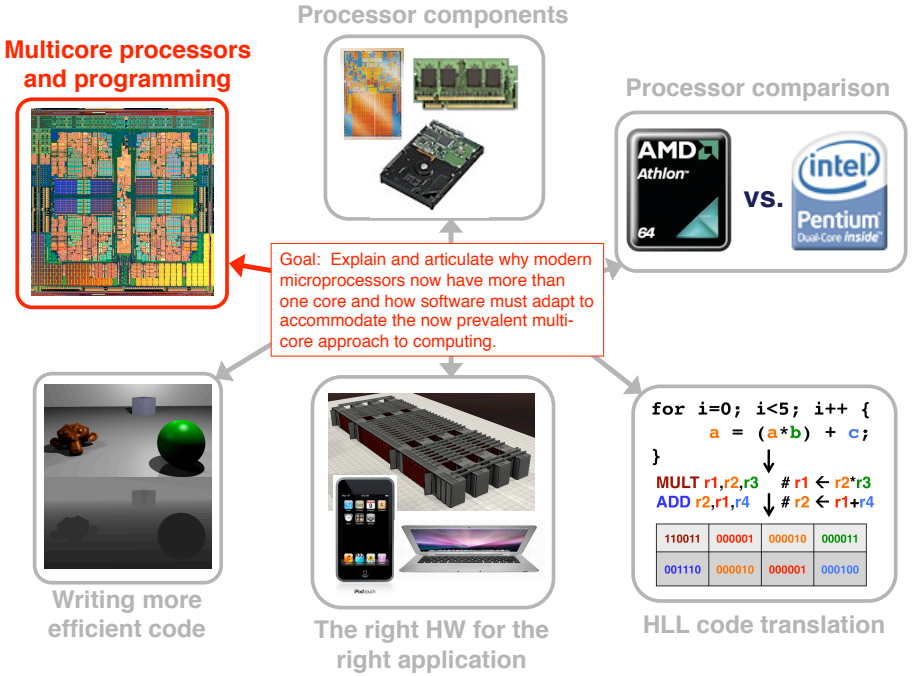
Lecture 23
Introduction to Parallel Processing

Suggested Readings

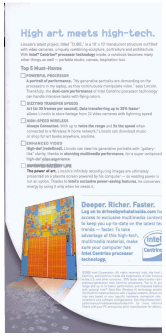
- Readings
  - H&P: Chapter 7
    - (Over next 2 weeks)

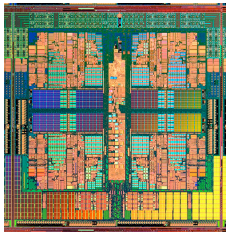


Introduction and Overview

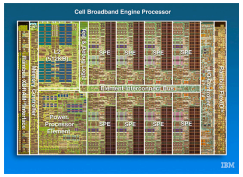
General context: Multiprocessors

- Multiprocessor is any computer with several processors

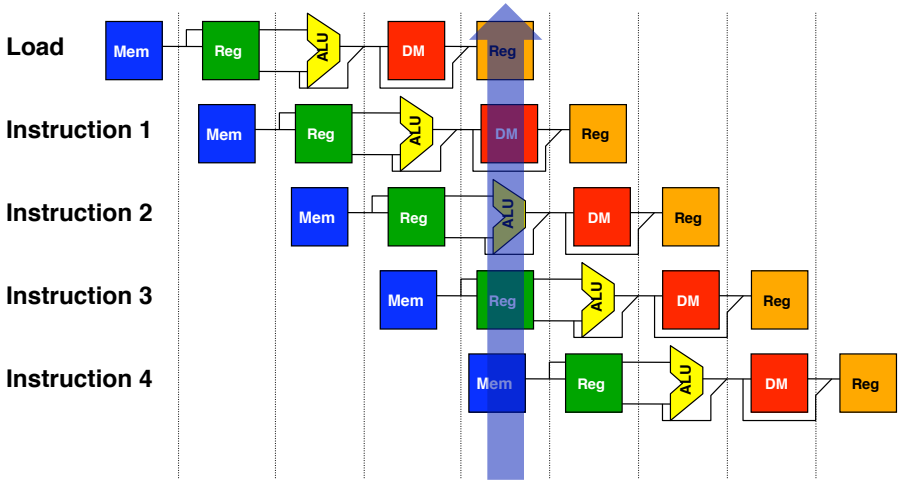




Lemieux cluster, Pittsburgh supercomputing center



Pipelining and "Parallelism"



Instructions execution overlaps (psuedo-parallel) but instructions in program issued sequentially.

