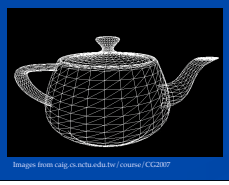
GPU Architectures for Non-Graphics People

David Black-Schaffer
david.black-schaffer@it.uu.se

UPMARC Uppsala Programming for Multicore Architectures Research Center

GPU Background

GPUs: Architectures for Drawing Triangles Fast



- Basic processing:
  - Project triangles into 2D
  - Find the pixels for each triangle
  - Determine color for each pixel
- Where is most of the work?
  - 10k triangles (30k vertices)
    - Project, clip, calculate lighting
  - 1920x1200 = 2.3M pixels
    - 8x oversampling = 18.4M pixels
    - 7 texture lookups
    - 43 shader ops
  - @ 60fps
    - Compute: 47.5 GOPs sustained
    - Memory: 123GB/s sustained
    - 4-core Nehalem: 106 GFLOPs, 32GB/s peak

Images from caig.cs.nctu.edu.tw/course/CG2007

Example Shader: Water

source

Water Shader

- Vectors
- Texture lookups
- Complex math
- Function calls
- Control flow
- But…
- No loops

From http://www2.ati.com/developer/gdc/D3DTutorial10_Half-Life2_Shading.pdf

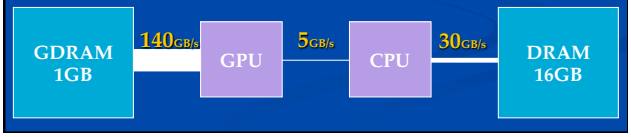
GPGPU: General Purpose GPUs

- Question: Can we use GPUs for non-graphics tasks?
- Answer: Yes!
  - They're incredibly fast and awesome
- Answer: Maybe
  - They're fast, but hard to program
- Answer: Not really
  - My algorithm runs slower on the GPU than on the CPU
- Answer: No
  - I need more precision/memory/synchronization/other

The Achilles Heel of GPUs

GPUs are fast, but…

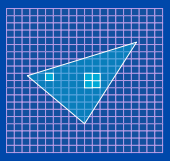
Getting data to the GPU is slow
Getting data to the GPU is slow
Getting data *from* the GPU is slow



GPU Design

GPU Design

1) Process pixels in parallel

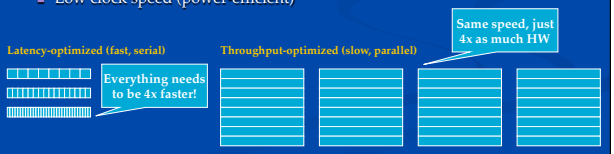


- Data-parallel:
  - 2.3M pixels per frame
    => lots of work
  - All pixels are independent
    => no synchronization
  - Lots of spatial locality
    => regular memory access
    (only need small caches)
- Great speedups
  - Limited only by the amount of hardware

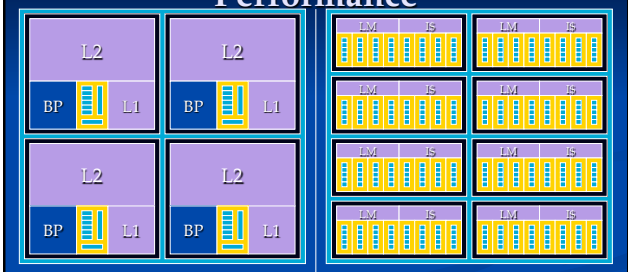
GPU Design

2) Focus on throughput, not latency

- Each pixel can take a long time…
  …as long as we process many at the same time.
- Great scalability
  - Lots of simple parallel processors
  - Low clock speed (power-efficient)



CPU vs. GPU Philosophy: Performance

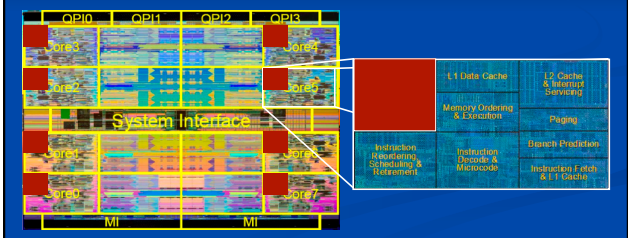


4 Massive CPU Cores: Big caches, branch predictors, out-of-order, multiple-issue, speculative execution, double-precision…
About 2 IPC per core, 8 IPC total @3GHz

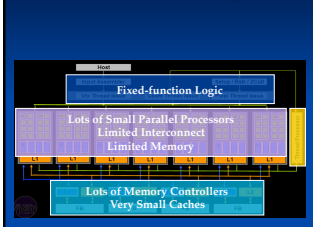
8*8 Wimpy GPU Cores: No caches, in-order, single-issue, single-precision…
About 1 IPC per core, 64 IPC total @1.5GHz

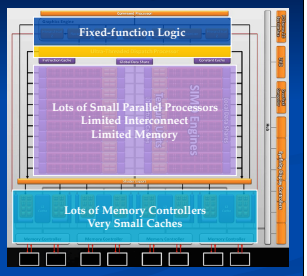
Example CPUs Today

- Intel 8-core 45nm Nehalem (2010)



Example GPUs Today

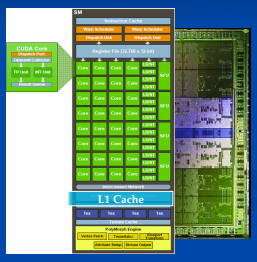




Nvidia G80 AMD 5870

GPU Trends

- More of the same (processors)
- More CPU features (caches, function pointers, double precision)
- More flexible (multiple kernels)



Nvidia Fermi

Images from hothardware.com

Questions So Far?

