Computer Systems

CS107

Cynthia Lee

Today's Topics

TODAY'S LECTURE:

› Performance optimization and the memory hierarchy

ANNOUNCEMENTS:

› Assign6 & 7 NO late days allowed!
  - **Submit early and often**!
    - – Submit whatever you have that could be worth any points at all as soon as you have it, re-submit any time you improve it
  - That way if a catastrophe happens at deadline time and you can't submit, at least you have some points from your earlier submissions!

Valgrind for instruction profiling

Your code is awaited at the gates of Valgrindhalla! It shall ride eternal, shiny and chrome.

Remember this slide? dynamic instruction counts

For loop construction

```
# straightforward assembly
Initialization
Test
Branch past loop if fails
Body
Increment
jmp to Test
```

5n

```
# what gcc actually emits
Initialization
jmp to Test
Body
Increment
Test
Branch to Body if succeeds
```

4n

Automated tool for creating dynamic instruction counts

```
            /* Simple selection sort algorithm, O(N^2) */
            void selsort(int *arr, int n)
            {
    4,005       for (int i = 0; i < n; i++) {
    2,000           int min = i;
2,005,000           for(int j = i+1; j < n; j++)
7,003,974               if (arr[j] < arr[min]) min = j;
   13,000           swap(&arr[i], &arr[min]);
                }
            }
```

Valgrind can show you dynamic instruction counts

- Valgrind is a tool for memory profiling!
- But wait, there's more! Can also do other kinds of performance profiling

```
% valgrind --tool=callgrind --simulate-cache=yes ./array
% callgrind_annotate --auto=yes callgrind.out.[procID]
```

- › The last argument in the first call is the name of the executable to profile *(array is only an example)*
- › The last argument in the second call is the name of the output file created by the first call (procID is a number, will be different each time)
- › Add --inclusive=yes if you want to include the cost of function calls (by default it only counts code actually in the current function)

Performance example: sorting

Cycle counters can be a more precise way of capturing intuitions similar to what Big-O analysis captures

Sorting algorithms reminder

```
/* Simple selection sort algorithm, O(N^2) */
void selsort(int *arr, int n)
{
    for (int i = 0; i < n; i++) {
        int min = i;
        for(int j = i+1; j < n; j++)
            if (arr[j] < arr[min]) min = j;
        swap(&arr[i], &arr[min]);
    }
}

/* Simple insertion sort algorithm, O(N^2) */
void inssort(int *arr, int n)
{
    for (int i = 1; i < n; i++) {
        for (int j = i; j > 0 && arr[j-1] > arr[j]; j--) {
            swap(&arr[j], &arr[j-1]);
        }
    }
}
```

Sorting algorithms reminder: Big-O

- Quicksort: O(nlogn)
  - Assuming not consistently bad pivot selection
- Selection Sort: $O(n^2)$
- Insertion Sort: $O(n^2)$

- We can empirically verify our CS106B knowledge of these algorithms by using cycle counter for a few array sizes:

```
myth> ./sorts
Sorting  5000 elements:
qsort cycles 2.78M (opt)
selsort cycles 26.66M (opt)

Sorting 10000 elements:
qsort cycles 4.04M (opt)
selsort cycles 105.15M (opt)

Sorting 20000 elements:
qsort cycles 8.53M (opt)
selsort cycles 429.05M (opt)
```

Interpreting cycle counts

SORTING 1000 ELEMENTS:

- Alg-A 0.88M cycles
- Alg-B 5.31M cycles
- Alg-B 5.07M cycles (on already-sorted data)
- Alg-C 7.14M cycles
- Alg-C 0.01M cycles (on already-sorted data)

- Guess the algorithm:

A. A = quicksort, B = selection sort, C = insertion sort
B. A = quicksort, B = insertion sort, C = selection sort
C. Something else

Performance example: array access

We’ll look at a few different patterns of array access

We have code that accesses an array in different ways

$O(n)$

```
for (int i = 0; i < n; i++)     //normal access
        sum += a[i];
```

$O(n)$

```
for (int i = n-1; i >= 0; i--) //backwards access
        sum += a[i];
```

$O(n)$

```
for (int i = 0; i < n; i+=2)    //every other one (evens)
        sum += a[i];
for (int i = 1; i < n; i+=2)    //then odds
        sum += a[i];
```

$O(n)$

```
for (int i = 0; i < n; i++)     //random order
        sum += a[indexes[i]];
```

You can probably guess that the punchline is that these have very different performance profiles (they do).

