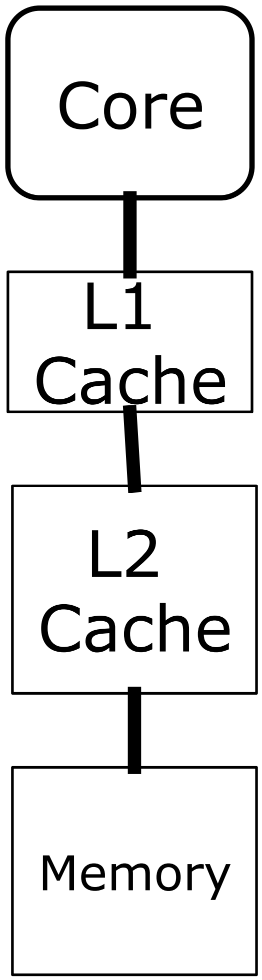
Lecture 16: Threads

William Gropp
www.cs.illinois.edu/~wgropp

Add to the (Model) Architecture

- What do you do with a billion transistors?
  - For a long time, try to make an individual processor (what we now call a *core*) faster
  - Increasingly complicated hardware yielded less and less benefit (speculation, out of order execution, prefetch, …)
- An alternative is to simply put multiple processing elements (cores) on the same chip
- Thus the "multicore processor" or "multicore chip"