Architecture

- Memory Philosophy
  - Latency vs. Throughput
- Instruction Bandwidth
  - Divergence

GPU Memory Philosophy

Computational Intensity

- Proportion of **math** ops : unique* **memory** ops
  Remember: memory is slow, math is fast
- Loop body: Low-intensity:

```
A[i] = B[i] + C[i]          1:3
A[i] = B[i] + C[i] * D[i]   2:4
A[i]++                      1:2
```

- Loop body: High(er)-intensity:

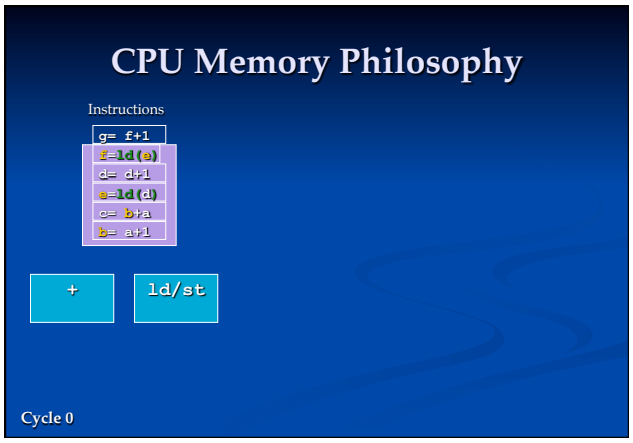
```
Temp+= A[i]*A[i]            2:1
A[i] = exp(temp)*erf(temp)  X:1
```

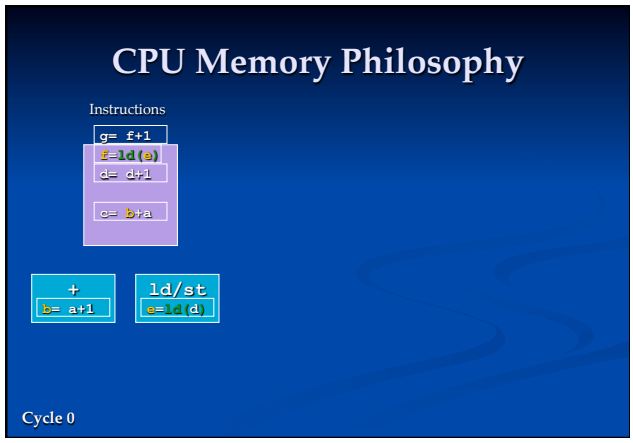
*Unique: not already in the cache.

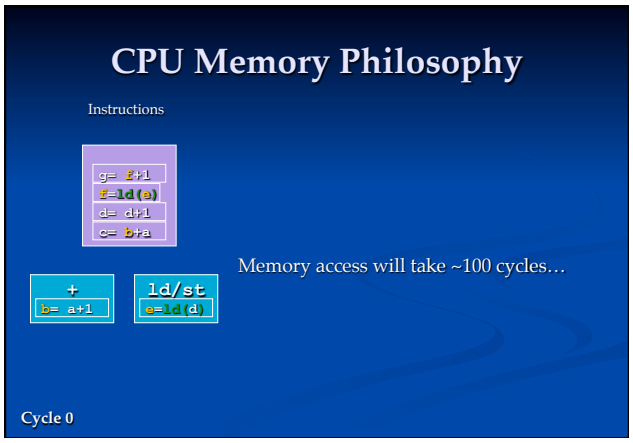
CPU Memory Philosophy

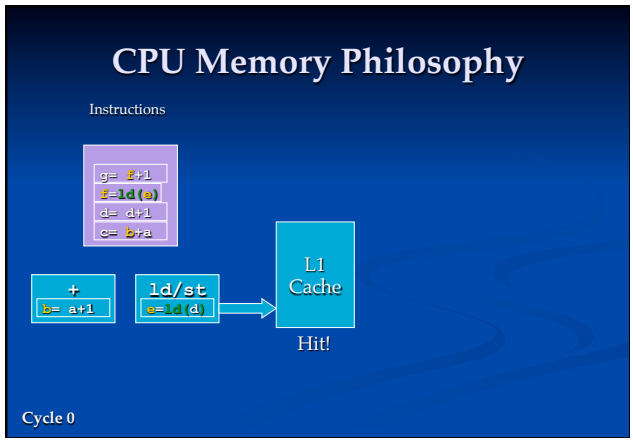
Instructions

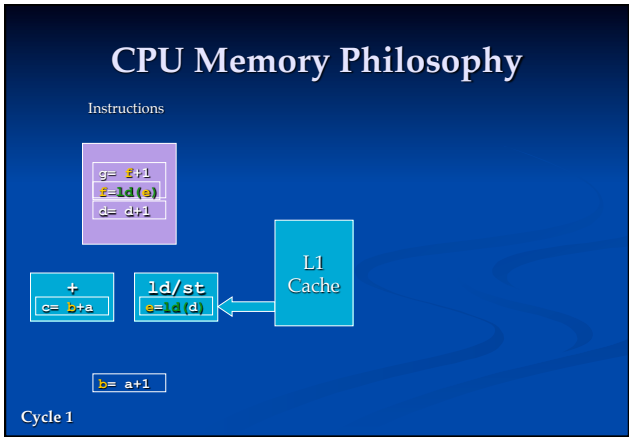
```
g= f+1
f=ld(e)
d= d+1
e=ld(d)
c= b+a
b= a+1
```

