What You Must Know about Memory, Caches, and Shared Memory

Kenjiro Taura
1. Introduction
2. Organization of processors, caches, and memory
3. Caches
4. So how costly is it to access data?
   - Latency
   - Bandwidth
   - More bandwidth = concurrent accesses
5. Many algorithms are bounded by memory not CPU
6. Other ways to get more bandwidth
   - Make addresses sequential
   - Make address generations independent
   - Prefetch by software (make address generations go ahead)
   - Use multiple threads/cores
7. How costly is it to communicate between threads?
Introduction

Organization of processors, caches, and memory

Caches

So how costly is it to access data?
- Latency
- Bandwidth
- More bandwidth = concurrent accesses

Many algorithms are bounded by memory not CPU

Other ways to get more bandwidth
- Make addresses sequential
- Make address generations independent
- Prefetch by software (make address generations go ahead)
- Use multiple threads/cores

How costly is it to communicate between threads?
so far, we have learned
- parallelization across cores,
- vectorization (SIMD) within a core, and
- instruction level parallelism

another critical factor you must know to understand program performance is \textit{data access}
Why data access is so important?

- accessing data is sometimes far more costly than calculation
Why data access is so important?

- accessing data is sometimes far more costly than calculation
- moreover, data access cost significantly differs depending on where data are coming from
  - registers
  - caches
  - main memory
  - another processor’s cache
Conceptual goals of the study

- understand how are processors, caches, and memory connected
- understand how to reason about cache hits/misses of a program
- \( \Rightarrow \) be able to reason about a performance limit of your program, due to memory access
Pragmatic goals of the study

- **latency**: get a sense of how many cycles it takes to get data from main memory, L3 caches, L2 caches, L1 caches.
Pragmatic goals of the study

- **latency**: get a sense of how many cycles it takes to get data from main memory, L3 caches, L2 caches, L1 caches
- **bandwidth**: get a sense of how much data CPU can bring from main memory, L3 caches, L2 caches, L1 caches
Pragmatic goals of the study

- **latency**: get a sense of how many cycles it takes to get data from main memory, L3 caches, L2 caches, L1 caches
- **bandwidth**: get a sense of how much data CPU can bring from main memory, L3 caches, L2 caches, L1 caches
- what does “memory bandwidth” we see in a processor spec sheet really mean? e.g.,
  - this page (by Intel)
    
    
    says its max memory bandwidth is 68 GB/s
Pragmatic goals of the study

- **latency**: get a sense of how many cycles it takes to get data from main memory, L3 caches, L2 caches, L1 caches

- **bandwidth**: get a sense of how much data CPU can bring from main memory, L3 caches, L2 caches, L1 caches

- what does “memory bandwidth” we see in a processor spec sheet really mean? e.g.,
  - this page (by Intel)
    
    
    says its max memory bandwidth is 68 GB/s

- how can we achieve this max memory bandwidth?
A simple `memcpy` experiment . . .

double t0 = cur_time();
memcpy(a, b, nb);
double t1 = cur_time();

$ gcc -O3 memcpy.c$
$ ./a.out $((1 << 26)) # 64M long elements = 512MB

536870912 bytes copied in 0.117333 sec

4.575611 GB/sec

much lower than the advertised number . . .
A simple \texttt{memcpy} experiment ...

double t0 = cur_time();
memcpy(a, b, nb);
double t1 = cur_time();

\$ gcc -O3 memcpy.c
\$ ./a.out $((1 \ll 26)) \ # 64M long elements = 512MB
536870912 bytes copied in 0.117333 sec \textbf{4.575611} GB/sec
A simple `memcpy` experiment . . .

```c
1 double t0 = cur_time();
2 memcpy(a, b, nb);
3 double t1 = cur_time();
```

```shell
$ gcc -O3 memcpy.c
$ ./a.out $((1 << 26)) # 64M long elements = 512MB
536870912 bytes copied in 0.117333 sec 4.575611 GB/sec
```

- much lower than the advertised number . . .
Contents

1 Introduction
2 Organization of processors, caches, and memory
3 Caches
4 So how costly is it to access data?
   - Latency
   - Bandwidth
   - More bandwidth = concurrent accesses
5 Many algorithms are bounded by memory not CPU
6 Other ways to get more bandwidth
   - Make addresses sequential
   - Make address generations independent
   - Prefetch by software (make address generations go ahead)
   - Use multiple threads/cores
7 How costly is it to communicate between threads?
you almost certainly know this (*caches* and main memory), don’t you?
... with multi level caches, ...

recent processors have *multiple levels* of caches (L1, L2, ...)

![Diagram of multi-level cache system]

- **(physical) core**
- **L1 cache**
- **L2 cache**
- **multi-level caches**
... with multicores in a chip, ...

- A single chip has several cores.
- Each core has its private caches (typically, L1 and L2).
- Cores in a chip share a cache (typical, L3) and main memory.
..., with simultaneous multithreading (SMT) in a core, ...

- each core has two **hardware threads**, which share L1/L2 caches and some or all execution units
..., and with multiple sockets per node.

