Lab 08 – Mini-Lecture Microbenchmarks

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F20)

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Outline

- What are microbenchmarks?
- The Membench Experiment
  - Design
  - Analysis
- Results of benchmark from various machines
- Instructions
- 1-slide on SIMD

Learning Objectives: By the end of Today’s Lab you should be able to

- (Knowledge) understand how to profile and analyze a memory structure of a single node/computer.
- (Skill) generate results for membench on Great Lakes.
- (Knowledge) quickly look for and identify relevant benchmarks for measuring system performance parameters
- (Knowledge) describe what a mini-app is.

Microbenchmarks

- Purpose
  - Measure the performance of the full stack of
    - Source code
    - Hardware
    - OS
    - Compiler
    - Compiler options
  - validate performance models
- Scope
  - Quite small, effectively they are single procedures or short snippets of code.

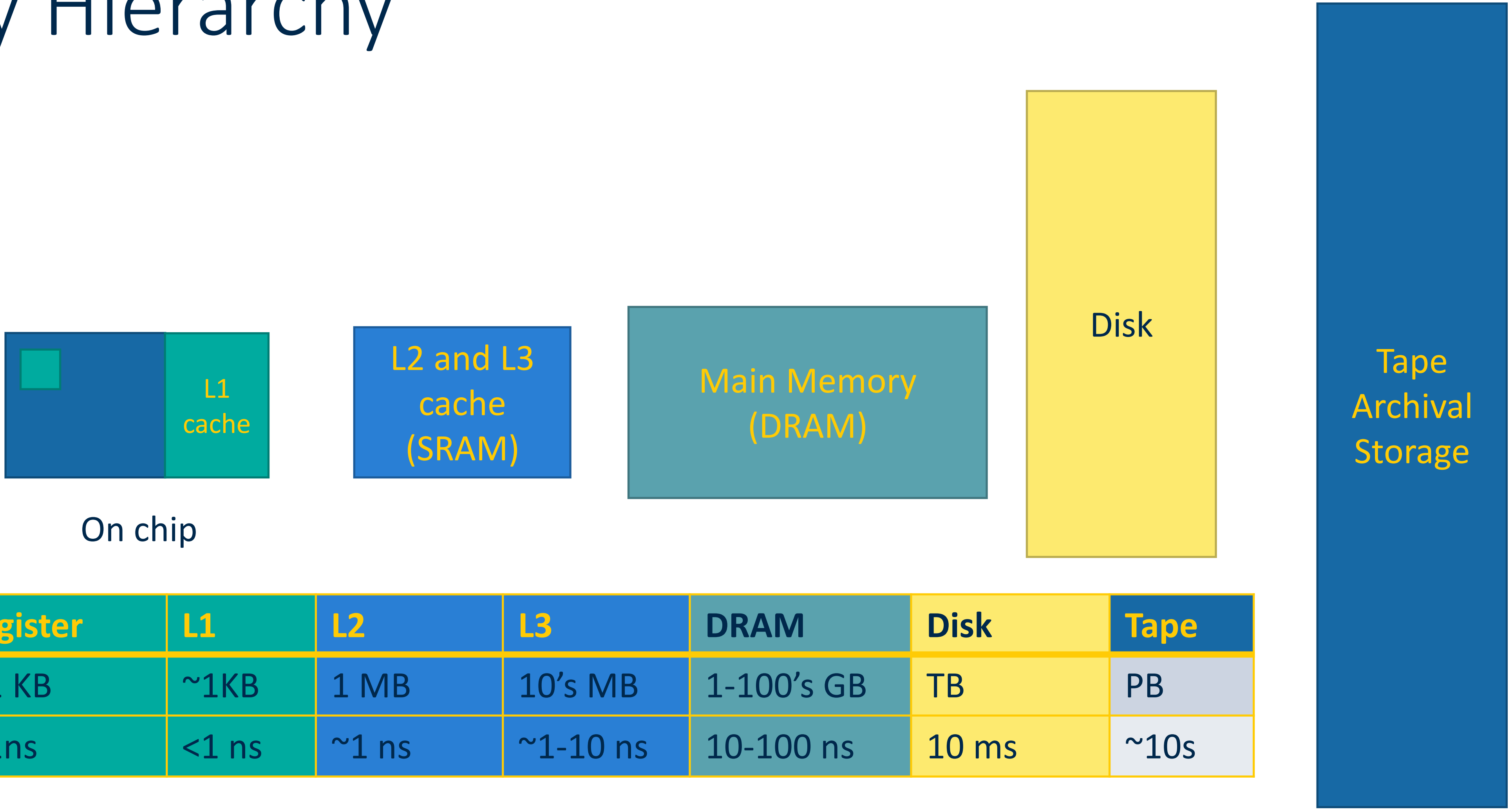
- Examples
  - Membench
  - STREAM
  - OSU MPI Micro-Benchmarks
  - OpenMP Benchmarks
  - PSNAP
  - IOR
  - MDTEST
  - PYNAMIC
  - SMB
  - UPC FT
  - MPIMEMU
- NERSC-8 Benchmark Suite

