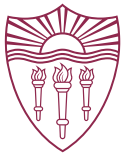


EE/CSCI 451: Parallel and Distributed Computation

Lecture #3

8/25/2020

Viktor Prasanna

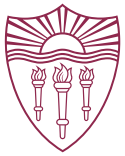
prasanna@usc.edu

ceng.usc.edu/~prasanna

University of Southern California

USC University of Southern California

Outline



- From last class
  - Processor organization
  - Implicit parallelism

- Today (Chapter 2.2)
  - Memory systems
    - Latency, Bandwidth: performance implications
    - Cache: impact on performance
    - Data layout
    - Latency Hiding: Multithreading, Prefetching

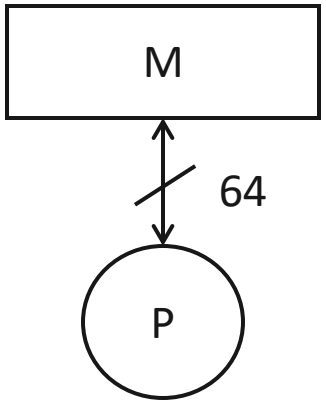
Announcements

- Homework 1
  - Due: September 1 2020, midnight AOE (Any time On Earth)
  - Submit typewritten/scanned copy via blackboard

- Programming Homework 1
  - Due: August 28 2020, midnight AOE
  - Submit zip file via Blackboard

Memory system

- System performance (execution time) depends on the "rate" at which data can be accessed by the program



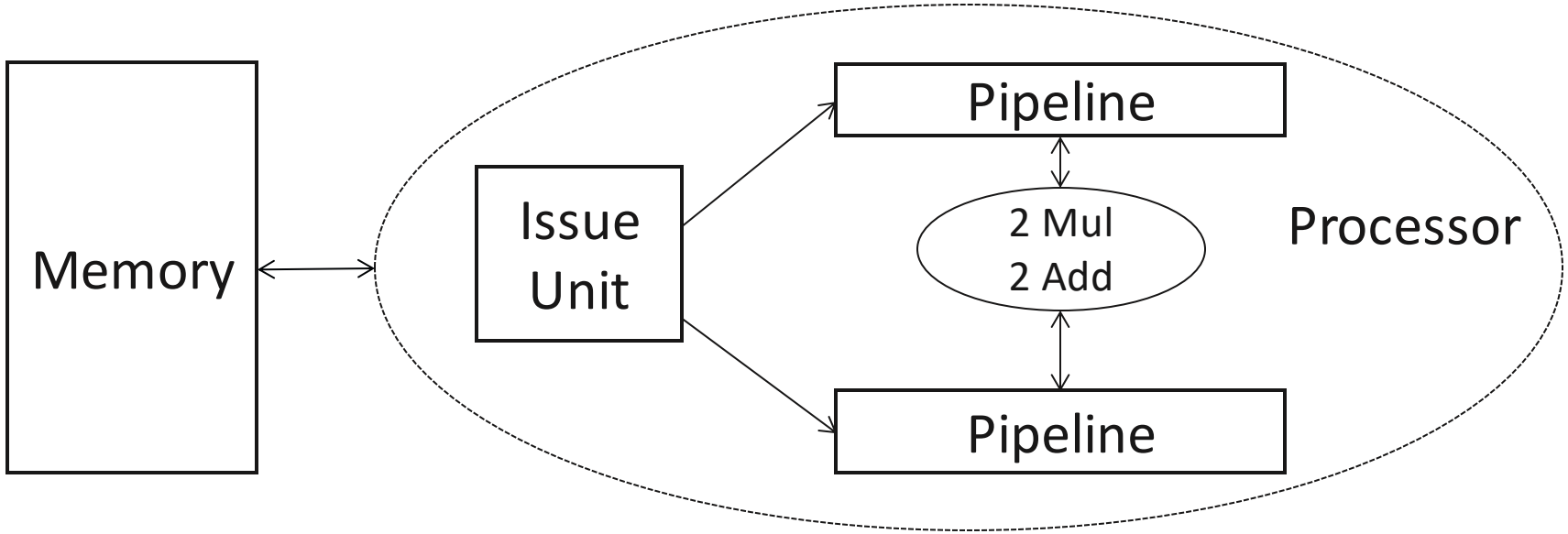
- **(DRAM) Hardware Latency** ~ 10 ns
- **(Peak) Processor-Memory Bandwidth** ~ 64 bits at 1 GHz
  (64 Gbits/sec or 8 GB/sec)
- Single cycle processor

- Effective (DRAM) bandwidth depends on access stride

  e.g. access consecutive addresses $i, i+1, \ldots$ [0,1,2,3,…]

  or arbitrary sequence $i_1, i_2, \ldots, i_k, \ldots$ [0,10,50,2,…]

System Organization



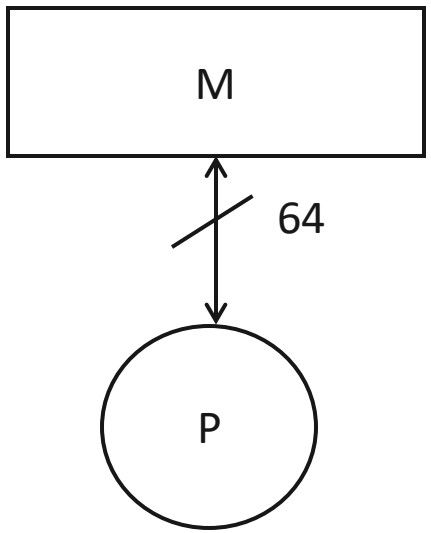
Performance of an application depends on

- Memory b/w and latency
- # of pipelines
- # of Functional Units (FPUs)

Example memory system performance (1)

Memory

Latency = 10 ns
(Peak) Bandwidth = 64 bits at 1 GHz
(64 Gbits/sec or 8 GB/sec)
(**bus frequency**)



Processor

2 GHz, 1 word = 64 bits (8 bytes)
Unit of data access = 1 word
Cycle time = 0.5 ns (processor cycle)
2-issue superscalar, single cycle processor
2 Double Precision multiply-add (2 multipliers, 2 adders) FPUs

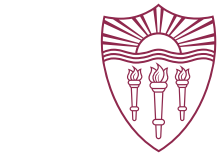
Peak performance of the processor = 2×4 = 8 GFlops/s
(Raw compute power)

Clock rate
Total # of FP ops /cycle

Peak performance can also be computed as 2×2 = 4 GFlops/s

Clock rate
2 pipelines

Example memory system performance (2)
Best Case ?

