

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) / dp(i,j,k,ie)
```

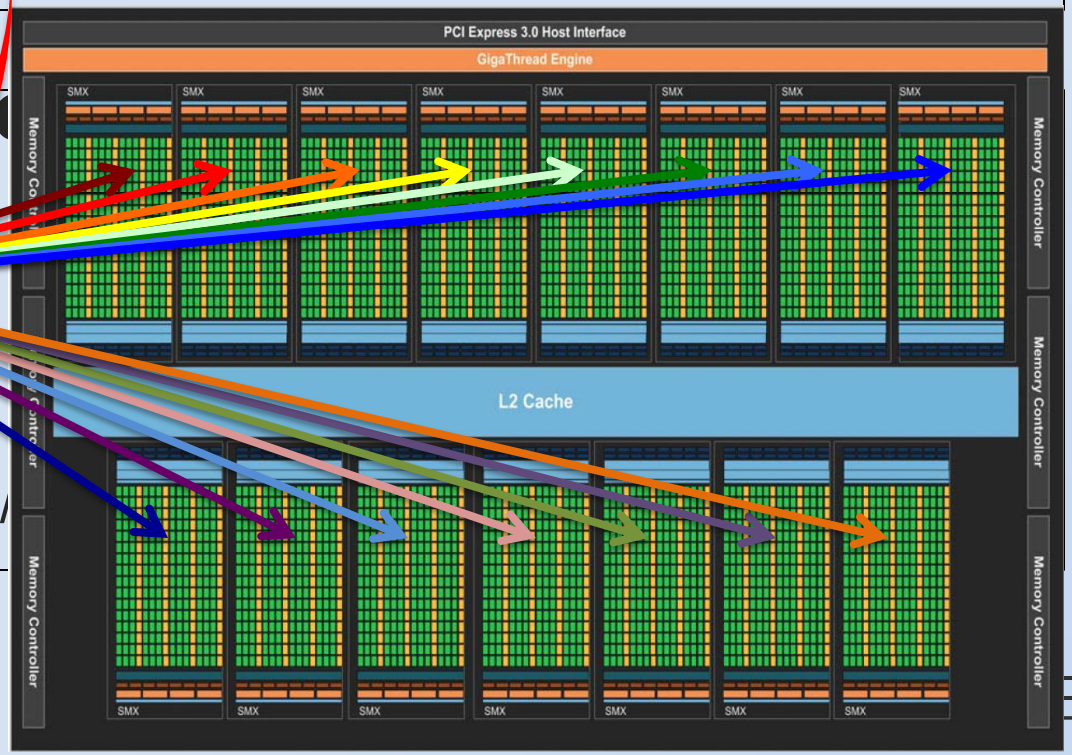
Prototypical Loop Transformation

CPU Code

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) / dp(i,j,k,ie)
```

Different Block Indices Are Computed On Different SMXs

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) /
```



Prototypical Loop Transformation

CPU Code

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) / dp(i,j,k,ie)
```

Innermost loop indexed as "threads"

