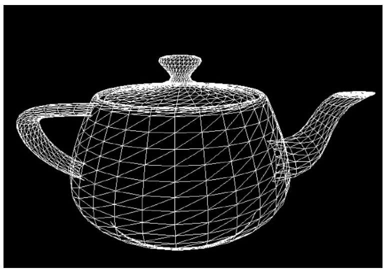
GPU Architectures for Non-Graphics People

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

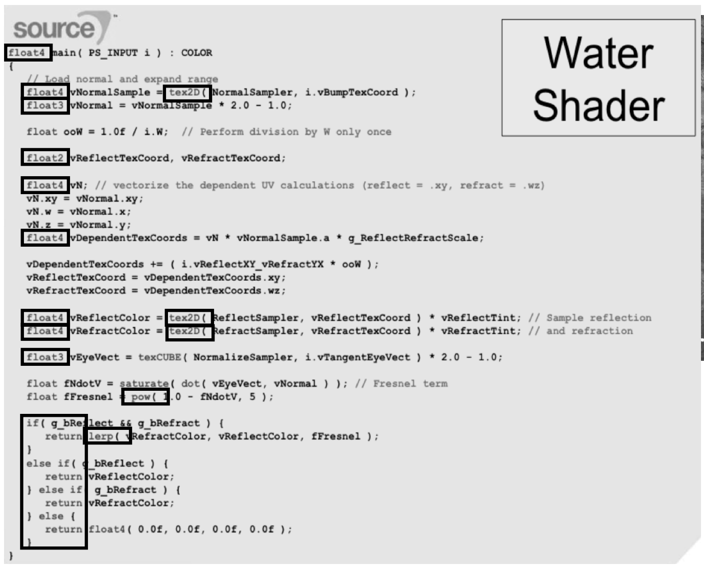


Images from caig.cs.nctu.edu.tw/course/CG2007

- 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
    - Intel Nehalem: 106 GFLOPs, 32GB/s peak

Graphics requires a lot of computation and a huge amount of bandwidth. This has encouraged people to look into using GPUs for more general purpose computing. These numbers are from a 2008-vintage video game compared to a 2009 vintage top-of-the-line CPU. Note that the video game is already comparable to a top-of-the-line server CPU from the year after it was introduced.

Example Shader: Water



```
float4 main( PS_INPUT i ) : COLOR
{
  // Load normal and expand range
  float4 vNormalSample = tex2D( NormalSampler, i.vBumpTexCoord );
  float3 vNormal = vNormalSample * 2.0 - 1.0;

  float ooW = 1.0f / i.W;  // Perform division by W only once

  float2 vReflectTexCoord, vRefractTexCoord;

  float4 vN; // vectorize the dependent UV calculations (reflect = .xy, refract = .wz)
  vN.xy = vNormal.xy;
  vN.w = vNormal.x;
  vN.z = vNormal.y;
  float4 vDependentTexCoords = vN * vNormalSample.a * g_ReflectRefractScale;

  vDependentTexCoords += ( i.vReflectXY_vRefractYX * ooW );
  vReflectTexCoord = vDependentTexCoords.xy;
  vRefractTexCoord = vDependentTexCoords.wz;

  float4 vReflectColor = tex2D( ReflectSampler, vReflectTexCoord ) * vReflectTint; // Sample reflection
  float4 vRefractColor = tex2D( RefractSampler, vRefractTexCoord ) * vRefractTint; // and refraction

  float3 vEyeVect = texCUBE( NormalizeSampler, i.vTangentEyeVect ) * 2.0 - 1.0;

  float fNdotV = saturate( dot( vEyeVect, vNormal ) ); // Fresnel term
  float fFresnel = pow( 1.0 - fNdotV, 5 );

  if( g_bReflect && g_bRefract ) {
     return lerp( vRefractColor, vReflectColor, fFresnel );
  }
  else if( g_bReflect ) {
     return vReflectColor;
  } else if( g_bRefract ) {
     return vRefractColor;
  } else {
     return float4( 0.0f, 0.0f, 0.0f, 0.0f );
  }
}
```

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

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

Graphics shaders are small programs that do almost everything regular programs do. The big exception is that they have (until very recently) been limited to only containing loops with static bounds.

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

AMD and Nvidia want you to believe that the first one is the right answer.

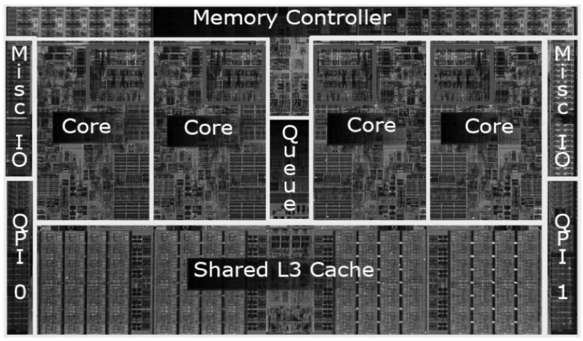
Intel admits the second is about right.

For a lot of people the third one is true unless they restructure/redesign their algorithm.

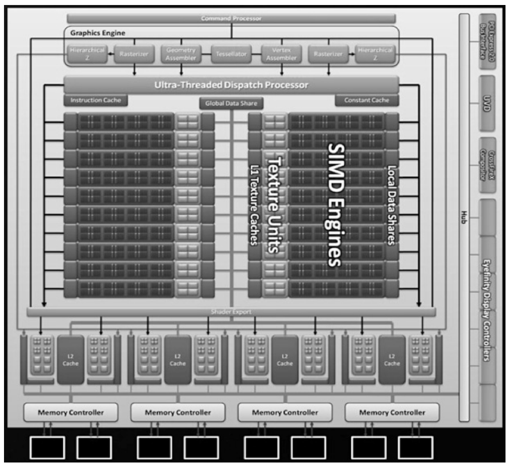
The fourth one is slowly becoming less of an issue as GPUs mature.

Why Should You Care?

Intel Nehalem 4-core



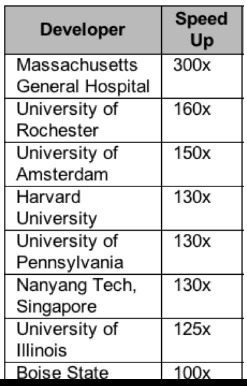
AMD Radeon 5870



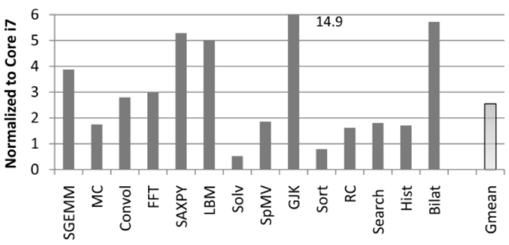
130W, 263mm^2	188W, 334mm^2
32 GB/s BW, **106 GFLOPs** (SP)	**154 GB/s** BW, **2720 GFLOPs** (SP)
Big caches (8MB)	Small caches (<1MB)
Out-of-order	Hardware thread scheduling
0.8 GFLOPs/W	14.5 GFLOPs/W

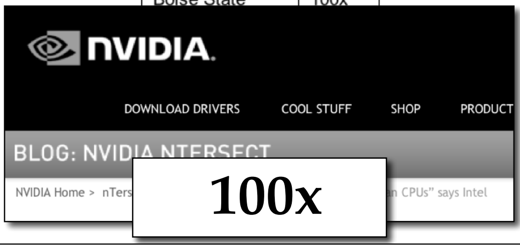
Here are two top-of-the-line chips in 2009. The images are scaled to approximately proportional sizes. The important thing to note is that the single-precision efficiency of the GPU is 1-to-2 orders of magnitude better than the CPU. It is hard to underestimate the significance of this, if your algorithm can work efficiently on the GPU architecture. Dual-precision performance will scale similarly, but with slightly less benefit to the GPU.

What Should You Expect?



Developer	Speed Up
Massachusetts General Hospital	300x
University of Rochester	160x
University of Amsterdam	150x
Harvard University	130x
University of Pennsylvania	130x
Nanyang Tech, Singapore	130x
University of Illinois	125x
Boise State	100x

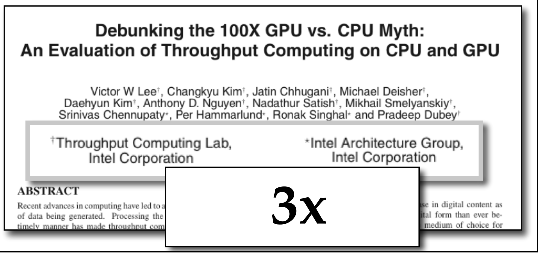




BLOG: NVIDIA NTERSECT

NVIDIA Home > nTers n CPUs" says Intel

100x



Debunking the 100X GPU vs. CPU Myth:
An Evaluation of Throughput Computing on CPU and GPU

Victor W Lee[†], Changkyu Kim[†], Jatin Chhugani[†], Michael Deisher[†], Daehyun Kim[†], Anthony D. Nguyen[†], Nadathur Satish[†], Mikhail Smelyanskiy[†], Srinivas Chennupaty[*], Per Hammarlund[*], Ronak Singhal[*] and Pradeep Dubey[†]

[†]Throughput Computing Lab, Intel Corporation

[*]Intel Architecture Group, Intel Corporation

ABSTRACT

Recent advances in computing have led to a se in digital content as of data being generated. Processing the al form than ever be- timely manner has made throughput com medium of choice for

3x

Nvidia claims 100x speedups, and lots of people have seen this.

Intel did a (what I consider to be quite honest from all appearances) study and found it was more along the lines of 3x. There is one problem with the Intel study: they used a 1-year-old GPU vs. a current CPU. However, they also did not take into account the time to transfer data to the GPU (next slide) which makes their results very very optimistic for the GPU.

What we'd really like to see is Intel vs. Nvidia, with each optimizing their own code.

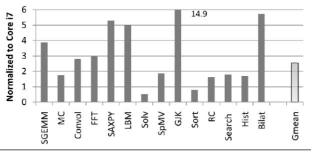
The Achilles Heel of GPUs

GPUs are fast, but…

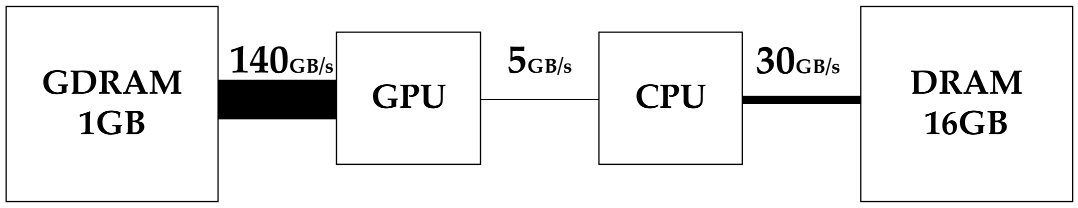
Getting data to the GPU is slow

Getting data to the GPU is slow

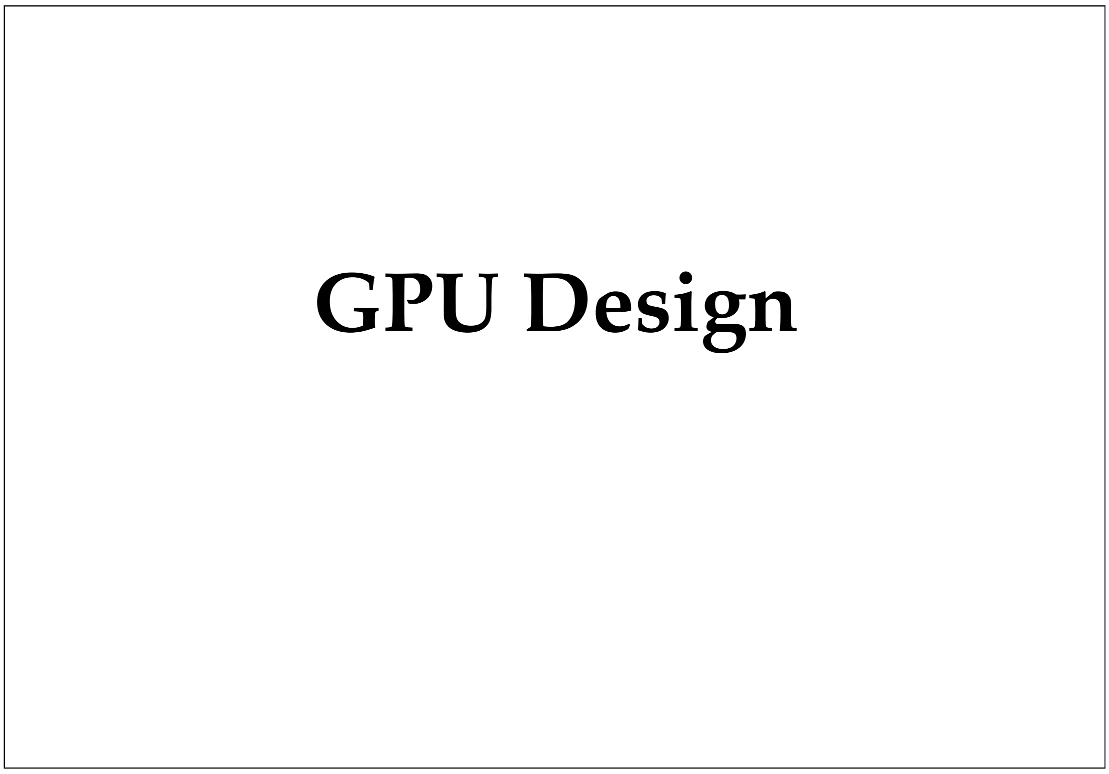
Getting data *from* the GPU is slow



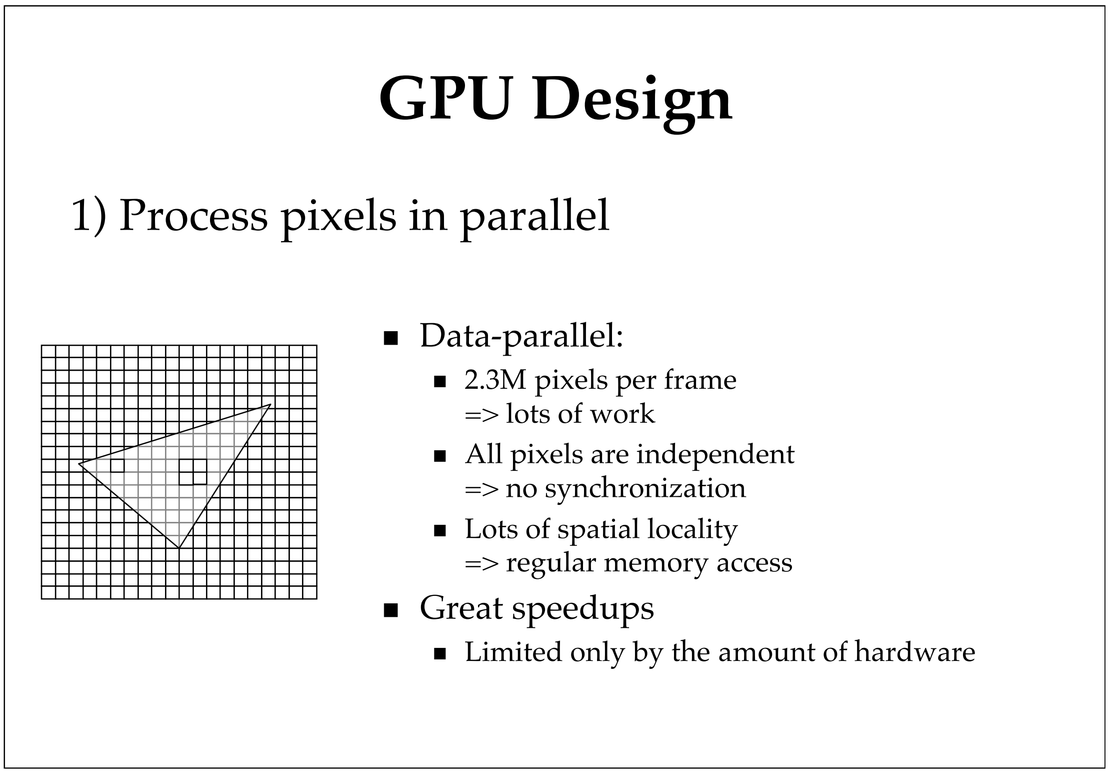
3x not counting data movement



No matter how fast your GPU code is, if you have to move data to/from the CPU frequently (or at all, really) you will see very little performance improvement. You want to move your input data to the GPU (or better yet, generate it there) and then do lots and lots of work on it before moving a small result back. Otherwise the transfer time over PCIe will kill your performance. This will improve as GPUs and CPUs move onto the same die, which may give AMD a huge advantage over Nvidia, and is one of the prime reasons Intel does not allow Nvidia to use their new QPI bus directly.

