



COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Lecture 10

Computer Architecture and Basic Performance

Dr. Brendan Kochunas
10/07/2018

NERS 590-004



Outline

- Motivating example:
 - Lab 4 Matrix-matrix Multiply
- Basic notions of serial performance
- Computer Architecture
 - How do we understand it?
 - Memory Hierarchy
 - How to characterize the “micro-architecture”
 - Things that happen in parallel and parallel architectures



Learning Objectives

- Insights into Lab 4
- Understand the definition of performance.
- Understand what parts of computer hardware affect performance.
- Understand detailed practical models for architecture of processors and supercomputers



Motivation



Matrix-matrix Multiply

- Lab 4
 - Write your own matrix-matrix multiply
 - Compare to `dgemm` from BLAS
 - For system BLAS library and OpenBLAS
 - Compare to NumPy

$$\mathbf{AB} = \mathbf{C} \rightarrow c_{i,j} = \sum_k a_{i,k} b_{k,j}$$



Results for Lab 4

Matrix Size	My Implementations			System BLAS	OpenBLAS
100	0.001	0.001	0.003	0.001	0.064
500	0.062	0.068	0.465	0.071	0.069
1000	0.505	0.559	4.109	0.575	0.291
2000	5.516	6.160	39.028	5.787	0.749
4000	47.922	50.341	>100	46.229	2.822

Results obtained on Great Lakes

Clearly something is going on here...



Matrix-matrix Multiply

- Lab 4
 - Write your own matrix-matrix multiply
 - Compare to dgemm from BLAS
 - For system BLAS library and OpenBLAS
 - Compare to NumPy

$$\mathbf{AB} = \mathbf{C} \rightarrow c_{i,j} = \sum_k a_{i,k} b_{k,j}$$

```
do i=1,n
  do j=1,n
    do k=1,n
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

```
do k=1,n
  do j=1,n
    do i=1,n
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

```
c=MATMUL(A,B)
```

Fortran one-liner



NumPy uses MKL!

```
$ python
>>> import numpy
>>> numpy.__config__.show()
lapack_opt_info:
  libraries = ['mkl_intel_lp64', 'mkl_intel_thread', 'mkl_core', 'iomp5', 'pthread']
  library_dirs = ['/sw/lsa/centos7/python-anaconda2/created-20170424/lib']
  define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
  include_dirs = ['/sw/lsa/centos7/python-anaconda2/created-20170424/include']
blas_opt_info:
  libraries = ['mkl_intel_lp64', 'mkl_intel_thread', 'mkl_core', 'iomp5', 'pthread']
  library_dirs = ['/sw/lsa/centos7/python-anaconda2/created-20170424/lib']
  define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
  include_dirs = ['/sw/lsa/centos7/python-anaconda2/created-20170424/include']
openblas_lapack_info:
  NOT AVAILABLE
lapack_mkl_info:
  libraries = ['mkl_intel_lp64', 'mkl_intel_thread', 'mkl_core', 'iomp5', 'pthread']
  library_dirs = ['/sw/lsa/centos7/python-anaconda2/created-20170424/lib']
  define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
  include_dirs = ['/sw/lsa/centos7/python-anaconda2/created-20170424/include']
blas_mkl_info:
  libraries = ['mkl_intel_lp64', 'mkl_intel_thread', 'mkl_core', 'iomp5', 'pthread']
  library_dirs = ['/sw/lsa/centos7/python-anaconda2/created-20170424/lib']
  define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
  include_dirs = ['/sw/lsa/centos7/python-anaconda2/created-20170424/include']
```

- Turns out...
 - NumPy uses Intel MKL!
- Anaconda (and probably Canopy) distributions of NumPy also include Intel MKL libraries!
- MKL has a free community edition library.



Performance Basics

To understand the performance you observe,
you first have to understand what performance to expect

To have an expectation of performance,
we all have to be on the same page about what performance *is*



What is performance?

What does it mean for a code to be fast?

- The real metric: Time
- Derived metrics
 - *FLOPS* = Floating Point Operations per Second
 - *Bandwidth* = data per unit time (sort of like a flow rate)
 - *Latency* = Minimum time for data to travel from point A to point B
- Theoretical Peak Performance
 - Very difficult to achieve in practice
 - Can be computed from hardware specs
- Do things efficiently in time

How do you get fast code?

- First: Choose the right algorithm
- Second: Understand how to express that algorithm in the programming language
- Third: Understand how the source code will get mapped to the hardware
- Fourth: Tune the code to the hardware
- A lot of this can be done with pen and paper



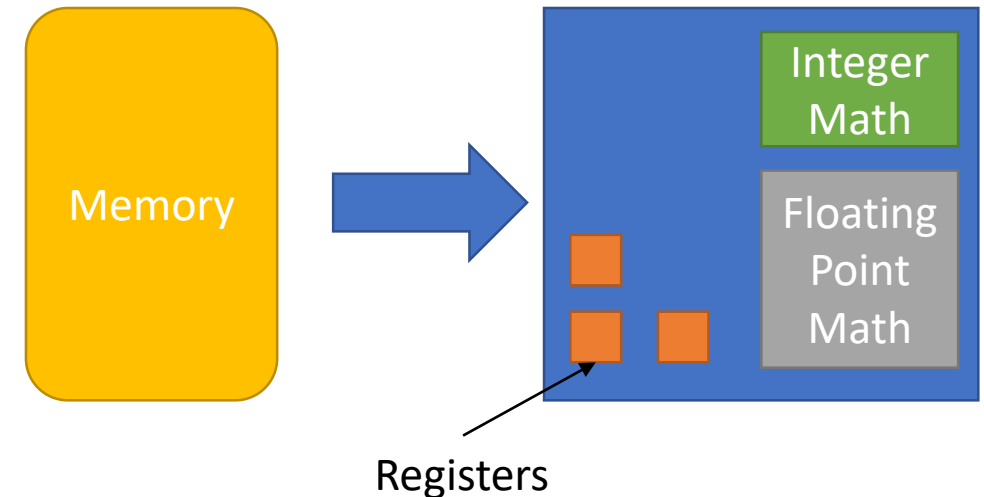
COLLEGE OF ENGINEERING
UNIVERSITY OF MICHIGAN


NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES

Computer Architecture

Idealized Processor Model

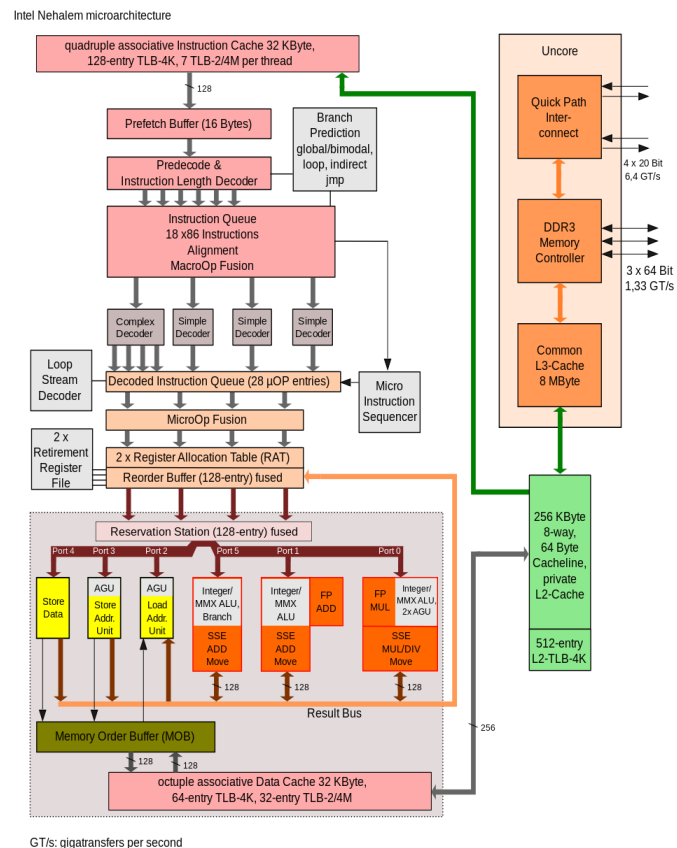
- Processor names bytes, words, etc. in its address space
 - These represent integers, floats, pointers, arrays, etc.
- Operations include
 - Read and write into very fast memory called registers
 - Arithmetic and other logical operations on register
- Order specified by program
 - Read and returns the most recently written data
 - Compiler and architecture translate high level expressions into “obvious” Lower level instructions
 - Hardware executes instructions in order specified by compiler
- Idealized Cost
 - Each operation has roughly the same cost (read, write, add, multiply, etc.)



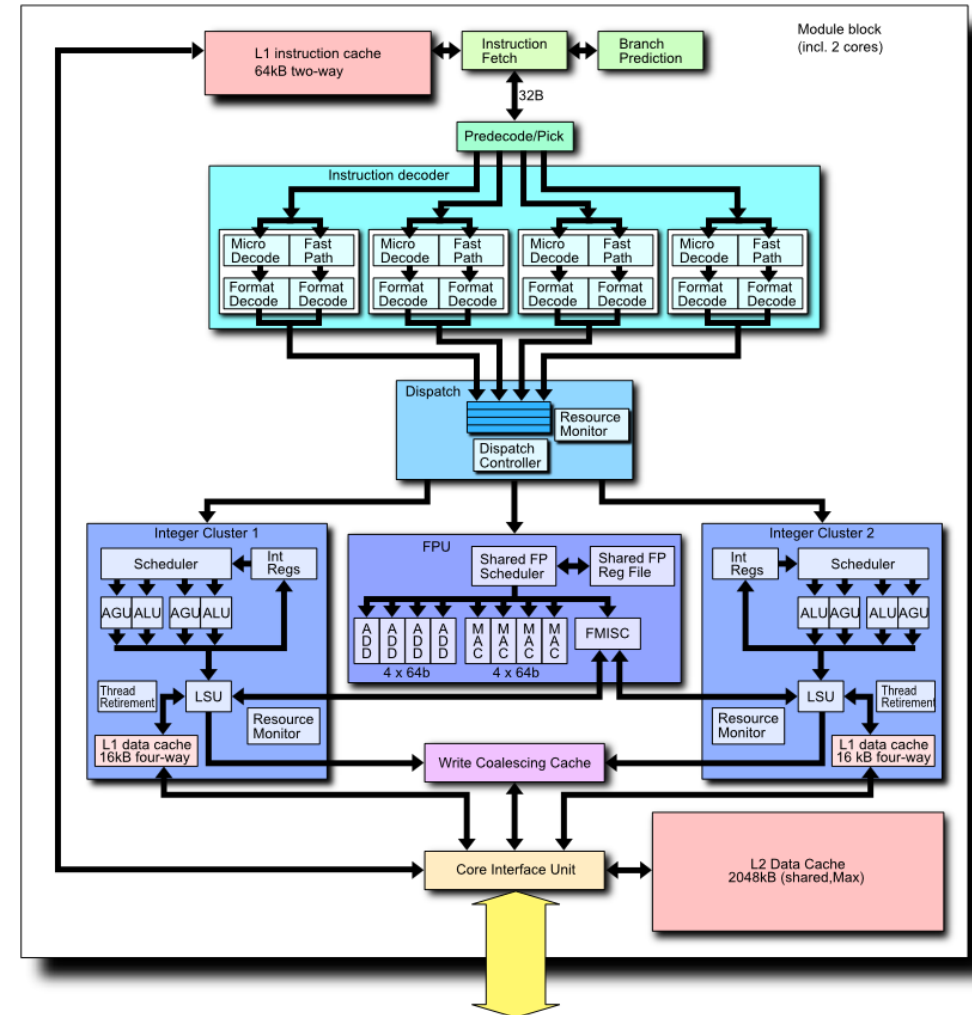
$A+B=C$ 

Read address(A) into R1
 Read address(B) into R2
 $R3 = R1 + R2$
 Write R3 to address(C)

Real World Processors



Intel “Nehalem”