GPU Code

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) / dp(i,j,k,ie)
```

Prototypical Loop Transformation

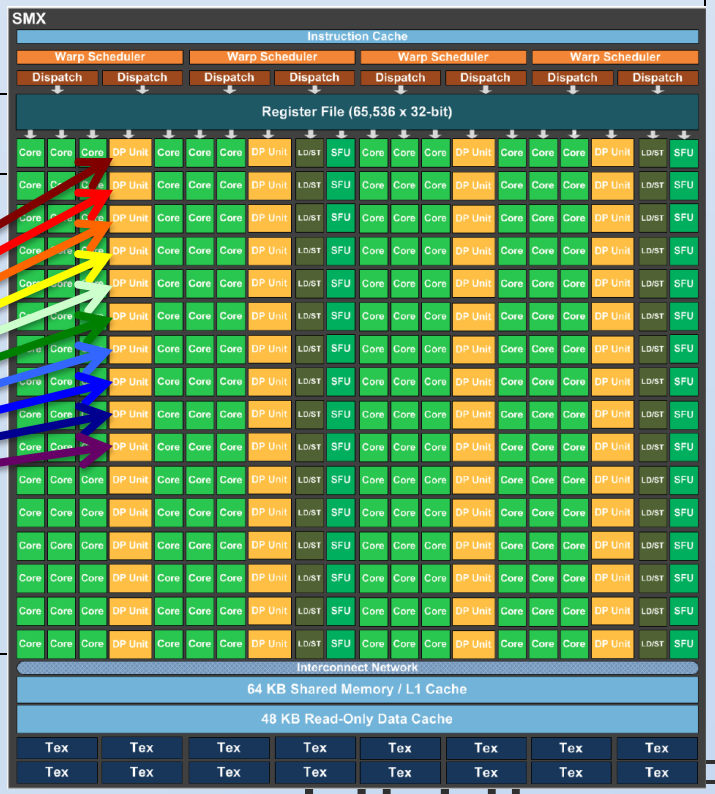
CPU Code

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) / dp(i,j,k,ie)
```

Different Thread Indices Are Computed On Different Cores

GPU Code

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) / dp(i,j,k,ie)
```



Prototypical Loop Transformation

CPU Code

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) / dp(i,j,k,ie)
```

GPU Code

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
Qdp(i,j,k,q,ie) = Qdp(i,j,k,q,ie) / dp(i,j,k,ie)
```

Keep fastest varying indices the same

Think Differently About Threading

CPU Code

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     coefs(1,i,j,k,q,ie) = ...
     coefs(2,i,j,k,q,ie) = ...
     coefs(3,i,j,k,q,ie) = ...
```

GPU Code

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
coefs(1,i,j,k,q,ie) = ...
coefs(2,i,j,k,q,ie) = ...
coefs(3,i,j,k,q,ie) = ...
```

Think Differently About Threading

CPU Code

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     coefs(1,i,j,k,q,ie) = ...
     coefs(2,i,j,k,q,ie) = ...
     coefs(3,i,j,k,q,ie) = ...
```

Coded to respect cache locality

GPU Code

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
coefs(1,i,j,k,q,ie) = ...
coefs(2,i,j,k,q,ie) = ...
coefs(3,i,j,k,q,ie) = ...
```

Think Differently About Threading

CPU Code

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     coefs(1,i,j,k,q,ie) = ...
     coefs(2,i,j,k,q,ie) = ...
     coefs(3,i,j,k,q,ie) = ...
```

Coded to respect cache locality

GPU Code

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
coefs(1,i,j,k,q,ie) = ...
coefs(2,i,j,k,q,ie) = ...
coefs(3,i,j,k,q,ie) = ...
```

However, these will not be sequential accesses on GPUs

Think Differently About Threading

CPU Code

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     coefs(1,i,j,k,q,ie) = ...
     coefs(2,i,j,k,q,ie) = ...
     coefs(3,i,j,k,q,ie) = ...
```

- Memory accessed in the order of instructions
  - coefs(1,1,1,1,…)
  - coefs(2,1,1,1,…)
  - coefs(3,1,1,1,…)
  - coefs(1,2,1,1,…)
  - coefs(2,2,1,1,…)
  - …

GPU Code

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
coefs(1,i,j,k,q,ie) = ...
coefs(2,i,j,k,q,ie) = ...
coefs(3,i,j,k,q,ie) = ...
```

- Memory accessed in the order of threads
  - coefs(1,1,1,1,…)
  - coefs(1,2,1,1,…)
  - |
  - coefs(1,N,1,1,…)
  - coefs(1,1,2,1,…)
  - coefs(1,2,2,1,…)

Think Differently About Threading

CPU Code

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     coefs(1,i,j,k,q,ie) = ...
     coefs(2,i,j,k,q,ie) = ...
     coefs(3,i,j,k,q,ie) = ...
```