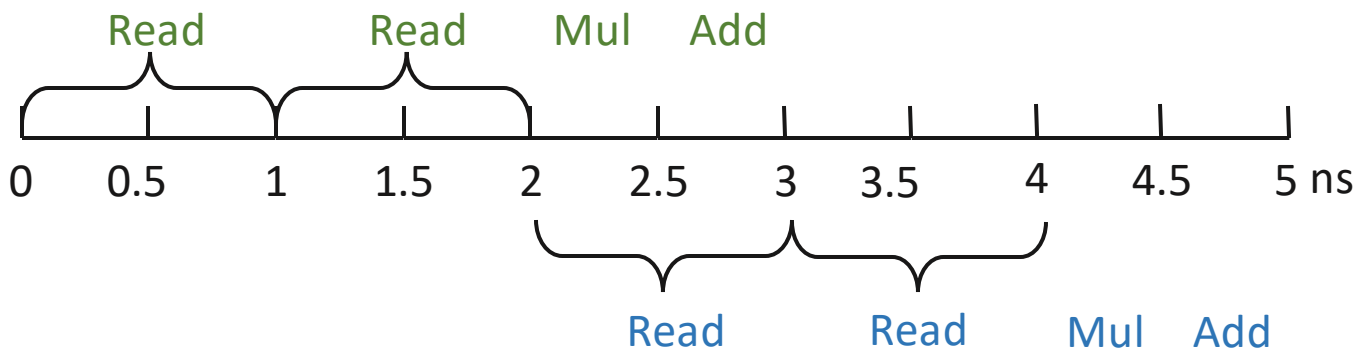


Example: Inner product: $a \cdot b = \sum_{i=1}^{n} a_i b_i$ (data in external memory)

2 data fetches for each multiply, add

Processor can do 8 mult or add/ns = 4 FP ops /cycle

Note: issue bandwidth limits performance to 2 FP ops/cycle (if no Fused mult add)



If we can stream the data (ie no DRAM latency (10ns) per access),

In the **best case**, i.e. , data is streamed:

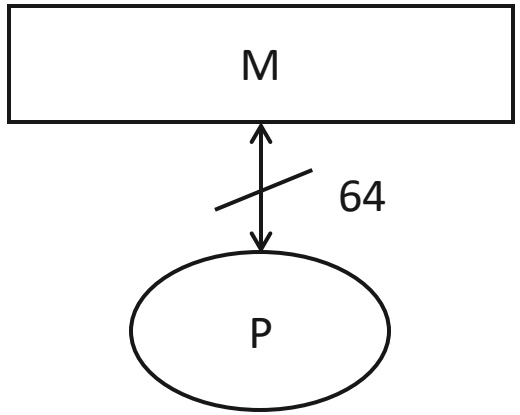
Sustained performance (possible best case) = 2 ops over 4 processor cycles = **1 GFlop/s**

Note: Initial access latency can be ignored for large *n*

Processor memory bandwidth determines possible best case sustained performance

Note: Processor idles, waiting for data.

Example memory system performance (3)
Worst Case ?



Memory Latency = 10 ns
(Peak) Bandwidth = 64 bits at 1 GHz
(64 Gbits/sec or 8 GB/sec)

Repeat

1. read a_i ⟵ 10 ns
2. read b_i ⟵ 10 ns
3. multiply (use register data)
4. add (use register data)

n times

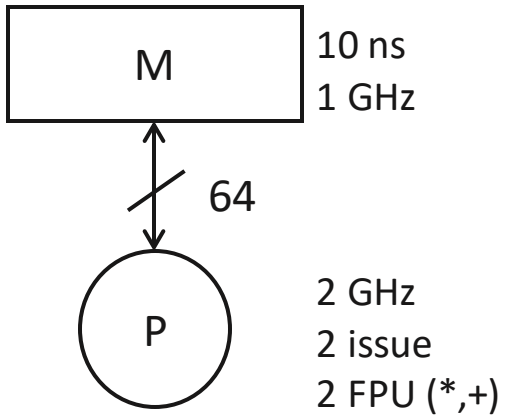
Overlap Instr. 3, 4 with Instr. 1, 2 (Pipelining)

Sustained performance (possible worst case)

Memory access time

$$\frac{\text{Total \# of FP ops}}{\text{Total time}} \approx \frac{2n}{20n \times 10^{-9}} = \textbf{0.1 GFlops/s}$$

Example memory system performance (4)



Inner product

Processor Peak = 8 GFlops/s (Raw compute power)

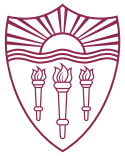
Processor Peak = 4 GFlops/s (Processor organization)

(Best case)
Sustained = 1 GFlop/s (Memory b/w bottleneck)

