

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

Kenjiro Taura

# Contents

- ➊ Introduction
- ➋ Many algorithms are bounded by memory not CPU
- ➌ Organization of processors, caches, and memory
- ➍ So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- ➎ 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
- ➏ How costly is it to communicate between threads?

# Contents

- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 How costly is it to communicate between threads?

# Introduction

- 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 *data access*

# Why data access is so important?

- no data, no computation

```
1 for (k = 0; k < A.nnz; k++) {  
2     i,j,Aij = A.elms[k];  
3     y[i] += Aij * x[j];  
4 }
```

```
1 for (i = 0; i < M; i++)  
2     for (j = 0; j < N; j++)  
3         for (k = 0; k < K; k++)  
4             C(i,j) += A(i,k) * B(k,j);
```

# Why data access is so important?

- no **data**, no **computation**

```
1 for (k = 0; k < A.nnz; k++) {  
2     i,j,Aij = A.elms[k];  
3     y[i] += Aij * x[j];  
4 }
```

```
1 for (i = 0; i < M; i++)  
2     for (j = 0; j < N; j++)  
3         for (k = 0; k < K; k++)  
4             C(i,j) += A(i,k) * B(k,j);
```

- accessing data is sometimes *far more costly* than calculation

# Why data access is so important?

- no **data**, no **computation**

```
1 for (k = 0; k < A.nnz; k++) {  
2   i,j,Aij = A.elems[k];  
3   y[i] += Aij * x[j];  
4 }
```

```
1 for (i = 0; i < M; i++)  
2   for (j = 0; j < N; j++)  
3     for (k = 0; k < K; k++)  
4       C(i,j) += A(i,k) * B(k,j);
```

- accessing data is sometimes *far more costly* than calculation
- moreover, the cost of the same data access instruction 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 are connected
- understand the behavior of caches, so as to reason about how much traffic the algorithm will generate between main memory  $\leftrightarrow$  caches (and among cache levels)
- $\Rightarrow$  be able to reason about *a performance limit of your program, due to the memory*



# Pragmatic goals of the study

- **latency**: get a sense of how many cycles it takes to get data from main memory and caches

# Pragmatic goals of the study

- **latency**: get a sense of how many cycles it takes to get data from main memory and caches
- **bandwidth**: get a sense of how much data CPU can bring from main memory and caches

# Pragmatic goals of the study

- **latency**: get a sense of how many cycles it takes to get data from main memory and caches
- **bandwidth**: get a sense of how much data CPU can bring from main memory and caches
- what does “memory bandwidth” we see in a processor spec sheet really mean? e.g.,

- the processor data sheet of E5-2698 (68 GB/s):

[http://ark.intel.com/products/81060/Intel-Xeon-Processor-E5-2698-v3-40M-Cache-2\\_30-GHz](http://ark.intel.com/products/81060/Intel-Xeon-Processor-E5-2698-v3-40M-Cache-2_30-GHz)

- in general,

8 bytes  $\times$  DDR frequency  $\times$  memory channel, per CPU socket

- our “big” CPU (Skylake-X Gold 6130)

8 bytes  $\times$  2666 MHz  $\times$  6 channels = 128 GB/sec per socket

128  $\times$  4 sockets = 512 GB/sec in the entire node

# Pragmatic goals of the study

- **latency**: get a sense of how many cycles it takes to get data from main memory and caches
- **bandwidth**: get a sense of how much data CPU can bring from main memory and caches
- what does “memory bandwidth” we see in a processor spec sheet really mean? e.g.,

- the processor data sheet of E5-2698 (68 GB/s):

[http://ark.intel.com/products/81060/Intel-Xeon-Processor-E5-2698-v3-40M-Cache-2\\_30-GHz](http://ark.intel.com/products/81060/Intel-Xeon-Processor-E5-2698-v3-40M-Cache-2_30-GHz)

- in general,

8 bytes  $\times$  DDR frequency  $\times$  memory channel, per CPU socket

- our “big” CPU (Skylake-X Gold 6130)

8 bytes  $\times$  2666 MHz  $\times$  6 channels = 128 GB/sec per socket

128  $\times$  4 sockets = 512 GB/sec in the entire node

- Can we achieve this easily? If not, when/how can we?

# Contents

- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 How costly is it to communicate between threads?

# What does memory performance imply for FLOPS?

- many computationally *efficient* algorithms do not touch the same data too many times

# What does memory performance imply for FLOPS?

- many computationally *efficient* algorithms do not touch the same data too many times
- e.g.,  $O(n)$  algorithms  $\rightarrow$  uses a single element only a constant number of times (on average)

# What does memory performance imply for FLOPS?

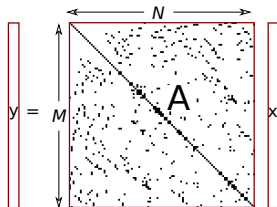
- many computationally *efficient* algorithms do not touch the same data too many times
- e.g.,  $O(n)$  algorithms  $\rightarrow$  uses a single element only a constant number of times (on average)
- if data  $\gg$  cache for such an algorithm, the algorithm's performance is often limited by the memory bandwidth (or, worse, latency), *not processor's compute throughput*



# Example: SpMV

- remember COO

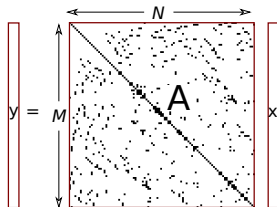
```
1 for (k = 0; k < A.nnz; k++) {  
2     i,j,Aij = A.elms[k];  
3     y[i] += Aij * x[j];  
4 }
```



# Example: SpMV

- remember COO

```
1 for (k = 0; k < A.nnz; k++) {  
2   i,j,Aij = A.elems[k];  
3   y[i] += Aij * x[j];  
4 }
```

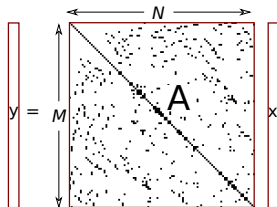


- accesses **16 nnz** bytes and performs **2 nnz** flops
  - assuming elements of **double** (8 bytes) and indexes of **ints** (4 bytes  $\times$  2), not counting access to  $x$  and  $y$
  - details aside, it performs only *an FMA / element*

# Example: SpMV

- remember COO

```
1 for (k = 0; k < A.nnz; k++) {  
2     i,j,Aij = A.elems[k];  
3     y[i] += Aij * x[j];  
4 }
```

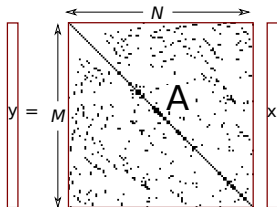


- accesses **16 nnz** bytes and performs **2 nnz** flops
  - assuming elements of **double** (8 bytes) and indexes of **ints** (4 bytes  $\times$  2), not counting access to  $x$  and  $y$
  - details aside, it performs only *an FMA / element*
- to achieve Skylake-X peak (**32 DP FMAs** per core per cycle), a core must access **32** matrix elements (= **512** bytes) / cycle

# Example: SpMV

- remember COO

```
1 for (k = 0; k < A.nnz; k++) {  
2     i,j,Aij = A.elms[k];  
3     y[i] += Aij * x[j];  
4 }
```



- accesses **16 nnz** bytes and performs **2 nnz** flops
  - assuming elements of **double** (8 bytes) and indexes of **ints** (4 bytes  $\times$  2), not counting access to  $x$  and  $y$
  - details aside, it performs only *an FMA / element*
- to achieve Skylake-X peak (**32 DP FMAs** per core per cycle), a core must access **32** matrix elements (= **512** bytes) / cycle
- assuming 2.0GHz processor and the matrix  $\gg$  cache, it requires the *main memory bandwidth* of
$$\approx 512 \text{ bytes} \times 2.0 \text{ GHz} = 1 \text{ TB/sec per core (no way!)}$$

# Memory-bound algorithms (applications)

- say an algorithm performs  $C$  flops (or computation in more general) on  $N$  bytes of data
  - assume it needs to access every element of the  $N$  bytes at least once (likely the case)

# Memory-bound algorithms (applications)

- say an algorithm performs  $C$  flops (or computation in more general) on  $N$  bytes of data
  - assume it needs to access every element of the  $N$  bytes at least once (likely the case)
- there are two obvious lower bounds on the time to complete the algorithm

$$T \geq \frac{C}{\text{the peak FLOPS}} \quad (\text{compute})$$

$$T \geq \frac{N}{\text{the peak memory bandwidth}} \quad (\text{memory})$$

# Memory-bound algorithms (applications)

- say an algorithm performs  $C$  flops (or computation in more general) on  $N$  bytes of data
  - assume it needs to access every element of the  $N$  bytes at least once (likely the case)
- there are two obvious lower bounds on the time to complete the algorithm

$$T \geq \frac{C}{\text{the peak FLOPS}} \quad (\text{compute})$$

$$T \geq \frac{N}{\text{the peak memory bandwidth}} \quad (\text{memory})$$

- often, the latter is much larger and such algorithms are called *“memory-bound”*
- $O(N)$ ,  $O(N \log N)$  algorithms are almost always memory bound