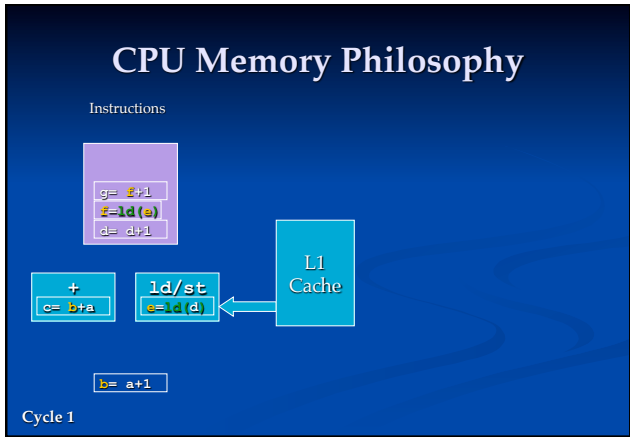

CPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
e=ld(d)
c= b+a
b= a+1
+
ld/st
Cycle 0

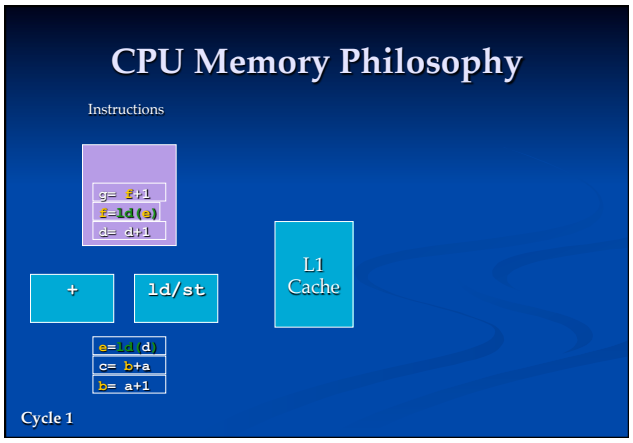

CPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
c= b+a
+
b= a+1
ld/st
e=ld(d)
Cycle 0

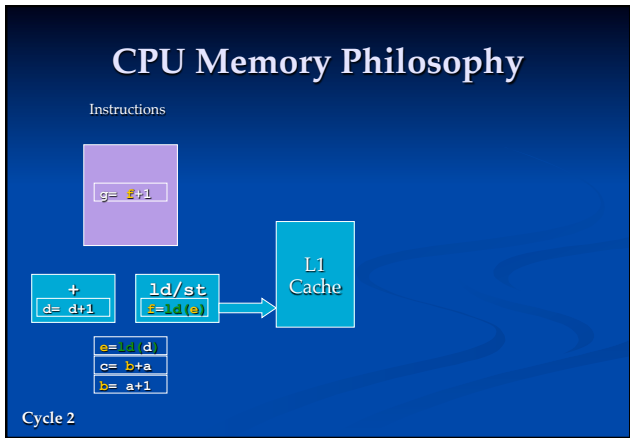

CPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
c= b+a
+
b= a+1
ld/st
e=ld(d)
Memory access will take ~100 cycles…
Cycle 0

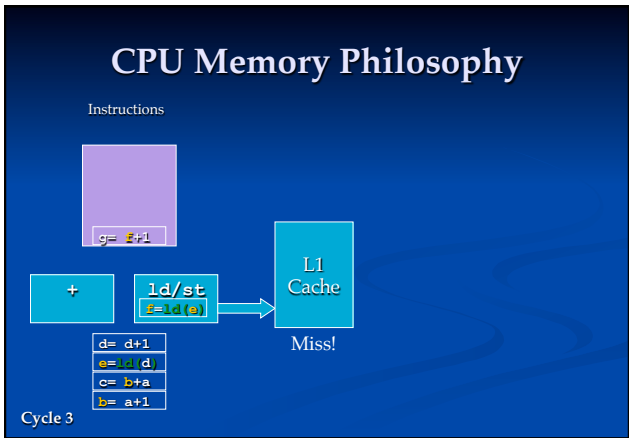

CPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
c= b+a
+
b= a+1
ld/st
e=ld(d)
L1
Cache
Hit!
Cycle 0

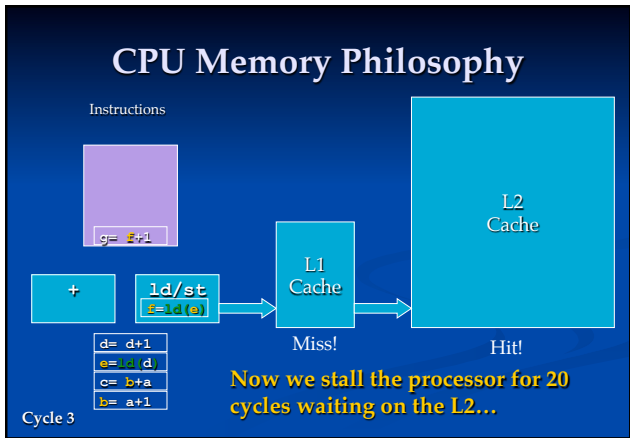

CPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
+
c= b+a
ld/st
e=ld(d)
L1
Cache
b= a+1
Cycle 1

