Massively Parallel Hybrid GPU Computers: What Do They Mean For Atmospheric Dycores?

Matthew R. Norman

Computational Climate Scientist

Scientific Computing Group

Oak Ridge National Laboratory

The 2012 Dynamical Core Model Intercomparison Project

OAK RIDGE National Laboratory

OLCF Oak Ridge Leadership Computing Facility

Outline

- **Brief Overview of Supercomputer Architecture**
- CPUs and Data Movement
- Introduction to GPUs and the Challenges
- Coding For GPUs
- Implications for Atmospheric Dycores
- Discussion & Questions

Massively Parallel Computers

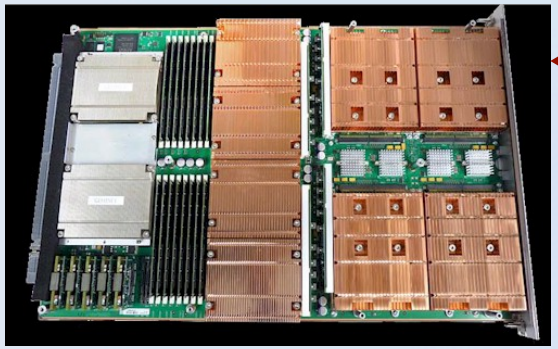
- Architectured in a hierarchy

Massively Parallel Computers

- Architectured in a hierarchy
- Collection of cabinets

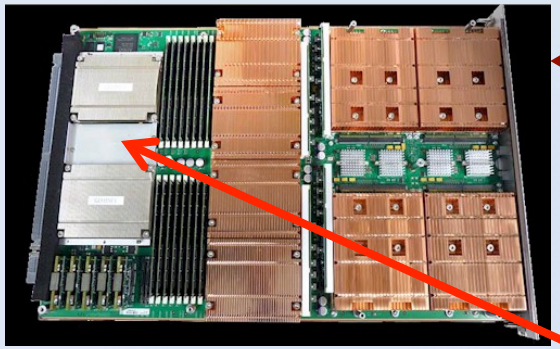
Massively Parallel Computers

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



Massively Parallel Computers

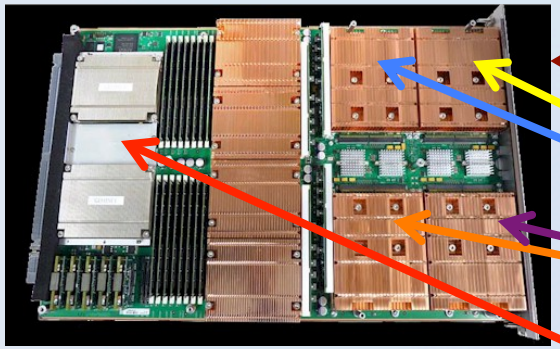
- Architectured in a hierarchy
- Collection of cabinets
- Each cabinet contains “blades”



- A blade has network interconnect

Massively Parallel Computers

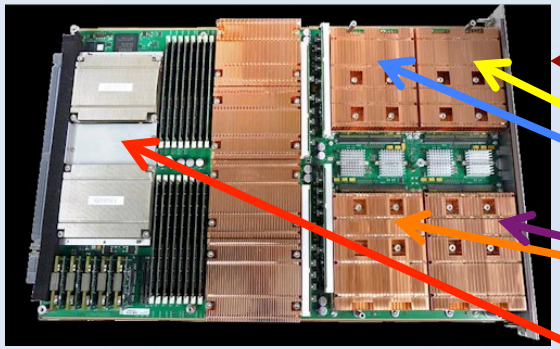
- Architectured in a hierarchy
- Collection of cabinets
- Each cabinet contains “blades”



- A blade has network interconnect and “nodes”

Massively Parallel Computers

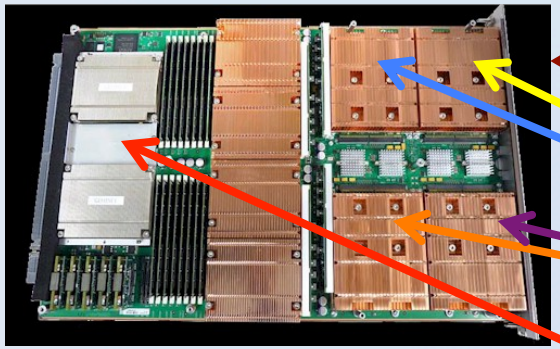
- Architectured in a hierarchy
- Collection of cabinets
- Each cabinet contains “blades”



- 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

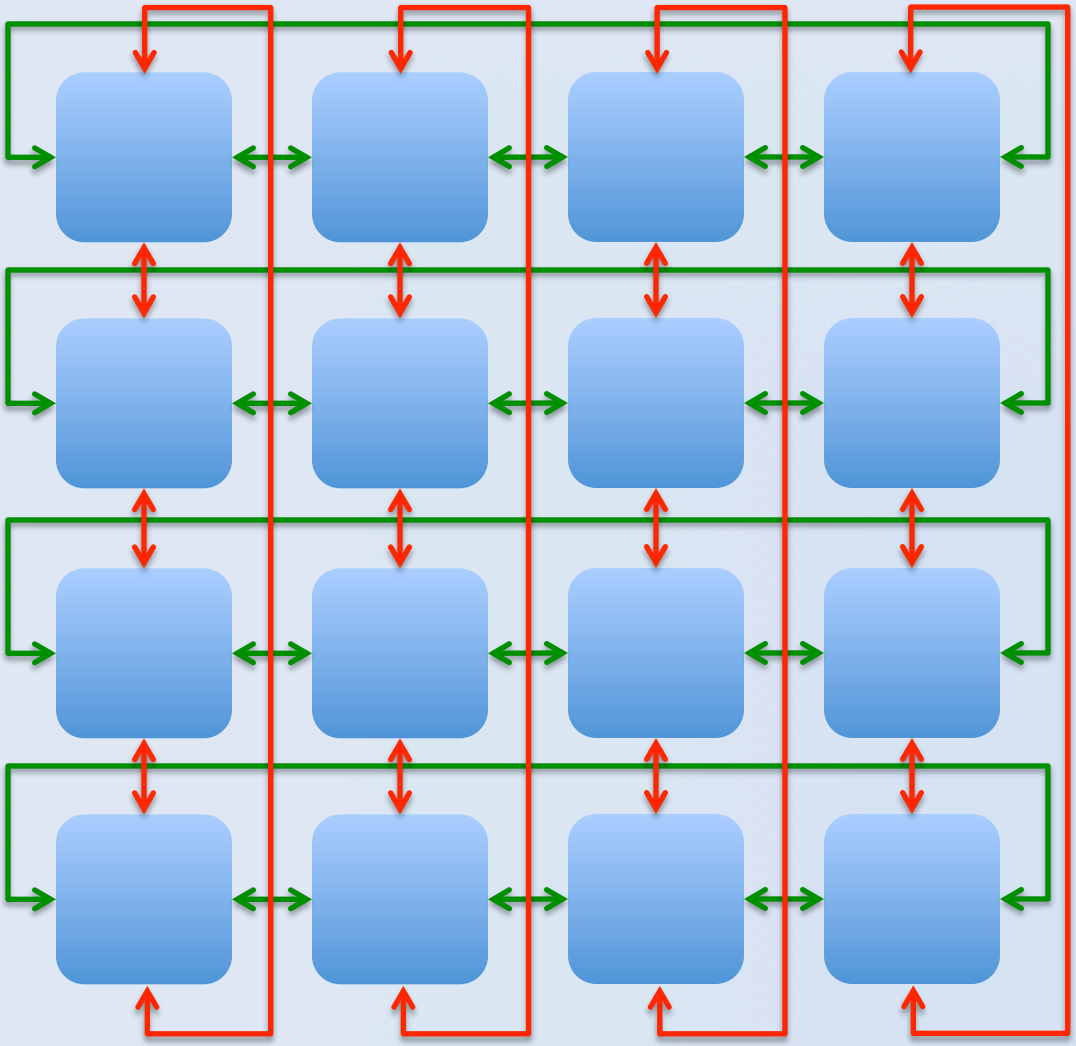
- Peak Performance: Fictitious “perfect” world
  - [Cycles per second per core] * [FP Instructions per cycle] * [cores per processor] * [processors per node] * [# nodes]
  - Often has little bearing on science
- Why?
  - Nothing * Nothing = Nothing (Don’t get too excited)

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)
  - We must feed the processors with useful data
  - Data movement significantly slower than processing data

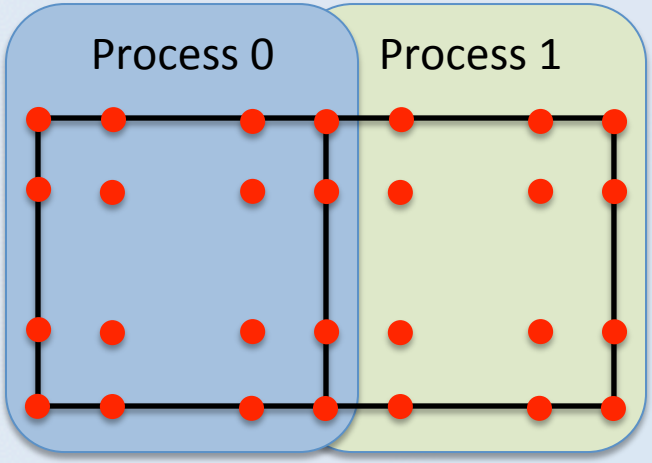
Nodes Communicating Over A Network

Problem: You almost always need data from other nodes

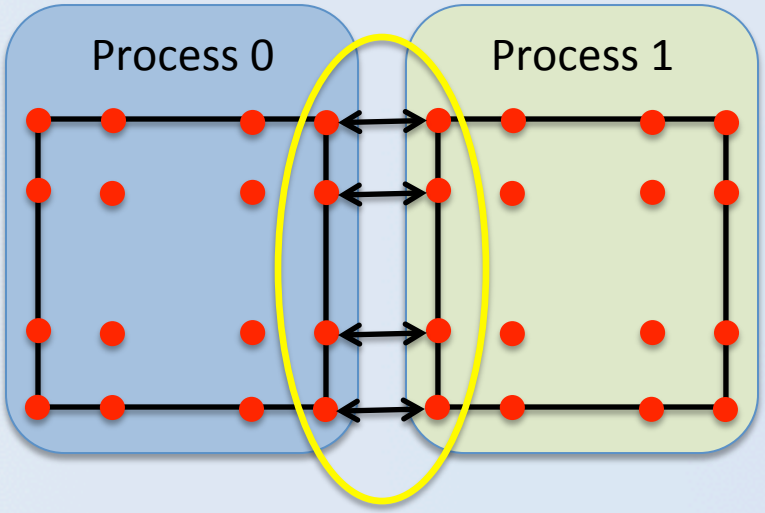


This is the slowest component of the machine, especially when data must transfer across most of the machine