We have code that accesses an array in different ways

1

```
for (int i = 0; i < n; i++) //forwards
        sum += a[i];
```

2

```
for (int i = n-1; i >= 0; i--) //backwards
        sum += a[i];
```

3

```
for (int i = 0; i < n; i+=2) //evens/odds
        sum += a[i];
for (int i = 1; i < n; i+=2)
        sum += a[i];
```

4

```
for (int i = 0; i < n; i++) //random order
        sum += a[indexes[i]];
```

How will their cycle counts rank? (guess)

A. 1< 2 < 3 < 4
B. 1 = 2 < 3 < 4
C. 1 < 2 = 3 < 4
D. 1 = 2 = 3 < 4
E. Something else

Taking a look at the assembly: FORWARD

```
080485f0 <sum_forward>:
 80485f0:       53                      push   %ebx
 80485f1:       b9 00 00 00 00          mov    $0x0,%ecx
 80485f6:       bb 00 00 00 00          mov    $0x0,%ebx
 80485fb:       eb 06                   jmp    8048603 <sum_forward+0x13>
 80485fd:       03 1c 88                add    (%eax,%ecx,4),%ebx
 8048600:       83 c1 01                add    $0x1,%ecx
 8048603:       39 d1                   cmp    %edx,%ecx
 8048605:       7c f6                   jl     80485fd <sum_forward+0xd>
 8048607:       89 d8                   mov    %ebx,%eax
 8048609:       5b                      pop    %ebx
 804860a:       c3                      ret
```

Taking a look at the assembly: BACKWARD

```
0804860b <sum_backward>:
 804860b:       83 ea 01                sub    $0x1,%edx
 804860e:       b9 00 00 00 00          mov    $0x0,%ecx
 8048613:       eb 06                   jmp    804861b <sum_backward+0x10>
 8048615:       03 0c 90                add    (%eax,%edx,4),%ecx
 8048618:       83 ea 01                sub    $0x1,%edx
 804861b:       85 d2                   test   %edx,%edx
 804861d:       79 f6                   jns    8048615 <sum_backward+0xa>
 804861f:       89 c8                   mov    %ecx,%eax
 8048621:       c3                      ret
```

Taking a look at the assembly: EVEN-ODD

```
08048656 <sum_evenodd>:
 8048656:       53                      push   %ebx
 8048657:       bb 00 00 00 00          mov    $0x0,%ebx
 804865c:       b9 00 00 00 00          mov    $0x0,%ecx
 8048661:       eb 06                   jmp    8048669 <sum_evenodd+0x13>
 8048663:       03 0c 98                add    (%eax,%ebx,4),%ecx
 8048666:       83 c3 02                add    $0x2,%ebx
 8048669:       39 d3                   cmp    %edx,%ebx
 804866b:       7c f6                   jl     8048663 <sum_evenodd+0xd>
 804866d:       bb 01 00 00 00          mov    $0x1,%ebx
 8048672:       eb 06                   jmp    804867a <sum_evenodd+0x24>
 8048674:       03 0c 98                add    (%eax,%ebx,4),%ecx
 8048677:       83 c3 02                add    $0x2,%ebx
 804867a:       39 d3                   cmp    %edx,%ebx
 804867c:       7c f6                   jl     8048674 <sum_evenodd+0x1e>
 804867e:       89 c8                   mov    %ecx,%eax
 8048680:       5b                      pop    %ebx
 8048681:       c3                      ret
```

Taking a look at the assembly: RANDOM

```
08048682 <sum_random>:
 8048682:       57                      push   %edi
 8048683:       56                      push   %esi
 8048684:       53                      push   %ebx
 8048685:       bb 00 00 00 00          mov    $0x0,%ebx
 804868a:       be 00 00 00 00          mov    $0x0,%esi
 804868f:       eb 09                   jmp    804869a <sum_random+0x18>
 8048691:       8b 3c 99                mov    (%ecx,%ebx,4),%edi
 8048694:       03 34 b8                add    (%eax,%edi,4),%esi
 8048697:       83 c3 01                add    $0x1,%ebx
 804869a:       39 d3                   cmp    %edx,%ebx
 804869c:       7c f3                   jl     8048691 <sum_random+0xf>
 804869e:       89 f0                   mov    %esi,%eax
 80486a0:       5b                      pop    %ebx
 80486a1:       5e                      pop    %esi
 80486a2:       5f                      pop    %edi
 80486a3:       c3                      ret
```

And the winner is…..

```
myth> ./array
This program sums a 1000000-elem array. It times the
traversal forward, backward, even/odd, vs randomly.

Sum array forward:    9.40M cycles
Sum array backward:   9.63M cycles
Sum array even/odd:   9.64M cycles
Sum array random:    36.29M cycles
```

Looking for answers in the code

```
for (int i = 0; i < n; i++) //forwards
       sum += a[i];

for (int i = n-1; i >= 0; i--) //backwards
       sum += a[i];

for (int i = 0; i < n; i+=2) //evens/odds
       sum += a[i];
for (int i = 1; i < n; i+=2)
       sum += a[i];

for (int i = 0; i < n; i++) //random order
       sum += a[indexes[i]];
```

```
forward:     9.40M cycles
backward:    9.63M cycles
even/odd:    9.64M cycles
random:     36.29M cycles
```

Does anything about this comparison strike you as particularly unfair?