Extension to Mini-Apps

- Purpose
  - Evaluate the performance and scalability of a particular algorithm or numerical method
- Scope
  - Small programs that mimic the computational kernels of every step, but do not use “real data”

- Examples
  - Exascale Computing Project
    - AMG, Ember, ExaMiniMD, Laghos, MACSio, miniAMR, miniQMC, miniVite, **NEKbone**, **PICSARlite**, SW4lite, SWFFT, Thornado-mini, **XSBench**
    - miniGAN, miniRL, CRADL, Cosmoflow-Benchmark, Mlperf-DeepCAM
  - NERSC-8 Benchmarks
  - LANL Proxy Apps

Memory Hierarchy

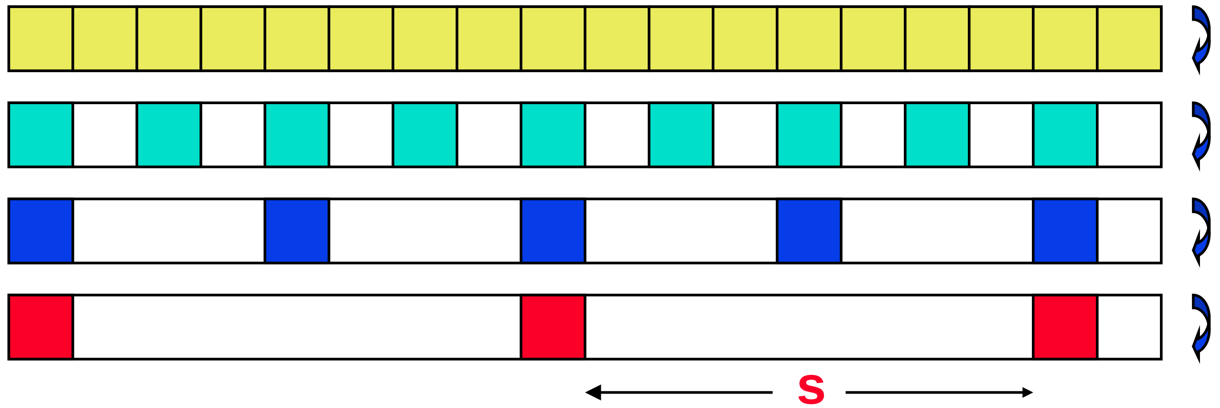


	Register	L1	L2	L3	DRAM	Disk	Tape
Size	< 1 KB	~1KB	1 MB	10's MB	1-100's GB	TB	PB
Speed	< 1ns	<1 ns	~1 ns	~1-10 ns	10-100 ns	10 ms	~10s

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)