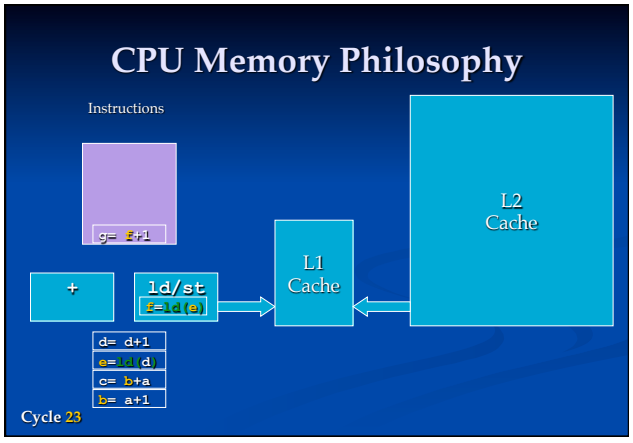

CPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
+
c= b+a
ld/st
e=ld(d)
L1
Cache
b= a+1
Cycle 1

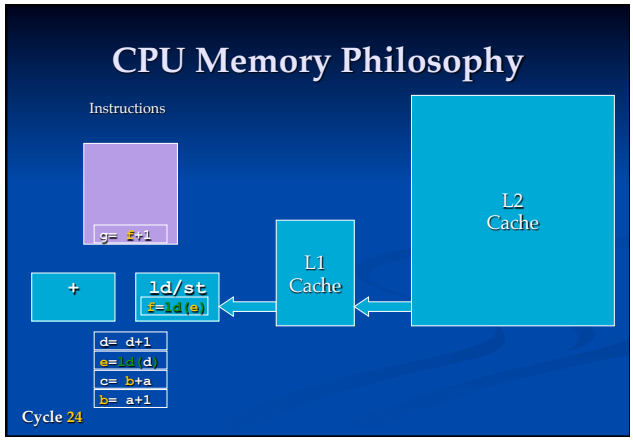

CPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
+
ld/st
L1 Cache
e=ld(d)
c= b+a
b= a+1
Cycle 1


CPU Memory Philosophy
Instructions
g= f+1
+
d= d+1
ld/st
f=ld(e)
L1 Cache
e=ld(d)
c= b+a
b= a+1
Cycle 2

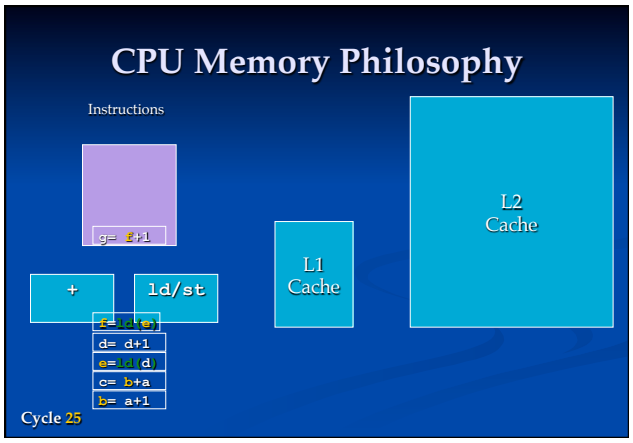

CPU Memory Philosophy
Instructions
g= f+1
+
ld/st
f=ld(e)
L1 Cache
Miss!
d= d+1
e=ld(d)
c= b+a
b= a+1
Cycle 3


CPU Memory Philosophy
Instructions
g= f+1
+
ld/st
f=ld(e)
L1 Cache
L2 Cache
Miss!
Hit!
d= d+1
e=ld(d)
c= b+a
b= a+1
Now we stall the processor for 20 cycles waiting on the L2…
Cycle 3

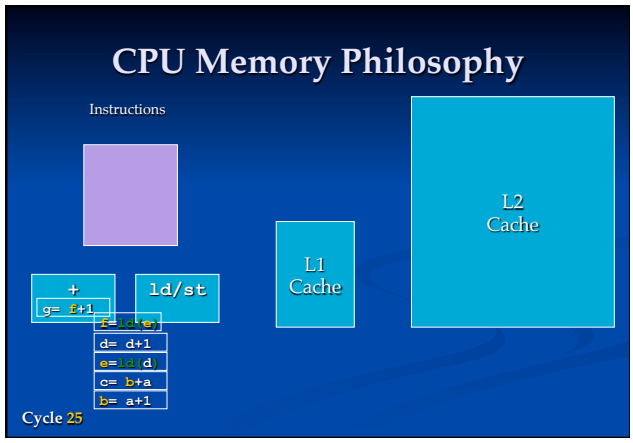

CPU Memory Philosophy
Instructions
g= f+1
+
ld/st
f=ld(e)
L1 Cache
L2 Cache
d= d+1
e=ld(d)
c= b+a
b= a+1
Cycle 23


CPU Memory Philosophy
Instructions
g= f+1
+
ld/st
f=ld(e)
L1 Cache
L2 Cache
d= d+1
e=ld(d)
c= b+a
b= a+1
Cycle 24