- each node has several chips (sockets), connected via an interconnect (e.g., Intel QuickPath, AMD HyperTransport, etc.)
- each socket serves a part of the entire main memory
- each core can still access any part of the entire main memory
Today’s typical single compute node

Typical cache sizes

- L1: 16KB - 64KB/core
- L2: 256KB - 1MB/core
- L3: ~ 50MB/socket
1. Introduction
2. Organization of processors, caches, and memory
3. Caches
4. So how costly is it to access data?
   - Latency
   - Bandwidth
   - More bandwidth = concurrent accesses
5. Many algorithms are bounded by memory not CPU
6. Other ways to get more bandwidth
   - Make addresses sequential
   - Make address generations independent
   - Prefetch by software (make address generations go ahead)
   - Use multiple threads/cores
7. How costly is it to communicate between threads?
ABC’s of caches

- speed:
  \[
  L_1 > L_2 > L_3 > \text{main memory}
  \]
ABC’s of caches

- **speed:**
  \[ L1 > L2 > L3 > \text{main memory} \]

- **capacity:**
  \[ L1 < L2 < L3 < \text{main memory} \]
ABC’s of caches

- speed:
  \[ L_1 > L_2 > L_3 > \text{main memory} \]

- capacity:
  \[ L_1 < L_2 < L_3 < \text{main memory} \]

- each cache holds a subset of data in the main memory
  \[ L_1, L_2, L_3 \subset \text{main memory} \]
ABC’s of caches

- speed:
  \[L_1 > L_2 > L_3 > \text{main memory}\]

- capacity:
  \[L_1 < L_2 < L_3 < \text{main memory}\]

- each cache holds a subset of data in the main memory
  \[L_1, L_2, L_3 \subseteq \text{main memory}\]

- typically but not necessarily,
  \[L_1 \subset L_2 \subset L_3 \subset \text{main memory}\]
ABC’s of caches

- speed:
  \[ L_1 > L_2 > L_3 > \text{main memory} \]

- capacity:
  \[ L_1 < L_2 < L_3 < \text{main memory} \]

- each cache holds a subset of data in the main memory
  \[ L_1, L_2, L_3 \subset \text{main memory} \]

- typically but not necessarily,
  \[ L_1 \subset L_2 \subset L_3 \subset \text{main memory} \]

- which subset is in caches? \(\rightarrow\) cache management (replacement) policy
a cache generally holds data in *the most recently accessed* distinct addresses, up to its capacity
Cache management (replacement) policy

- a cache generally holds data in *the most recently accessed* distinct addresses, up to its capacity
- this is accomplished by the **LRU replacement** policy:
  - every time a load/store instruction misses a cache, the least recently used data in the cache will be replaced
Cache management (replacement) policy

- a cache generally holds data in the *most recently accessed* distinct addresses, up to its capacity
- this is accomplished by the LRU replacement policy:
  - every time a load/store instruction misses a cache, the least recently used data in the cache will be replaced
- ⇒ a (very crude) approximation; data in 32KB L1 cache

  \[ \approx \text{most recently accessed 32768 distinct addresses} \]
Cache management (replacement) policy

- a cache generally holds data in *the most recently accessed* distinct addresses, up to its capacity.
- this is accomplished by the **LRU replacement** policy:
  - every time a load/store instruction misses a cache, the least recently used data in the cache will be replaced.
- $\Rightarrow$ a (very crude) approximation; data in 32KB L1 cache

  $\approx$ *most recently accessed 32768 distinct addresses*

- due to implementation constraints, real caches are slightly more complex.
Cache organization: cache line

- A cache = a set of fixed size lines
- Typical line size = 64 bytes or 128 bytes,
- A 32KB cache with 64 bytes lines (holds most recently accessed 512 distinct blocks)
Cache organization: cache line

- A cache = a set of fixed size *lines*
  - Typical line size = 64 bytes or 128 bytes,
- A single line is the minimum unit of data transfer between levels (and replacement)

A 32KB cache with 64 bytes lines (holds most recently accessed 512 distinct blocks)
Cache organization: cache line

- A cache = a set of fixed size lines
  - Typical line size = 64 bytes or 128 bytes,
- A single line is the minimum unit of data transfer between levels (and replacement)

A 32KB cache with 64 bytes lines (holds most recently accessed 512 distinct blocks)

Data in 32KB L1 cache (line size 64B)

≈ *most recently accessed 512 distinct lines*
**Associativity of caches**

**full associative:** a block can occupy any line in the cache, regardless of its address

**direct map:** a block has only *one* designated “seat” (*set*), determined by its address

**K-way set associative:** a block has *K* designated “seats”, determined by its address

- direct map $\equiv$ 1-way set associative
- full associative $\equiv$ $\infty$-way set associative
An example cache organization

Haswell E5-2686

<table>
<thead>
<tr>
<th>level</th>
<th>line size</th>
<th>capacity</th>
<th>associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>64B</td>
<td>32KB/core</td>
<td>8</td>
</tr>
<tr>
<td>L2</td>
<td>64B</td>
<td>256KB/core</td>
<td>8</td>
</tr>
<tr>
<td>L3</td>
<td>64B</td>
<td>46MB/socket</td>
<td>20</td>
</tr>
</tbody>
</table>
What you want to remember about associativity

- **Avoid frequently used addresses or addresses used together “a-large-power-of-two” bytes apart;** corollary:
  - Avoid having a matrix with a-large-power-of-two number of columns (a common mistake)
  - Avoid managing your memory by chunks of large-powers-of-two bytes (a common mistake)
  - Avoid experiments only with \( n = 2^p \) (a *very* common mistake)

- Why? ⇒ they tend to go to the same set and “conflict misses” result
Conflict misses

- consider 8-way set associative L2 cache with 256KB (line size = 64B)
  - $256\text{KB}/64\text{B} = 4\text{K} = 2^{12}$ lines
  - $4\text{K}/8 = 512 = 2^9$ sets

- $\Rightarrow$ given an address $a$, $a[6:14]$ (9 bits) designates the set it belongs to (indexing)

- if two addresses $a$ and $b$ are a multiple of $2^{15}$ (32KB) bytes apart, they go to the same set
Conflict misses

- e.g., if you have a matrix:

\[
\text{float } a[100][8192]\;;
\]

then \(a[i][j] \text{ and } a[i+1][j]\) go to the same set;

\(\Rightarrow\) scanning a column of such a matrix will experience almost 100\% cache miss

- a remedy is as simple as:

\[
\text{float } a[100][8192+16]\;;
\]
What are in the cache?

