More $ (caches, yes)

- Trick or treat!

- Midterm questions?
  - Note: practice midterms posted

- HW 2 due today

- Lab 3 will be released soon
  - You will implement a buffer overflow attack! Huahuahua! 😊
Deja-vu

```c
int array[SIZE];
int A = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
    for (int j = 0 ; j < SIZE ; ++ j) {
        A += array[j];
    }
}
```
Not to forget...

CPU

A little of super fast memory (cache$)

Lots of slower Mem
General Cache Mechanics

Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into “blocks”
General Cache Concepts: Hit

Request: 14

Data in block b is needed

Block b is in cache: Hit!
General Cache Concepts: Miss

Request: 12

Data in block b is needed

Block b is not in cache: Miss!

Oh no! What now?
General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache: Miss!

Block b is fetched from memory
General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache: Miss!

Block b is fetched from memory

Block b is stored in cache
• Placement policy: determines where b goes
• Replacement policy: determines which block gets evicted (victim)
Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses)
    \[= 1 - \text{hit rate}\]
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.

- **Hit Time**
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 1-2 clock cycle for L1
    - 5-20 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (trend: increasing!)
Lets think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- Would you believe 99% hits is twice as good as 97%?
  - Consider:
    - cache hit time of 1 cycle
    - miss penalty of 100 cycles
Let's think about those numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    - cache hit time of 1 cycle
    - miss penalty of 100 cycles

  - Average access time:
    - 97% hits: 1 cycle + 0.03 * 100 cycles = \(4\) cycles
    - 99% hits: 1 cycle + 0.01 * 100 cycles = \(2\) cycles

- **This is why “miss rate” is used instead of “hit rate”**
Why do caches work?
Why Caches Work

- **Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently
Why Caches Work

- **Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- **Temporal locality**: Recently referenced items are *likely* to be referenced again in the near future

  - Why is this important?
Why Caches Work

- **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

- **Temporal locality:** Recently referenced items are likely to be referenced again in the near future

- **Spatial locality?**
Why Caches Work

- **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

- **Temporal locality:**
  - Recently referenced items are *likely* to be referenced again in the near future

- **Spatial locality:**
  - Items with nearby addresses *tend* to be referenced close together in time

- How do caches take advantage of this?
Example: Locality?

```c
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```
Example: Locality?

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data:
  - Temporal: \texttt{sum} referenced in each iteration
  - Spatial: array \texttt{a[]} accessed in stride-1 pattern
Example: Locality?

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data:**
  - Temporal: `sum` referenced in each iteration
  - Spatial: array `a[ ]` accessed in stride-1 pattern

- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence
Example: Locality?

```java
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data:**
  - Temporal: `sum` referenced in each iteration
  - Spatial: array `a[]` accessed in stride-1 pattern

- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

**Being able to assess the locality of code is a crucial skill for a programmer**
Locality Example #1

```c
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```
Locality Example #1