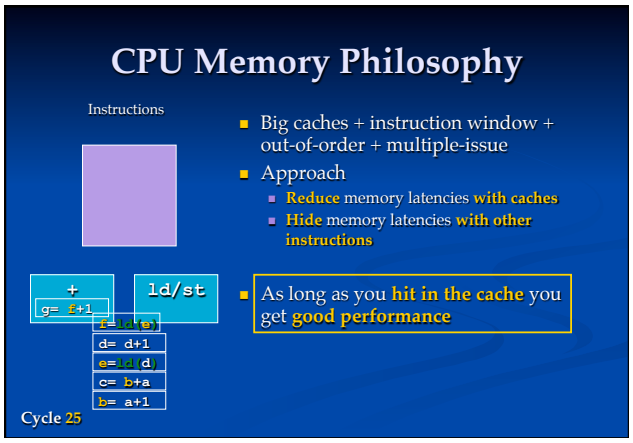
CPU Memory Philosophy



CPU Memory Philosophy

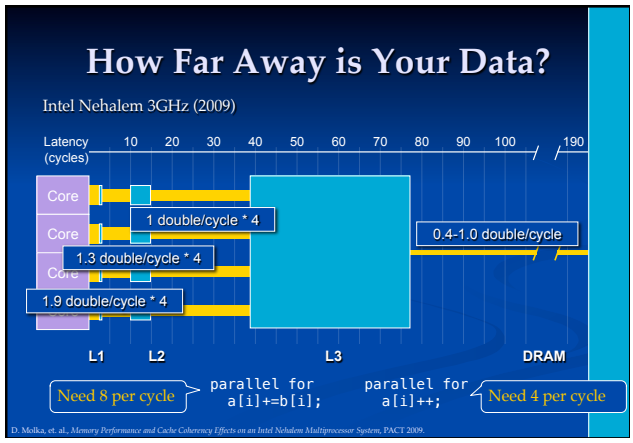


CPU Memory Philosophy

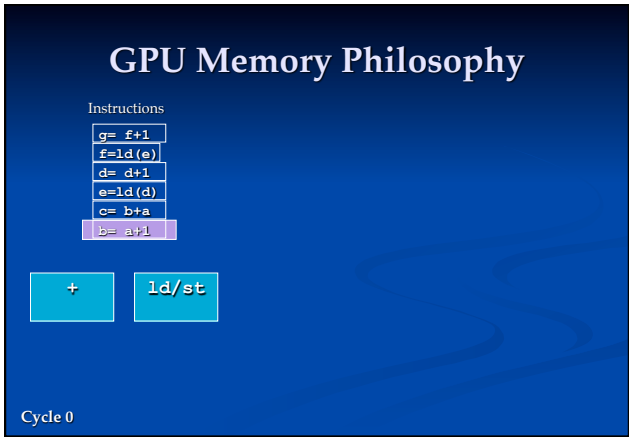


- Big caches + instruction window + out-of-order + multiple-issue
- Approach
  - **Reduce** memory latencies **with caches**
  - **Hide** memory latencies **with other instructions**
- As long as you **hit in the cache** you get **good performance**

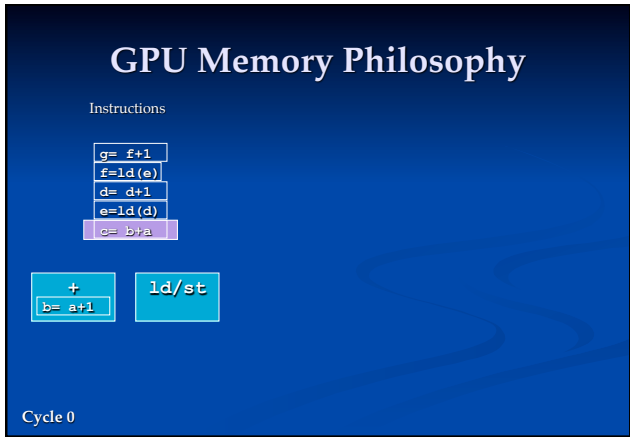
How Far Away is Your Data?