# Memory-bound algorithms (applications)

- memory-bound  $\iff$

$$\frac{C}{\text{the peak FLOPS}} \ll \frac{N}{\text{the peak memory bandwidth}}$$

$\iff$

$$\frac{C}{N} \ll \frac{\text{the peak FLOPS}}{\text{the peak memory bandwidth}}$$

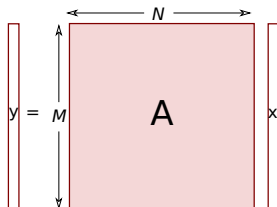
- the LHS: *arithmetic intensity* or *compute intensity* of the algorithm
- the reciprocal of RHS: the *byte per FLOPS* of the machine
- note that being memory-bound suggests it is inefficient in the processor utilization view point, but it is efficient in time-complexity sense (*it is not necessarily a bad thing*)



# Note: *dense* matrix-vector multiply

- the same argument applies even if the matrix is *dense*

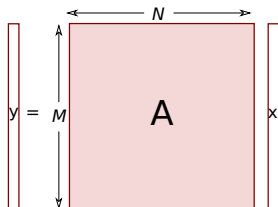
```
1 for (i = 0; i < M; i++)  
2   for (j = 0; j < N; j++)  
3     y[i] += a[i][j] * x[j];
```



# Note: *dense* matrix-vector multiply

- the same argument applies even if the matrix is *dense*

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

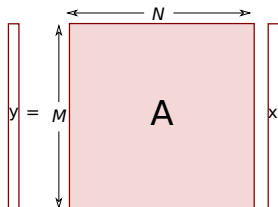


- $MN$  flops on  $(MN + M + N)$  elements

# Note: *dense* matrix-vector multiply

- the same argument applies even if the matrix is *dense*

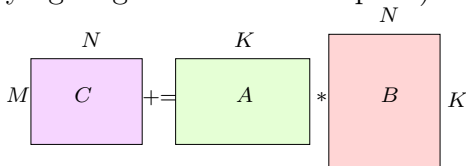
```
1 for (i = 0; i < M; i++)  
2   for (j = 0; j < N; j++)  
3     y[i] += a[i][j] * x[j];
```



- $MN$  flops on  $(MN + M + N)$  elements
- $\Rightarrow$  it performs only an FMA / matrix element

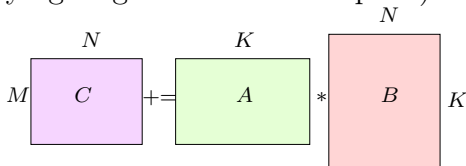
# Dense matrix-matrix multiply

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



# Dense matrix-matrix multiply

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



- for  $N \times N$  square matrices, it performs  $N^3$  FMAs on  $3N^2$  elements

# Why dense matrix-matrix multiply *can* be efficient?

- assume  $M \sim N \sim K$

```
1 for (i = 0; i < M; i++)  
2   for (j = 0; j < N; j++)  
3     for (k = 0; k < K; k++)  
4       C(i,j) += A(i,k) * B(k,j);
```

- a microscopic argument
  - the innermost statement

```
1 C(i,j) += A(i,k) * B(k,j)
```

still performs (only) 1 FMA for accessing 3 elements

- but the same element (say  $C(i,j)$ ) is used many ( $K$ ) times in the innermost loop
- similarly, the same  $A(i,k)$  is used  $N$  times
- $\Rightarrow$  after you use an element, *if you reuse it many times before it is evicted from a cache (even a register)*, then the memory traffic is hopefully not a bottleneck

# A simple `memcpy` experiment ...

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

# A simple `memcpy` experiment ...

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

```
1 $ gcc -O3 memcpy.c  
2 $ ./a.out $((1 << 26)) # 64M long elements = 512MB  
3 536870912 bytes copied in 0.117333 sec 4.575611 GB/sec
```



# A simple `memcpy` experiment ...

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

```
1 $ gcc -O3 memcpy.c  
2 $ ./a.out $((1 << 26)) # 64M long elements = 512MB  
3 536870912 bytes copied in 0.117333 sec 4.575611 GB/sec
```

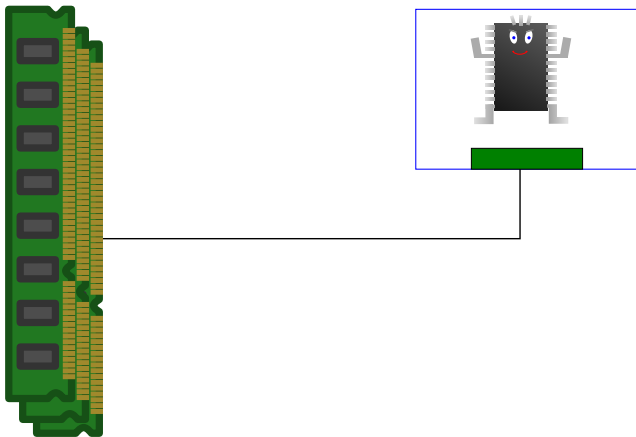
- much lower than the advertised number ...

# Contents

- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 How costly is it to communicate between threads?

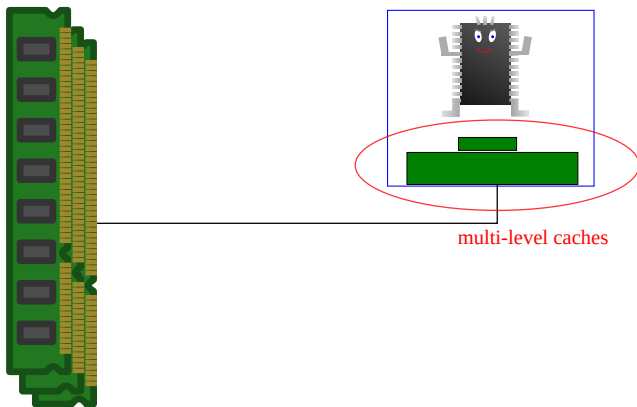
# Cache and memory in a single-core processor

you almost certainly know this (*caches* and main memory), don't you?



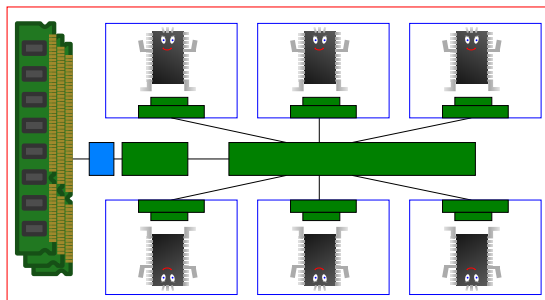
..., with multi level caches, ...

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



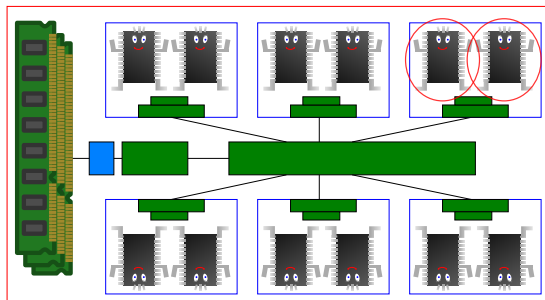
..., with multicores in a chip, ...

- 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



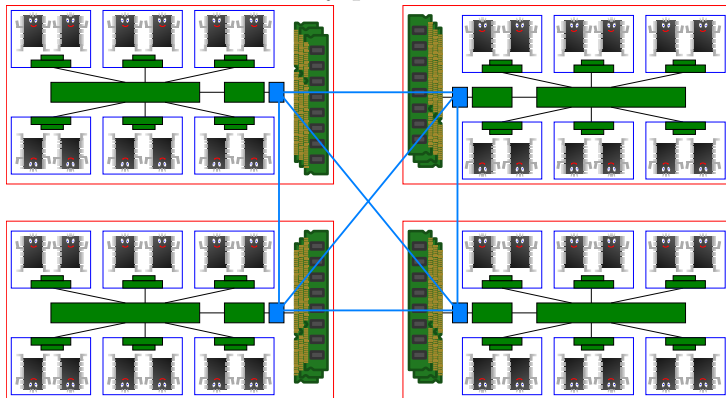
..., with simultaneous multithreading (SMT) in a core, ...

- each core has two *hardware threads*, which share L1/L2 caches and some or all execution units

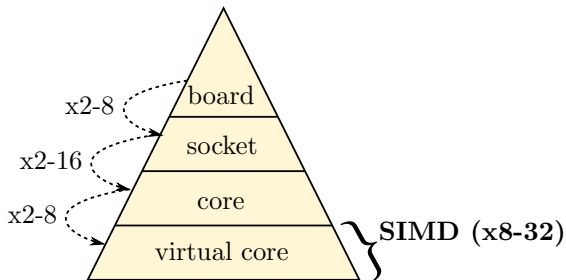


..., and with multiple sockets per node.