consider $K$-way set associative cache with capacity $= C$ bytes and line size $= Z$ bytes

- approximation 0.0 (only consider $C$; $\equiv Z = 1, K = \infty$):
  
  Cache $\approx$ most recently accessed $C$ distinct addresses
What are in the cache?

consider $K$-way set associative cache with capacity $= C$ bytes and line size $= Z$ bytes

1. **approximation 0.0** (only consider $C$; $\equiv Z = 1, K = \infty$):
   
   Cache $\approx$ most recently accessed $C$ distinct addresses

2. **approximation 1.0** (only consider $C$ and $Z$; $K = \infty$):
   
   Cache $\approx$ most recently accessed $C/Z$ distinct lines

more pragmatically, if you typically access data larger than cache line granularity (i.e., when you touch an element, you almost certainly touch the surrounding $Z$ bytes), forget $Z$; otherwise cache $\approx$ most recently accessed $C/Z$ elements
What are in the cache?

consider $K$-way set associative cache with capacity = $C$ bytes
and line size = $Z$ bytes

- **approximation 0.0** (only consider $C$; $\equiv Z = 1$, $K = \infty$):
  
  Cache $\approx$ most recently accessed $C$ distinct addresses

- **approximation 1.0** (only consider $C$ and $Z$; $K = \infty$):
  
  Cache $\approx$ most recently accessed $C/Z$ distinct lines

more pragmatically, if you typically access data larger than cache line granularity (i.e., when you touch an element, you almost certainly touch the surrounding $Z$ bytes), forget $Z$;
otherwise cache $\approx$ most recently accessed $C/Z$ elements

- **approximation 2.0**:
  
  - large associativities of recent caches alleviate the need to worry too much about it
  
  pragmatically, avoid conflicts I mentioned
So how costly is it to access data?

- Latency
- Bandwidth
- More bandwidth = concurrent accesses

Many algorithms are bounded by memory not CPU

Other ways to get more bandwidth

- Make addresses sequential
- Make address generations independent
- Prefetch by software (make address generations go ahead)
- Use multiple threads/cores

How costly is it to communicate between threads?
Assessing the cost of data access

- We like to obtain cost to access data in each level of the caches as well as main memory.
- **Latency**: time until the result of a load instruction becomes available.
- **Bandwidth**: the maximum amount of data per unit time that can be transferred between the layer in question to CPU (registers).
Contents

1 Introduction
2 Organization of processors, caches, and memory
3 Caches
4 So how costly is it to access data?
   - Latency
   - Bandwidth
   - More bandwidth = concurrent accesses
5 Many algorithms are bounded by memory not CPU
6 Other ways to get more bandwidth
   - Make addresses sequential
   - Make address generations independent
   - Prefetch by software (make address generations go ahead)
   - Use multiple threads/cores
7 How costly is it to communicate between threads?
How to measure a latency?

- prepare an array of $N$ records and access them repeatedly

```c
for (N times) {
    p = p->next;
}
```
How to measure a latency?

- prepare an array of $N$ records and access them repeatedly
- to measure the latency, make sure $N$ load instructions make a chain of dependencies (link list traversal)

```
for (N times) {
    p = p->next;
}
```
How to measure a latency?

- prepare an array of $N$ records and access them repeatedly
- to measure the *latency*, make sure $N$ load instructions *make a chain of dependencies* (link list traversal)

```c
1  for (N times) {
2    p = p->next;
3  }
```

- make sure $p->next$ links all the elements in a random order (the reason becomes clear later)
Data size vs. latency

- main memory is local to the accessing thread

```
numactl --cpunodebind 0 --interleave 0 ./traverse
```

Latency per load in a list traversal (local) [$\geq 0$]
How long are latencies

- heavily depends on in which level of the cache data fit
- environment: Haswell E5-2686 (32KB/256KB/46MB)

<table>
<thead>
<tr>
<th>size</th>
<th>level</th>
<th>latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12,736</td>
<td>L1</td>
<td>4.02</td>
</tr>
<tr>
<td>101,312</td>
<td>L2</td>
<td>9.82</td>
</tr>
<tr>
<td>1,047,232</td>
<td>L3</td>
<td>50.35</td>
</tr>
<tr>
<td>10,251,712</td>
<td>L3</td>
<td>47.59</td>
</tr>
<tr>
<td>208,779,968</td>
<td>main</td>
<td>208.417</td>
</tr>
</tbody>
</table>
A remark about replacement policy

- if a cache strictly follows the LRU replacement policy, once data overflow the cache, repeated access to the data will quickly become *almost-always-miss*
- according to this theory, we would expect much sharper “cliffs” when data overflow each level of the cache
- the experimental data suggest something other than LRU is going on (c.f. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/)

![Graph showing cache miss rate and size to repeatedly scan](image)
A remark about replacement policy

- if a cache strictly follows the LRU replacement policy, once data overflow the cache, repeated access to the data will quickly become *almost-always-miss*
- according to this theory, we would expect much sharper “cliffs” when data overflow each level of the cache
- the experimental data suggest something other than LRU is going on (c.f. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/)
A remark about replacement policy

- If a cache strictly follows the LRU replacement policy, once data overflow the cache, repeated access to the data will quickly become *almost-always-miss*.
- According to this theory, we would expect much sharper “cliffs” when data overflow each level of the cache.
- The experimental data suggest something other than LRU is going on (c.f. [http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/](http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/)).
Latency when main memory is remote

- make main memory remote to the accessing thread

```bash
numactl --cpunodebind 0 --interleave 1 ./traverse
```

Latency per load in a list traversal (local and remote) \([\geq 0]\)

![Graph showing latency per load vs. size of the region (bytes)]
So how costly is it to access data?