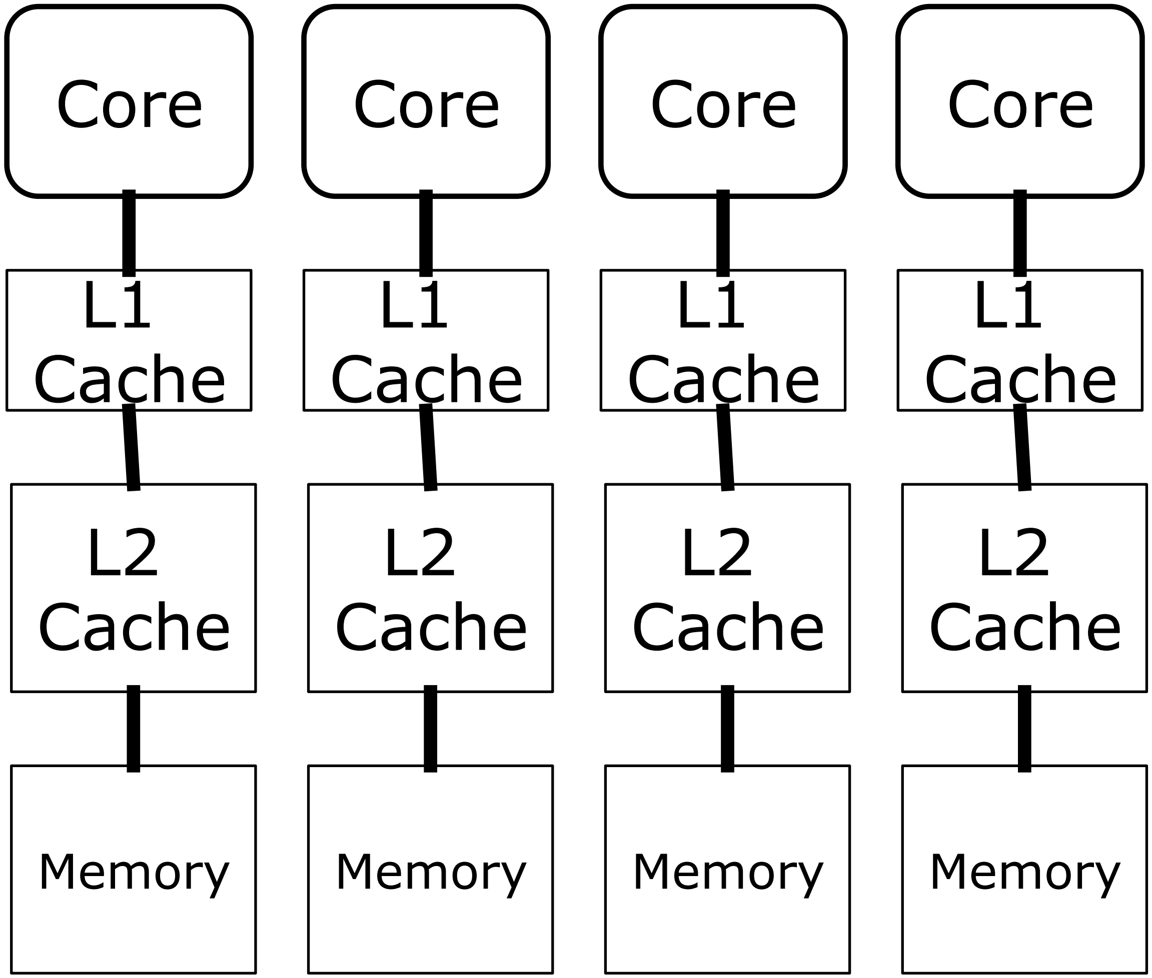
Adding Processing Elements



- Here's our model so far, with the vector and pipelining part of the "core"
  - Most systems today have an L3 cache as well)
- We can (try to) replicate everything…

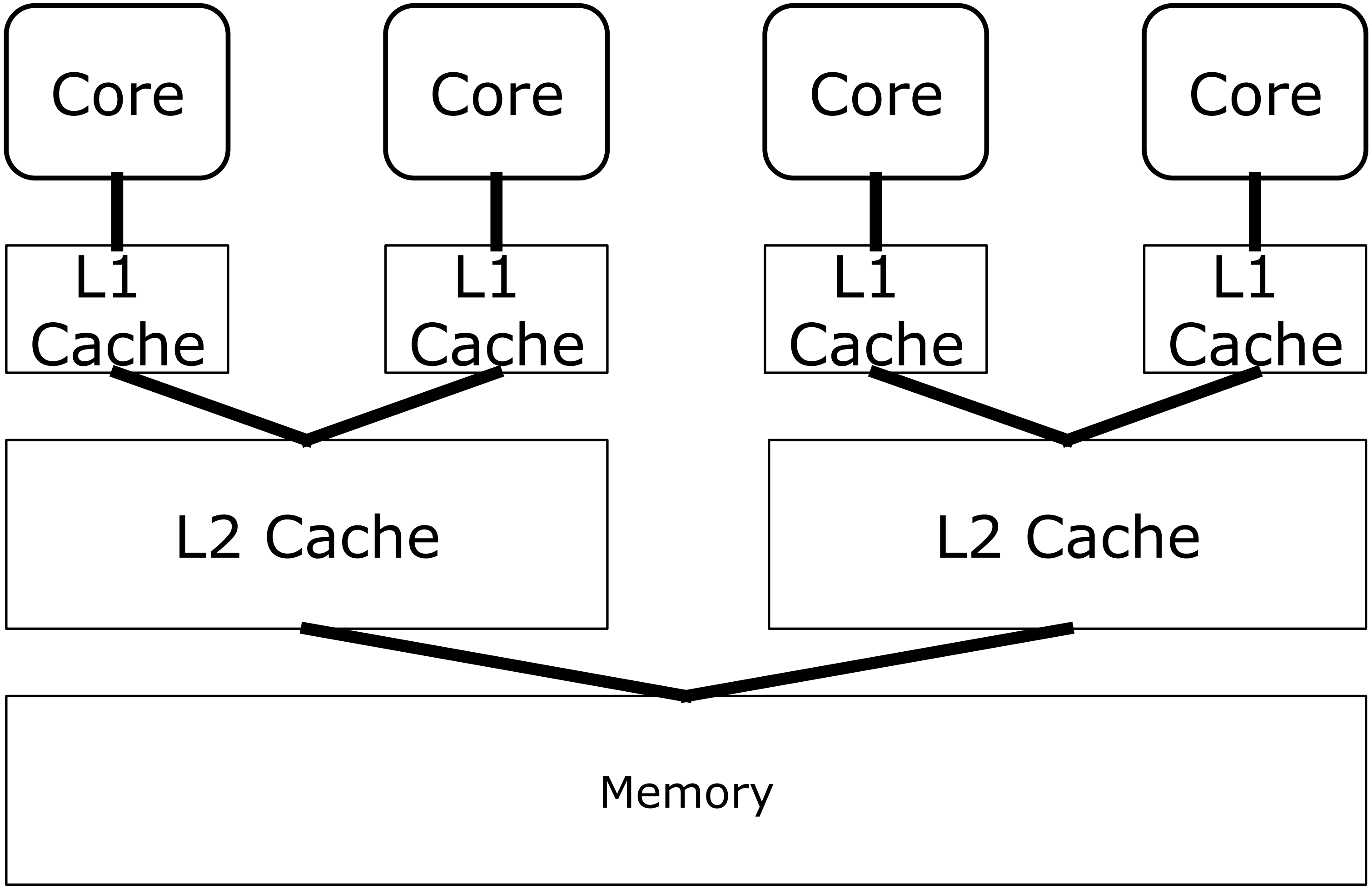
PARALLEL@ILLINOIS

Adding Processing Elements



- Something like this would be simple
- But in practice, some resources are shared, giving us…