Techniques for exposing details of Memory

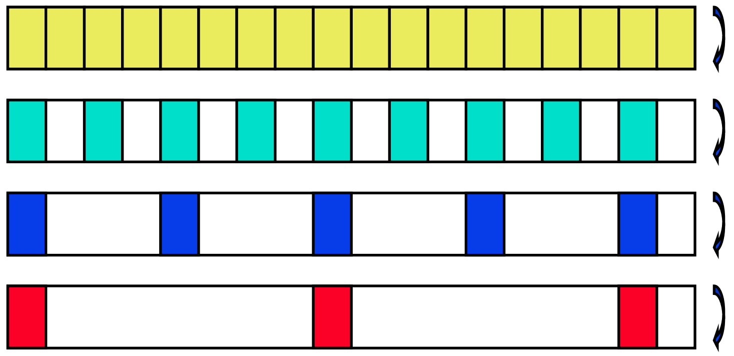


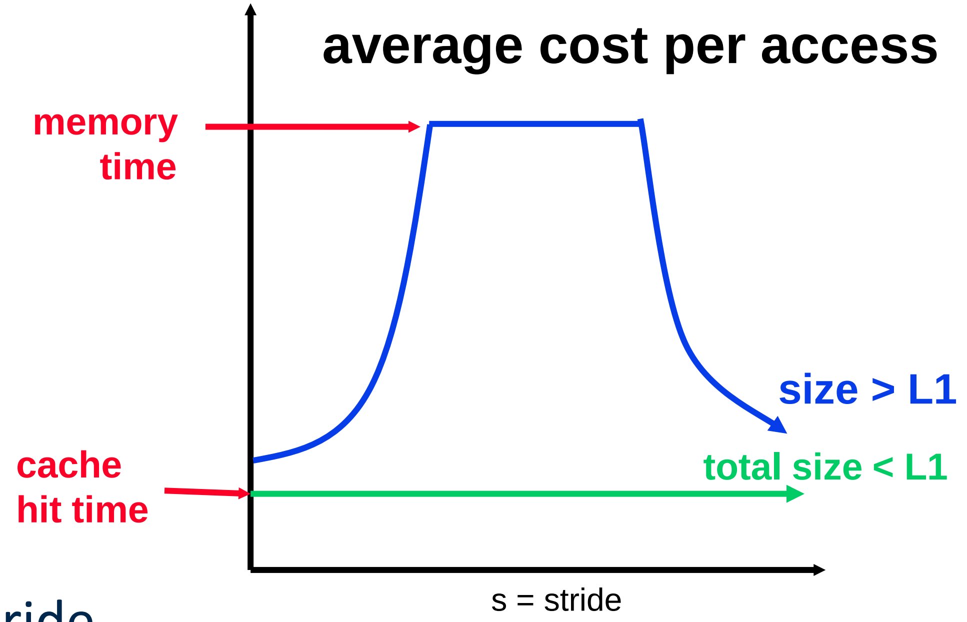
- Determine memory access times experimentally using “micro”-benchmarks
  - Membench (Saavedra-Barrera), STREAM, others…

```
• 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)
```