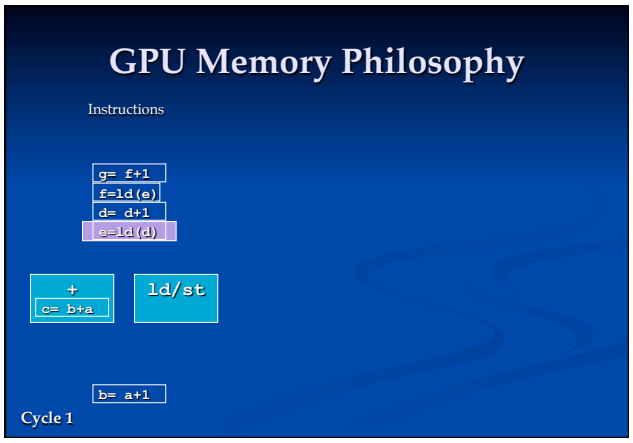


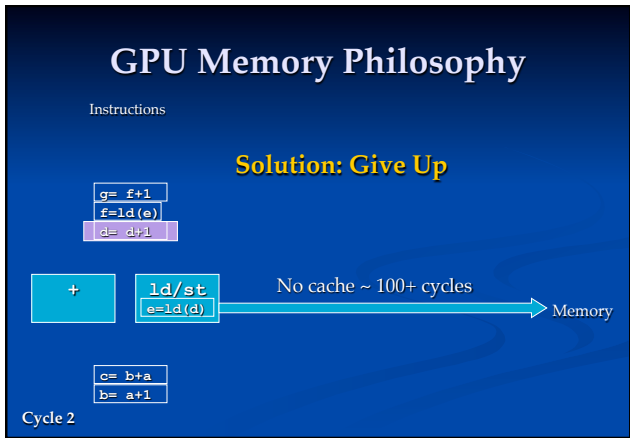
GPU Memory Philosophy

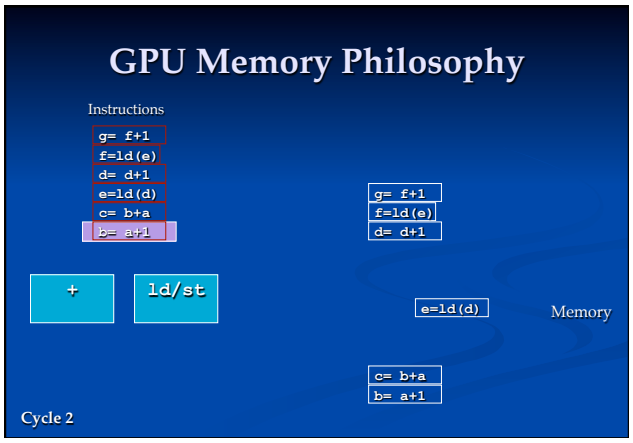


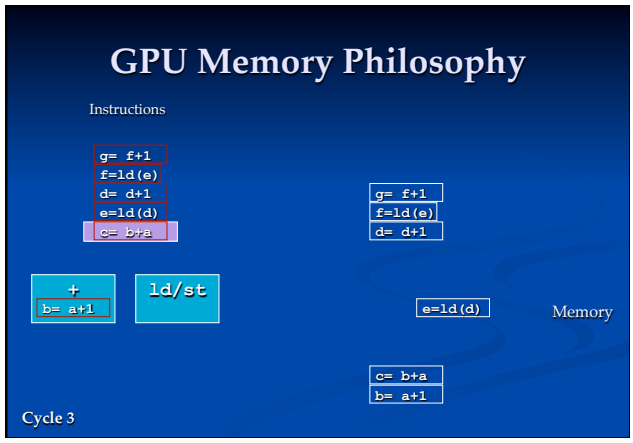
GPU Memory Philosophy

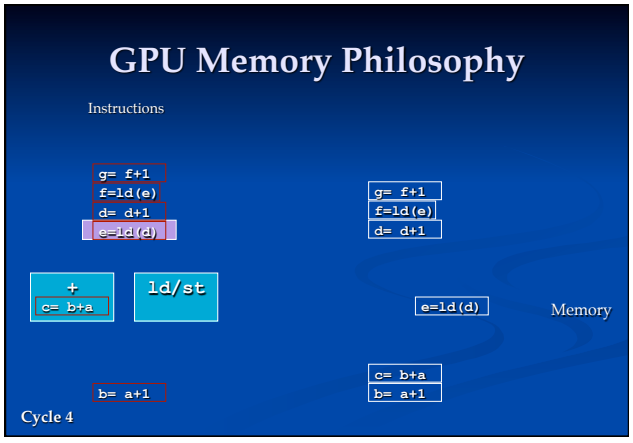


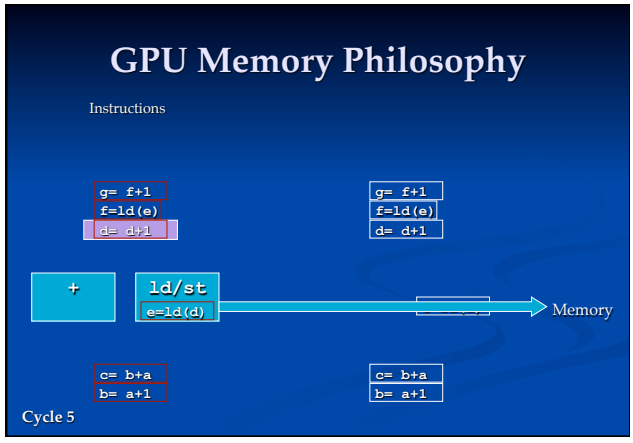

GPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
e=ld(d)
+
c= b+a
ld/st
b= a+1
Cycle 1

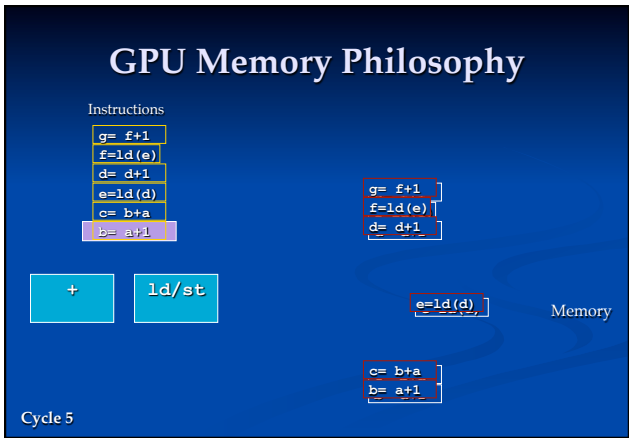

GPU Memory Philosophy
Instructions
Solution: Give Up
g= f+1
f=ld(e)
d= d+1
+
ld/st
e=ld(d)
No cache ~ 100+ cycles
Memory
c= b+a
b= a+1
Cycle 2