GPU Code

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
coefs(1,i,j,k,q,ie) = ...
coefs(2,i,j,k,q,ie) = ...
coefs(3,i,j,k,q,ie) = ...
```

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
coefs(i,j,k,q,ie,1) = ...
coefs(i,j,k,q,ie,2) = ...
coefs(i,j,k,q,ie,3) = ...
```

Think Differently About Threading

CPU Code

```
do ie=1,nelemd
 do q=1,qsize
  do k=1,nlev
   do j=1,np
    do i=1,np
     coefs(1,i,j,k,q,ie) = ...
     coefs(2,i,j,k,q,ie) = ...
     coefs(3,i,j,k,q,ie) = ...
```

GPU Code

```
ie = blockidx%y
q  = blockidx%x
k  = threadidx%z
j  = threadidx%y
i  = threadidx%x
coefs(i,j,k,q,ie,1) = ...
coefs(i,j,k,q,ie,2) = ...
coefs(i,j,k,q,ie,3) = ...
```

- Memory accessed in the order of threads
  - coefs(1,1,1,…)
  - coefs(2,1,1,…)
  - |
  - coefs(N,1,1,…)
  - coefs(1,2,1,…)
  - coefs(2,2,1,…)

Outline

- Motivation and data movement
- Basic GPU Architecture
- Basic Programming in CUDA
- **Other Programming Models**
- Challenges
- Opportunities

Various GPU Coding Options

- CUDA and CUDA FORTRAN
  - Similar to the above examples, lower level, hand-optimized
  - Likely the best option for “hot spots” in your code
- OpenCL (a little more cumbersome than CUDA)
  - Works on ATI & Nvidia GPUs, multi-core processors
  - Performance portability is a work in progress

Various GPU Coding Options

- CUDA and CUDA FORTRAN
  - Similar to the above examples, lower level, hand-optimized
  - Likely the best option for “hot spots” in your code
- OpenCL (a little more cumbersome than CUDA)
  - Works on ATI & Nvidia GPUs, multi-core processors
  - Performance portability is a work in progress
- Libraries (BLAS, LAPACK, FFT, etc)

Various GPU Coding Options

- CUDA and CUDA FORTRAN
  - Similar to the above examples, lower level, hand-optimized
  - Likely the best option for “hot spots” in your code
- OpenCL (a little more cumbersome than CUDA)
  - Works on ATI & Nvidia GPUs, multi-core processors
  - Performance portability is a work in progress
- Libraries (BLAS, LAPACK, FFT, etc)
- Directives (A good option going forward)
  - Like OpenMP in nature, more sustainable software development practices
  - They are currently limited in usability, and the API is evolving
  - They are not automatic, you will still have to change your code
    - Luckily these changes usually improve CPU performance as well

Various GPU Coding Options

- CUDA and CUDA FORTRAN
  - Similar to the above examples, lower level, hand-optimized
  - Likely the best option for “hot spots” in your code
- OpenCL (a little more cumbersome than CUDA)
  - Works on ATI & Nvidia GPUs, multi-core processors
  - Performance portability is a work in progress
- Libraries (BLAS, LAPACK, FFT, etc)
- Directives (A good option going forward)
  - Like OpenMP in nature, more sustainable software development practices
  - They are currently limited in usability, and the API is evolving
  - They are not automatic, you will still have to change your code
    - Luckily these changes usually improve CPU performance as well
- GPU-Aware MPI
  - Avoid the explicit PCI-e copies, automatically pipeline large transfers

Outline

- Motivation and data movement
- Basic GPU Architecture
- Basic Programming in CUDA
- Other Programming Models
- **Challenges**
- Opportunities

PCI-e and MPI Considerations

- All MPI transfers from GPU to GPU involve PCI-e
- PCI-e and MPI have roughly the same bandwidth
  - Large transfers effectively double in cost, unless…
  - You break up transfers, pipeline, & overlap PCI-e with MPI
  - Use GPU-aware MPI if available & efficiently implemented
- PCI-e latency is 1,000x longer than MPI (Titan)
  - Small transfers take a huge hit that cannot be hidden
  - Can you overlap this with other computations?
  - Can you rework the algorithm to better chunk MPI comms?
- PCI-e considerations are your dominant constraint
  - Can the data stay resident on GPU?