Communication Between Elements

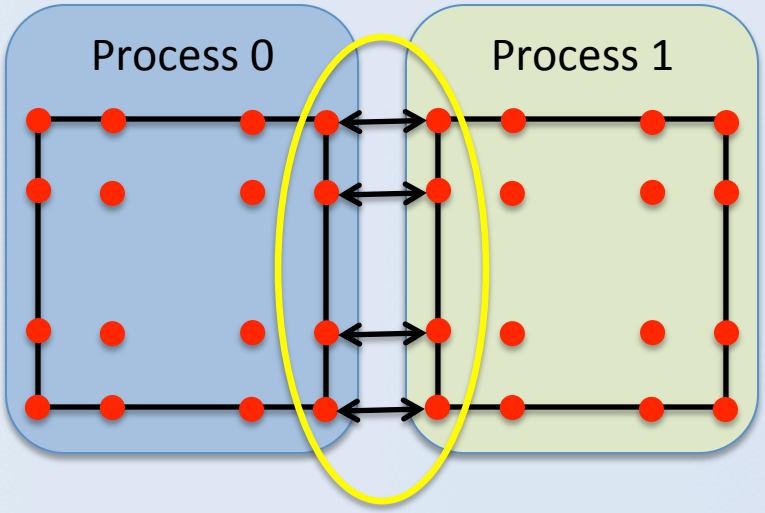


Communication Between Elements



- 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

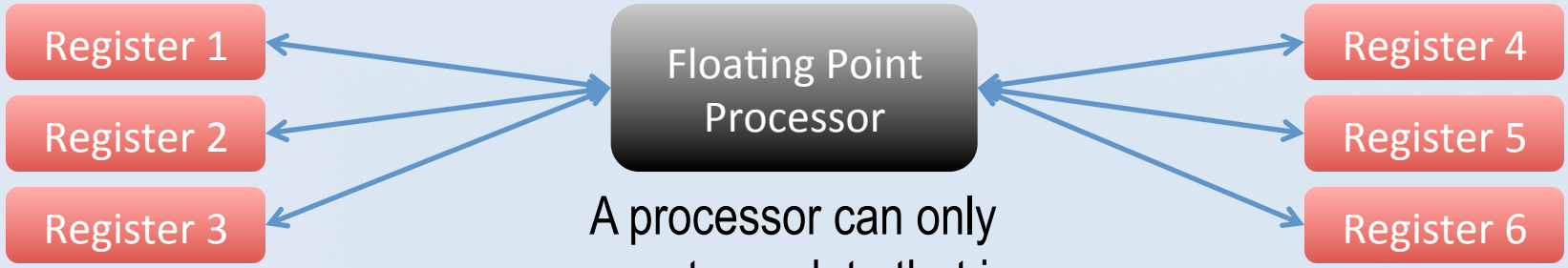
Data must be swapped between the two processes

Something like this happens for every atmospheric scheme

Outline

- Brief Overview of Supercomputer Architecture
- **CPUs and Data Movement**
- Introduction to GPUs and the Challenges
- Coding For GPUs
- Implications for Atmospheric Dycores
- Discussion & Questions

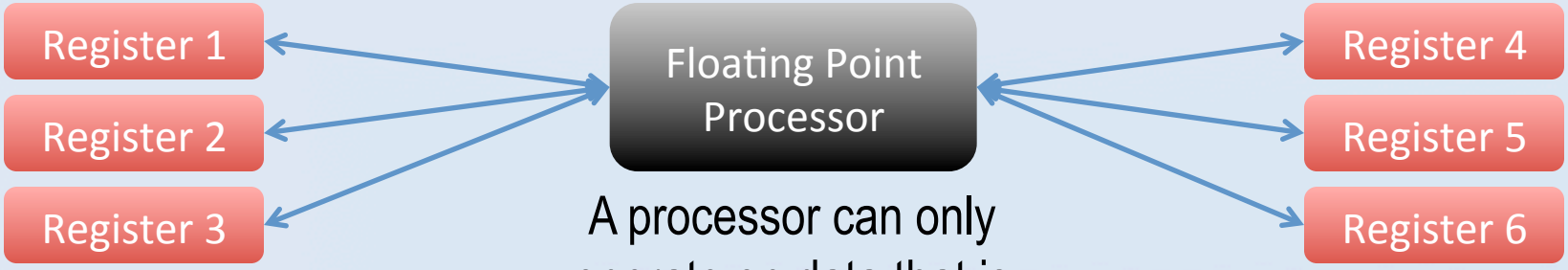
Processors Require Data In “Registers”



Registers are very small, very fast, and very expensive

A processor can only operate on data that is in registers

Processors Require Data In “Registers”



A processor can only operate on data that is in registers

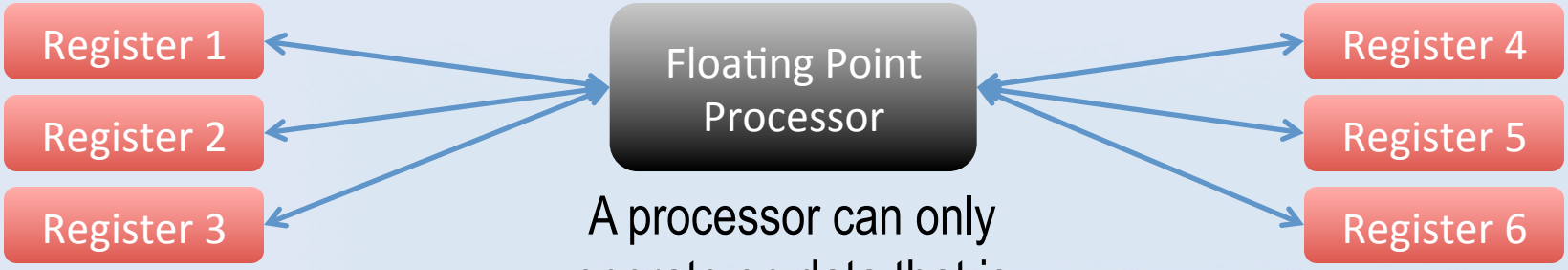
Registers are very small, very fast, and very expensive

Example

Register 3 = Register 1 + Register 2

Register 6 = Register 4 * Register 5

Processors Require Data In "Registers"



Registers are very small, very fast, and very expensive

A processor can only operate on data that is in registers

Example

Register 3 = Register 1 + Register 2

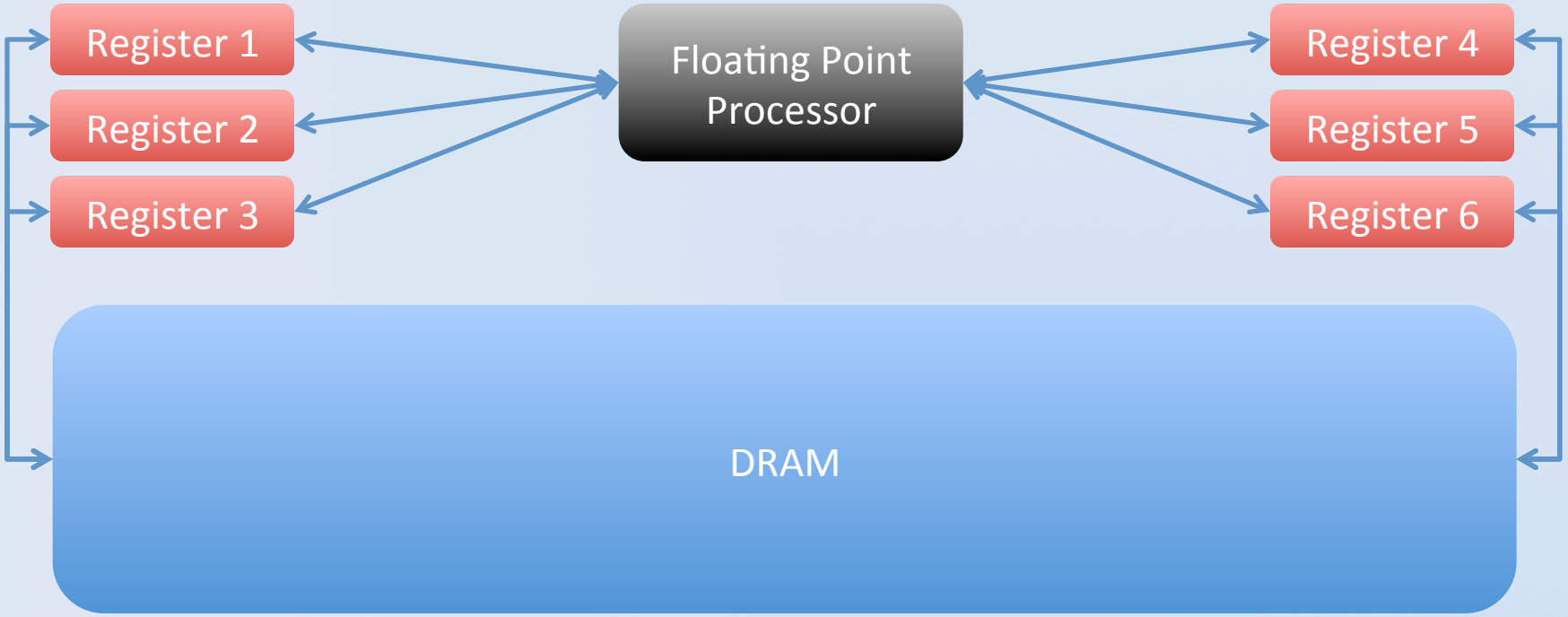
Register 6 = Register 4 * Register 5

Problem:

It takes time to get data into registers

Registers Are Fed Data from DRAM

DRAM is relatively slow memory traveling over slow wires

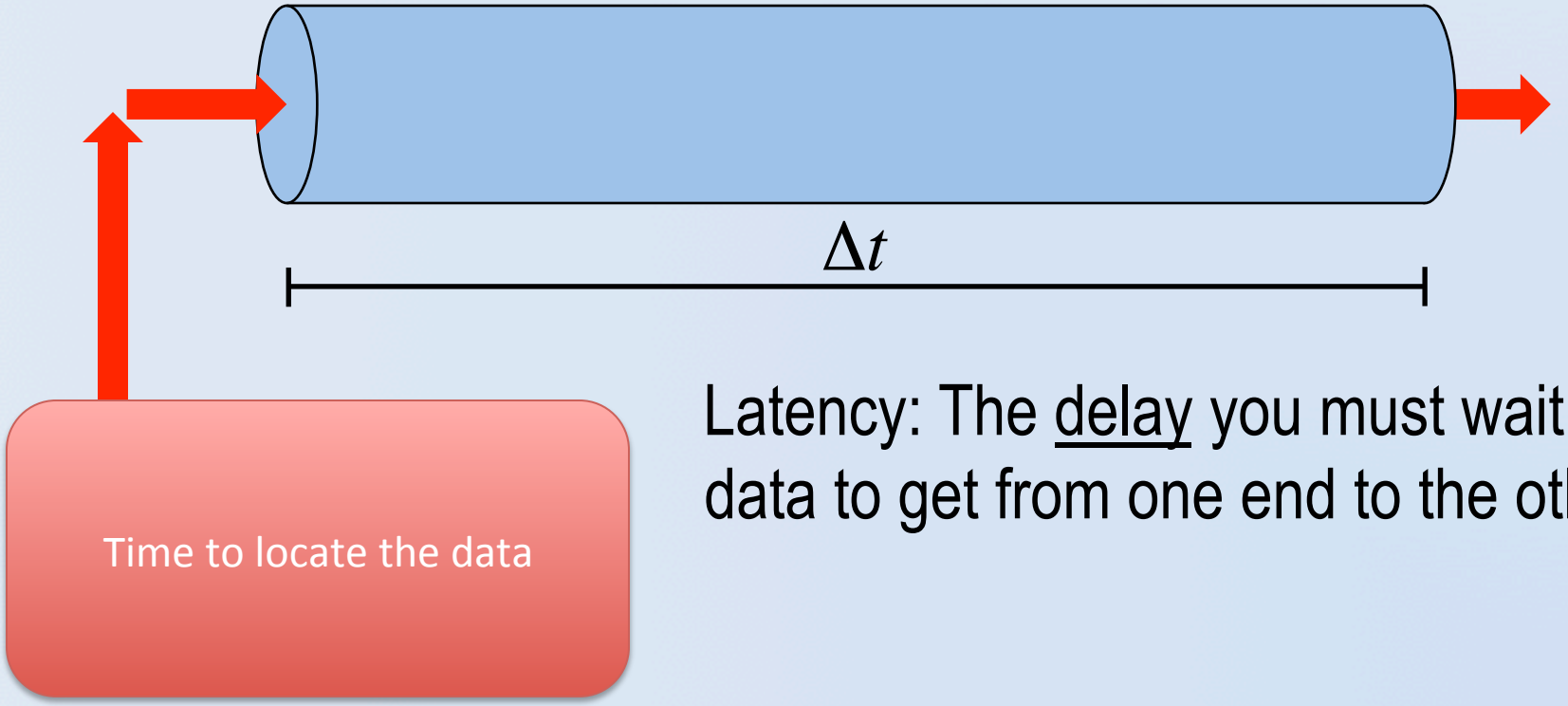


Two Concepts For Data Transfer

Data is transferred over wires. Think of them as pipelines
Two main properties: Latency & Bandwidth

Two Concepts For Data Transfer

Data is transferred over wires. Think of them as pipelines
Two main properties: Latency & Bandwidth



Latency: The delay you must wait for data to get from one end to the other

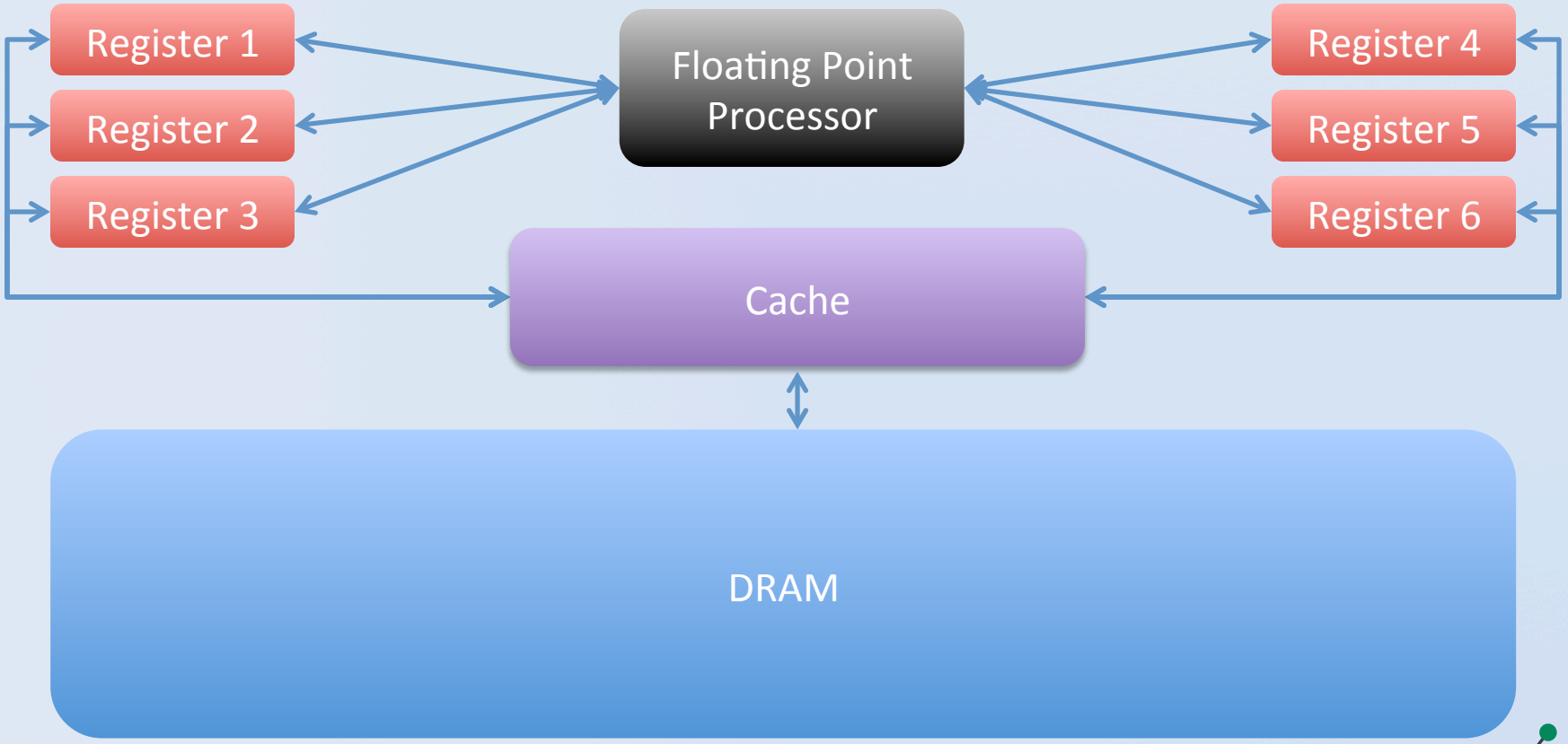
Two Concepts For Data Transfer

Data is transferred over wires. Think of them as pipelines
Two main properties: Latency & Bandwidth

Bandwidth: The amount of data you can pass through per second

Caching Is Faster For Reused Data

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



The Power Cost Of CPUs

- Most of the CPU doesn't directly compute
- Multiple Cache Levels, Cache Policies, Branch Prediction, Prefetching, Out of Order Execution Engine
  - These all consume power
  - High GHz clock rates leak significant power and heat
  - Cooling costs are a significant portion of the power budget
- Multi-core generally improves power efficiency
  - More difficult to code, hyper-threading very hard to access
  - Paired cores contend for the same resources
  - Cache coherence & "snoopers" consume energy
  - Work needs to be separated and fairly independent

Outline

- Brief Overview of Supercomputer Architecture
- CPUs and Data Movement
- **Introduction to GPUs and the Challenges**

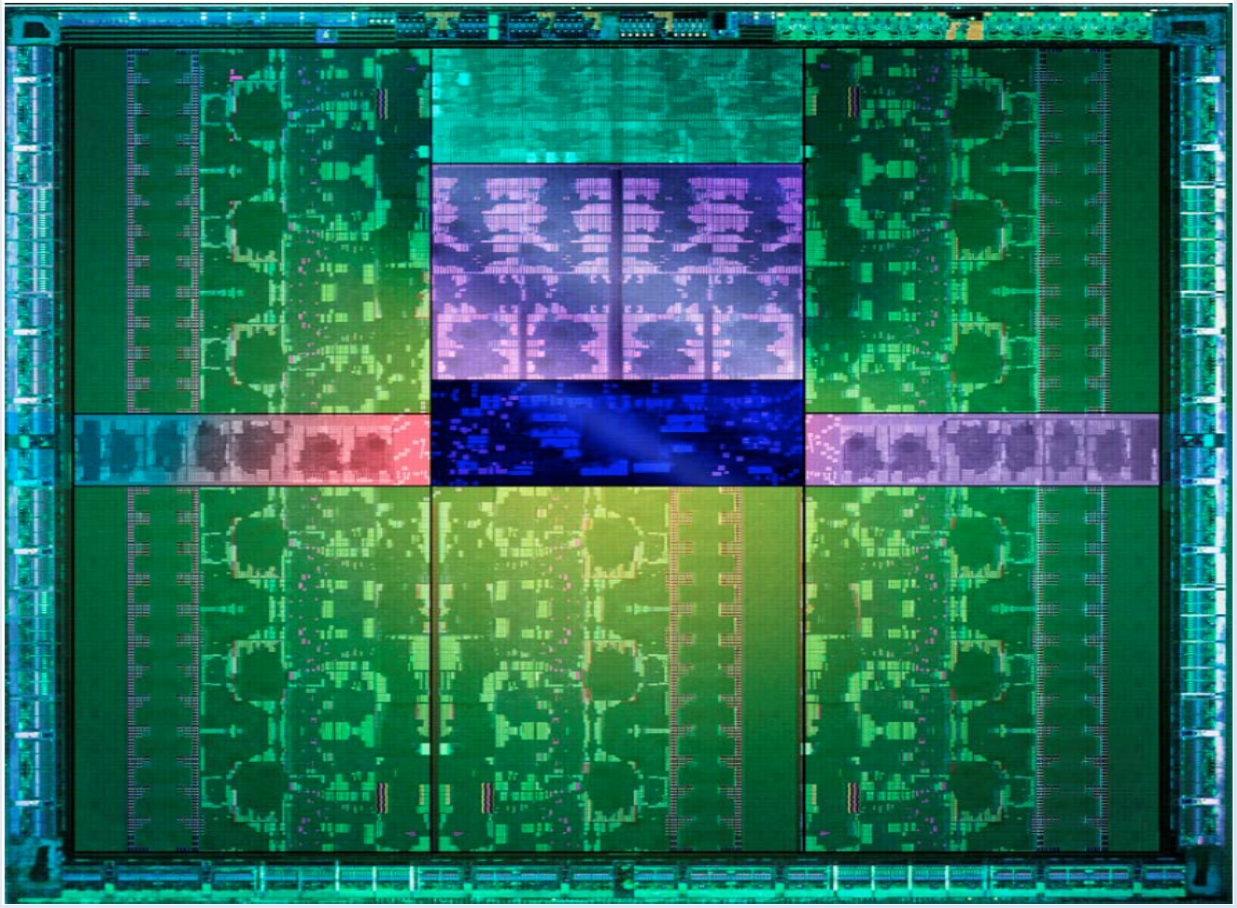


Image Source: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Diagram Of The Latest GPU: Kepler