1 experiment

Membench: What to Expect



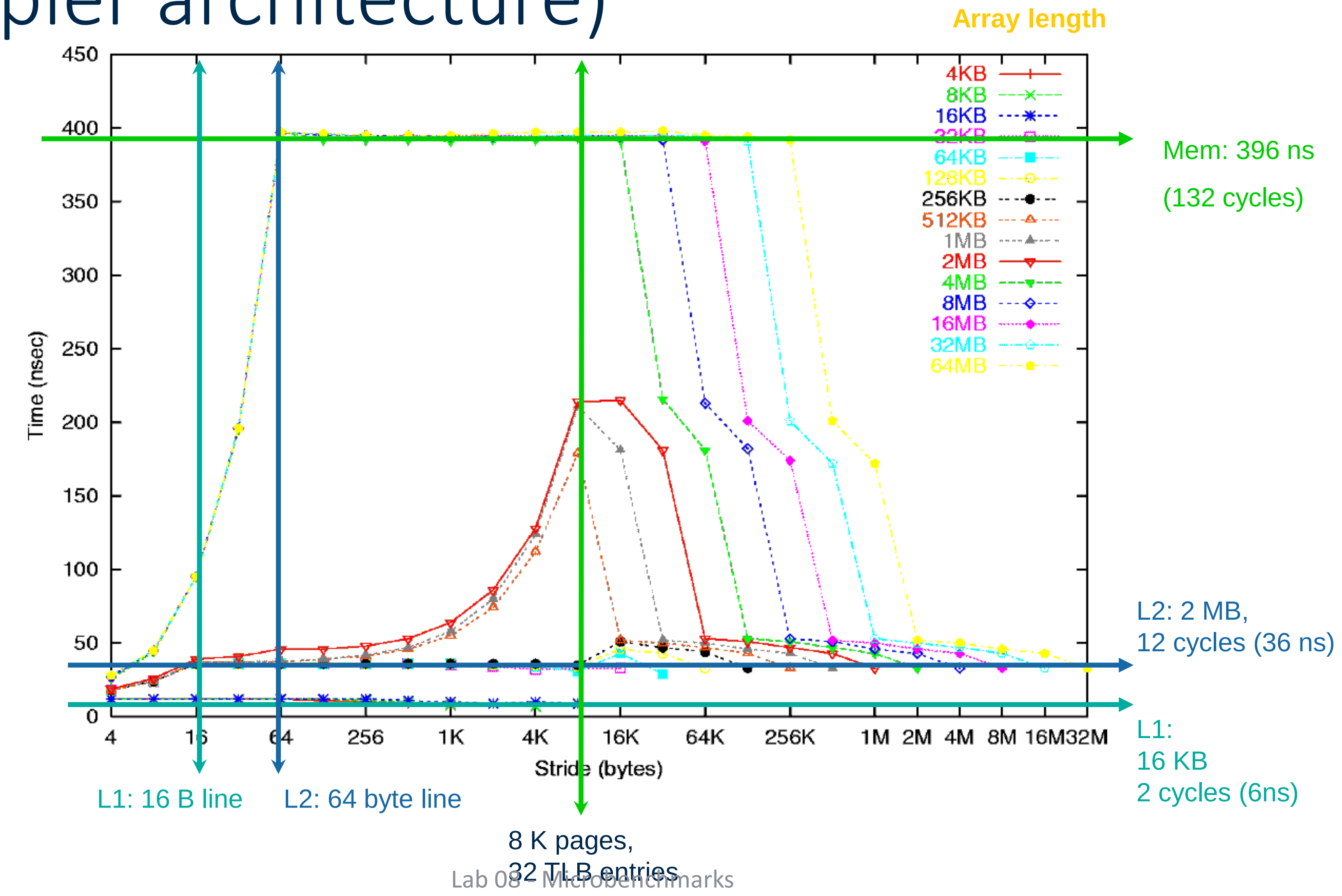


- Consider the average cost per load
  - Plot one line for each array length: time vs. stride
  - If array is smaller than a given cache all accesses will hit (after first run)
  - Small stride is best: e.g. if cache line holds 4 words, expect ¼ miss
  - Picture assumes one-level cache
  - More difficult to measure on modern processors due to more complex memory systems