Maximizing DRAM Bandwidth

- Must address memory from DRAM contiguously!
- PCI-e bandwidth 18x slower than DRAM bandwidth
- DRAM bandwidth 48x slower than FP throughput
- Need to <u>at least</u> be as fast as GPU DRAM

- Often, rearranging how you thread mid-kernel helps to make memory accesses more regular

DRAM Bandwidth << FLOPS

- Example: Tesla series GK110 architecture "Kepler"
  - "Peak" flops in single-prec (SP) per GPU: 5,184 Gigaflops
  - "Peak" flops in double-prec (DP) per GPU: 1,728 Gigaflops
  - Peak memory bandwidth: 36 Billion doubles / sec
- How illusive is peak performance on these GPUs?
  - [Peak SP ops per sec] / [Peak SP data per sec] : **72**
  - [Peak DP ops per sec] / [Peak DP data per sec] : **48**
  - This many computations per memory access is hard to do
  - Using the local cache significantly improves this
  - Re-use data fetched from DRAM by storing in small local cache
- However, even bandwidth-limited apps will improve
  - GPU DRAM bandwidth > main system DRAM bandwidth

Software Engineering Perspective

- Do you have staff available to re-factor the code?
- What is the expected workload of porting?
- Will directives be an option for your code?
  - If not, are you willing to support two separate sources?
- How active is the development on the ported sections of your code?
- Given your initial profile, does the best-case speed-up make sense for a porting effort?
- How will unported code effect core-hour allocations?
  - INCITE, ALCC, DD, etc

The Optimization Hierarchy

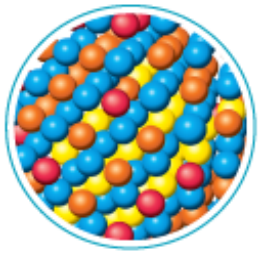
- Most importantly, minimize and / or overlap PCI-e transfers
  - Usually, you overlap w/ MPI or with independent CPU or GPU code
- Provide enough threads to occupy most of the GPU
  - Straightforwardly, you don't want parts of the GPU idle, but also…
  - DRAM latency hidden by switching threads when waiting for memory
  - Only works when enough threads are provided
- Make sure DRAM accesses from threads are sequential
  - This usually gets worse if DRAM accesses are strided
  - This gets much, much worse if DRAM accesses are irregular
- Cache & reuse data in "shared" memory when possible
  - Worst case: shared memory is 8x slower than registers
- Other optimizations we don't have time to cover

“FLOPs Are Free” & Algorithms

- Popular saying because peak flops >> peak bandwidth
- <u>However</u>, it is quite rare to add flops without adding data
- Flops require data, data’s not free, so flops aren’t <u>really</u> free
- But, algorithmic changes may provide some benefit if …
  - Allows More Threading (fills the device better)
  - Decreases Data Dependence (more threading, less syncing)
  - Decreases Local Data Requirements (fits in cache)
- Example: Community Atmosphere Model Vertical Remap
  - Previously used monotone splines & summation-based integrations
    - Neither splines nor summations are data-independent
  - Switched to PPM, integrated twice instead of using summations
  - Spline-based had 2x speed-up, PPM-based had 6x speed-up

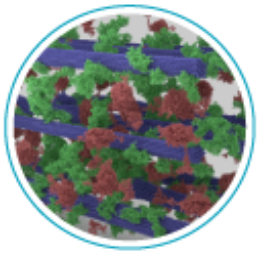
Outline

- Motivation and data movement
- Basic GPU Architecture
- Basic Programming in CUDA
- Other Programming Models
- Challenges
- **Opportunities**



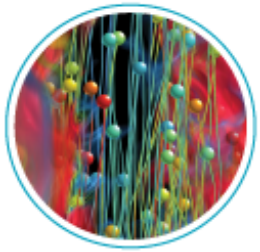
WL-LSMS

Illuminating the role of material disorder, statistics, and fluctuations in nanoscale materials and systems.