- 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 :  $\sim$  50MB/socket



# Cache 101

- speed :

$L1 > L2 > L3 > \text{main memory}$

# Cache 101

- speed :

$L1 > L2 > L3 > \text{main memory}$

- capacity :

$L1 < L2 < L3 < \text{main memory}$

# Cache 101

- speed :

$$L1 > L2 > L3 > \text{main memory}$$

- capacity :

$$L1 < L2 < L3 < \text{main memory}$$

- each cache holds a subset of data in the main memory

$$L1, L2, L3 \subset \text{main memory}$$

# Cache 101

- speed :

$$L1 > L2 > L3 > \text{main memory}$$

- capacity :

$$L1 < L2 < L3 < \text{main memory}$$

- each cache holds a subset of data in the main memory

$$L1, L2, L3 \subset \text{main memory}$$

- typically but not necessarily,

$$L1 \subset L2 \subset L3 \subset \text{main memory}$$

# Cache 101

- speed :

$$L1 > L2 > L3 > \text{main memory}$$

- capacity :

$$L1 < L2 < L3 < \text{main memory}$$

- each cache holds a subset of data in the main memory

$$L1, L2, L3 \subset \text{main memory}$$

- typically but not necessarily,

$$L1 \subset L2 \subset L3 \subset \text{main memory}$$

- *which subset is in caches?*  $\rightarrow$  cache management (replacement) policy

# Cache management (replacement) policy

- a cache generally holds data in *recently accessed* addresses, up to its capacity

# Cache management (replacement) policy

- a cache generally holds data in *recently accessed* addresses, up to its capacity
- this is accomplished by the **LRU replacement** policy (or its approximation):
  - 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 *recently accessed* addresses, up to its capacity
- this is accomplished by the **LRU replacement** policy (or its approximation):
  - 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 32K bytes*

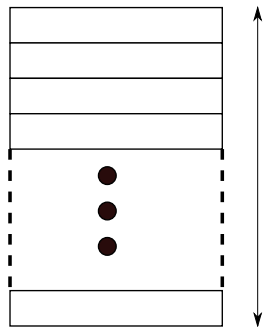


# Cache management (replacement) policy

- a cache generally holds data in *recently accessed* addresses, up to its capacity
- this is accomplished by the **LRU replacement** policy (or its approximation):
  - 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 32K bytes*
- due to implementation constraints, real caches are slightly more complex

# Cache organization : cache line

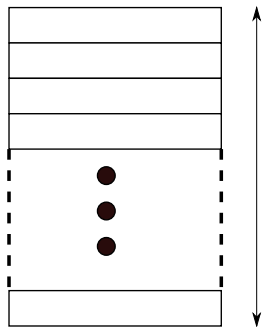
- a cache = a set of fixed size *lines*
  - typical line size = 64 bytes or 128 bytes,



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

# Cache organization : cache line

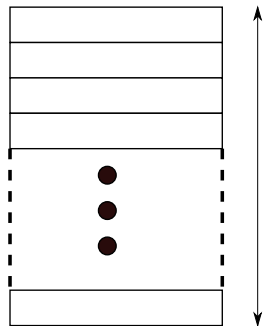
- a cache = a set of fixed size *lines*
  - typical line size = 64 bytes or 128 bytes,
- a single line is the minimum unit of data transfer between levels (and replacement)



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)

$\approx$  *most recently accessed 512 distinct lines*

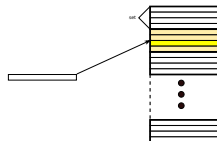
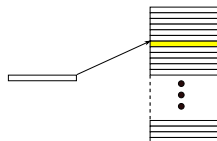
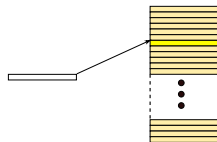
# Associativity of caches

**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

- Skylake-X Gold 6130

level	line size	capacity	associativity
L1	64B	32KB/core	8
L2	64B	1MB/core	16
L3	64B	22MB/socket (16 cores)	11

- Ivy Bridge E5-2650L

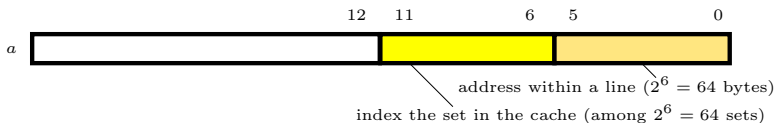
level	line size	capacity	associativity
L1	64B	32KB/core	8
L2	64B	256KB/core	8
L3	64B	36MB/socket (8 cores)	20

# What you need to remember in practice about associativity

- *avoid having addresses used together “a-large-power-of-two” bytes apart*
- corollaries:
  - 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?  $\Rightarrow$  they tend to go to the same set and “conflict misses” result

# Conflict misses

- consider 8-way set associative L1 cache with 32KB (line size = 64B)
  - $32\text{KB}/64\text{B} = 512 (= 2^9)$  lines
  - $512/8 = 64 (= 2^6)$  sets
- $\Rightarrow$  given an address  $a$ ,  $a[6:11]$  (6 bits) designates the set it belongs to (indexing)

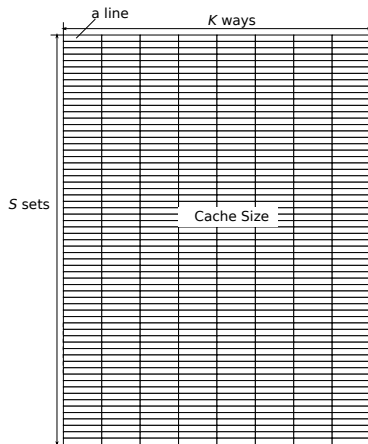


- if two addresses  $a$  and  $b$  are a multiple of  $2^{12}$  (4096) bytes apart, they go to the same set



# A convenient way to understand conflicts

- it's convenient to think of a cache as two dimensional array of lines.  
e.g. 32KB, 8-way set associative =  
 $64 \text{ (sets)} \times 8 \text{ (ways)} \text{ array of lines}$



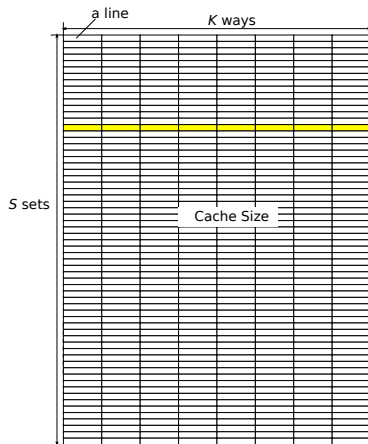
# A convenient way to understand conflicts

- formula 1:

$$\text{worst stride} = \frac{\text{cache size}}{\text{associativity}} \text{ bytes}$$

if addresses are this much apart,  
they go to the same set

- e.g., 32KB 8-way set associative  
⇒ the worst stride = 4096



# A convenient way to understand conflicts

- lesser powers of two are significant too; continuing with the same setting (32KB, 8way-set associative)

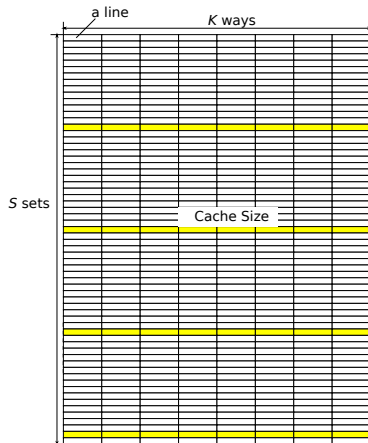
stride	the number of sets they are mapped to	utilization
2048	2	1/32
1024	4	1/16
512	8	1/8
256	16	1/4
128	32	1/2
64	64	1

- formula 2: you stride by

$$P \times \text{line size} \quad (P \text{ divides } S)$$

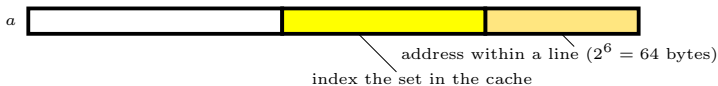
$\Rightarrow$  you utilize only  $1/P$  of the capacity

- N.B. formula 1 is a special case, with  $P = S$



# A remark about virtually-indexed vs. physically-indexed caches

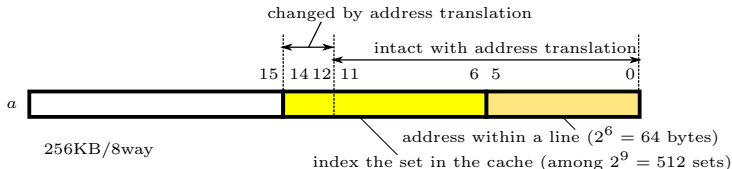
- caches typically use *physical* addresses to select the set an address maps to
- so “addresses” I have been talking about are physical addresses, not virtual addresses you can see as pointer values



- since virtual  $\rightarrow$  physical mapping is determined by the OS (based on the availability of physical memory),  
*“two virtual addresses  $2^b$  bytes apart”*  
does *not* necessarily imply  
*“their physical addresses  $2^b$  bytes apart”*
- so what’s the significance of the stories so far?