- Latency
- Bandwidth
- More bandwidth = concurrent accesses

Many algorithms are bounded by memory not CPU

Other ways to get more bandwidth

- Make addresses sequential
- Make address generations independent
- Prefetch by software (make address generations go ahead)
- Use multiple threads/cores

How costly is it to communicate between threads?
Bandwidth of a random link list traversal

\[ \text{bandwidth} = \frac{\text{total bytes read}}{\text{elapsed time}} \]

- in this experiment, we set record size = 64

![Graph showing bandwidth versus size of the region (bytes).](image)
Zooming into the “main memory”

much lower than the `memcpy` bandwidth we have seen (4.5 GB/s)
not to mention the “memory bandwidth” in the processor spec (68 GB/s)
Why is the bandwidth so low?

- while traversing a single link list, only a single record access (64 bytes) is “in flight” at a time

- in this condition,

\[
\text{bandwidth} = \frac{\text{a record size}}{\text{latency}}
\]

- assuming frequency = 2.0GHz,

\[
\approx \frac{64 \text{ bytes}}{200 \text{ cycles}} = 0.32 \text{ bytes/cycle} \approx 0.64 \text{ GB/s}
\]
How to get more bandwidth?

- just like flops/clock, the only way to get a better throughput (bandwidth) is to perform *many load operations concurrently*
How to get more bandwidth?

- just like flops/clock, the only way to get a better throughput (bandwidth) is to perform *many load operations concurrently*

- there are several ways to make it happen, but let’s look at conceptually the simplest:
How to get more bandwidth?

- just like flops/clock, the only way to get a better throughput (bandwidth) is to perform *many load operations concurrently*.

- there are several ways to make it happen, but let’s look at conceptually the simplest:
- in our example, it happens when traversing multiple link lists.

```
for (N times) {
    p1 = p1->next;
    p2 = p2->next;
    ... 
}
```
Contents

1 Introduction
2 Organization of processors, caches, and memory
3 Caches
4 So how costly is it to access data?
   - Latency
   - Bandwidth
   - More bandwidth = concurrent accesses
5 Many algorithms are bounded by memory not CPU
6 Other ways to get more bandwidth
   - Make addresses sequential
   - Make address generations independent
   - Prefetch by software (make address generations go ahead)
   - Use multiple threads/cores
7 How costly is it to communicate between threads?
The number of lists vs. bandwidth

let’s zoom into “main memory” regime (size > 100MB)
Bandwidth to local main memory (not cache)

- an almost proportional improvement up to 10 lists

![Graph showing bandwidth with a number of chains.]

- Bandwidth (GB/sec) vs. size of the region (bytes).
Bandwidth to remote main memory (not cache)

- pattern is the same (improve up to 10 lists)
- remember the remote latency is longer, so the bandwidth is accordingly lower

![Graph showing bandwidth with a number of chains [remote, ≥ 100000000]](image)
The number of lists vs. bandwidth

- **observation**: bandwidth increase fairly proportionally to the number of lists, matching our understanding, . . .
The number of lists vs. bandwidth

- **observation:** bandwidth increase fairly proportionally to the number of lists, matching our understanding, . . .

- **question:** . . . but up to $\approx 10$, why?
The number of lists vs. bandwidth

- **observation**: bandwidth increase fairly proportionally to the number of lists, matching our understanding, . . .

- **question**: . . . but up to $\approx 10$, why?

- **answer**: there is a limit in the number of load operations in flight at a time
Line Fine Fill Buffer

- **Line fill buffer (LFB)** is the processor resource that keeps track of outstanding cache misses, and its size is 10 in Haswell.
Line Fine Fill Buffer

- *Line fill buffer (LFB)* is the processor resource that keeps track of outstanding cache misses, and its size is 10 in Haswell.
- This gives the maximum attainable bandwidth per core:

\[
\text{cache line size} \times \text{LFB size} \quad \text{latency}
\]
Line Fine Fill Buffer

- **Line fill buffer (LFB)** is the processor resource that keeps track of outstanding cache misses, and its size is 10 in Haswell.
- This gives *the maximum attainable bandwidth per core*

\[
\text{cache line size} \times \text{LFB size} \over \text{latency}
\]

- This is what we’ve seen (still much lower than what we see in the “memory bandwidth” in the spec sheet).
Line Fine Fill Buffer

- Line fill buffer (LFB) is the processor resource that keeps track of outstanding cache misses, and its size is 10 in Haswell.
- This gives the maximum attainable bandwidth per core:
  \[
  \text{cache line size} \times \text{LFB size} \over \text{latency}
  \]
- This is what we’ve seen (still much lower than what we see in the “memory bandwidth” in the spec sheet).
- How can we go beyond this? \( \Rightarrow \) the only way is to use multiple cores (covered later).
Introduction

Organization of processors, caches, and memory

Caches

So how costly is it to access data?
- Latency
- Bandwidth
- More bandwidth = concurrent accesses

Many algorithms are bounded by memory not CPU

Other ways to get more bandwidth
- Make addresses sequential
- Make address generations independent
- Prefetch by software (make address generations go ahead)
- Use multiple threads/cores

How costly is it to communicate between threads?
What do these numbers imply to FLOPS?

- many computationally efficient algorithms do not touch the same data too many times
What do these numbers imply to FLOPS?

- many computationally efficient algorithms do not touch the same data too many times
- e.g., $O(n)$ algorithms → touches a single element only a constant number of times
What do these numbers imply to FLOPS?