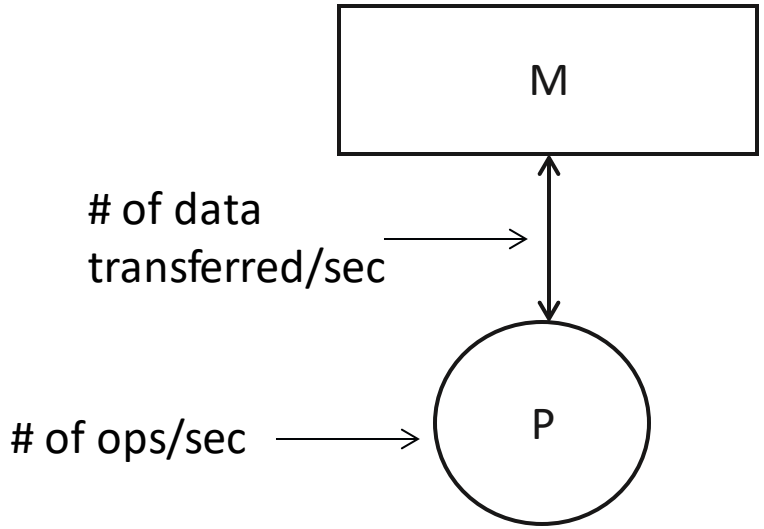
(Worst case)
Sustained = 0.1 GFlops/s (Memory latency bottleneck)

Memory Performance dictates overall application performance

Compute Bound/Memory (bandwidth) Bound (1)

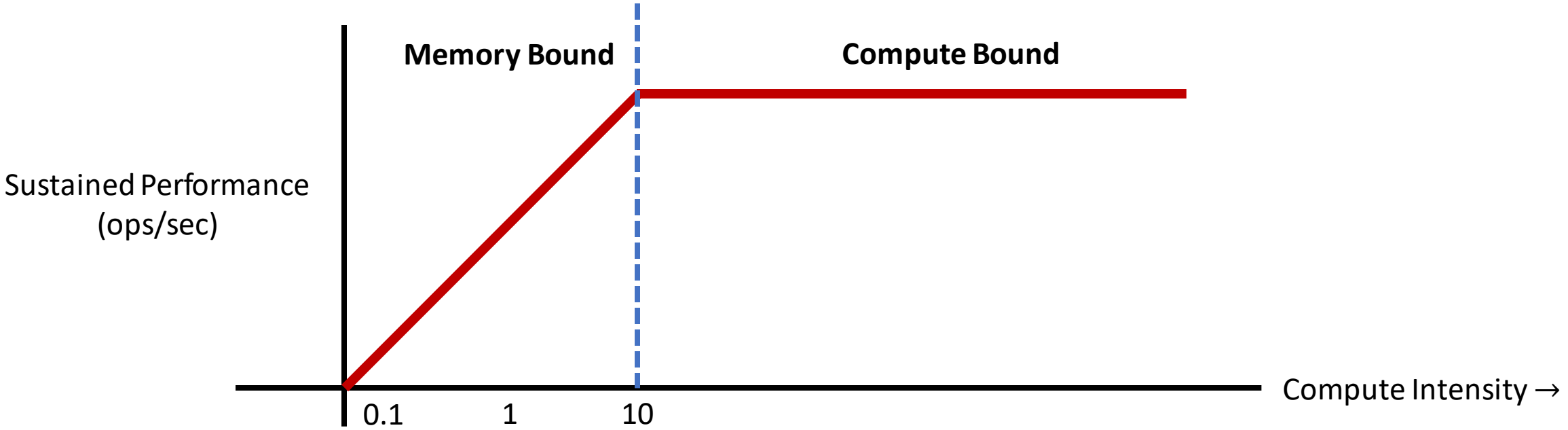


- Memory gap/Memory wall
- **Processor Performance ≫ Memory Performance**
- Processor may idle waiting for data



- Compute intensity = For every fetched data (from memory) how many ops make use of them

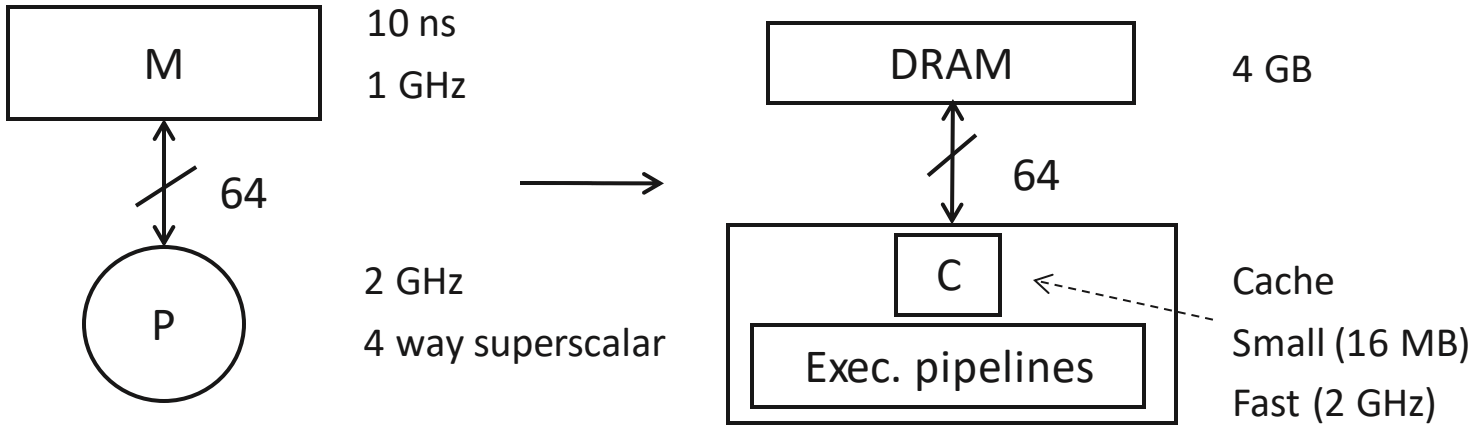
Compute Bound/Memory (bandwidth) Bound (2)



- Compute Bound: Performance is limited by available compute resources i.e. if we add more compute resources, the performance may improve (for a given bandwidth)

Cache (1)