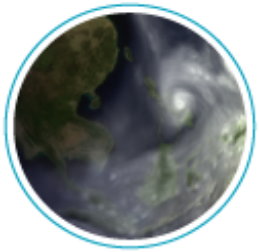
LAMMPS

A molecular description of membrane fusion, one of the most common ways for molecules to enter or exit living cells.



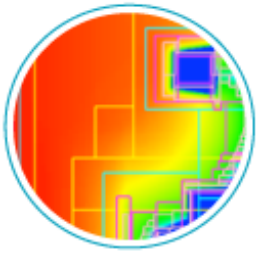
S3D

Understanding turbulent combustion through direct numerical simulation with complex chemistry.



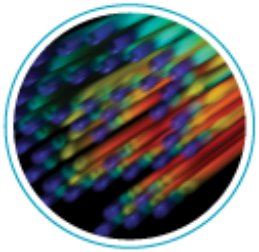
CAM-SE

Answering questions about specific climate change adaptation and mitigation scenarios; realistically represent features like precipitation patterns / statistics and tropical storms.



NRDF

Radiation transport – important in astrophysics, laser fusion, combustion, atmospheric dynamics, and medical imaging – computed on AMR grids.



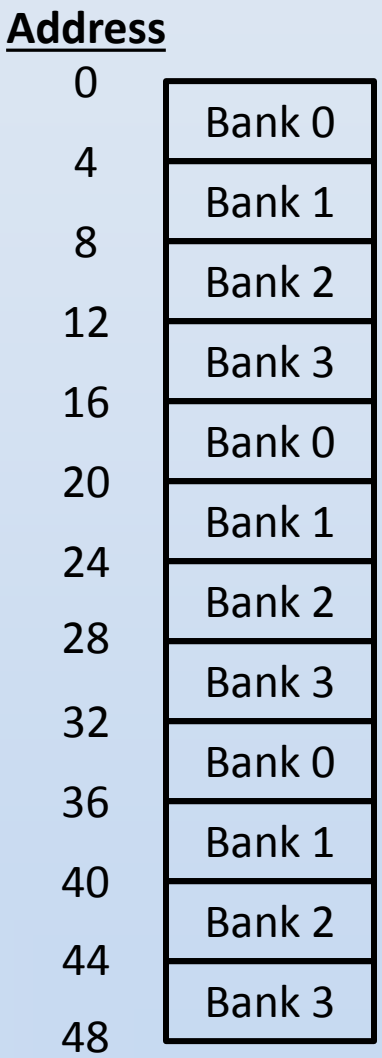
Denovo

Discrete ordinates radiation transport calculations that can be used in a variety of nuclear energy and technology applications.

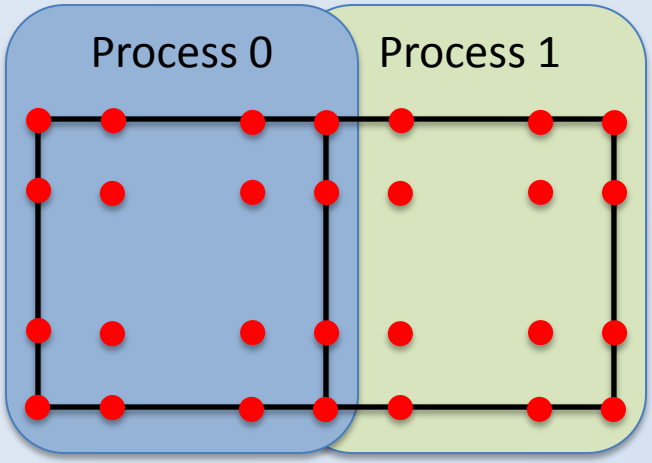
Questions?