Multiprocessing (Parallel) Machines

- Flynn's Taxonomy of Parallel Machines
  - How many Instruction streams?
  - How many Data streams?
- SISD: Single I Stream, Single D Stream
  - A uni-processor
  - Could have psuedo-parallel execution here
  - Example: Intel Pentium 4
- SIMD: Single I, Multiple D Streams
  - Each "processor" works on its own data
  - But all execute the same instructions in lockstep
  - Example: SSE instructions in Intel x86
    - (e.g. for vector addition in graphics processing)

Flynn's Taxonomy

- MISD: Multiple I, Single D Stream
  - Not used much, no example today
- MIMD: Multiple I, Multiple D Streams
  - Each processor executes its own instructions and operates on its own data
  - Pre multi-core, typical off-the-shelf multiprocessor (made using a bunch of "normal" processors)
    - Not superscalar
    - Each node is superscalar
    - Lessons will apply to multi-core too!

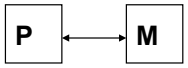
} What's superscalar?

  - Example: Intel Xeon e5345

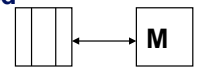
```
lw r2, 0(R1)     # page fault
add r3, r2, r2   # waits
sub r6, r7, r8   # starts
```

In Pictures:

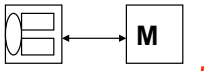
- Uni:



- Pipelined

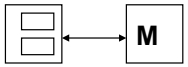


- Superscalar

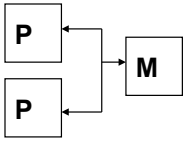


SISD

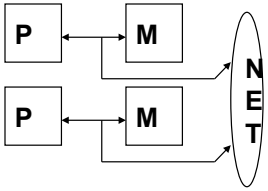
- VLIW/"EPIC"



- Centralized Shared Memory



- Distributed Shared Memory



MIMD

Digression:
More on SIMD (Superscalar)

What's Superscalar?

Dynamic Scheduling: Motivation

	1	2	3	4	5	6	7	8	9	10
`divf f0,f2,f4`	F	D	E/	E/	E/	E/	W			
`addf f6,f0,f2`		F	D	d*	d*	d*	E+	E+	W	
`mulf f8,f2,f4`			F	p*	p*	p*	D	E*	E*	W

- cycle4: **`addf`** stalls due to *RAW hazard*
  - OK, fundamental problem
- also cycle4: **`mulf`** stalls due to *pipeline hazard* (**`addf`** stalls)
  - why? **`mulf`** can't proceed into ID because **`addf`** is there
  - but that's the only reason ⇒ not good enough!
- why can't we decode **`mulf`** in cycle 4 and execute it in c5?
  - no fundamental reason why we can't do this!

What's Superscalar?

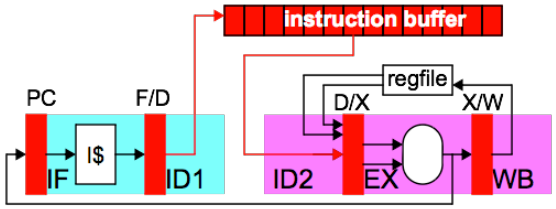
Dynamic Scheduling

dynamic scheduling (out-of-order execution)

- execute instructions in non-sequential (non-vonNeumann) order
  - + reduce stalls
  - + improve functional unit utilization
  - + enable parallel execution (not in-order ⇒ can be in parallel)
- make it *appear* like sequential execution: precise interrupts
  - very important
  - – but hard

How Superscalar Machines Work

Instruction Buffer

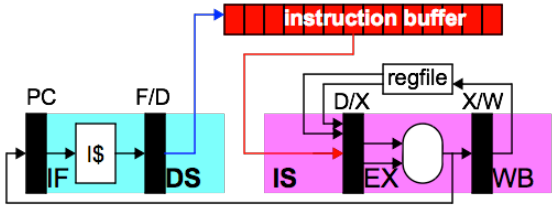


trick: *instruction buffer* (many names for this buffer)