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

Diagram Of The Latest GPU: Kepler

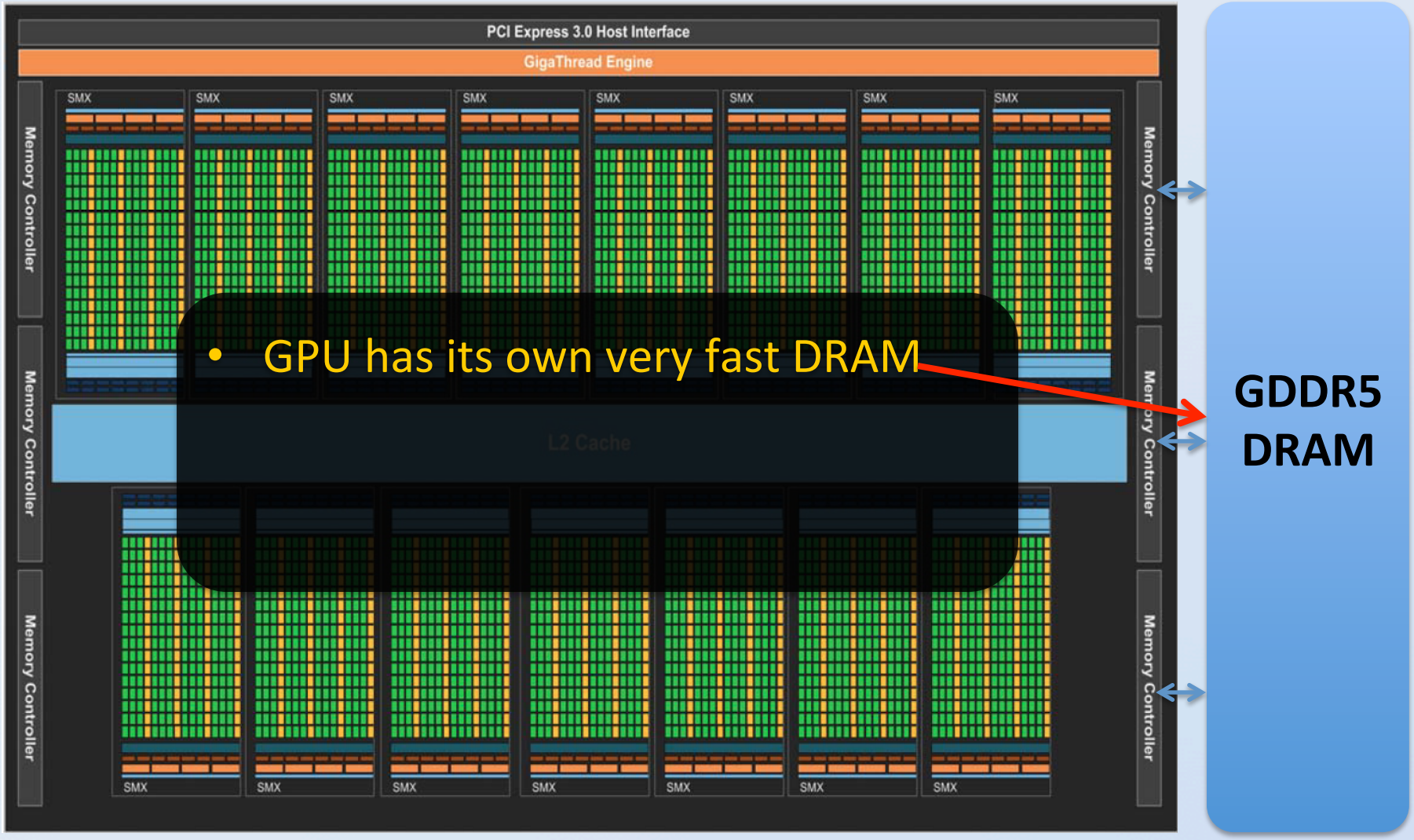


Image Source: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Diagram Of The Latest GPU: Kepler

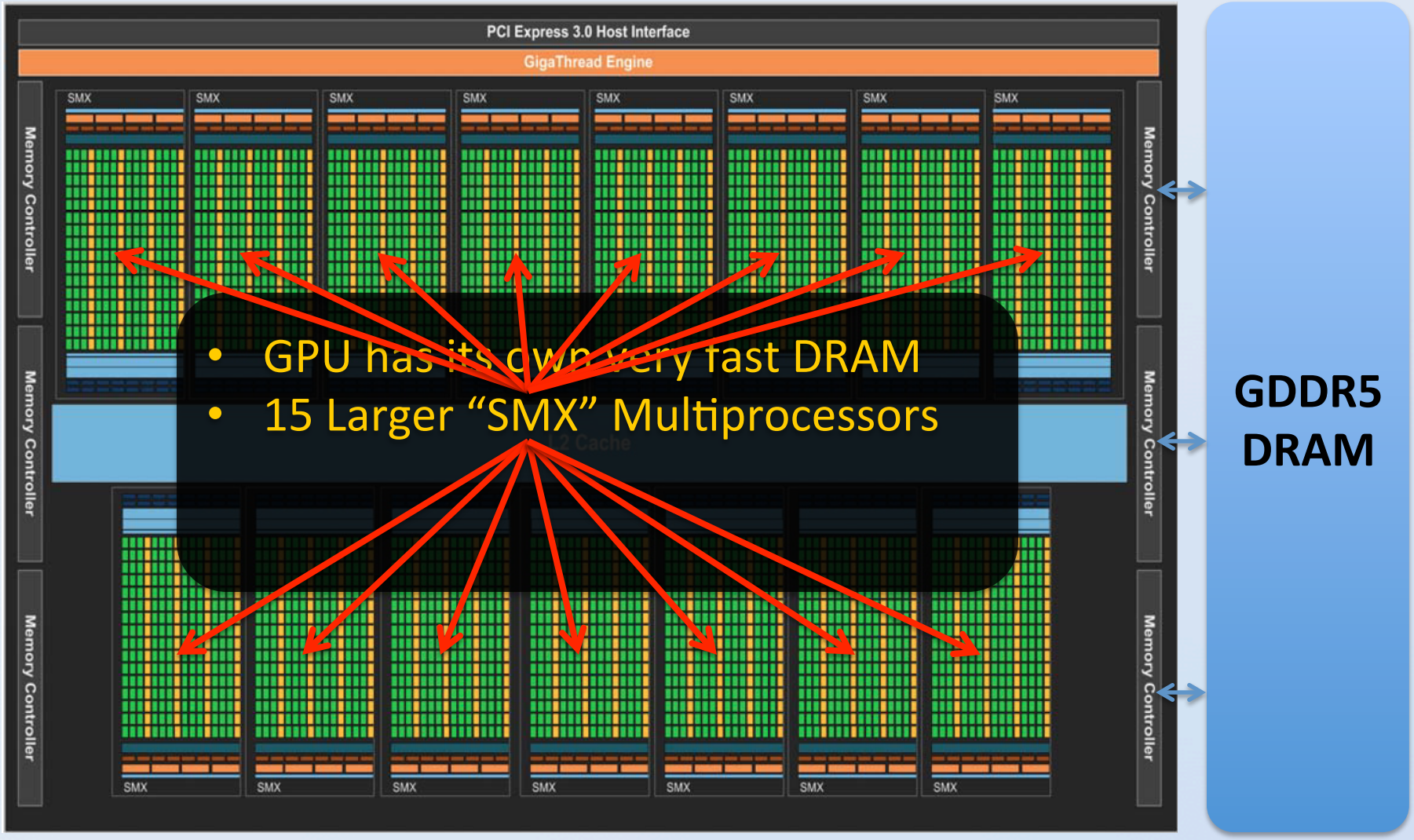


Image Source: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Diagram Of The Latest GPU: Kepler

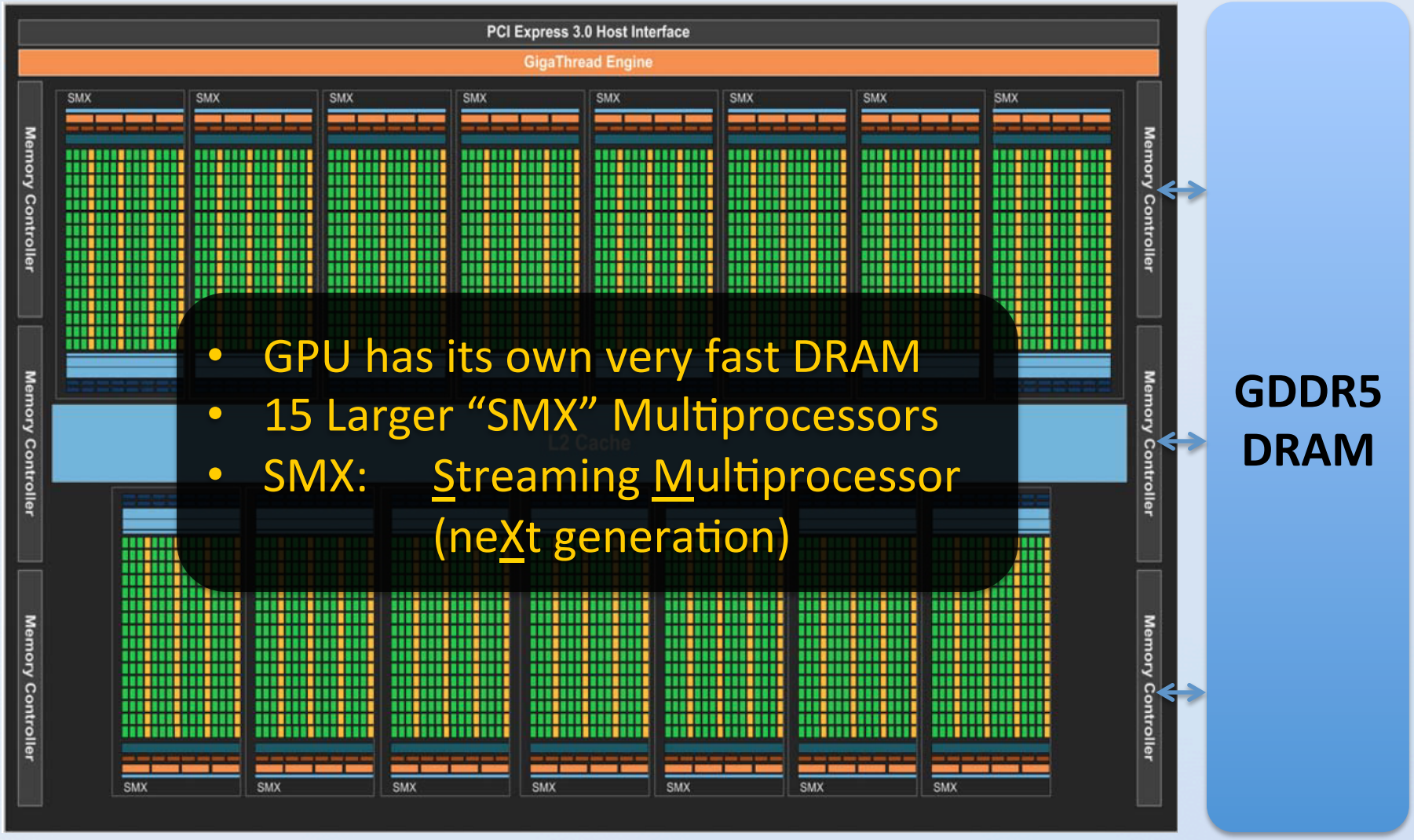
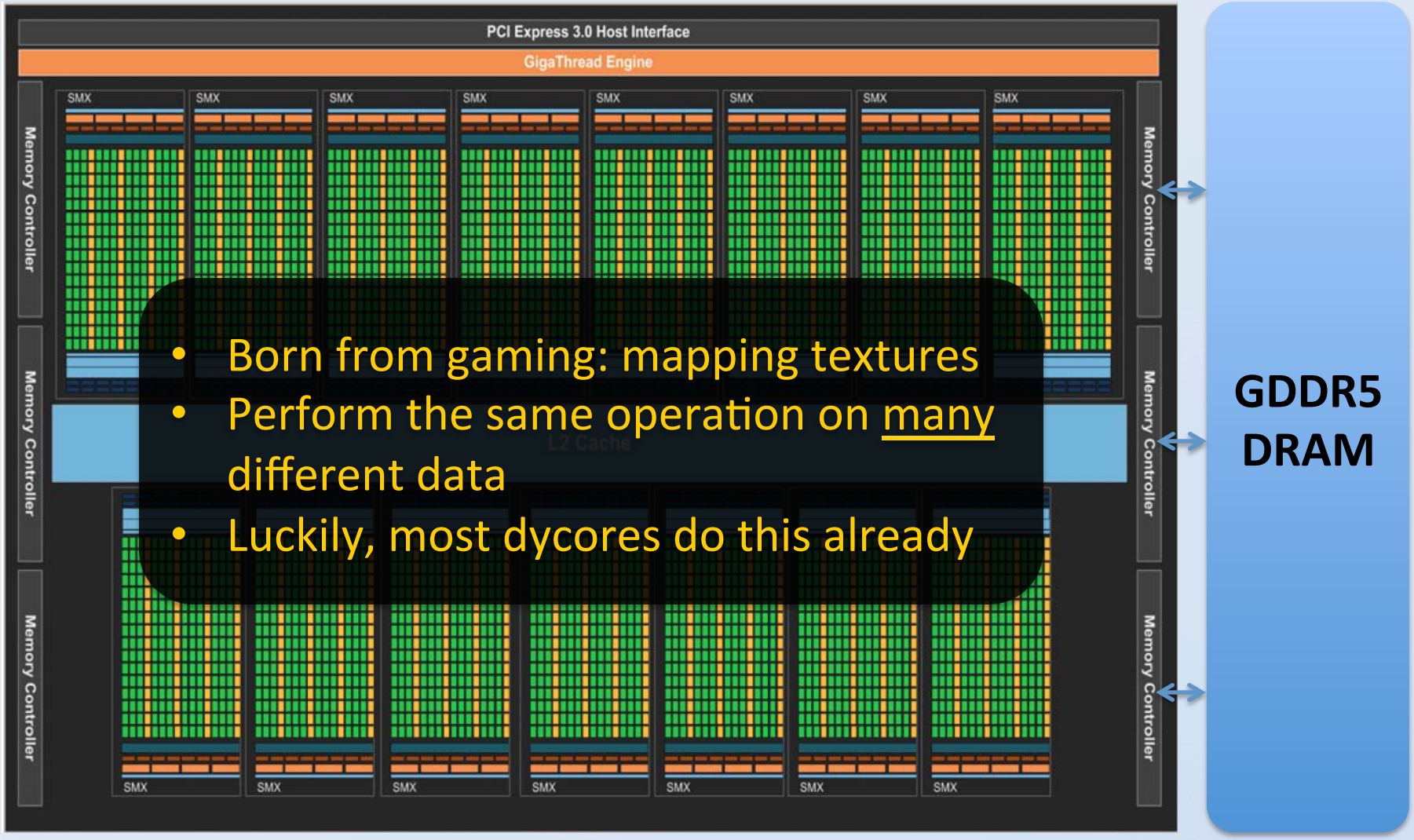