- many computationally *efficient* algorithms do not touch the same data too many times
- e.g., $O(n)$ algorithms → touches a single element only a constant number of times
- if data > cache for such an algorithm, the algorithm’s performance is often limited by memory bandwidth (or, worse, latency), *not CPU*
Example: matrix-vector multiply

- compute $Ax$ ($A: M \times N$ matrix; $x: N$-vector; 4 bytes/element)

```c
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    y[i] += a[i][j] * x[j];
```

This computes $Ax$ with $M \times N$ operations and $4MN$ bytes of data movement. In fact, it touches each matrix element only once! To sustain Haswell’s CPU peak (e.g., 16 multiply-adds per cycle), a core must access 16 matrix elements (= 64 bytes) per cycle, all from memory (unless they are not in the cache before the algorithm starts). Assuming a 2.0GHz processor, it requires memory bandwidth of:

$$64 \times 2 \times 1000 \text{GHz} = 128 \text{ GB/s}$$

per core, or

greater than it provides.
Example: matrix-vector multiply

compute \(Ax\) (\(A: M \times N\) matrix; \(x: N\)-vector; 4 bytes/element)

\[
\begin{align*}
1 & \text{for (i = 0; i < M; i++)} \\
2 & \quad \text{for (j = 0; j < N; j++)} \\
3 & \quad y[i] += a[i][j] * x[j];
\end{align*}
\]

2\(MN\) flops, 4\(MN\) bytes (ignore \(x\))
  
  in fact, it touches \textit{each matrix element only once}!
Example: matrix-vector multiply

- compute $Ax$ ($A : M \times N$ matrix; $x : N$-vector; 4 bytes/element)

```c
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    y[i] += a[i][j] * x[j];
```

- $2MN$ flops, $4MN$ bytes (ignore $x$)
  - in fact, it touches each matrix element only once!
- to sustain Haswell’s CPU peak (e.g., 16 multiply-adds per cycle), a core must access 16 matrix elements (= 64 bytes) per cycle, all from memory (unless they are not in the cache before the algorithm starts)
Example: matrix-vector multiply

- compute $Ax$ ($A: M \times N$ matrix; $x: N$-vector; 4 bytes/element)

```c
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    y[i] += a[i][j] * x[j];
```

- $2MN$ flops, $4MN$ bytes (ignore $x$)
  - in fact, it touches each matrix element only once!

- to sustain Haswell’s CPU peak (e.g., 16 multiply-adds per cycle), a core must access 16 matrix elements ($= 64$ bytes) per cycle, all from memory (unless they are not in the cache before the algorithm starts)

- assuming 2.0GHz processor, it requires memory bandwidth of:

  $$\approx 64 \times 2.0 \text{ GHz} = 128 \text{ GB/s}$$

  per core, or $\approx 20 \times$ more than it provides
Note about matrix-matrix multiply

- the argument does *not* apply to matrix-matrix multiply (we’ve been trying to get close to CPU peak)
Note about matrix-matrix multiply

- the argument does *not* apply to matrix-matrix multiply (we’ve been trying to get close to CPU peak)
- $2N^3$ flops, $12N^2$ bytes (for square matrices)
Note about matrix-matrix multiply

- the argument does not apply to matrix-matrix multiply (we’ve been trying to get close to CPU peak)
- $2N^3$ flops, $12N^2$ bytes (for square matrices)
- any straightforward algorithm uses a single element $O(N)$ times, so it may be possible to design a clever algorithm that
  - brings an element into a cache, and
  - uses that element many times before it’s evicted
Note about matrix-matrix multiply

- the argument does \textit{not} apply to matrix-matrix multiply (we’ve been trying to get close to CPU peak)
- \(2N^3\) flops, \(12N^2\) bytes (for square matrices)
- any straightforward algorithm uses a single element \(O(N)\) times, so it may be possible to design a clever algorithm that
  - brings an element into a cache, and
  - uses that element many times before it’s evicted
- I don’t mean this does not happen automatically for any algorithm; \textit{the order of computation is important}
Contents

1 Introduction
2 Organization of processors, caches, and memory
3 Caches
4 So how costly is it to access data?
   - Latency
   - Bandwidth
   - More bandwidth = concurrent accesses
5 Many algorithms are bounded by memory not CPU
6 Other ways to get more bandwidth
   - Make addresses sequential
   - Make address generations independent
   - Prefetch by software (make address generations go ahead)
   - Use multiple threads/cores
7 How costly is it to communicate between threads?
Other ways to get more bandwidth

we’ve learned:

- maximum bandwidth $\approx$ as many memory accesses as possible always in flight
- there is a limit due to LFB entries (10 in Haswell)
Other ways to get more bandwidth

- we’ve learned:
  - maximum bandwidth \(\approx\) as many memory accesses as possible always in flight
  - there is a limit due to LFB entries (10 in Haswell)
- so far, we have achieved larger bandwidth by traversing multiple lists explicitly (sometimes difficult if not impossible to apply)
Other ways to get more bandwidth

- we’ve learned:
  - maximum bandwidth $\approx$ as many memory accesses as possible always in flight
  - there is a limit due to LFB entries (10 in Haswell)
- so far, we have achieved larger bandwidth by traversing multiple lists explicitly (sometimes difficult if not impossible to apply)
- fortunately, the life is not always that tough; there are other ways to issue many memory accesses concurrently
  1. make addresses sequential
  2. make address generations independent
  3. prefetch by software (make address generations go ahead)
  4. use multiple threads/cores
Other ways to get more bandwidth

- we’ve learned:
  - maximum bandwidth \( \approx \) as many memory accesses as possible always in flight
  - there is a limit due to LFB entries (10 in Haswell)
- so far, we have achieved larger bandwidth by traversing multiple lists explicitly (sometimes difficult if not impossible to apply)
- fortunately, the life is not always that tough; there are other ways to issue many memory accesses concurrently
  1. make addresses sequential
  2. make address generations independent
  3. prefetch by software (make address generations go ahead)
  4. use multiple threads/cores
- remember, all boil down to keep as many memory accesses as possible (up to LFB entries) in flight
Contents