- basically: a bunch of latches for holding instructions
  - this is the scope of instructions that the scheduler can see
- split ID into two pieces
  - accumulate decoded instructions in buffer in-order
  - buffer sends instructions down rest of pipe out-of-order

How Superscalar Machines Work

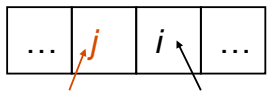
Dispatch and Issue



- dispatch (DS): first part of ID
  - allocate resources in instruction buffer
  - – new kind of structural hazard (instruction buffer could be full)
  - *dispatch is in-order, and stall propagates to younger instructions*
- issue (IS): second part of ID
  - send instructions from instruction buffer to execution units
  - *out-of-order, wait does NOT propagate to younger instructions*

Superscalar Introduces New Hazards

- **With WAW hazard, instruction *j* tries to write an operand before instruction *i* writes it.**
- **The writes are performed in wrong order leaving the value written by earlier instruction**
- **Graphically/Example:**



Instruction j is a write instruction issued after i

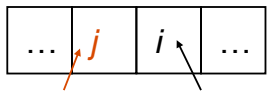
Instruction *i* is a write instruction issued before *j*

i: DIV F1, F2, F3
j: SUB F1, F4, F6

Problem: If *j* after *i*, and *j* finishes first, and then *i* writes to F1, the wrong value is in F1.

Superscalar Introduces New Hazards

- **With WAR hazard, instruction *j* tries to write an operand before instruction *i* reads it.**
- **Instruction *i* would incorrectly receive newer value of its operand;**
  - **Instead of getting old value, it could receive some newer, undesired value:**
- **Graphically/Example:**



Instruction j is a write instruction issued after i

Instruction *i* is a read instruction issued before *j*

i: DIV F7, F1, F3
j: SUB F1, F4, F6

Problem: what if instruction *j* completes before instruction *i* reads F1?

Solution: Register Renaming

Freeing Registers in R10K

raw instruction	map table r1	map table r2	map table r3	free locations	renamed instruction
	11	12	13	14,15,16,17	
add r1,r2,r3	14	12	13	15,16,17	add 14,12,13
sub r3,r2,r1	14	12	15	16,17	sub 15,12,14
mul r1,r2,r3	16	12	15	17	mul 16,12,15
div r2,r1,r3	16	17	15		div 17,16,15

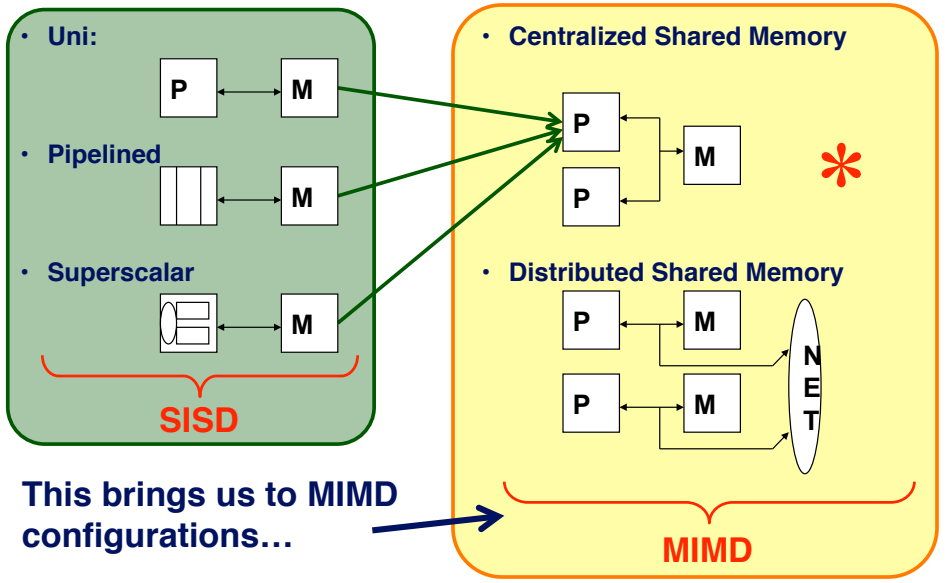
- when **add** commits: free **11**
- when **sub** commits: free **13**
- when **mul** commits: free **?**
- when **div** commits: free **?**
- see the pattern?

Board Example: Superscalar Trace

MIMD Machine Types

What's Next?