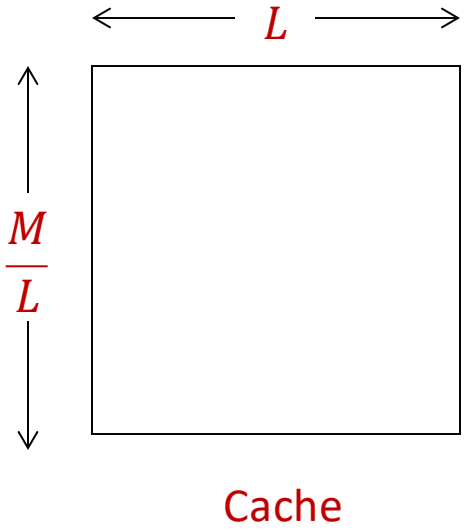
Improving effective memory access latency



- Why it works?
  - Data reuse
    - Accessing from DRAM is expensive (DRAM latency)
    - Repeatedly use the data in cache, if possible (fast access)
  - Hit ratio: fraction of the memory references served by the cache

Cache (2)

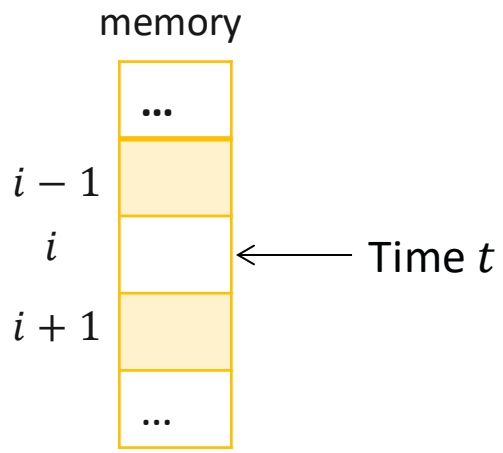
- Cache characteristics
  - Cache size (M), Cache line size (L)
  - Unit of DRAM access—line size
  - Cache organization (virtual to physical mapping)
    - Direct Mapped
    - Set Associative
    - Fully Associative
  - Cache replacement policy (what is stored in cache?0
    - First in first out
    - Most recently used
    - Random replacement
    - …
  - Write policy
    - Write through (to cache and to DRAM)
    - Write back



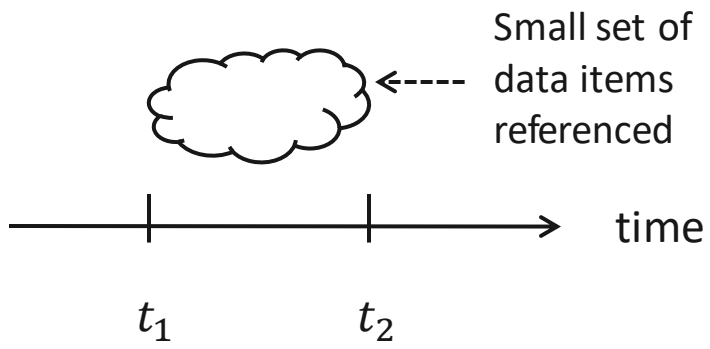
Cache (3)

Locality of references (program behavior)

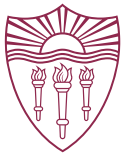
- **Spatial locality**
  - If location i is referenced at time t, then locations near i are referenced in a small window of time following t
  - Data in a cache **line** is effectively used



- **Temporal locality**
  - In a small window of time, repeated references to a small set of data items
  - Reduces traffic to main memory



Cache (4)



Example: bubble sort

```
for i from 1 to N
    for j from 0 to N-2
        R1 ⟵ a[j]
        R2 ⟵ a[j+1]
        if R1 > R2
            swap (R1 and R2)
        end if
        a[j]   ⟵ R1
        a[j+1] ⟵ R2
    end for
end for
```

Assumptions:

- Direct mapped
- First in first out policy
- Cache line size = 1 data element
- Cache size 2 data elements

$i = 1$ Read a[0], a[1] → miss, miss
Read a[1], a[2] → hit, miss
Read a[2], a[3] → hit, miss
… …
Read a[6], a[7] → hit, miss

$i = 2$ Read a[0], a[1] → miss, miss
Read a[1], a[2] → hit, miss
…

Cache hit ratio = $\frac{\text{\# of times data found in cache}}{\text{Total \# of accesses}}$

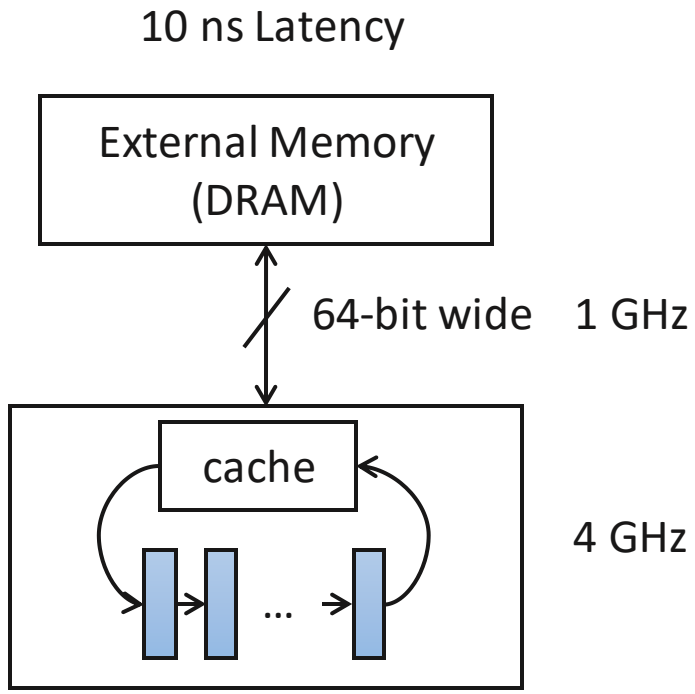
$i = 1$, Read hit ratio = $\frac{N-2}{(N-1)*2} = \frac{6}{7*2} = 0.43$

Impact of cache on performance (1)