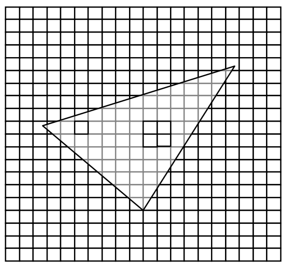


GPU Design



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
- Great speedups
  - Limited only by the amount of hardware

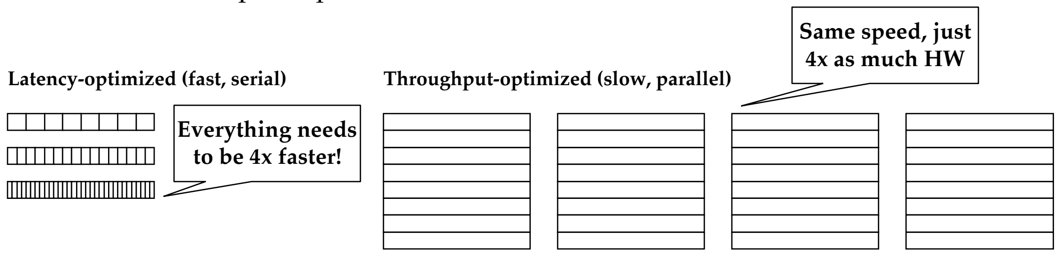
Nearby pixels are very likely to accesses similar texture data (they are close together on screen, so they will have similar final results).

This means that a small texture cache can be extremely valuable. Indeed GPUs today only have texture caches, and they are quite small.

GPU Design

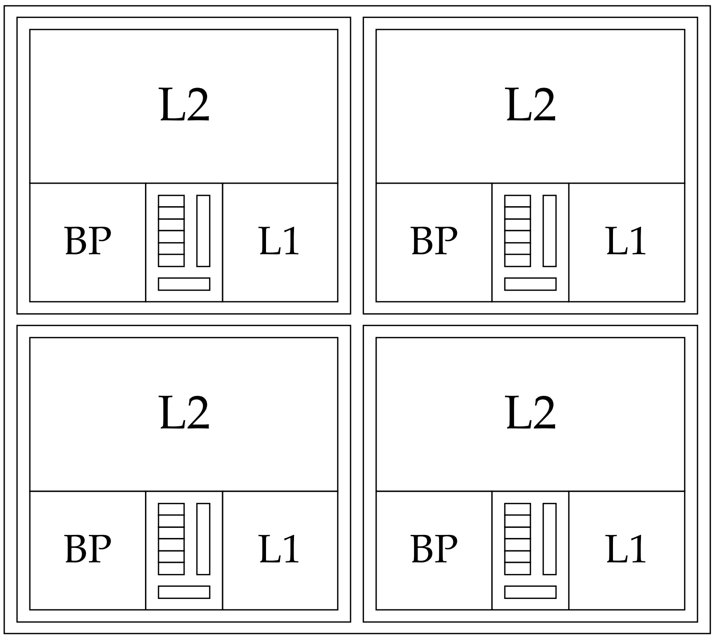
2) Focus on throughput, not latency

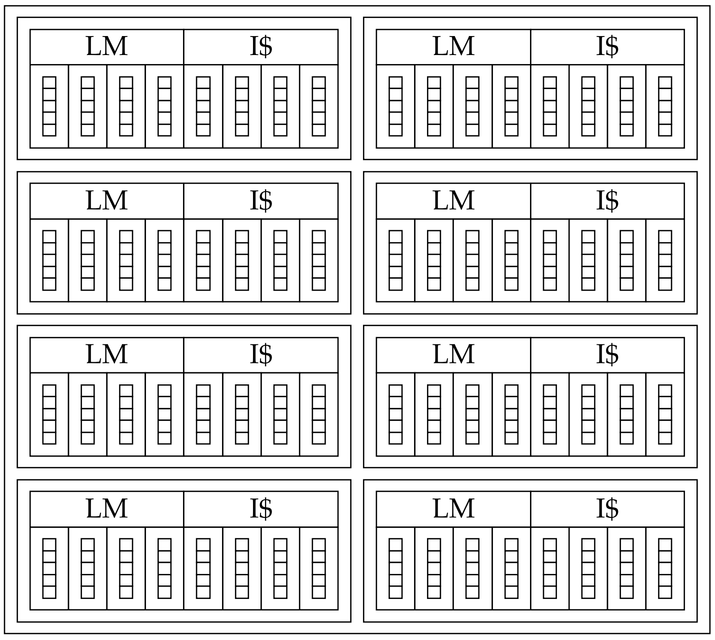
- Each pixel can take a long time…
  …as long as we process many at the same time.
- Great scalability
  - Lots of simple parallel processors
  - Low clock speed (power-efficient)


Latency-optimized (fast, serial)
Everything needs to be 4x faster!
Throughput-optimized (slow, parallel)
Same speed, just 4x as much HW

Making a whole processor run 4x faster is far harder than stamping out 4x as many processing elements.

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

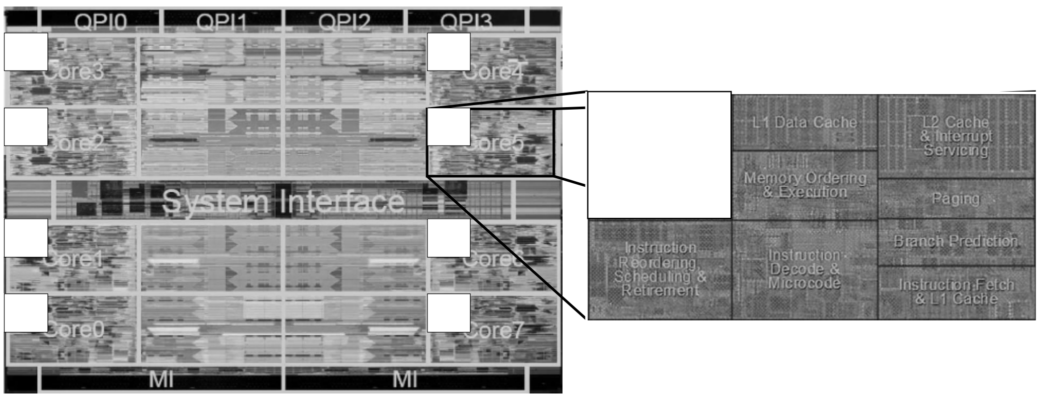
8*8 Wimpy GPU Cores: No caches, in-order, single-issue, single-precision…

About 1 IPC per core, 64 IPC total @1.5GHz

Note the difference in area that does real work (yellow) between the two architectures. CPUs dedicate huge amounts of area to making sure that one thread can run really fast (caches and predictors to avoid ever having to stall it). GPUs dedicate huge amounts of area to making sure lots of threads can run simultaneously, and target aggregate throughput over latency.

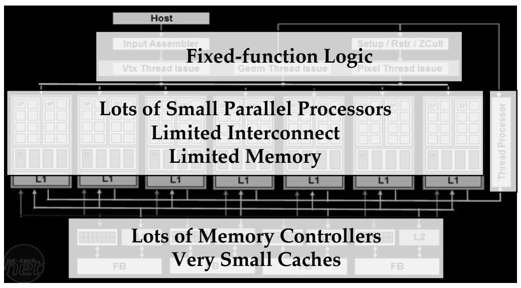
Example CPUs Today

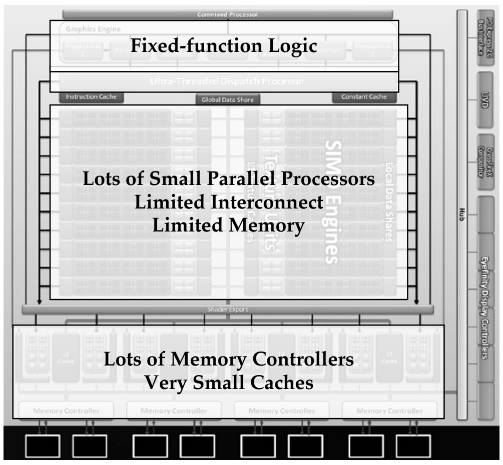
- Intel 8-core 45nm Nehalem (2010)


QPI0
QPI1
QPI2
QPI3
Core3
Core4
Core2
Core5
System Interface
Core1
Core6
Core0
Core7
MI
MI
L1 Data Cache
L2 Cache & Interrupt Servicing
Memory Ordering & Execution
Paging
Instruction Reordering, Scheduling & Retirement
Instruction Decode & Microcode
Branch Prediction
Instruction Fetch & L1 Cache

Intel's high-end server chip uses a very small amount of the area to actually compute. Most of the area is cache (which is actually a very good heat sink) and support logic to enable the computations to go quickly.

Example GPUs Today





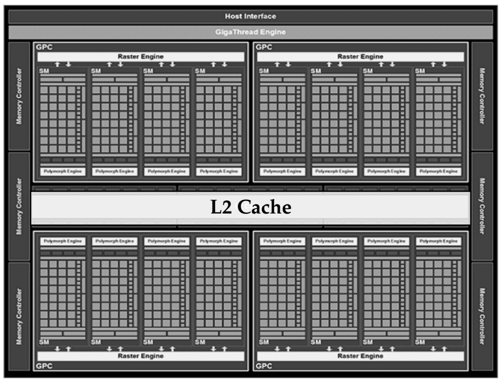
Nvidia G80 **AMD 5870**

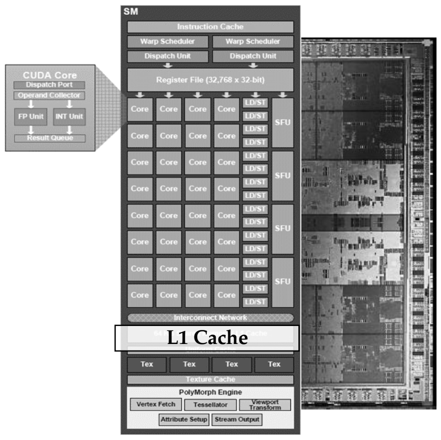
Most of a GPU is dedicated to small parallel processors with small local memories. This is how they get such good efficiency because they have a lot of silicon for compute.

In addition they have hardware thread schedulers to make it efficient to swap threads very frequently and fixed-function logic for the portions of the graphics pipeline that are simply too slow to do in software.

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

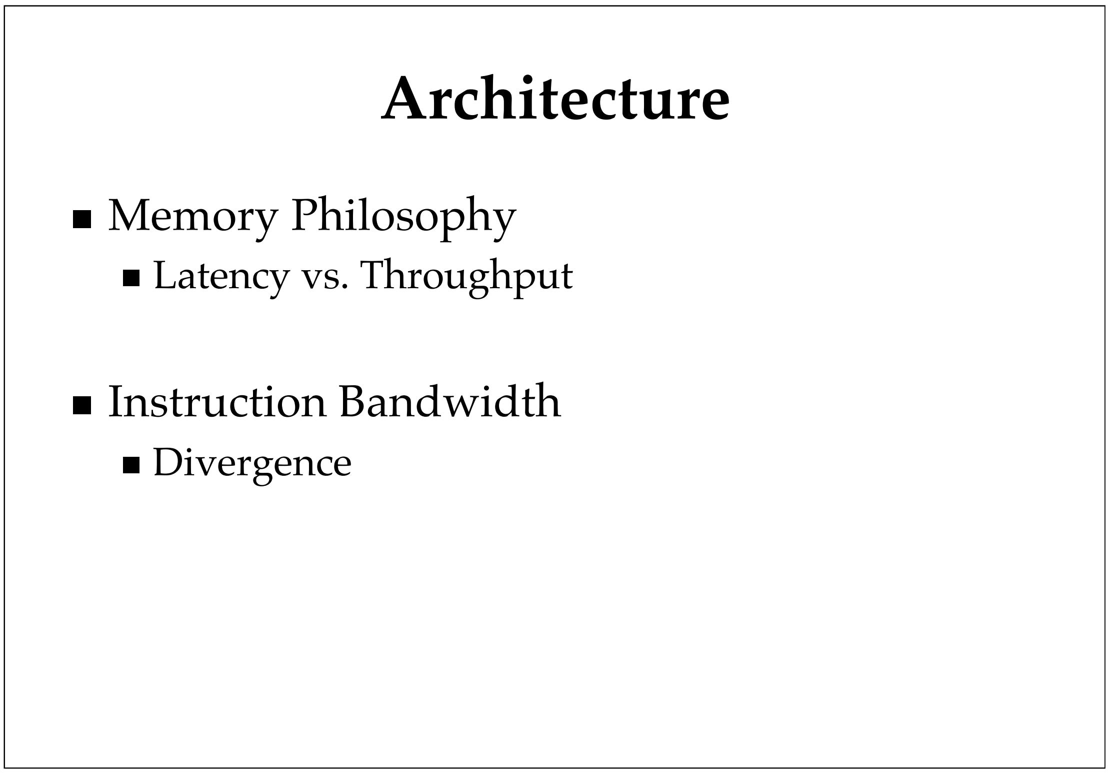
Nvidia's new architecture adds a few really nice features: data caches, function pointers, and multiple kernel execution.

Data caches will provide most developers with ~90% of the benefit of software-managed memories with 0% of the hassle.

Function pointers are important for more advanced language features (Nvidia claims C++ support).

Multiple kernel execution is essential to maintain interactivity for any OS that uses the GPU for its window manager and wants to simultaneously do compute.

Questions So Far?



Architecture

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

The rest of this presentation will cover two topics: how GPUs and CPUs differ in their approach to handling memory accesses, and issues surrounding instruction bandwidth on GPUs.

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.

Computational intensity is a key metric of an application. The more math a an application does on its data the better it will scale with more and more cores. This is true for both CPUs and GPUs, and algorithms that do more work and access less data are far more likely to scale in the future. Note that these effects happen at each level of the memory hierarchy (caches, DRAM, registers), with the size and performance impact being dictated by the particular memory level. For unique memory accesses, caches do not help, since the data is never reused. (Prefetching can help, though.)

Here A[i] indicates that the array A is being read at a different location for every math operation. Temp indicates that the variable is not being read from memory constantly, which would allow it to be kept in a register or cached very effectively. Note that cache-line effects are ignored here. In the best case they let you get away with a 4x difference, or only 25 cycles down from 100+.

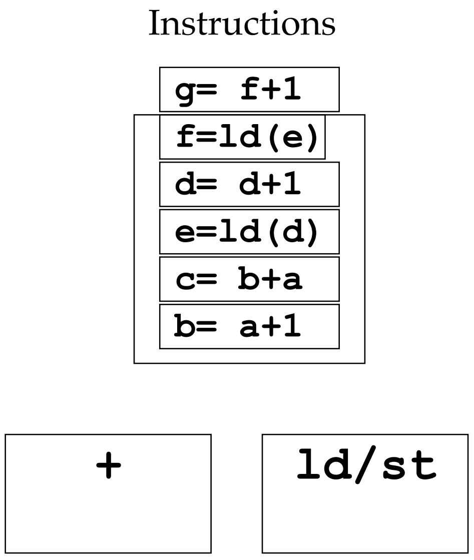
CPU Memory Philosophy

Instructions

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

The CPU is going to process this stream of instructions.

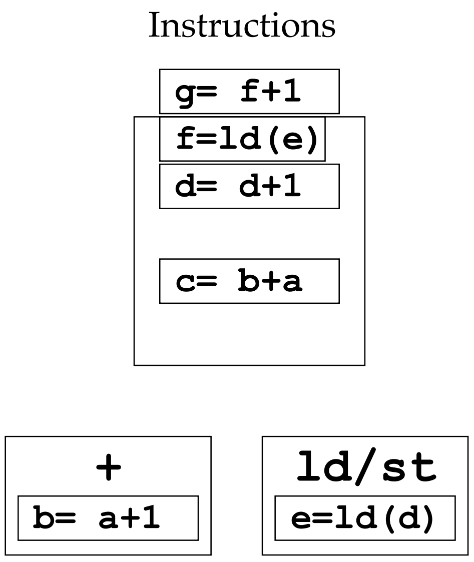
CPU Memory Philosophy



Cycle 0

CPUs dedicate lots of hardware to trying to run fast. Here there is a large instruction window that analyzes as many instructions as possible to detect dependencies (colored) and instructions that can be run in parallel. It also enables executing multiple instructions at the same time.

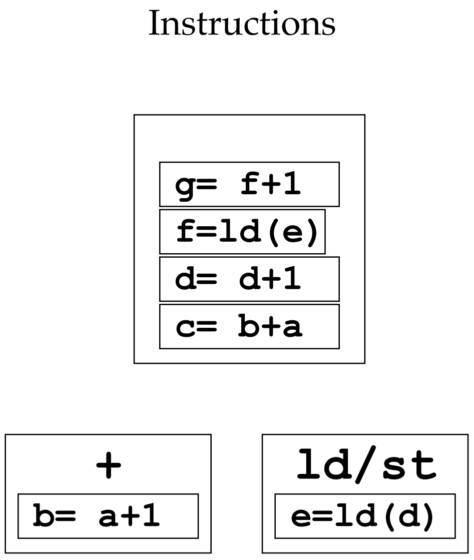
CPU Memory Philosophy



Cycle 0

The CPU finds two instructions that can run at the same time and starts executing both of them.