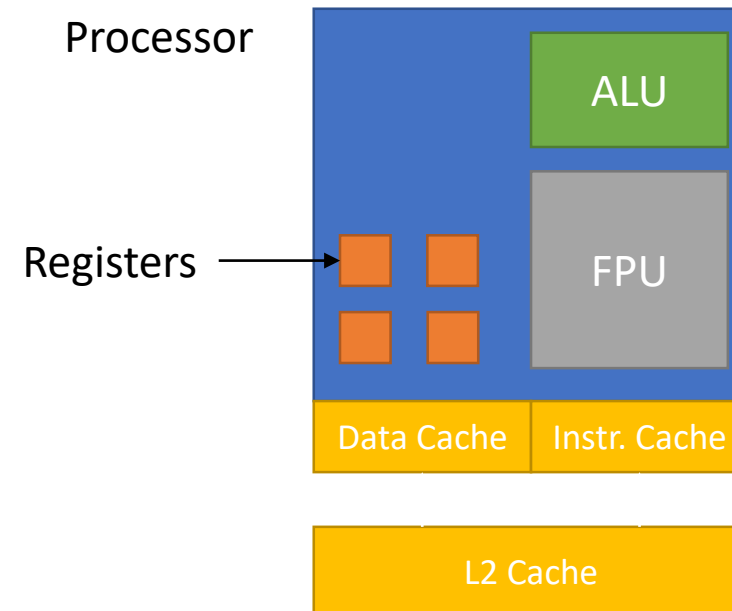


AMD “Bulldozer”

“Single” Processor Concept

- Real world processors have
 - *Registers and caches*
 - Small amounts of fast memory
 - Stores values of recently used data nearby
 - Different memory operations can have very different costs
 - *Parallelism*
 - Multiple “functional units” that can run in parallel
 - *Pipelining*
 - A form of parallelism like assembly line
- Why is this your problem?
 - In theory, compilers and hardware “understand” all this complexity
 - and can optimize our programs; in practice they do not often do this well
 - Compilers do not know about different algorithms that might be better

- We want to know the details to use processors effectively
 - Don’t want to know all the details
 - Don’t want to have an incomplete model.





Cache Basics

- Cache is fast (expensive) memory which keeps copy of data in main memory;
 - Typically it is hidden from software (e.g. no standard way of programming directly)
 - Simplest example: data at memory address xxxxx1101 is stored at cache location 1101
- Cache hit: in-cache memory access—cheap
- Cache miss: non-cached memory access—expensive
 - Need to access next, slower level of cache
- Cache line length: # of bytes loaded together in one entry
 - Ex: If either xxxxx1100 or xxxxx1101 is loaded, both are
- Associativity
 - direct-mapped: only 1 address (line) in a given range in cache
 - Data stored at address xxxxx1101 stored at cache location 1101, in 16 word cache
 - n -way: $n \geq 2$ lines with different addresses can be stored
 - Up to $n \leq 16$ words with addresses xxxxx1101 can be stored at cache location 1101 (so cache can store $16n$ words)

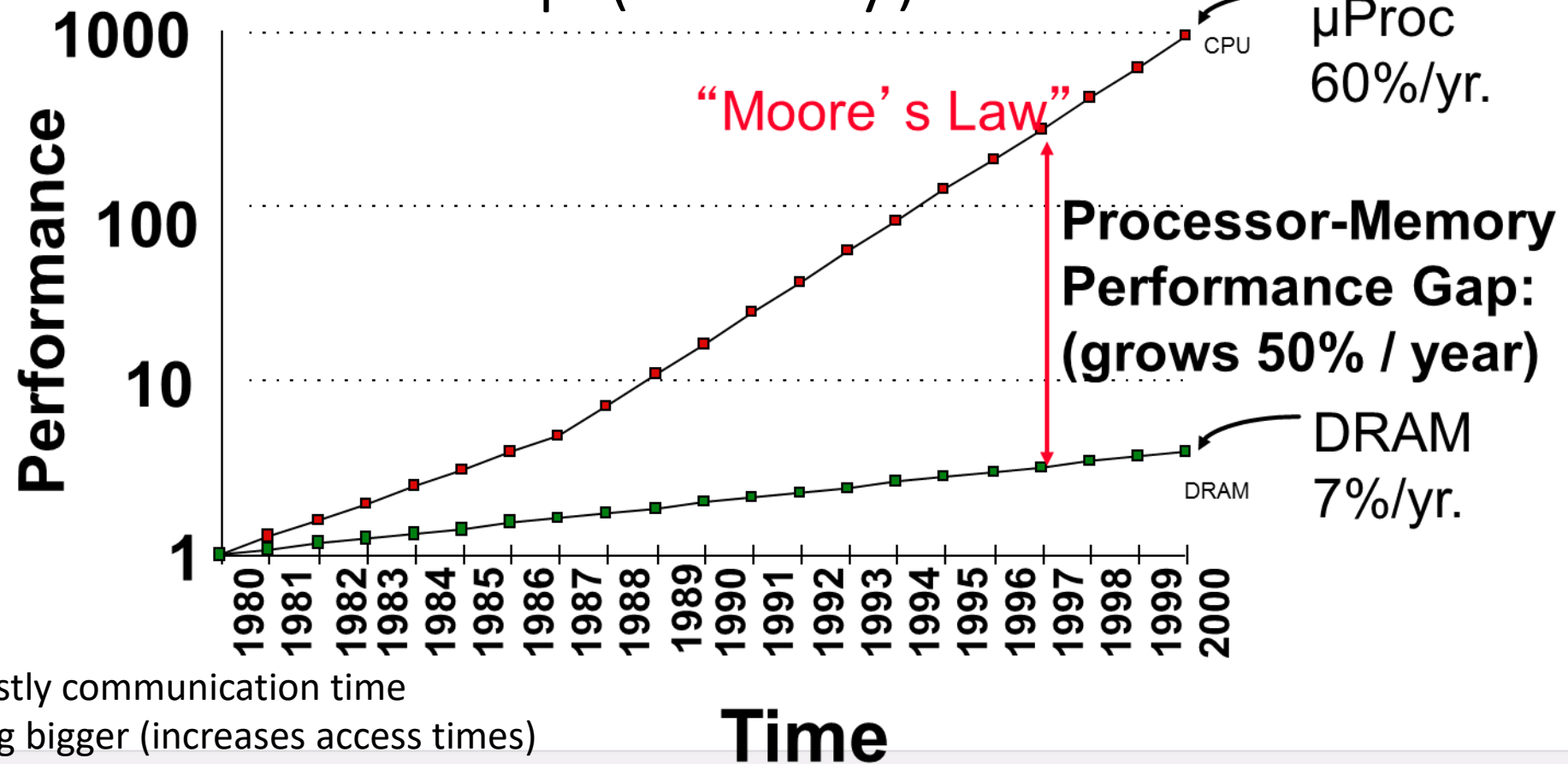


Why have multiple levels of cache/memory?