- Matrix multiplication with on-chip cache

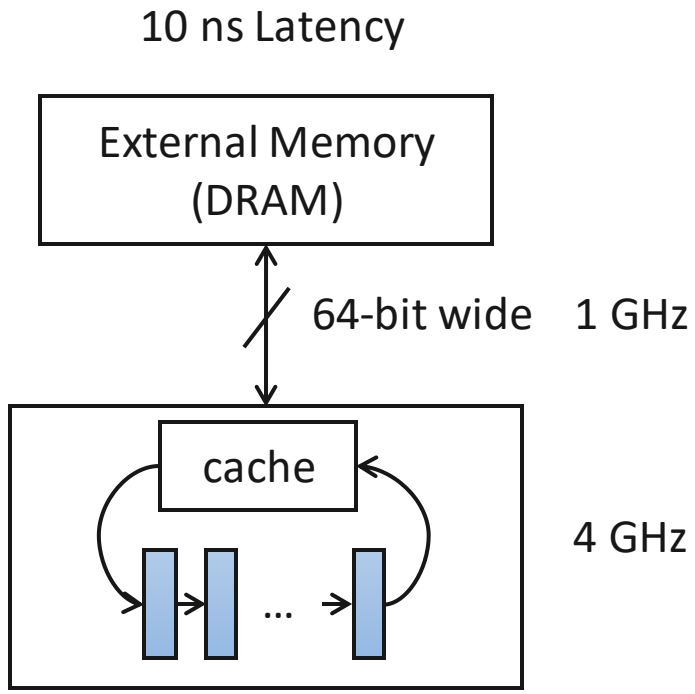
$$\boldsymbol{C} = (\boldsymbol{A} \times \boldsymbol{B})_{\boldsymbol{32\times32}}$$

- 4 GHz Processor + DRAM
- 2 issue/cycle
- 5 stage pipeline
- Cache access latency = processor cycle time (0.5 ns)
- Multiple access to cache

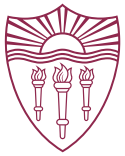


Impact of cache on performance (2)

- Steps:
  - Read $\boldsymbol{A}$ into cache
  - Read $\boldsymbol{B}$ into cache
  - Perform n^3 multiply and n^3 add to compute $\boldsymbol{C}$
  - Store the results in cache
  - Store the results from cache to DRAM
  - Cache size $\geq$ 3K words for 32×32 matrix multiplication

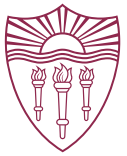


Impact of cache on performance (3)



- Assume
  - No cache misses (no conflicts) – doable by intelligent data placement
  - Ideal case (no data dependencies) – doable by program restructuring
  - There is some overhead (load, compute) – 2 instructions / compute step
- Case 1: 10 ns latency / memory access
  - Case 1a: with cache
  - Case 1b: without cache
- Case 2: 1 ns latency / memory access (streaming)
  - Case 2a
  - Case 2b

Impact of cache on performance (4)



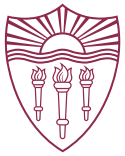
- Case 1a: **worst case access latency to memory with cache**
  - Read $\boldsymbol{A}$ into cache: (32× 32) ×10 ns = 10 = 10 μs
  - Read $\boldsymbol{B}$ into cache: 10 μs
  - Perform $2n^3$ mult/add to compute $\boldsymbol{C}$ and store in cache:
    $2\times0.25\text{ ns}\times2\times 32^3/2 = 16.384\ \mu s$
  - Store the results from cache to DRAM: 10 μs
  - Total time: 46.384 μs
  - Total number of ops: $2n^3 = 2(32)^3 = 65{,}536$

Load
Compute

Sustained performance: $\frac{65536}{46.384\times10^{-6}} = \mathbf{1.41\ GFlops/s}$

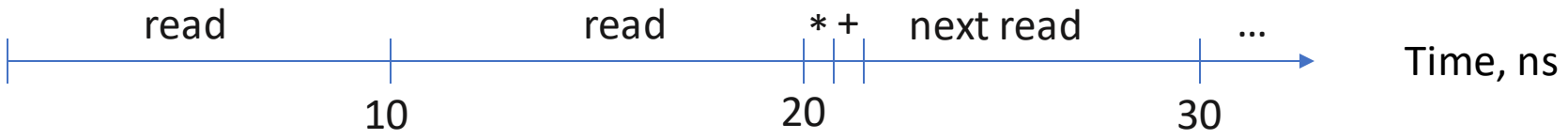
Note: we used worst case access latency to external memory

Impact of cache on performance (5)

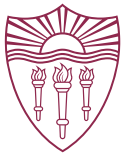


- Case 1b: **worst case access latency to memory without cache**

- Sustained performance
  - Latency of 10 ns per access to each data
  - 2 words lead to one Multiply and one Add
  - Overlap read with arithmetic operations
  - = 2/20 GFlops/s
    = 0.1 GFlops/s

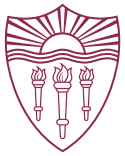


Impact of cache on performance (6)



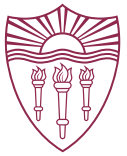
- Case 2a: **best case access latency to memory with cache**
  - Total time = time to compute + data access
    = 16.384+1+1=18.384 μs (time to load matrix A and B = 1 μs)
  - Sustained Performance = $\frac{65536}{18.384\times10^{-6}}$ = 3.56 GFLOPS/s
- Case 2b: **best case access latency to memory without cache**
  - Two operations for two data access, streaming access at 1 ns per data
  - Sustained Performance = $\frac{2}{2\times1\times10^{-9}}$ = 1 GFLOPS/s

Impact of cache on performance (7)