Making the code more fair ("apples to apples" comparison)

```
for (int i = 0; i < n; i++) //forwards
        sum += a[i];

for (int i = 0; i < n; i++) //random order
        sum += a[indexes[i]];  // must access memory TWICE!

for (int i = 0; i < n; i++) //forwards, but with 2-step index
        sum += a[indexes[i]];  // must still access TWICE
```

Taking a look at the assembly: RANDOM

```
08048682 <sum_random>:
 8048682:       57                      push   %edi
 8048683:       56                      push   %esi
 8048684:       53                      push   %ebx
 8048685:       bb 00 00 00 00          mov    $0x0,%ebx
 804868a:       be 00 00 00 00          mov    $0x0,%esi
 804868f:       eb 09                   jmp    804869a <sum_random+0x18>
 8048691:       8b 3c 99                mov    (%ecx,%ebx,4),%edi
 8048694:       03 34 b8                add    (%eax,%edi,4),%esi
 8048697:       83 c3 01                add    $0x1,%ebx
 804869a:       39 d3                   cmp    %edx,%ebx
 804869c:       7c f3                   jl     8048691 <sum_random+0xf>
 804869e:       89 f0                   mov    %esi,%eax
 80486a0:       5b                      pop    %ebx
 80486a1:       5e                      pop    %esi
 80486a2:       5f                      pop    %edi
 80486a3:       c3                      ret
```

Taking a look at the assembly: FWD-INDEX

```
080486a4 <sum_fwd_ind>:
 80486a4:       57                       push   %edi
 80486a5:       56                       push   %esi
 80486a6:       53                       push   %ebx
 80486a7:       bb 00 00 00 00           mov    $0x0,%ebx
 80486ac:       be 00 00 00 00           mov    $0x0,%esi
 80486b1:       eb 09                    jmp    80486bc <sum_fwd_ind+0x18>
 80486b3:       8b 3c 99                 mov    (%ecx,%ebx,4),%edi
 80486b6:       03 34 b8                 add    (%eax,%edi,4),%esi
 80486b9:       83 c3 01                 add    $0x1,%ebx
 80486bc:       39 d3                    cmp    %edx,%ebx
 80486be:       7c f3                    jl     80486b3 <sum_fwd_ind+0xf>
 80486c0:       89 f0                    mov    %esi,%eax
 80486c2:       5b                       pop    %ebx
 80486c3:       5e                       pop    %esi
 80486c4:       5f                       pop    %edi
 80486c5:       c3                       ret
```

100 cycles

We have code that accesses an array in different ways

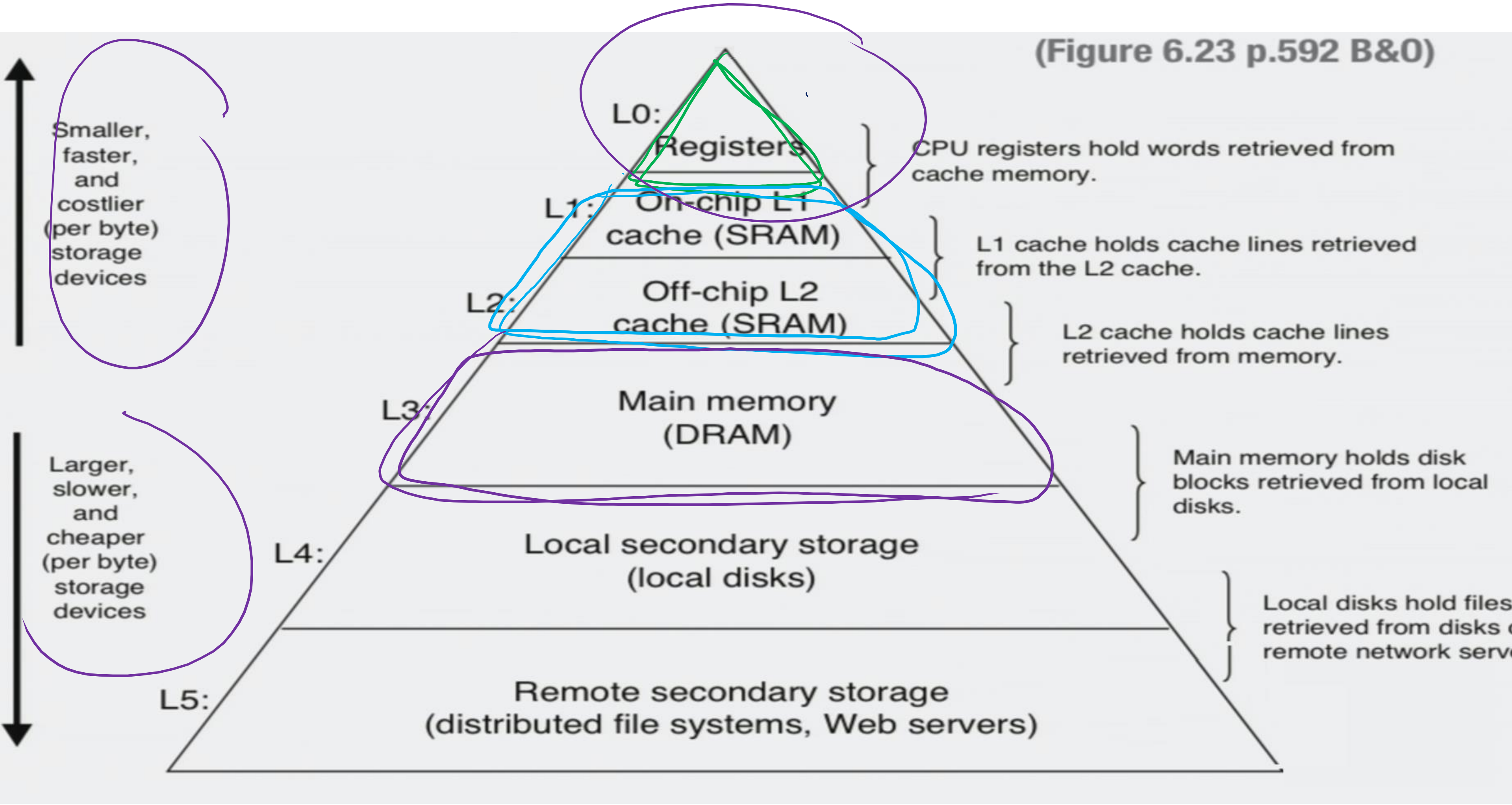
```
myth10:/usr/class/cs107/samples/lect16> ./array
This program sums a 1000000-elem array. It times the
traversal forward, backward, even/odd, vs randomly.

Sum array forward:     9.40M cycles
Sum array backward:    9.63M cycles
Sum array even/odd:    9.64M cycles
Sum array random:     36.29M cycles
Sum array fwd ind:    11.89M cycles
Sum array unrolled:    4.78M cycles
```