CPU Memory Philosophy

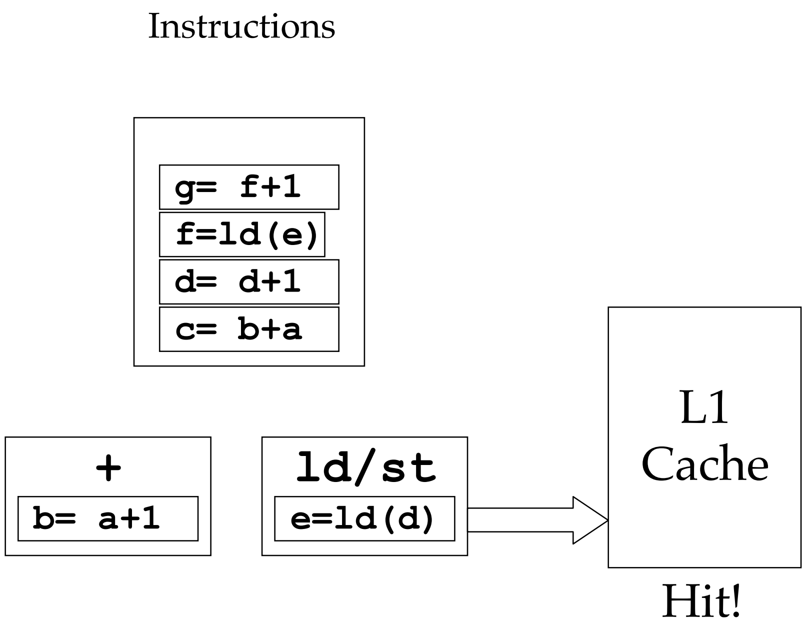


Memory access will take ~100 cycles…

Cycle 0

However, the load instruction is going to take a very long time to access memory. How can we speed this up?

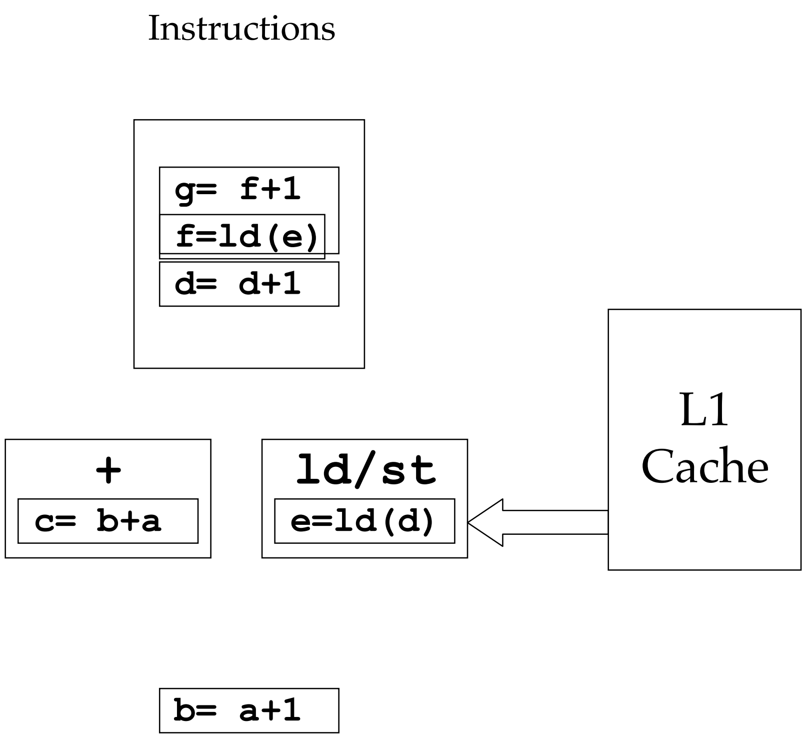
CPU Memory Philosophy



Cycle 0

We'll build a cache. The cache will only take 1 cycle (realistically 3-4 today) to return the data.

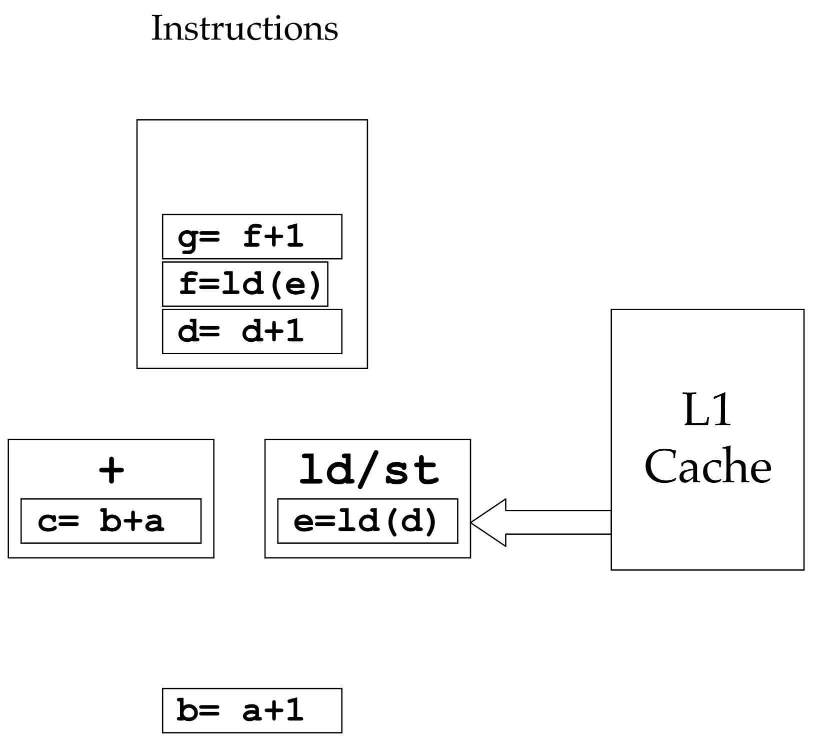
CPU Memory Philosophy



Cycle 1

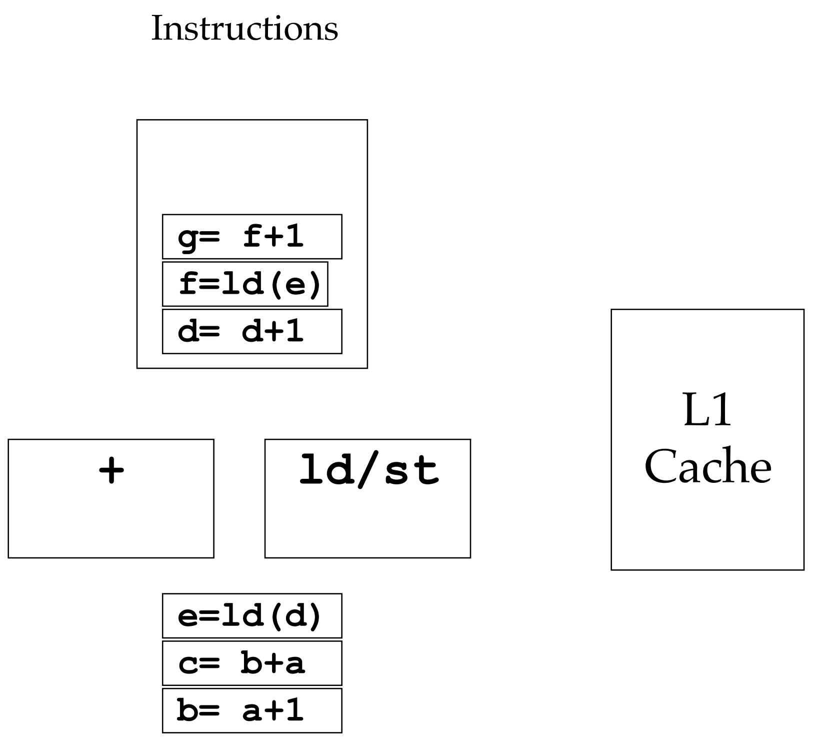
While we are waiting for the data to come back we take advantage of having analyzed the instruction dependencies to execute another instruction.

CPU Memory Philosophy



Cycle 1

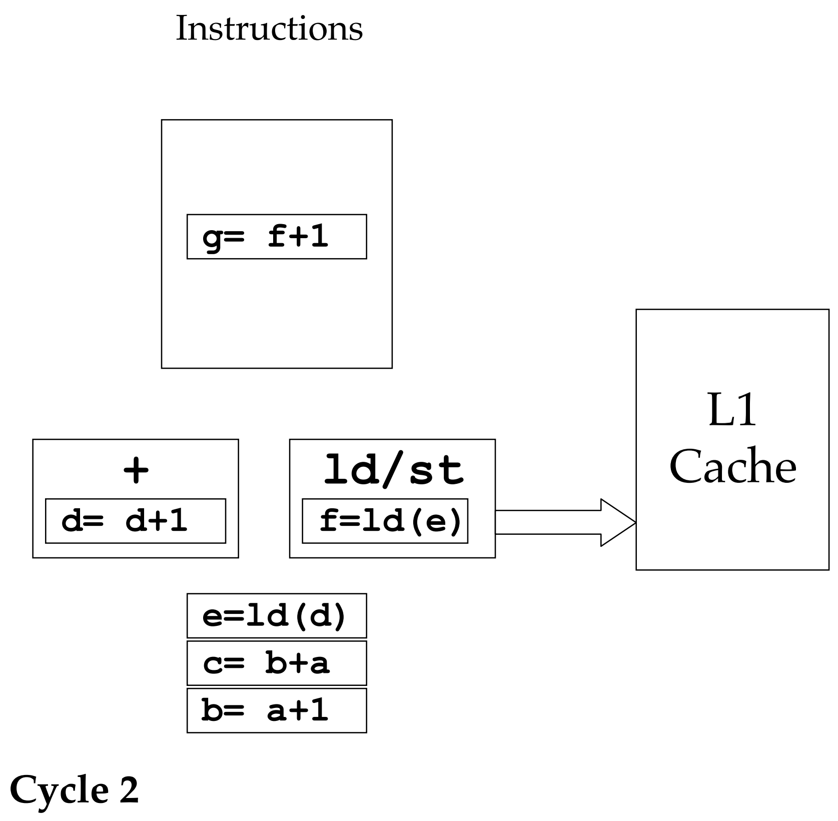
CPU Memory Philosophy



Cycle 1

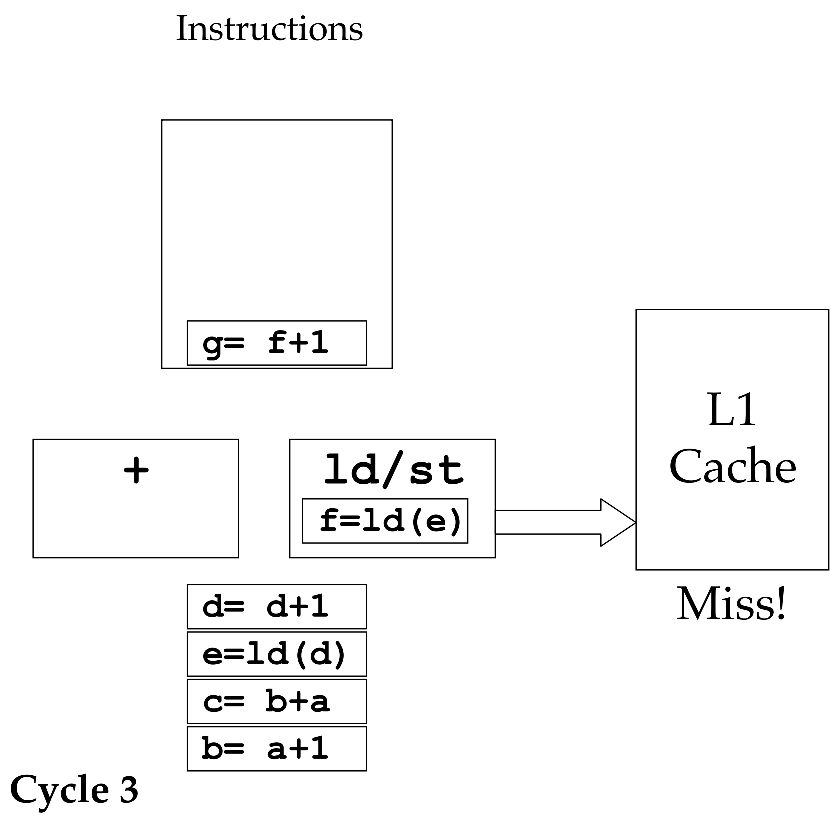
Now by the 2nd cycle (cycle 1) we have executed two math instructions and one memory instruction that hit in the cache due to our dependency analysis, cache, and ability to execute multiple instructions at once.

CPU Memory Philosophy



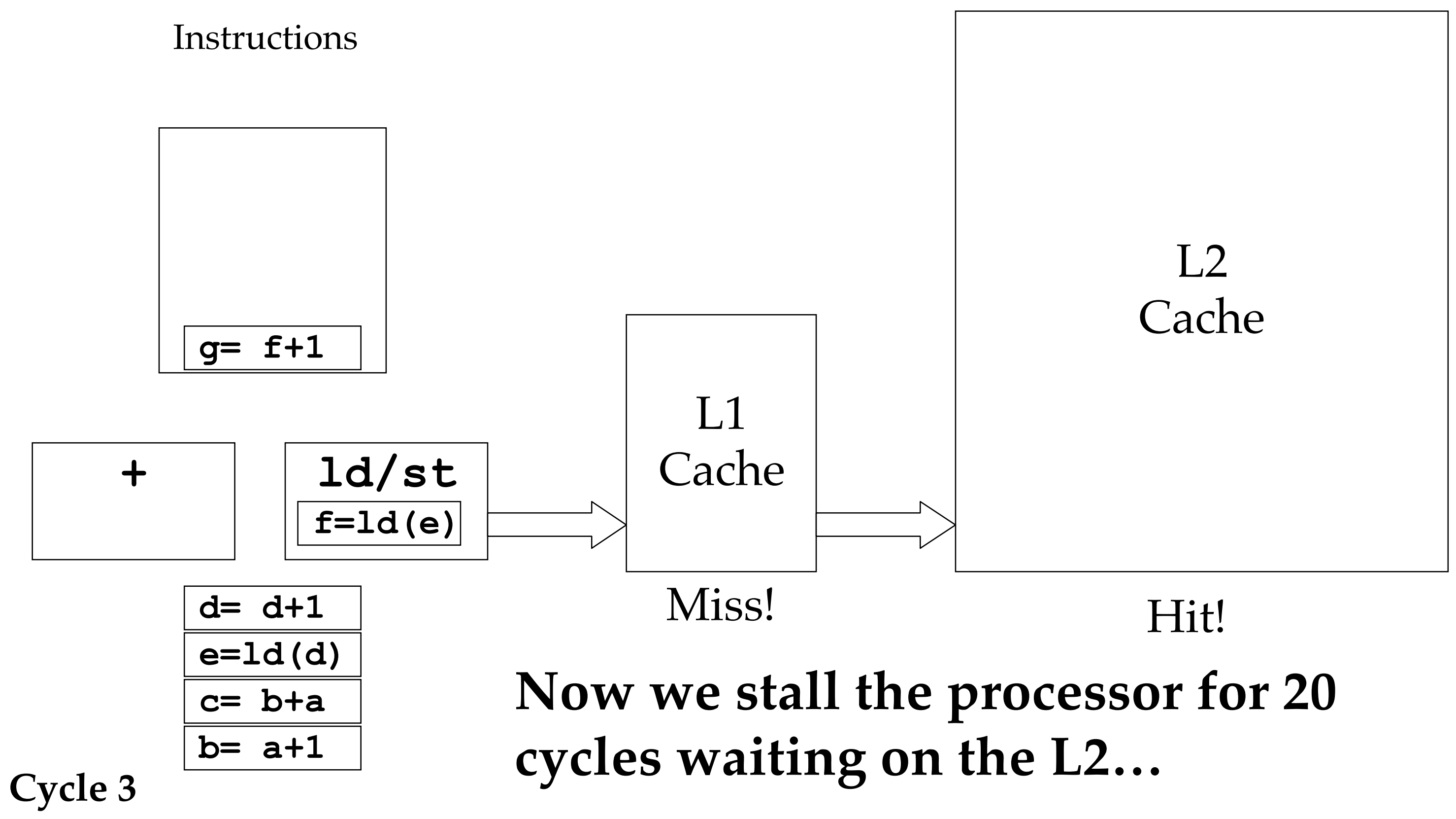
On the next cycle we have an addition and a load…

CPU Memory Philosophy



But this load misses in the L1. What to do?