Image Source: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Diagram Of The Latest GPU: Kepler



- Born from gaming: mapping textures
- Perform the same operation on many different data
- Luckily, most dycores do this already

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

Diagram Of The Latest GPU: Kepler

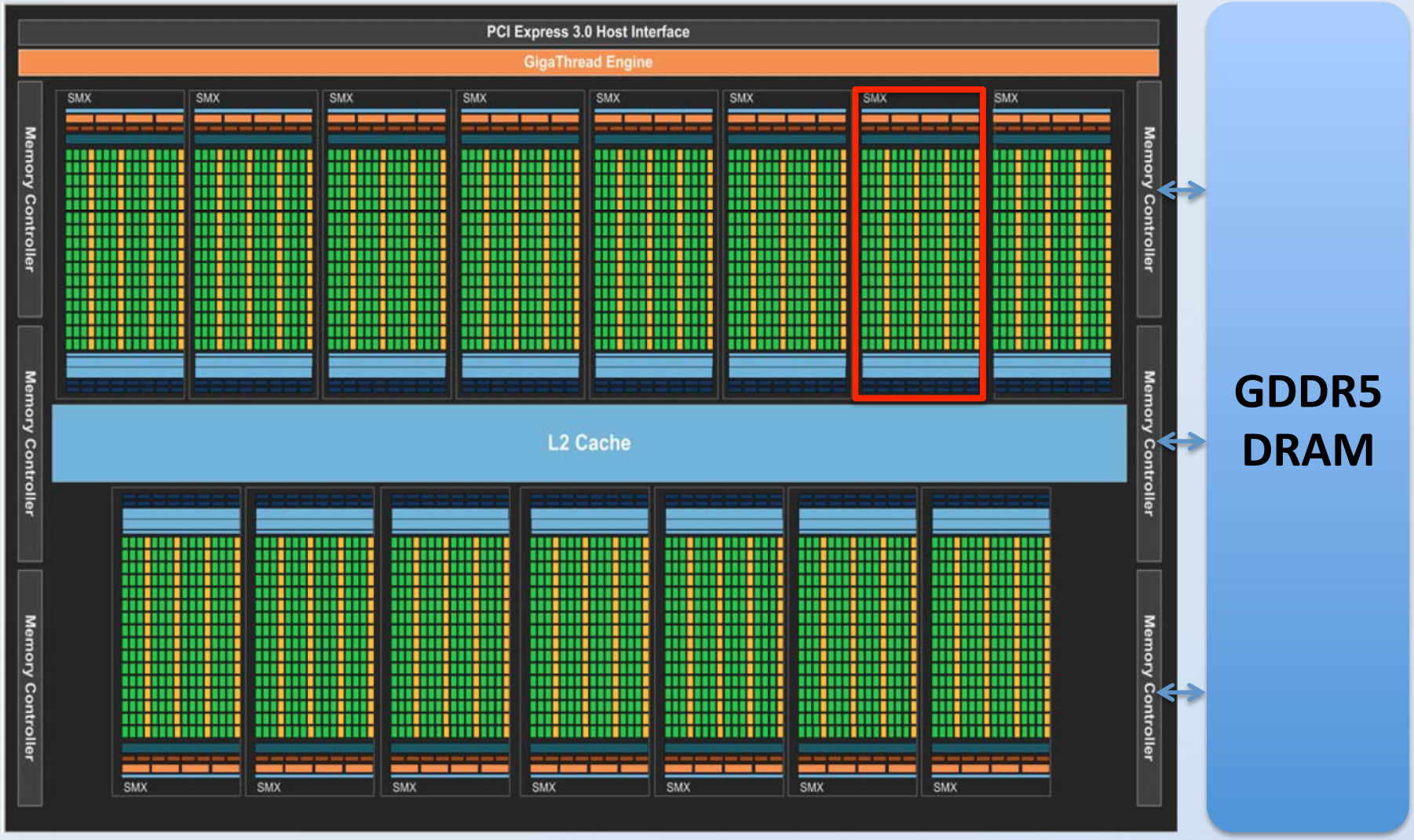


Image Source: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Diagram Of An SMX Multiprocessor

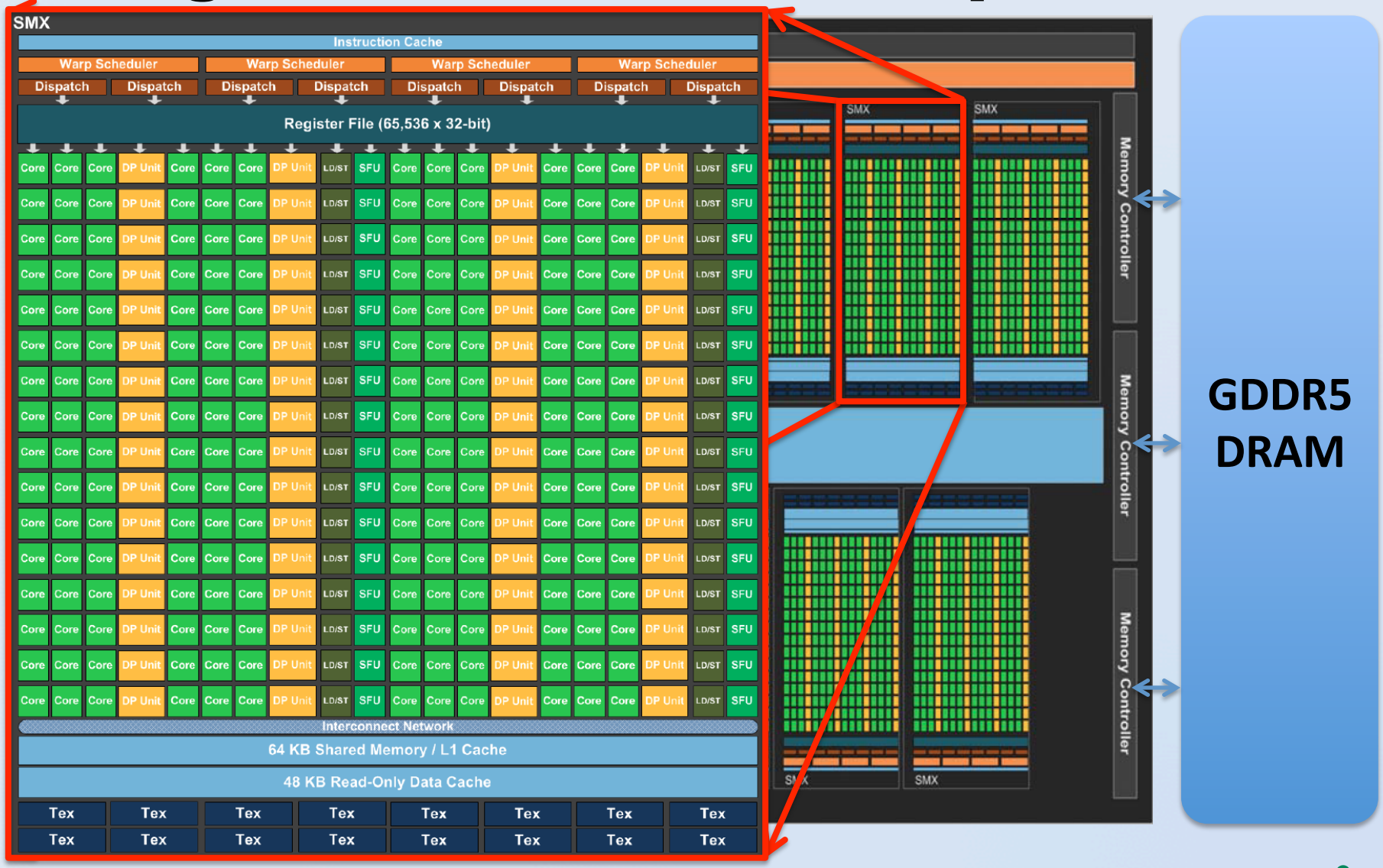
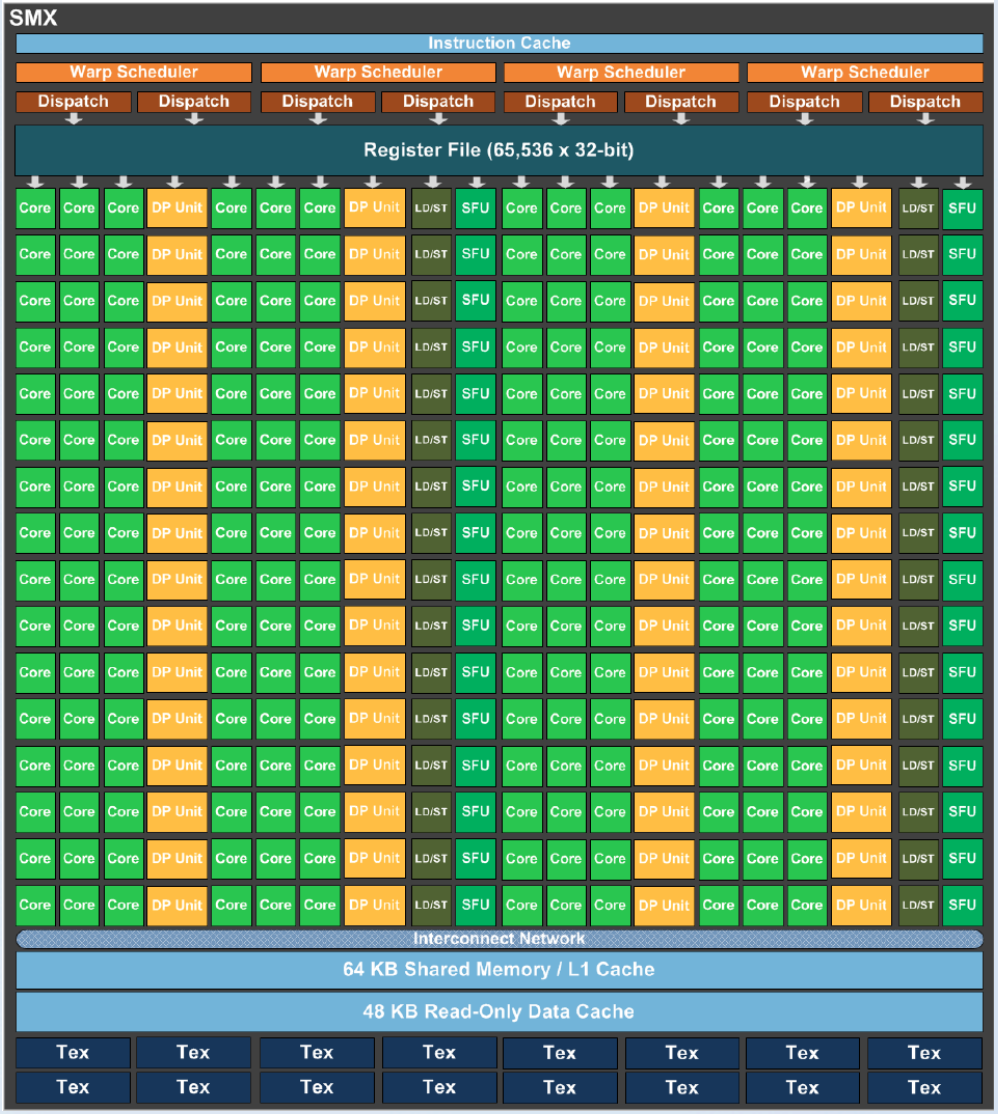