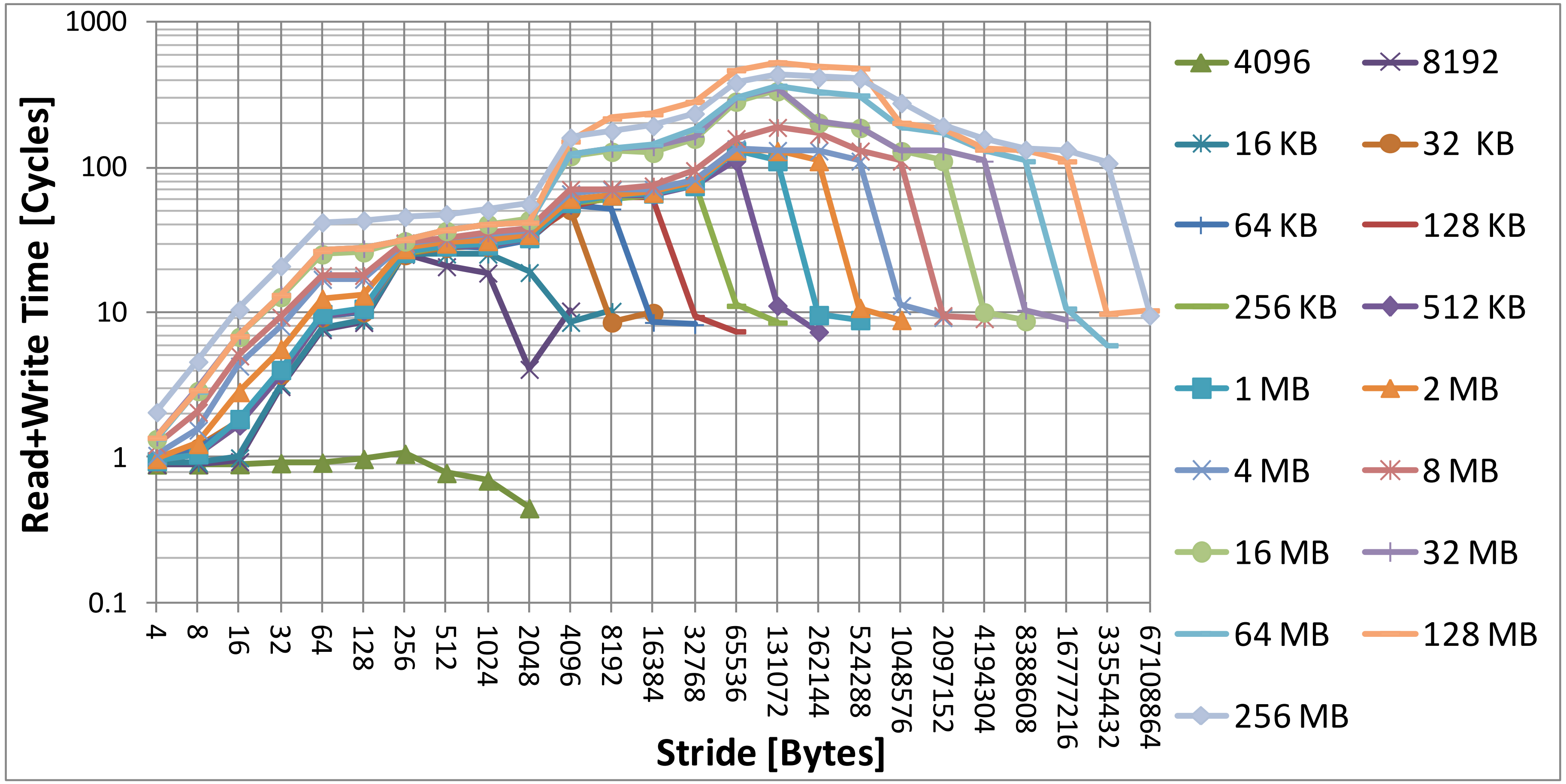
Results for a 25 year old computer (e.g. simpler architecture)