```c
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
a[0][0]  a[0][1]  a[0][2]  a[0][3]
a[1][0]  a[1][1]  a[1][2]  a[1][3]
```

stride-1
Locality Example #2

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}

Locality Example #2

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}

// Example array:

<table>
<thead>
<tr>
<th></th>
<th>a[0][0]</th>
<th>a[0][1]</th>
<th>a[0][2]</th>
<th>a[0][3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a[1][0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a[1][1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a[1][2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a[1][3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a[2][0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a[2][1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a[2][2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>a[2][3]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

stride-N
```
Locality Example #3

```c
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```

- What is wrong with this code?
- How can it be fixed?
Important questions about $\$

- When we copy a block of data from main memory to the cache, where exactly should we put it?

- How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?

- Eventually, the small cache memory might fill up. To load a new block from main RAM, we’d have to replace one of the existing blocks in the cache... which one?

- How can write operations be handled by the memory system?
Where should we put data in the cache?

- A **direct-mapped** cache is the simplest approach: each main memory address maps to exactly one cache block.

- For example, on the right is a 16-byte main memory and a 4-byte cache (four 1-byte blocks).

- Memory locations 0, 4, 8 and 12 all map to cache block 0.

- Addresses 1, 5, 9 and 13 map to cache block 1, etc.

- How can we compute this mapping?
Ok, we know where to look for data..

- But how do we know if the data is what we want??
We need to add tags to the cache, which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block.
What’s a cache block? (or cache line)
Direct Mapped Caches

- Any problems with them?
Disadvantage of direct mapping

- The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, because each memory address belongs in exactly one block.
- But, what happens if a program uses addresses 2, 6, 2, 6, 2, ...?

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00</td>
</tr>
<tr>
<td>0001</td>
<td>01</td>
</tr>
<tr>
<td>0010</td>
<td>10</td>
</tr>
<tr>
<td>0011</td>
<td>11</td>
</tr>
<tr>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>
Associativity

- What if we could store data in *any* place in the cache?
Associativity

- What if we could store data in any place in the cache?
- But that might slow down caches... so we do something in between.

<table>
<thead>
<tr>
<th></th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 sets, 1 block each</td>
<td>4 sets, 2 blocks each</td>
<td>2 sets, 4 blocks each</td>
<td>1 set, 8 blocks</td>
</tr>
<tr>
<td>Set</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>direct mapped</td>
<td></td>
<td></td>
<td>fully associative</td>
</tr>
</tbody>
</table>
But now how do I know where data goes?

- m-bit Address
- (m-k-n) bits
- k bits
- (m-k-n) bits
- n-bit Block Offset
But now how do I know where data goes?

Our example used a $2^2$-block cache with $2^1$ bytes per block. Where would 13 (1101) be stored?
A puzzle.

- What can you infer from this:
  - Cache starts *empty*
  - Access (addr, hit/miss) stream
    - (10, miss), (11, hit), (12, miss)
General Cache Organization ($S, E, B$)

$E = 2^e$ lines per set

$S = 2^i$ sets

$B = 2^b$ bytes data block per cache line (the data)

Cache size:

$S \times E \times B$ data bytes
Cache Read

- $E = 2^e$ lines per set
- $S = 2^s$ sets
- $B = 2^b$ bytes data block per cache line (the data)

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Address of word:
- $t$ bits: tag
- $s$ bits: set index
- $b$ bits: block offset

Data begins at this offset

valid bit
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes

S = $2^5$ sets

Address of int:

<table>
<thead>
<tr>
<th>t bits</th>
<th>0...01</th>
<th>100</th>
</tr>
</thead>
</table>

find set
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes

Address of int:

Valid? + match: assume yes = hit

Block offset
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes

No match: old line is evicted and replaced
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

\[
\begin{array}{c|c|c}
\text{t bits} & 0 & 01 & 100 \\
\end{array}
\]
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