1. Introduction
2. Organization of processors, caches, and memory
3. Caches
4. So how costly is it to access data?
   - Latency
   - Bandwidth
   - More bandwidth = concurrent accesses
5. Many algorithms are bounded by memory not CPU
6. Other ways to get more bandwidth
   - Make addresses sequential
   - Make address generations independent
   - Prefetch by software (make address generations go ahead)
   - Use multiple threads/cores
7. How costly is it to communicate between threads?
Make addresses sequential

- again build a (single) linked list, but this time, `p->next` always points to the immediately following block
- note that the instruction sequence is identical to before; only addresses differ

![Diagram showing sequential linked list with cache line size and N elements]

- next pointers
  - (link all elements in the sequential order)
- cache line size
- $N$ elements
Bandwidth of traversing address-ordered list

- a factor of 10 faster than random case, but this time with only a single list
The reason this is faster

- **hardware prefetcher**
- CPU watches the sequence of addresses accessed
- sequential addresses (addresses of a small constant stride) trigger CPU’s hardware prefetcher
- CPU issues load instruction ahead of actual data stream on your behalf, to keep the maximum number of loads in flight
Contents

1 Introduction
2 Organization of processors, caches, and memory
3 Caches
4 So how costly is it to access data?
   - Latency
   - Bandwidth
   - More bandwidth = concurrent accesses
5 Many algorithms are bounded by memory not CPU
6 Other ways to get more bandwidth
   - Make addresses sequential
   - Make address generations independent
   - Prefetch by software (make address generations go ahead)
   - Use multiple threads/cores
7 How costly is it to communicate between threads?
Make address generations independent

- if addresses of memory accesses can be computed without values returned from previous loads, CPU can issue them concurrently

```cpp
for (N times) {
    j = ... /* not use a[] */
    a[j];
}
```

- note: it’s not a prefetch (but a real fetch)
Bandwidth when not traversing a list

- **ptrchase**: chase pointers of a random list
- **random**: access random addresses, but w/o pointer chasing
- **sequential**: access sequential addresses, w/o pointer chasing

The graph shows the bandwidth of random list traversal vs random array traversal for different sizes of the region (in bytes). The data points are color-coded for each traversal type:

- **ptrchase** (purple)
- **random** (green)
- **sequential** (blue)

The bandwidth values range from 0 (on the y-axis) to 160 GB/sec, and the size of the region ranges from 10000 to 1 x 10^9 bytes on the x-axis.
Main memory bandwidth

- pointer chase ≪ random < sequential
- random is as fast as chasing 10 lists

bandwidth of random list traversal vs random array traversal \([\geq 100000000]\)
Main memory bandwidth (random vs. sequential)

- sequential gets $\approx 2x$ more bandwidth than random
- may not be as bad as you thought?
- but why is there any difference, if both have the same number of loads in flight?

![Graph showing bandwidth of random list traversal vs random array traversal with ptrchase, random, and sequential operations.](image-url)
When random access is really bad

- in practice, when random vs. sequential makes a large ($\gg 2$) difference, it’s because
  
  a single element < a single cache line

- recall that touching a single byte in a cache line still brings the whole line (64 bytes)

- e.g., if you access an array of `float` (4 bytes) randomly, the bandwidth of *useful* data is amplified by a factor of 16 ($= 64/4$)
So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses

Many algorithms are bounded by memory not CPU

Other ways to get more bandwidth
  - Make addresses sequential
  - Make address generations independent
  - **Prefetch by software** (make address generations go ahead)
  - Use multiple threads/cores

How costly is it to communicate between threads?
Software prefetch

- hardware prefetch happens only for sequential (a small constant stride) accesses
- for other patterns, you the programmer may know addresses you are going to access soon

intrinsic:

```c
__builtin_prefetch(a[, rw, hint])
```
Software prefetch

- hardware prefetch happens only for sequential (a small constant stride) accesses
- for other patterns, you the programmer may know addresses you are going to access soon
- *if* you can generate those addresses much ahead of actual load instructions, you can *prefetch* them
Software prefetch

- hardware prefetch happens only for sequential (a small constant stride) accesses
- for other patterns, you the programmer may know addresses you are going to access soon
- if you can generate those addresses much ahead of actual load instructions, you can `prefetch` them

instructions:
  - `prefetchcht{0,1,2}`
  - `prefetchnta`
Software prefetch

- hardware prefetch happens only for sequential (a small constant stride) accesses
- for other patterns, you the programmer may know addresses you are going to access soon
- if you can generate those addresses much ahead of actual load instructions, you can \textit{prefetch} them
- instructions:
  - \texttt{prefetch\{0,1,2\}}
  - \texttt{prefetchnta}
- intrinsics:
  \begin{verbatim}
  int __builtin_prefetch(a [, rw, hint ])
  \end{verbatim}
How to apply software prefetch?

- truth is, there are actually not many circumstances this is useful
How to apply software prefetch?

- truth is, there are actually not many circumstances this is useful
- why? by the time you can *prefetch* it, you can likewise *load* it!
How to apply software prefetch?

• truth is, there are actually not many circumstances this is useful
• why? by the time you can *prefetch* it, you can likewise *load* it!
• in our example,
  • no point in applying it to index-based accesses (CPU will issue many load instructions already)
How to apply software prefetch?

- truth is, there are actually not many circumstances this is useful
- why? by the time you can *prefetch* it, you can likewise *load* it!
- in our example,
  - no point in applying it to index-based accesses (CPU will issue many load instructions already)
  - on the other hand, it’s difficult to apply it to list traversal (it takes equally long time to generate address to prefetch)
How to apply software prefetch?

- truth is, there are actually not many circumstances this is useful
- why? by the time you can *prefetch* it, you can likewise *load* it!
- in our example,
  - no point in applying it to index-based accesses (CPU will issue many load instructions already)
  - on the other hand, it’s difficult to apply it to list traversal (it takes equally long time to generate address to prefetch)
- the only way to apply it is to change the data structure of the linked list
How to apply software prefetch?