- Most programs have a high degree of locality in their memory access patterns
 - Spatial locality: accessing data nearby previously accessed data
 - Temporal Locality: access data and reuse that data a lot
 - A memory hierarchy attempts to exploit locality to improve overall average access time.
- Cache is small and fast (speed = \$\$\$)
 - A large cache always has delays: time to check addresses is longer
 - There are other parts to memory hierarchy (TLB, pages, swap, etc.)
- Attempts to reconcile Processor/Memory Gap



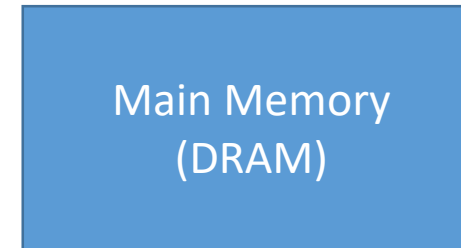
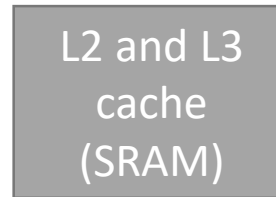
Processor-DRAM Gap (latency)



Main delay is mostly communication time
Memory is getting bigger (increases access times)

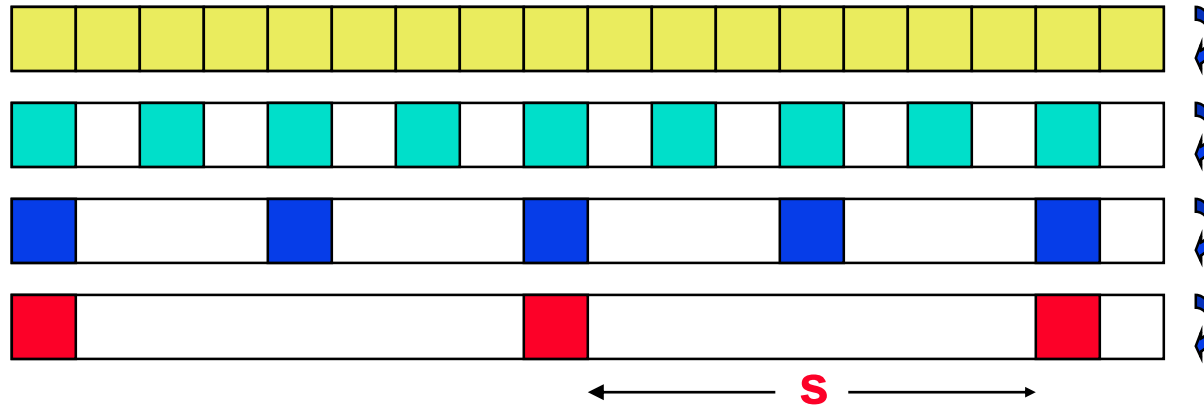


Memory Hierarchy



	Register	L1	L2	L3	DRAM	Disk	Tape
Size	< 1 KB	~1KB	1 MB	10's MB	1-100's GB	TB	PB
Speed	< 1ns	<1 ns	~1 ns	~1-10 ns	10-100 ns	10 ms	~10s

Techniques for exposing details of Memory

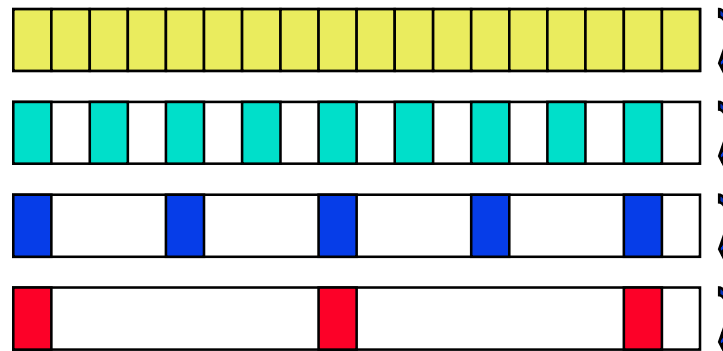


- for array A of length L from 4KB to 8MB by 2x
for stride s from 4 Bytes (1 word) to L/2 by 2x
time the following loop
(repeat many times and average)
for i from 0 to L **by s**
load A[i] from memory (4 Bytes)

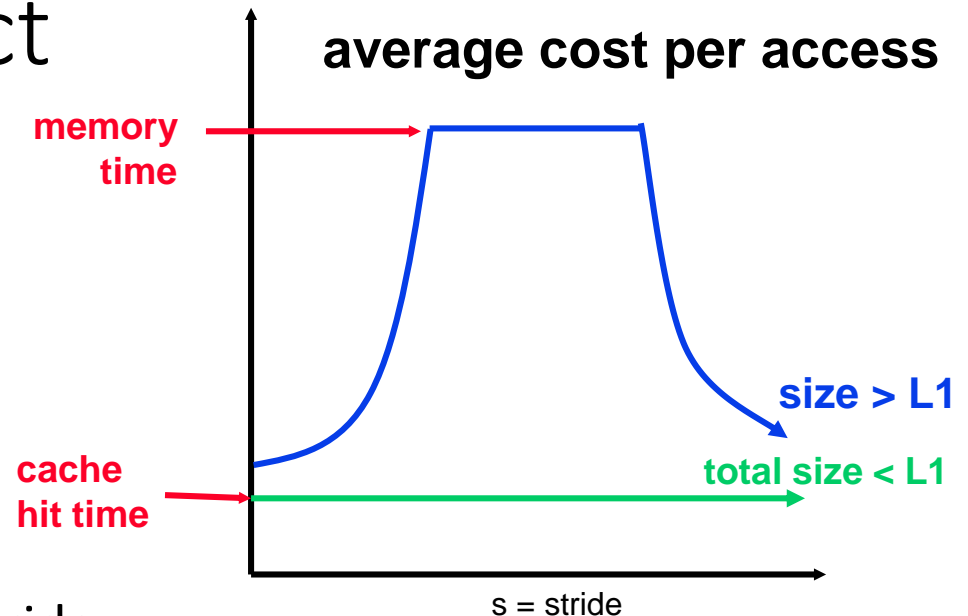
- Determine memory access times experimentally using “micro”-benchmarks
 - Membench (Saavedra-Barrera), STREAM, others...