# A remark about virtually-indexed vs. physically-indexed caches

- virtual  $\rightarrow$  physical translation happens with page granularity (typically,  $2^{12} = 4096$  bytes)
- $\rightarrow$  the last 12 bits are intact with the translation



# A remark about virtually-indexed vs. physically-indexed caches

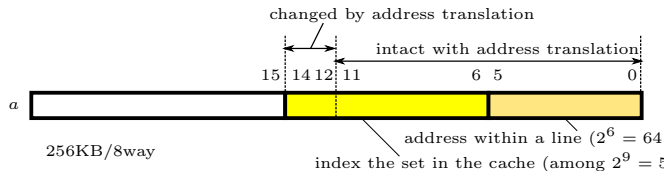
- therefore,

*“two **virtual** addresses  $2^b$  bytes apart”  $\rightarrow$  “their **physical** addresses  $2^b$  bytes apart”*

*for up to page size ( $2^b \leq \text{page size}$ )*

- $\rightarrow$  the formula 2 is valid for strides up to page size

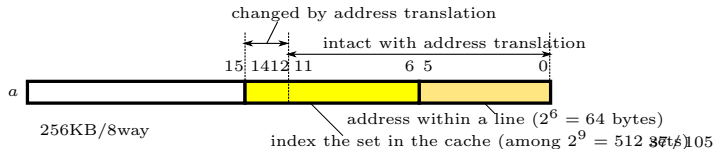
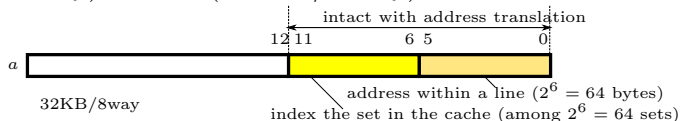
stride	utilization
4096	1/64
2048	1/32
1024	1/16
512	1/8
256	1/4
128	1/2
64	1



# Remarks applied to different cache levels

- small caches that use only the last 12 bits to index the set make no difference between virtually- and physically-indexed caches
- for larger caches, the utilization will similarly drop up to stride = 4096, after which it will stay around 1/64
- L1 (32KB/8-way) vs. L2 (256KB/8-way)

stride	utilization
...	$\sim 1/64$
16384	$\sim 1/64$
8192	$\sim 1/64$
4096	$1/64$
2048	$1/32$
1024	$1/16$
512	$1/8$
256	$1/4$
128	$1/2$
64	1



# Avoiding conflict misses

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

```
1 float a[100][1024];
```

then  $a[i][j]$  and  $a[i+1][j]$  go to the same set in L1 cache;

- $\Rightarrow$  scanning a column of such a matrix will experience almost 100% cache miss
- avoid it by:

```
1 float a[100][1024+16];
```



# What are in the cache?

- consider a cache of
  - capacity =  $C$  bytes
  - line size =  $Z$  bytes
  - associativity =  $K$

# What are in the cache?

- consider a cache of
  - capacity =  $C$  bytes
  - line size =  $Z$  bytes
  - associativity =  $K$
- 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 a cache of
  - capacity =  $C$  bytes
  - line size =  $Z$  bytes
  - associativity =  $K$
- 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

# What are in the cache?

- consider a cache of
  - capacity =  $C$  bytes
  - line size =  $Z$  bytes
  - associativity =  $K$
- 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

- approximation 2.0 (consider associativity too):
  - depending on the stride of the addresses you use, reason about the utilization (effective size) of the cache
  - in practice, avoid strides of “line size  $\times 2^b$ ”

# Contents

- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 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 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 How costly is it to communicate between threads?

# How to measure a latency?

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



# 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)

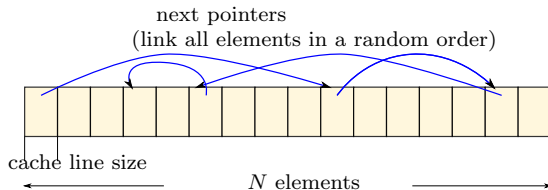
```
1  for ( $N$  times) {  
2      p = p->next;  
3  }
```

# 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)

```
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)

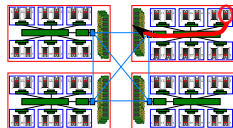
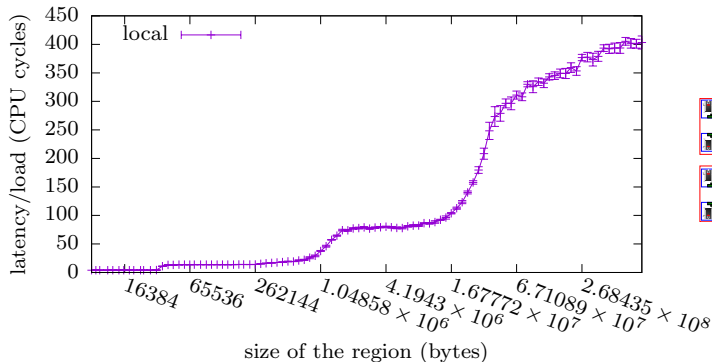


# Data size vs. latency

- main memory is local to the accessing thread

```
1 $ numactl --cpunodebind 0 --interleave 0 ./mem
2 $ numactl -N 0 -i 0 ./mem # abbreviation
```

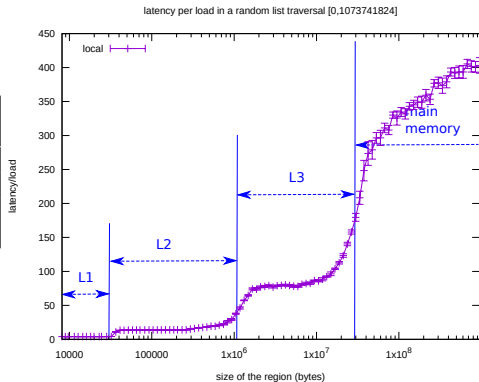
latency per load in a random list traversal [0,1073741824]



# How long are latencies

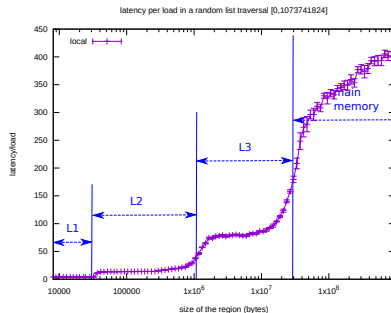
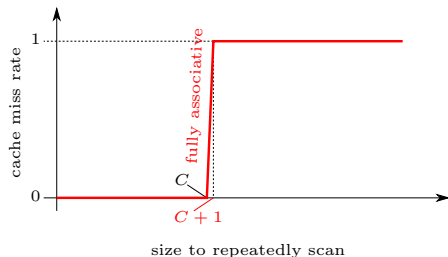
- heavily depends on in which level of the cache data fit
- environment: Skylake-X Xeon Gold 6130  
(32KB/1MB/22MB)

size	level	latency (cycles)	latency (ns)
12,736	L1	4.004	1.31
103,616	L2	13.80	4.16
2,964,928	L3	77.40	24.24
301,307,584	main	377.60	115.45



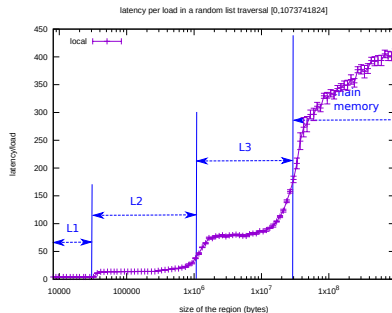
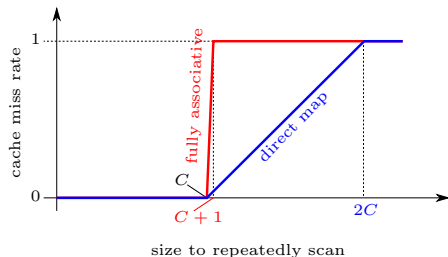
# A remark about replacement policy

- 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*
- the “cliffs” in the experimental data look gentler than the theory would suggest



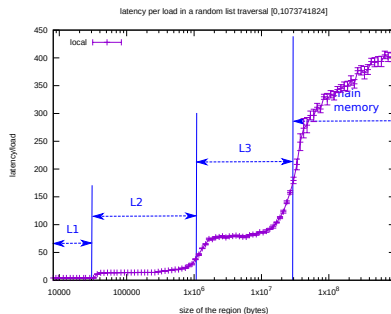
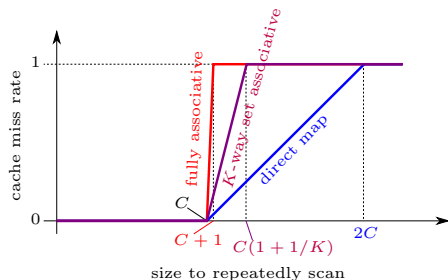
# A remark about replacement policy

- 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*
- the “cliffs” in the experimental data look gentler than the theory would suggest



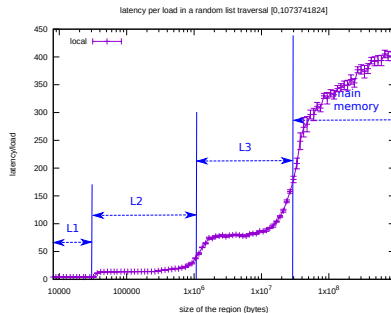
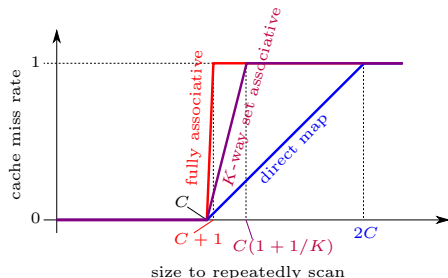
# A remark about replacement policy

- 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*
- the “cliffs” in the experimental data look gentler than the theory would suggest



# A remark about replacement policy

- part of the gap is due to virtual  $\rightarrow$  physical address translation
- another factor, especially for L3 cache, will be a recent replacement policy for cyclic accesses (c.f. <http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/>)



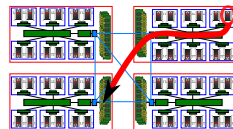
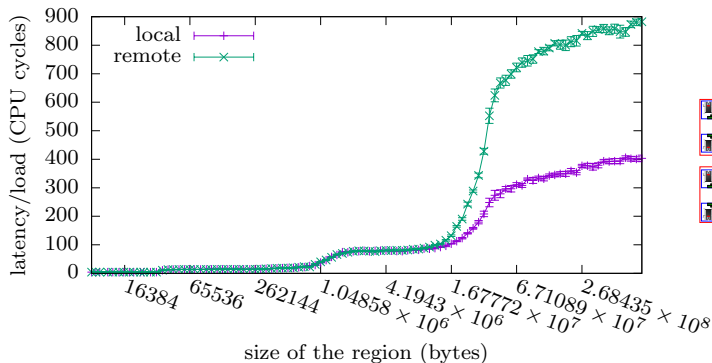


# Latency to a remote main memory

- make main memory remote to the accessing thread