Adding Processing Elements



Notes on Multicore

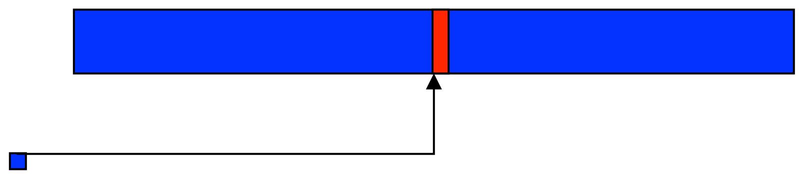
- Some resources are *shared*
  - ♦ Typically the larger (slower) caches, path to memory
  - ♦ May share functional units *within the core* (variously called simultaneous multithreading (SMT) or hyperthreading)
  - ♦ Rarely enough bandwidth for shared resources (cache, memory) to supply all cores *at the same time*.
- Variations trade complexity of core with number of cores
  - ♦ Manycore vs. Multicore

Programming Models For Multicore processors

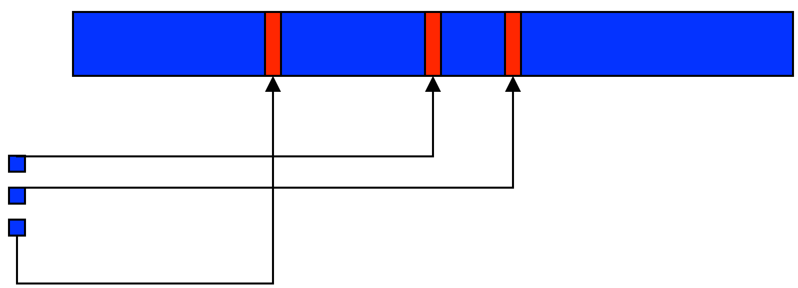
- Parallelism within a process
  - ♦ Compiler-managed parallelism
    - Transparent to programmer
    - Rarely successful
- Threads
  - ♦ Within a process, all memory shared
  - ♦ Each "thread" executes "normal" code
  - ♦ Many subtle issues (more later)
- Parallelism between processes within a node covered later

What are Threads?