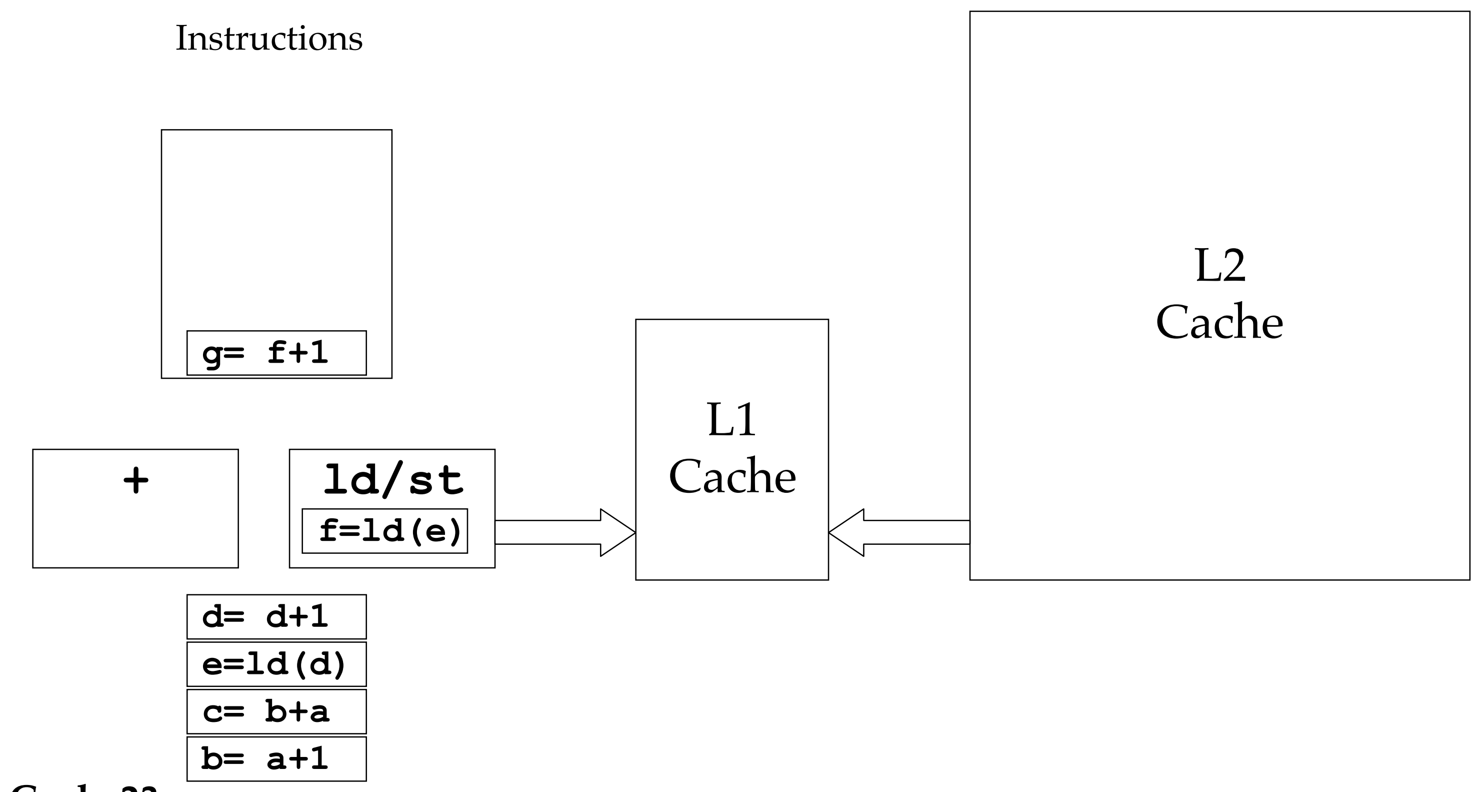
CPU Memory Philosophy



Now we stall the processor for 20 cycles waiting on the L2…

We'll build a much bigger (4x-8x) L2 cache. Remember that hit rate scales as the square root of size, so it has to be a lot bigger to be more helpful. However, larger caches are slower (longer wires) so this time it will take 20 cycles (optimistic) to get the data back. Since we don't have any other instructions that are independent of the load to execute the processor stalls.

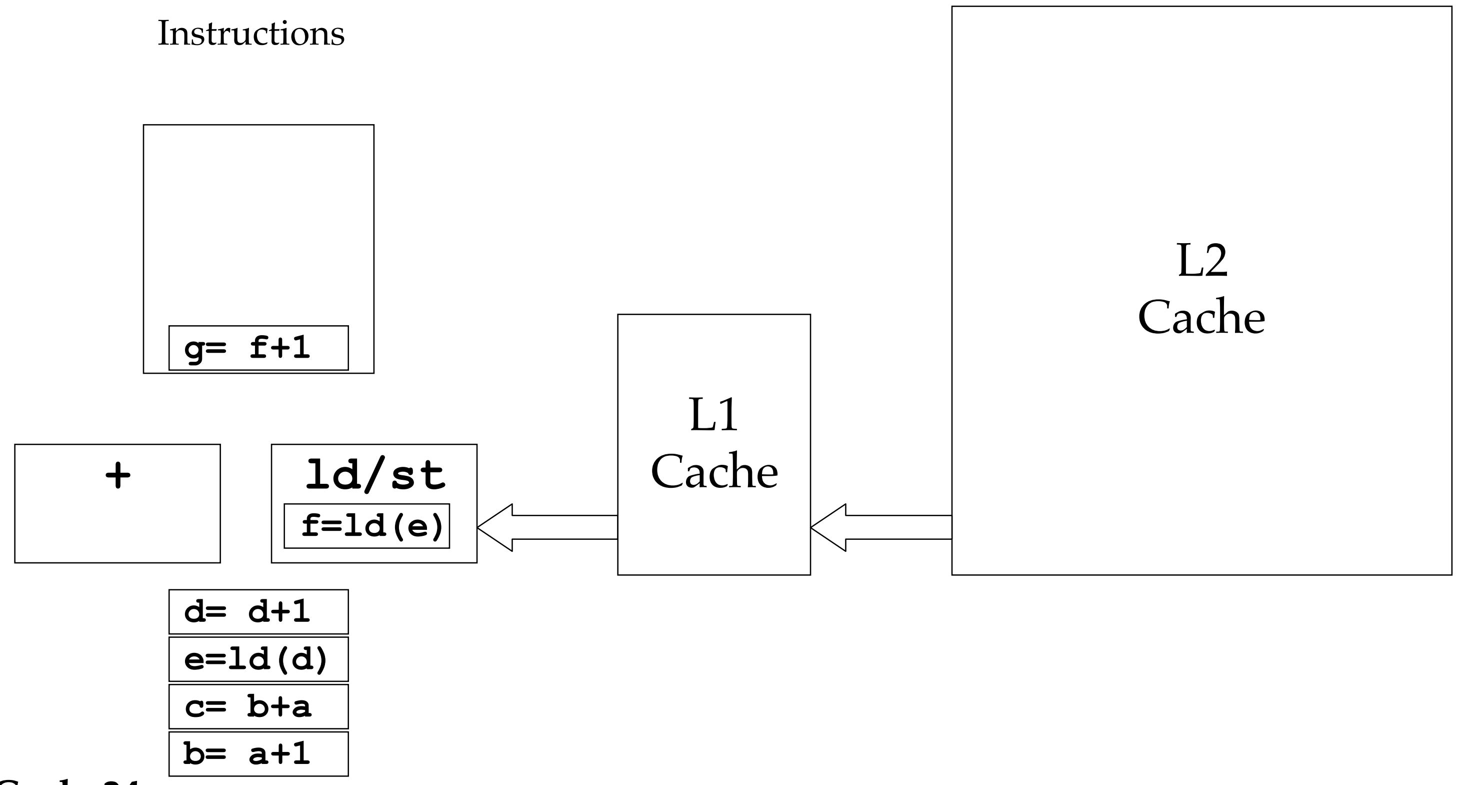
CPU Memory Philosophy



Cycle 23

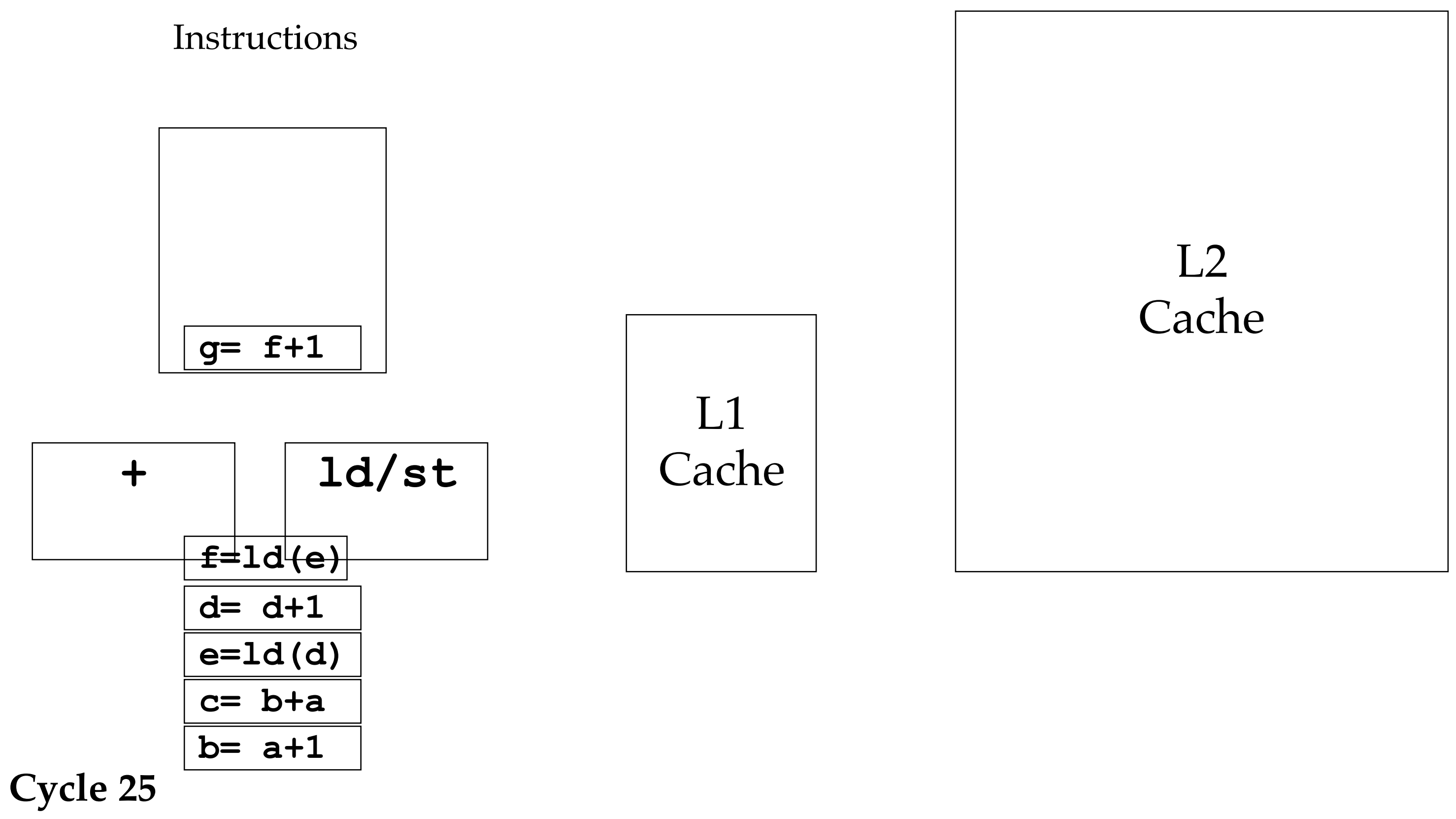
20 cycles later we get our data back and we can continue.

CPU Memory Philosophy

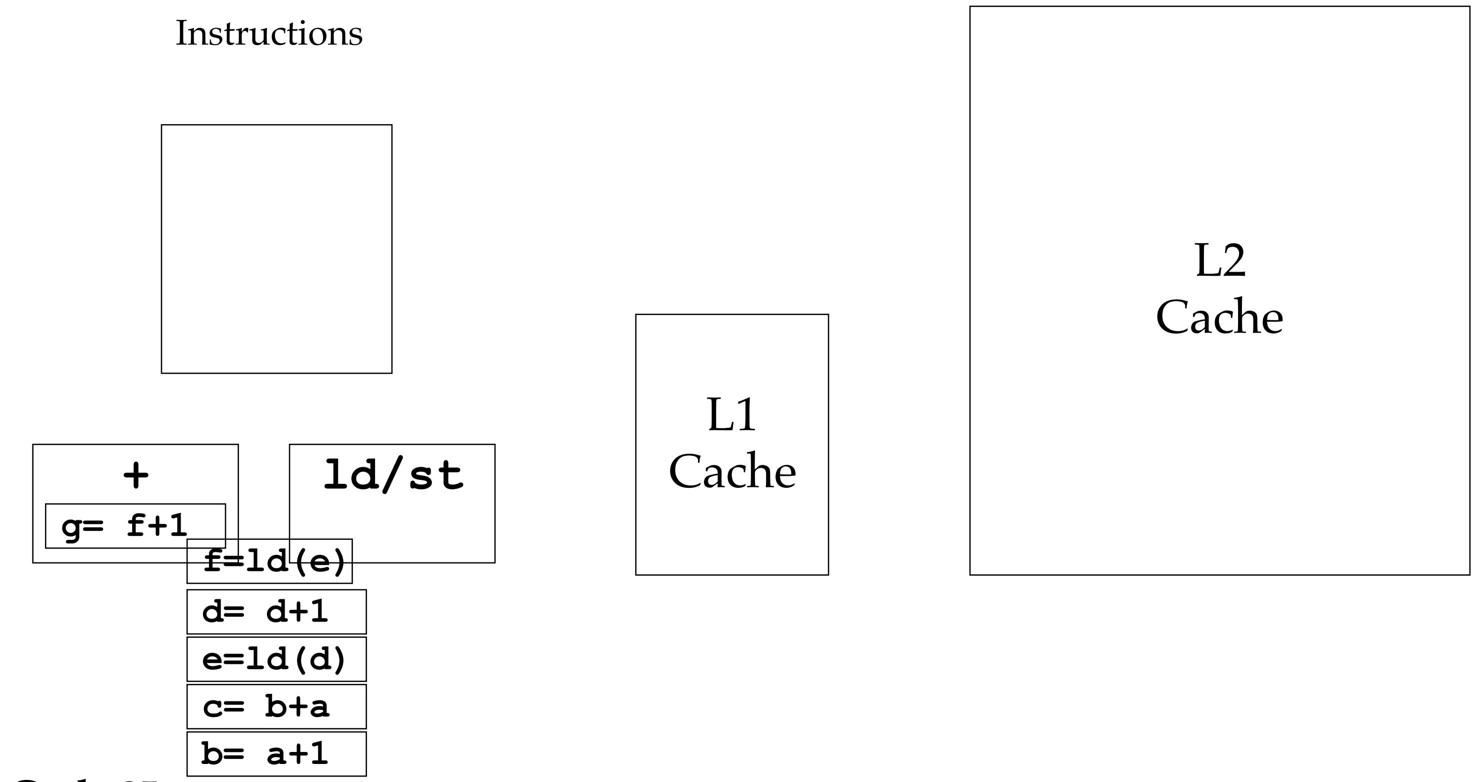


Cycle 24

CPU Memory Philosophy

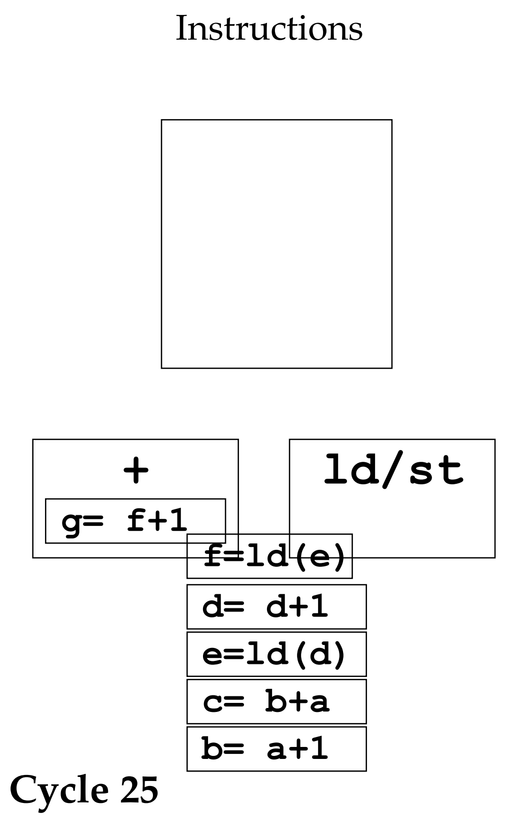


CPU Memory Philosophy



Cycle 25

CPU Memory Philosophy

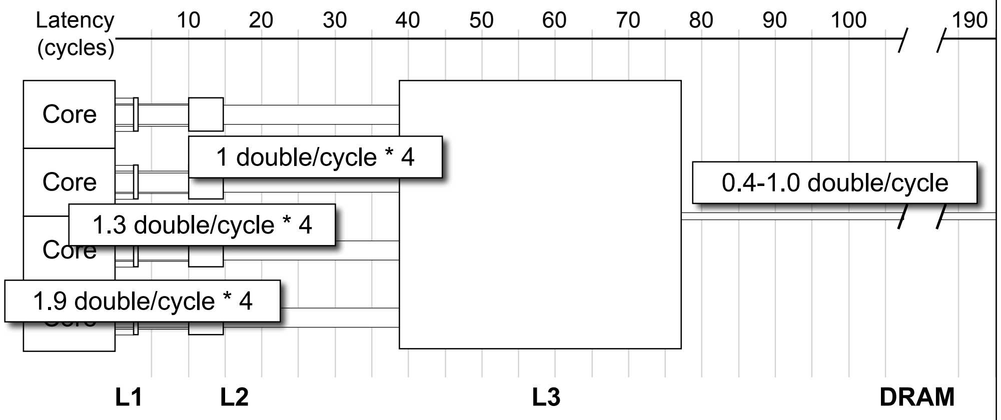


- Big caches + instruction window + out-of-order + multiple-issue
- Approach
  - **Reduce** memory latencies **with caches**
  - **Hide** memory latencies **with other instructions**
- As long as you **hit in the cache** you get **good performance**

CPUs try really hard to get performance by reducing effective memory latencies via caches and hiding memory latency with other instructions. When your data fits in the cache this works really well and your thread can run at full speed. If it doesn't, the CPU delivers terrible performance.