```
1 $ numactl -N 0 -i 1 ./mem
```

latency per load in a random list traversal [0,1073741824]



# Contents

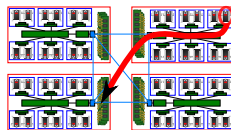
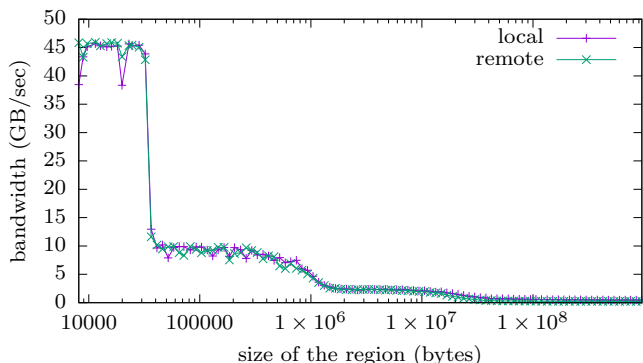
- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - **Bandwidth**
  - More bandwidth = concurrent accesses
- 5 Other ways to get more bandwidth
  - Make addresses sequential
  - Make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

# Bandwidth of a random link list traversal

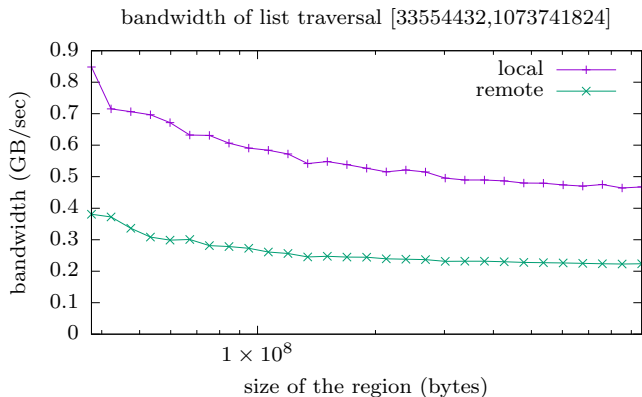
$$\text{bandwidth} = \frac{\text{total bytes read}}{\text{elapsed time}}$$

- in this experiment, we set record size = 64

bandwidth of list traversal [0,1073741824]



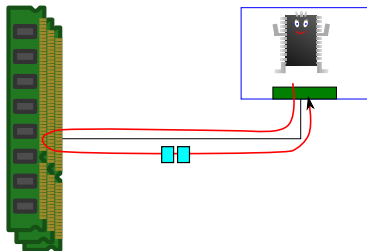
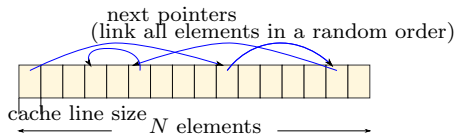
# The “main memory” bandwidth



- $\ll$  the `memcpy` bandwidth we have seen ( $\approx 4.5$  GB/s)
- not to mention the “memory bandwidth” in the spec

# 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,

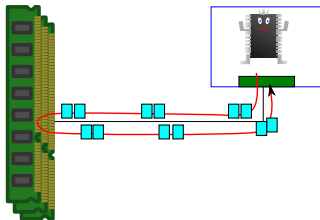
$$\text{bandwidth} = \frac{\text{a record size}}{\text{latency}}$$

- e.g., take 115.45 ns as a latency

$$\frac{64 \text{ bytes}}{115.45 \text{ ns}} \approx 0.55 \text{ GB/s}$$

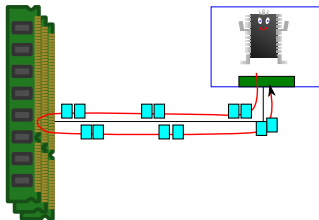
# How to get more bandwidth?

- 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; let's look at conceptually the most straightforward: traverse multiple lists

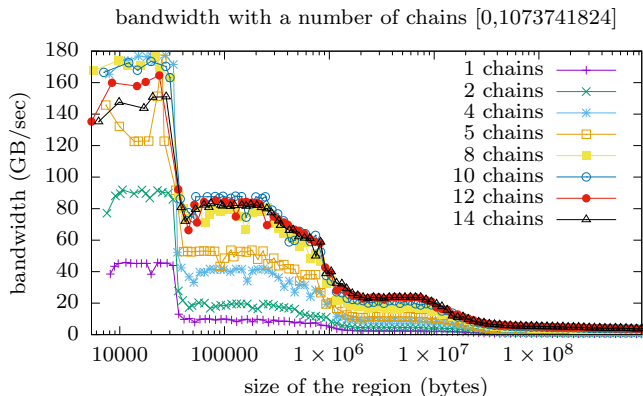
```
1  for (N times) {  
2      p1 = p1->next;  
3      p2 = p2->next;  
4      ...  
5  }
```

# Contents

- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 How costly is it to communicate between threads?



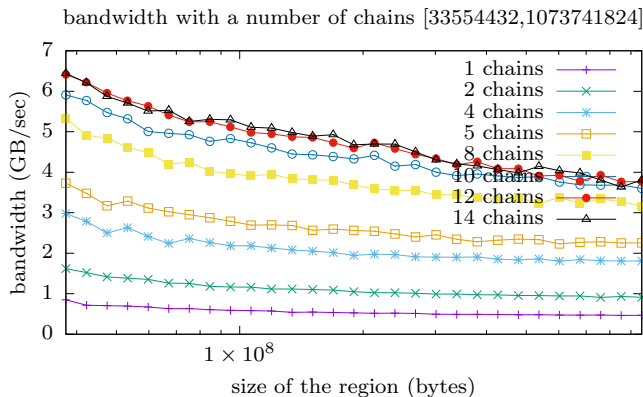
# The number of lists vs. bandwidth



- let's zoom into “main memory” regime (size > 100MB)

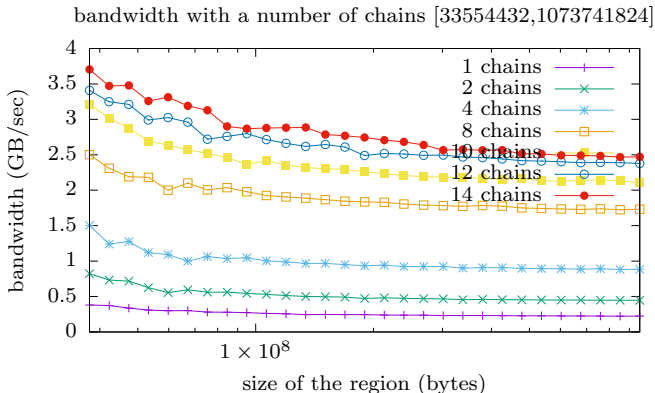
# Bandwidth to the local main memory (not cache)

- an almost proportional improvement up to  $\sim 10$  lists



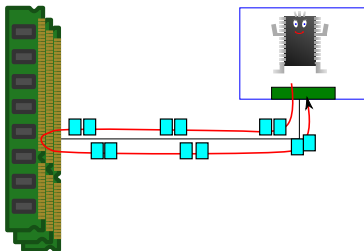
# Bandwidth to a remote main memory (not cache)

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



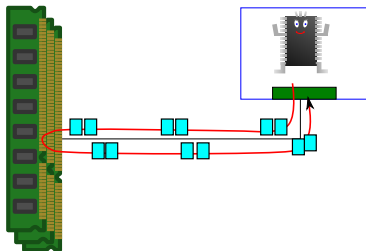
# The number of lists vs. bandwidth

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



# The number of lists vs. bandwidth

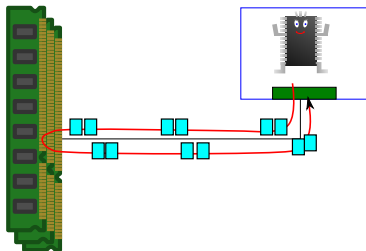
- **observation:** bandwidth increase fairly proportionally to the number of lists, matching our understanding, ...



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

# The number of lists vs. bandwidth

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



- **question:** ...but up to  $\sim 10$ , why?
- **answer:** there is a limit in the number of load operations in flight at a time

# Line Fill Buffer

- *Line fill buffer (LFB)* is the processor resource that keeps track of outstanding cache misses, and its size is 10 in Haswell
  - I could not find the definitive number for Skylake-X, but it will probably be the same

# Line Fill Buffer

- *Line fill buffer (LFB)* is the processor resource that keeps track of outstanding cache misses, and its size is 10 in Haswell
  - I could not find the definitive number for Skylake-X, but it will probably be the same
- this gives *the maximum attainable bandwidth per core*

$$\frac{\text{cache line size} \times \text{LFB size}}{\text{latency}}$$



# Line Fill Buffer

- *Line fill buffer (LFB)* is the processor resource that keeps track of outstanding cache misses, and its size is 10 in Haswell
  - I could not find the definitive number for Skylake-X, but it will probably be the same
- this gives *the maximum attainable bandwidth per core*

$$\frac{\text{cache line size} \times \text{LFB size}}{\text{latency}}$$

- this is what we've seen (still much lower than what we see in the “memory bandwidth” in the spec sheet)

# Line Fill Buffer

- *Line fill buffer (LFB)* is the processor resource that keeps track of outstanding cache misses, and its size is 10 in Haswell
  - I could not find the definitive number for Skylake-X, but it will probably be the same
- this gives *the maximum attainable bandwidth per core*

$$\frac{\text{cache line size} \times \text{LFB size}}{\text{latency}}$$

- 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)

# Contents

- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 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

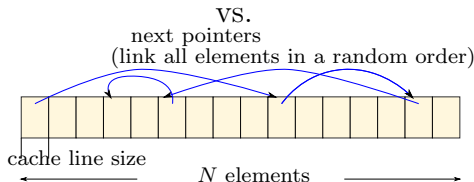
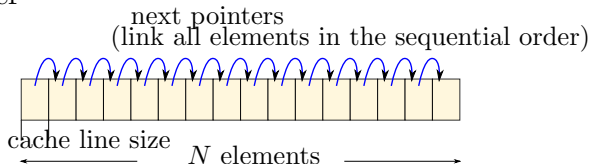
# Contents

- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 How costly is it to communicate between threads?



# Make addresses sequential

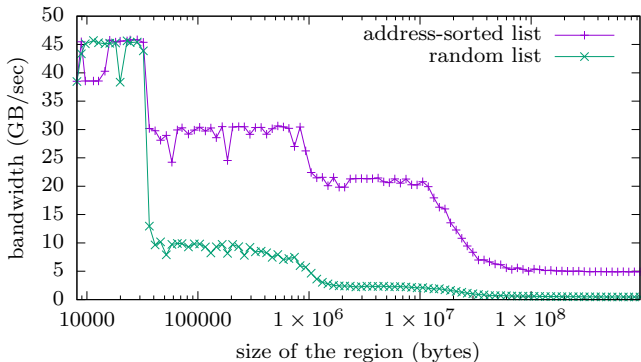
- again build a (single) linked list, but this time, `p->next` always points to the immediately following block
- note that *the instruction sequence is identical* to before; only addresses differ



# Bandwidth of traversing address-ordered list

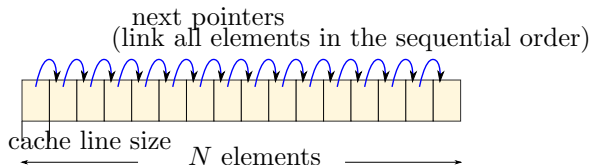
- a factor of 10 faster than random case, but this time with only a single list

bandwidth of random list traversal vs address-ordered list traversal [0,1073741824]



# 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



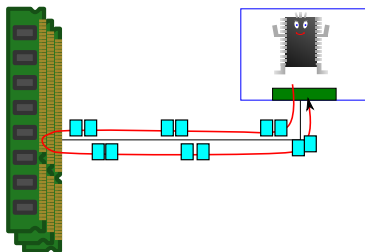
# Contents

- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 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

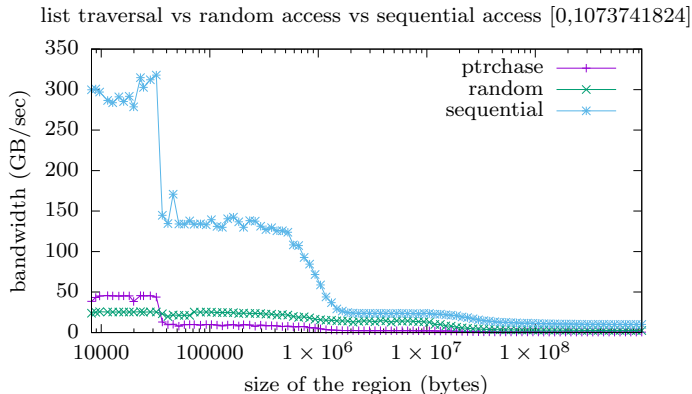
```
1 for (N times) {  
2   j = ... /* not use a[.] */  
3   a[j];  
4 }
```



- note: it's *not* a prefetch (but a real fetch)

# Bandwidth when not traversing a list

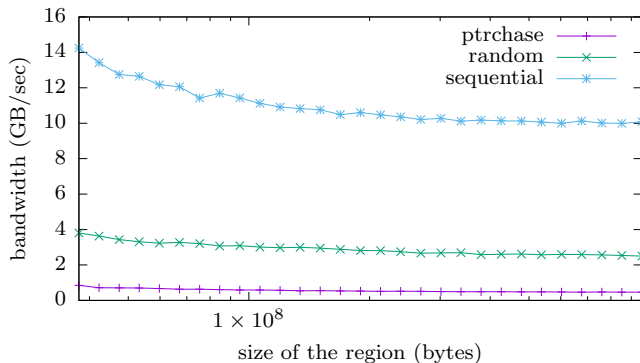
- ptrchase : chase pointers of a random list
- random : access random addresses, but w/o pointer chasing
- sequential : access sequential addresses, w/o pointer chasing



# Main memory bandwidth

- pointer chase  $\ll$  random  $<$  sequential
- random is  $\approx 5\times$  faster than traversing a single random list

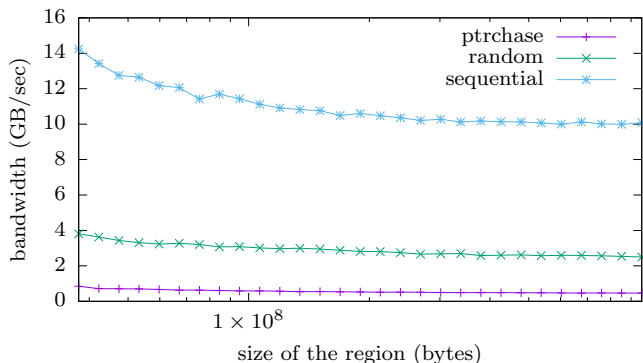
list traversal vs random access vs sequential access [33554432,1073741824]



# Main memory bandwidth (random vs. sequential)

- sequential gets  $\approx 3\times$  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?

list traversal vs random access vs sequential access [33554432,1073741824]





# Random (index) vs. sequential

- if both can have up to 10 (LFB entries) outstanding L1 cache misses, why is there *any* difference?
- I don't have a definitive answer, but presumably,
  - the hardware prefetcher happens at multiple levels ( $\rightarrow$  L1 and  $\rightarrow$  L2)
  - prefetchers to L2 are not subject of the LFP entries limit (the limit will be slightly more)
  - prefetching to L2 make effective latency to the processor smaller

# 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 ( $= 64/4$ )

# Contents

- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 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

# 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:
  - `prefetcht{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 *prefetch* them
- instructions:
  - `prefetcht{0,1,2}`
  - `prefetchnta`
- intrinsics:

```
1 __builtin_prefetch(a [, rw, hint ])
```

# 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?

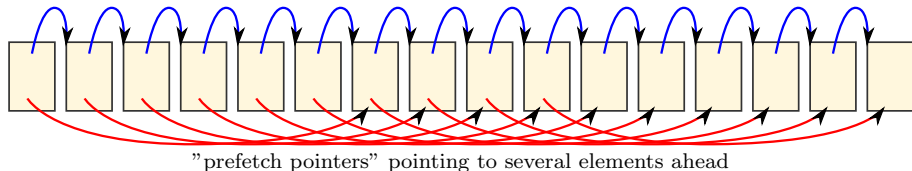
- truth is, there are actually not many circumstances this is useful
- why? by the time you can *prefetch* it, you can likewise *load* it!
- in our example,
  - no point in applying it to index-based accesses (CPU will issue many load instructions already)
  - on the other hand, it's difficult to apply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

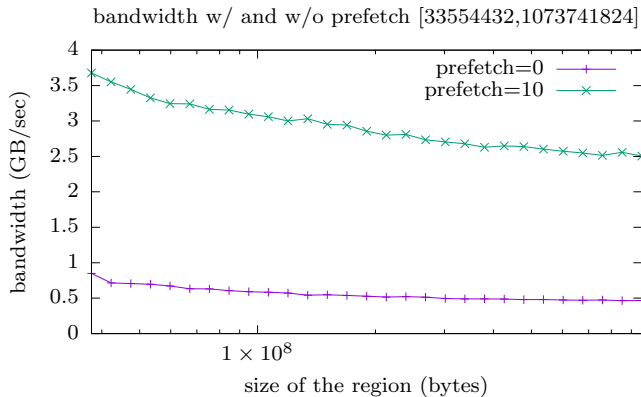
- have another pointer pointing many elements ahead