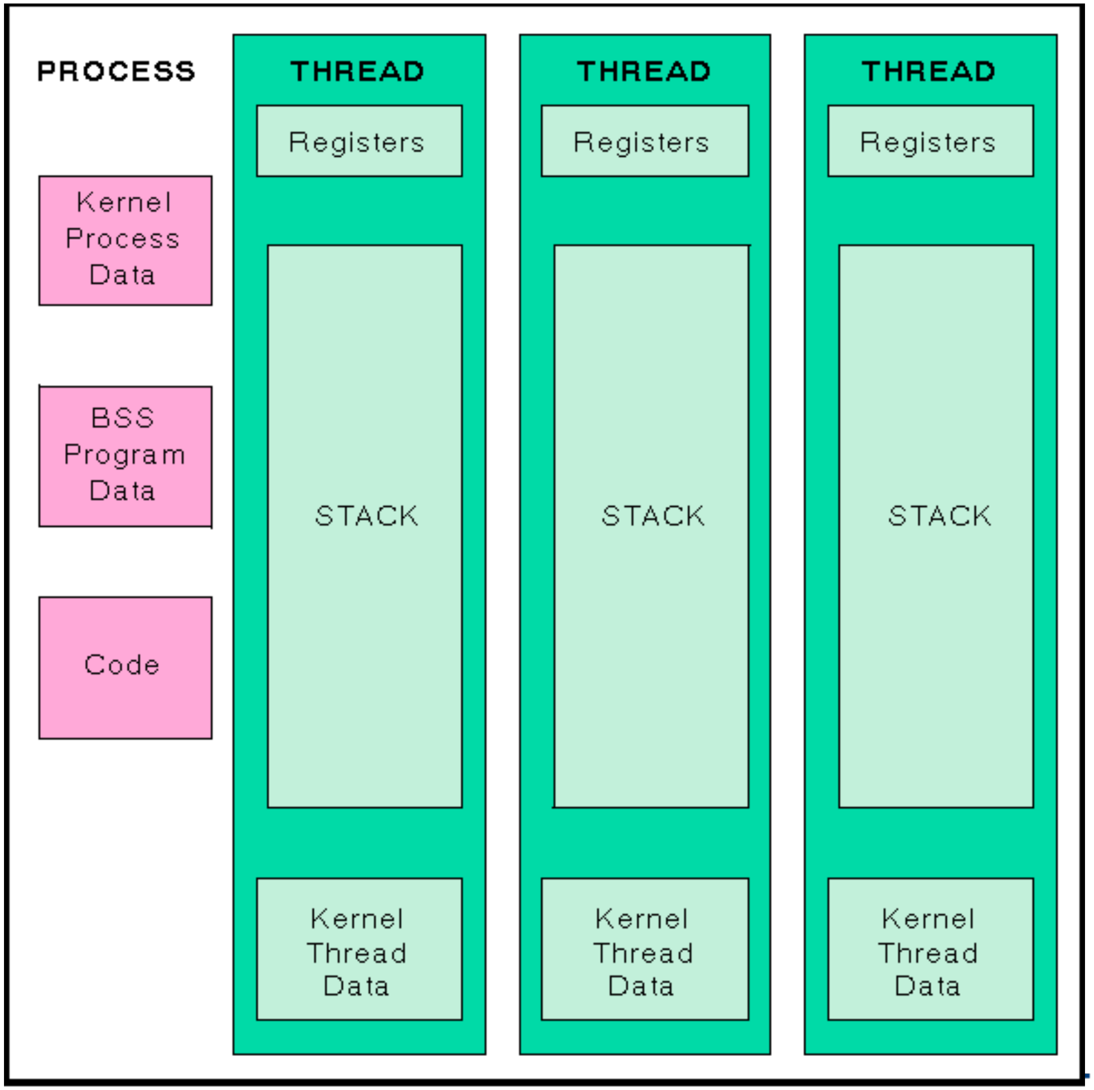
- Executing program (process) is defined by
  - Address space
  - Program Counter



- Threads are multiple program counters



Inside a Thread's Memory



Kinds of Threads

- Almost a process
  - Kernel (Operating System) schedules
  - Each thread can make independent system calls
- Co-routines and lightweight processes
  - User schedules (sort of…)
- Memory references
  - Hardware schedules

Kernel Threads

- System calls (e.g., read, accept) block calling thread but not process
- Alternative to “nonblocking” or “asynchronous” I/O:
  - create_thread
    thread calls blocking read
- Can be expensive (many cycles to start, switch between threads)

User Threads

- System calls (may) block all threads in process
- Allows multiple cores to cooperate on data operations
  - loop: create # threads = # cores - 1
    each thread does part of loop
- Cheaper than kernel threads
  - Still must save registers (if in same core)
  - Parallelism requires OS to schedule threads on different cores

Hardware Threads

- Hardware controls threads
- Allows single core to interleave memory references and operations
  - ♦ Unsatisfied memory reference changes thread
  - ♦ Separate registers for each thread
- Single cycle thread switch with appropriate hardware
  - ♦ Basis of Tera MTA computer http://www.tera.com Now YarcData Urika
  - ♦ Like kernel threads, replaces nonblocking hardware operations - multiple pending loads
  - ♦ Even lighter weight—just change program counter (PC)

Simultaneous Multithreading (SMT)

- Share the functional units in a single core
  - ♦ Remember the pipelining example – not all functional units (integer, floating point, load/store) are busy each cycle
  - ♦ SMT idea is to have two threads sharing a single set of functional units
  - ♦ May be able to keep more of the hardware busy (thus improving throughput)
- Each SMT thread takes *more time* that it would if it was the only thread
- Almost entirely managed by hardware

Why Use Threads?

- Manage multiple points of interaction
  - Low overhead steering/probing
  - Background checkpoint save
- Alternate method for nonblocking operations
  - CORBA method invocation (no funky nonblocking calls)
- Hiding memory latency
- Fine-grain parallelism
  - Compiler parallelism

Latency Hiding

Common Thread Programming Models

- Library-based (invoke a routine in a separate thread)
  - pthreads (POSIX threads)
  - See "Threads cannot be implemented as a library," H. Boehm http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf
- Separate enhancements to existing languages
  - OpenMP, OpenACC, OpenCL, CUDA, …
- Within the language itself
  - Java, C11, others

Thread Issues

- Synchronization
  - Avoiding conflicting operations (memory references) between threads
- Variable Name Space
  - Interaction between threads and the language
- Scheduling
  - Will the OS do what you want?