Processors can be any of these configurations



This brings us to MIMD configurations…

Multiprocessors

- Why did/do we need multiprocessors?
  - Uni-processor speed improved fast
  - But there are problems that needed even more speed
    - Before: Wait for a few years for Moore's law to catch up?
      - Or use multiple processors and do it sooner?
    - Now: Moore's Law still catching up.

- Multiprocessor software problem
  - Most code is sequential (for uni-processors)
    - MUCH easier to write and debug
  - Correct parallel code very, very difficult to write
    - Efficient and correct is much more difficult
    - Debugging even more difficult

Let's look at example MIMD configurations…

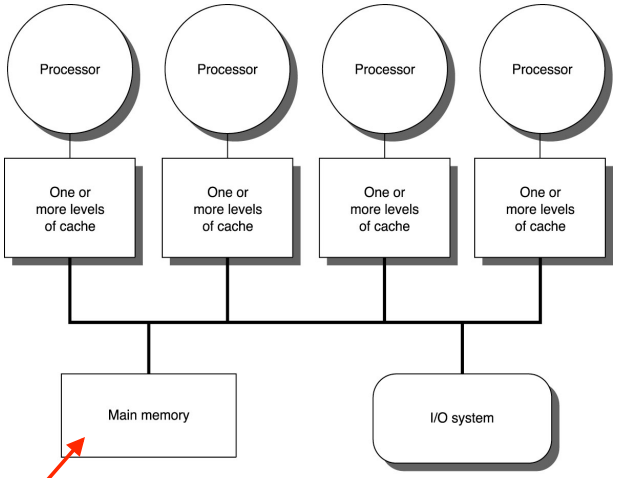
Multiprocessor Memory Types

- Shared Memory:
  - In this model, there is one (large) common shared memory for all processors

- Distributed memory:
  - In this model, each processor has its own (small) local memory, and its content is not replicated anywhere else

MIMD Multiprocessors

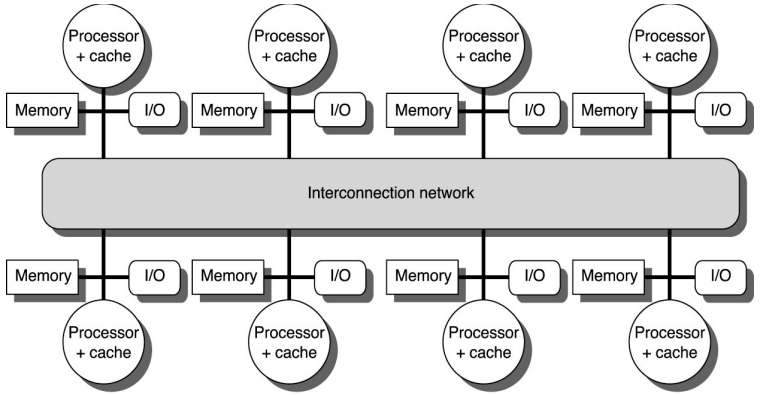
Centralized Shared Memory



Note: just 1 memory

MIMD Multiprocessors

Distributed Memory



Multiple, distributed memories here.

More on Centralized-Memory Machines

- "Uniform Memory Access" (UMA)
  - All memory locations have similar latencies
  - Data sharing through memory reads/writes
  - P1 can write data to a physical address A, P2 can then read physical address A to get that data
- Problem: Memory Contention
  - All processor share the one memory
  - Memory bandwidth become bottlenecks
  - Used only for smaller machines
    - Most often 2,4, or 8 processors
    - Up to 16-32 processors

More on Distributed-Memory Machines

- Two kinds
  - Distributed Shared-Memory (DSM)
    - All processors can address all memory locations
    - Data sharing possible
    - Also called NUMA (non-uniform memory access)
    - Latencies of different memory locations can differ
      - (local access faster than remote access)
  - Message-Passing
    - A processor can directly address only local memory
    - To communicate with other processors, must explicitly send/receive messages
- Most accesses local, so less memory contention (can scale to well over 1000 processors)

Message-Passing Machines

- A cluster of computers
  - Each with its own processor and memory
  - An interconnect to pass messages between them
  - Producer-Consumer Scenario:
    - P1 produces data D, uses a SEND to send it to P2
    - The network routes the message to P2
    - P2 then calls a RECEIVE to get the message
  - Two types of send primitives
    - Synchronous: P1 stops until P2 confirms receipt of message
    - Asynchronous: P1 sends its message and continues
  - Standard libraries for message passing: Most common is MPI – Message Passing Interface

