Cache Memories, Cache Complexity

Marc Moreno Maza

Ontario Research Center for Computer Algebra
Departments of Computer Science and Mathematics
University of Western Ontario, Canada

CS4402 - CS9535, January 31, 2024

Cache Memories, Cache Complexity

Marc Moreno Maza

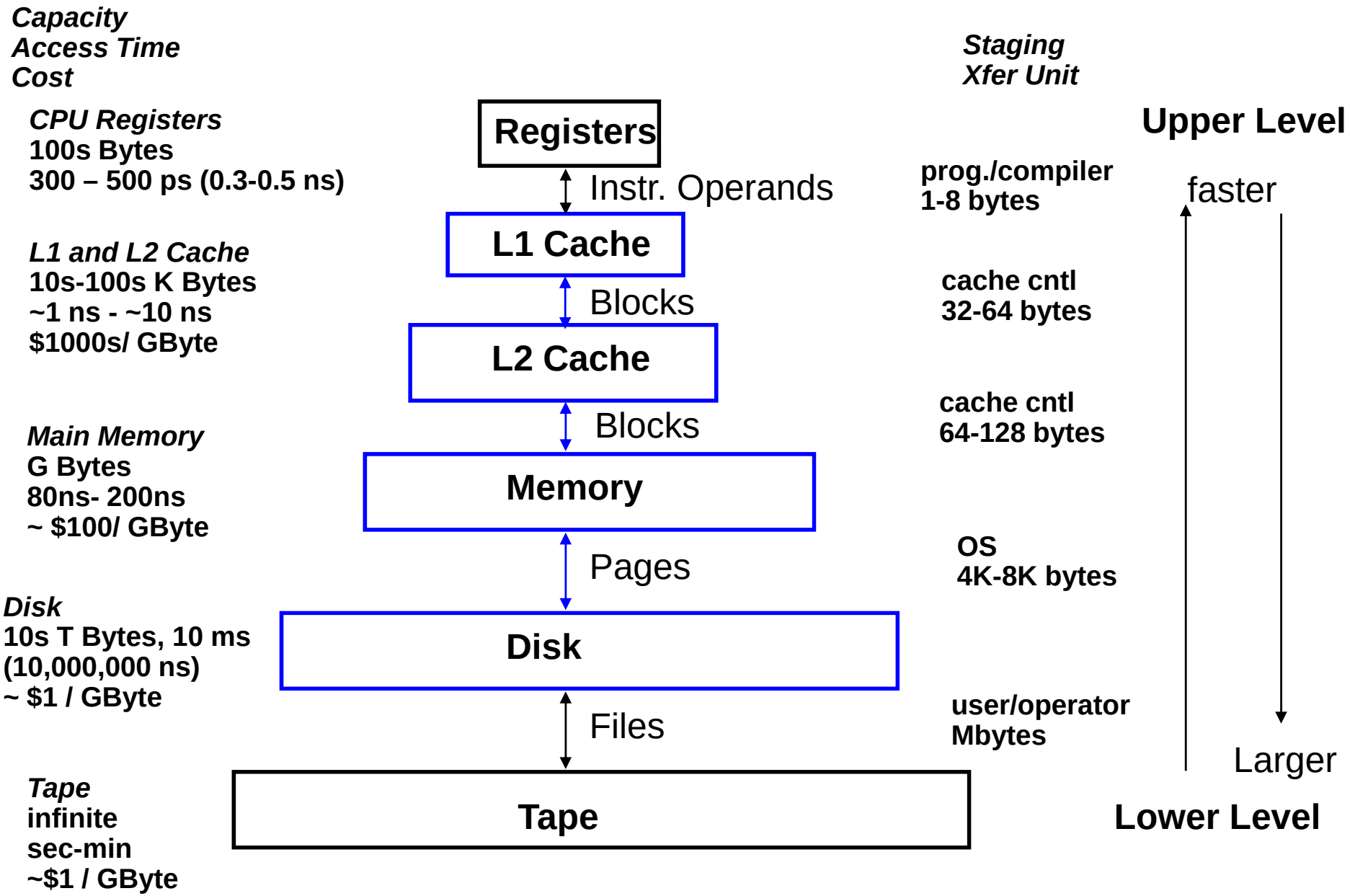
Ontario Research Center for Computer Algebra
Departments of Computer Science and Mathematics
University of Western Ontario, Canada

CS4402 - CS9535, January 31, 2024

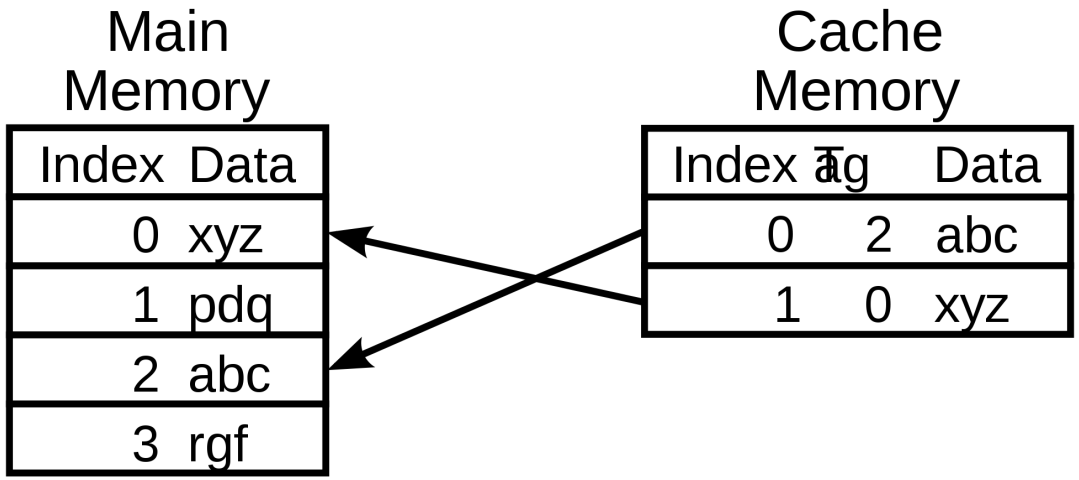
Plan

Outline

Outline

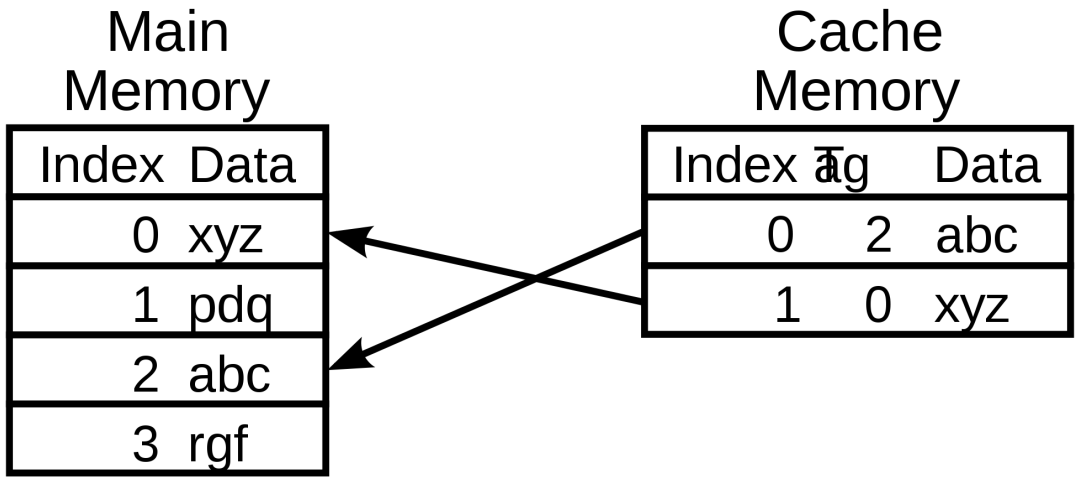

Capacity
Access Time
Cost
CPU Registers
100s Bytes
300 – 500 ps (0.3-0.5 ns)
L1 and L2 Cache
10s-100s K Bytes
~1 ns - ~10 ns
$1000s/ GByte
Main Memory
G Bytes
80ns- 200ns
~ $100/ GByte
Disk
10s T Bytes, 10 ms
(10,000,000 ns)
~ $1 / GByte
Tape
infinite
sec-min
~$1 / GByte
Registers
Instr. Operands
L1 Cache
Blocks
L2 Cache
Blocks
Memory
Pages
Disk
Files
Tape
Staging
Xfer Unit
prog./compiler
1-8 bytes
cache cntl
32-64 bytes
cache cntl
64-128 bytes
OS
4K-8K bytes
user/operator
Mbytes
Upper Level
faster
Larger
Lower Level

CPU Cache (1/6)



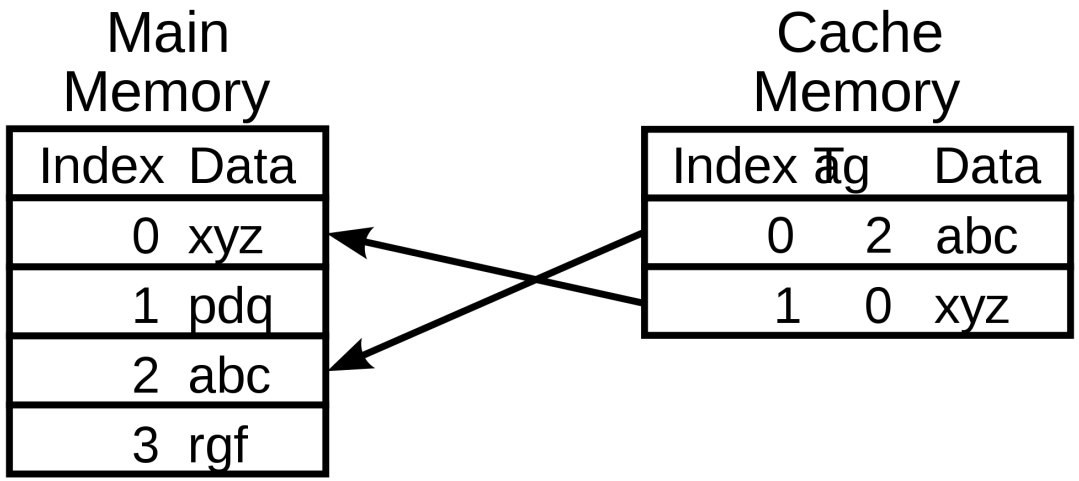
- A CPU cache is an auxiliary memory which is smaller, faster memory than the main memory and which stores copies of the main memory locations that are expectedly frequently used.

CPU Cache (1/6)



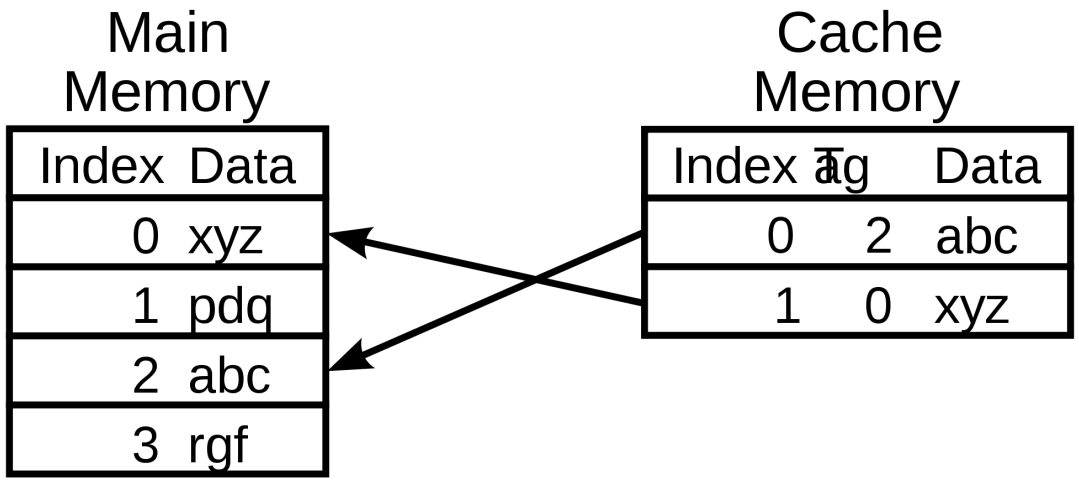
- A CPU cache is an auxiliary memory which is smaller, faster memory than the main memory and which stores copies of the main memory locations that are expectedly frequently used.
- Most modern desktop and server CPUs have at least three independent caches: the data cache, the instruction cache and the translation look-aside buffer.

CPU Cache (2/6)



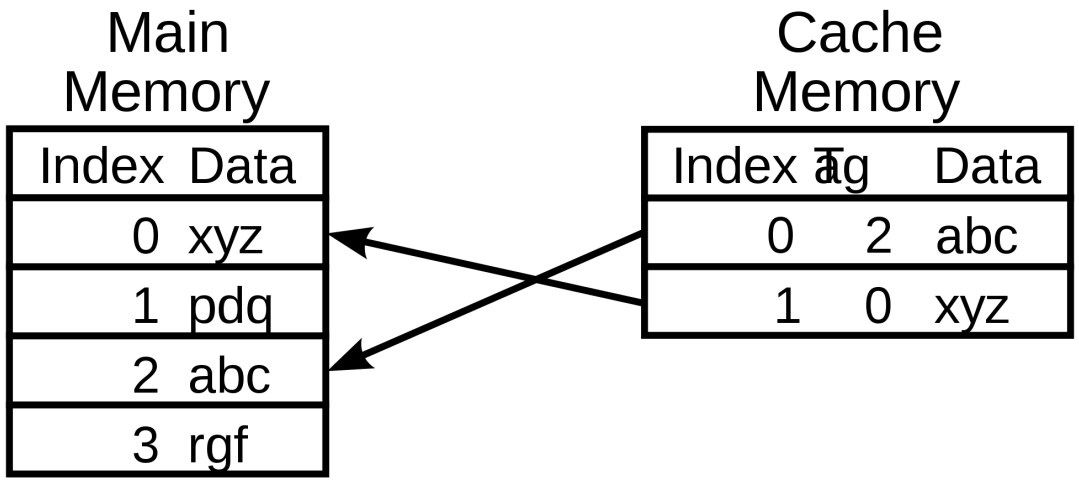
- Each location in each memory (main or cache) has

CPU Cache (2/6)



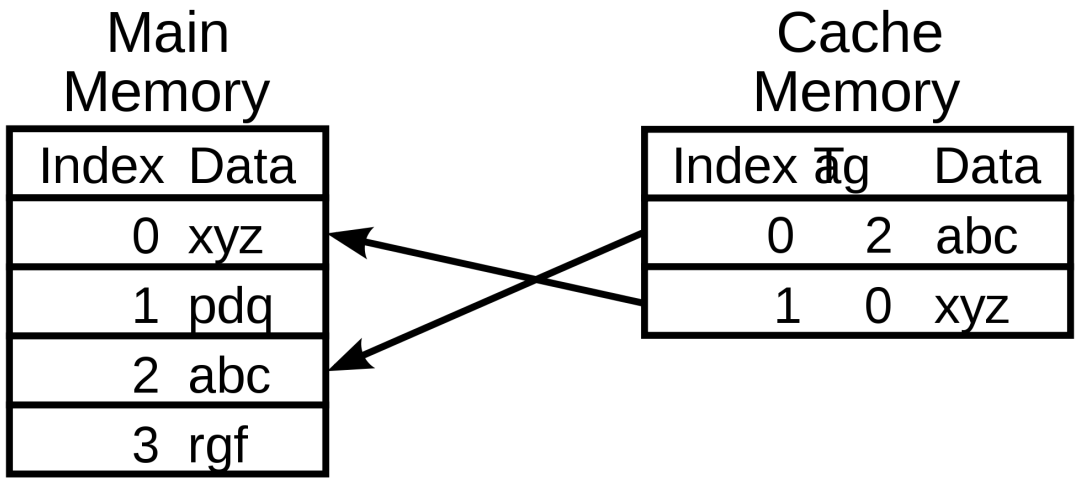
- Each location in each memory (main or cache) has
  - ↳ a cache line which ranges between 8 and 512 bytes in size, while a datum requested by a CPU instruction ranges between 1 and 16,

CPU Cache (2/6)



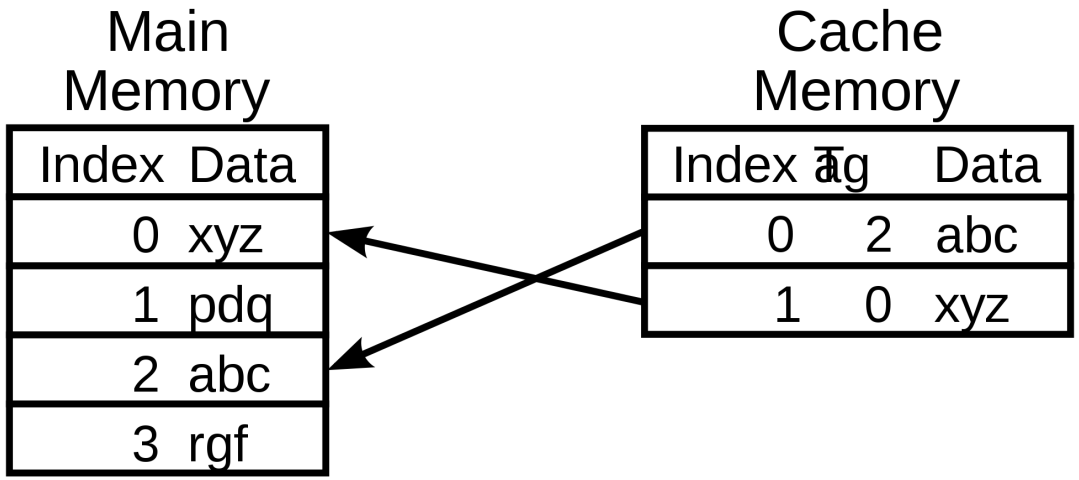
- Each location in each memory (main or cache) has
  - ↳ a cache line which ranges between 8 and 512 bytes in size, while a datum requested by a CPU instruction ranges between 1 and 16,
  - ↳ a unique index (called address in the case of the main memory).

CPU Cache (2/6)



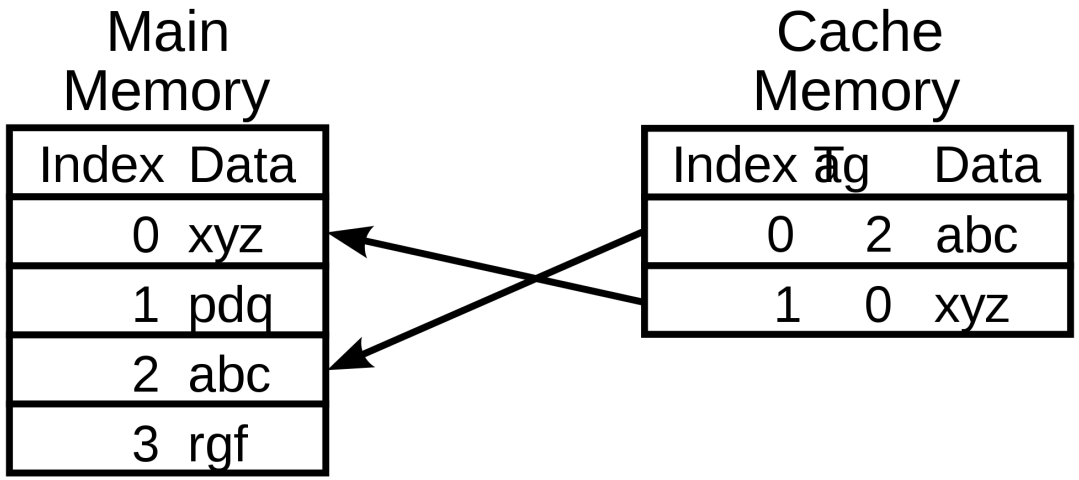
- Each location in each memory (main or cache) has
  - ↳ a cache line which ranges between 8 and 512 bytes in size, while a datum requested by a CPU instruction ranges between 1 and 16,
  - ↳ a unique index (called address in the case of the main memory).
- In the cache, each location has also a tag (storing the address of the corresponding cached datum).