Image Source: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Diagram Of An SMX Multiprocessor



- **192** single-precision cores
- **64** double-precision cores
- Only 64KB general cache
  - Less than 1 MB per GPU!
- 64K registers
  - Can hold 32K doubles
- Threads launched over cores
  - May have >1 thread per core
  - Threads synchronize and share registers & cache only within SMXs not between them

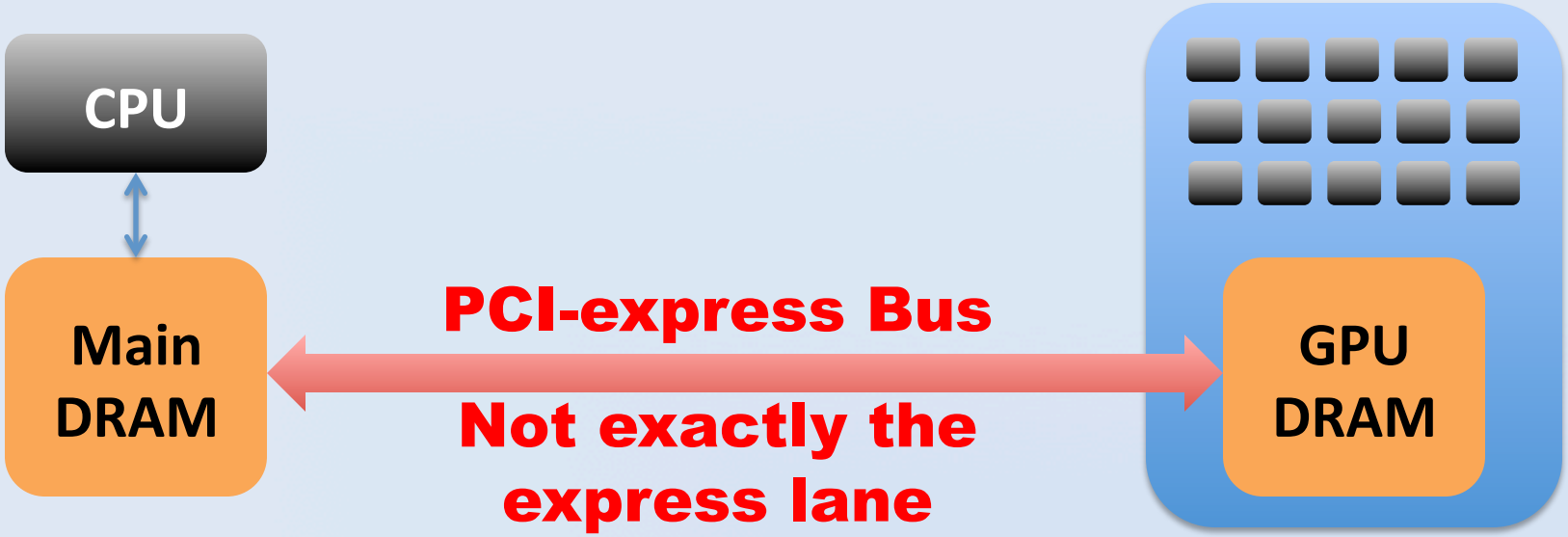
Image Source: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

OLCF
Oak Ridge Leadership Computing Facility

GPUs: Massively Multicore Chips

- 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
  - This also doesn’t take into account **latency**
    - But latency effects are greatly reduced by thread switching

The Dreaded PCI-e Bus



- The PCI-express bus connects main DRAM & GPU DRAM
- Painfully low bandwidth
  - 1,296 & 864 times slower than peak flops in SP & DP
- Also very high latency for small transfers

The Optimization Hierarchy

- Most importantly, minimize and / or overlap PCI-e transfers
  - Usually, you overlap w/ MPI or with independent CPU or GPU code

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

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
- Next, 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

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
- Next, 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
- Next, cache reused data in "shared" memory when possible
  - Worst case: shared memory is 8x slower than registers
- Other optimizations we don't have time to cover

Common GPU Fallacies

- "Flops are free"
  - Because peak flops >> peak bandwidth
  - However, it is very rare in dycores to add flops without adding data
  - Flops require data, data's not free, so flops aren't really free

Common GPU Fallacies

- "Flops are free"
  - Because peak flops >> peak bandwidth
  - However, it is very rare in dycores to add flops without adding data
  - Flops require data, data's not free, so flops aren't really free
- GPUs perform 200x faster than CPUs
  - Did you compare 4 highly optimized single-precision GPUs against a single unoptimized double precision CPU core? Did you ignore PCI-e and MPI communication times?
  - Apples to apples, you'll rarely see more than 5-10x at the high end
  - Still, even 3-4x is a great result

Common GPU Fallacies

- "Flops are free"
  - Because peak flops >> peak bandwidth
  - However, it is very rare in dycores to add flops without adding data
  - Flops require data, data's not free, so flops aren't really free
- GPUs perform 200x faster than CPUs
  - Did you compare 4 highly optimized single-precision GPUs against a single unoptimized double precision CPU core? Did you ignore PCI-e and MPI communication times?
  - Apples to apples, you'll rarely see more than 5-10x at the high end
  - Still, even 3-4x is a great result
- Flops per Watt is orders of magnitude better with GPUs
  - Maybe by peak performance, but that's irrelevant
  - Maybe by the Top500 benchmark, but that's often irrelevant
  - What is the **science** per Watt? Modest improvements are still a big win

Outline

- Brief Overview of Supercomputer Architecture
- CPUs and Data Movement
- Introduction to GPUs and the Challenges
- **Coding For GPUs**
- Implications for Atmospheric Dycores
- Discussion & Questions

Coding For GPUs: Introduction

- GPUs ideally perform the same operation on many data
  - Think of a "thread" as a sequence of operations
- GPU "kernel" gives the operations done by 1 arbitrary thread
  - These operations are done identically by millions of threads
  - Each thread chooses different data based on a unique ID
- Instead of loops, you launch kernels with millions of threads
  - Each thread will have a unique index in 5 dimensions
    - Tx, Ty, & Tz (thread indices within a block) and Bx & By (block indices)
  - The "Tx" index varies the fastest (innermost loop)
  - The "By" index varies the slowest (outermost loop)
- Threads execute in groups of 32 called "warps"
  - All threads in a warp execute the same instruction before moving onto the next instruction
  - A strong contrast to CPU threads, which execute independently

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

```
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)
```

Outermost loop indexed as “blocks”

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)
```

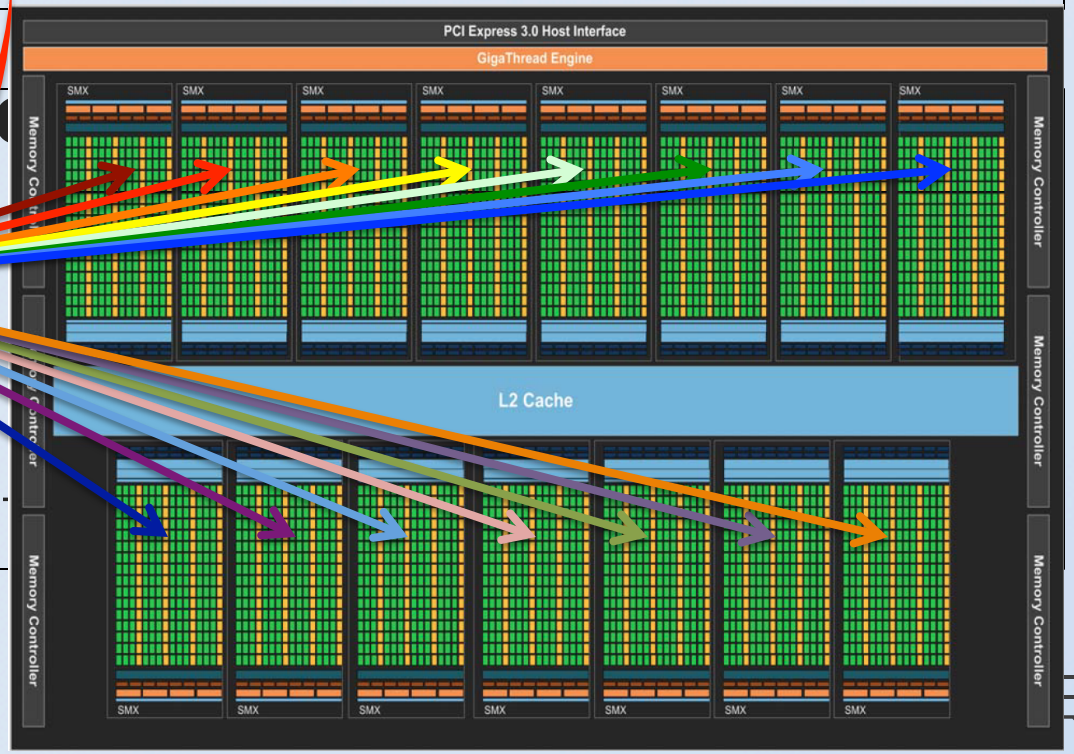
Prototypical Loop Transformation

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(
```