```
1 for (N times) {  
2   p = p->next;  
3   prefetch(p->prefetch);  
4 }
```

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

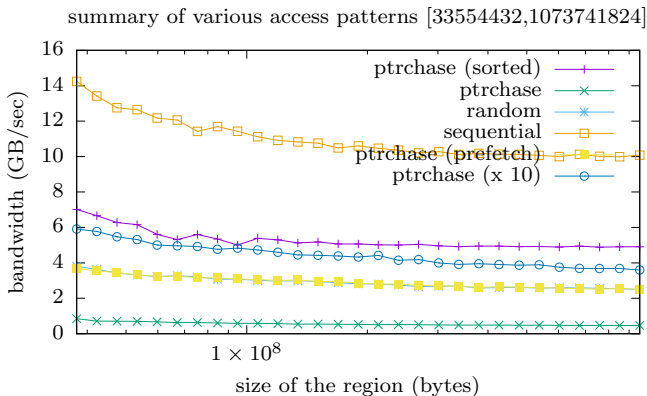


# Result



# Summary: bandwidth of various access patterns

- sequential (w/o pointer chase) > sorted list  
> random (w/o pointer chase)  $\approx$  5 random lists  $\approx$  a random list + software prefetch  
> a random list





# Contents

- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 How costly is it to communicate between threads?

# Memory bandwidth with multiple cores

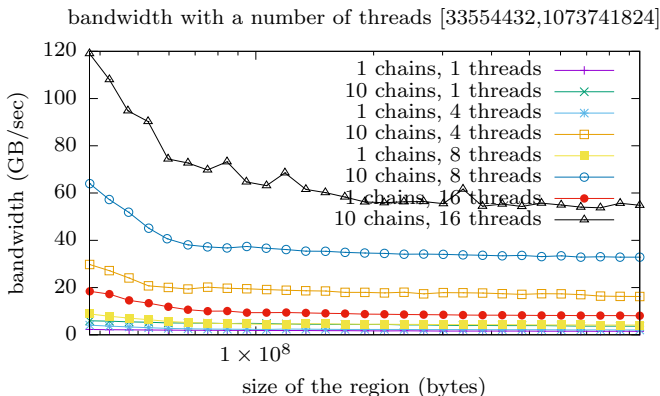
- the bandwidth to a single core is limited by LFB entries and is much lower than the memory bandwidth itself

$$\frac{\text{transfer (line) size} \times \text{LFB entries}}{\text{latency}}$$

- you can go beyond that by using multiple cores and *this is the only way*

# Memory bandwidth with multiple cores

- run up to 16 threads,
- each running on a distinct physical core of a single socket
- allocate all the data on the same socket (`numactl -N 0 -i 0`)
- note: they are still random pointer chasing

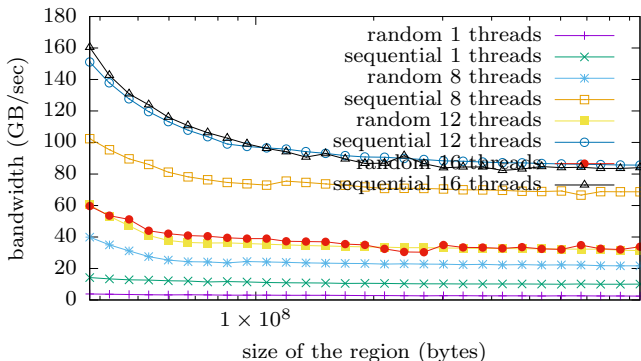


# With random indexing and sequential accesses

- similar experiments with random indexing/sequential accesses
- $\sim 80$  GB/sec with sequential accesses by  $\geq 12$  threads
- the theoretical peak is

$$8 \text{ bytes} \times 2.666 \text{ GHz} \times 6 \text{ channels} = 128 \text{ GB/sec}$$

bandwidth with various methods and number of threads [33554432,1073741824]

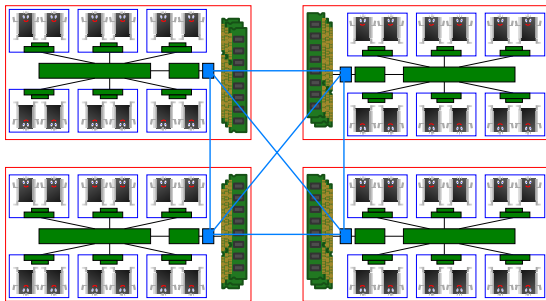


# With multiple CPU sockets

- the total bandwidth depends on how to place threads and data

threads\data	CPU $x$	CPU $y$	all CPUs	local CPU
CPU $x$	1-local	1-remote	1-all	1-local
all CPUs	all-1	all-1	all-all	all-local

- control threads/data placement by `numactl` command
- combine it with `OMP_PROC_BIND=true` to get a desired effect



# numactl command (1)

- usage (see `man numactl` for details)

```
1 $ numactl options command
```

- for underlying system calls, see `man -s 3 numa`
- processors
  - `-N x` runs threads only on the CPU(s) *x*. e.g.,

```
1 $ numactl -N 0 command # threads on CPU 0
```

- `--physcpubind x` runs threads only on *core(s)* *x*. e.g.,

```
1 # threads on cores 0-11 and 16-27
```

```
2 $ numactl --physcpubind 0-11,16-27 command
```

## numactl command (2)

- memory (data)

- `-i y` allocates data (physical pages) on CPU(s) *y*

```
1 $ numactl -i 0,1 command # data on CPU 0 or 1
2 $ numactl -i all command # data on all CPUs
```

- `-l` allocates physical pages to the CPU that touches the page for the first time (*first touch policy*; the default policy of Linux)

```
1 $ numactl -l command
```

# About the `-l` option

- `-l` (equivalent: `--localalloc`) allocates the physical page for a logical page *on the CPU that first touches it (first touch)*
- allocated physical pages do not move thereafter (unless you do so by `move_pages()` system call)
- don't be fooled by its name; it is *not* a policy that automagically makes memory accesses local
- quite contrary, it often makes a *hotspot* in a single CPU, especially when only one thread initializes (first-touches) the data
- `-iall` is not optimal, but often much safer for parallel applications



# OpenMP thread placement

- combine them with `OMP_NUM_THREADS=` and `OMP_PROC_BIND=true` to get a desired effect. e.g.,