- truth is, there are actually not many circumstances this is useful
- why? by the time you can *prefetch* it, you can likewise *load* it!
- in our example,
  - no point in applying it to index-based accesses (CPU will issue many load instructions already)
  - on the other hand, it’s difficult to apply it to list traversal (it takes equally long time to generate address to prefetch)
- the only way to apply it is to change the data structure of the linked list
- but how?
How to apply software prefetch?

- have another pointer pointing many elements ahead

```c
for (N times) {
    p = p->next;
    prefetch(p->prefetch);
}
```

- it should point to $Q$ elements ahead to have $Q$ concurrent accesses in flight

"prefetch pointers" pointing to several elements ahead
Result

bandwidth with a number of chains $[\geq 100000000]$
Summary: bandwidth of various access patterns

- sequential (w/o pointer chase) > sorted list
- > random (w/o pointer chase) ≈ 10 random lists ≈ random list + prefetch pointers
- > random list
1 Introduction
2 Organization of processors, caches, and memory
3 Caches
4 So how costly is it to access data?
   • Latency
   • Bandwidth
   • More bandwidth = concurrent accesses
5 Many algorithms are bounded by memory not CPU
6 Other ways to get more bandwidth
   • Make addresses sequential
   • Make address generations independent
   • Prefetch by software (make address generations go ahead)
   • Use multiple threads/cores
7 How costly is it to communicate between threads?
Memory bandwidth with multiple cores

- run up to 16 threads, all in a single socket
- each thread runs on a distinct physical core
- all memory allocated to socket 0 (`numactl -N 0 -i 0`)

<table>
<thead>
<tr>
<th>Size of the Region (bytes)</th>
<th>Bandwidth with a Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 chain, 1 thread</td>
<td>40 GB/sec</td>
</tr>
<tr>
<td>1 chain, 4 thread</td>
<td>35 GB/sec</td>
</tr>
<tr>
<td>1 chain, 16 thread</td>
<td>30 GB/sec</td>
</tr>
<tr>
<td>10 chain, 1 thread</td>
<td>25 GB/sec</td>
</tr>
<tr>
<td>10 chain, 4 thread</td>
<td>20 GB/sec</td>
</tr>
<tr>
<td>10 chain, 16 thread</td>
<td>15 GB/sec</td>
</tr>
</tbody>
</table>

![Bandwidth with a number of threads graph]
Contents

1 Introduction
2 Organization of processors, caches, and memory
3 Caches
4 So how costly is it to access data?
   • Latency
   • Bandwidth
   • More bandwidth = concurrent accesses
5 Many algorithms are bounded by memory not CPU
6 Other ways to get more bandwidth
   • Make addresses sequential
   • Make address generations independent
   • Prefetch by software (make address generations go ahead)
   • Use multiple threads/cores
7 How costly is it to communicate between threads?
if thread $P$ writes to an address $a$ and then another thread $B$ reads from $a$, $Q$ observes the value written by $P$
Shared memory

- if thread $P$ writes to an address $a$ and then another thread $B$ reads from $a$, $Q$ observes the value written by $P$

- ordinary load/store instructions accomplish this (*hardware shared memory*)

- this should not be taken for granted; processors have *caches* and a single address may be cached by multiple cores/sockets
Shared memory

- processors sharing memory are running a complex, cache coherence protocol to accomplish this
- roughly,
Shared memory

- ⇒ processors sharing memory are running a complex, *cache coherence protocol* to accomplish this
- roughly,
  - a write to an address by a processor “invalidates” all other cache lines holding the address, so that no caches hold “stale” values
⇒ processors sharing memory are running a complex, *cache coherence protocol* to accomplish this

- roughly,
  - *write*: a write to an address by a processor “invalidates” all other cache lines holding the address, so that no caches hold “stale” values
  - *read*: a read to an invalid line causes a miss and searches for a cache holding its “valid” value
An example protocol: the MSI protocol

- each line of a cache is in one of the following states
  - Modified (green), Shared (yellow), Invalid (red)
An example protocol: the MSI protocol

- each line of a cache is in one of the following states
  - *Modified* (green), *Shared* (yellow), *Invalid* (red)

- Modified (green) $\iff$ you can read and write the line without invoking a transaction
- Shared (yellow) $\iff$ you can read but not write the line without invoking a transaction
- Invalid (red) $\iff$ you can neither read nor write the line without invoking a transaction
An example protocol: the MSI protocol

A single address may be cached in multiple caches (lines). There are only two legitimate states for each line:

1. Modified (owner) + others Invalid
2. No Modified

Memory controller
Hardware thread (virtual core, CPU)
(physical) core
L2 cache
L1 cache
Chip (socket, node, CPU)
Interconnect
An example protocol: the MSI protocol

- a single address may be cached in multiple caches (lines)
An example protocol: the MSI protocol

- A single address may be cached in multiple caches (lines).
- \( \Rightarrow \) There are only two legitimate states for each line:
  - One Modified *(owner)* + others Invalid (*-*, *-*, *-*, *-*, *-*, ...)
  - No Modified (*-*, *-*, *-*, *-*, *-*, ...)

---

**Diagram:**
- Memory controller
- L3 cache
- Hardware thread *(virtual core, CPU)*
- (Physical) core
- L2 cache
- L1 cache
- Chip *(socket, node, CPU)*
- Interconnect
An example protocol: the MSI protocol

- a single address may be cached in multiple caches (lines)
- ⇒ there are only two legitimate states for each line
  1. one Modified (owner) + others Invalid (■, ■, ■, ■, ■, ...)
  2. no Modified (■, ■, ■, ■, ■, ■, ...)