Q: Why??

A: THE MEMORY HIERARCHY

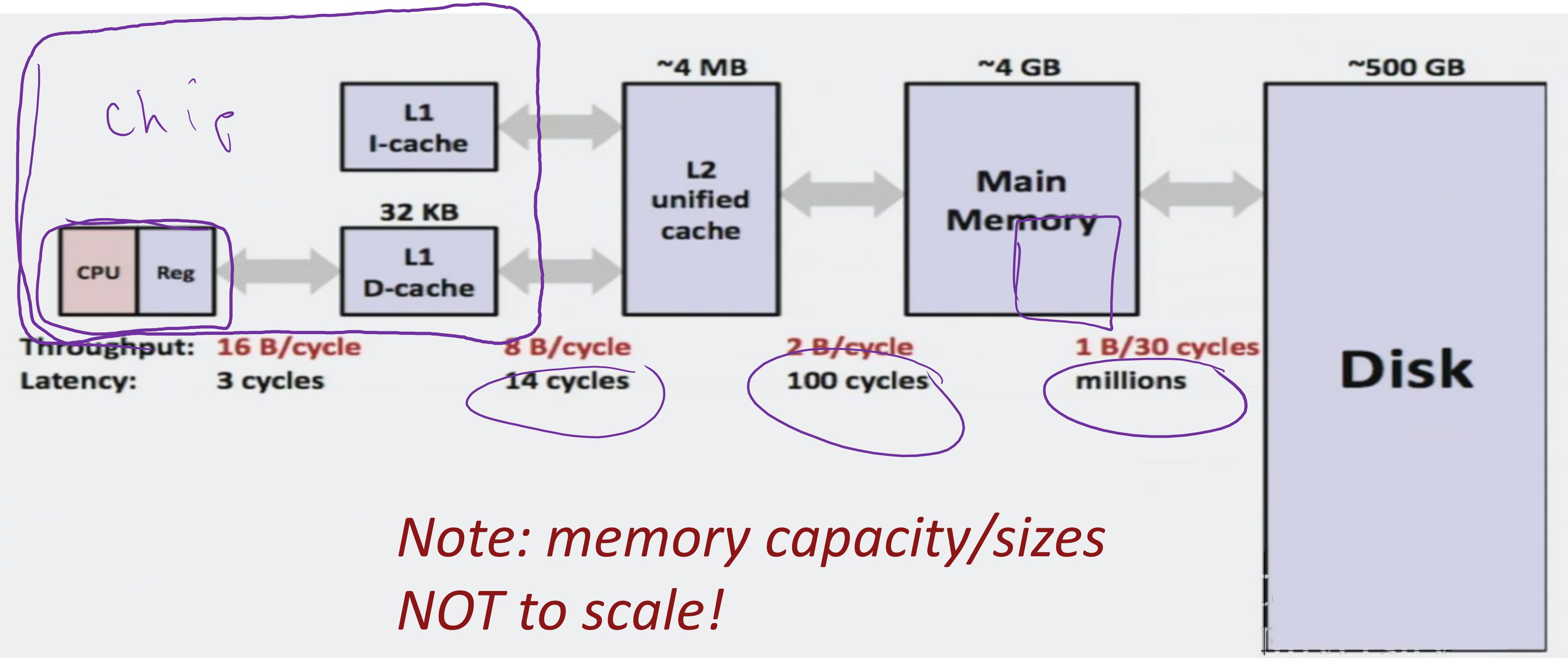
Why?? The memory hierarchy



Cache Performance Goal

- Goal:
  - We want to have the entirety of main memory available to the ALU to perform operations at the speed of register access
- Reality says: LOL / “We all want things”
- It’s partly the cost that is a barrier (smaller/faster memory is more expensive), but there are also physical limitations