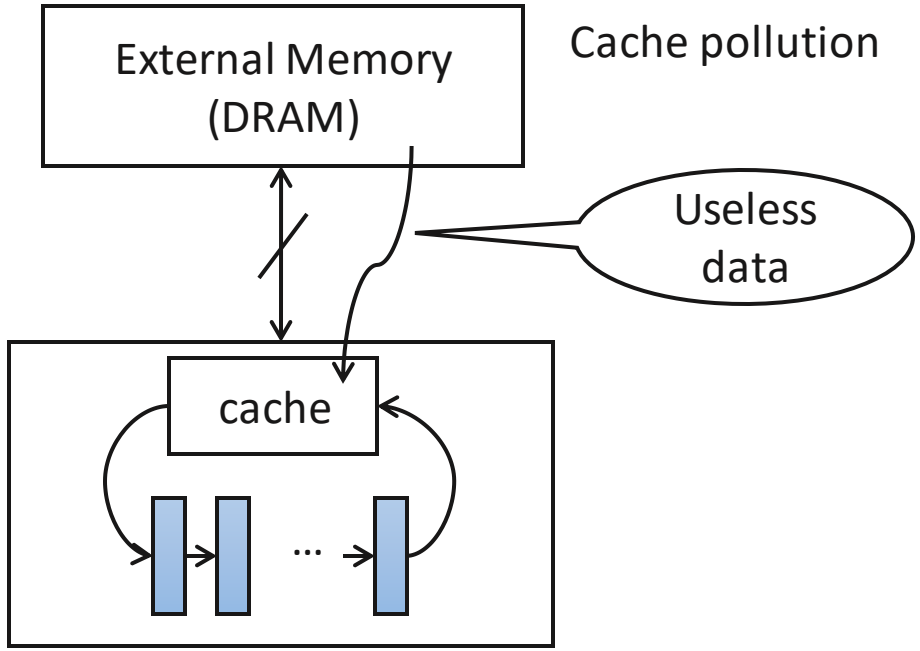


- Performance improvement due to cache
  - Sustained performance (with cache)/Sustained performance (without cache)
  - Case 1: **1.41/0.1 ≈ 14x**
  - Case 2: **3.56/1 ≈ 3.5x**
- Note
  - Parallel algorithm should be carefully designed to achieve such speed-ups by exploiting cache
  - Idea can be extended for large scale matrix multiplication (matrix size > cache size)

Cache pollution and sustained performance (1)



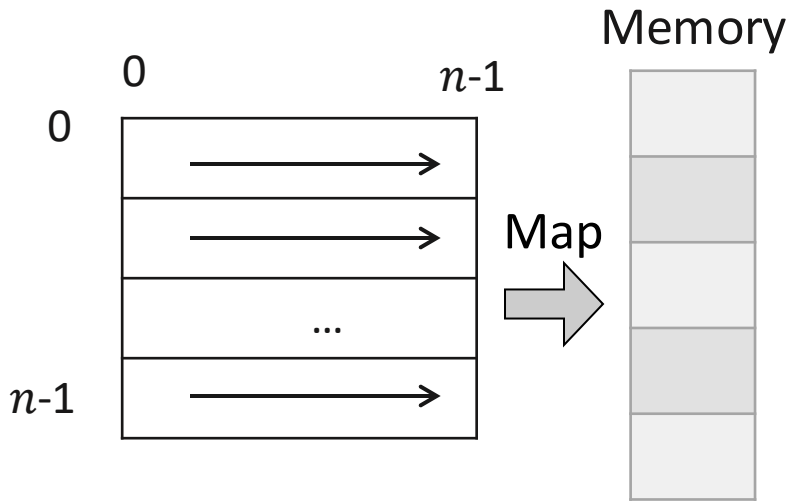
- Cache pollution
  - **All** fetched data in a cache line is not needed/used by the program
    - Reason: Mismatch between the way data is stored in DRAM and access pattern of the program (Low Spatial Locality)



Data layout and data access pattern (1)

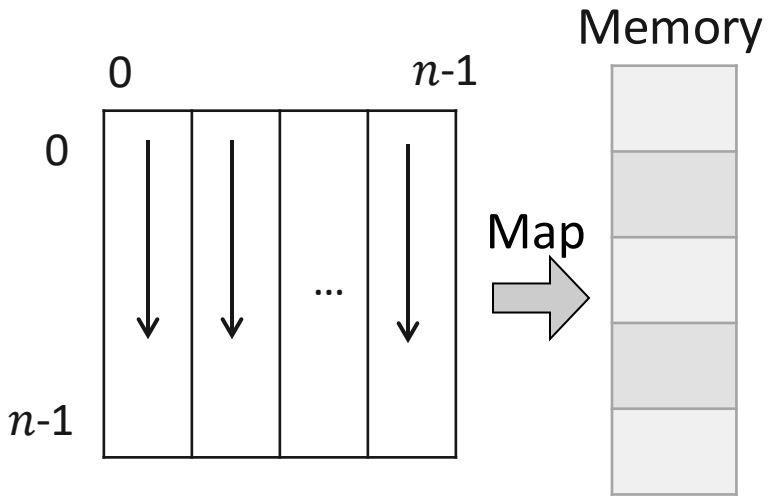
Storing a 2-dimemsional array in memory

Row major order



$$A(i,j) \rightarrow \text{Memory}(i \cdot n + j)$$
$$0 \leq i,j < n$$

Column major order



$$A(i,j) \rightarrow \text{Memory}(i + n \cdot j)$$
$$0 \leq i,j < n$$

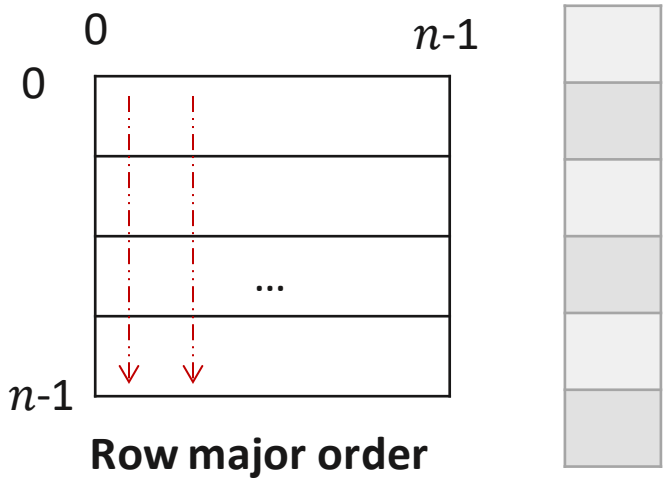
Data layout and data access pattern (2)