Message Passing Pros and Cons

- Pros
  - Simpler and cheaper hardware
  - Explicit communication makes programmers aware of costly (communication) operations
- Cons
  - Explicit communication is painful to program
  - Requires manual optimization
    - If you want a variable to be local and accessible via LD/ST, you must declare it as such
    - If other processes need to read or write this variable, you must explicitly code the needed sends and receives to do this

Message Passing: A Program

- Calculating the sum of array elements

```
#define ASIZE 1024
#define NUMPROC 4
double myArray[ASIZE/NUMPROC];
double mySum=0;
for(int i=0;i<ASIZE/NUMPROC;i++)
  mySum+=myArray[i];
if(myPID=0){
  for(int p=1;p<NUMPROC;p++){
    int pSum;
    recv(p,pSum);
    mySum+=pSum;
  }
  printf("Sum: %lf\n",mySum);
}else
  send(0,mySum);
```

Must manually split the array

"Main" processor adds up partial sums and prints the result

Other processors send their partial results to main

Shared Memory Pros and Cons

- Pros
  - Communication happens automatically
  - More natural way of programming
    - Easier to write correct programs and gradually optimize them
  - No need to manually distribute data (but can help if you do)
- Cons
  - Needs more hardware support
  - Easy to write correct, but inefficient programs (remote accesses look the same as local ones)

Shared Memory: A Program

- Calculating the sum of array elements

```
#define ASIZE 1024
#define NUMPROC 4
shared double array[ASIZE];
shared double allSum=0;
shared mutex sumLock;
double mySum=0;
for(int i=myPID*ASIZE/NUMPROC;i<(myPID+1)*ASIZE/NUMPROC;i++)
  mySum+=array[i];
lock(sumLock);
allSum+=mySum;
unlock(sumLock);
if(myPID=0)
  printf("Sum: %lf\n",allSum);
```

Array is shared

Each processor sums up "its" part of the array

Each processor adds its partial sums to the final result

Main processor prints the result

The Challenge(s) of Parallel Computing

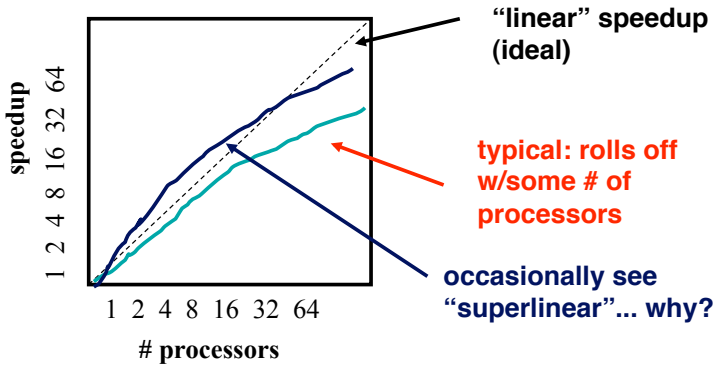
Real Issues and Problems

- Motivation is 2 fold
  - 50s ,60s, 70s, … today – have rooms filled with machines working to solve a common problem
  - Could consider multi-core chips as "DSM"
    - i.e. 4-core chip, each core has L1 cache, share L2 cache
- Problems are often the same, just on different scales
- As technology scales, problems once seen at room level can be found "on chip"

Speedup

metric for performance on latency-sensitive applications

- Time(1) / Time(P) for P processors
  - note: must use the best ***sequential*** algorithm for Time(1) -- the parallel algorithm may be different.



Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!
- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead

Challenges: Cache Coherency

- Shared memory easy with no caches
  - P1 writes, P2 can read
  - Only one copy of data exists (in memory)
- Caches store their own copies of the data
  - Those copies can easily get inconsistent
  - Classical example: adding to a sum
    - P1 loads allSum, adds its mySum, stores new allSum
    - P1's cache now has dirty data, but memory not updated
    - P2 loads allSum from memory, adds its mySum, stores allSum
    - P2's cache also has dirty data
    - Eventually P1 and P2's cached data will go to memory
    - Regardless of write-back order, final value ends up wrong

If moderate # of nodes, write-through not practical.