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

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

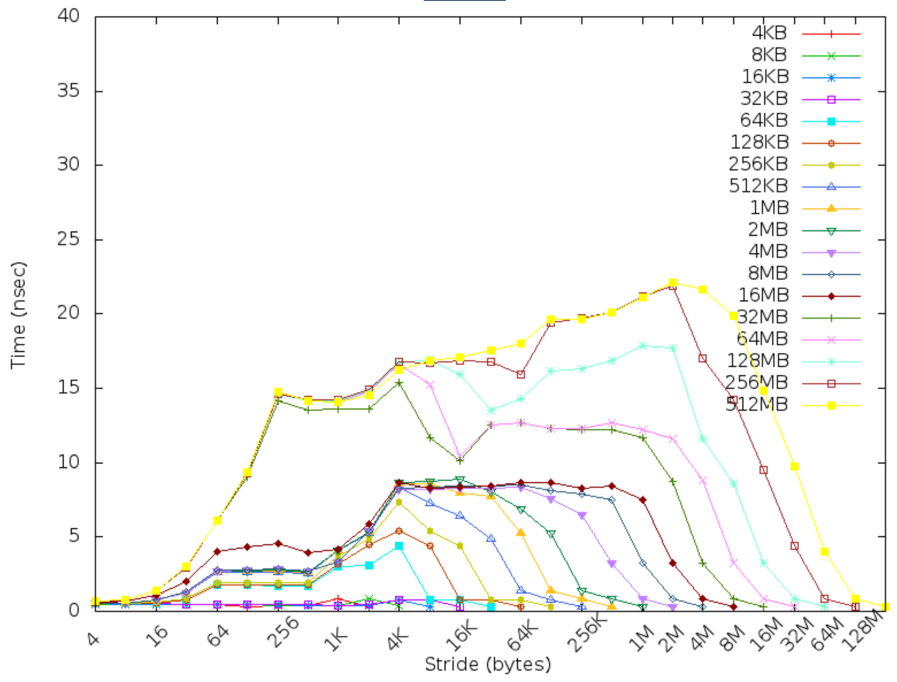


Results from 10-year old Workstation

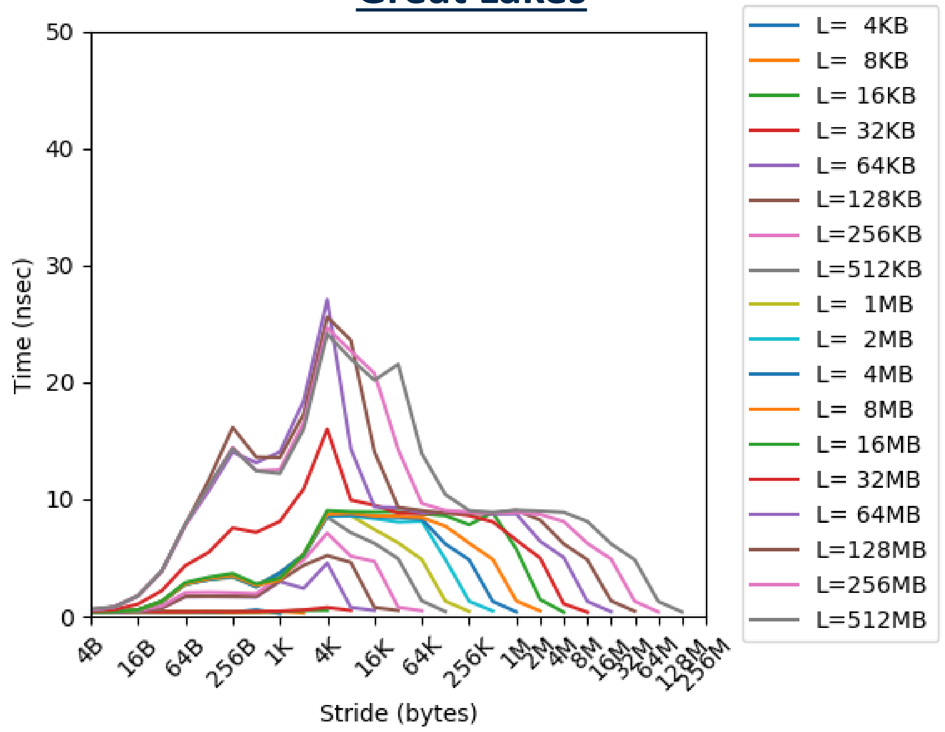


More Recent Results

Flux



Great Lakes



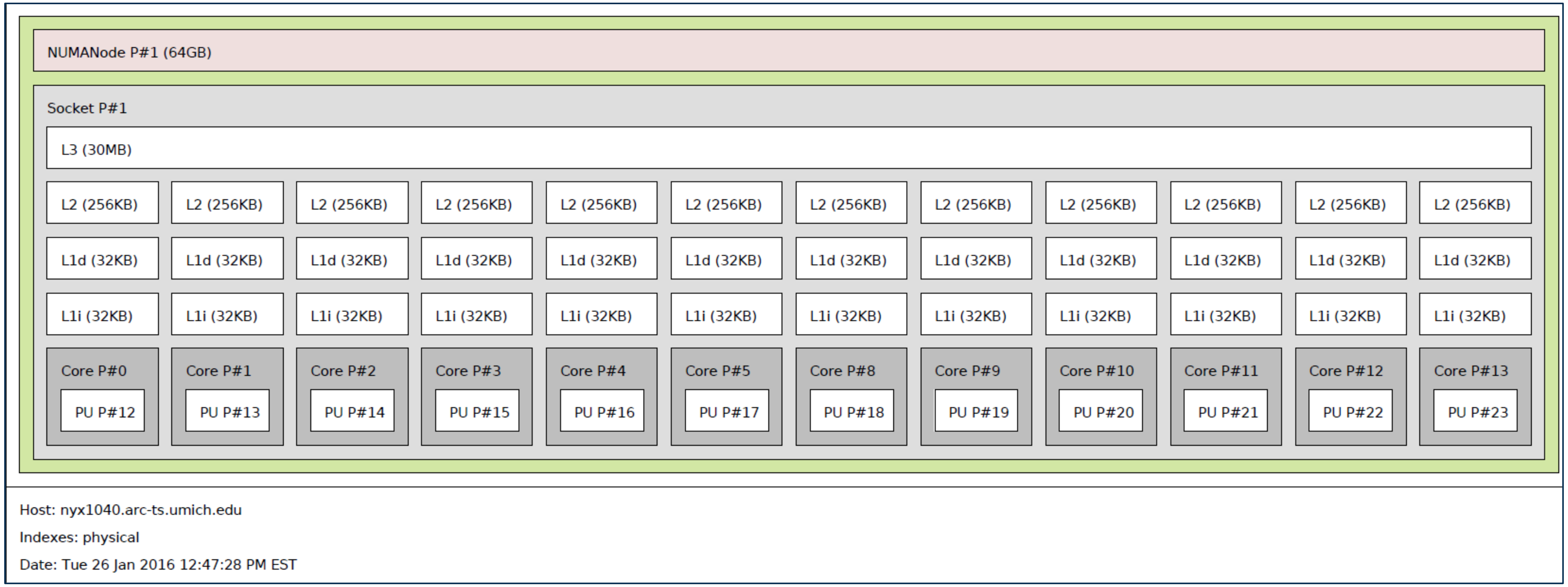
For the experiment you are expected to:

- Modify the program to consider arrays of up to 1GB.
- Modify the program to fix any output formatting
- Modify the provided data processing scripts to generate the plots for the given number of array trials.
    - Two scripts are provided: a script for gnuplot and one for Python
- Run the experiment on a fully dedicated compute node.