How Far Away is Your Data?

Intel Nehalem 3GHz (2009)

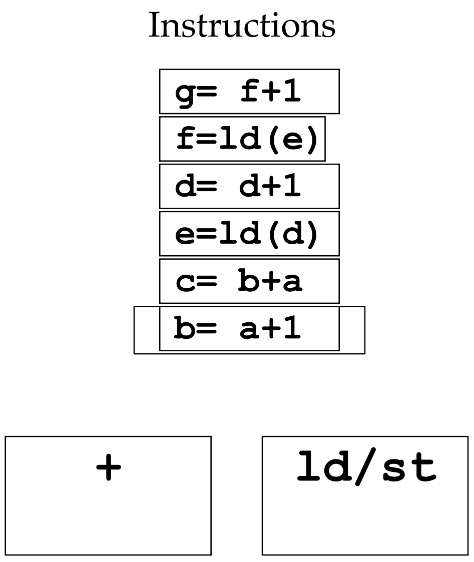


D. Molka, et. al., *Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System*, PACT 2009.

Here are some real numbers for a high-end CPU in 2009. (I highly recommend this paper to anyone who is serious about performance.) Note that unique data can be accessed at a rate of UP TO 1 double every cycle for all four cores. Think about what this means for a parallel operation such as a reduction.

This figure is to scale with widths representing bandwidths, sizes capacity, and distances latency.

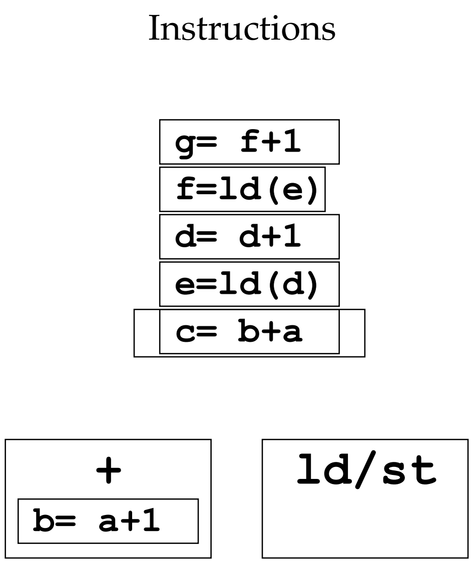
GPU Memory Philosophy



Cycle 0

GPUs are much simpler. They have no instruction window and only execute one instruction at a time. This makes it possible to build those much smaller cores that are needed to fit more on the chip.

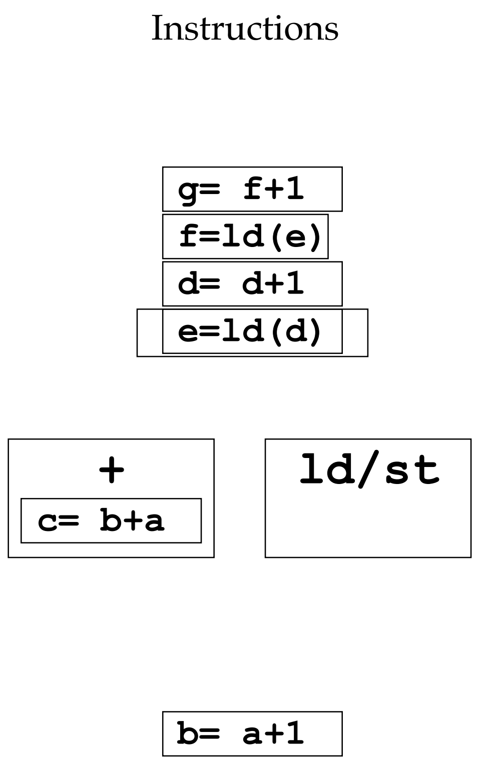
GPU Memory Philosophy



Cycle 0

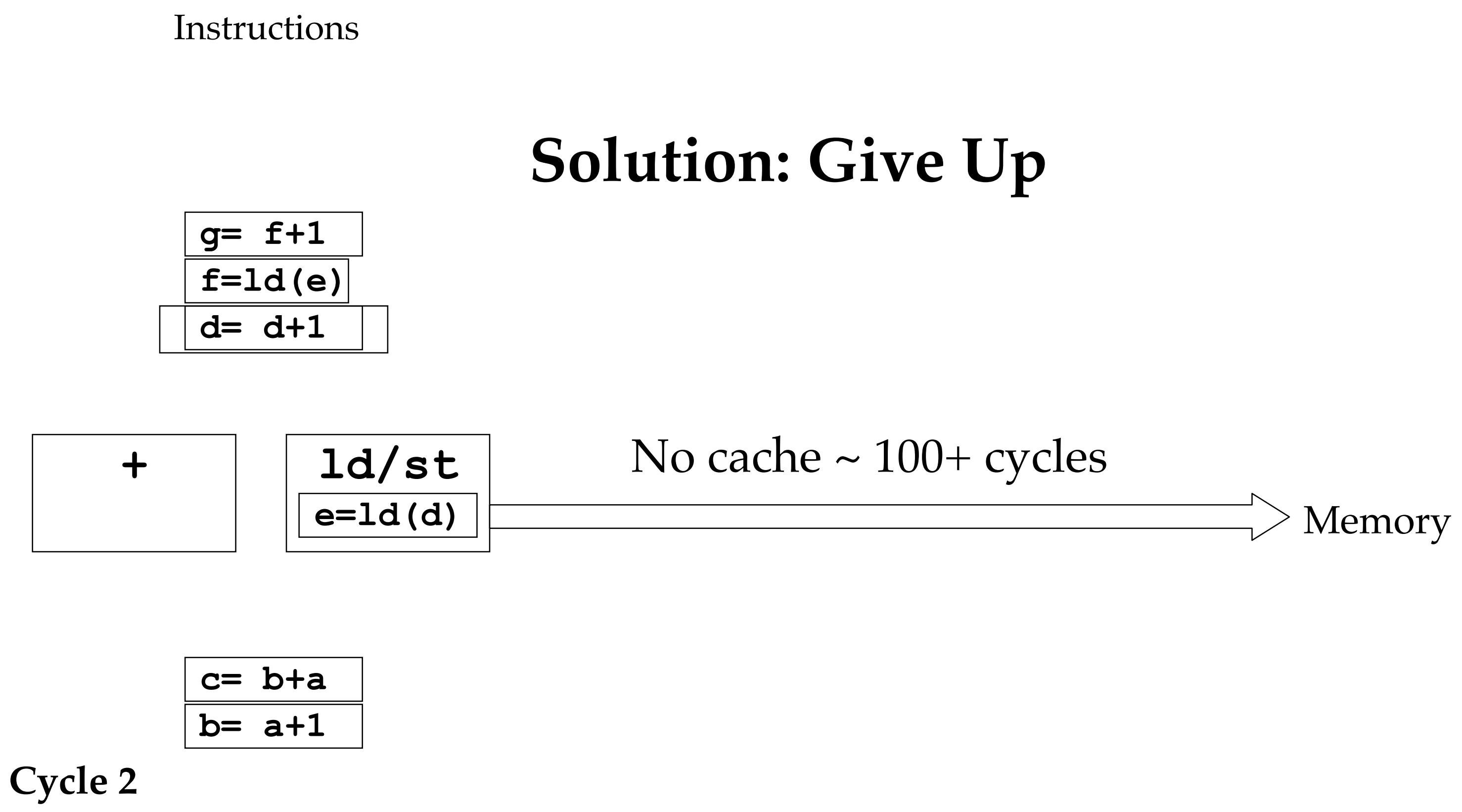
Because GPUs only execute one instruction at a time, the independent add and load instructions must wait for the first add to execute.

GPU Memory Philosophy



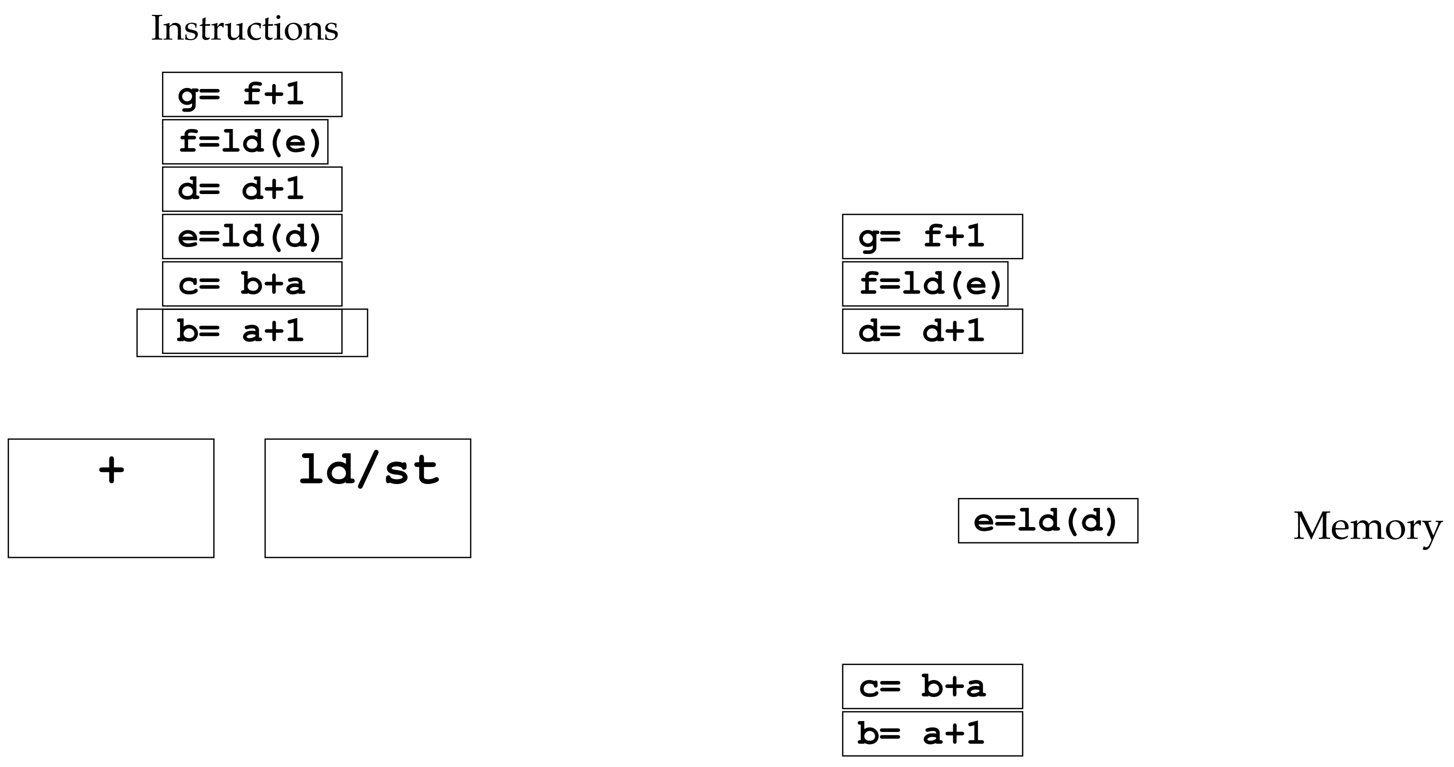
Cycle 1

GPU Memory Philosophy



When we execute the load instruction we're going to have a similar latency to what the CPU would have, but we have no cache on the GPU. (The newest GPUs are starting to introduce very small caches.) Since this is a simple architecture we have no way to do anything about this delay so we just "give up" on this thread.

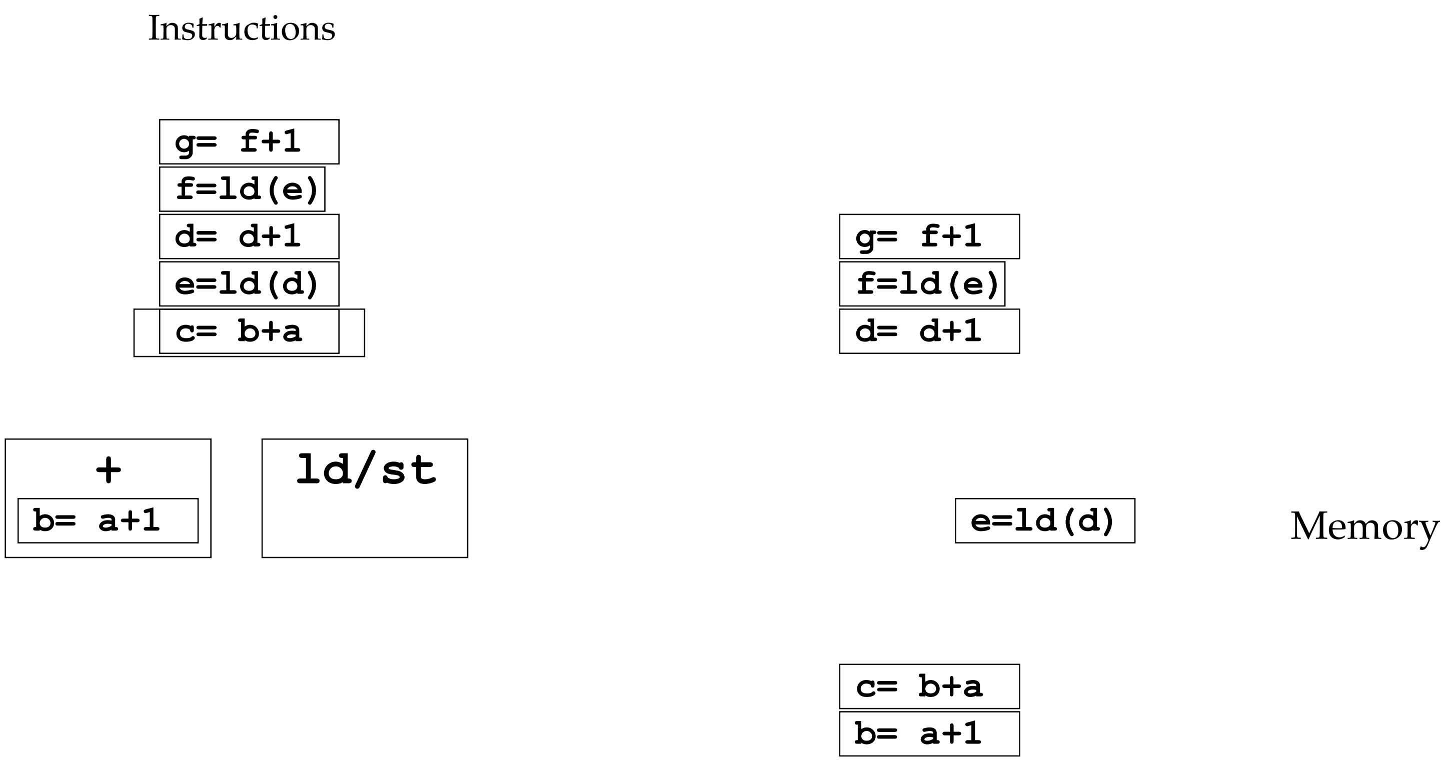
GPU Memory Philosophy



Cycle 2

The GPU then finds another thread (red), and starts running it. The first thread (white) is suspended while it waits for the memory access.