Example: 4-Way Banked Memory

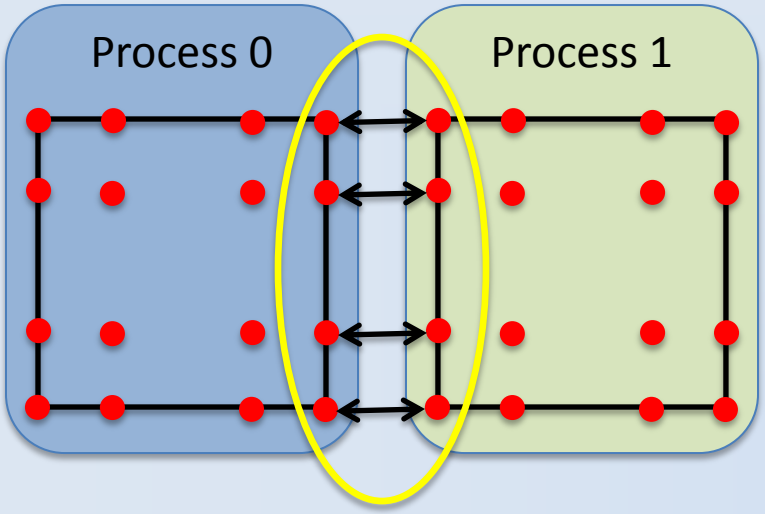
- Each 4-byte section belongs to a different bank
- Successive threads should access successive banks
  - Access to different banks is completely parallel
  - Access to the same bank is serialized
- For most GPUs, L1 cache is 16-way banked
- A warp of threads (32 threads) launches cache memory requests in two groups of 16
  - These requests should be aligned with bank 0
  - If in single precision and well-coded, the data should be retrieved in one cycle per request, as fast as registers



Communication Between Elements

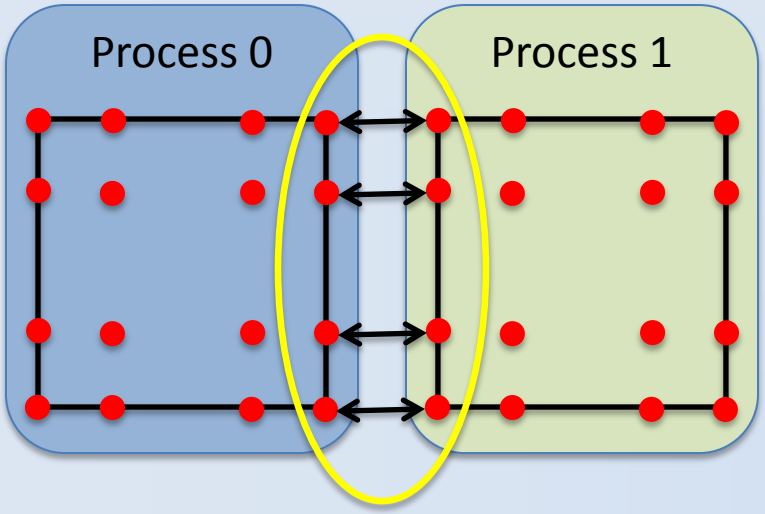


Communication Between Elements



- Boundary points occupy the same location
- Spectral Element requires them to be equal (averaging)
- Discontinuous Galerkin require a flux between them

Communication Between Elements



- Boundary points occupy the same location
- Spectral Element requires them to be equal (averaging)
- Discontinuous Galerkin require a flux between them

Data must be swapped between the two processes

Something like this happens for every atmospheric scheme