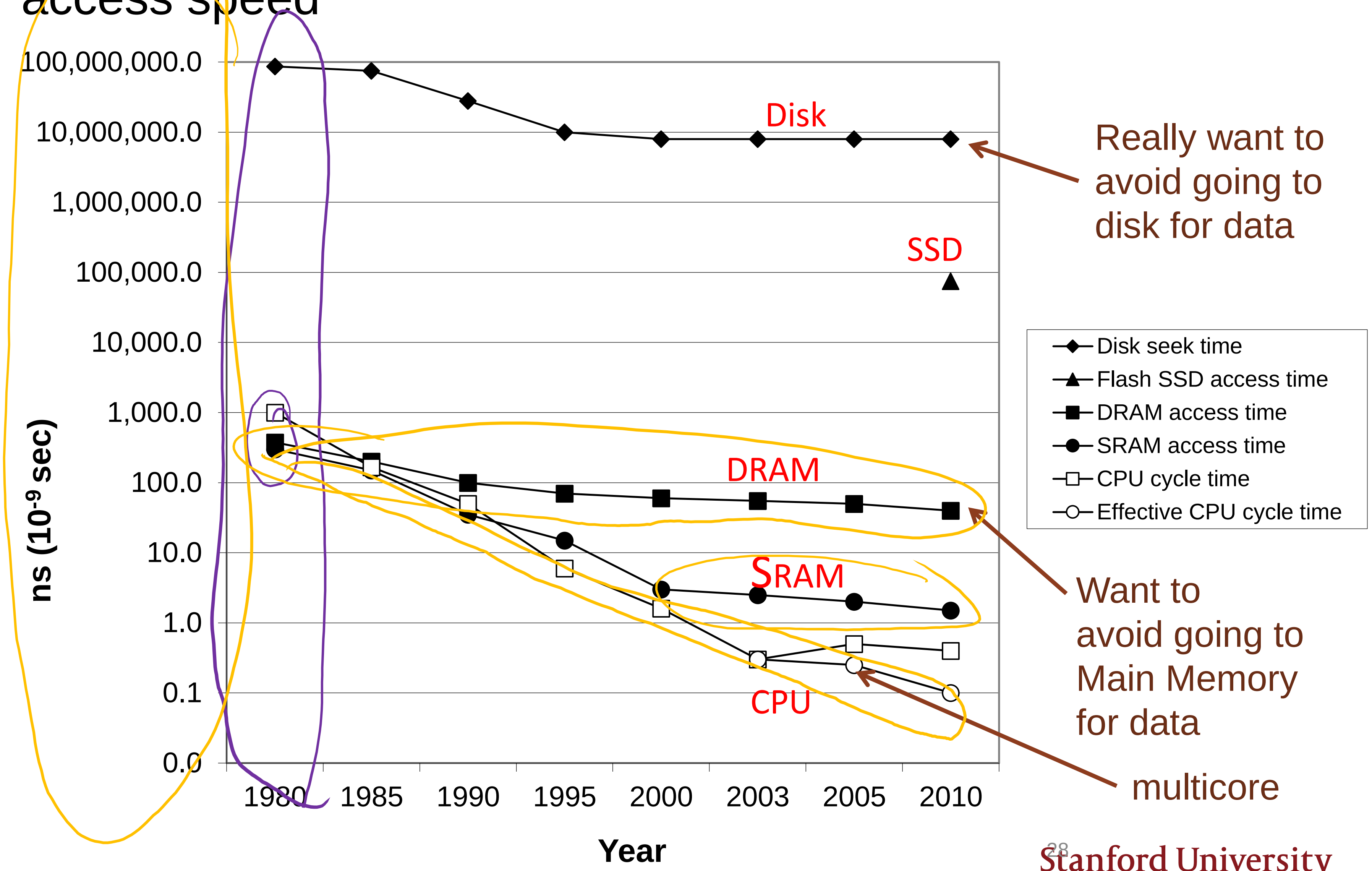
The memory hierarchy



Note: memory capacity/sizes NOT to scale!