```
t bits 0...01 100
```

find set
E-way Set-Associative Cache (Here: $E = 2$)

$E = 2$: Two lines per set
Assume: cache block size 8 bytes

Valid? + match: yes = hit

Address of short int:

$0...01$  
$100$

Block offset
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

| t bits | 0...01 | 100 |

match both

valid? + match: yes = hit

short int (2 Bytes) is here

No match:
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...
Example placement in set-associative caches

Where would data from memory byte address 6195 be placed, assuming the eight-block cache designs below, with 16 bytes per block?

- 6195 in binary is 00...0110000 011 0011.
Example placement in set-associative caches

- Where would data from memory byte address 6195 be placed, assuming the eight-block cache designs below, with 16 bytes per block?
- 6195 in binary is 00...0110000 011 0011.
- Each block has 16 bytes, so the **lowest 4 bits are the block offset**.
- For the 1-way cache, the next three bits (011) are the set index.
  For the 2-way cache, the next two bits (11) are the set index.
  For the 4-way cache, the next one bit (1) is the set index.
- The data may go in *any* block, shown in green, within the correct set.

- **1-way associativity**
  - 8 sets, 1 block each
- **2-way associativity**
  - 4 sets, 2 blocks each
- **4-way associativity**
  - 2 sets, 4 blocks each
Block replacement

- Any empty block in the correct set may be used for storing data.
- If there are no empty blocks, which one should we replace?

1-way associativity
8 sets, 1 block each

2-way associativity
4 sets, 2 blocks each

4-way associativity
2 sets, 4 blocks each
Block replacement

- Any empty block in the correct set may be used for storing data.
- If there are no empty blocks, the cache controller will attempt to replace the least recently used block, just like before.
- For highly associative caches, it’s expensive to keep track of what’s really the least recently used block, so some approximations are used. We won’t get into the details.

1-way associativity
8 sets, 1 block each

2-way associativity
4 sets, 2 blocks each

4-way associativity
2 sets, 4 blocks each
Another puzzle.

- What can you infer from this:
  - Cache starts *empty*
  - Access (addr, hit/miss) stream
    - (10, miss); (12, miss); (10, miss)
Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block
Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block

- **Conflict miss**
  - Most hardware caches limit blocks to a small subset (sometimes just one) of the available cache slots
    - if one (e.g., block i must be placed in slot (i mod size)), **direct-mapped**
    - if more than one, n-way **set-associative** (where n is a power of 2)
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time
Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block

- **Conflict miss**
  - Most hardware caches limit blocks to a small subset (sometimes just one) of the available cache slots
    - if one (e.g., block i must be placed in slot (i mod size)), **direct-mapped**
    - if more than one, n-way **set-associative** (where n is a power of 2)
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time

- **Capacity miss**
  - Occurs when the set of active cache blocks (the **working set**) is larger than the cache (just won’t fit)
What about writes?

- **Multiple copies of data exist:**
  - L1, L2, Main Memory, Disk

- **What to do on a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - *How do we know when to write?*

- **What to do on a write-miss?**
  - Write-allocate (load into cache, then write)
    - *When is this useful?*
  - No-write-allocate (writes immediately to memory)

- **Typical**
  - Write-through + No-write-allocate
  - Write-back + Write-allocate
Memory Hierarchies

- Some fundamental and enduring properties of hardware and software systems:
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True for: registers ↔ cache, cache ↔ DRAM, DRAM ↔ disk, etc.
  - Well-written programs tend to exhibit good locality

- These properties complement each other beautifully

- They suggest an approach for organizing memory and storage systems known as a memory hierarchy
An Example Memory Hierarchy

- **L0:** registers
- **L1:** on-chip L1 cache (SRAM)
- **L2:** off-chip L2 cache (SRAM)
- **L3:** main memory (DRAM)
- **L4:** local secondary storage (local disks)
- **L5:** remote secondary storage (distributed file systems, web servers)

- CPU registers hold words retrieved from L1 cache
- L1 cache holds cache lines retrieved from L2 cache
- L2 cache holds cache lines retrieved from main memory
- Main memory holds disk blocks retrieved from local disks
- Local disks hold files retrieved from disks on remote network servers

Smaller, faster, costlier per byte

Larger, slower, cheaper per byte
Typical Memory Hierarchy (Intel Core i7)

- **L0:** CPU registers (optimized by compiler)
- **L1:** on-chip L1 cache (SRAM)  (8-way associative in Intel Core i7)
- **L2:** off-chip L2 cache (SRAM)  (8-way associative in Intel Core i7)
- **L3:** off-chip cache L3 shared by multiple cores (SRAM)  (16-way associative in Intel Core i7)
- **L4:** main memory (DRAM)
- **L5:** local secondary storage (local disks)
- **L6:** remote secondary storage (distributed file systems, web servers)

---

- Larger, slower, cheaper per byte
- Smaller, faster, costlier per byte
### Examples of Caching in the Hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What is Cached?</th>
<th>Where is it Cached?</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-byte words</td>
<td>CPU core</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-bytes block</td>
<td>On-Chip L1</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-bytes block</td>
<td>Off-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB page</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware+OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Network cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>File system client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web server</td>
</tr>
</tbody>
</table>
Memory Hierarchy: Core 2 Duo

L1/L2 cache: 64 B blocks

Throughput:
Latency:

CPU
Reg

L1
l-cache

32 KB
L1
D-cache

~4 MB
L2
unified
cache

~4 GB
Main
Memory

Disk

Not drawn to scale

Throughput:
Latency:

16 B/cycle
3 cycles

8 B/cycle
14 cycles

2 B/cycle
100 cycles

1 B/30 cycles
millions

~4 MB
32 KB
~4 GB
~500 GB
Where else is caching used?
Software Caches are More Flexible

Examples
- File system buffer caches, web browser caches, etc.

Some design differences
- Almost always fully-associative
  - so, no placement restrictions
  - index structures like hash tables are common (for placement)
- Often use complex replacement policies
  - misses are very expensive when disk or network involved
  - worth thousands of cycles to avoid them
- Not necessarily constrained to single “block” transfers
  - may fetch or write-back in larger units, opportunistically
Optimizations for the Memory Hierarchy

- Write code that has locality
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time

- How to achieve?
  - Proper choice of algorithm
  - Loop transformations

- Cache versus register-level optimization:
  - In both cases locality desirable
  - Register space much smaller
    + requires scalar replacement to exploit temporal locality
  - Register level optimizations include exhibiting instruction level parallelism (conflicts with locality)
Example: Matrix Multiplication