- Example
  - Cache line 4 words
  - Sum each row of a matrix

$$B(i) = \sum_{j=0}^{n-1} A(i,j) \quad 0 \le i < n$$

  - Assume A is stored in row major order
  - Implementation 1:
    1. $B(i) \leftarrow 0, 0 \le i < n$
    2. for $j = 0$ to $n-1$ do
    3. for $i = 0$ to $n-1$ do
    4. $B(i) \leftarrow B(i) + A(i,j)$

Column major access pattern



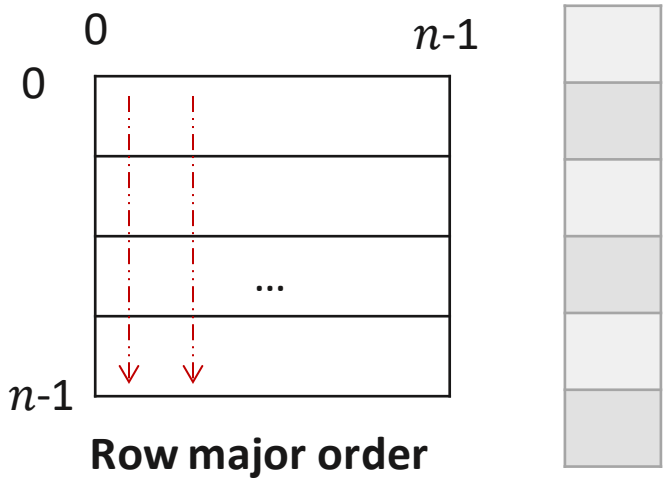
Row major order

$$A(i,j) \rightarrow \text{Memory}(i \cdot n + j)$$
$$0 \le i,j < n$$

Data layout and data access pattern (3)

- Example (cont.)
  - Poor spatial locality
    - cache pollution
    - memory cache line access: only one word is used

Access pattern of
the program

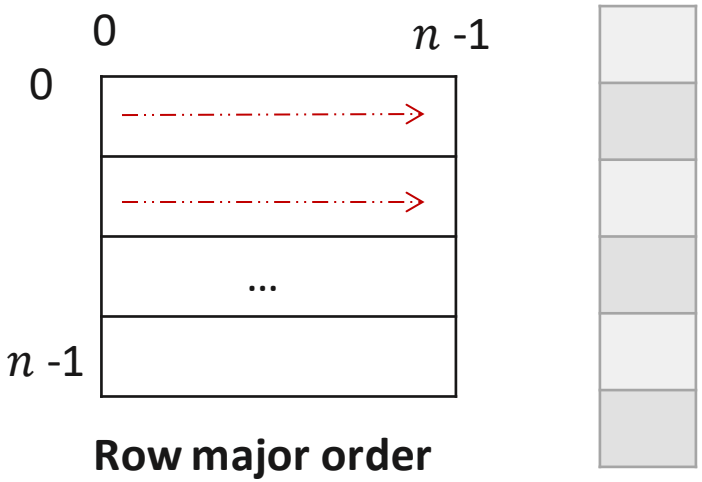


$$A(i,j) \rightarrow \text{Memory}(i \cdot n + j)$$
$$0 \leq i,j < n$$

Data layout and data access pattern (4)

- Example –alternate implementation
  - Assume A is stored in row major order, cache line = 4 words
  - Implementation 2:
    1. $B(i) \leftarrow 0, 0 \leq i < n$
    2. for $i = 0$ to $n - 1$ do
    3. for $j = 0$ to $n - 1$ do
    4. $B(i) \leftarrow B(i) + A(i, j)$

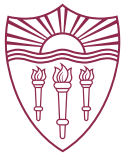
Row major access pattern



$$A(i, j) \rightarrow \text{Memory}(i \cdot n + j)$$
$$0 \leq i, j < n$$

  - High spatial locality
    - All 4 words of data accessed in each cycle are needed
    - Up to 4 times the performance as Implementation 1

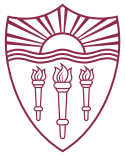
Data layout and data access pattern (5)



Note: can also do column major layout and use Implementation 1

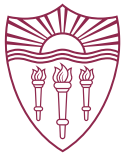
→ high performance

Memory Systems Performance Summary



- Sustained performance
  - Main memory: latency and bandwidth affect performance
  - Caches can be very effective: Improvement depends on how cache is effectively accessed/exploited (Cache friendly/Cache oblivious algorithms)
  - Hit ratio
  - Data Locality
    - spatial locality
    - temporal locality
  - Cache pollution
  - Data layout and program access pattern affect
    - Effective processor-memory bandwidth
    - Sustained performance (GFlops/sec)

STREAM benchmark for memory performance (1)

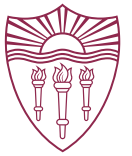


- A simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels
  - designed to be more indicative of the performance of very large, vector style applications

- It counts how many bytes the user asked to be read plus how many bytes the user asked to be written

http://www.cs.virginia.edu/stream/

STREAM benchmark for memory performance (2)

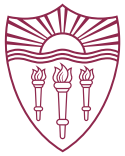


- The table below shows how many Bytes and FLOPs are counted in each iteration of the STREAM loops.