Mapping the Memory Hierarchy

Growing gap between processor speed and memory access speed

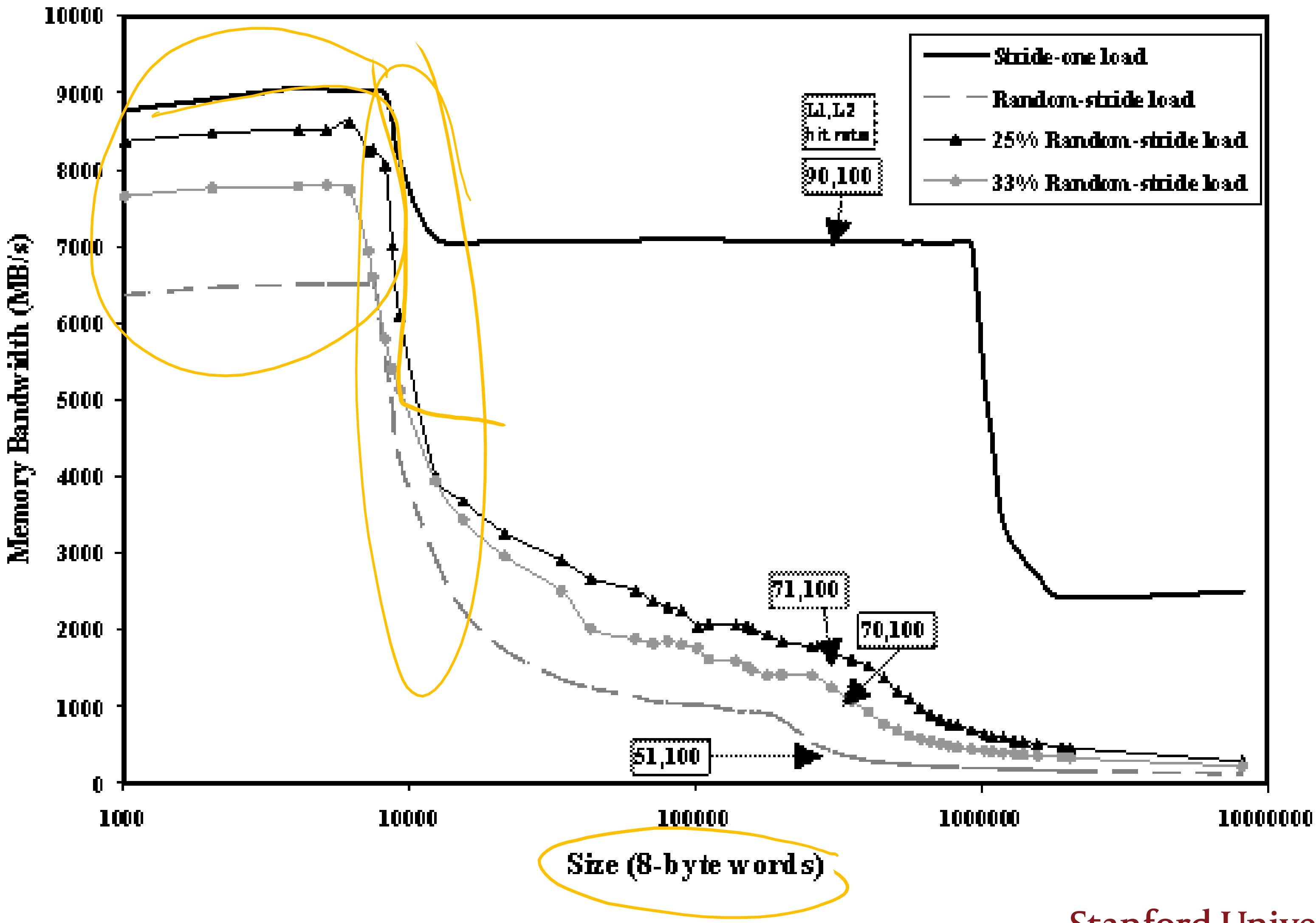


Really want to avoid going to disk for data

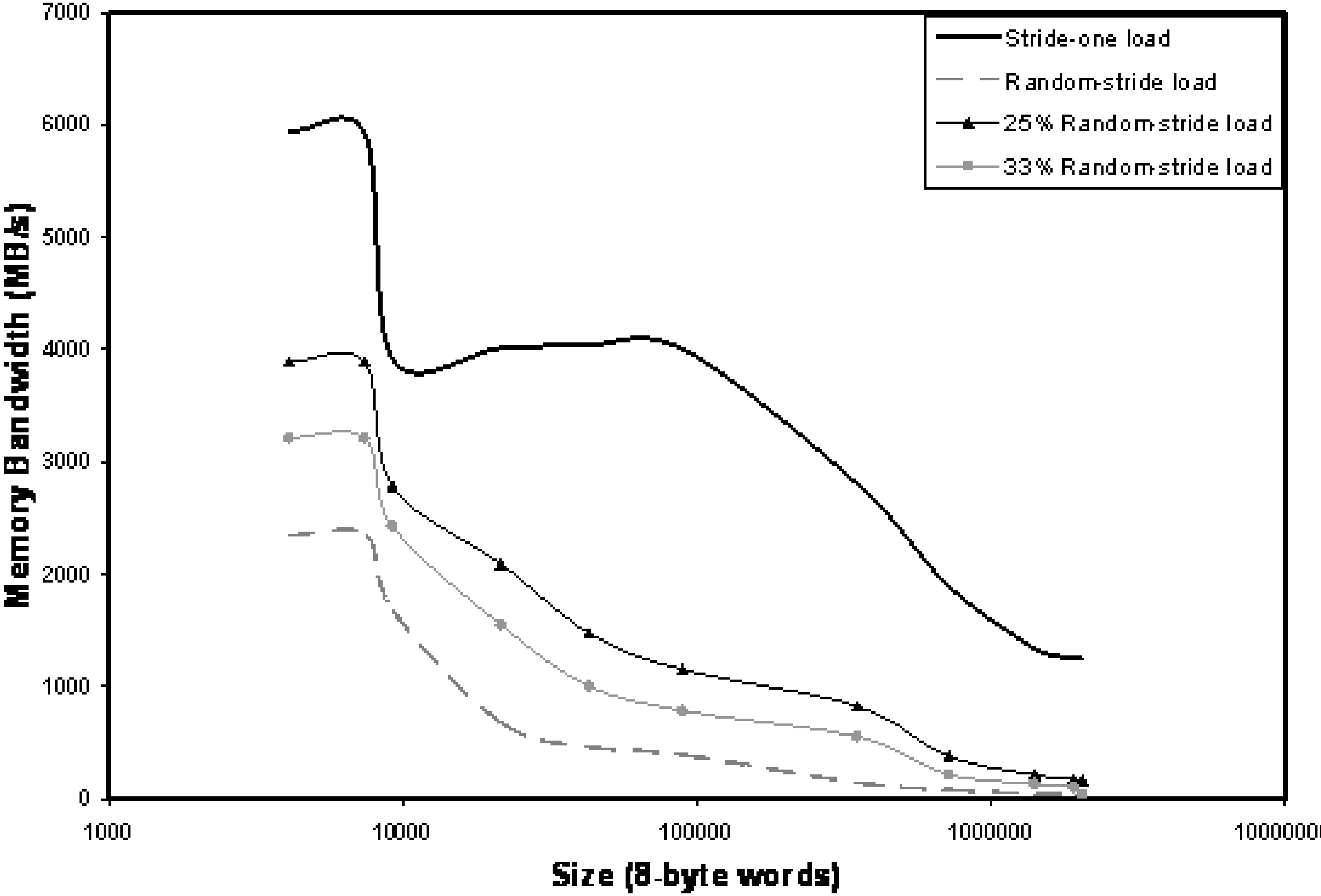
Want to avoid going to Main Memory for data