Synchronization of Access

- Read/write model

```
a = 1;                 b = 1;
barrier();             barrier();
b = 2;                 while (a==1) ;
a = 2;                 printf( "%d\n", b );
```

What does thread 2 print?

- Take a few minutes and think about the possibilities

Synchronization of Access

- Read/write model

```
a = 1;
barrier();
b = 2;
a = 2;
```

```
b = 1;
barrier();
while (a==1) ;
printf( "%d\n", b );
```

  What does thread 2 print?

- Many possibilities:
  - ♦ 2 (what the programmer expected)
  - ♦ 1 (thread 1 reorders stores so a=2 executed before b=2 (valid in language)
  - ♦ Nothing: a never changes in thread 2
  - ♦ Some other value from thread 1 (value of b before this code starts)

How Can We Fix This?

- Need to impose an order on the memory updates
  - OpenMP has FLUSH (more than required)
  - Memory barriers (more on this later)
- Need to ensure that data updated by another thread is reloaded
  - Copies of memory in cache may update *eventually*
  - In this example, a may be (is likely to be) in register, *never updated*
  - volatile in C, Fortran indicate value might be changed outside of program

Synchronization of Access

- Often need to ensure that updates happen *atomically* (all or nothing)
  - Critical sections, lock/unlock, and similar methods
- Java has "synchronized" methods (procedures)
- C11 provides atomic memory operations

Variable Names

- Each thread can access all of a processes memory (except for the thread's stack*)
  - Named variables refer to the address space—thus visible to all threads
  - Compiler doesn't distinguish `A` in one thread from `A` in another
  - No modularity
  - Like using Fortran blank COMMON for all variables
- "Thread private" extensions are becoming common
  - "Thread local storage" (tls) is becoming common as an attribute
  - NEC has a variant where all variables names refer to different variables unless specified
    - All variables are on thread stack by default (even globals)
    - More modular

Scheduling Threads

- If threads used for latency hiding
  - ♦ Schedule on the same core
    - Provides better data locality, cache usage
- If threads used for parallel execution
  - ♦ Schedule on different cores using different memory pathways
  - ♦ Appropriate for data parallelism
  - ♦ Appropriate for certain types of task parallelism

The Changing Computing Model

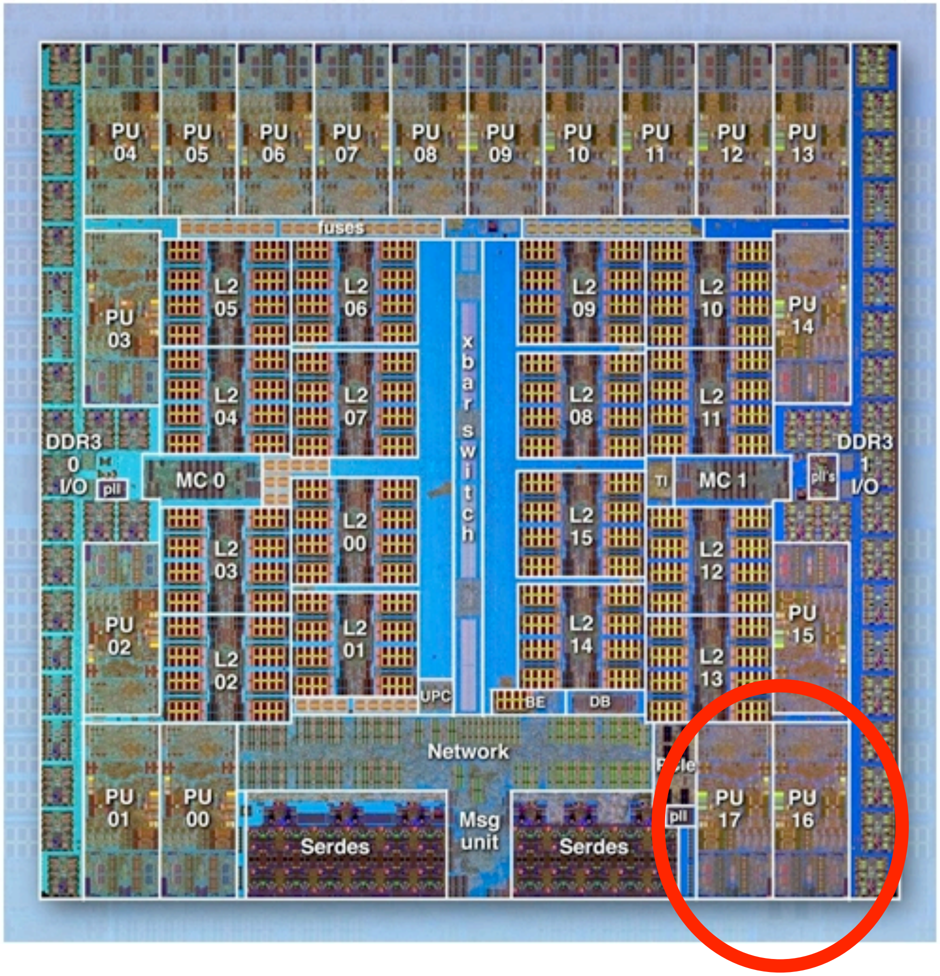
- More interaction
  - ♦ Threads allow low-overhead agents on any computation
    - OS schedules if necessary; no overhead if nothing happens (almost…)
  - ♦ Changes the interaction model from batch (give commands, wait for results) to constant interaction
- Fine-grain parallelism
  - ♦ Simpler programming model
- Lowering the Memory Wall
  - ♦ CPU speeds increasing much faster than memory
  - ♦ Hardware threads can hide memory latency