CPU Cache (3/6)



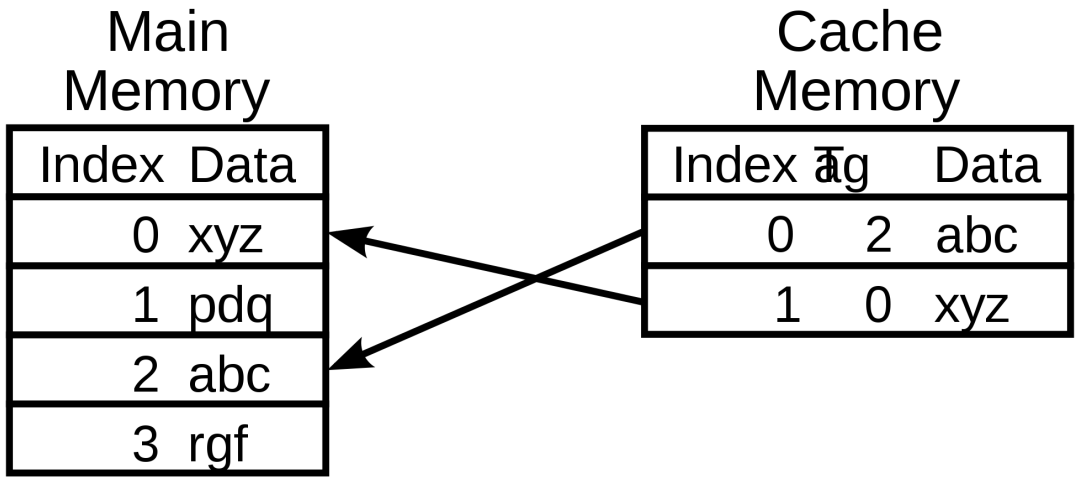
- When the CPU needs to read or write a location, it checks the cache:

CPU Cache (3/6)



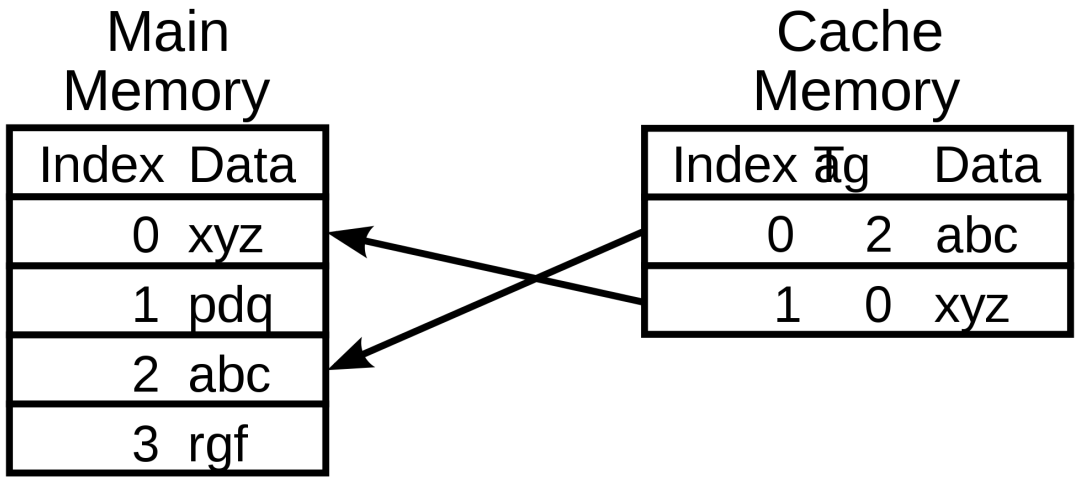
- When the CPU needs to read or write a location, it checks the cache:
  - ↳ if it finds it there, we have a cache hit

CPU Cache (3/6)



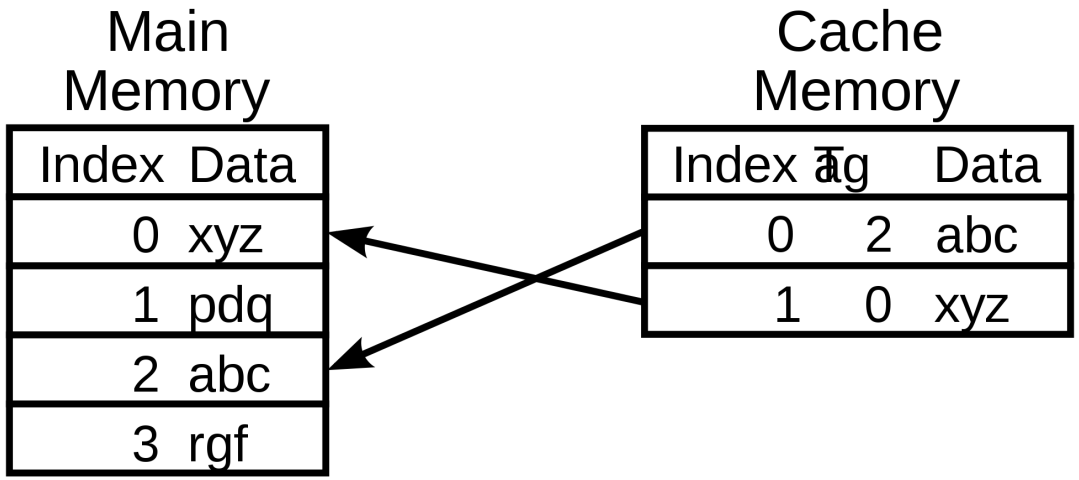
- When the CPU needs to read or write a location, it checks the cache:
  - ↳ if it finds it there, we have a cache hit
  - ↳ if not, we have a cache miss and (in most cases) the processor needs to create a new entry in the cache.

CPU Cache (3/6)



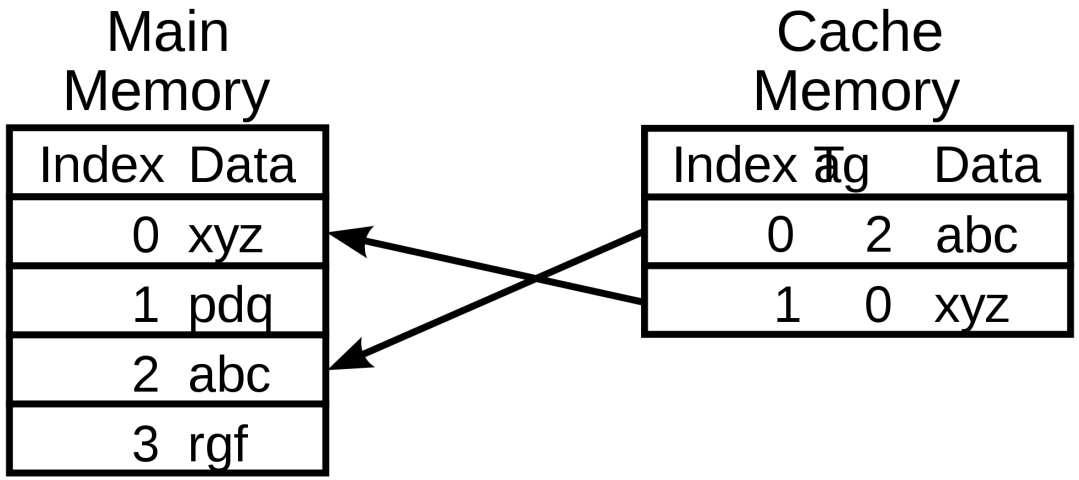
- When the CPU needs to read or write a location, it checks the cache:
  - ↳ if it finds it there, we have a cache hit
  - ↳ if not, we have a cache miss and (in most cases) the processor needs to create a new entry in the cache.
- Making room for a new entry requires a replacement policy: the Least Recently Used (LRU) discards the least recently used items first; this requires to use age bits.

CPU Cache (4/6)



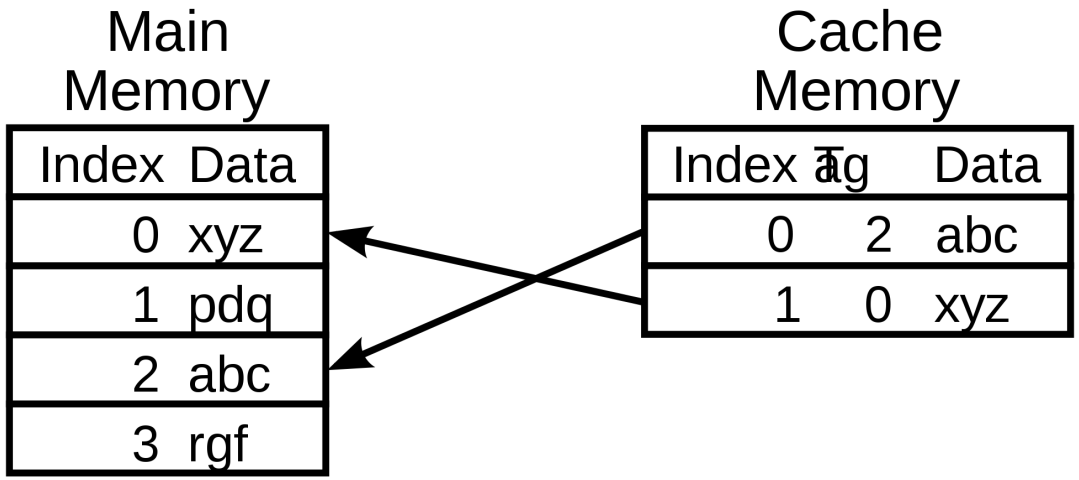
- Modifying data in the cache requires a write policy for updating the main memory

CPU Cache (4/6)



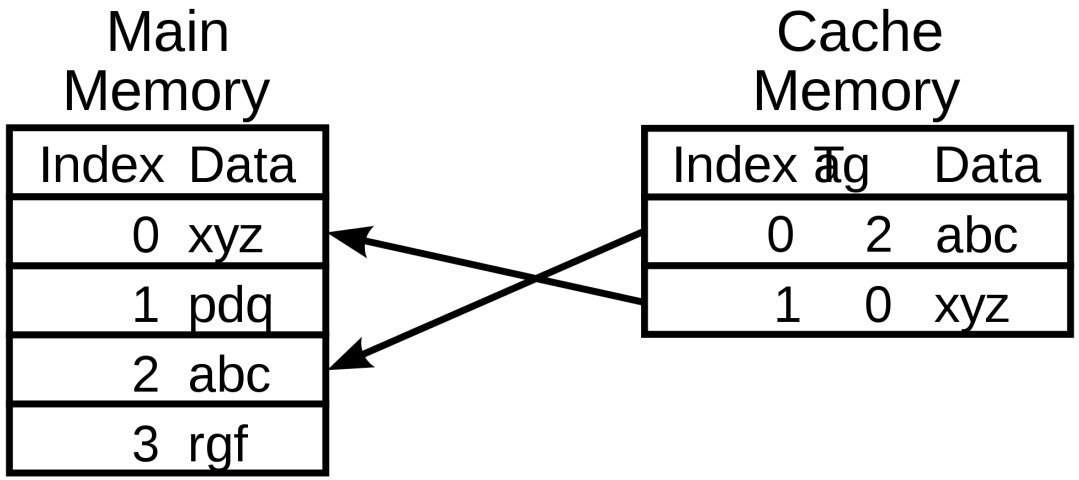
- Modifying data in the cache requires a write policy for updating the main memory
  - write-through cache: writes are immediately mirrored to main memory

CPU Cache (4/6)



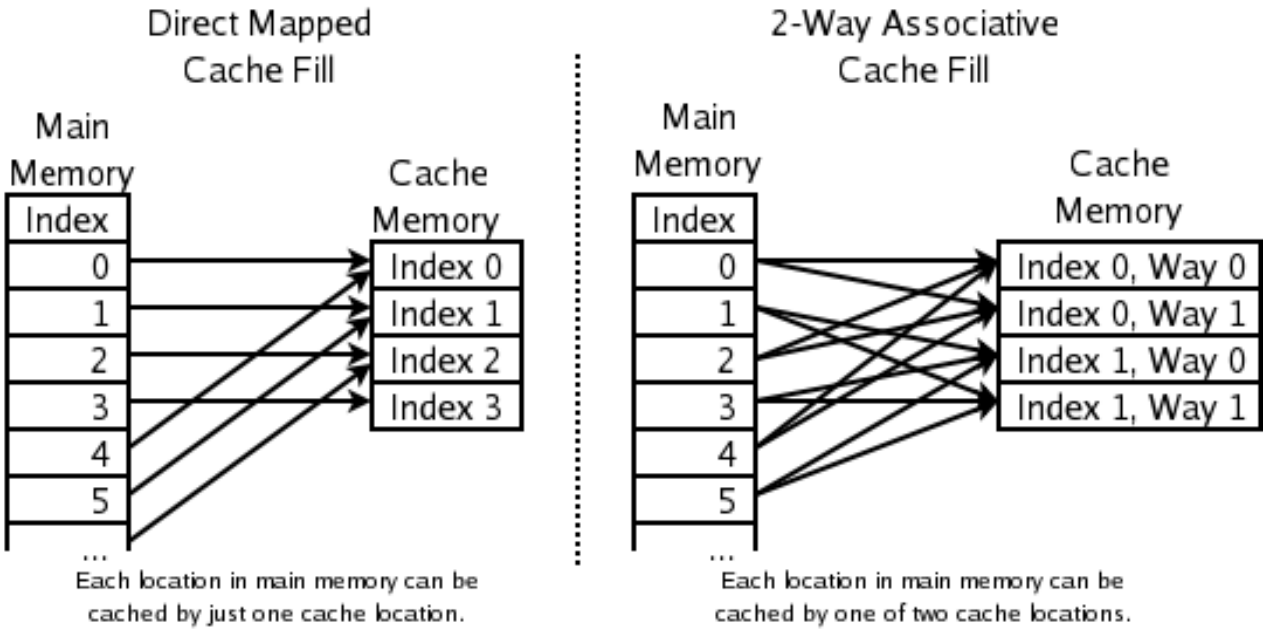
- Modifying data in the cache requires a write policy for updating the main memory
  - write-through cache: writes are immediately mirrored to main memory
  - write-back cache: the main memory is mirrored when that data is evicted from the cache

CPU Cache (4/6)



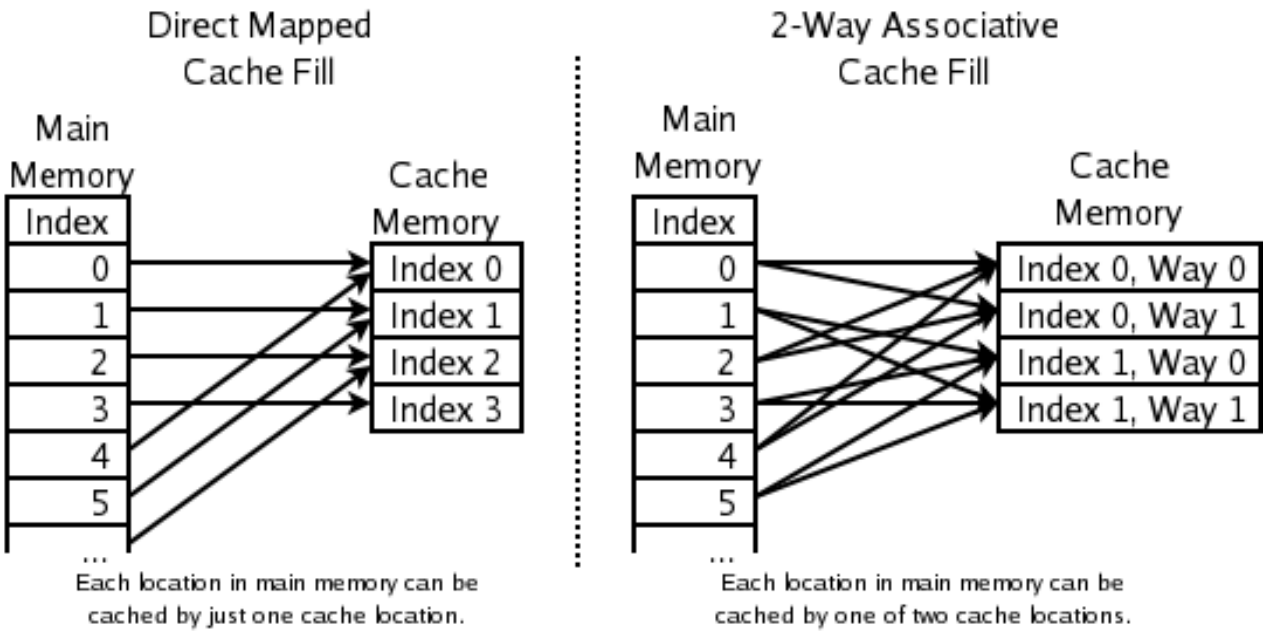
- Modifying data in the cache requires a write policy for updating the main memory
  - write-through cache: writes are immediately mirrored to main memory
  - write-back cache: the main memory is mirrored when that data is evicted from the cache
- The cached copy may become out-of-date or stale, if other processors modify the original entry in the main memory.

CPU Cache (5/6)



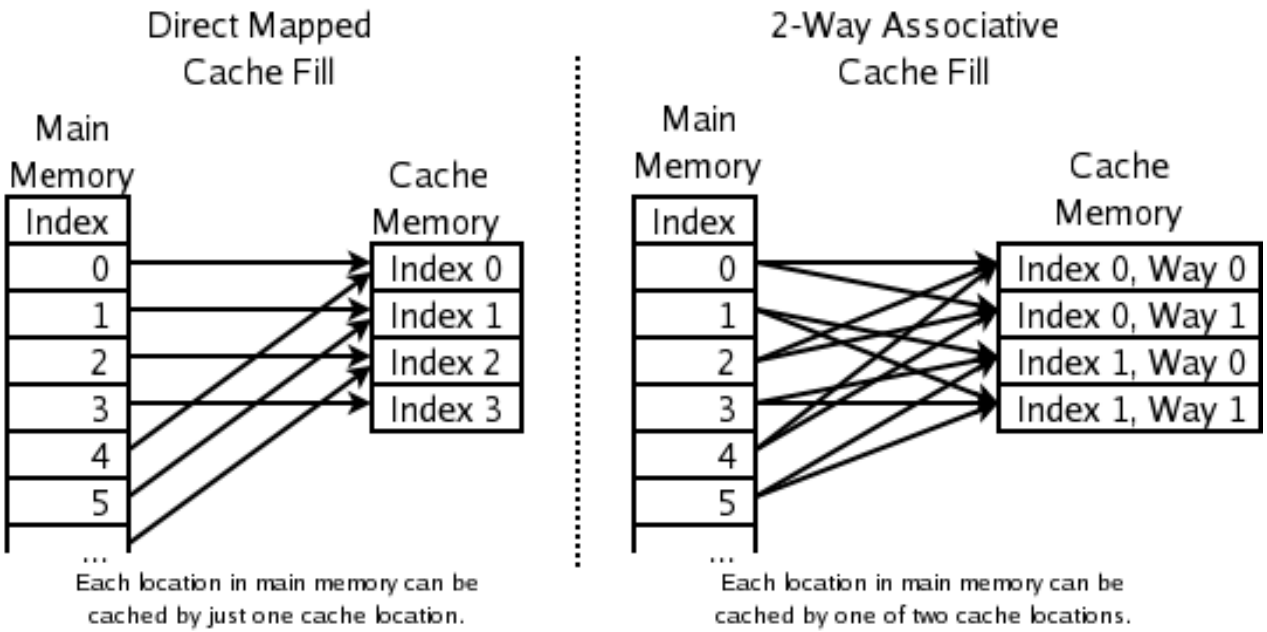
- The replacement policy decides, where in the cache, a copy of a particular entry of main memory will go:

CPU Cache (5/6)



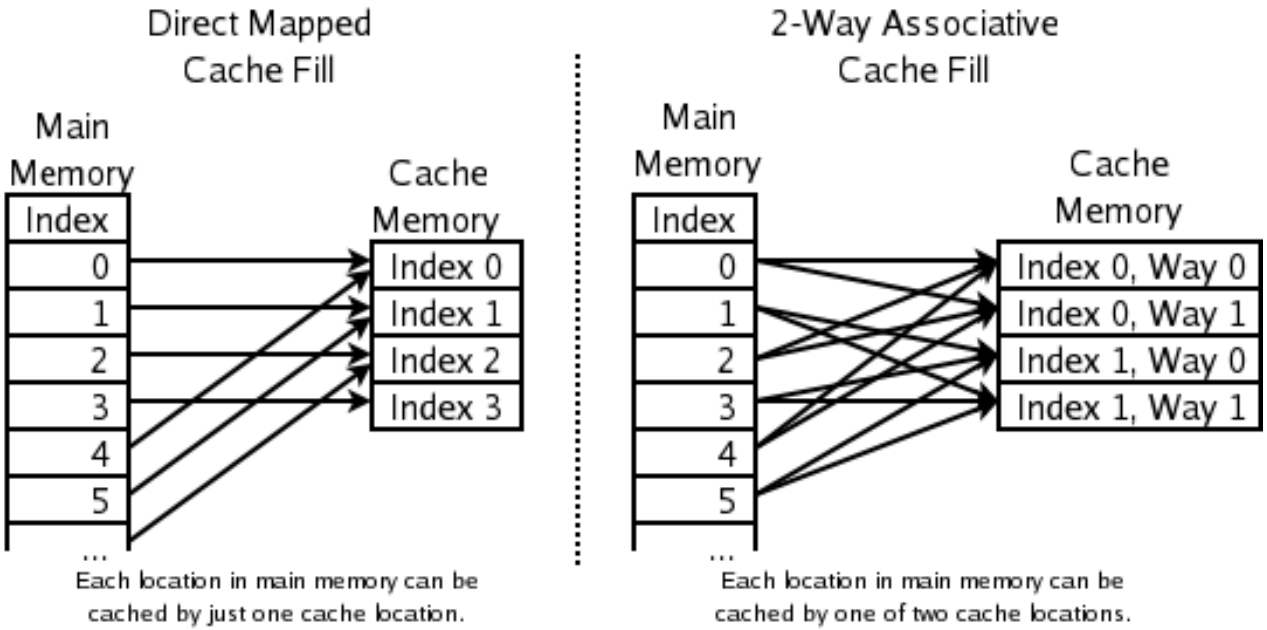
- The replacement policy decides, where in the cache, a copy of a particular entry of main memory will go:
  - fully associative: any entry in the cache can hold it

CPU Cache (5/6)