multicore

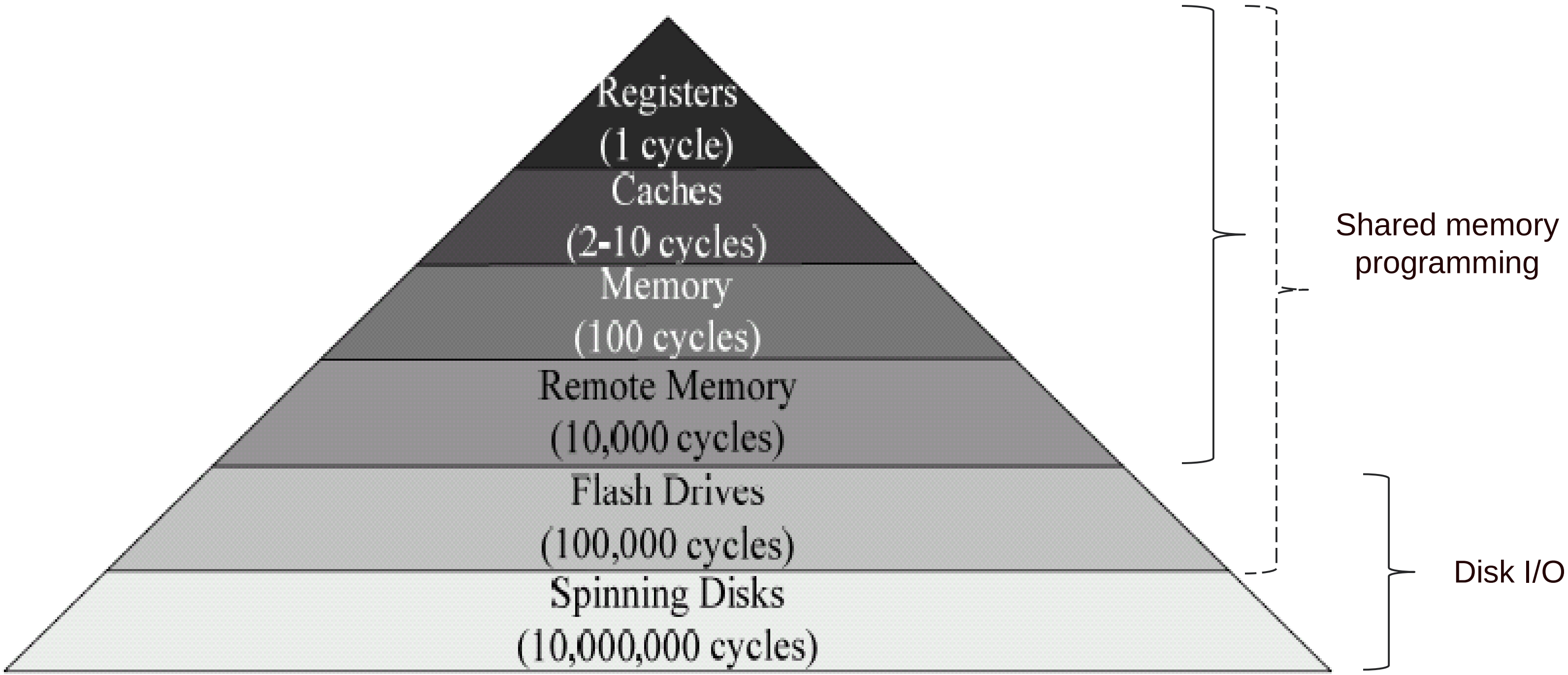
Memory Map of a Compaq Alphaserver



Memory Map of an IBM SP-3



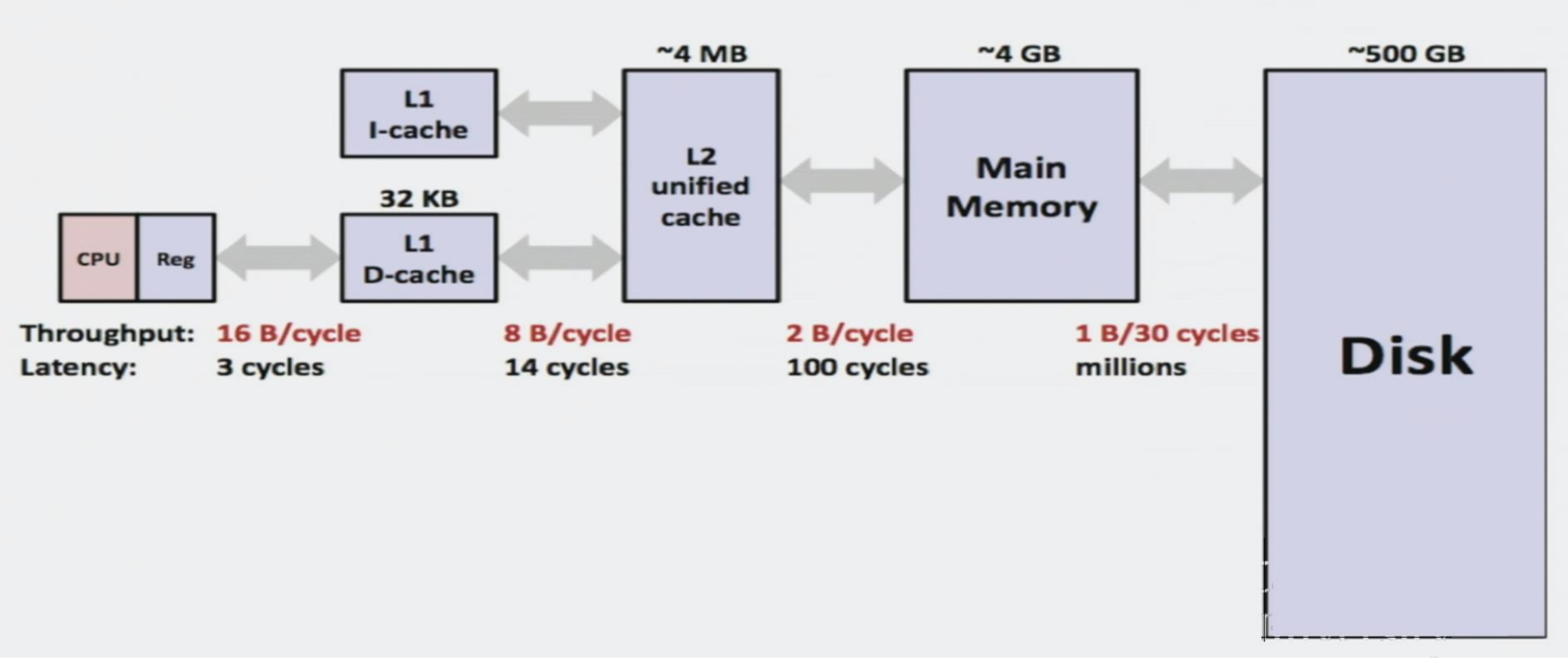
The Memory Hierarchy of Gordon (a large "supercomputer")



Caching basics

Principles of caching

The memory hierarchy



All of caching relies on LOCALITY

- **Temporal locality**:
  - Uses of the same piece of data tend to be near each other in TIME
  - Things I have recently used, I am more likely to use again
  - *Ugliest shirts that I should probably just give away are at the very bottom of my shirt drawer, becasue I never wear them*
- **Spatial locality**:
  - Uses of pieces of data tend to be near each other in SPACE
  - Even if I have never used something, if it is near a used item, it is more likely to be used soon
  - *Coat closet gets ignored during summer, but once autumn hits and I use one item from there (scarf) it is likely I will soon use other items from there (various jackets)*

Cache outcomes

- **Cache hit**:
  - What I wanted is in cache—lucky me!
  - Hit rate: % of accesses that are cache hits

- **Cache miss**:
  - What I wanted is not in cache—sad times!
  - Go to main memory (or lower level of cache) to get the item, will take much longer

Example: cache performance

- Pretty realistic scenario:
  - 97% cache hit rate
  - Cache hit: 1 cycle to access
  - Cache miss: 100 cycles to access
- **What percent of your total memory access time is spent on the 3% of memory accesses that are cache misses?**
  A. <=3%
  B. 30%
  C. 50%
  D. >50%
  - Bonus discussion: How much does this change if your cache hit rate goes up slightly to 99%?