1 experiment

Membench: What to Expect

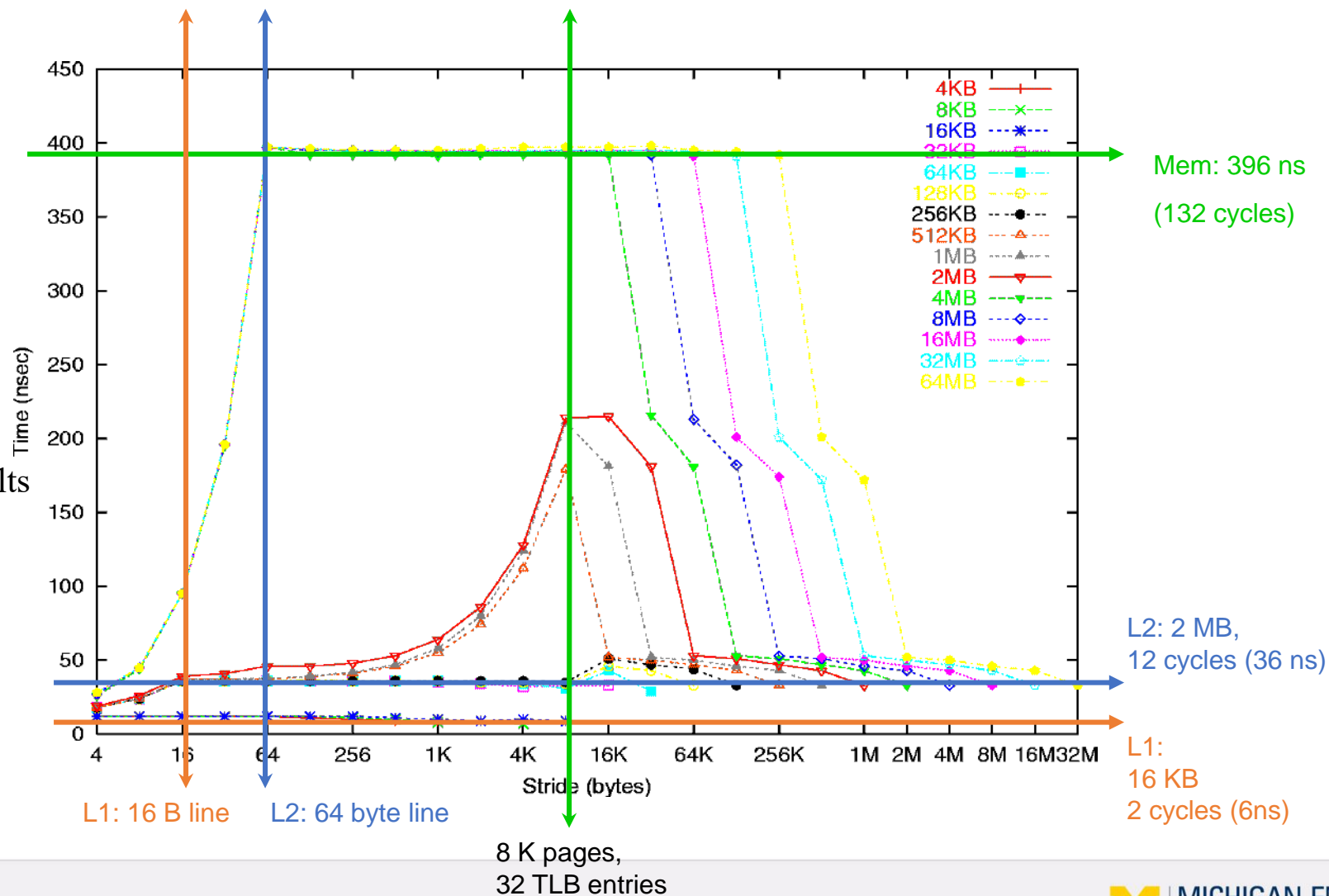


- Consider the average cost per load
 - Plot one line for each array length: time vs. stride
 - If array is smaller than a given cache all accesses will hit (after first run)
 - Small stride is best: e.g. if cache line holds 4 words, expect $\frac{1}{4}$ miss
 - Picture assumes one-level cache
 - More difficult to measure on modern processors due to more complex memory systems





Example Membench Results
for
Sun Ultra-2i, 333 MHz



See www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps for details



STREAM Benchmark

- STREAM: Sustainable Memory Bandwidth in High Performance Computers
 - Measures sustainable memory bandwidth (in MB/s) based on simple vector kernels
 - Developed & maintained by Dr. John D. McCalpin at University of Virginia

name	kernel	bytes/iter	FLOPS/iter
COPY:	$a(i) = b(i)$	16	0
SCALE:	$a(i) = q * b(i)$	16	1
SUM:	$a(i) = b(i) + c(i)$	24	1
TRIAD:	$a(i) = b(i) + q * c(i)$	24	2

- Developing “second-generation” STREAM benchmark (STREAM2)
 - Emphasis is on measuring
 - Bandwidth of memory hierarchy, latency, other access patterns, locality (e.g. bandwidth and latency between distributed shared memory systems)
 - <https://www.cs.virginia.edu/stream/ref.html>