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

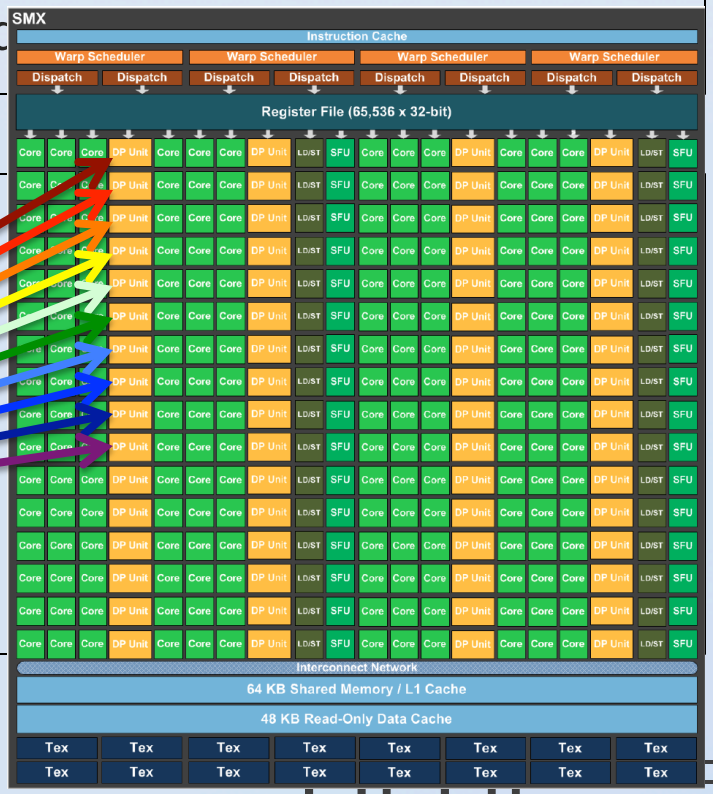
CPU Code

```
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
```

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)
```



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,…)`

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,
     coefs(
     coefs(
     coefs(
```

Gained a 2x speed-up on GPUs from this alone

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,…)`

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, and Intel MIC

Various GPU Coding Options

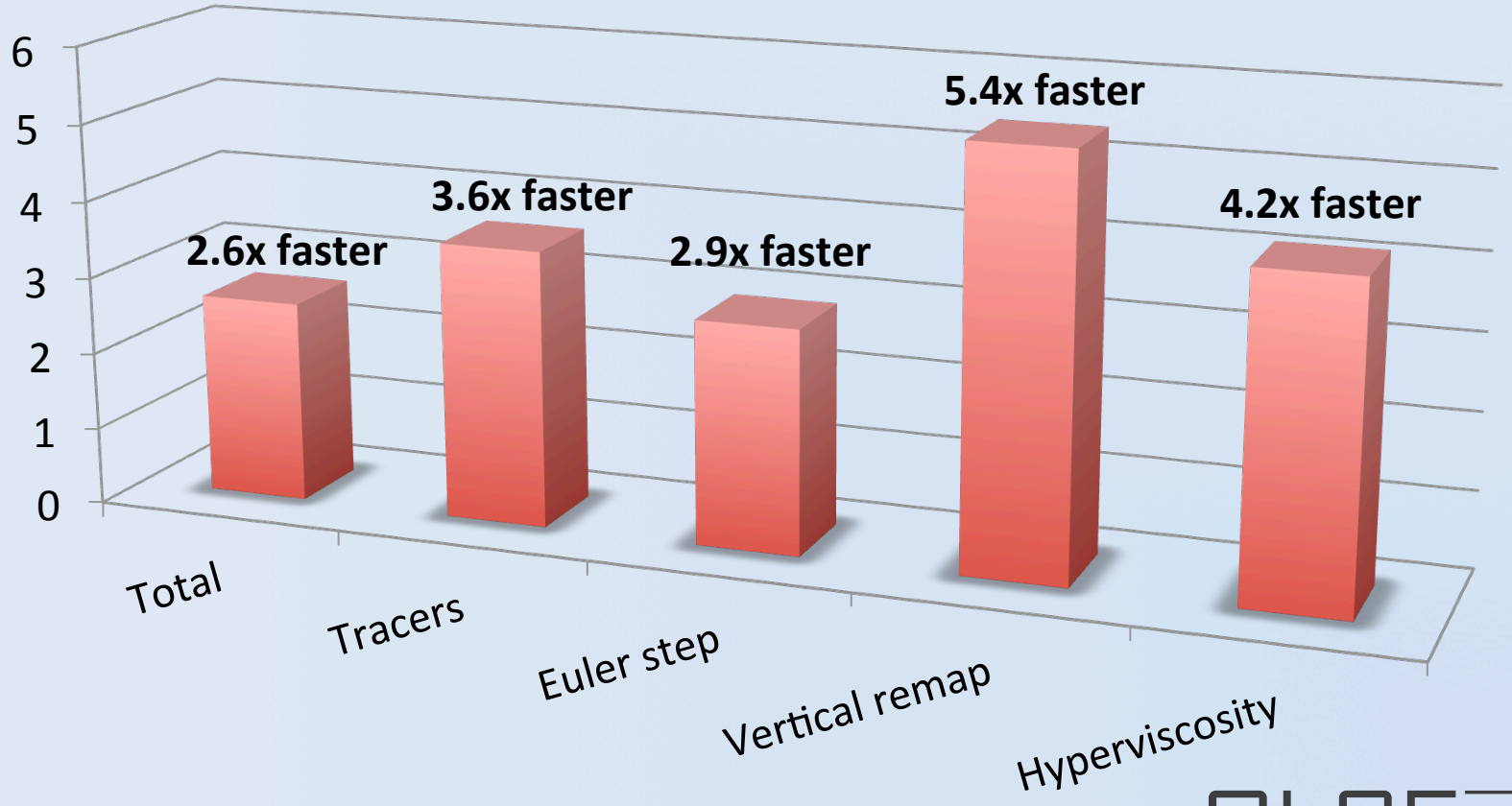
- CUDA and CUDA FORTRAN
  - Similar to the above examples, lower level, hand-optimized
  - Likely the best option for “hot spots” in your code
- OpenCL (a little more cumbersome than CUDA)
  - Works on ATI & Nvidia GPUs, multi-core processors, and Intel MIC
- Libraries (BLAS, LAPACK, FFT, etc)
  - Atmospheric dycores don’t often use much linear algebra or Newton-Krylov
  - But if yours does, then this is your best bet
  - Some physics packages will be able to make use of these

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, and Intel MIC
- Libraries (BLAS, LAPACK, FFT, etc)
  - Atmospheric dycores don’t often use much linear algebra or Newton-Krylov
  - But if yours does, then this is your best bet
  - Some physics packages will be able to make use of these
- Directives (A good option going forward)
  - Like OpenMP in nature, more sustainable software development practices
  - They are currently in their infancy, and the API is evolving
  - They are not automatic, you will have to change your code
    - Luckily these changes usually improve CPU performance as well

CAM-SE: Fermi GPU vs 1 Interlagos / Node

- Benchmarks performed on XK6 using end-to-end wall timers
- Port used CUDA FORTRAN for tracer transport only
- CAM-SE with MOZART Chemistry (106 tracers) at 14km



Why Was Hyperviscosity So Fast?

- Euler Step is called before everything else in tracer transport
- During Euler Step's GPU computations, we perform all initialization efforts for the subsequent kernels
  - Thus Vertical Remap and Hyperviscosity are much faster
  - Euler Step and Hyperviscosity are extremely similar
  - So the speed-up difference gives the advantage of overlapping computation where possible

Why Was Vertical Remap So Fast?

- Originally used splines for reconstruction
  - Splines require a linear solve → vertical dependence within loops
  - Vertical index could not be threaded, only horizontal
- We replaced reconstruction with Piecewise Parabolic Method
  - Vertically independent → vertical index was threaded → 30x more threads

Why Was Vertical Remap So Fast?

- Originally used splines for reconstruction
  - Splines require a linear solve → vertical dependence within loops
  - Vertical index could not be threaded, only horizontal
- We replaced reconstruction with Piecewise Parabolic Method
  - Vertically independent → vertical index was threaded → 30x more threads
- Original remapping used a summation to reduce flops
  - Summations are vertically dependent and harder to thread
- We changed it to do two integrations instead
  - This double the work for remapping (small compared to PPM though)
  - But it also reduced data requirements and dependence

Why Was Vertical Remap So Fast?

- Originally used splines for reconstruction
  - Splines require a linear solve → vertical dependence within loops
  - Vertical index could not be threaded, only horizontal
- We replaced reconstruction with Piecewise Parabolic Method
  - Vertically independent → vertical index was threaded → 30x more threads
- Original remapping used a summation to reduce flops
  - Summations are vertically dependent and harder to thread
- We changed it to do two integrations instead
  - This double the work for remapping
  - But it also reduced data requirements and dependence
- As a result, all data in the reconstruction and remap fit into cache
  - Only accesses to DRAM were at the very beginning and end of kernel with a lot of work in between, all done in-cache
  - Thus, >5x speed-up over PPM remap on CPU

Why Was Vertical Remap So Fast?