Node Execution Models

- Where do threads run on a node?
  - Typical user expectation: User's applications uses all cores and has complete access to them
- Reality is complex. Common cases include:
  - OS pre-empts core 0; Or cores 0,2
  - OS pre-empts user threads, distributes across cores
  - Hidden core (BG/Q)

Blue Gene/Q Processor

- 1 spare core for yield
- 1 core reserved for system (OS, services)



PARALLEL@ILLINOIS

Performance Models

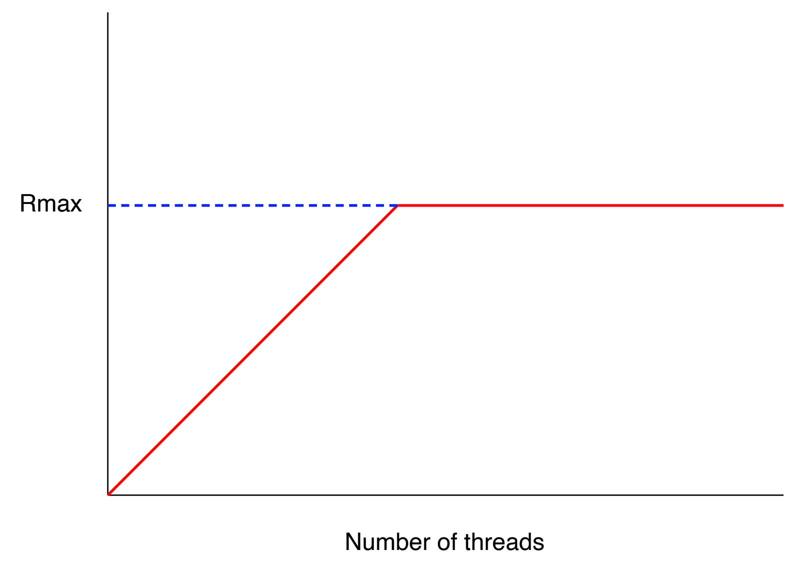
- Easiest: Everything independent
  - Usually appropriate for L1 cache
  - L2 may be shared, L3 almost certainly shared
  - Two limits on performance: Maximum performance per thread and maximum overall (aggregate).

Performance Models: Memory

- Assume the time to move a unit of memory is t_m
  - Due to latency in hardware; clock rate of data paths
  - Rate is $1/t_m = r_m$
- Also assume that there is a maximum rate r_{max}
  - E.g., width of data path * clock rate
- Then the rate at which k threads can move data is
  - $\min(k/t_m, r_{max}) = \min(kr_m, r_{max})$

Limits on Thread Performance

- Threads share memory resources
- Performance is roughly linear with additional threads *until* the maximum bandwidth is reached
- At that point each thread receives a decreasing fraction of available bandwidth



PARALLEL@ILLINOIS

Questions

- How do you expect a multithreaded STREAM to perform as you add threads? Sketch a graph.
- What's the difference between a software thread and a hardware thread?
- What happens if there are more threads that cores? Can programs run faster in that case?