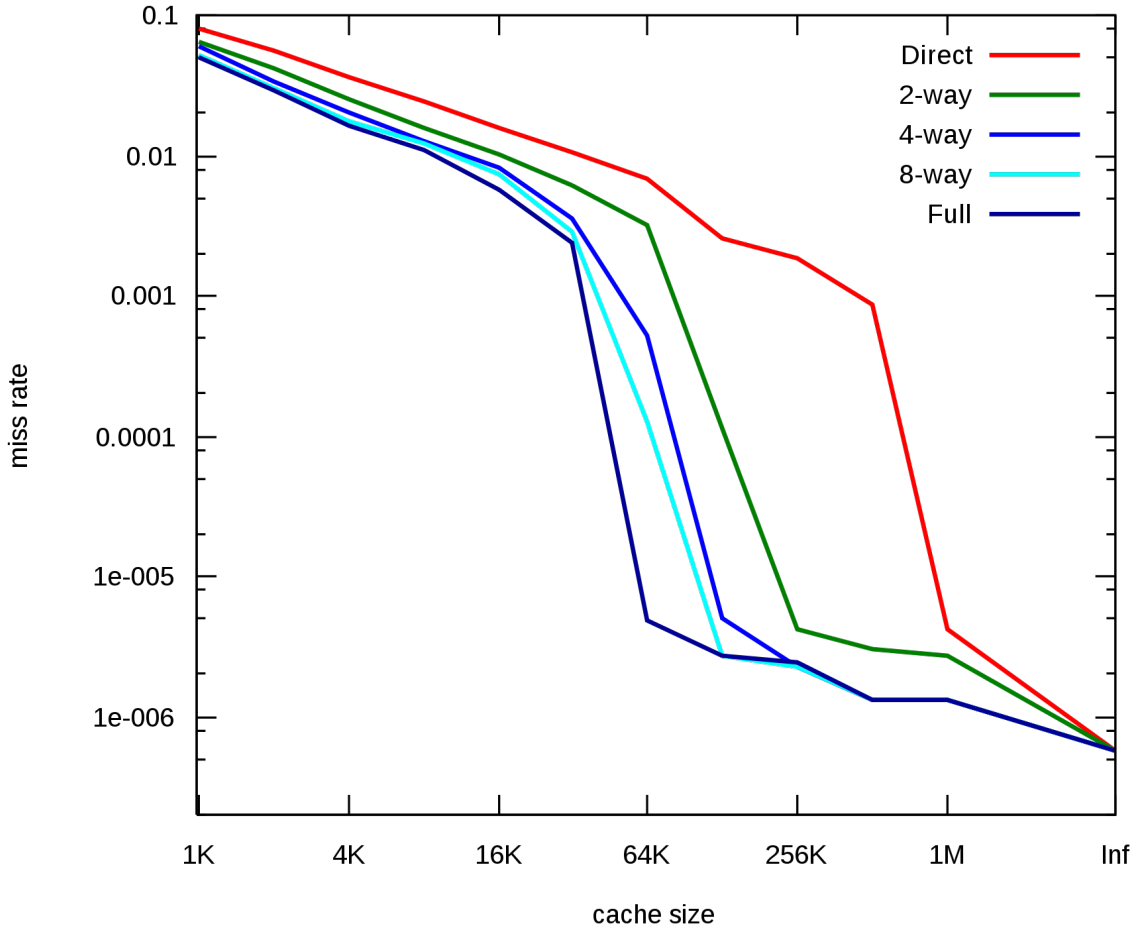


- The replacement policy decides, where in the cache, a copy of a particular entry of main memory will go:
  - fully associative: any entry in the cache can hold it
  - direct mapped: only one possible entry in the cache can hold it

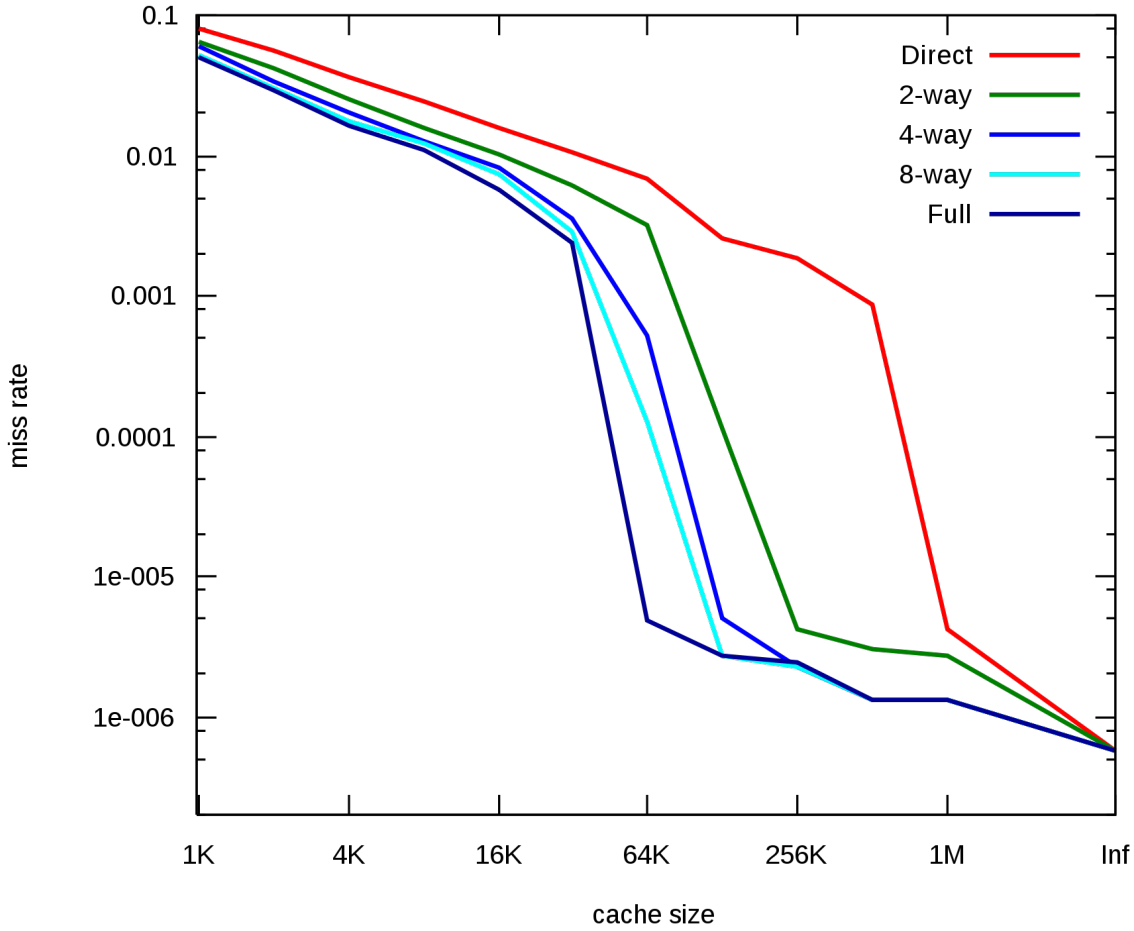
CPU Cache (5/6)



- The replacement policy decides, where in the cache, a copy of a particular entry of main memory will go:
  - fully associative: any entry in the cache can hold it
  - direct mapped: only one possible entry in the cache can hold it
  - N-way set associative: N possible entries can hold it



- *Cache Performance for SPEC CPU2000* by J.F. Cantin and M.D. Hill.



- *Cache Performance for SPEC CPU2000* by J.F. Cantin and M.D. Hill.
- The SPEC CPU suites are collections of compute-intensive, non-trivial programs used to evaluate the performance of a computer's CPU, memory system, and compilers (`http://www.spec.org/osg`).

Cache issues

- **Cold miss:** The first time the data is available.

Cache issues

- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.

Cache issues

- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.
- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large.

Cache issues

- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.
- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large. Cure: Reorganize the data access such that *reuse* occurs before eviction.

Cache issues

- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.
- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large. Cure: Reorganize the data access such that *reuse* occurs before eviction.
- **Conflict miss:** Multiple data items mapped to the same location with eviction before cache is full.

Cache issues

- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.
- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large. Cure: Reorganize the data access such that *reuse* occurs before eviction.
- **Conflict miss:** Multiple data items mapped to the same location with eviction before cache is full. Cure: Rearrange data and/or pad arrays.

Cache issues

- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.
- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large. Cure: Reorganize the data access such that *reuse* occurs before eviction.
- **Conflict miss:** Multiple data items mapped to the same location with eviction before cache is full. Cure: Rearrange data and/or pad arrays.
- **True sharing miss:** Occurs when a thread in another processor wants the same data.

Cache issues

- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.
- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large. Cure: Reorganize the data access such that *reuse* occurs before eviction.
- **Conflict miss:** Multiple data items mapped to the same location with eviction before cache is full. Cure: Rearrange data and/or pad arrays.
- **True sharing miss:** Occurs when a thread in another processor wants the same data. Cure: Minimize sharing.

Cache issues

- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.
- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large. Cure: Reorganize the data access such that *reuse* occurs before eviction.
- **Conflict miss:** Multiple data items mapped to the same location with eviction before cache is full. Cure: Rearrange data and/or pad arrays.
- **True sharing miss:** Occurs when a thread in another processor wants the same data. Cure: Minimize sharing.
- **False sharing miss:** Occurs when another processor uses different data in the same cache line.

Cache issues

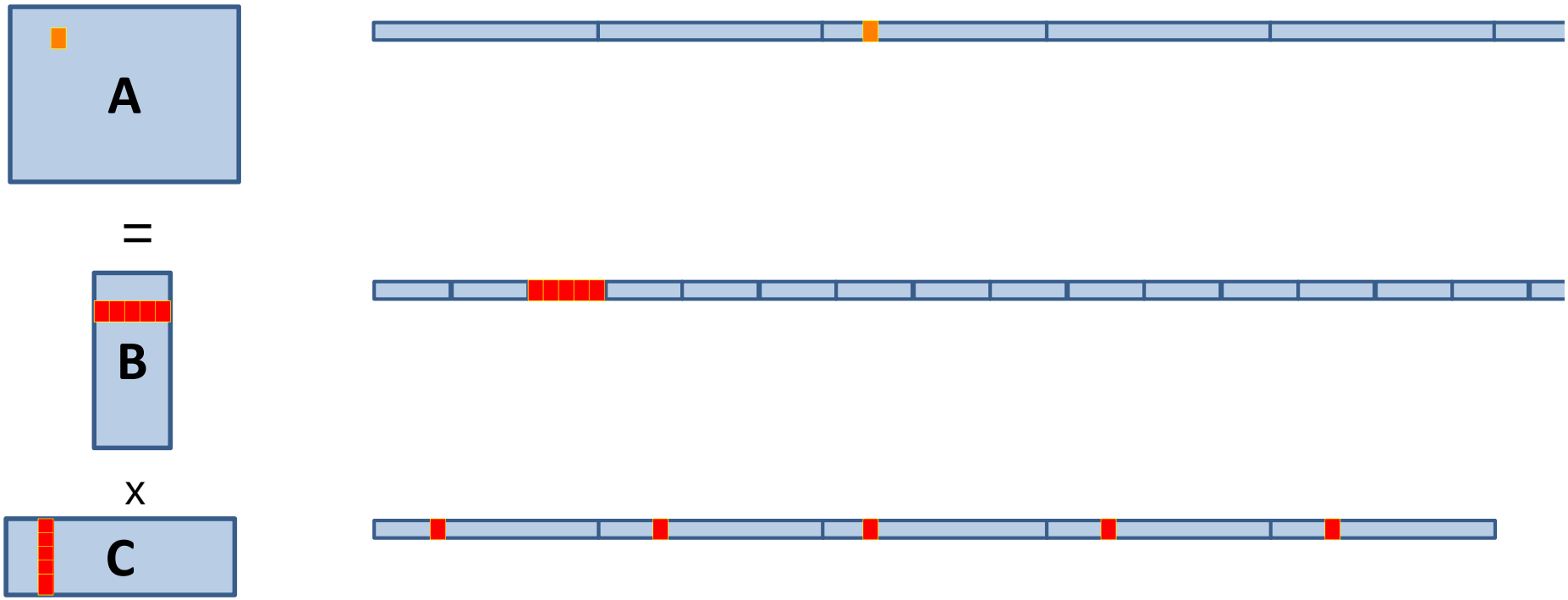
- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.
- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large. Cure: Reorganize the data access such that *reuse* occurs before eviction.
- **Conflict miss:** Multiple data items mapped to the same location with eviction before cache is full. Cure: Rearrange data and/or pad arrays.
- **True sharing miss:** Occurs when a thread in another processor wants the same data. Cure: Minimize sharing.
- **False sharing miss:** Occurs when another processor uses different data in the same cache line. Cure: Pad data.