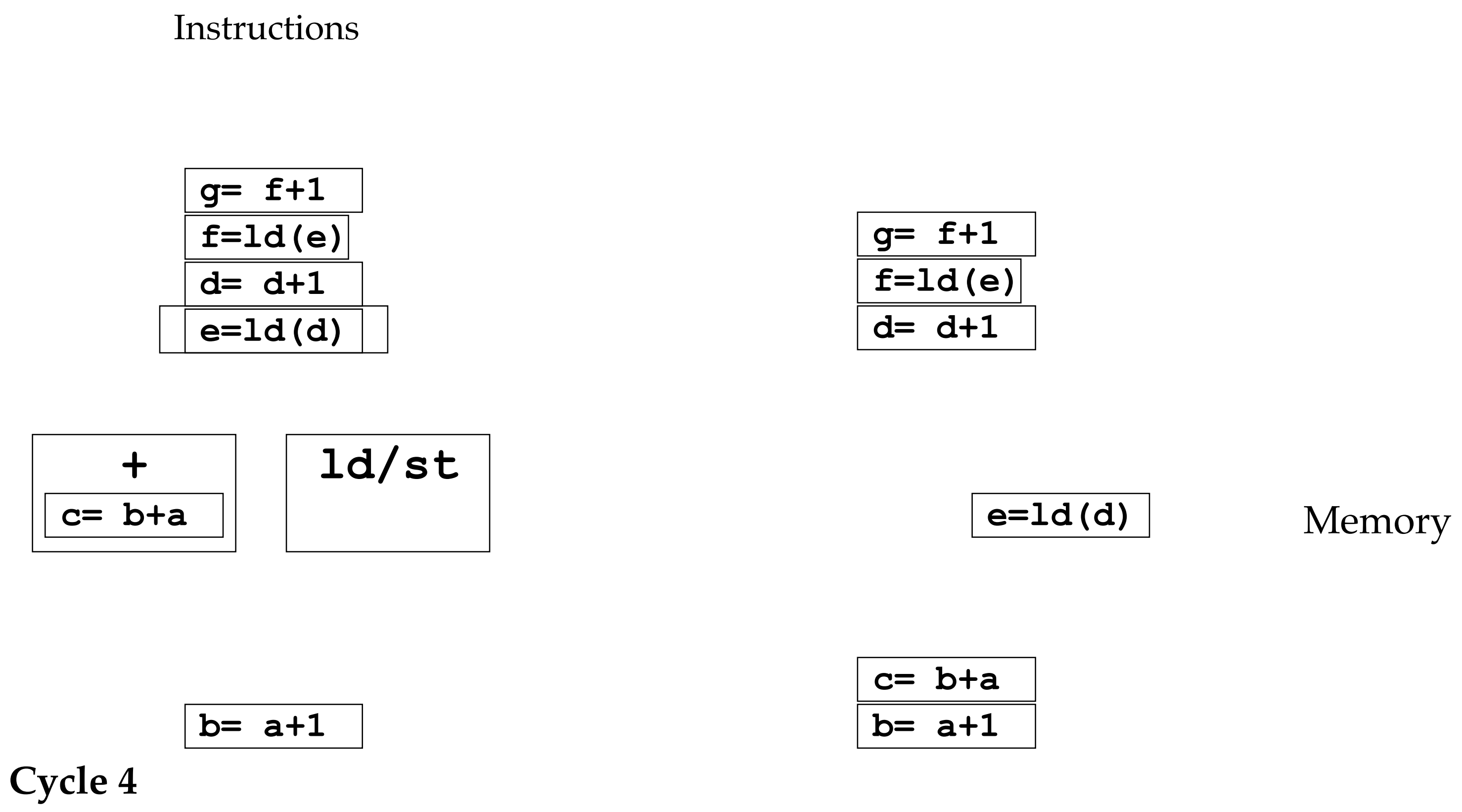
GPU Memory Philosophy



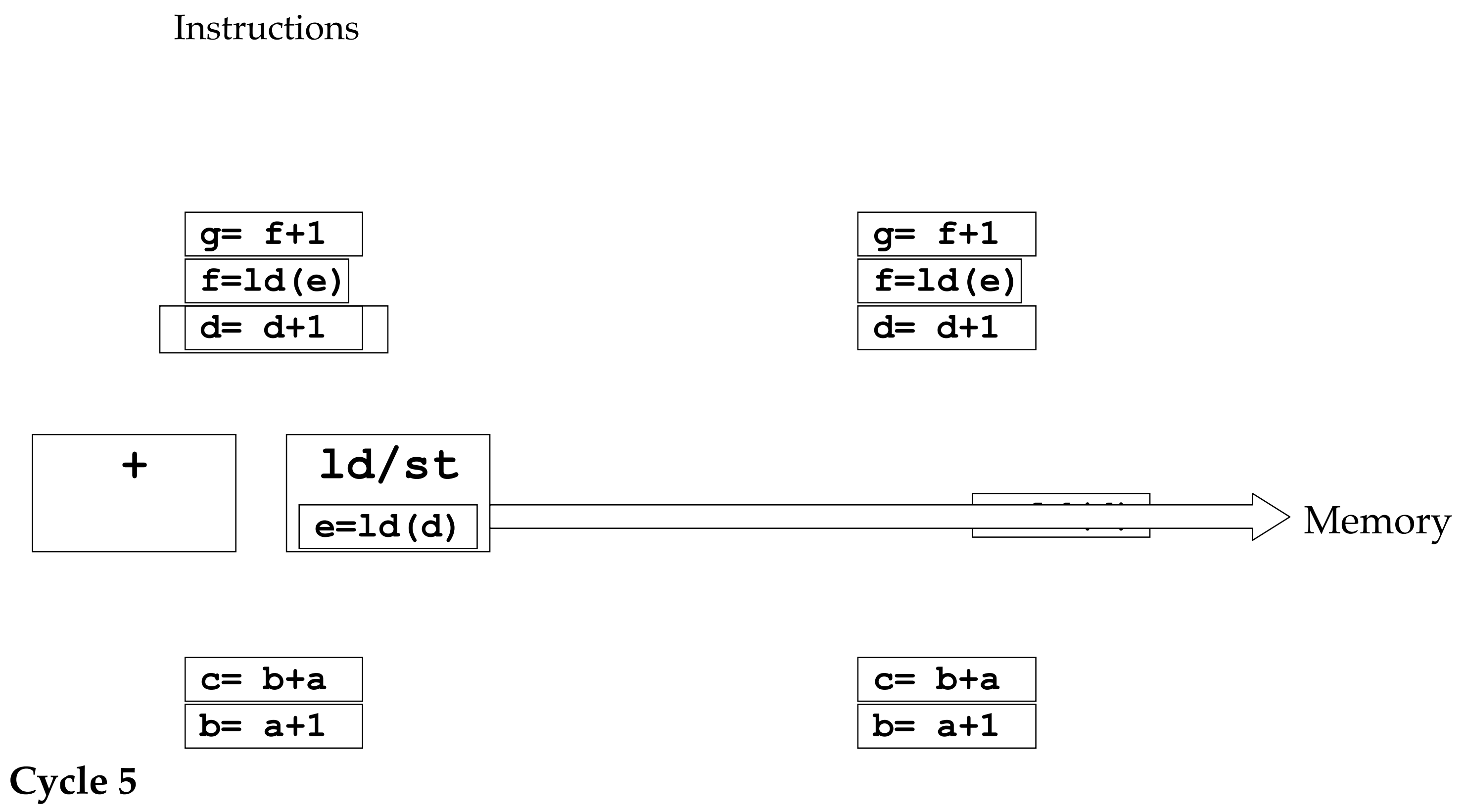
Cycle 3

The second thread executes…

GPU Memory Philosophy

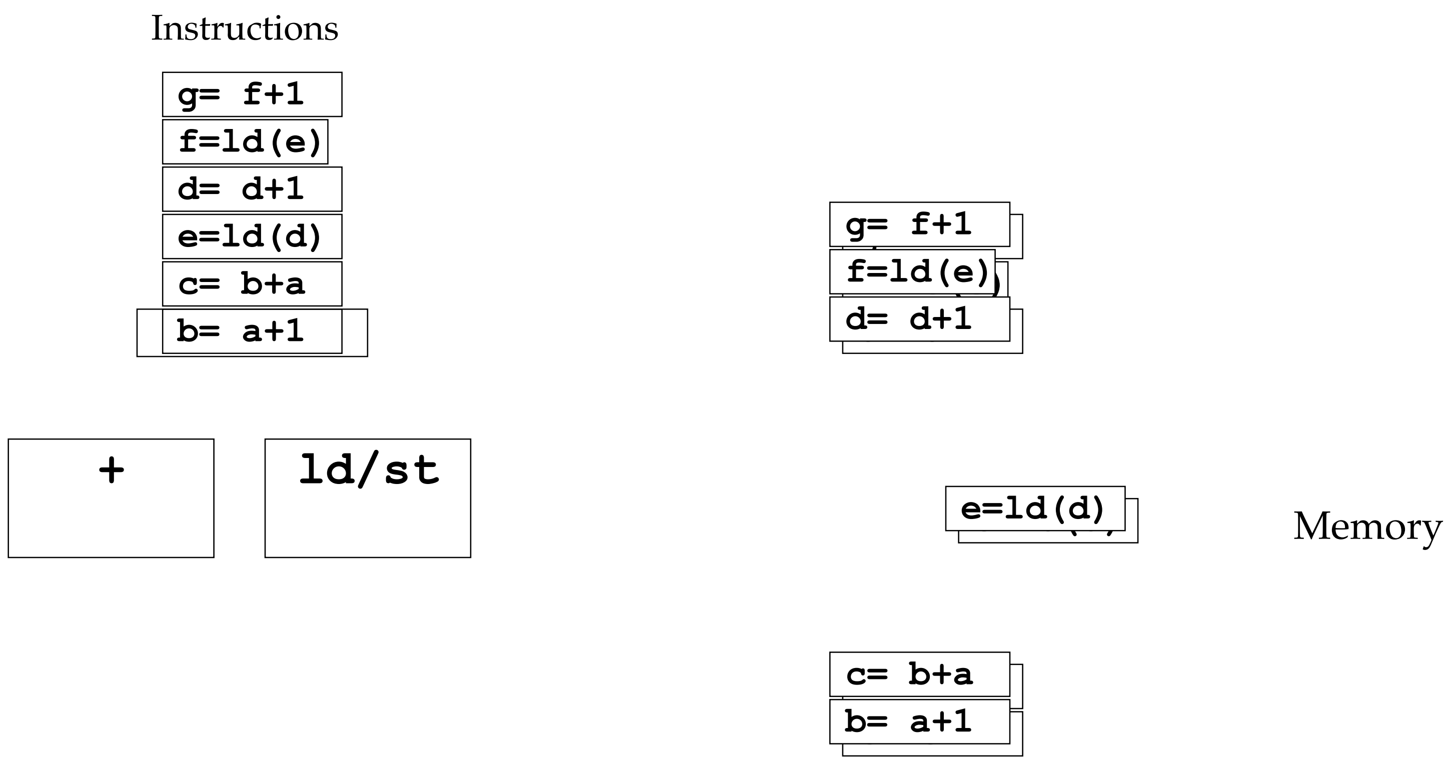


GPU Memory Philosophy



Until it hits a load, at which point we do the same thing…

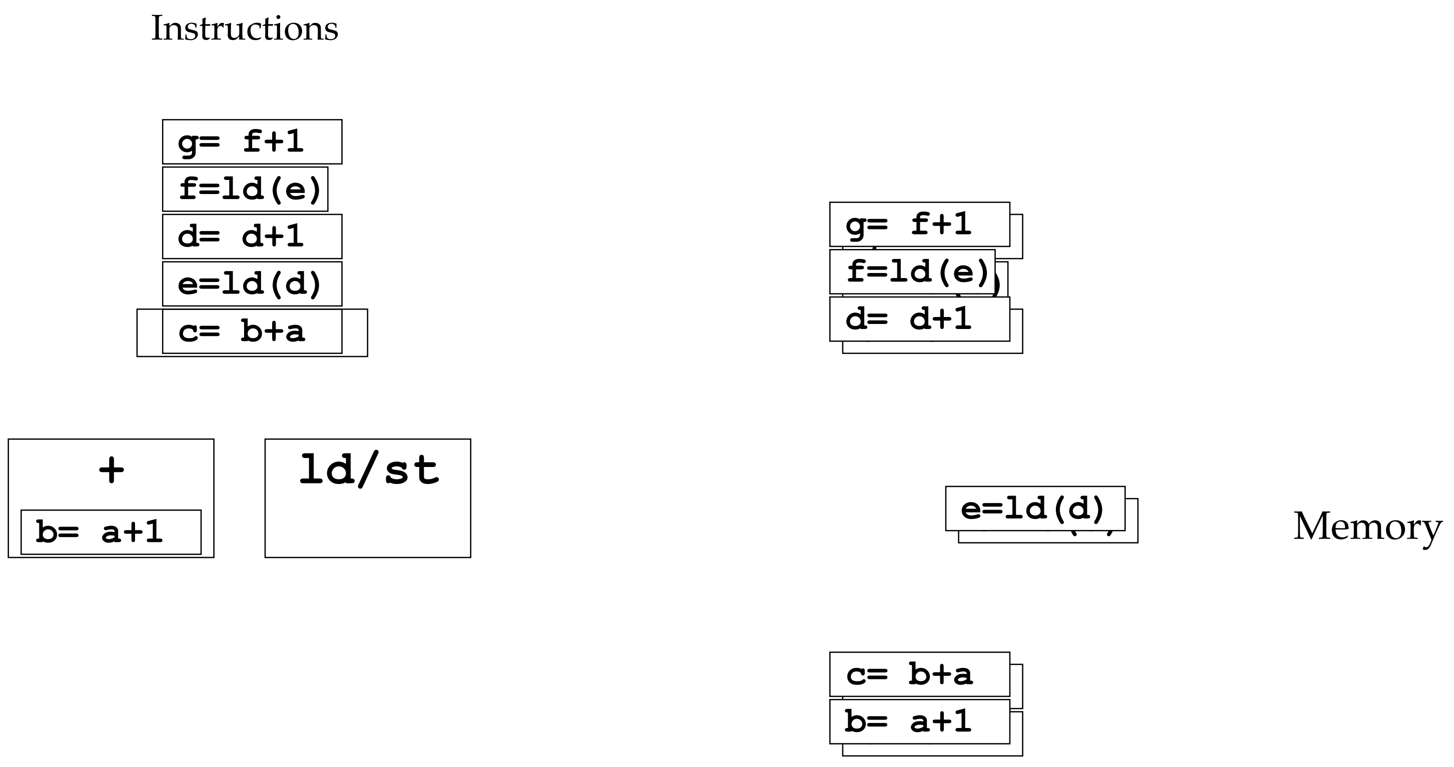
GPU Memory Philosophy



Cycle 5

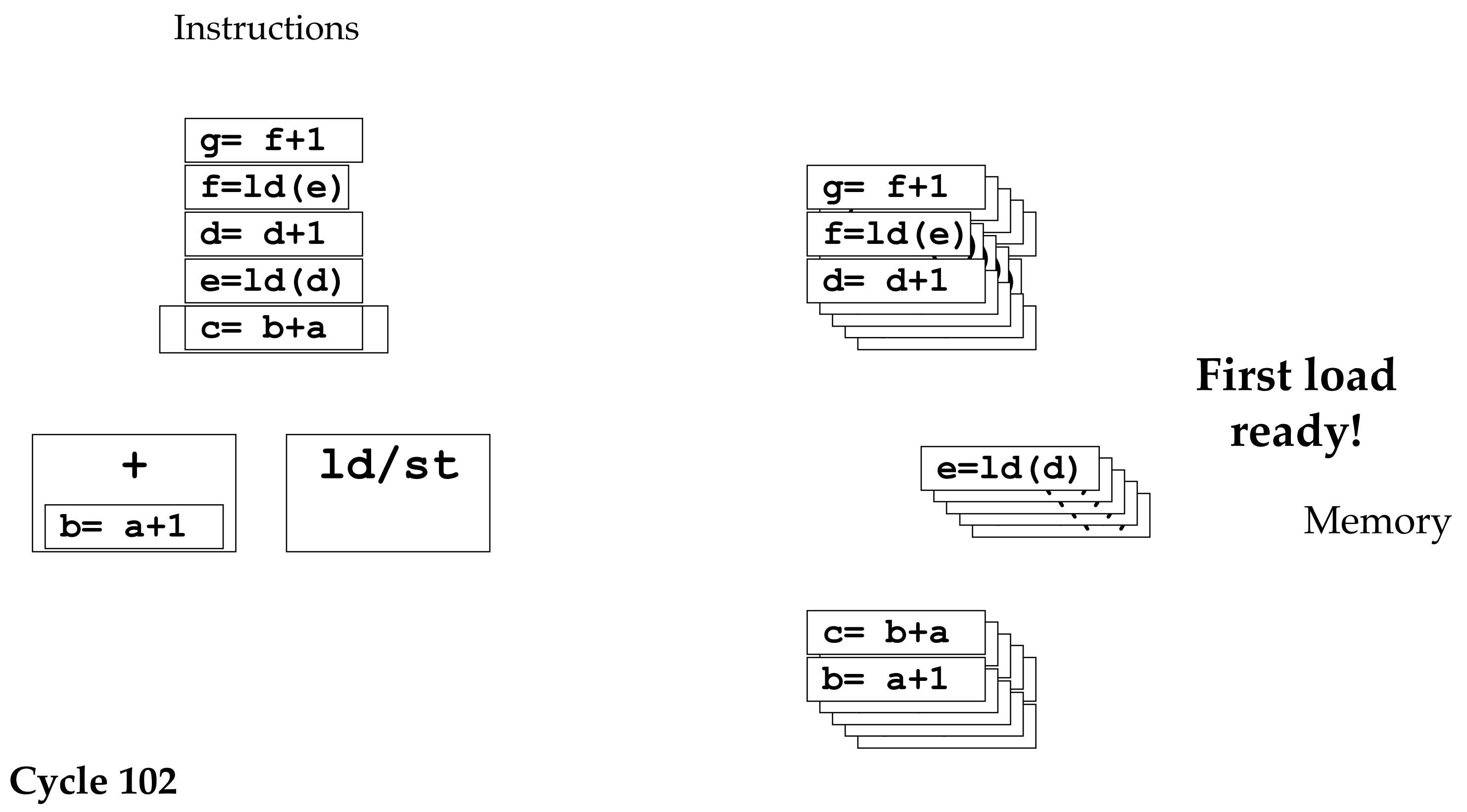
The GPU suspends it and finds another thread (yellow) to execute.