Single Core Parallelism

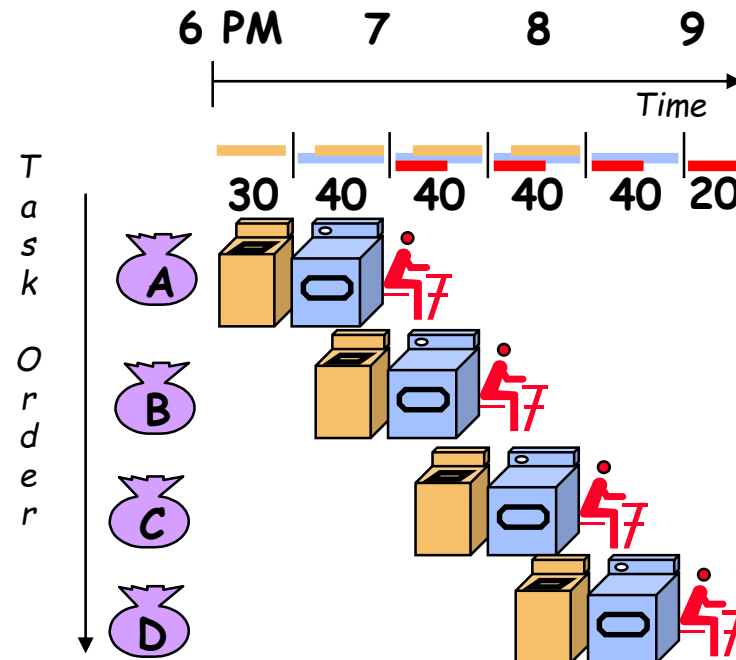
- Pipelining
 - Like an assembly line, have different parts of processor doing things in parallel
 - Just know that this happens, you don't have much ability to control this, other than perhaps some compiler options
- Vector Instructions (SIMD)
 - SIMD = Single Instruction Multiple Data
 - As a programmer, you have some control over this...
 - Use of compiler options
 - Also have to write your code in a way that lets compiler recognize SIMD
 - Easily expressible in Fortran with intrinsic array operators and functions

Pipelining

Dave Patterson's Laundry example: 4 people doing laundry

wash (30 min) + dry (40 min) + fold (20 min) = 90 min

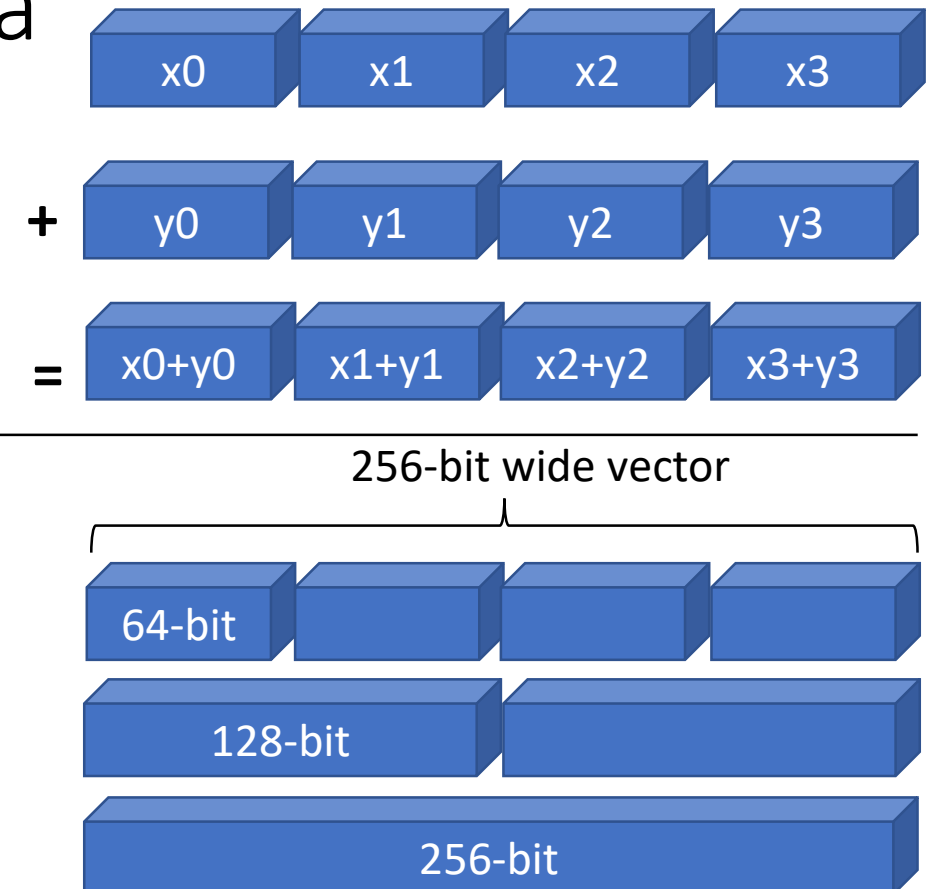
Latency



- In this example:
 - Sequential execution takes $4 * 90\text{min} = 6 \text{ hours}$
 - Pipelined execution takes $30 + 4 * 40 + 20 = 3.5 \text{ hours}$
- **Bandwidth** = loads/hour
- $BW = 4/6 \text{ l/h w/o pipelining}$
- $BW = 4/3.5 \text{ l/h w pipelining}$
- $BW \leq 1.5 \text{ l/h w pipelining, more total loads}$
- Pipelining helps **bandwidth** but not **latency** (90 min)
- Bandwidth limited by **slowest** pipeline stage
- Potential speedup = **Number pipe stages**

Single Instruction Multiple Data

- One operation produces multiple results
- Implemented as SSE assembly instructions by compiler
 - SSE = Streaming SIMD Instructions
 - Several standards SSE (128-bit), SSE2, ... SSE4, AVX (256-bit), AVX2 (512-bit)
- Operate on anything that fits into x bytes (e.g. 16 bytes)
 - Operations include add, multiply, etc.
- Challenges
 - Need to be contiguous in memory *and aligned*
 - Some instructions are needed to move data around from one register to another



Cores, Processors, and Nodes OH MY!

- We have been narrowly focused on a “single core”.
 - Many processors are “multi-core”
 - Common for motherboards to have multiple “sockets” or processors
 - A node has one motherboard, with multiple processors, and each processor has multiple cores
- Other terminology
 - Symmetric Multi-processor (SMP)
 - Non-Uniform Memory Access (NUMA)