Outline

A typical matrix multiplication C code

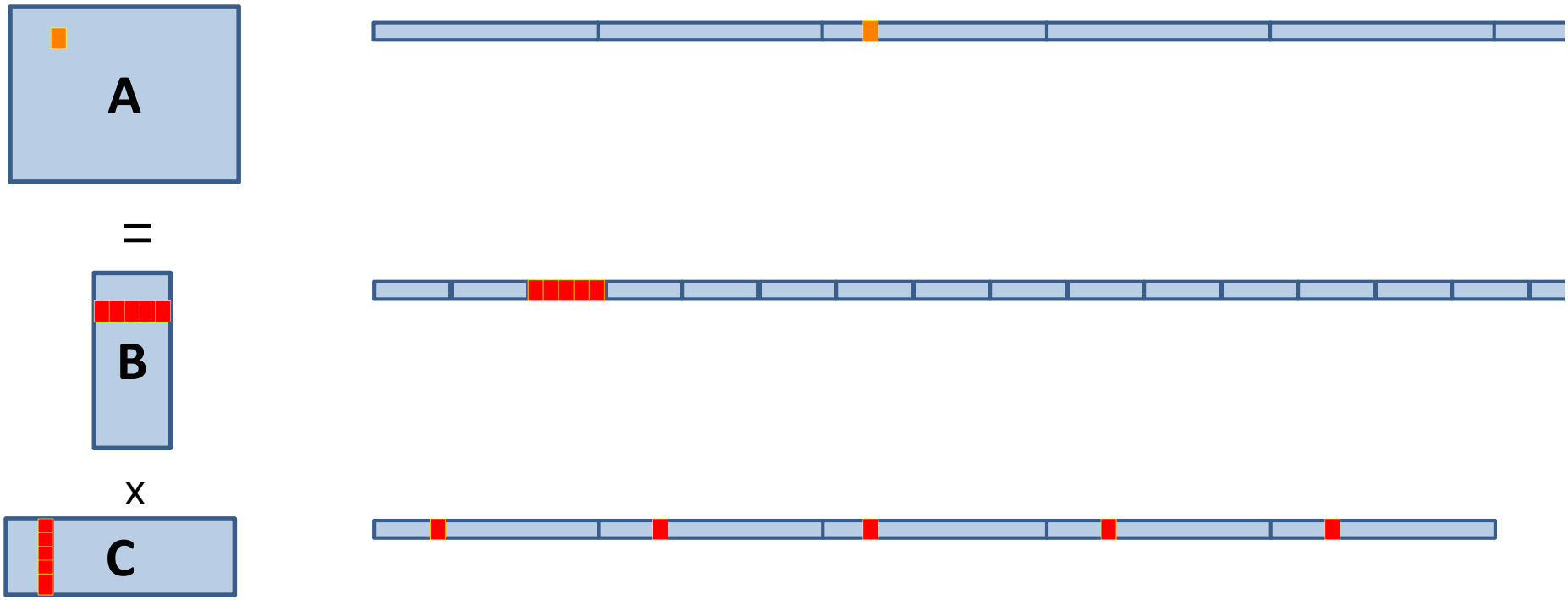
```
#define IND(A, x, y, d) A[(x)*(d)+(y)]
uint64_t testMM(const int x, const int y, const int z)
{
  double *A; double *B; double *C;
        long started, ended;
        float timeTaken;
        int i, j, k;
        srand(getSeed());
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
                    //  A[i][j] += B[i][k] * C[k][j];
                    IND(A,i,j,y) += IND(B,i,k,z) * IND(C,k,j,z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with matrix representation



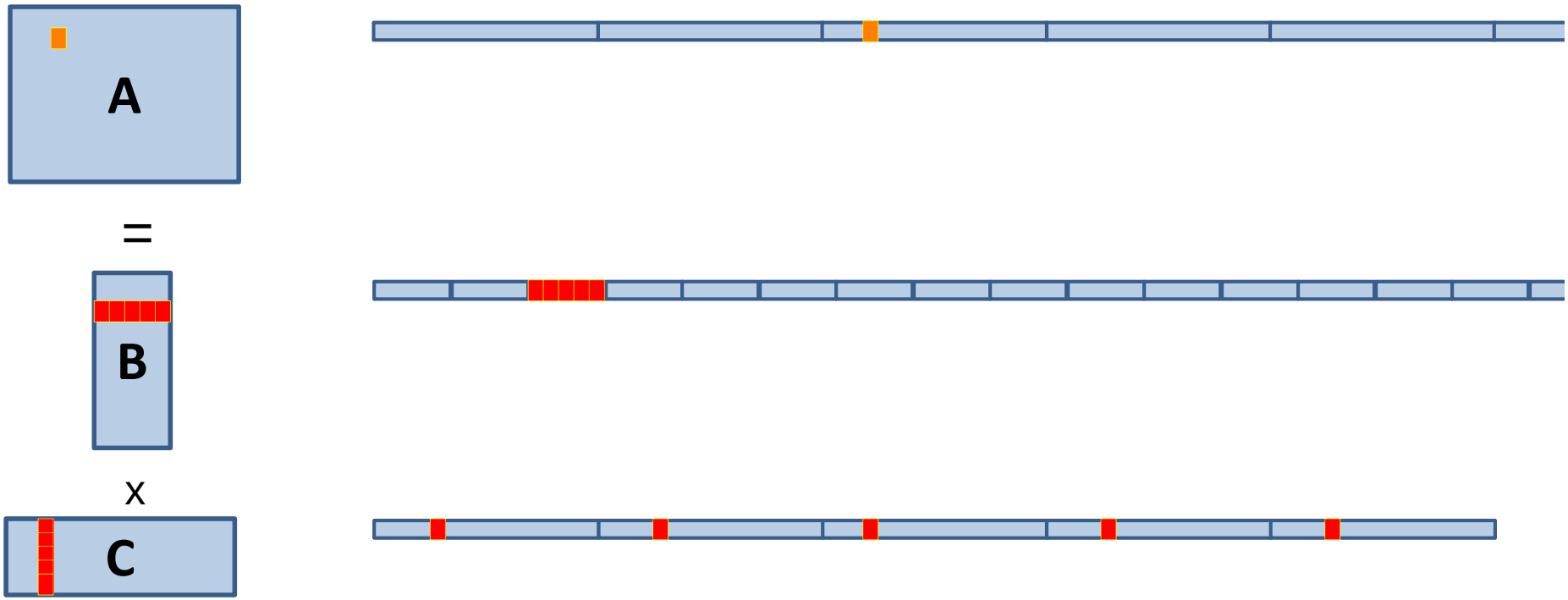
- The matrices `A`, `B`, `C` are stored in row-major layout.

Issues with matrix representation



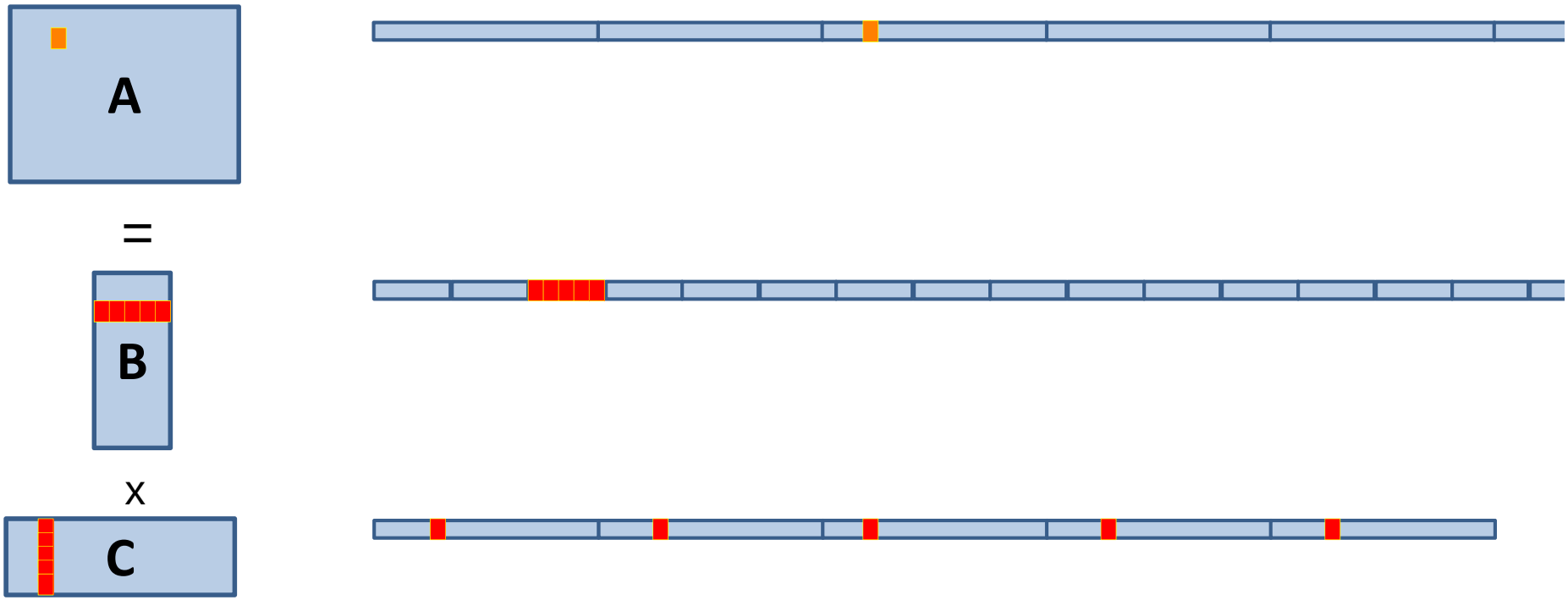
- The matrices `A`, `B`, `C` are stored in row-major layout.
- Consequently, memory accesses to `B` (but not `C`) are contiguous.

Issues with matrix representation



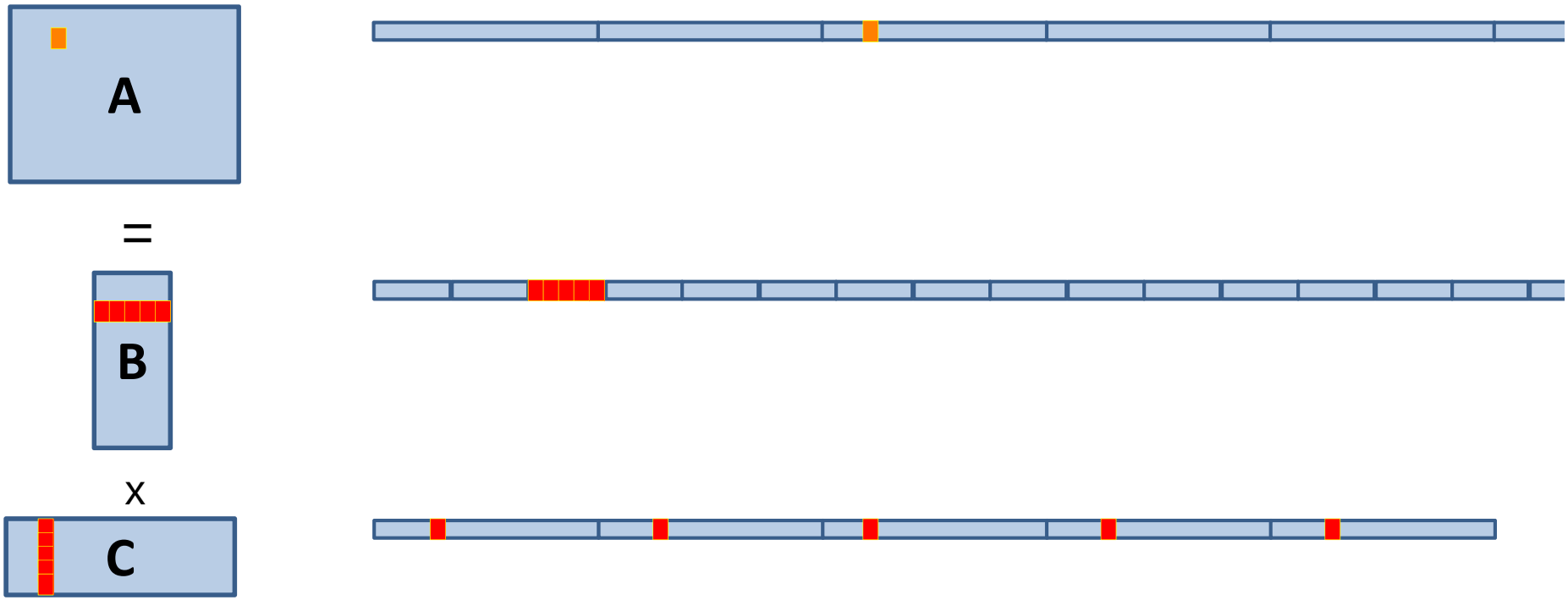
- The matrices `A`, `B`, `C` are stored in row-major layout.
- Consequently, memory accesses to `B` (but not `C`) are contiguous.
- Contiguous accesses are better:

Issues with matrix representation



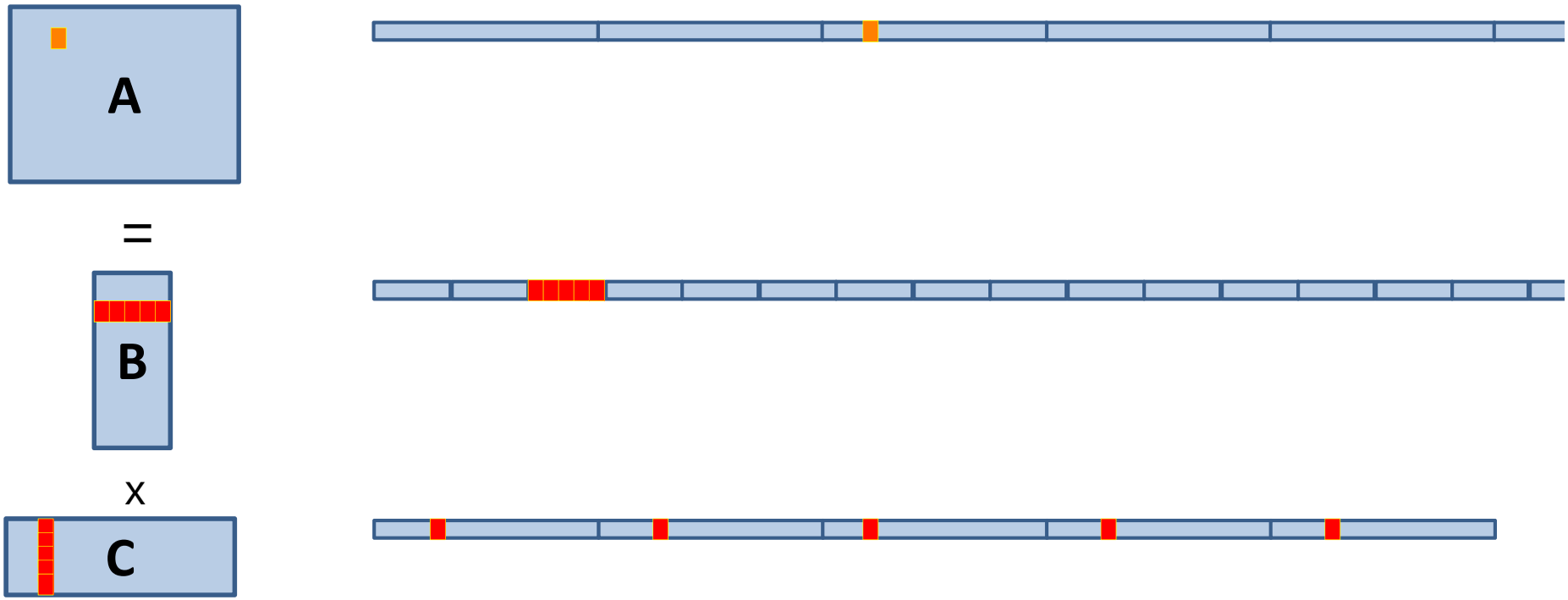
- The matrices `A`, `B`, `C` are stored in row-major layout.
- Consequently, memory accesses to `B` (but not `C`) are contiguous.
- Contiguous accesses are better:

Issues with matrix representation



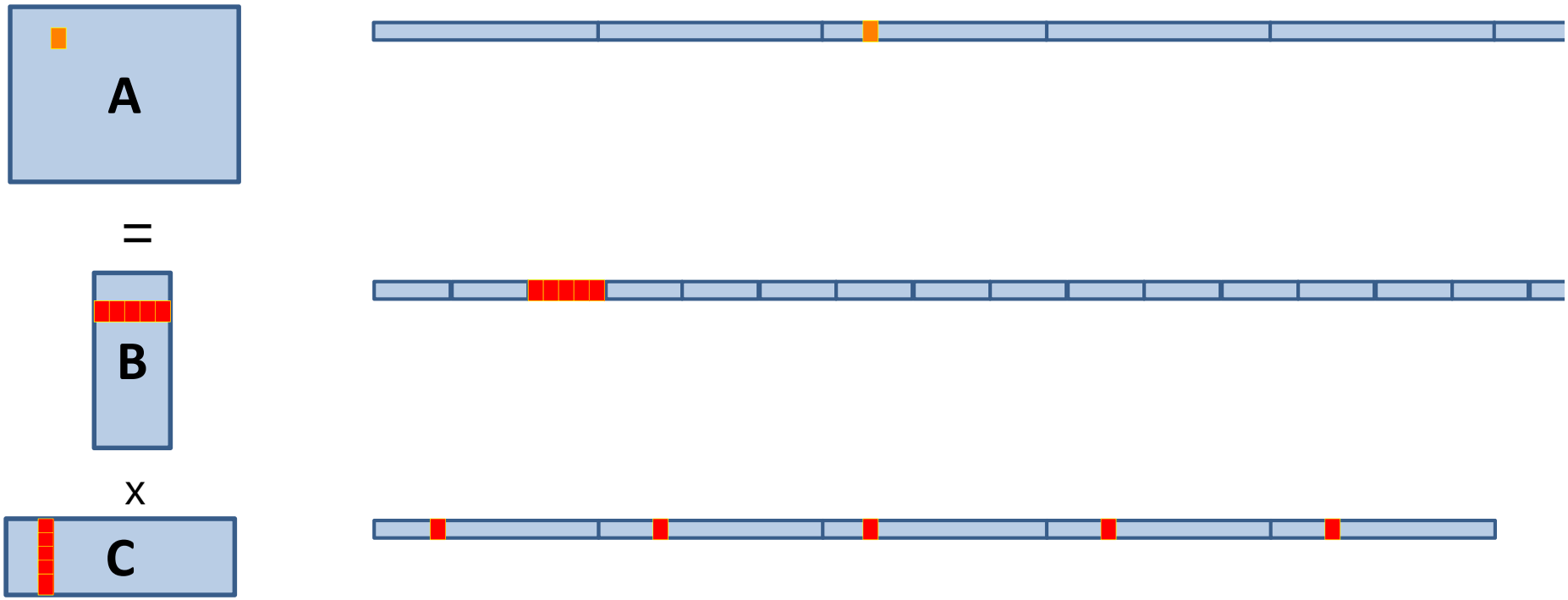
- The matrices `A`, `B`, `C` are stored in row-major layout.
- Consequently, memory accesses to `B` (but not `C`) are contiguous.
- Contiguous accesses are better:
  - ↳ Data fetch as cache line (Core 2 Duo: 64 byte per cache line)

Issues with matrix representation



- The matrices `A`, `B`, `C` are stored in row-major layout.
- Consequently, memory accesses to `B` (but not `C`) are contiguous.
- Contiguous accesses are better:
  - ↳ Data fetch as cache line (Core 2 Duo: 64 byte per cache line)
  - ↳ With contiguous data, a single cache fetch supports 8 reads of doubles.

Issues with matrix representation

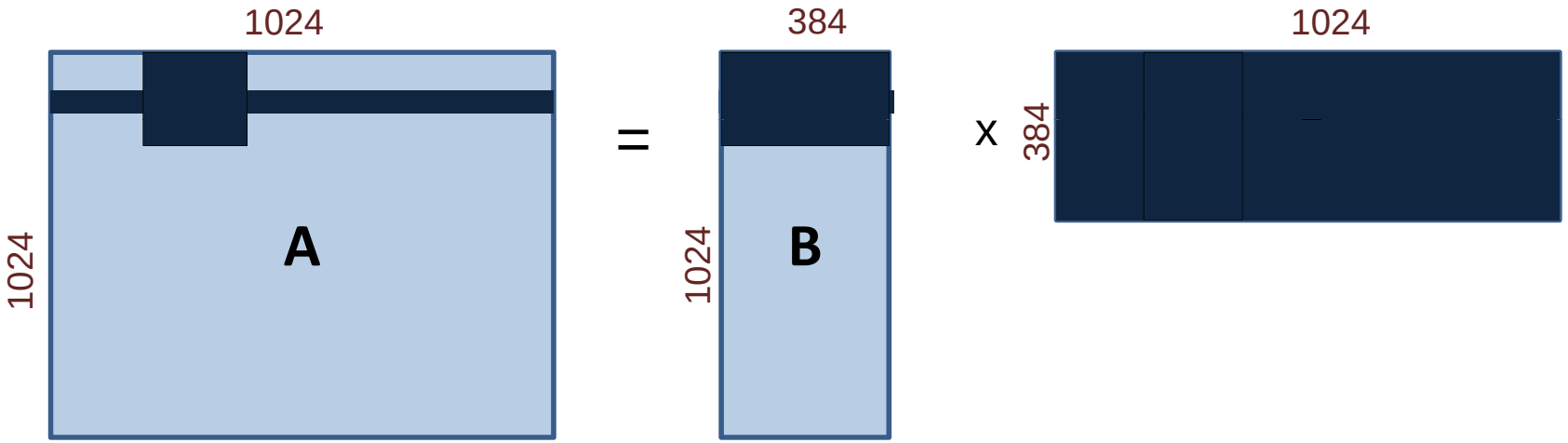


- The matrices `A`, `B`, `C` are stored in row-major layout.
- Consequently, memory accesses to `B` (but not `C`) are contiguous.
- Contiguous accesses are better:
  - ↳ Data fetch as cache line (Core 2 Duo: 64 byte per cache line)
  - ↳ With contiguous data, a single cache fetch supports 8 reads of doubles.
  - ↳ Transposing the matrix `C` should reduce L1 cache misses!

Transposing for optimizing spatial locality

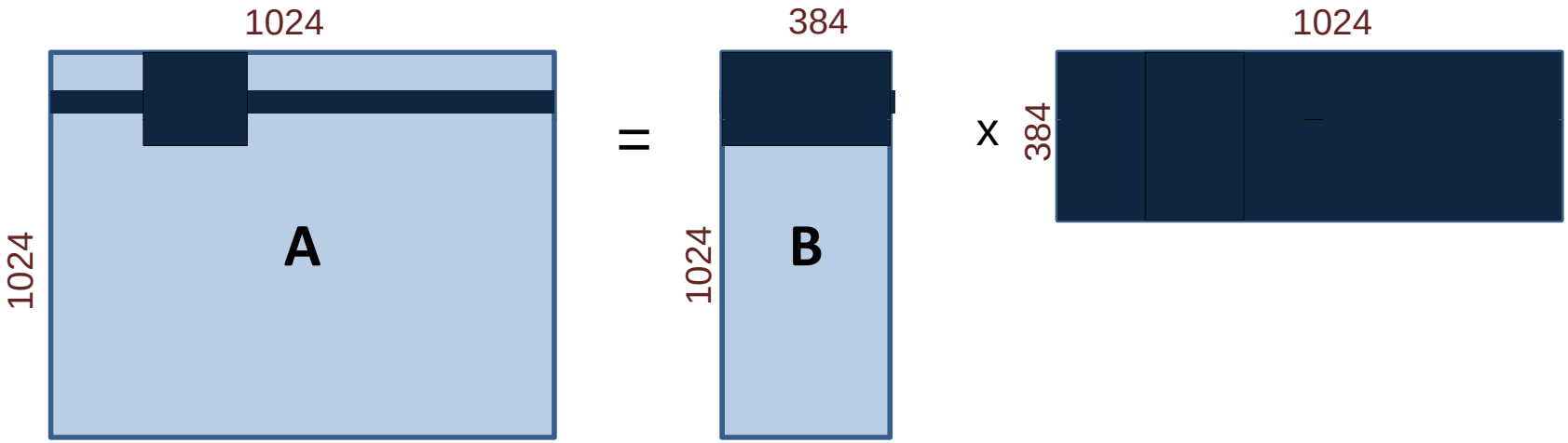
```
float testMM(const int x, const int y, const int z)
{
  double *A; double *B; double *C; double *Cx;
        long started, ended; float timeTaken; int i, j, k;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        Cx = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for(j =0; j < y; j++)
          for(k=0; k < z; k++)
            IND(Cx,j,k,z) = IND(C,k,j,y);
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
               IND(A, i, j, y) += IND(B, i, k, z) *IND(Cx, j, k, z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with data reuse



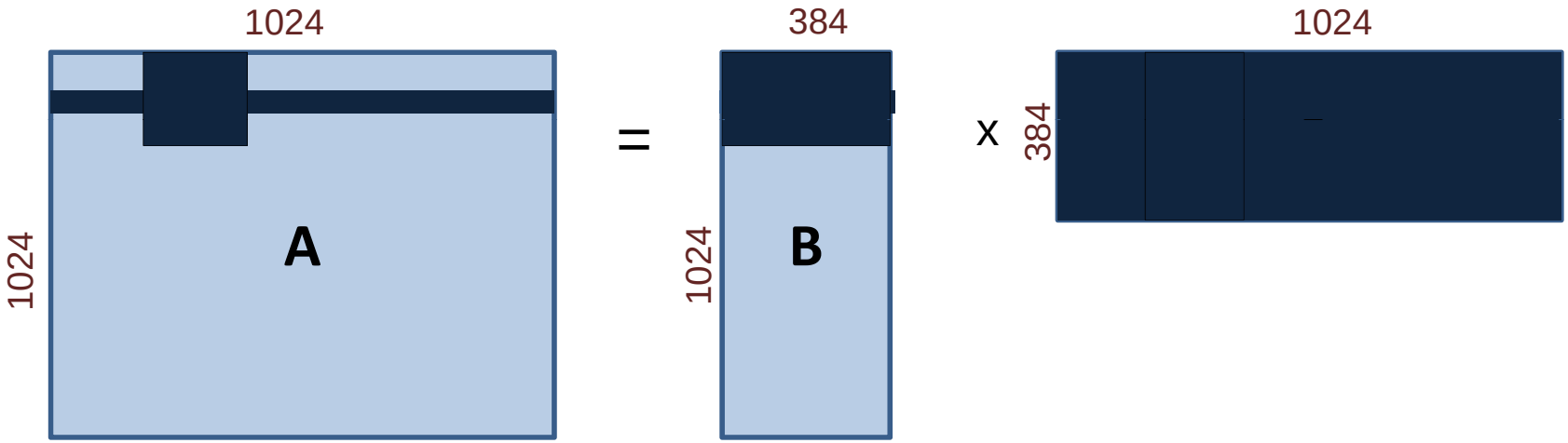
- Remember the formats of `A`, `B` and `C`.

Issues with data reuse



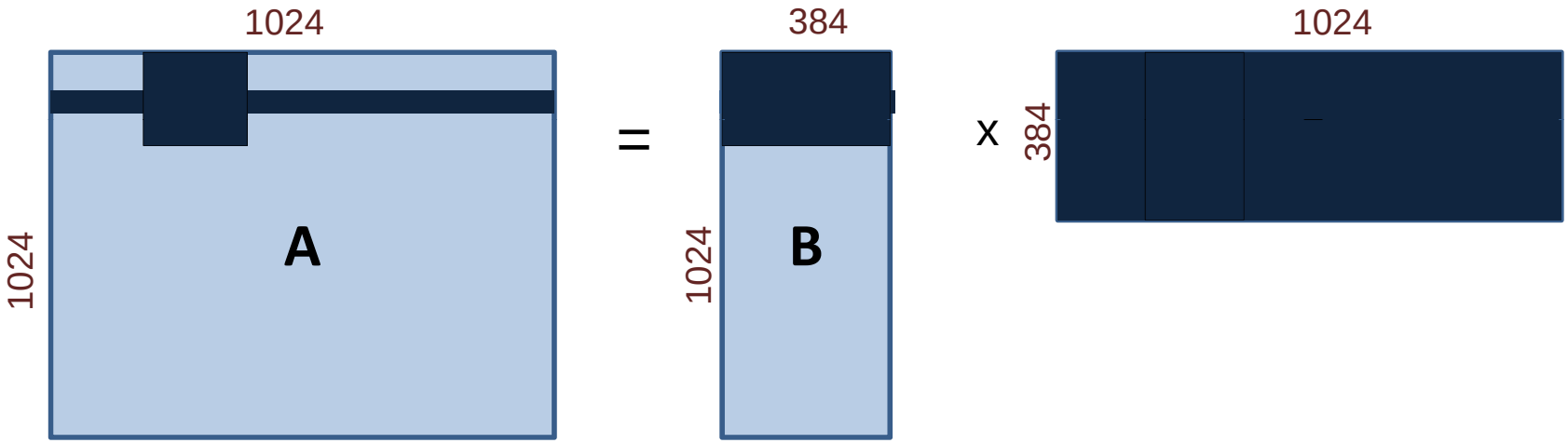
- Remember the formats of `A`, `B` and `C`.
- We compare two strategies for computing 1024 coefficients of `A`.

Issues with data reuse



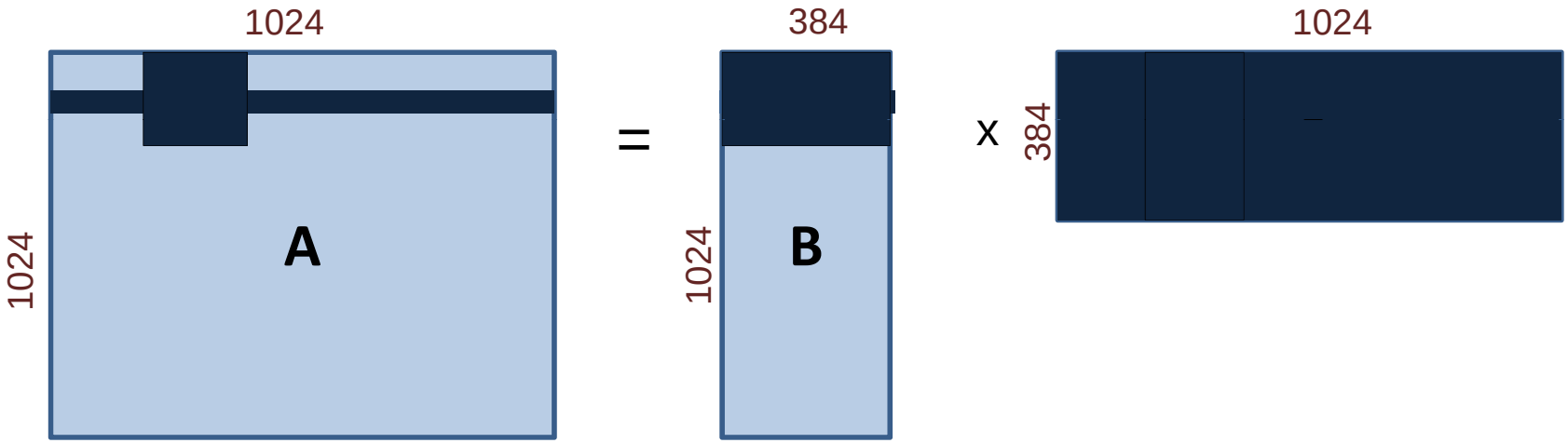
- Remember the formats of `A`, `B` and `C`.
- We compare two strategies for computing 1024 coefficients of `A`.
- Naive calculation of a row of `A`, so computing 1024 coefficients: 1024 accesses in `A`, 384 in `B` and $1024 \times 384 = 393,216$ in `C`. Total $= 394,524$.

Issues with data reuse



- Remember the formats of `A`, `B` and `C`.
- We compare two strategies for computing 1024 coefficients of `A`.
- Naive calculation of a row of `A`, so computing 1024 coefficients: 1024 accesses in `A`, 384 in `B` and $1024 \times 384 = 393,216$ in `C`. Total $= 394,524$.
- Computing a 32×32-block of `A`, so computing again 1024 coefficients: 1024 accesses in `A`, 384×32 in `B` and 32×384 in `C`. Total $= 25,600$.