GPU Memory Philosophy



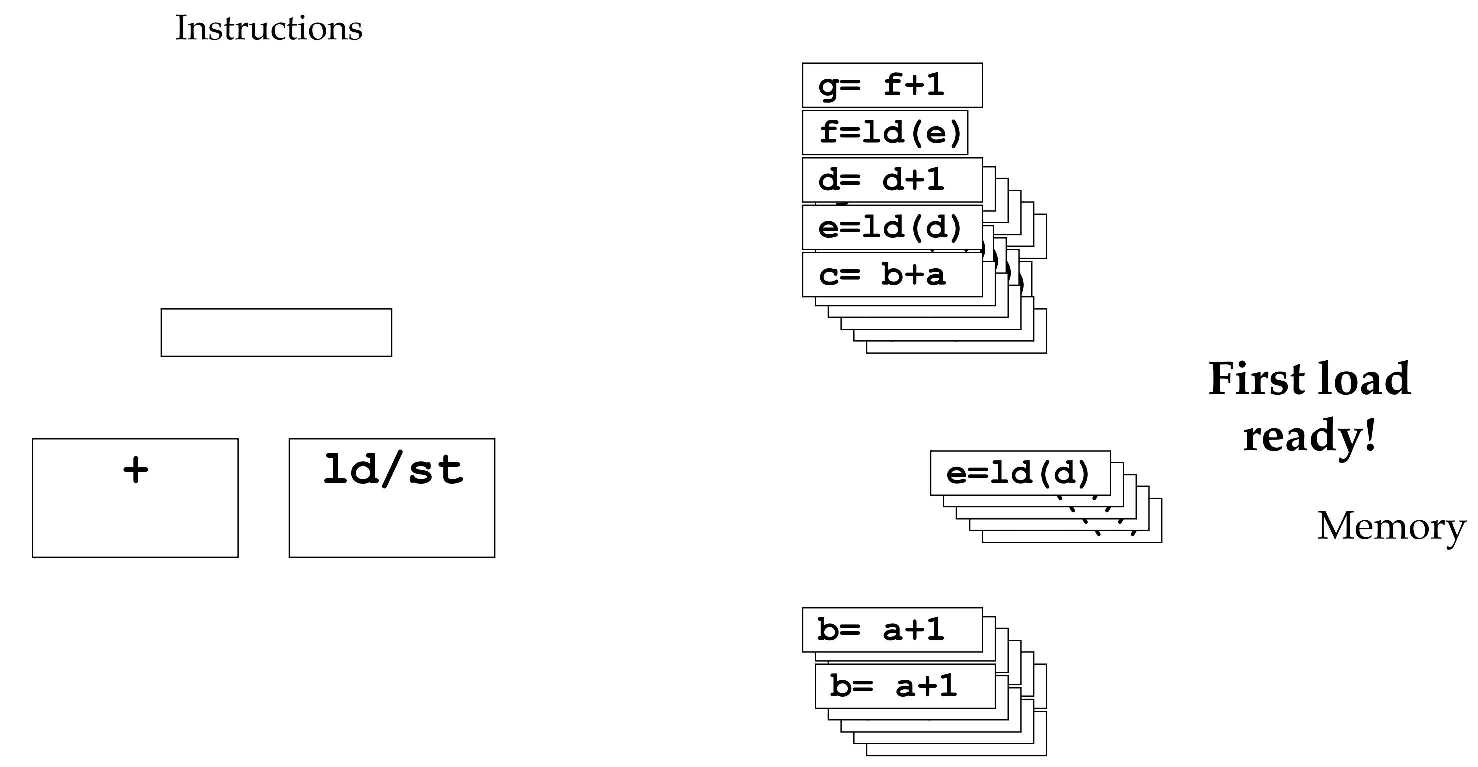
Cycle 6

GPU Memory Philosophy



After 100 cycles the memory is ready with the data for the first thread. Note: if we have enough threads (50*2 cycles before the read here) we have managed to keep the processor busy the whole time by swapping in new work rather than trying to reduce the latency of each thread. This allows high throughput, but also high latency.

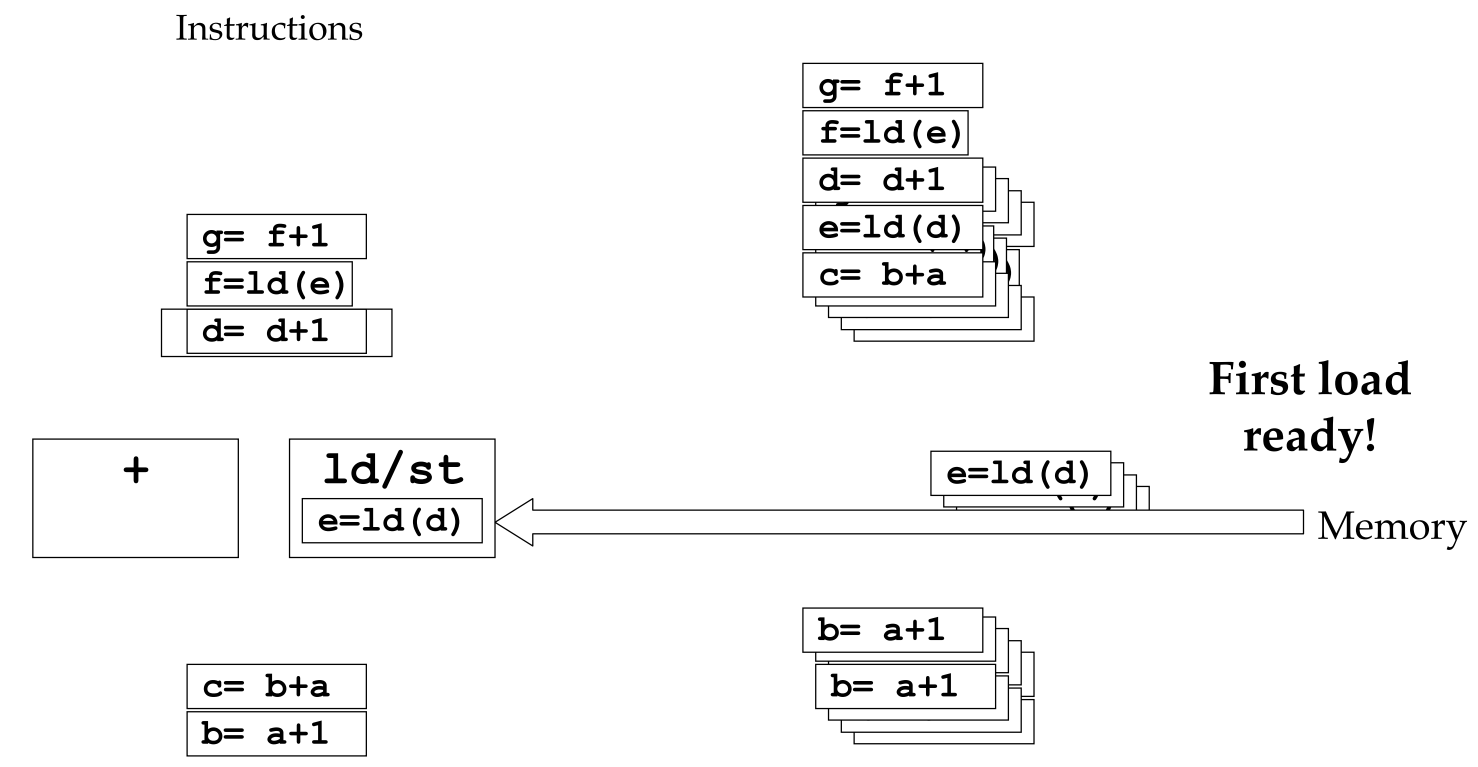
GPU Memory Philosophy



Cycle 103

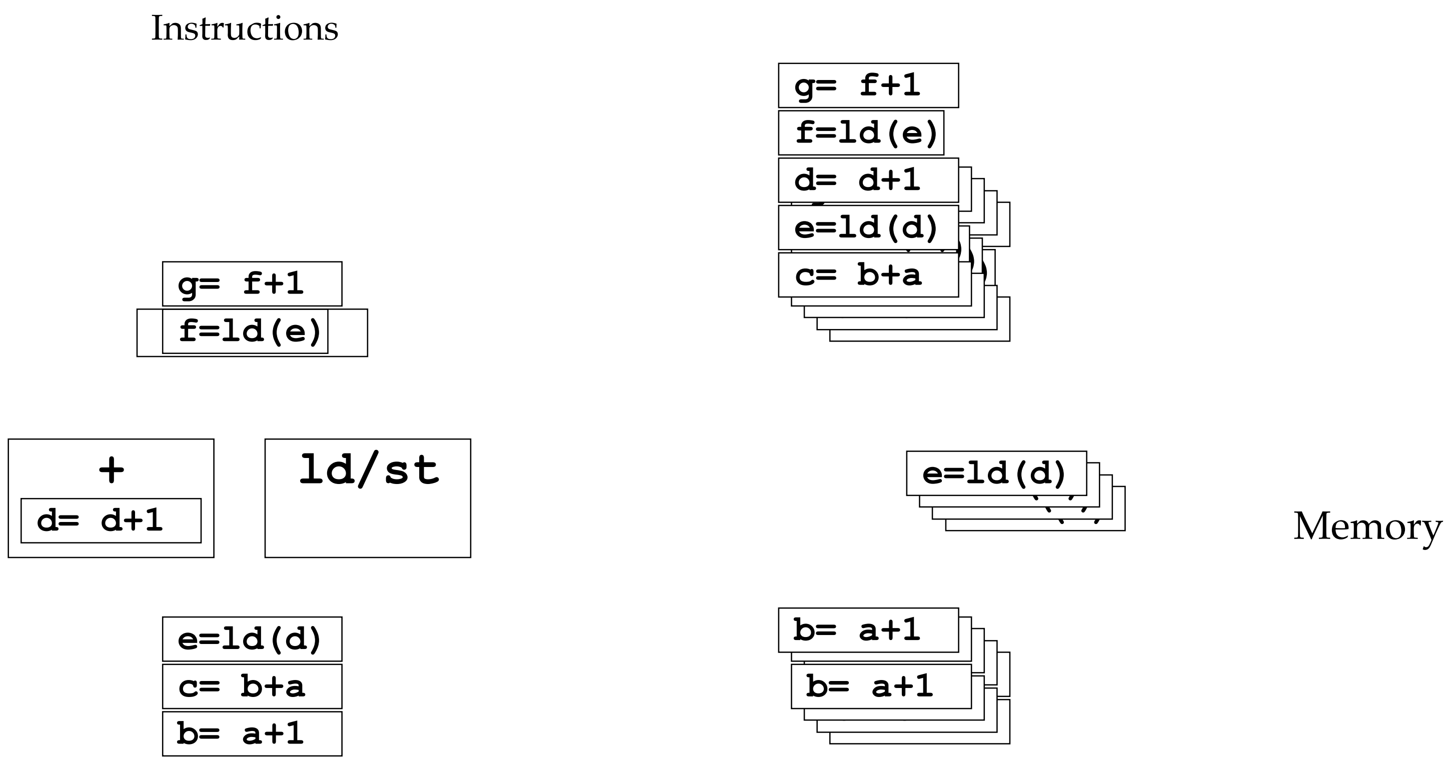
We can now suspend the current thread and go back to using the data from memory.

GPU Memory Philosophy



Cycle 103

GPU Memory Philosophy



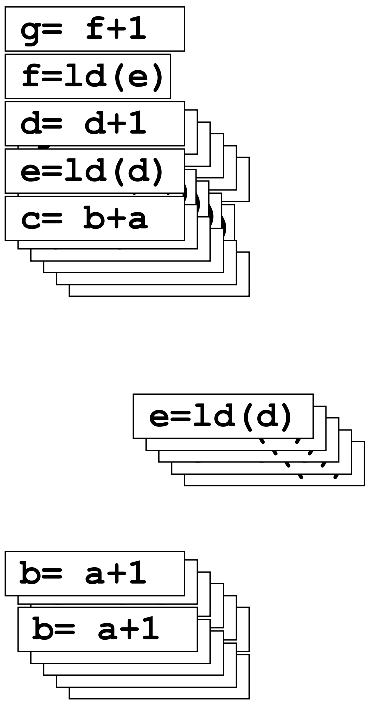
Cycle 104

At cycle 104 we've executed 1 instruction every single cycle despite a 100 cycle memory latency and no cache. But unlike the CPU, we've executed instructions from 50 different threads. Our total throughput is excellent, but no single thread has made more than 3 instructions of progress.

GPU Memory Philosophy

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

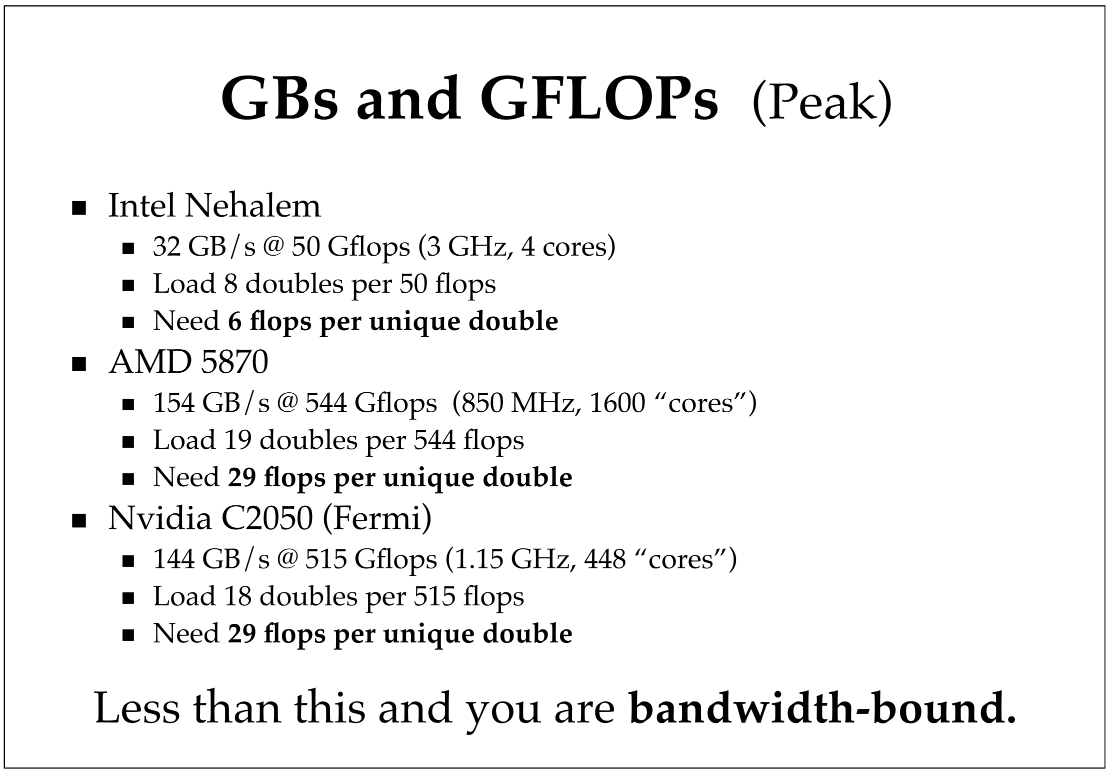
- As long as there is **enough work** in other threads **to cover latency** you get **high throughput**.



Notes:

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

This is why GPUs want thousands of threads for good performance. This is also how they are able to get good bandwidth: if you have lots and lots of threads running in parallel you can try to be clever about how you service all the memory accesses to try and take advantage of your DRAM system.