AMD Opteron™ 6274 (Interlagos) CPU

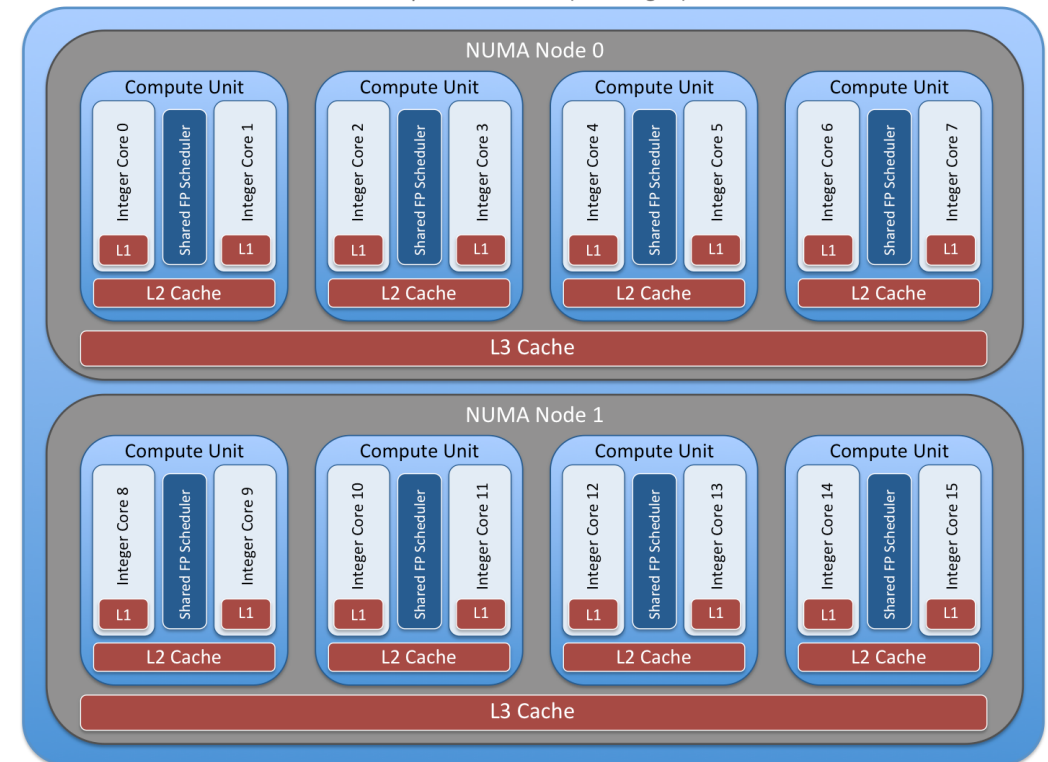
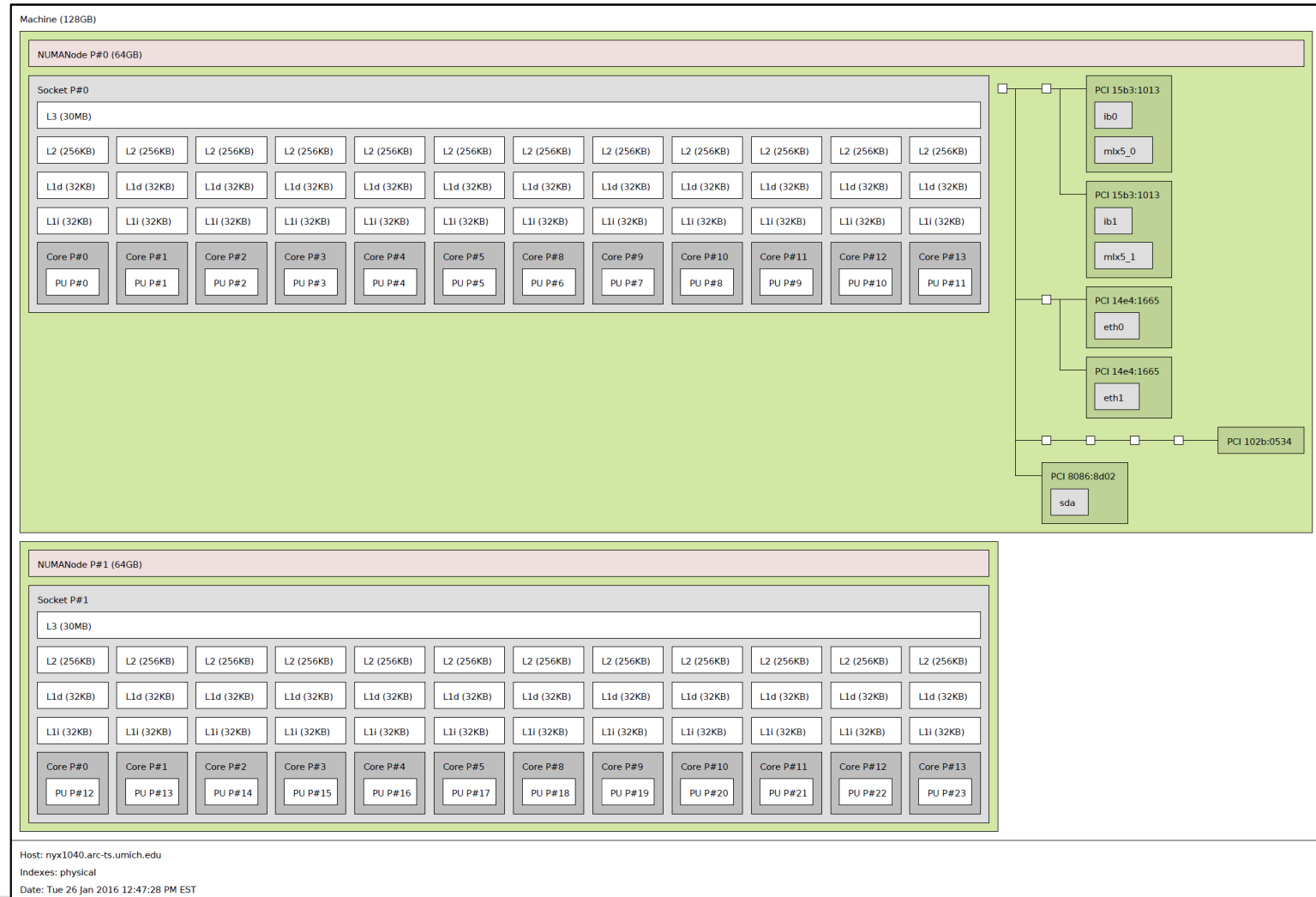


Illustration of Titan Compute Node [1]