Issues with data reuse



- Remember the formats of `A`, `B` and `C`.
- We compare two strategies for computing 1024 coefficients of `A`.
- Naive calculation of a row of `A`, so computing 1024 coefficients: 1024 accesses in `A`, 384 in `B` and $1024 \times 384 = 393,216$ in `C`. Total $= 394,524$.
- Computing a 32×32-block of `A`, so computing again 1024 coefficients: 1024 accesses in `A`, 384×32 in `B` and 32×384 in `C`. Total $= 25,600$.
- With the second strategy, the iteration space is traversed so as to reduce memory accesses.

Blocking for optimizing temporal locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C;
        long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i += BLOCK_X)
          for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
              for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                   for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
                       IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,k0,j0,y);
         ended = example_get_time();
         timeTaken = (ended - started)/1.f;
   return timeTaken;
}
```

Transposing and blocking for optimizing data locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C;
        long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i += BLOCK_X)
          for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
              for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                   for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
                       IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,j0,k0,z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
        return timeTaken;
}
```

Experimental results

Computing the product of two $n \times n$ matrices on my 12-year laptop (Core2 Duo CPU P8600 @ 2.40GHz, L1 cache of 3072 KB, 4 GBytes of RAM).

n	naive	transposed	speedup	64×64-tiled	speedup	t. & t.	speedup
128	7	3		7		2	
256	26	43		155		23	
512	1805	265	6.81	1928	0.936	187	9.65
1024	24723	3730	6.62	14020	1.76	1490	16.59
2048	271446	29767	9.11	112298	2.41	11960	22.69
4096	2344594	238453	**9.83**	1009445	**2.32**	101264	**23.15**

Timings are in milliseconds.

Experimental results

Computing the product of two $n \times n$ matrices on my 12-year laptop (Core2 Duo CPU P8600 @ 2.40GHz, L1 cache of 3072 KB, 4 GBytes of RAM).

n	naive	transposed	speedup	64×64-tiled	speedup	t. & t.	speedup
128	7	3		7		2	
256	26	43		155		23	
512	1805	265	6.81	1928	0.936	187	9.65
1024	24723	3730	6.62	14020	1.76	1490	16.59
2048	271446	29767	9.11	112298	2.41	11960	22.69
4096	2344594	238453	**9.83**	1009445	**2.32**	101264	**23.15**

Timings are in milliseconds.

The cache-oblivious multiplication (more on this later) runs within 12978 and 106758 for $n = 2048$ and $n = 4096$ respectively.

Experimental results

Computing the product of two $n \times n$ matrices on my 12-year laptop (Core2 Duo CPU P8600 @ 2.40GHz, L1 cache of 3072 KB, 4 GBytes of RAM).

n	naive	transposed	speedup	64×64-tiled	speedup	t. & t.	speedup
128	7	3		7		2	
256	26	43		155		23	
512	1805	265	6.81	1928	0.936	187	9.65
1024	24723	3730	6.62	14020	1.76	1490	16.59
2048	271446	29767	9.11	112298	2.41	11960	22.69
4096	2344594	238453	**9.83**	1009445	**2.32**	101264	**23.15**

Timings are in milliseconds.

The cache-oblivious multiplication (more on this later) runs within 12978 and 106758 for $n = 2048$ and $n = 4096$ respectively.

Use my C programs to do those benchmarks on your machine.

Other performance counters

Hardware counter events

- CPI – Clock cycles Per Instruction: the number of clock cycles that happen when an instruction is being executed. With pipelining we can improve the CPI by exploiting instruction level parallelism

	CPI		L1 Miss Rate		L2 Miss Rate	Percent SSE Instructions	Instructions Retired	
In C	4.78		0.24		0.02	43%	13,137,280,000	
		5x		2x				1x
Transposed	1.13		0.15		0.02	50%	13,001,486,336	
		3x		8x				0.8x
Tiled	0.49		0.02		0	39%	18,044,811,264	

Other performance counters

Hardware counter events

- CPI – Clock cycles Per Instruction: the number of clock cycles that happen when an instruction is being executed. With pipelining we can improve the CPI by exploiting instruction level parallelism
- L1 and L2 Cache Miss Rate.

	CPI		L1 Miss Rate		L2 Miss Rate	Percent SSE Instructions	Instructions Retired	
In C	4.78		0.24		0.02	43%	13,137,280,000	
		5x		2x				1x
Transposed	1.13		0.15		0.02	50%	13,001,486,336	
		3x		8x				0.8x
Tiled	0.49		0.02		0	39%	18,044,811,264	

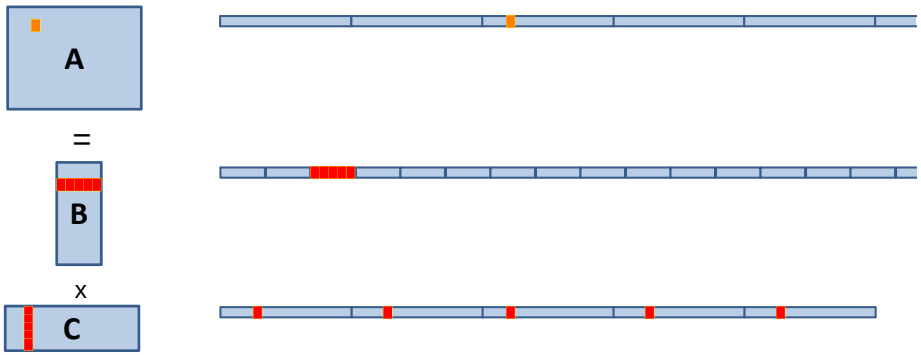
Other performance counters

Hardware counter events

- CPI – Clock cycles Per Instruction: the number of clock cycles that happen when an instruction is being executed. With pipelining we can improve the CPI by exploiting instruction level parallelism
- L1 and L2 Cache Miss Rate.
- Instructions Retired: In the event of a misprediction, instructions that were scheduled to execute along the mispredicted path must be canceled; the other ones (those needed by the program flow) are called retired.

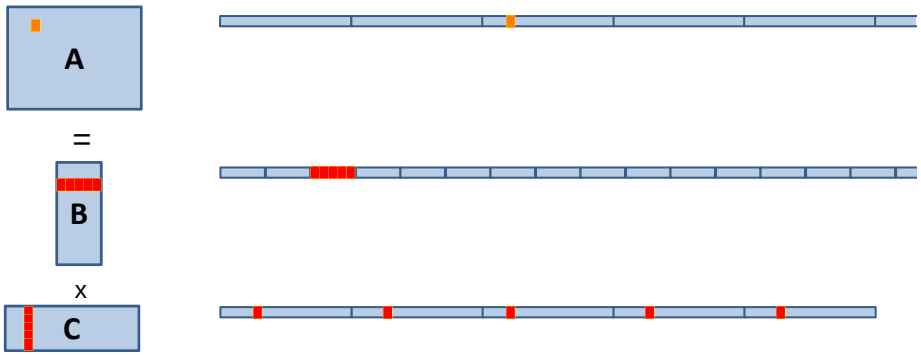
	CPI		L1 Miss Rate		L2 Miss Rate	Percent SSE Instructions	Instructions Retired	
In C	4.78	5x	0.24	2x	0.02	43%	13,137,280,000	1x
Transposed	1.13	3x	0.15	8x	0.02	50%	13,001,486,336	0.8x
Tiled	0.49		0.02		0	39%	18,044,811,264	

Analyzing cache misses in the naive and transposed multiplication



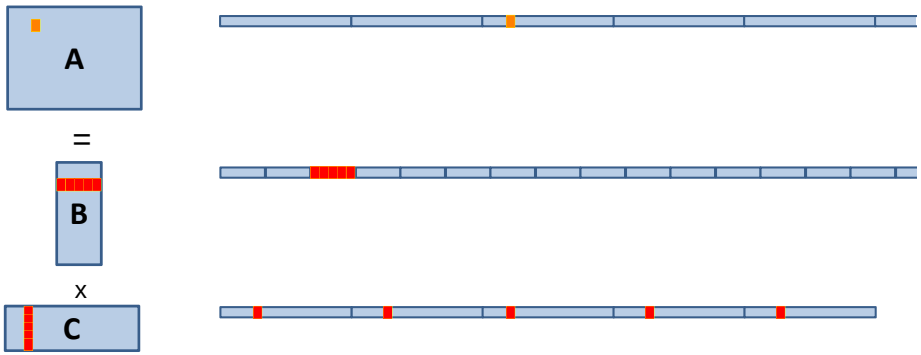
- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.

Analyzing cache misses in the naive and transposed multiplication



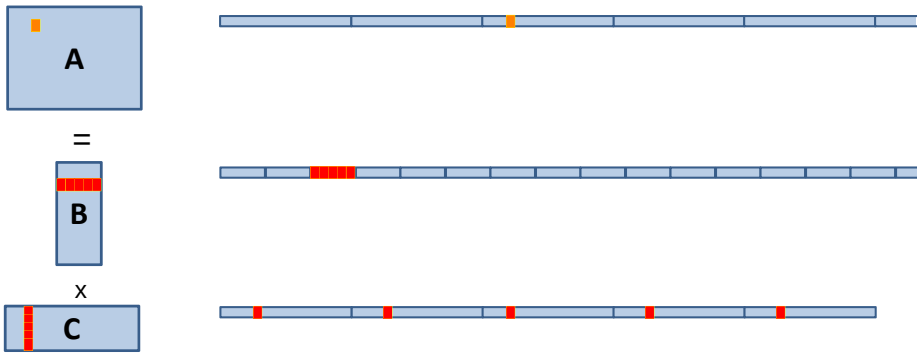
- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- A is scanned once, so mn/L cache misses if L is the number of coefficients per cache line.

Analyzing cache misses in the naive and transposed multiplication



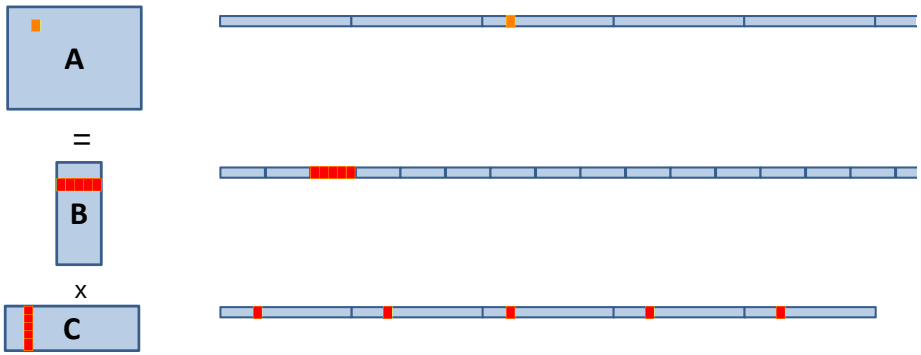
- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- A is scanned once, so mn/L cache misses if L is the number of coefficients per cache line.
- B is scanned n times, so mnp/L cache misses if the cache cannot hold a row.

Analyzing cache misses in the naive and transposed multiplication



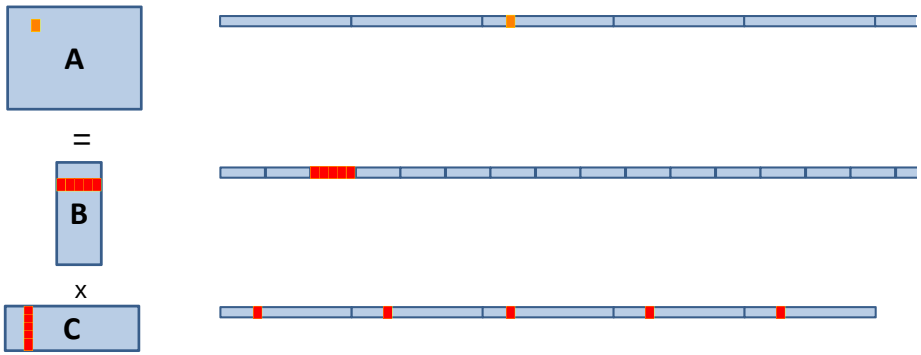
- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- A is scanned once, so mn/L cache misses if L is the number of coefficients per cache line.
- B is scanned n times, so mnp/L cache misses if the cache cannot hold a row.
- C is accessed "nearly randomly" (for m large enough) leading to mnp cache misses.

Analyzing cache misses in the naive and transposed multiplication



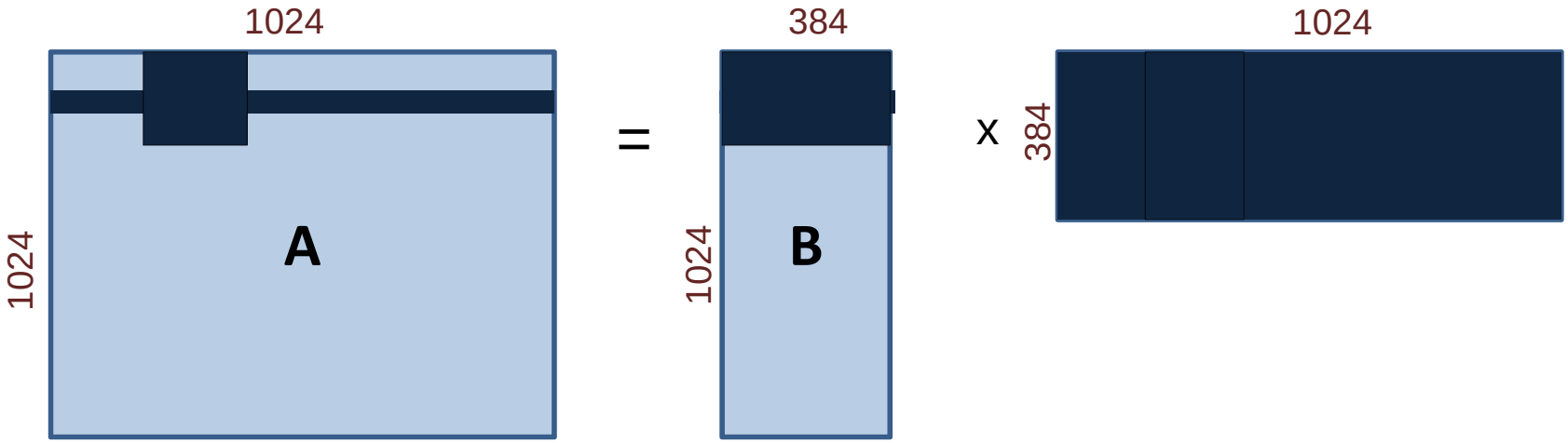
- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- A is scanned once, so mn/L cache misses if L is the number of coefficients per cache line.
- B is scanned n times, so mnp/L cache misses if the cache cannot hold a row.
- C is accessed "nearly randomly" (for m large enough) leading to mnp cache misses.
- Since $2m\,n\,p$ arithmetic operations are performed, this means roughly **one cache miss for two flops!**

Analyzing cache misses in the naive and transposed multiplication



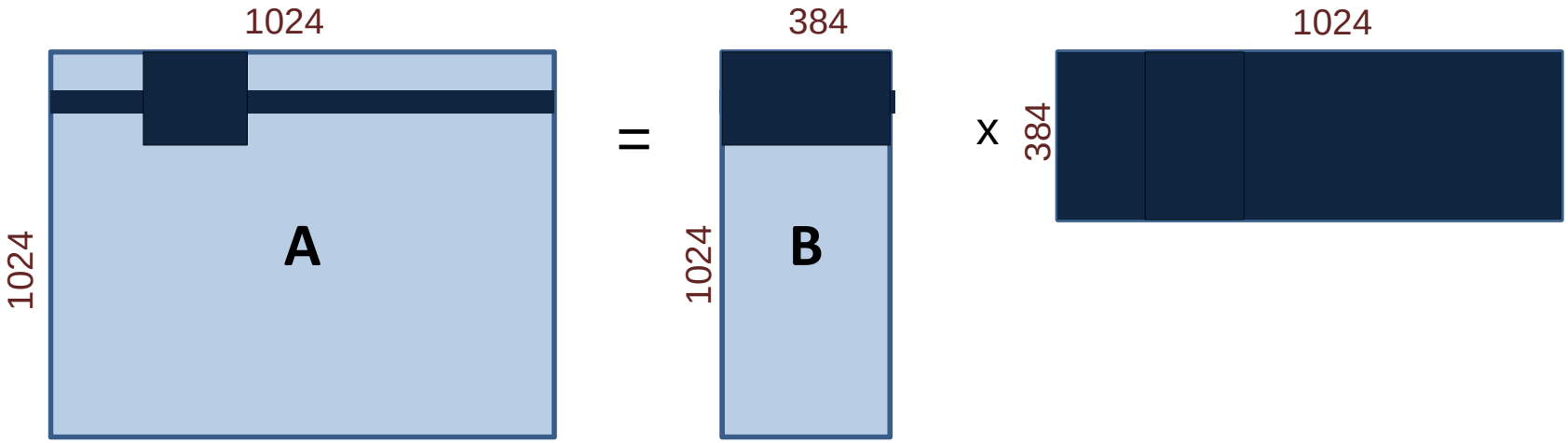
- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- A is scanned once, so mn/L cache misses if L is the number of coefficients per cache line.
- B is scanned n times, so mnp/L cache misses if the cache cannot hold a row.
- C is accessed "nearly randomly" (for m large enough) leading to mnp cache misses.
- Since $2m\,n\,p$ arithmetic operations are performed, this means roughly **one cache miss for two flops!**
- If C is transposed, then the ratio improves to 1 for L.

Analyzing cache misses in the tiled multiplication



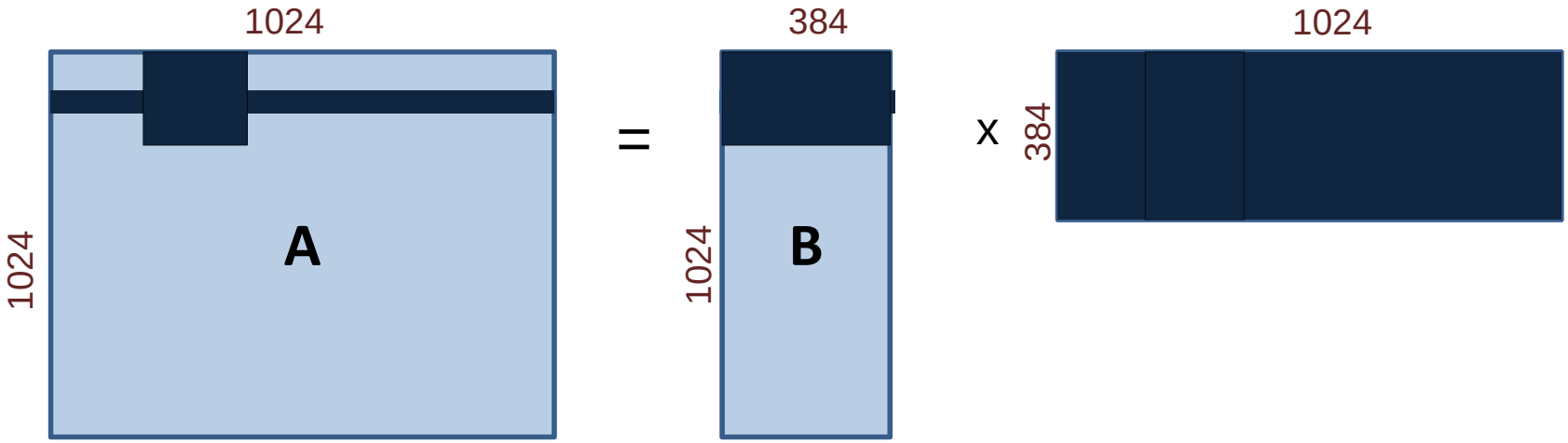
- Let A, B and C are all square of order of n.

Analyzing cache misses in the tiled multiplication



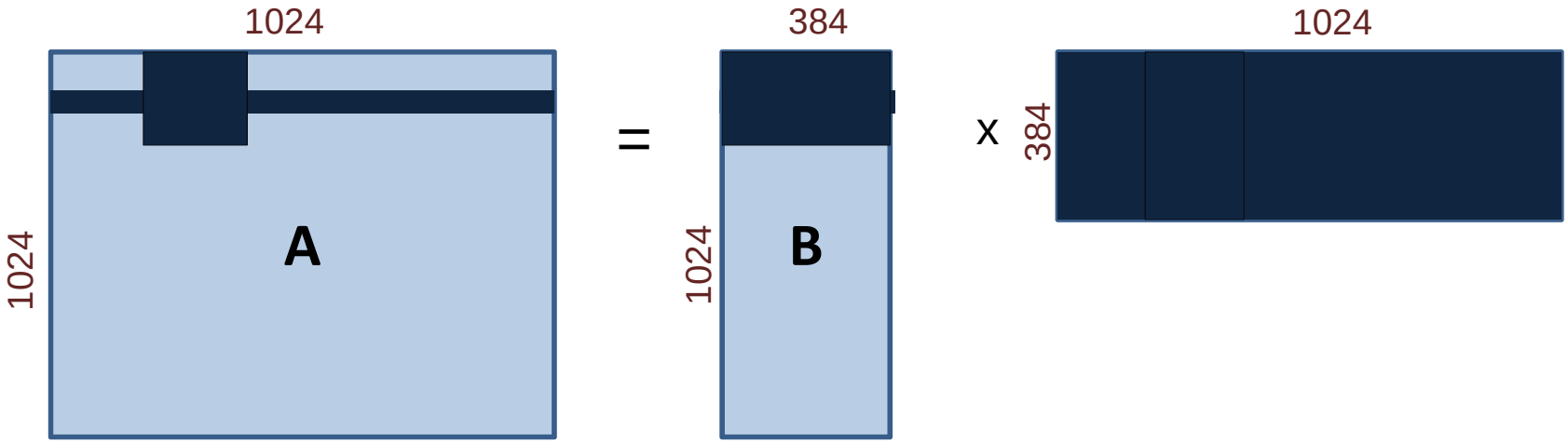
- Let A, B and C are all square of order of n.
- Assume all tiles are square of order b and three fit in cache.