- Originally used splines for reconstruction
  - Splines require a linear solve → vertical dependence within loops
  - Vertical index could not be threaded, only horizontal
- We re hod
  - Ve e threads
- Origi
  - Su
- We c
  - Th
  - Bu
- As a re cache
  - Only accesses to DRAM were at the very beginning and end of kernel with a lot of work in between
  - Thus, >5x speed-up over PPM remap on CPU

- **If Increasing The Workload**
  - **Allows More Threading**
  - **Decreases Data Dependence**
  - **Decreases Local Data Requirements**
- **Then It's Worth Investigating**

Outline

- Brief Overview of Supercomputer Architecture
- CPUs and Data Movement
- Introduction to GPUs and the Challenges
- Coding For GPUs
- **Implications for Atmospheric Dycores**
- Discussion & Questions

How To Get Peak Flops

- How to make your code reach peak performance on GPUs

How To Get Peak Flops

- How to make your code reach peak performance on GPUs
  - Step 1: Formulate your problem as time-implicit

How To Get Peak Flops

- How to make your code reach peak performance on GPUs
  - Step 1: Formulate your problem as time-implicit
  - Step 2: Maximize the problem size per node and decompose

How To Get Peak Flops

- How to make your code reach peak performance on GPUs
  - Step 1: Formulate your problem as time-implicit
  - Step 2: Maximize the problem size per node and decompose
  - Step 3: Pretend the system is dense

How To Get Peak Flops

- How to make your code reach peak performance on GPUs
  - Step 1: Formulate your problem as time-implicit
  - Step 2: Maximize the problem size per node and decompose
  - Step 3: Pretend the system is dense
  - Step 4: Do an L-U decomposition with partial pivoting: $O(N^3)$

How To Get Peak Flops

- How to make your code reach peak performance on GPUs
  - Step 1: Formulate your problem as time-implicit
  - Step 2: Maximize the problem size per node and decompose
  - Step 3: Pretend the system is dense
  - Step 4: Do an L-U decomposition with partial pivoting: $O(N^3)$
  - Step 5: Watch climate evolve faster than your simulation

How To Get Peak Flops

- How to make your code reach peak performance on GPUs
  - Step 1: Formulate your problem as time-implicit
  - Step 2: Maximize the problem size per node and decompose
  - Step 3: Pretend the system is dense
  - Step 4: Do an L-U decomposition with partial pivoting: $O(N^3)$
  - Step 5: Watch climate evolve faster than your simulation
- The Point: Flops do not equate to efficiency
  - Time step matters, cost per time step matters
  - Throughput and scaling matter

How To Get Peak Flops

- How to make your code reach peak performance on GPUs
  - Step 1: Formulate your problem as time-implicit
  - Step 2: Maximize the problem size per node and decompose
  - Step 3: Pretend the system is dense
  - Step 4: Do an L-U decomposition with partial pivoting: $O(N^3)$
  - Step 5: Watch climate evolve faster than your simulation
- The Point: Flops do not equate to efficiency
  - Time step matters, cost per time step matters
  - Throughput and scaling matter
  - Most of all, **accuracy** matters
    - Resolution, damping, oscillations, consistency, tracer correlations, conservation, maintaining balances, isotropy, coupling, etc

GPU-Friendly Algorithmic Properties

- Problem size should be in multiples of the GPU resources
  - E.g., CAM-SE: 4x4 bases, 32 levels, 64 elements / node, 128 tracers, etc

GPU-Friendly Algorithmic Properties

- Problem size should be in multiples of the GPU resources
  - E.g., CAM-SE: 4x4 bases, 32 levels, 64 elements / node, 128 tracers, etc
- Avoid irregularly divergent if-then logic & gotos
  - All threads in a warp execute exactly the same instruction
    - If threads in a warp take two branches, both branches get executed

GPU-Friendly Algorithmic Properties

- Problem size should be in multiples of the GPU resources
  - E.g., CAM-SE: 4x4 bases, 32 levels, 64 elements / node, 128 tracers, etc
- Avoid irregularly divergent if-then logic & gotos
  - All threads in a warp execute exactly the same instruction
    - If threads in a warp take two branches, both branches get executed
- Plenty of data-independent, fine-scale parallelism
  - Typically found in nested loops, usually there if you look for it
  - All time-explicit stencil calculations (FV, Galerkin, etc) have this
  - However, you should avoid dependency within those loops (i.e., an iteration depends on data computed in the previous iteration)
  - Inversions, reductions, & summations exhibit dependency: spline coefficients, time-implicit schemes, hydrostatic summations, energy fixers, positivity filters
    - Usually cannot thread vertical index, often giving too few threads per GPU
- **More data per node is always a good thing**

We Need More Data Per Node

- Time-explicit time steps reduce linearly with grid (h-)refinement
  - 2x horizontal grid refinement requires 8x more work (2x smaller time step)
  - Yet it only introduces 4x more data
  - Keeping same throughput, 4x more divided over > 8x more processors

We Need More Data Per Node

- Time-explicit time steps reduce linearly with grid (h-)refinement
  - 2x horizontal grid refinement requires 8x more work (2x smaller time step)
  - Yet it only introduces 4x more data
  - Keeping same throughput, 4x more divided over > 8x more processors
- We need alternative means of increasing the data per node
  - One option is to transport more tracers with more advanced physics
    - Our team at OLCF did this for CAM-SE, using active chemistry
    - CAM4: 3 tracers ; CAM5: 26 tracers ; full chemistry: 108 tracers
  - Another option is to increase the number of ensemble members
    - More spatial resolution and more physics means more uncertainty
    - I think this is a very useful option!

We Need More Data Per Node

- Time-explicit time steps reduce linearly with grid (h-)refinement
  - 2x horizontal grid refinement requires 8x more work (2x smaller time step)
  - Yet it only introduces 4x more data
  - Keeping same throughput, 4x more divided over > 8x more processors
- We need alternative means of increasing the data per node
  - One option is to transport more tracers with more advanced physics
    - Our team at OLCF did this for CAM-SE, using active chemistry
    - CAM4: 3 tracers ; CAM5: 26 tracers ; full chemistry: 108 tracers
  - Another option is to increase the number of ensemble members
    - More spatial resolution and more physics means more uncertainty
    - I think this is a very useful option!
- There's another means of increasing data per node: algorithms
  - Faster runtime or less comm → fewer nodes → more data per node

We Need New Algorithms

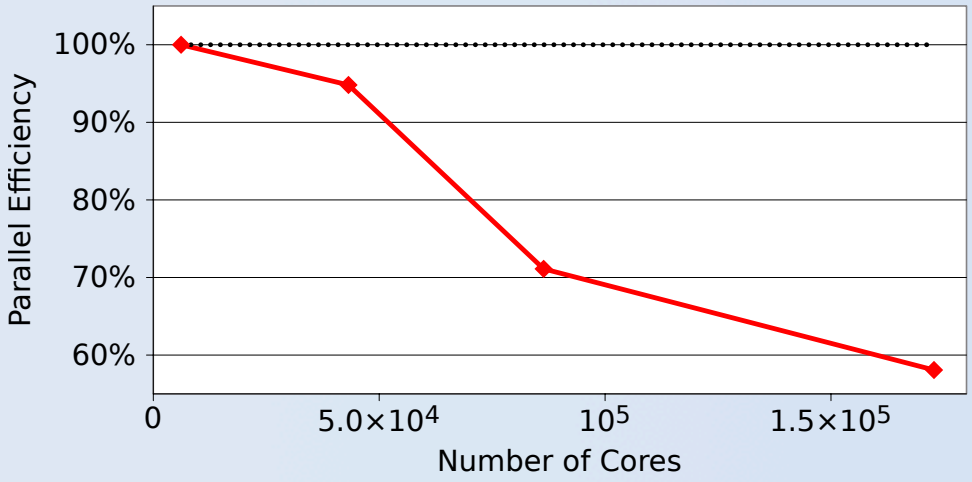
- CAM-SE is an attractive and very scalable dycore
  - No data exchanges for reconstruction (entirely local)
  - Minimal data exchanges for boundary averaging
  - Very cheap per time step

We Need New Algorithms

- CAM-SE is an attractive and very scalable dycore
  - No data exchanges for reconstruction (entirely local)
  - Minimal data exchanges for boundary averaging
  - Very cheap per time step
- Pitfalls of CAM-SE
  - R-K method should be same order as spatial
  - Time step is very small due to variational form of PDEs
  - Hyperdiffusion (subcycled) incurs significant excess communication
  - 3-Stage Runge-Kutta requires communication between each stage

We Need New Algorithms

- CAM-SE is an attractive and very scalable dycore
  - No data exchanges for reconstruction (entirely local)
  - Minimal data exchanges for boundary averaging
  - Very cheap per time step
- Pitfalls of CAM-SE
  - R-K method should be same order as spatial
  - Time step is very small due to variational form of PDEs
  - Hyperdiffusion (subcycled) incurs significant excess communication
  - 3-Stage Runge-Kutta requires communication between each stage



14 km CAM-SE Scaling: XT5

On 172,800 cores, MPI waiting alone consumes 42% of total model time

We Need New Spatial Methods

- New Spatial Operators
  - Spectral Finite-Volume (Cheruvu et al, ANM, 2007)
    - Time step decreases less rapidly than Galerkin during p-refinement
  - Creative combinations of different moments (Ii & Xiao, JCP, 2007)
    - Constrained Interpolation Profile (CIP) that evolves point values, derivatives, and cell means – all in the same method
  - Multi-Moment Finite-Volume (Prather,JGR, 1986; Norman & Finkel, JCP, 2012)
    - Time step constant (CFL= ½ in 2-D) during high-order p-refinement
- "New" Limiting Procedures
  - Weighted Essentially Non-Oscillatory (WENO) & Hermite WENO (Liu et al, JCP, 1994 ; Qiu & Shu, JCP, 2004)
    - Robust even for shocks, done once per time step, HWENO low-comm
  - Flux-Corrected Transport (Boris & Book, JCP, 1973 ; Zalezak, JCP, 1979)
    - Easily adapted to non-structured grids & cubed sphere

We Need New Temporal Methods

- New Time Integration Methods
  - Semi-Lagrangian (SL) Finite-Volume & SL Galerkin for transport
    - Much larger time step, transport only (Lauritzen et al, JCP, 2010 ; Bonaventura et al, Comm SIAM Congr, 2006)
  - Characteristic Flux-Form Semi-Lagrangian (Ii and Xiao, JCP, 2007 ; Norman et al, JCP, 2011)
    - One-stage, one-step, very large time step
  - Arbitrary-order DErivative Riemann (ADER) & ADER-Continuous Galerkin
    - One-stage, One-step, low-memory, fully non-linear, very high-order
    - And now very cheap (Norman & Finkel, JCP, 2012)
    - Easily adapted for mesh refinement local time stepping

Your Contributions Here

Outline

- Brief Overview of Supercomputer Architecture
- CPUs and Data Movement
- Introduction to GPUs and the Challenges
- Coding For GPUs
- Implications for Atmospheric Dycores
- **Discussion & Questions**

Communication-Reducing Algorithms

- Different algorithms require different communication amounts
- Time-implicit & elliptic splittings
  - Require significant global communications and reductions
  - Can only scale to 10,000 nodes with maximal problem size per node
  - These problem sizes rarely give feasible atmospheric throughputs
- (Horizontally) Time-explicit methods
  - On cubed-sphere and icosahedral grids, they scale quite well
  - Significant flexibility within time-explicit methods for time step size
  - Galerkin methods require minimal data exchange per time step
    - Yet the time step is extremely small
  - Multi-moment finite-volume & spectral volume are promising
  - Runge-Kutta requires multiple data exchanges per time step
    - Flux-Form Semi-Lagrangian & ADER do not

Ways CPUs Improve Performance

- **Pipelining**: Just like the Ford factory's assembly line
  - Break instructions into many pieces
  - Different operations on different instructions is parallel
- **Instruction Prefetching**
  - CPU is too fast to wait for instructions from slow memory
- **Branch Prediction**
  - Enables prefetching and pipelining across if and "go to" logic
  - If you're wrong, flush the pipeline and restart
- **Out of Order Execution**
  - Analyze dependence in a sequence of instructions
  - While waiting on data, perform any independent computations

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