[1] - <https://www.olcf.ornl.gov/support/system-user-guides/titan-user-guide/>

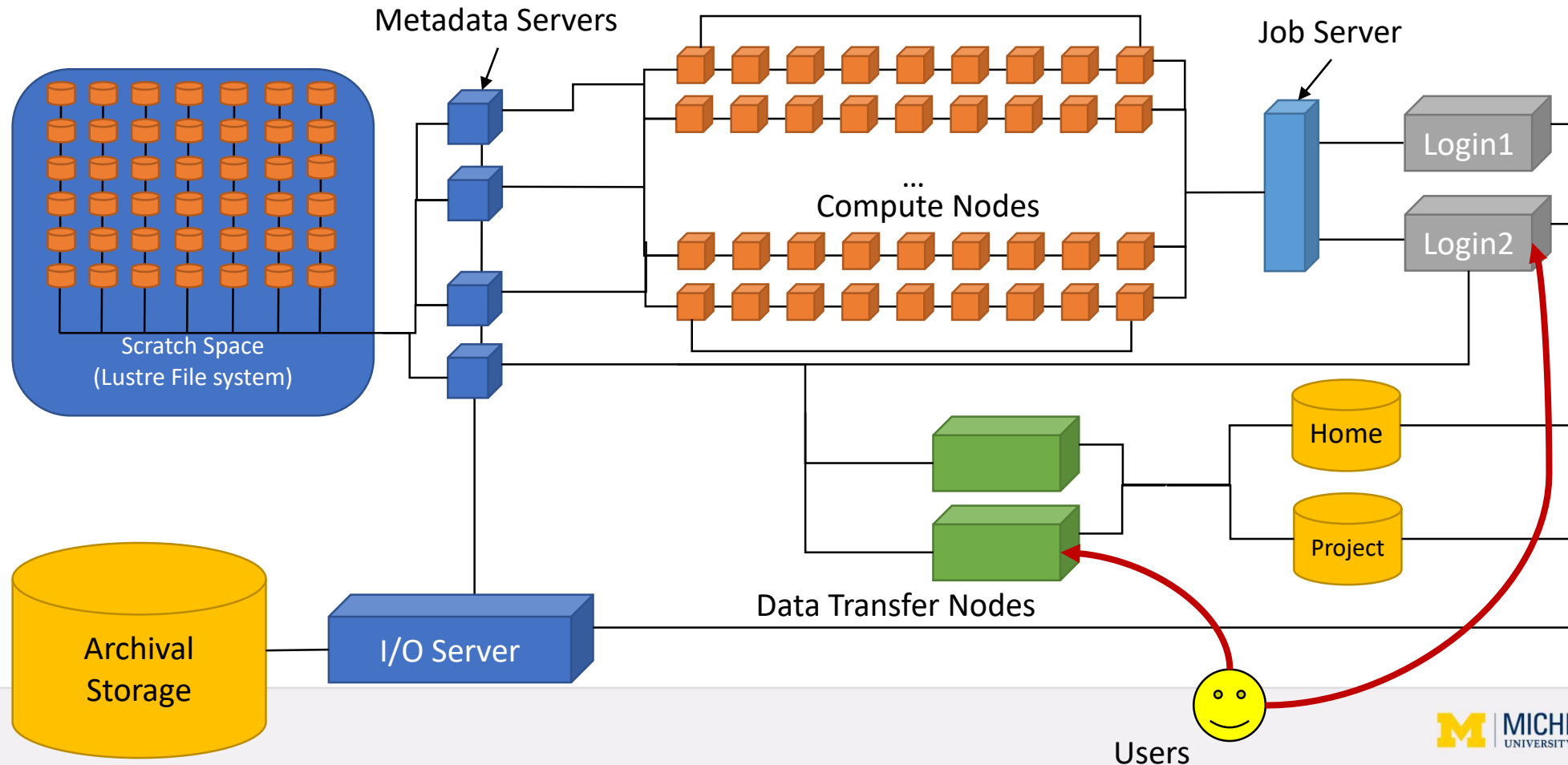


Flux node
(generated by `lstopo`)



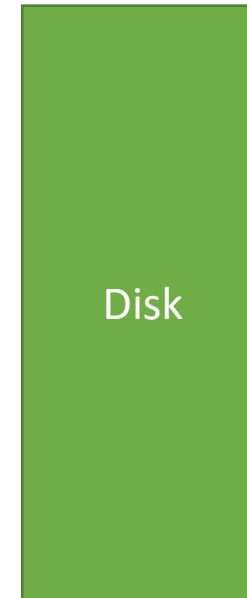
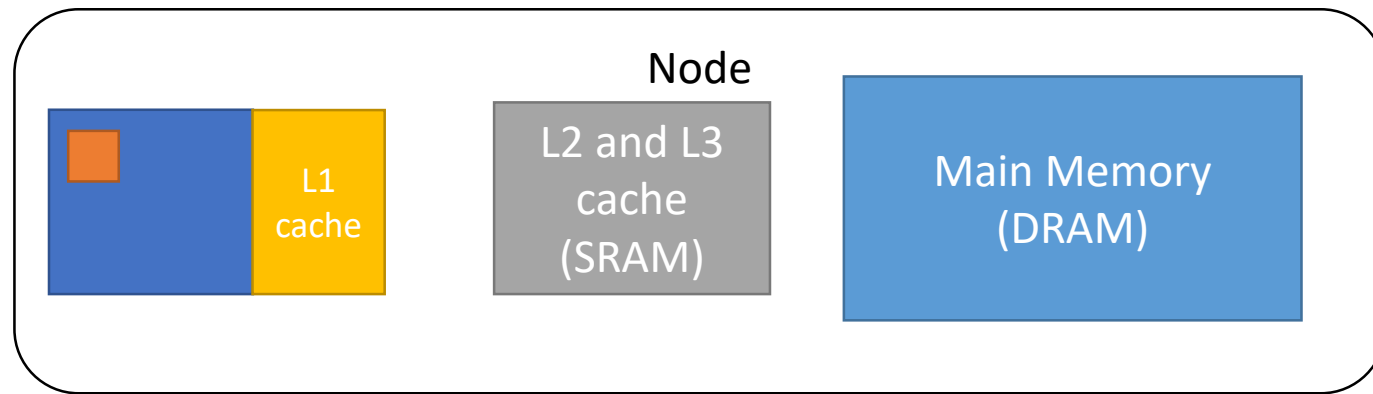


Contemporary HPC Platforms





Memory Hierarchy for Distributed Machines



	Register	L1	L2	L3	DRAM	Cluster	Disk	Tape
Size	< 1 KB	~1KB	1 MB	10's MB	1-100's GB	TB	TB	PB
Speed	< 1ns	<1 ns	~1 ns	1-10 ns	20-100ns	1-100 μ s	10 ms	~10s