Analyzing cache misses in the tiled multiplication



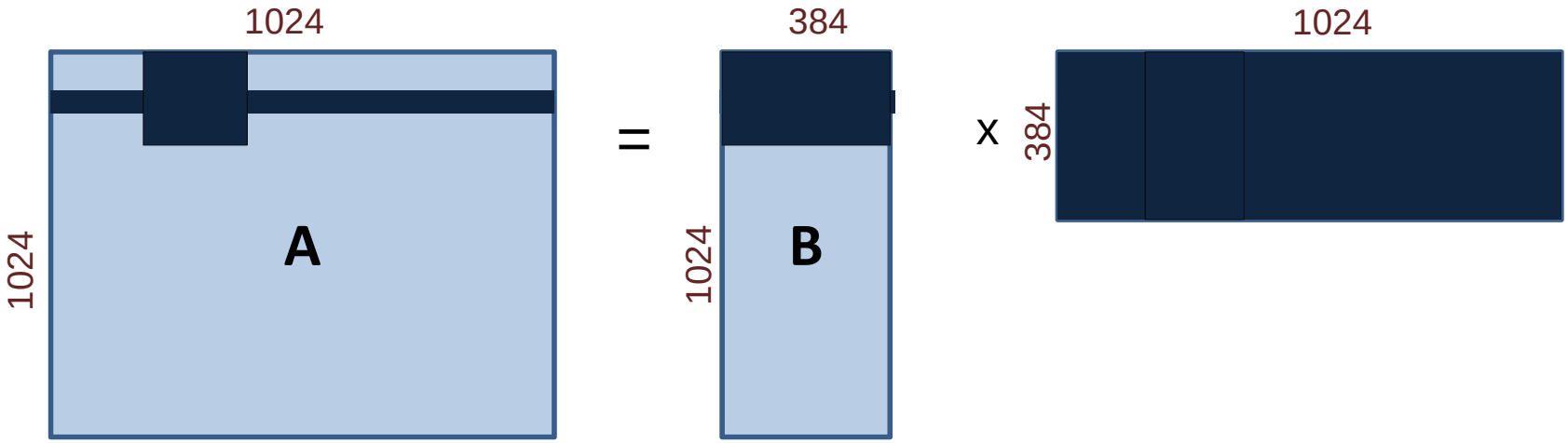
- Let A, B and C are all square of order of n.
- Assume all tiles are square of order b and three fit in cache.
- If C is transposed, then loading three blocks in cache cost $3b^2/L$.

Analyzing cache misses in the tiled multiplication



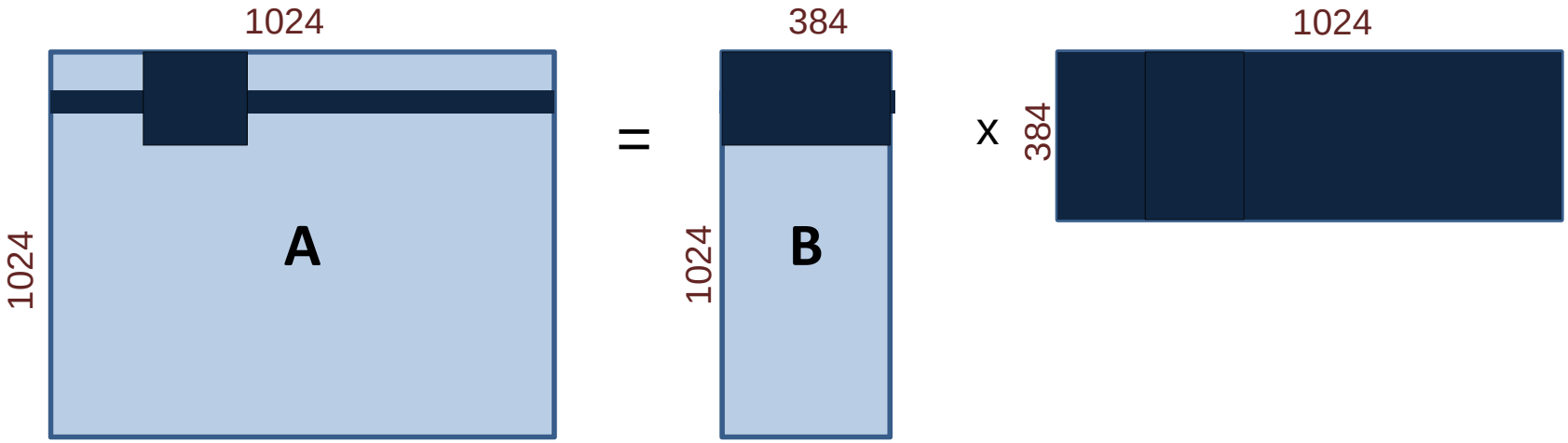
- Let A, B and C are all square of order of n.
- Assume all tiles are square of order b and three fit in cache.
- If C is transposed, then loading three blocks in cache cost $3b^2/L$.
- This process happens n^3/b^3 times, leading to $3n^3/(bL)$ cache misses.

Analyzing cache misses in the tiled multiplication



- Let A, B and C are all square of order of n.
- Assume all tiles are square of order b and three fit in cache.
- If C is transposed, then loading three blocks in cache cost $3b^2/L$.
- This process happens n^3/b^3 times, leading to $3n^3/(bL)$ cache misses.
- Three blocks fit in cache for $3b^2 < Z$, if Z is the cache size.

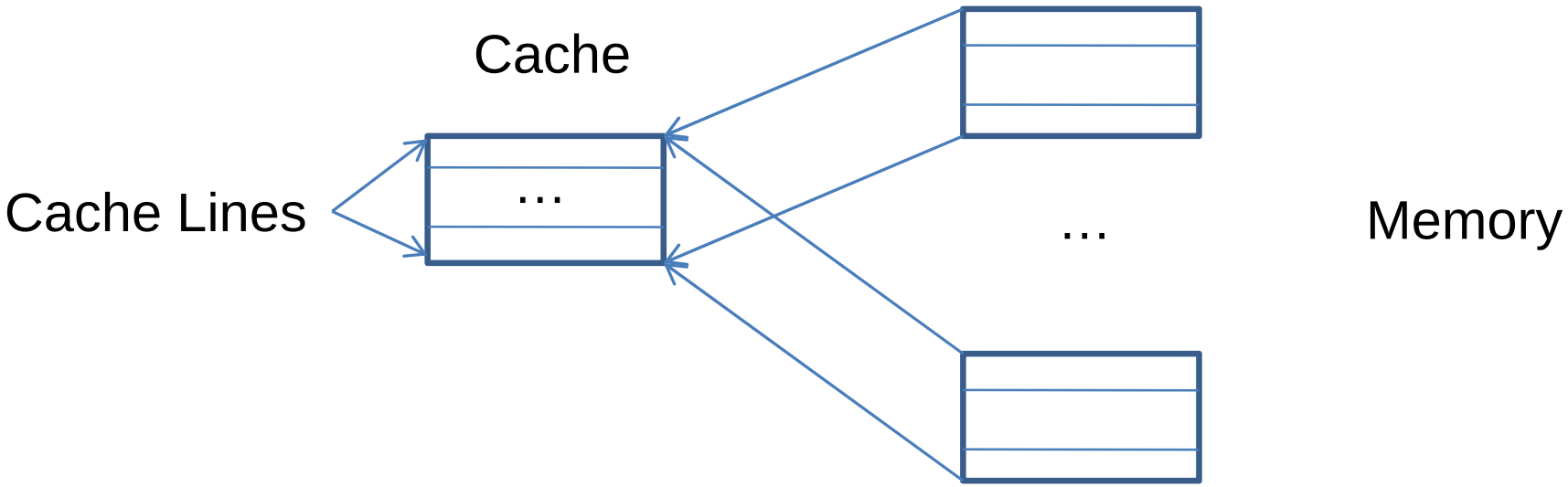
Analyzing cache misses in the tiled multiplication



- Let A, B and C are all square of order of n.
- Assume all tiles are square of order b and three fit in cache.
- If C is transposed, then loading three blocks in cache cost $3b^2/L$.
- This process happens n^3/b^3 times, leading to $3n^3/(bL)$ cache misses.
- Three blocks fit in cache for $3b^2 < Z$, if Z is the cache size.
- So $O(n^3/(\sqrt{Z}L))$ cache misses, if b is well chosen, which is optimal.

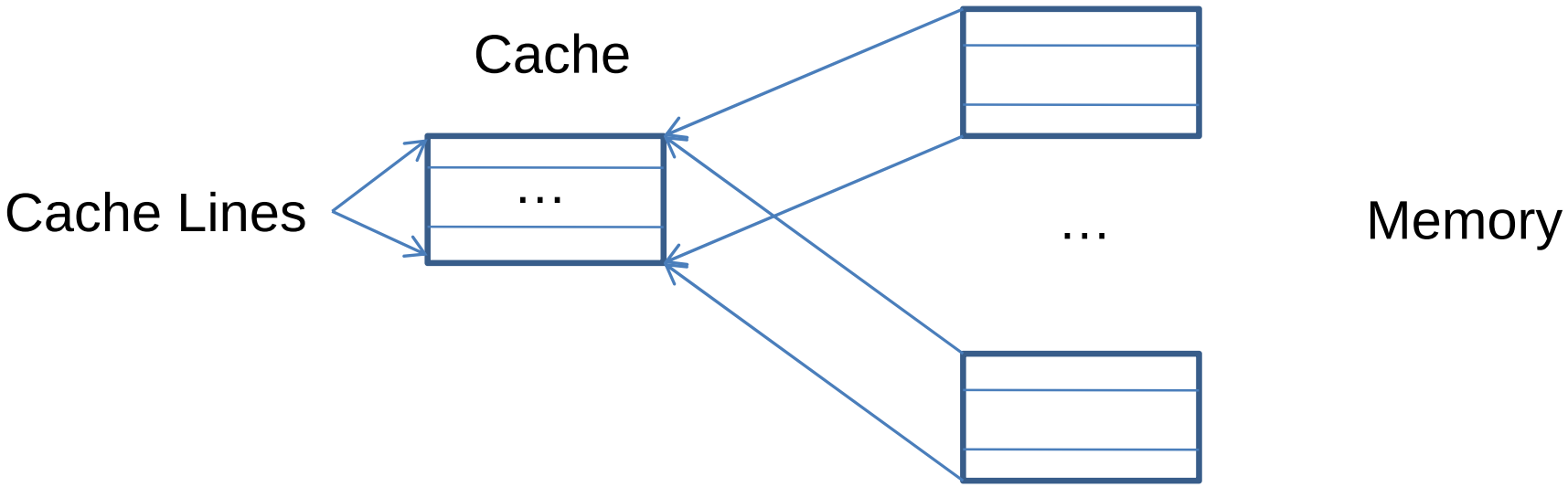
Outline

Basic idea of a cache memory (review)



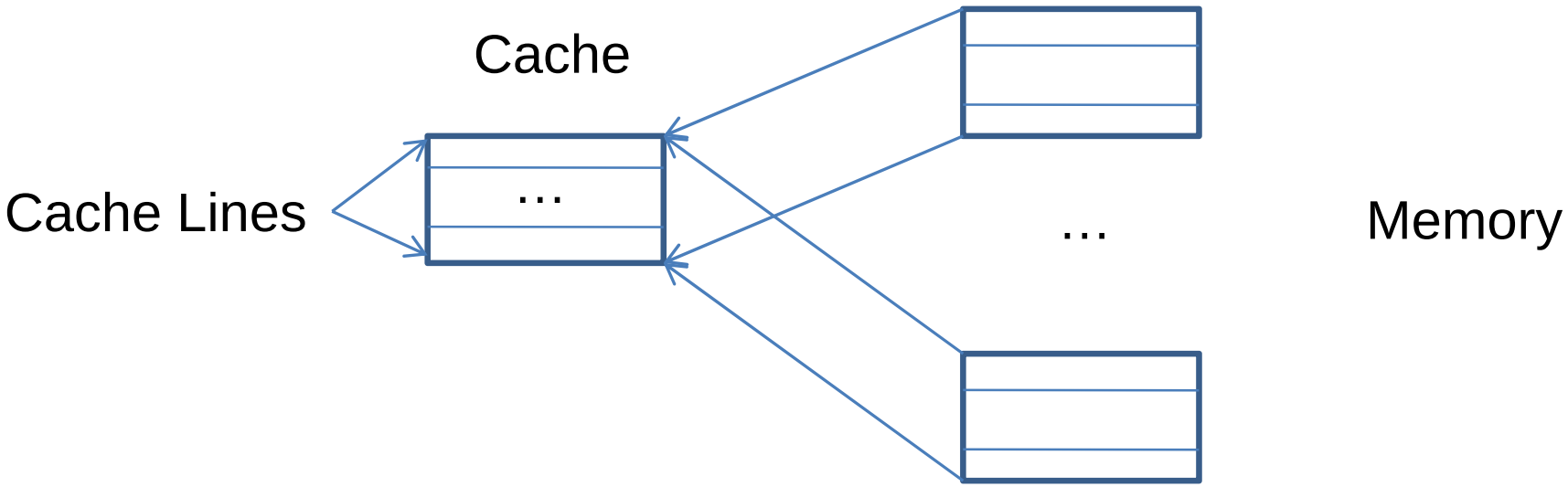
- Recall that a cache is a smaller memory, faster to access.

Basic idea of a cache memory (review)



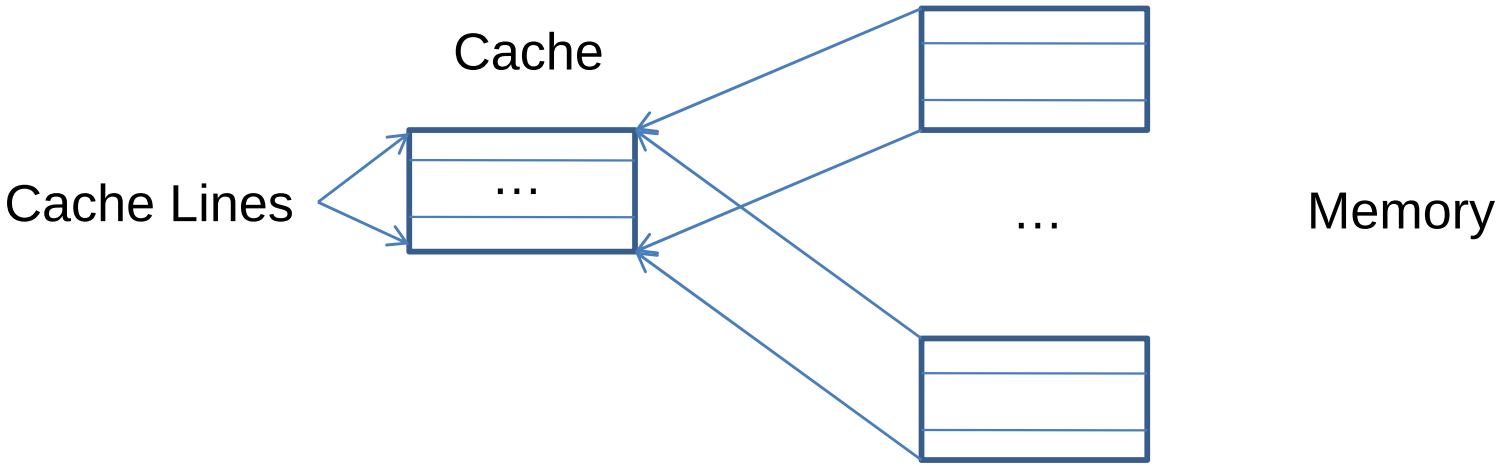
- Recall that a cache is a smaller memory, faster to access.
- Using smaller memory to cache contents of larger memory provides the illusion of fast larger memory.

Basic idea of a cache memory (review)



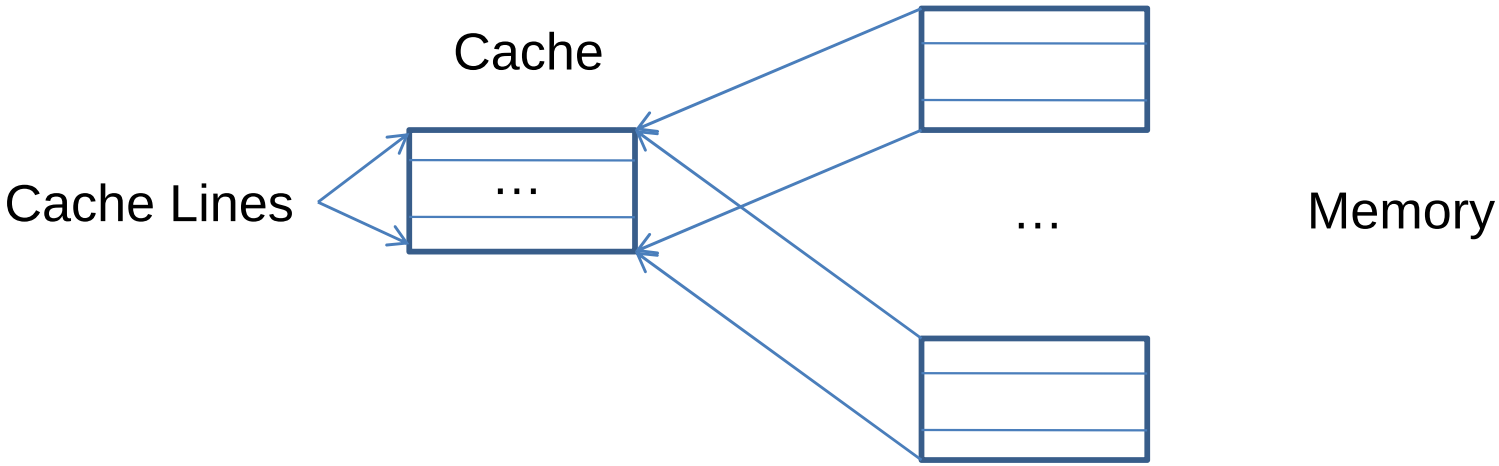
- Recall that a cache is a smaller memory, faster to access.
- Using smaller memory to cache contents of larger memory provides the illusion of fast larger memory.
- Key reasons why this works: **temporal locality** and **spatial locality**.

A simple cache example



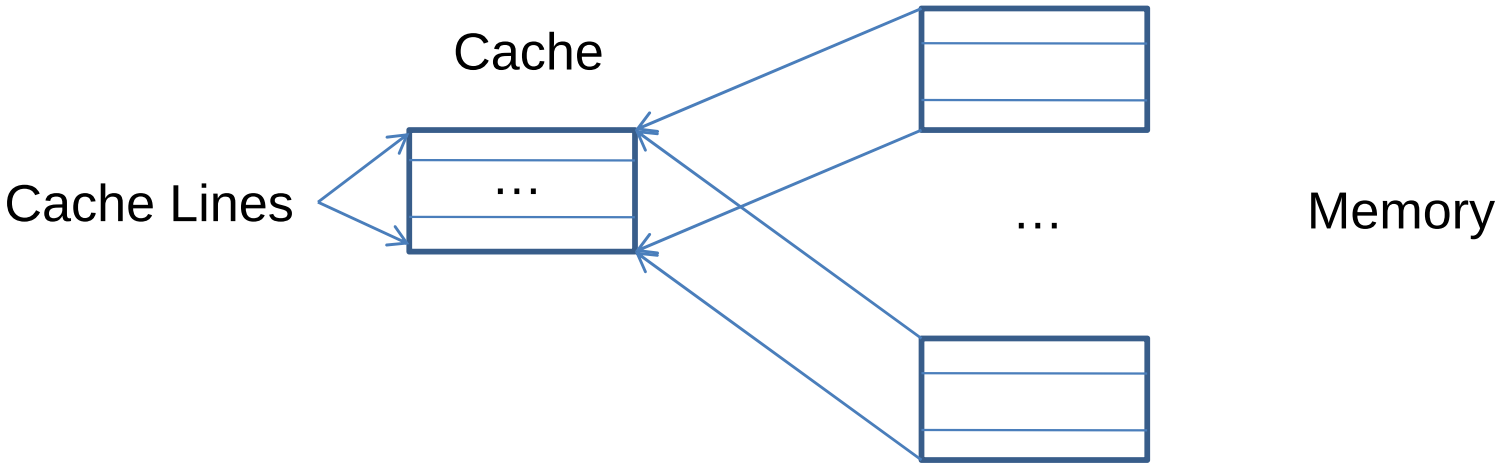
- Byte addressable memory

A simple cache example



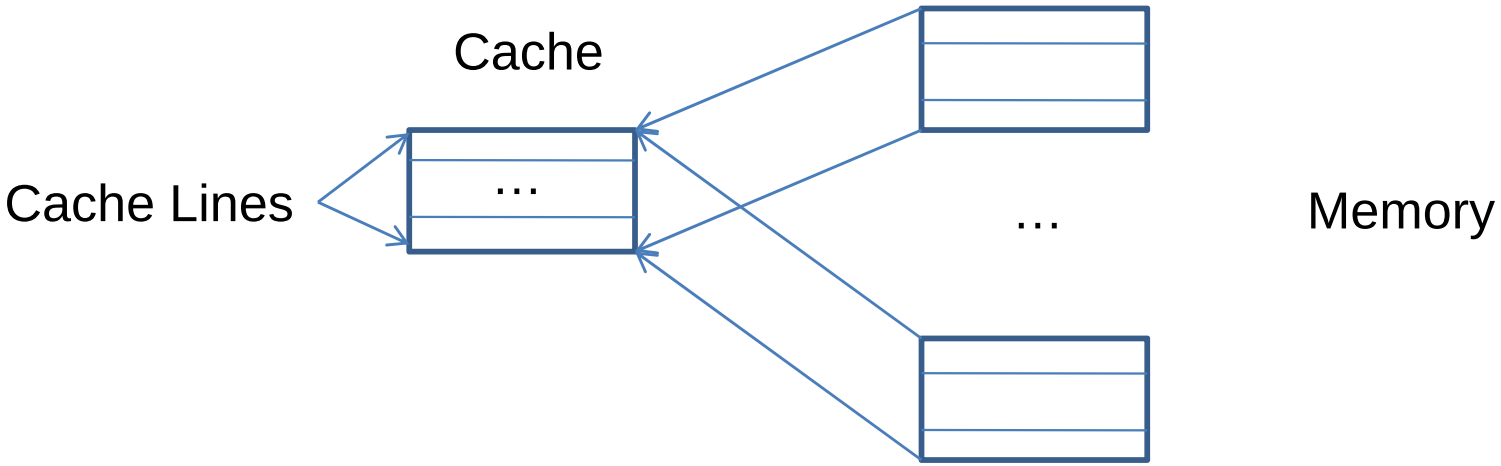
- Byte addressable memory
- Cache of 32 Kbyte with direct mapping and 64 byte lines (thus 512 lines) so the cache can fit $2^9 \times 2^4 = 2^{13}$ `int`.