For the Data Analysis

- Identify as many features as you can from the plot you generate and annotate this plot.
  - Similar to slide 8
- At a minimum, you are expected to identify:
  - L1 cache size, line size, and access time.
  - L2 cache size, line size, and access time.
  - L3 cache size, line size, and access time.
  - Main memory access time.
- Provide a reasoned explanation for why your data suggests that the <memory QOI> is <value>.
- Think about how you can verify the hardware specifications. Can you find info about the hardware and see if it matches the features you identify in your plot?
- For data that you are not quite sure how to explain, provide possible explanations as to what could be going on.
- If you are uncertain about values derived from your plot(s), or if the measurements contain some noise, it is ok to provide a range of values.
- You may have to create multiple plots to zoom in on certain features.

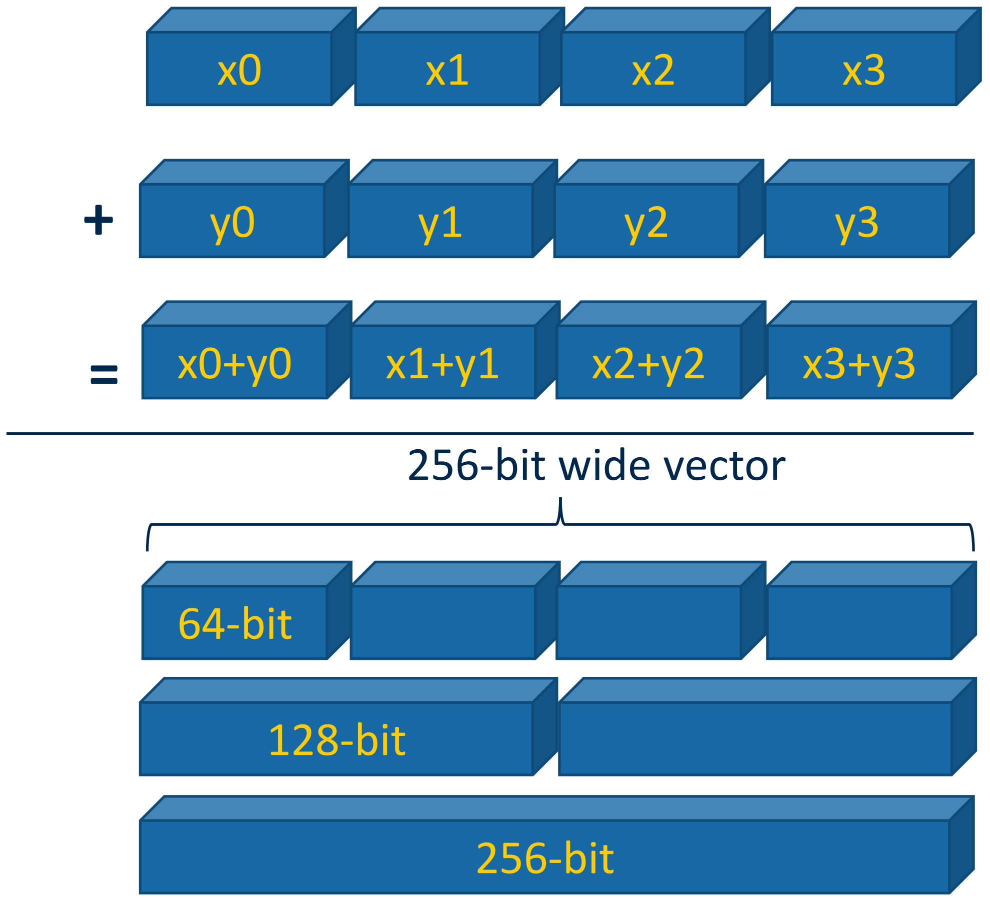
Note about node hardware



Host: nyx1040.arc-ts.umich.edu
Indexes: physical
Date: Tue 26 Jan 2016 12:47:28 PM EST

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


x0
x1
x2
x3
+
y0
y1
y2
y3
=
x0+y0
x1+y1
x2+y2
x3+y3
256-bit wide vector
64-bit
128-bit
256-bit