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

http://www.cs.virginia.edu/stream/

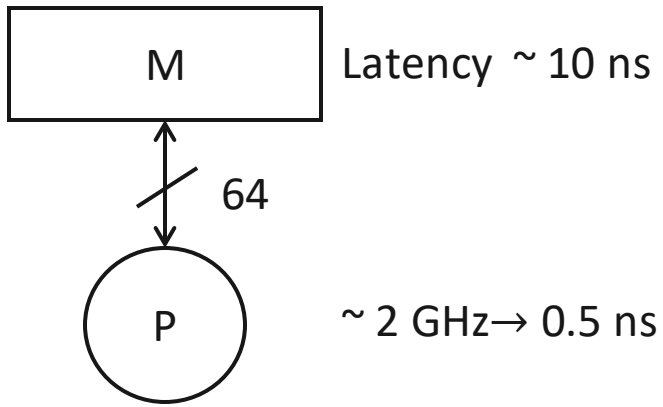
Modern CPU STREAM benchmark results



Processor	# of cores	COPY	SCALE	ADD	TRIAD
AMD EPYC 7601	64	283 GB/s	286 GB/s	292 GB/s	290 GB/s
Intel Xeon Platinum 8160	40	175 GB/s	150 GB/s	170 GB/s	175 GB/s
IBM POWER9	40	240 GB/s	250 GB/s	255 GB/s	260 GB/s

https://www.amd.com/system/files/2017-06/AMD-EPYC-SoC-Delivers-Exceptional-Results.pdf
https://www.mdpi.com/2079-9292/9/6/1035/pdf

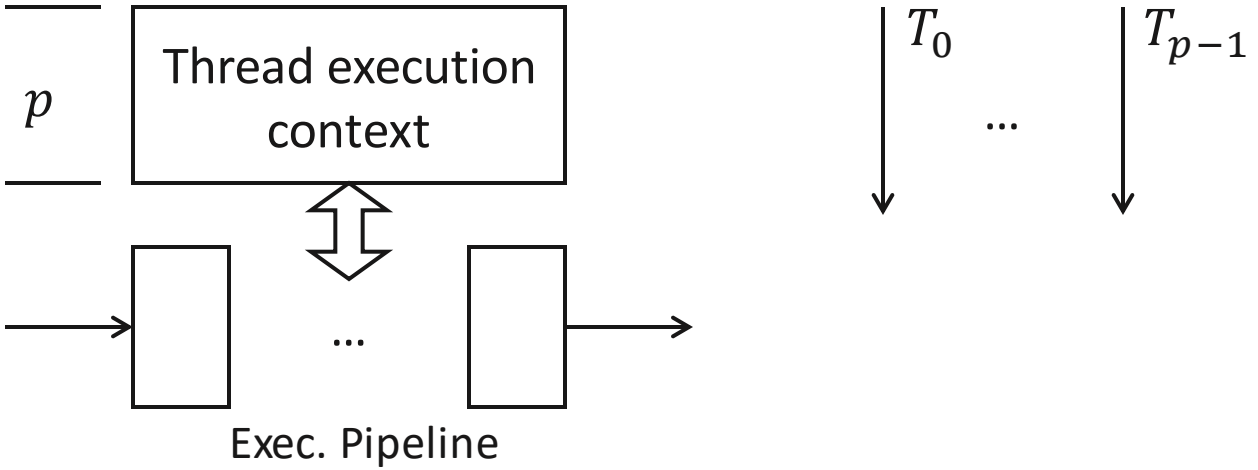
Architectural support for latency hiding (1)



no interactions

- **Program**: Collection of *independent sequence of instructions* → *thread*
- Architecture → switch between threads when there is access to main memory by a thread
  - Typically hardware supports single-cycle switch between threads

Architectural support for latency hiding (2)



- Example
  - Assume 20 cycles for DRAM access [0.5 ns processor, 10 ns DRAM]
  - Assume 1 cycle overhead for context switch
  - Single thread: can be blocked for 20 cycles
  - Multiple threads (e.g., $p \geq 10$): we can make sure the processor is kept busy executing some thread (i.e. doing useful work)

Multi-threading

Prefetching to hide memory latency

Example

1. for $i = 0\ to\ n - 1$ do
2. $C(i) \leftarrow A(i) + B(i)$

$A(0), B(0) \rightarrow$ **cache miss**

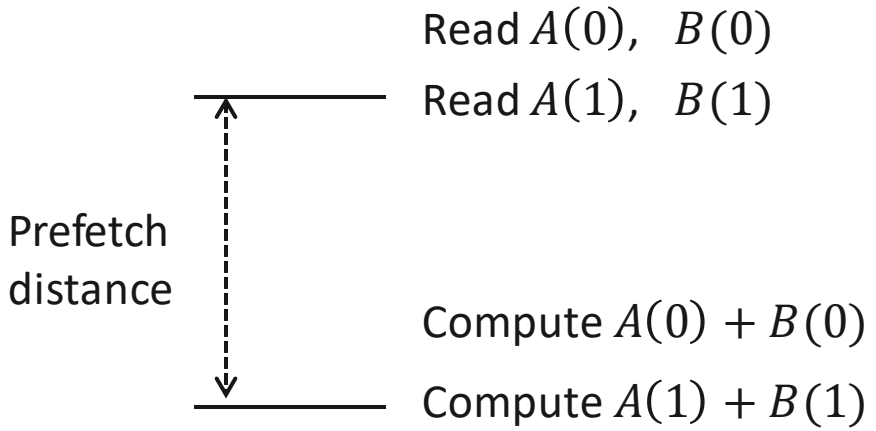
Issue request to read $A(0), B(0)$ in advance

Issue request to read $A(1), B(1)$ in advance

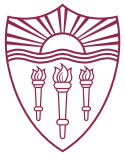
to mask 20-cycle memory latency

inserted by compiler

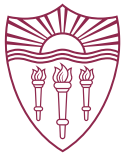
…



Summary

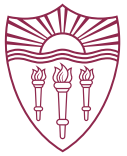


- Memory systems
  - Main memory latency, throughput
  - Cache
  - Data reuse
  - To achieve high performance (reduce execution time)
    - Latency hiding
    - Reduce processor – memory traffic (increase data reuse)



Backup Slides

Compute intensity example



$A \leftarrow B * C$ $n \times n$ matrix multiplication

Do $i = 1$ to n

Read i^{th} row of B

Do $j = 1$ to n

Read j^{th} col of C

Inner product $B(i,*), C(*,j)$

Compute intensity of row of $B = n$

Compute intensity of col of $C = 1$