A simple cache example



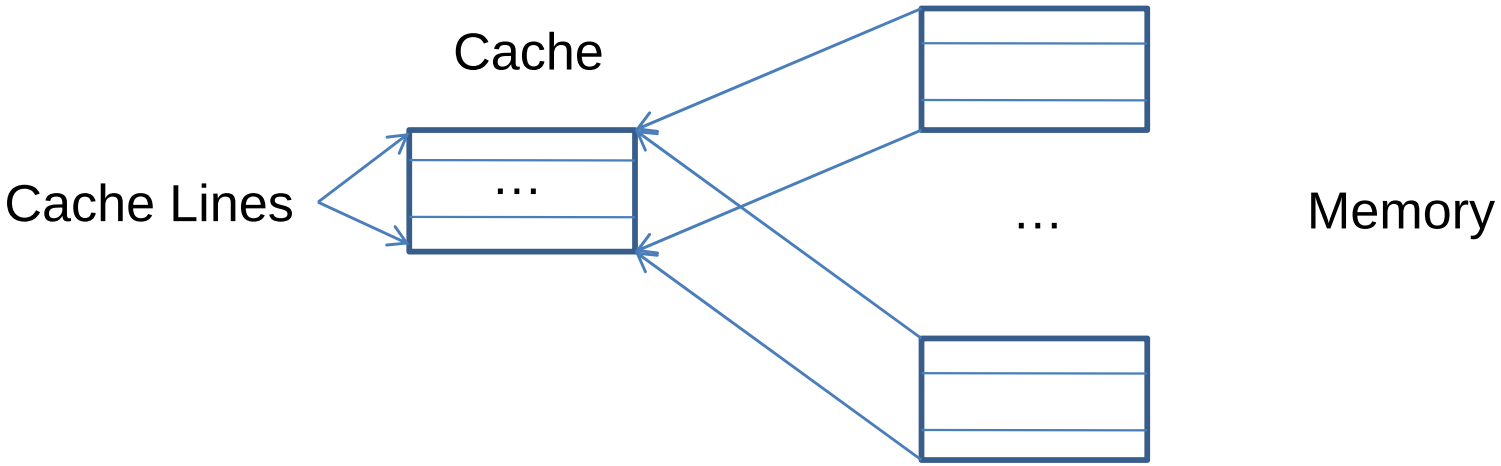
- Byte addressable memory
- Cache of 32 Kbyte with direct mapping and 64 byte lines (thus 512 lines) so the cache can fit $2^9 \times 2^4 = 2^{13}$ `int`.
- "Therefore" successive 32 Kbyte memory blocks line up in cache

A simple cache example



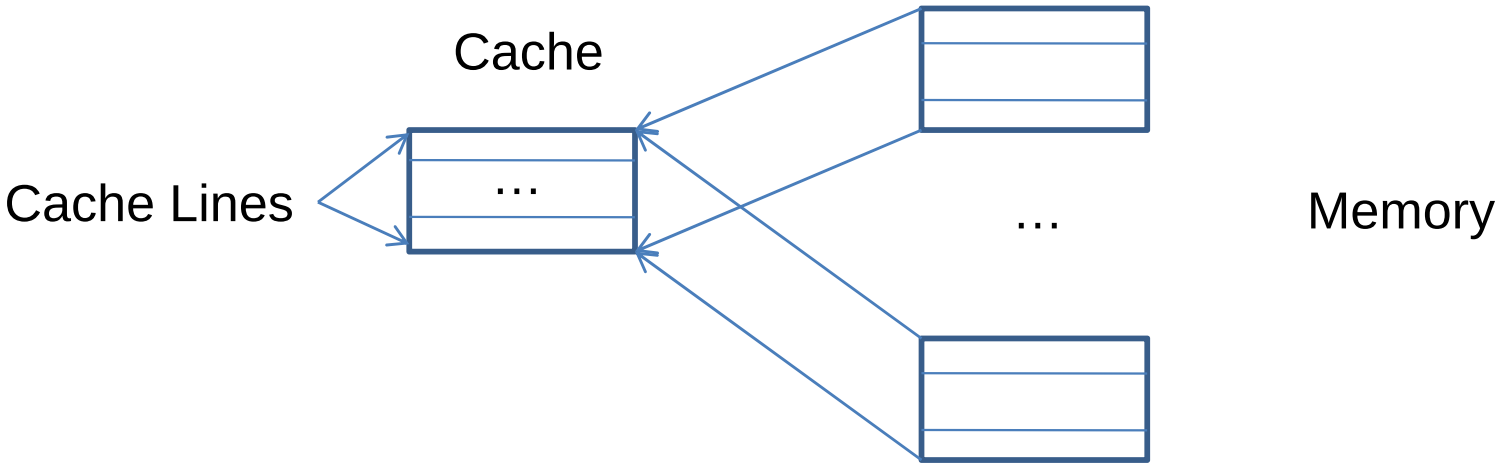
- Byte addressable memory
- Cache of 32 Kbyte with direct mapping and 64 byte lines (thus 512 lines) so the cache can fit $2^9 \times 2^4 = 2^{13}$ `int`.
- "Therefore" successive 32 Kbyte memory blocks line up in cache
- A cache access costs 1 cycle while a memory access costs $100 = 99 + 1$ cycles.

A simple cache example



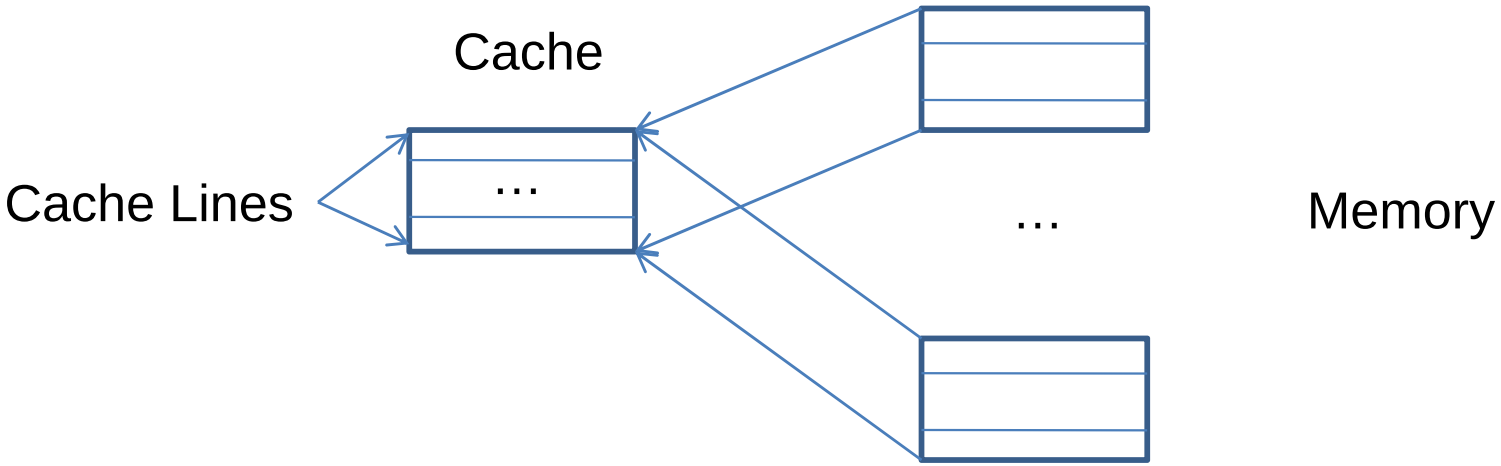
- Byte addressable memory
- Cache of 32 Kbyte with direct mapping and 64 byte lines (thus 512 lines) so the cache can fit $2^9 \times 2^4 = 2^{13}$ `int`.
- "Therefore" successive 32 Kbyte memory blocks line up in cache
- A cache access costs 1 cycle while a memory access costs $100 = 99 + 1$ cycles.
- How addresses map into cache?

A simple cache example



- Byte addressable memory
- Cache of 32 Kbyte with direct mapping and 64 byte lines (thus 512 lines) so the cache can fit $2^9 \times 2^4 = 2^{13}$ `int`.
- "Therefore" successive 32 Kbyte memory blocks line up in cache
- A cache access costs 1 cycle while a memory access costs $100 = 99 + 1$ cycles.
- How addresses map into cache?
  - ↳ The bottom 6 bits are used as offset in a cache line,

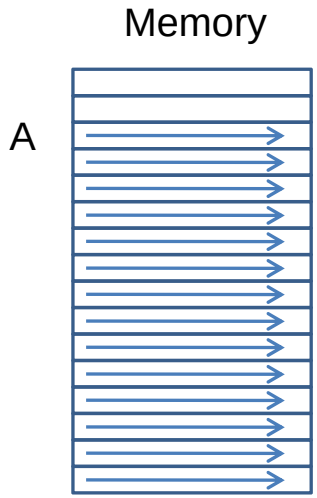
A simple cache example



- Byte addressable memory
- Cache of 32 Kbyte with direct mapping and 64 byte lines (thus 512 lines) so the cache can fit $2^9 \times 2^4 = 2^{13}$ `int`.
- "Therefore" successive 32 Kbyte memory blocks line up in cache
- A cache access costs 1 cycle while a memory access costs $100 = 99 + 1$ cycles.
- How addresses map into cache?
  - ↳ The bottom 6 bits are used as offset in a cache line,
  - ↳ The next 9 bits determine the cache line

Exercise 1 (1/2)

```
// sizeof(int) = 4 and Array laid out sequentially in memory
#define S ((1<<20)*sizeof(int))
int A[S];
// Thus size of A is 2^(20) x 4
for (i = 0; i < S; i++) {
        read A[i];
}
```



Total access time? What kind of locality? What kind of misses?

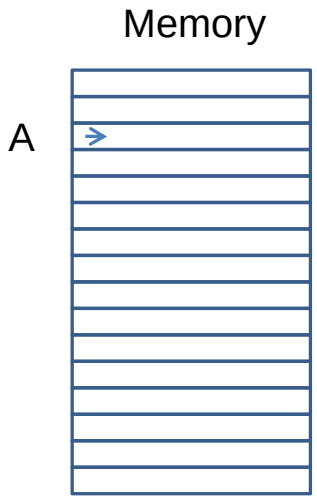
Exercise 1 (2/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[i];
}
```

- S reads to A.
- 16 elements of A per cache line
- 15 of every 16 hit in cache.
- Total access time: $15(S/16) + 100(S/16)$.
- spatial locality, cold misses.

Exercise 2 (1/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[0];
}
```



Total access time? What kind of locality? What kind of misses?

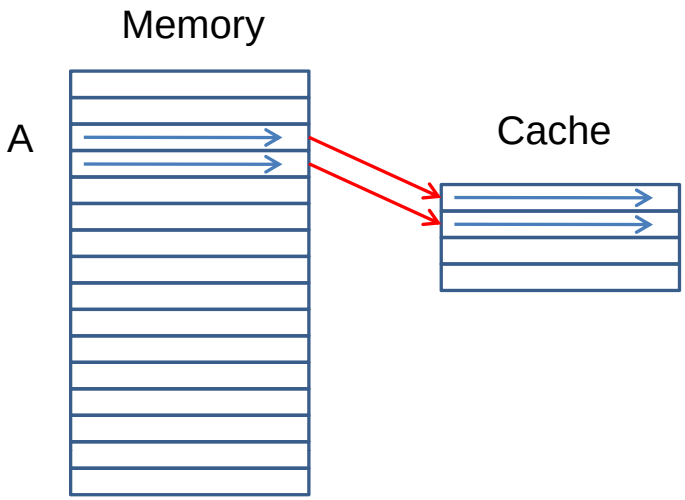
Exercise 2 (2/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[0];
}
```

- S reads to A
- All except the first one hit in cache.
- Total access time: $100 + (S - 1)$.
- Temporal locality
- Cold misses.

Exercise 3 (1/2)

```
// Assume 4 <= N <= 13
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[i % (1<<N)];
}
```



Total access time? What kind of locality? What kind of misses?

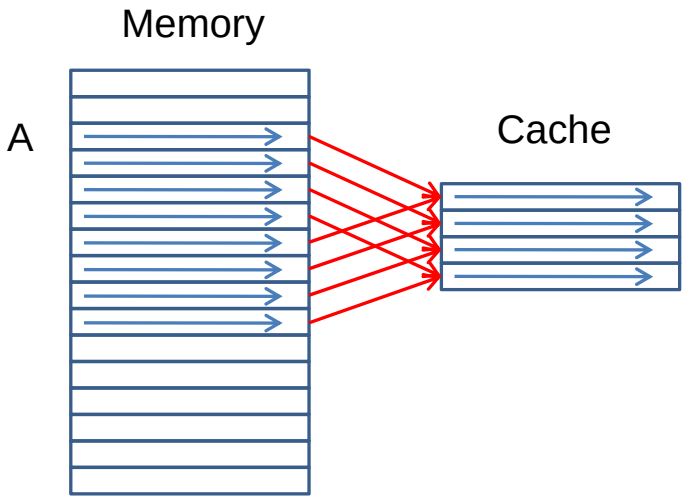
Exercise 3 (2/2)

```
// Assume 4 <= N <= 13
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[i % (1<<N)];
}
```

- S reads to A
- One miss for each accessed line, rest hit in cache.
- Number of accessed lines: 2^{N-4}.
- Total access time: $2^{N-4}100 + (S - 2^{N-4})$.
- Temporal and spatial locality
- Cold misses.

Exercise 4 (1/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
 read A[i % (1<<N)];
}
```



Total access time? What kind of locality? What kind of misses?

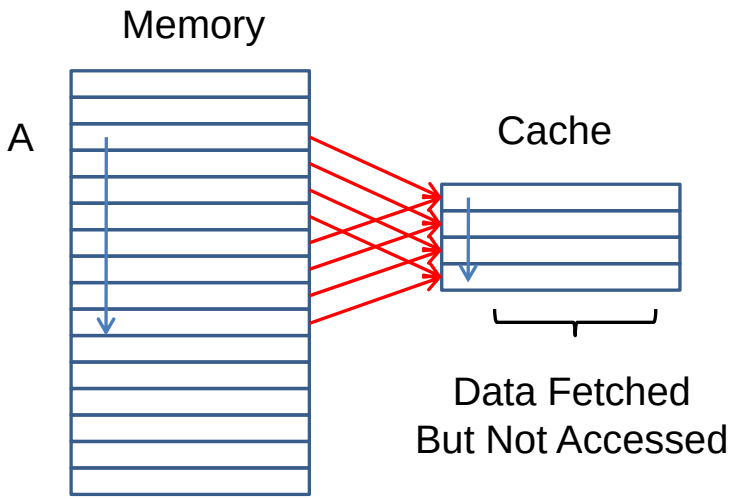
Exercise 4 (2/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
 read A[i % (1<<N)];
}
```

- S reads to A.
- First access to each line misses
- Rest accesses to that line hit.
- Total access time: $15(S/16) + 100(S/16)$.
- Spatial locality
- Cold and capacity misses.

Exercise 5 (1/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
read A[(i*16) % (1<<N)];
}
```



Total access time? What kind of locality? What kind of misses?

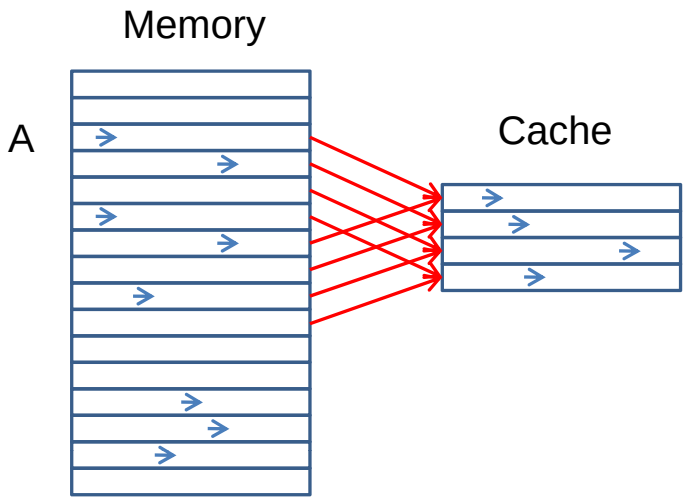
Exercise 5 (2/2)

```
// Assume 14 <= N
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
read A[(i*16) % (1<<N)];
}
```

- S reads to A.
- First access to each line misses
- One access per line.
- Total access time: $100S$.
- No locality!
- Cold and conflict misses.

Exercise 6 (1/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[random()%S];
}
```



Total access time? What kind of locality? What kind of misses?

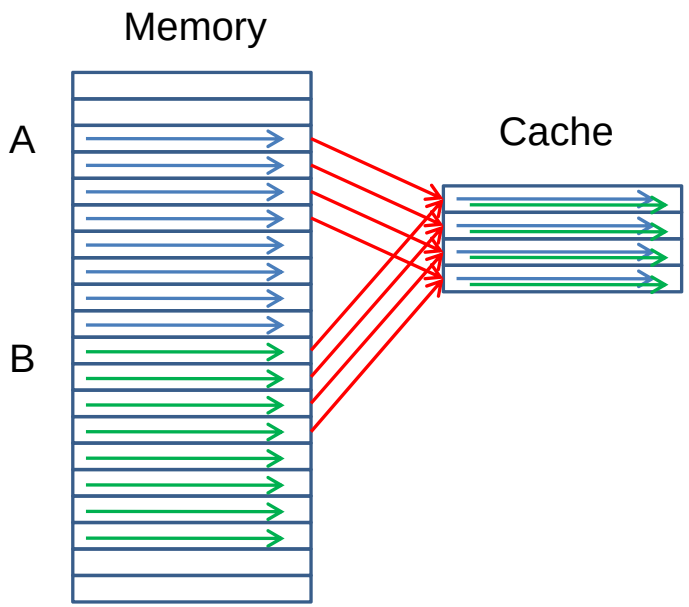
Exercise 6 (2/2)

```
#define S ((1<<20)*sizeof(int))
int A[S];
for (i = 0; i < S; i++) {
        read A[random()%S];
}
```

- S reads to A.
- After N iterations, for some N, the cache is full.
- Them the chance of hitting in cache is $2^9/2^{18} = 1/512$, that is the number of lines in the cache divided by the total number of cache lines used by A.
- Estimated total access time: $S(511/512)100 + S(1/512)$.
- Almost no locality!
- Cold, capacity conflict misses.

Exercise 7 (1/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```



Total access time? What kind of locality? What kind of misses?

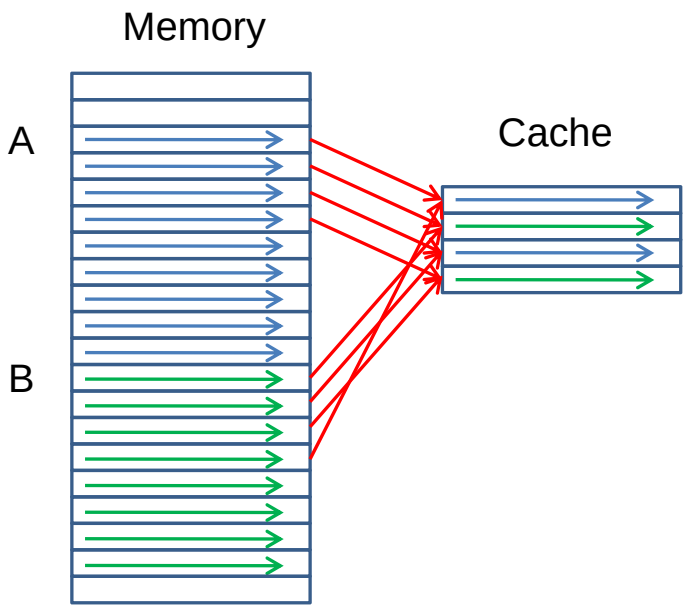
Exercise 7 (2/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```

- S reads to A and B.
- A and B interfere in cache: indeed two cache lines whose addresses differ by a multiple of 2^9 have the *same way to cache*.
- Total access time: $200S$.
- Spatial locality but the cache cannot exploit it.
- Cold and conflict misses.

Exercise 8 (1/2)

```
#define S ((1<<19+4)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```



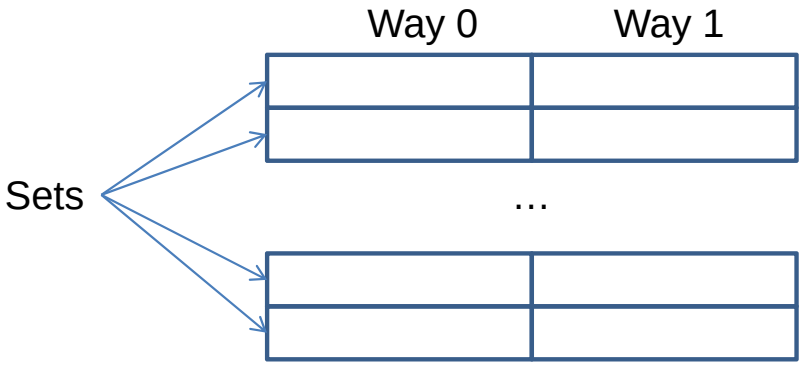
Total access time? What kind of locality? What kind of misses?