```
1 $ OMP_NUM_THREADS=48 OMP_PROC_BIND=true numactl --physcpubind  
    0-11,16-27,32-43,48-59 -l command
```

to

- run 12 threads on each CPU (of a host in the big partition)
- and use the first touch policy

# Achieved bandwidth

- Skylake X 6130  $\times 4$  CPUs (a host of the “big” partition)
- use 12 (of 16) cores on each CPU
- in each measurement, each thread reads  $\approx 640\text{MB}$  sequentially 10 times

setting	threads	bandwidth (GB/sec)
1-local	12	85
1-remote	12	16
1-all	12	57
all-1	48	2
all-all	48	97
all-local	48	320

# Remarks on remote access bandwidths

- *numbers for remote accesses* are ridiculously low
- the measurement is repeated 6 times and there were almost no variations in the result (within a few per cents)
- I am suspecting a wrong BIOS snoop setting (<https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-mon/topic/602160>)

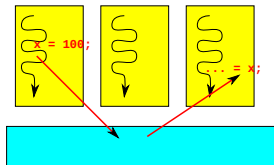
setting	threads	bandwidth (GB/sec)
1-local	12	85
1-remote	12	16
1-all	12	57
all-1	48	2
all-all	48	97
all-local	48	320

# Contents

- 1 Introduction
- 2 Many algorithms are bounded by memory not CPU
- 3 Organization of processors, caches, and memory
- 4 So how costly is it to access data?
  - Latency
  - Bandwidth
  - More bandwidth = concurrent accesses
- 5 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
- 6 How costly is it to communicate between threads?

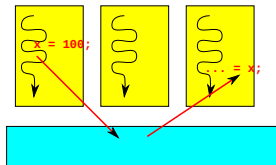
# Shared memory

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



# Shared memory

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

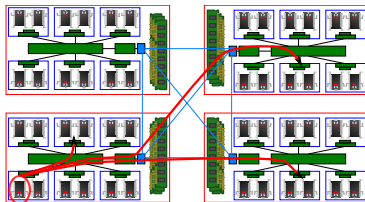


- ordinary load/store instructions accomplish this (*hardware shared memory*)
- this should not be taken for granted; processors have *caches* and a single address may be cached by multiple cores/sockets



# Shared memory

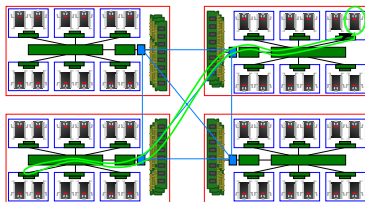
- $\Rightarrow$  processors sharing memory are running a complex, *cache coherence protocol* to accomplish this
- roughly,
  - 1 a write to an address by a processor “invalidates” all other cache lines holding the address, so that no caches hold “stale” values





# Shared memory

- $\Rightarrow$  processors sharing memory are running a complex, *cache coherence protocol* to accomplish this
- roughly,
  - ① a write to an address by a processor “invalidates” all other cache lines holding the address, so that no caches hold “stale” values
  - ② 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* (■), *Shared* (■), *Invalid* (■)

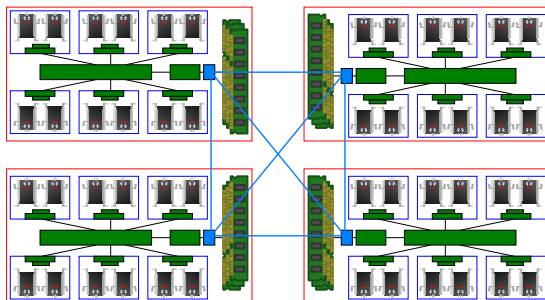
# An example protocol : the MSI protocol

- each line of a cache is in one of the following states

*Modified* (■), *Shared* (■), *Invalid* (■)

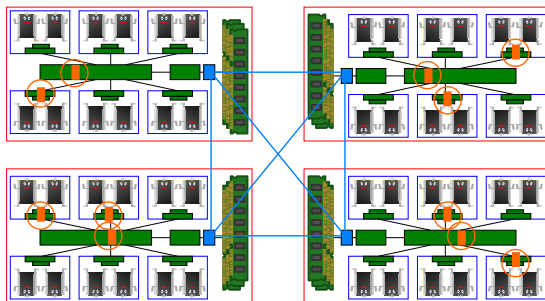
- Modified (■)  $\iff$  you can **read and write** the line without invoking a transaction
- Shared (■)  $\iff$  you can **read but not write** the line without invoking a transaction
- Invalid (■)  $\iff$  you can **neither read nor write** the line without invoking a transaction

# An example protocol : the MSI protocol



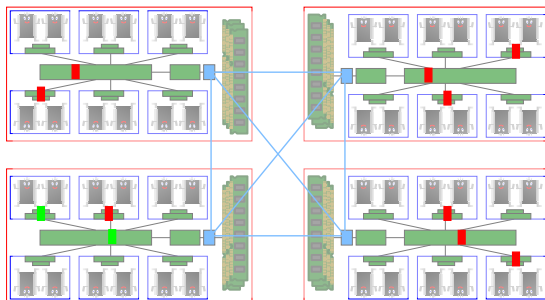
# An example protocol : the MSI protocol

- a single address may be cached in multiple caches (lines)



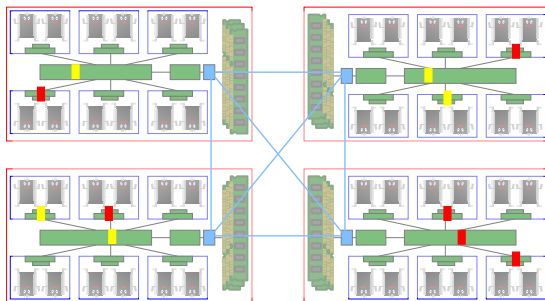
# An example protocol : the MSI protocol

- a single address may be cached in multiple caches (lines)
- $\Rightarrow$  there are only two legitimate states for each line
  - 1 one Modified (*owner*) + others Invalid (—, ■, —, —, —, ...)



# 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
  - 1 one Modified (*owner*) + others Invalid (—, ■, —, —, —, ...)
  - 2 no Modified (■, —, ■, ■, —, ...)



# 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:











	Modified	Shared	Invalid
read	hit	hit	read miss
write	hit	write miss	read miss; write miss



# 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:





















	Modified	Shared	Invalid
read	hit	hit	read miss
write	hit	write miss	read miss; write miss

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

# 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:

	Modified	Shared	Invalid
read	hit	hit	read miss
write	hit	write miss	read miss; write miss

- read miss:  $\rightarrow$ 
  - there may be a cache holding it in Modified state (*owner*)
  - searches for the owner and if found, downgrade it to Shared
  - , , , [], , ...  $\Rightarrow$  , , , [], , ...
- write miss:  $\rightarrow$ 
  - there may be caches holding it in Shared state (*sharer*)
  - searches for sharers and downgrade them to Invalid
  - , , , [], , ...  $\Rightarrow$  , , , [], , ...

# 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)

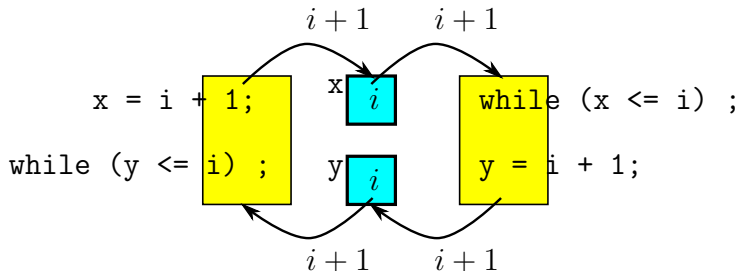
# How to measure communication latency?

- measure “ping-pong” latency between two threads

```
1 volatile long x = 0;  
2 volatile long y = 0;
```

```
1 (ping thread)  
2 for (i = 0; i < n; i++) {  
3   x = i + 1;  
4   while (y <= i) ;  
5 }
```

```
1 (pong thread)  
2 for (i = 0; i < n; i++) {  
3   while (x <= i) ;  
4   y = i + 1;  
5 }
```



# Environment

- Skylake X Gold 6130 (“big” partition of the IST cluster)
- 2 hardware threads  $\times$  16 cores  $\times$  4 sockets (= 128 processors seen by OS)
- ensure variables  $\mathbf{x}$  and  $\mathbf{y}$  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 reference clocks) between processor  $i$  and  $j$

src	dest	latency
0	1-15	$\approx 800$
0	16-63	$\approx 1100$
0	64	$\approx 110$
0	65-79	$\approx 450$
0	80-127	$\approx 1100$

- a beautiful pattern emerges which is obviously telling



# Result

- e.g., which processor is “close” to processor 0?
  - 64 is closest
  - 1-15 and 65-79 are close
  - 16-63 and 80-127 are farthest
- a natural interpretation
  - $x$  and  $(x + 64)$  are two hardware threads on a core
  - 0-15 (and 65-79) are the 16 physical cores (32 hwts) on a socket
  - others are on different sockets

# What they imply to parallel algorithms?

- you do not want to have many threads concurrently updating the same data
- remember SpMV COO?

```
1 // assume inside #pragma omp parallel
2 ...
3 #pragma omp for
4 for (k = 0; k < A.nnz; k++) {
5     i,j,Aij = A.elems[k];
6     #pragma omp atomic
7     y[i] += Aij * x[j];
8 }
```

- `y[i] +=` may be costing 1000 cycles when its single-thread execution would take just dozens of cycles

# 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

## Summary (3): inter processor communication

- 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