```c
// Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k]*b[k*n + j];
}
```

\[
c = (double *) \text{calloc} \left( \text{sizeof(double)}, n*n\right);
\]

\[
\begin{array}{c}
\text{c} = (\text{double } *) \text{calloc} \left( \text{sizeof(double)}, n*n\right); \\
\end{array}
\]
Cache Miss Analysis

Assume:
- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than $n$)

First iteration:
- $n/8 + n = 9n/8$ misses (omitting matrix $c$)
- Afterwards in cache: (schematic)
Cache Miss Analysis

Assume:
- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size $C << n$ (much smaller than $n$)

Other iterations:
- Again:
  \[ \frac{n}{8} + n = \frac{9n}{8} \text{ misses} \]
  (omitting matrix $c$)

Total misses:
- \[ 9n/8 \times n^2 = (9/8) \times n^3 \]
```c
double *c = calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k,
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n + j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

Block size B x B
Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size \( C \ll n \) (much smaller than \( n \))
  - Four blocks fit into cache: \( 4B^2 < C \)

- First (block) iteration:
  - \( B^2/8 \) misses for each block
  - \( 2n/B \times B^2/8 = nB/4 \) (omitting matrix \( c \))

- Afterwards in cache (schematic)
Cache Miss Analysis

Assume:
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than $n$)
- Three blocks fit into cache: $3B^2 < C$

Other (block) iterations:
- Same as first iteration
- $2n/B \times B^2/8 = nB/4$

Total misses:
- $nB/4 \times (n/B)^2 = n^3/(4B)$
Summary

- No blocking: \( (9/8) \times n^3 \)
- Blocking: \( 1/(4B) \times n^3 \)

If \( B = 8 \) difference is \( 4 \times 8 \times 9 / 8 = 36x \)
If \( B = 16 \) difference is \( 4 \times 16 \times 9 / 8 = 72x \)

- Suggests largest possible block size \( B \), but limit \( 4B^2 < C \)!
  (can possibly be relaxed a bit, but there is a limit for \( B \))

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: \( 3n^2 \), computation \( 2n^3 \)
    - Every array elements used \( O(n) \) times!
  - But program has to be written properly