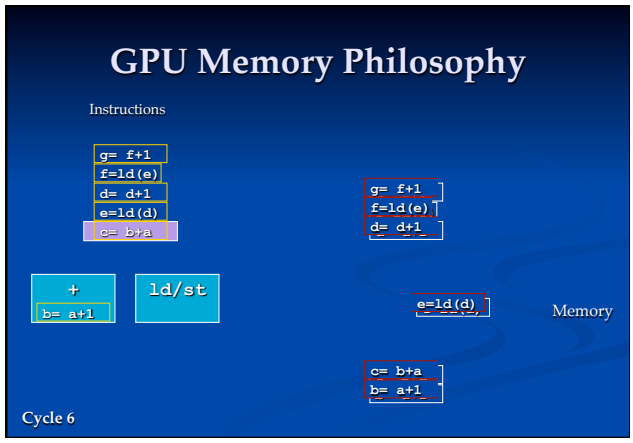

GPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
e=ld(d)
c= b+a
b= a+1
+
ld/st
g= f+1
f=ld(e)
d= d+1
e=ld(d)
Memory
c= b+a
b= a+1
Cycle 2

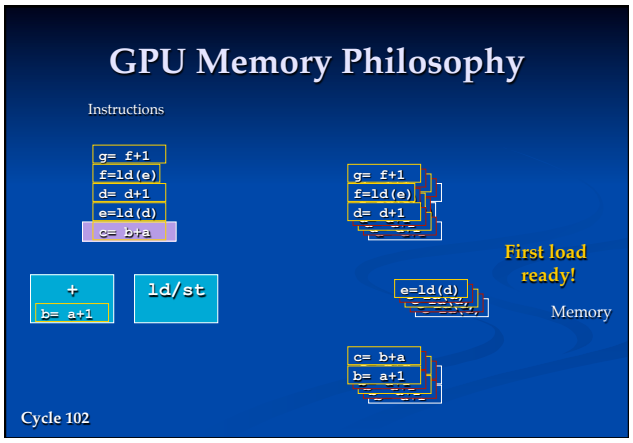

GPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
e=ld(d)
c= b+a
+
b= a+1
ld/st
g= f+1
f=ld(e)
d= d+1
e=ld(d)
Memory
c= b+a
b= a+1
Cycle 3

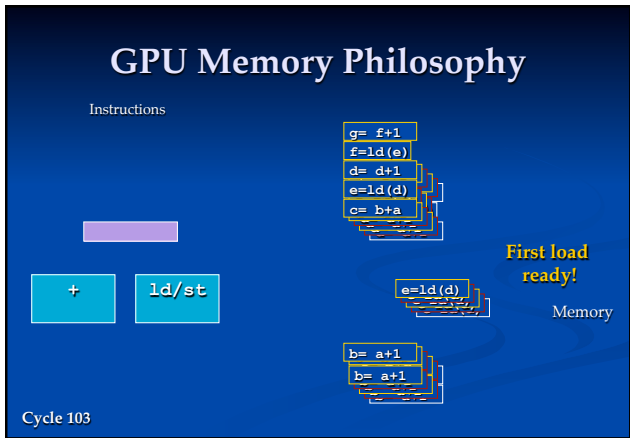

GPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
e=ld(d)
+
c= b+a
ld/st
g= f+1
f=ld(e)
d= d+1
e=ld(d)
Memory
c= b+a
b= a+1
b= a+1
Cycle 4

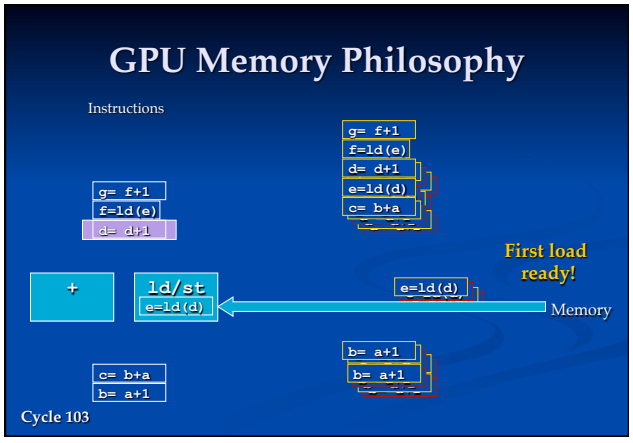

GPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
+
ld/st
e=ld(d)
g= f+1
f=ld(e)
d= d+1
Memory
c= b+a
b= a+1
c= b+a
b= a+1
Cycle 5

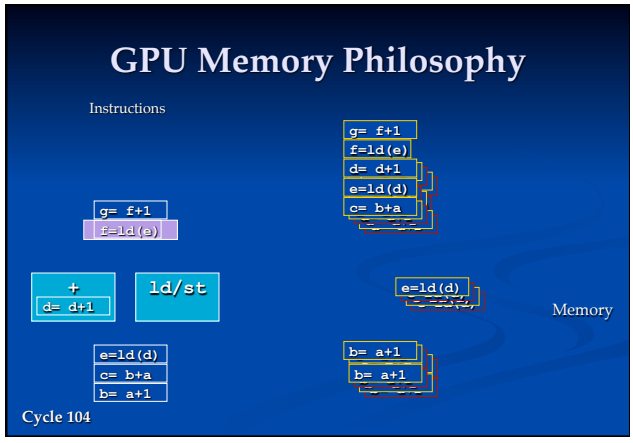

GPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
e=ld(d)
c= b+a
b= a+1
+
ld/st
Memory
Cycle 5


GPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
e=ld(d)
c= b+a
b= a+1
+
ld/st
Memory
Cycle 6


GPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
e=ld(d)
c= b+a
b= a+1
+
ld/st
First load ready!
Memory
Cycle 102

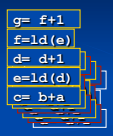

GPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
e=ld(d)
c= b+a
b= a+1
+
ld/st
First load ready!
Memory
Cycle 103


GPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
e=ld(d)
c= b+a
b= a+1
+
ld/st
First load ready!
Memory
Cycle 103

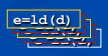

GPU Memory Philosophy
Instructions
g= f+1
f=ld(e)
d= d+1
e=ld(d)
c= b+a
b= a+1
+
ld/st
Memory
Cycle 104

GPU Memory Philosophy

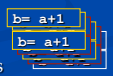
- Thousands of hardware threads
- 1 cycle context switching
- Hardware thread scheduling



- As long as there is **enough work** in other threads **to cover latency** you get **high throughput**.



Notes:
- GPUs have caches for textures
- Newer GPUs have (very small) data caches
- To get full bandwidth you need good access patterns

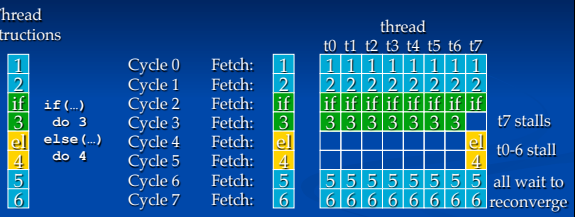


GPU Instruction Bandwidth

GPU Instruction Bandwidth

- GPU compute units fetch 1 instruction per cycle…
  …and share it with 8 processor cores.
- What if they don't all want the same instruction? (divergent execution)

Divergent Execution



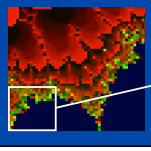
Divergent execution can dramatically hurt performance. Avoid it on GPUs today.

Divergent Execution for Real

Per-pixel Mandelbrot calculation:

```
while (x*x + y*y <= (4.0f) && iteration < max_iterations) {
 float xtemp = x*x - y*y + x0;
 y = 2*y*x + y0;
 x = xtemp;
 iteration++;
}
color = iteration;
```

Color determined by iteration count…
…each color took a different number of iterations.



Every different color is a divergent execution.

Instruction Divergence

- Some architectures are worse…
  - Intel's Larabee/MIC/Knight's Corner is 16-way SIMD
    Theoretically the compiler can handle this, but I wouldn't bet on it.
- Some architectures are getting better…
  - Fermi (Nvidia) can fetch 2 instructions per cycle
  - But it has twice as many cores
- In general:
  - Data-parallel will always be fastest
  - Penalty for control-flow varies from none to huge

Questions?

Memory Philosophy
Instruction Philosophy

Summary & The Future

CPU and GPU Architecture

- GPUs are throughput-optimized
  - Each thread may take a long time, but thousands of threads
- CPUs are latency-optimized
  - Each thread runs as fast as possible, but only a few threads

- GPUs have hundreds of wimpy cores
- CPUs have a few massive cores

- GPUs excel at regular math-intensive work
  - Lots of ALUs for math, little hardware for control
- CPUs excel at irregular control-intensive work
  - Lots of hardware for control, few ALUs

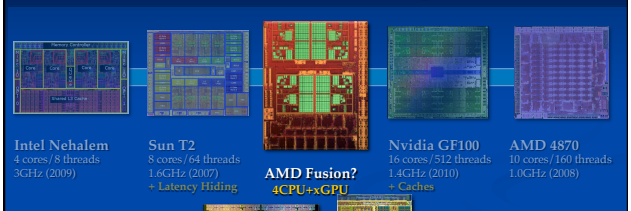
CPU/GPU Convergence

CPU
Few cores/threads
High clock speed

GPU
Many cores/threads
Low clock speed

- CPUs are looking more like GPUs…
  - Multiple cores
  - Multiple threads per core
  - Multiple memory controllers
- GPUs are looking more like CPUs…
  - Caches
  - Function pointers
  - Multiple simultaneous kernels

CPU/GPU Convergence



Intel Nehalem
4 cores/8 threads
3GHz (2009)

Sun T2
8 cores/64 threads
1.6GHz (2007)
+ Latency Hiding

AMD Fusion?
4CPU+xGPU

Nvidia GF100
16 cores/512 threads
1.4GHz (2010)
+ Caches

AMD 4870
10 cores/160 threads
1.0GHz (2008)

Intel MIC?
64CPU+Vectors

STI Cell?
1CPU+8SPU

What Does CPU/GPU Convergence Mean For You?

- **Why are CPUs are moving towards GPUs?**
  - CPUs can't keep scaling performance=latency (Power and complexity)
- **Why are GPUs are moving towards CPUs?**
  - CPUs are easier to program=flexibility
  - GPUs can afford the increased complexity

- Conclusions:
  1) If your algorithm runs well on a GPU today it will continue to run well in the future.
  2) If your algorithm does **not** run well on a GPU today, it may run better in the future, but it is unlikely to scale easily in the long run.
  3) The longer you wait, the easier GPUs will be to program.


Questions?