Diagram with: memory controller, L3 cache, hardware thread (virtual core, CPU), (physical) core, L2 cache, L1 cache, chip (socket, node, CPU), interconnect.
Cache states and transaction

- suppose a processor reads or writes an address and finds a line caching it
- what happens when the line is in each state:

<table>
<thead>
<tr>
<th></th>
<th>Modified</th>
<th>Shared</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>hit</td>
<td>hit</td>
<td>read miss</td>
</tr>
<tr>
<td>write</td>
<td>hit</td>
<td>write miss</td>
<td>read miss; write miss</td>
</tr>
</tbody>
</table>
Cache states and transaction

- suppose a processor reads or writes an address and finds a line caching it

- what happens when the line is in each state:

<table>
<thead>
<tr>
<th></th>
<th>Modified</th>
<th>Shared</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>hit</td>
<td>hit</td>
<td>read miss</td>
</tr>
<tr>
<td>write</td>
<td>hit</td>
<td>write miss</td>
<td>read miss; write miss</td>
</tr>
</tbody>
</table>

- read miss: →
  - there may be a cache holding it in Modified state (owner)
  - searches for the owner and if found, downgrade it to Shared

- , , , [ ], , , . . . ⇒ , , , [ ], , , . . .
Cache states and transaction

- suppose a processor reads or writes an address and finds a line caching it

- what happens when the line is in each state:

<table>
<thead>
<tr>
<th></th>
<th>Modified</th>
<th>Shared</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>hit</td>
<td>hit</td>
<td>read miss</td>
</tr>
<tr>
<td>write</td>
<td>hit</td>
<td>write miss</td>
<td>read miss; write miss</td>
</tr>
</tbody>
</table>

- read miss: →
  - there may be a cache holding it in Modified state *(owner)*
  - searches for the owner and if found, downgrade it to Shared
  - 

- write miss: →
  - there may be caches holding it in Shared state *(sharer)*
  - searches for sharers and downgrade them to Invalid
  - 


MESI and MESIF

- Extensions to MSI have been commonly used
MESI and MESIF

- Extensions to MSI have been commonly used
- **MESI:** MSI + Exclusive (owned but not modified)
  - when a read request finds no other caches that have the line, it owns it as Exclusive
  - Exclusive lines do not have to be written back to main memory when discarded
MESI and MESIF

- Extensions to MSI have been commonly used

**MESI:** MSI + Exclusive (owned but not modified)
  - when a read request finds no other caches that have the line, it owns it as Exclusive
  - Exclusive lines do not have to be written back to main memory when discarded

**MESIF:** MESI + Forwarding (a cache responsible for forwarding a line)
  - used in Intel QuickPath
  - when a line is shared by many readers, one is designated as the Forwarder
  - when another cache requests the line, only the forwarder sends it and the new requester becomes the forwarder
  - (in MSI or MESI, all sharers forward it)
How to measure communication latency?

- measure “ping-pong” latency between two threads

```java
volatile long x = 0;
volatile long y = 0;

(ping thread)
for (i = 0; i < n; i++) {
    x = i + 1;
    while (y <= i);
}

(pong thread)
for (i = 0; i < n; i++) {
    while (x <= i);
    y = i + 1;
}
```
Remarks

- environment
  - Haswell E5-2686
  - 2 hardware threads × 16 cores × 2 sockets (= 64 processors seen by OS)
- ensure variables x and y are at least 64 bytes apart (not on the same cache line)
- bind both threads on specific processors by OpenMP environment variable `OMP_BIND_PROC=true`
- try all combinations of threads (i.e., with p threads, measure all the p(p − 1) pairs) and show a matrix
Result

- \((i, j)\) indicates the roundtrip latency (in clocks) between processor \(i\) and \(j\)

\[
\begin{array}{cccccccc}
0 & 8 & 16 & 24 & 32 & 40 & 48 & 56 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

- A beautiful pattern emerges which is obviously telling

<table>
<thead>
<tr>
<th>src</th>
<th>dest</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1-15</td>
<td>(\approx 500)</td>
</tr>
<tr>
<td>0</td>
<td>16-31</td>
<td>(\approx 1200)</td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td>(\approx 50)</td>
</tr>
<tr>
<td>0</td>
<td>33-47</td>
<td>(\approx 500)</td>
</tr>
<tr>
<td>0</td>
<td>48-63</td>
<td>(\approx 1200)</td>
</tr>
</tbody>
</table>
e.g., which processor is “close” to processor 0?
  - 32 is closest
  - 1-15 and 33-47 are close
  - 16-31 and 48-63 are farthest
  - a natural interpretation
    - $x$ and $(x + 32)$ are two hardware threads on a core
    - 0-15 are 16 cores on a socket
  - latencies
    | hwts within a core | 50 |
    | cores within a socket | 500 |
    | across sockets | 1200 |
Summary (1): latency and bandwidth

- **latency** of data access heavily depends on which level of caches you actually access:
  \[ L1 \ (a \ few \ cycles) \leq \ main \ memory \ (> \ 200 \ cycles) \]

- a single core bandwidth is limited by:
  \[
  \frac{\text{cache line size} \times \text{LFB size}}{\text{latency}}
  \]

- for main memory, it’s much lower than what you see in the spec
- max bandwidth is attainable only with multiple cores
Summary (2): bandwidth differs by access patterns

- \[
\text{bandwidth} = \frac{\text{line size} \times \text{number of accesses in flight}}{\text{latency}}
\]

- **bandwidth** heavily depends on the number of in-flight accesses, which depend on *access patterns*
  - random address pointer chasing
  - random but independent addresses
  - sequential
Common misunderstanding

- pointer chasing is always bad
  - not when data fit in L1 (perhaps L2) cache
  - not when accessed addresses are sequential
  - not when you manage to chase many pointer chains
- random access is always worse than sequential access
  - not so much when an element $\approx$ cache size
cores communicate as a side effect of memory accesses (cache misses)

it is naturally as expensive as L2/L3 misses (or more), depending on whom you communicate with

shared memory is nice, but you cannot forget the cost