GBs and GFLOPs (Peak)

- Intel Nehalem
  - 32 GB/s @ 50 Gflops (3 GHz, 4 cores)
  - Load 8 doubles per 50 flops
  - Need **6 flops per unique double**
- AMD 5870
  - 154 GB/s @ 544 Gflops (850 MHz, 1600 "cores")
  - Load 19 doubles per 544 flops
  - Need **29 flops per unique double**
- Nvidia C2050 (Fermi)
  - 144 GB/s @ 515 Gflops (1.15 GHz, 448 "cores")
  - Load 18 doubles per 515 flops
  - Need **29 flops per unique double**

Less than this and you are **bandwidth-bound.**

So how did we do? Compare the peak performance numbers for some CPUs and GPUs. For the CPU, if you have fewer than 6 floating point operations per unique double loaded you are bandwidth bound. For GPUs that number is 29. (And they don't have large caches, which makes the "unique" part a lot less important.) This doesn't sound good…

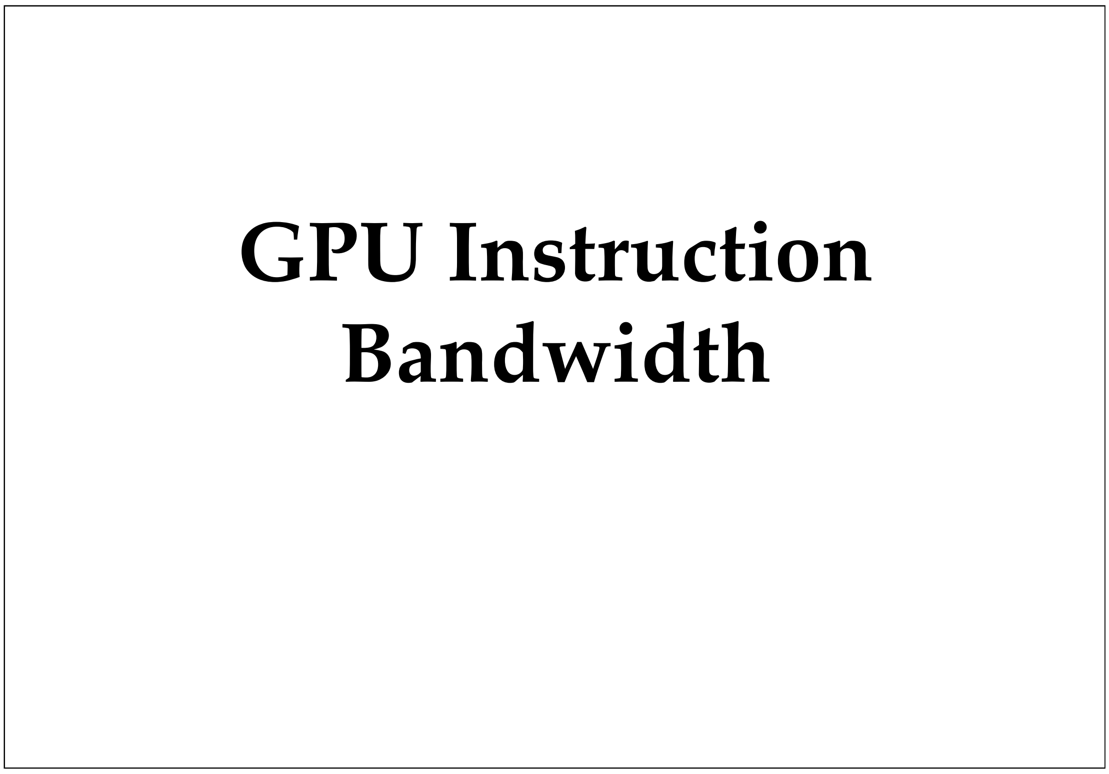
So Do GPUs Really Help?

- We went **from 6** flops per double (CPU) **to 29** flops per double (GPU). **Is this good?**
- **No! It's bad.** (If everything else was the same)
- GPU raw bandwidth is much higher:
  - 32GB/s (CPU) vs. 150GB/s (GPU)
  - If you can get 33%, which is better?
- Thousands of threads for throughput:
  - "flops" from one thread cover "loads" from another
  - Can do **lots** of useful work while waiting for memory

Easier to get a higher percentage of BW on GPU.

In fact, this isn't good.

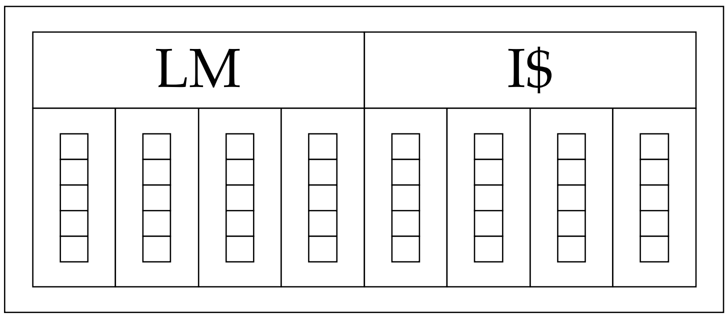
Except that GPUs have a much larger bandwidth, so even if you get only a fraction of it you're better off, plus they have hardware (lots of threads) to help you get a higher percentage of it.



GPU Instruction Bandwidth

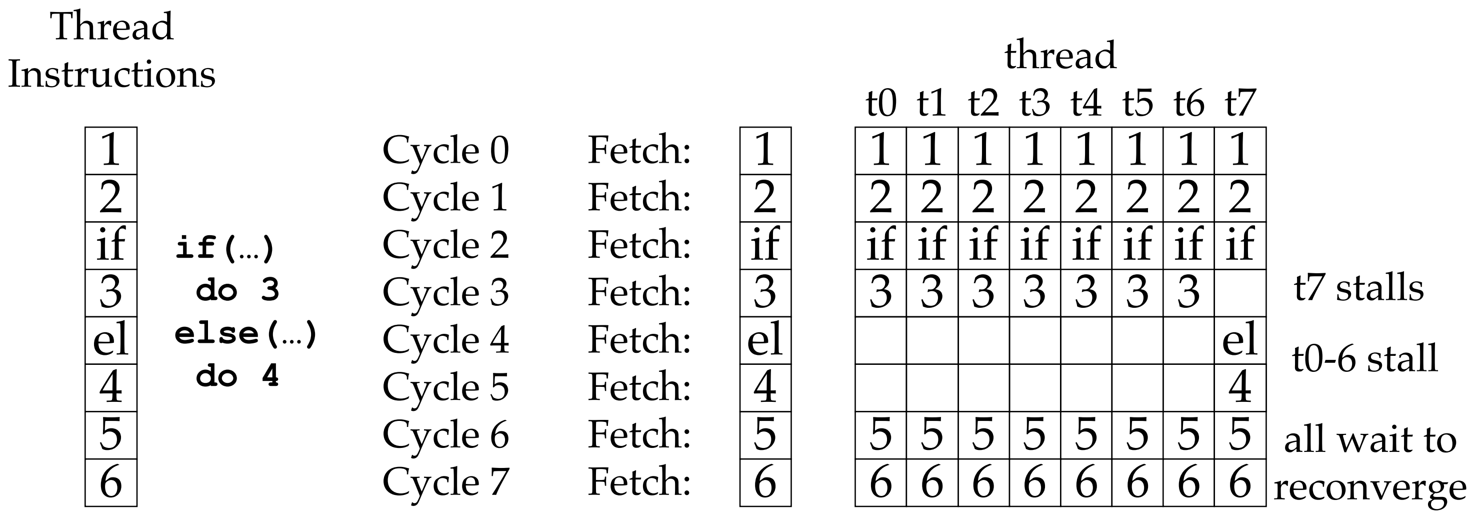
GPU Instruction Bandwidth

- 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)



Now we'll take a look at another one of the tradeoffs GPUs made in making more of their area compute and less of it support, namely instruction fetch.

Divergent Execution



Cycle	Fetch	t0	t1	t2	t3	t4	t5	t6	t7	
Cycle 0	Fetch: 1	1	1	1	1	1	1	1	1	
Cycle 1	Fetch: 2	2	2	2	2	2	2	2	2	
Cycle 2	Fetch: if	if	if	if	if	if	if	if	if	
Cycle 3	Fetch: 3	3	3	3	3	3	3	3		t7 stalls
Cycle 4	Fetch: el								el	t0-6 stall
Cycle 5	Fetch: 4								4	
Cycle 6	Fetch: 5	5	5	5	5	5	5	5	5	all wait to
Cycle 7	Fetch: 6	6	6	6	6	6	6	6	6	reconverge

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

If thread 7 takes the else branch and all the other threads take the if branch, the performance drops off terribly. This becomes far worse if each thread takes a different path through the code. The take-away lesson is that GPUs do support each thread doing its own thing, but not with good performance. This contrasts with a CPU which can fetch multiple instructions each cycle for a single thread. (Note that there are various tricks to deal with simple if/else cases, but if you have truly divergent execution you will experience bad performance.)

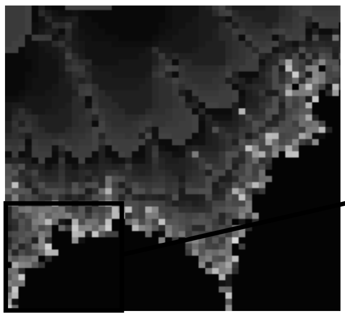
Divergent Execution for Real

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 happens in real-world applications. There is some support in current architectures to avoid the pain of simple if-then-else divergence, but beyond that performance falls off a cliff.

Instruction Divergence

- Some architectures are worse…
  - AMD's GPUs are 4-way SIMD
    If you don't process 4-wide vectors you lose.
  - Intel's Larabee is/was 16-way SIMD
    Theoretically the compiler could handle this.
- 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

Ignoring Instruction Divergence

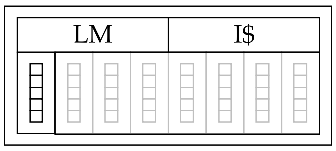
- What if you run 1 thread on each core?

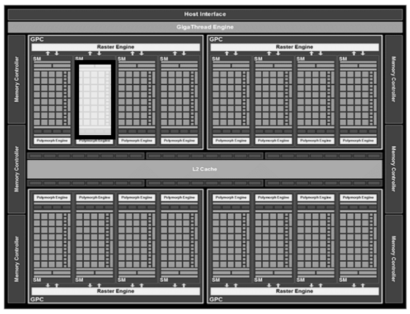
  One I-cache per thread…

  …no divergence problems…

  …but only tiny % of peak performance.

  (Still 8-16 1GHz+ cores. Not bad. E.g., M. Nanjundappa, *SCGPSim: A fast SystemC simulator on GPUs*)





You can ignore this by running one thread on each cluster of cores on a GPU. (SM in Nvidia parlance.) This sacrifices up to 15/16 of your performance, but then gives you one instruction fetch per cycle per core.

Questions?

Memory Philosophy
Instruction Philosophy

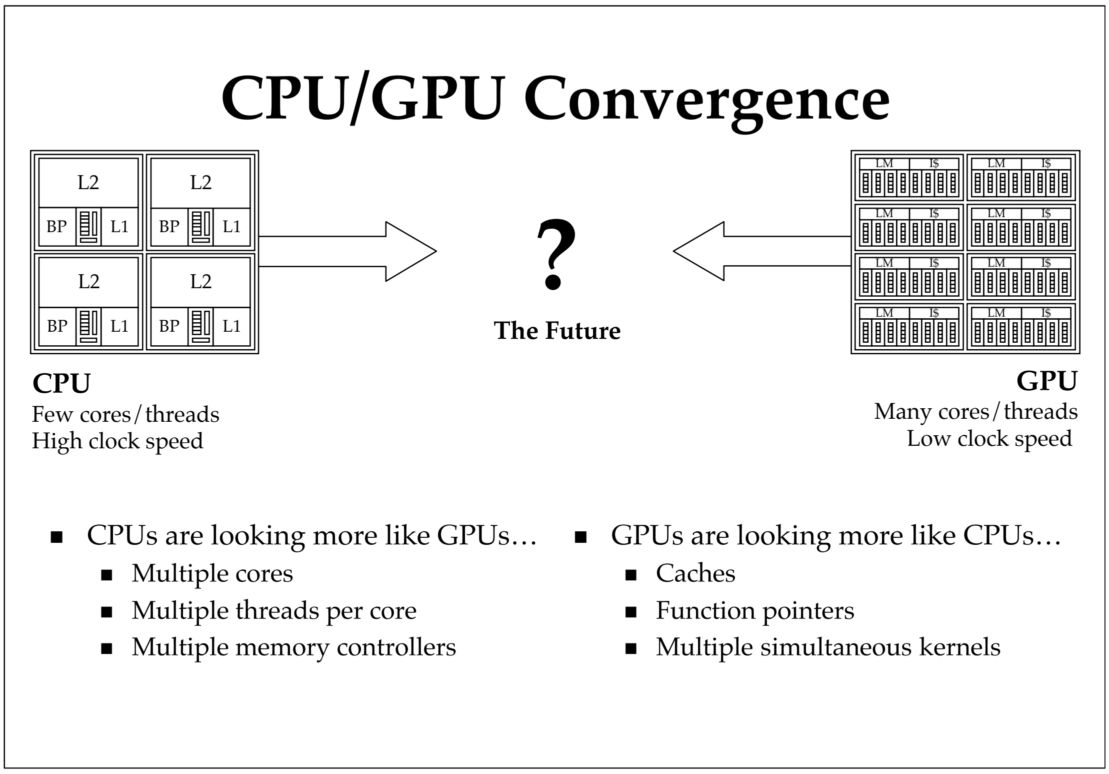
Summary & The Future

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

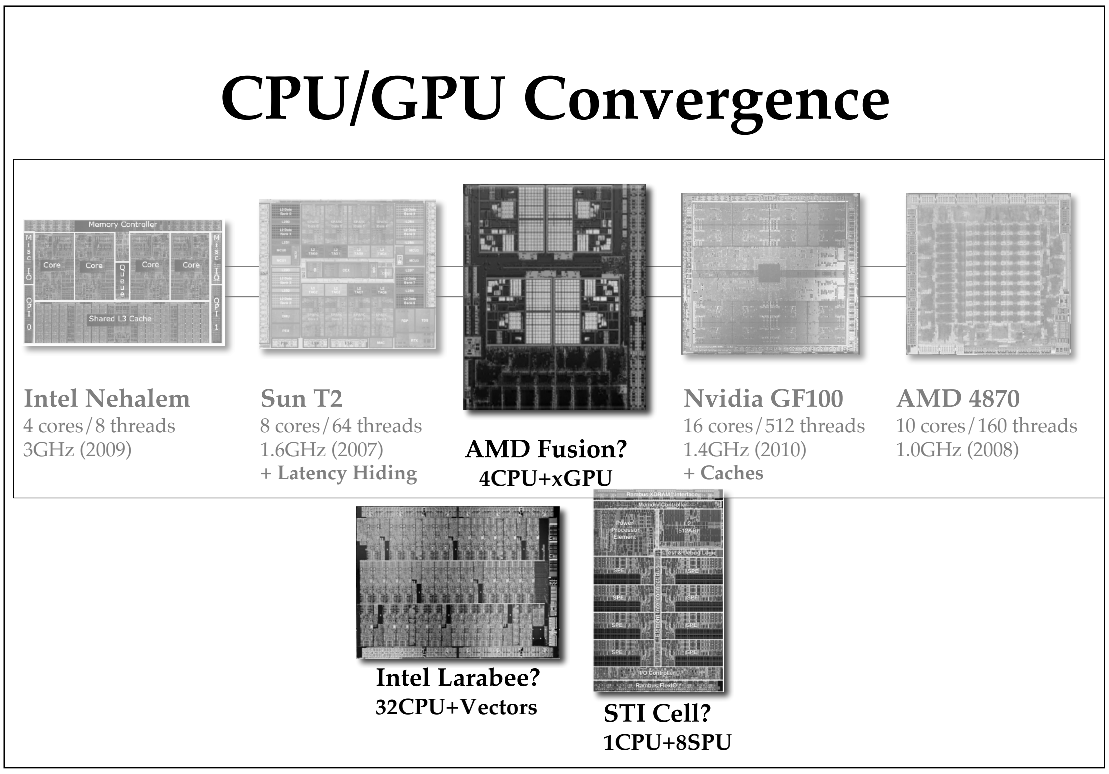
L2
BP
L1
?
The Future
LM
I$

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

Clearly things are moving together. Intel has (wasted?) spent a huge amount of money on Larabee to make their CPUs look more like GPUs. Nvidia keeps adding features that make their GPUs look more like CPUs, and AMD is busily trying to build a GPU and CPU on the same chip to avoid the whole convergence issue and get the best of both worlds.



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 Larabee?
32CPU+Vectors

STI Cell?
1CPU+8SPU

So where will we end up? AMD “got” there first with their Fusion processor, but no one has yet really tried it out. This will solve the bandwidth issues of getting data from main memory to the GPU and make GPUs far more useful for small frequent computations as opposed to just large bulk ones. Intel’s Larabee derivatives will undoubtedly influence their future cores with enhanced vector operations, scatter/gather, and higher degrees of multi-threading, but unless they move to a heterogeneous design I doubt it will be much of a player. Nvidia is in a difficult position unless they start putting x86 cores on their parts and selling the whole system or invest in a lot of binary translation. Interestingly, Cell demonstrated a hybrid system many years ago, and developers were able to get huge performance wins out of it due to the tightly-coupled memory system between the GPU-like SPUs and the CPU.

It’s worth noting that the largest GPU manufacturer today (Intel) does not support OpenCL on its GPUs, and even if it did, the performance would be so bad no one would use it.

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?