Challenges: Contention

- Contention for access to shared resources - esp. memory banks or remote elements - may dominate overall system scalability
  - The problem:
    - Neither network technology nor chip memory bandwidth has grown at the same rate as the processor execution rate or data access demands
  - With success of Moore's Law:
    - Amt. of data per memory chip is growing such that it takes an increasing # of CCs to touch all bytes per chip at least once
      - Imposes a fundamental bound on system scalability
      - Is a significant contributor to single digit performance efficiencies by many of today's large scale apps.

Challenges: Latency

- …is already a major source of performance degradation
  - Architecture charged with hiding local latency
    - (that's why we talked about registers, caches, IC, etc.)
  - Hiding global latency is task of programmer
    - (I.e. manual resource allocation)
- Today:
  - multiple clock cycles to cross chip
  - access to DRAM in 100s of CCs
  - round trip remote access in 1000s of CCs
- In spite of progress in NW technology, *increases* in clock rate may cause delays to reach 1,000,000s of CCs in worst cases

Challenges: Reliability

- Reliability:
  - Improving yield and achieving high "up time"
    - (think about how performance might suffer if one of the 1 million nodes fails every x number of seconds or minutes…)
  - Solve with checkpointing, other techniques
- Programming languages, environments, & methodologies:
  - Need simple semantics and syntax that can also expose computational properties to be exploited by large-scale architectures

Challenges: Languages

- Programming languages, environments, & methodologies:
  - Need simple semantics and syntax that can also expose computational properties to be exploited by large-scale architectures

Perspective: The Future of Supercomputing

Perspective: Parallel Processing & "Supercomputing"

- Need more high performance supercomputing capability
  - FLOP History:
    - MegaFLOPS 1970s
    - GigaFLOPS 1980s
    - TeraFLOPS 1990s (1994)
    - PetaFLOPS (2010) (ish)
  - About 3 orders of magnitude every 12 years…
- Next target
  - ExaFLOPS (10^{18} FLOPS)
- Ultimate limit?
  - ZettaFLOPS (10^{21} FLOPS)
    - Can we get to 10^{21} FLOPS?

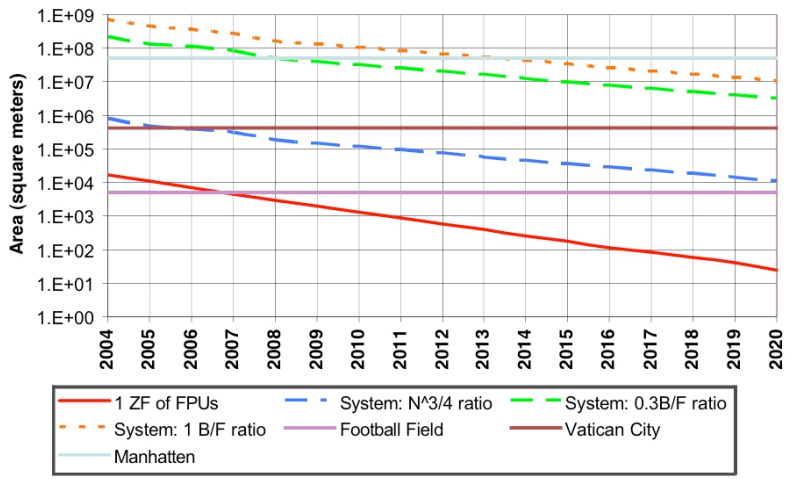
Parallel processing enables solutions to important problems.

- Why zettaFLOPS?
  - More computational capacity needed for science, national defense, society as a whole…
- Good example:
  - Climate modeling…
    - Climate modelers want - actually should say need - 1021 FLOPS to do accurate modeling…
    - …Ideally with a program-based model…

A silicon zettaflop won't be easy. (technology still can limit)

- As just mentioned, to get to a ZF, also need to consider storage?
- Thus, question #1: how much is enough?
  - Actually some debate about this…
    - Option 1: 1 ZB for 1 ZF
    - Option 2: 0.3 ZB / ZF
    - Option 3:
      - Gigabytes should equal at least sustained GF to the 3/4th power
      - This leads to 1 EB / ZF
  - We'll consider all of these…

It's big.



Note: this is the BEST case…

It's hot.