Exercise 8 (2/2)

```
#define S ((1<<19+4)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```

- S reads to A and B.
- A and B almost do not interfere in cache.
- Total access time: $2(15S/16 + 100S/16)$.
- Spatial locality.
- Cold misses.

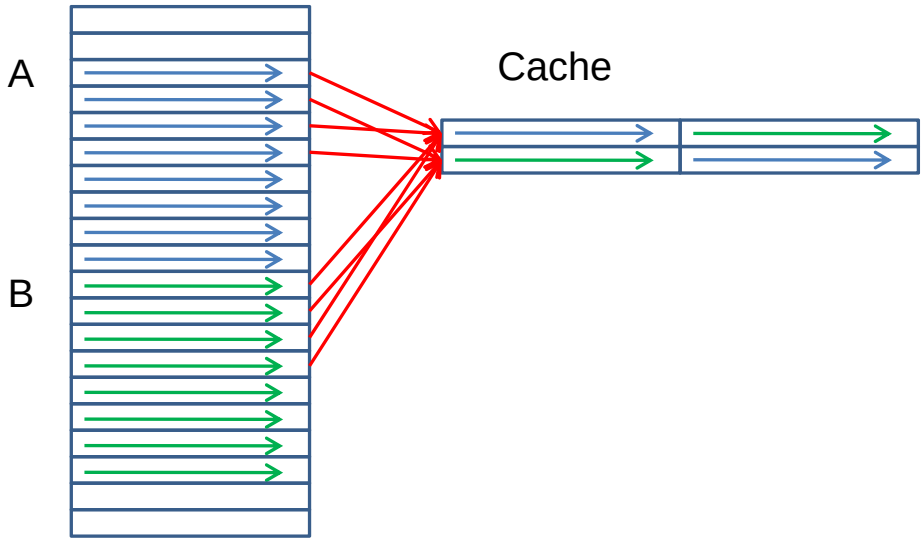
Set Associative Caches



- **Set associative caches** have sets with multiple lines per set.
- Each line in a set is called a way
- Each memory line maps to a specific set and can be put into any cache line in its set
- In our example, we assume a 32 Kbyte cache, with 64 byte lines, 2-way associative. Hence we have:
  - ↳ 256 sets
  - ↳ Bottom six bits determine offset in cache line
  - ↳ Next 8 bits determine the set.

Exercise 9 (1/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```



Total access time? What kind of locality? What kind of misses?

Exercise 9 (2/2)

```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
for (i = 0; i < S; i++) {
read A[i], B[i];
}
```

- S reads to A and B.
- A and B lines hit same set, but enough lines in a set.
- Total access time: $2(15S/16 + 100S/16)$.
- Spatial locality.
- Cold misses.

Extra Exercise A

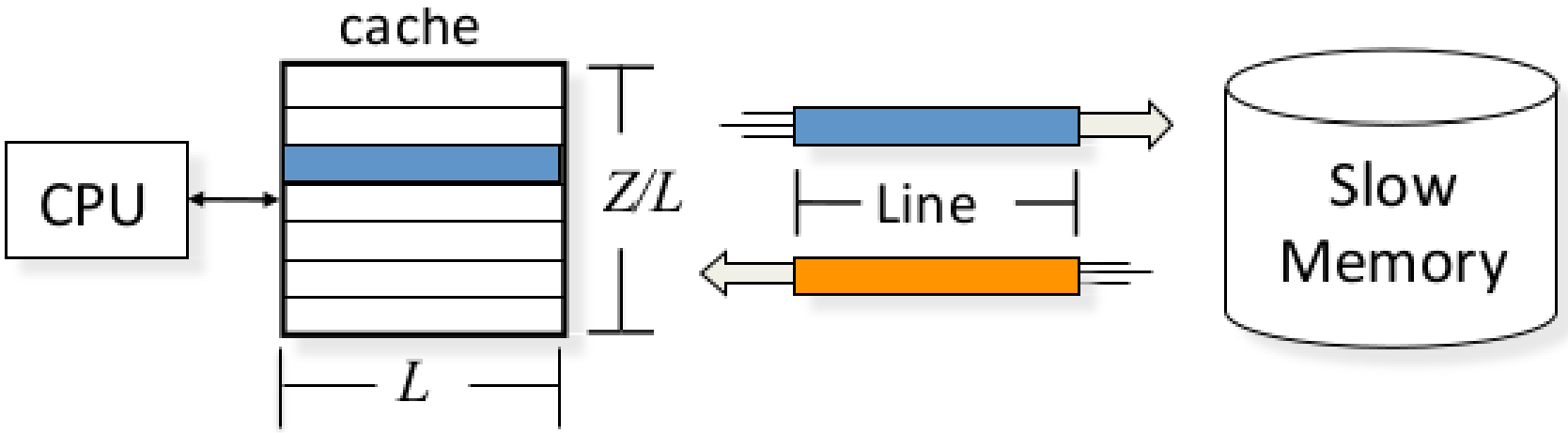
```
#define S ((1<<19)*sizeof(int))
int A[S];
int B[S];
int C[S};
for (i = 0; i < S; i++) {
        C[i] := A[i] + B[i];
}
```

For the above 2-way associative cache (of size 32 Kbyte cache, and with 64 byte lines): Total access time? What kind of locality? What kind of misses?

Outline

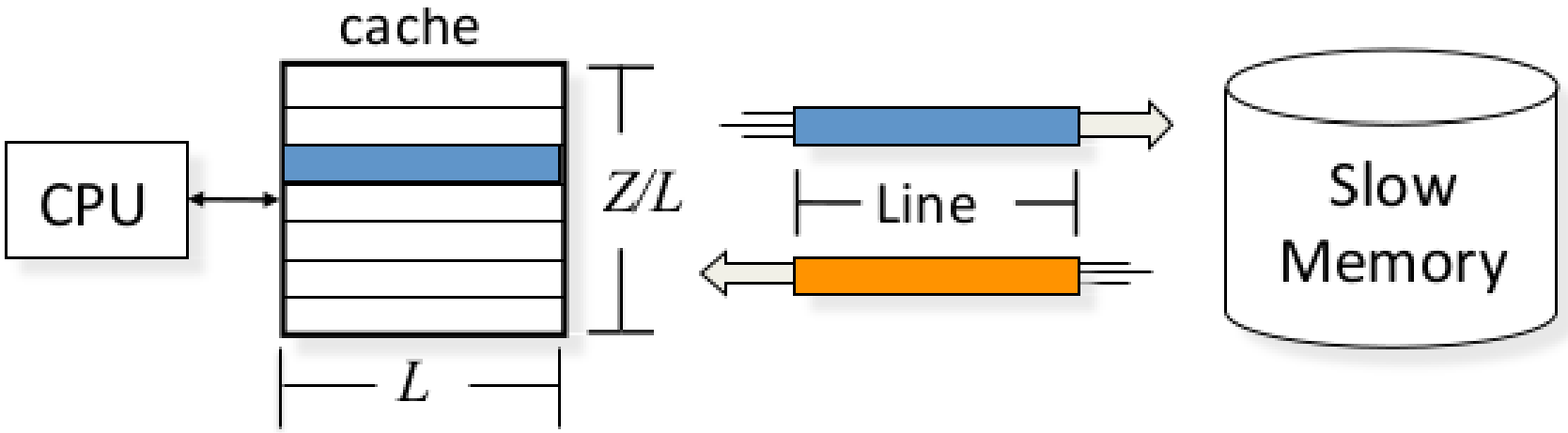
Outline

The ideal cache model (1/5)



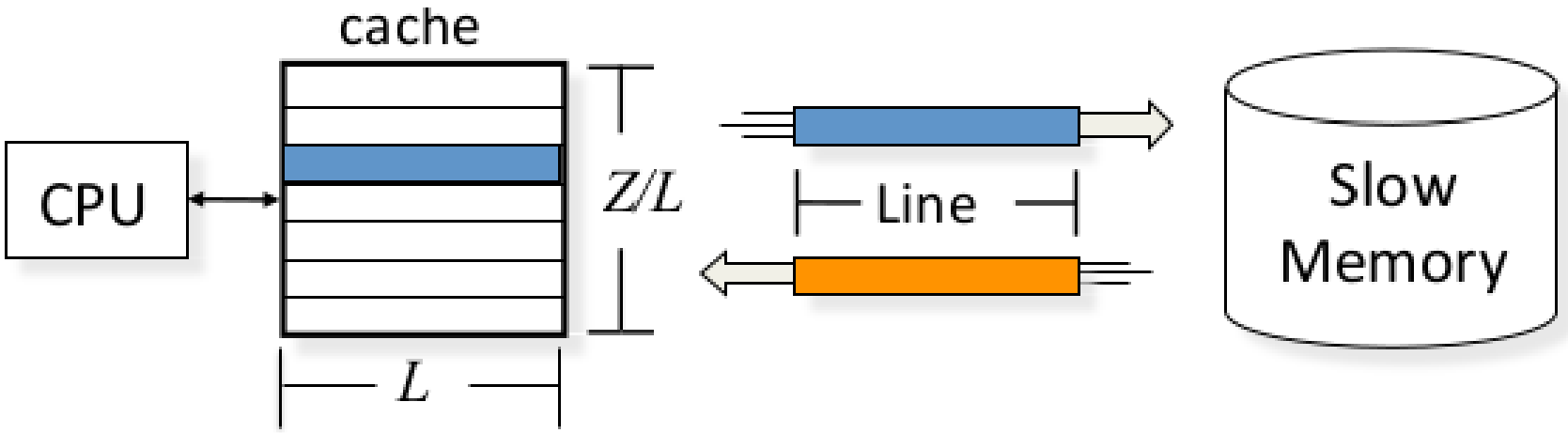
- Computer with a **two-level memory hierarchy**:

The ideal cache model (1/5)



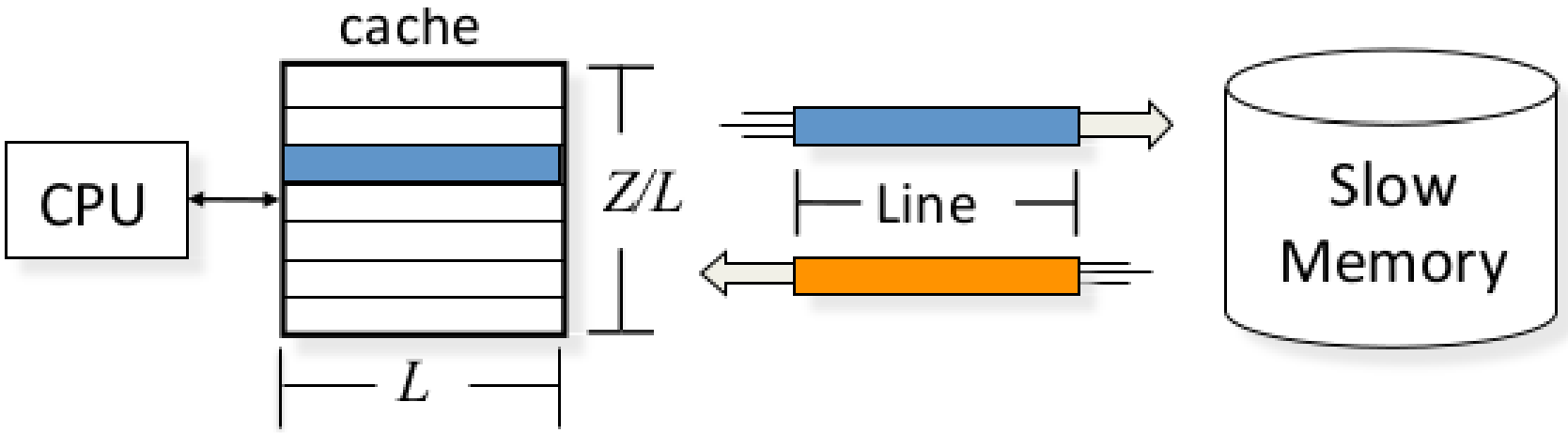
- Computer with a **two-level memory hierarchy**:
  - ↳ an ideal (data) cache of Z words partitioned into Z/L *cache lines*, where L is the number of words per cache line.

The ideal cache model (1/5)



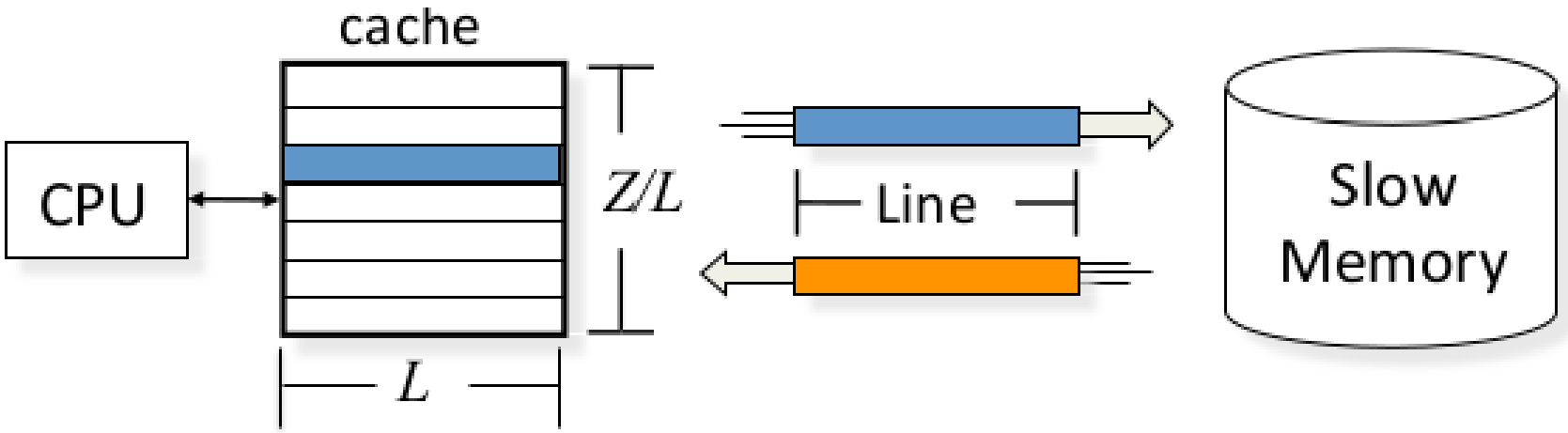
- Computer with a **two-level memory hierarchy**:
  - ↳ an ideal (data) cache of Z words partitioned into Z/L *cache lines*, where L is the number of words per cache line.
  - ↳ an arbitrarily large main memory.

The ideal cache model (1/5)



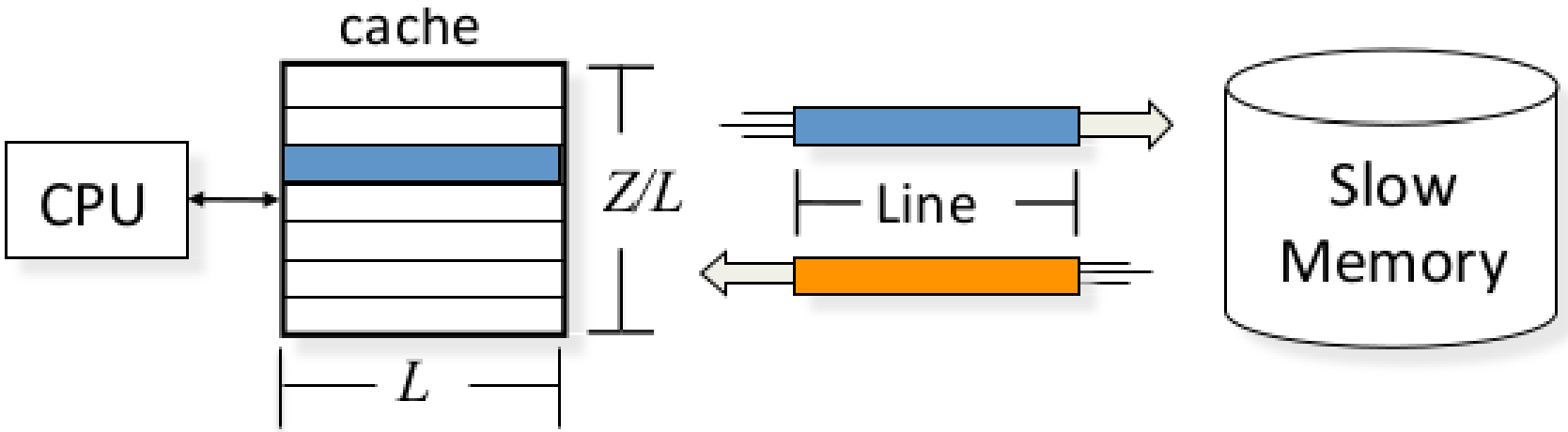
- Computer with a **two-level memory hierarchy**:
  - ↳ an ideal (data) cache of Z words partitioned into Z/L *cache lines*, where L is the number of words per cache line.
  - ↳ an arbitrarily large main memory.
- Data moved between cache and main memory are always cache lines.

The ideal cache model (1/5)



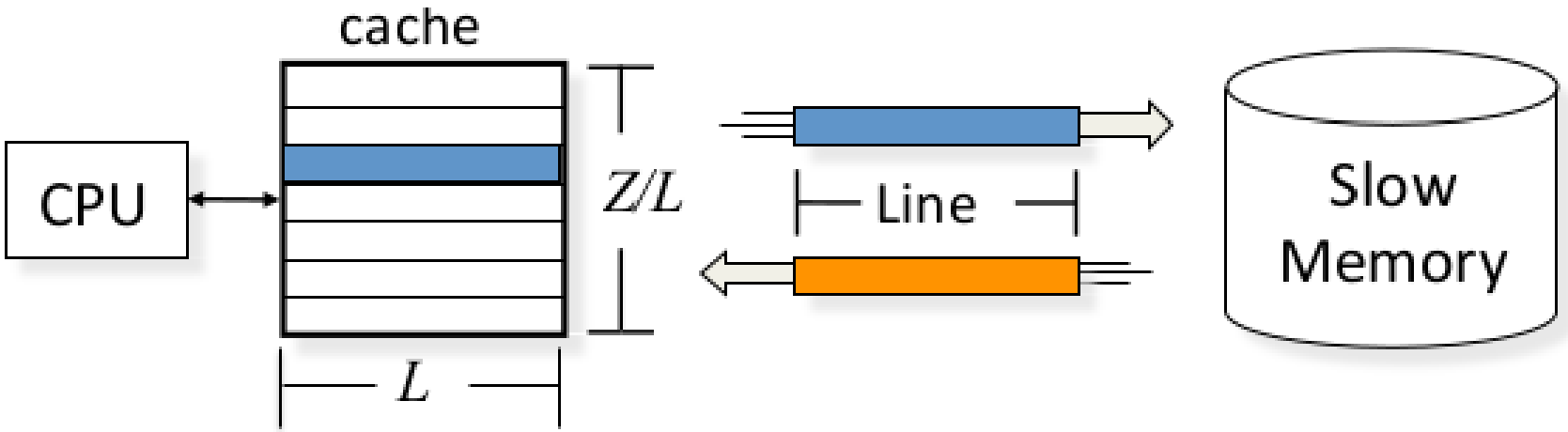
- Computer with a **two-level memory hierarchy**:
  - ↳ an ideal (data) cache of Z words partitioned into Z/L *cache lines*, where L is the number of words per cache line.
  - ↳ an arbitrarily large main memory.
- Data moved between cache and main memory are always cache lines.
- The cache is **tall**, that is, Z is much larger than L, say $Z \in \Omega(L^2)$.

The ideal cache model (2/5)



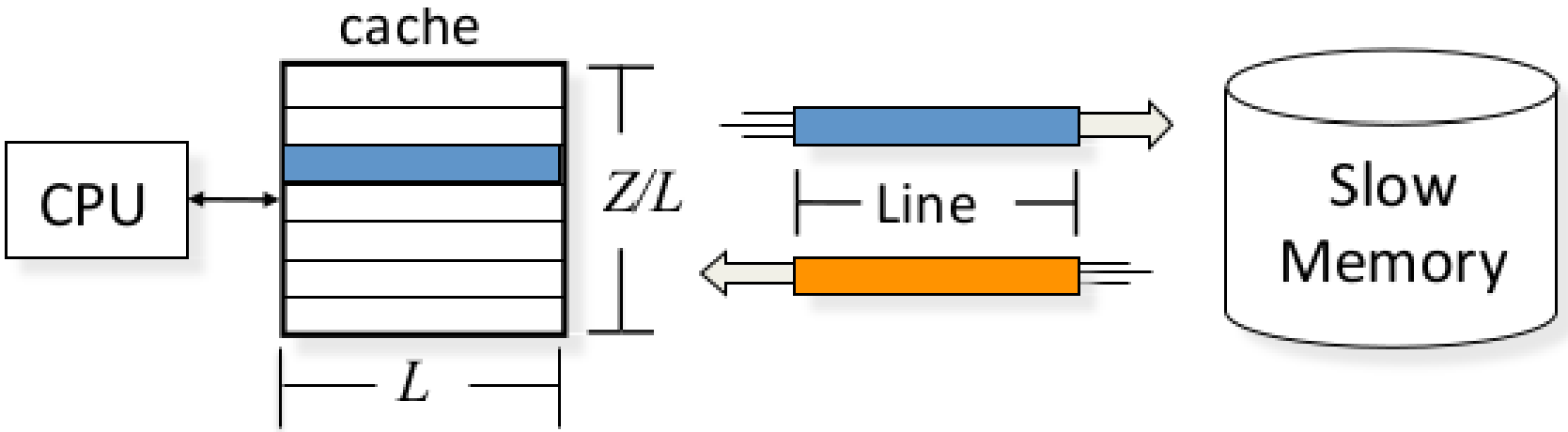
- The processor can only reference words that reside in the cache.

The ideal cache model (2/5)



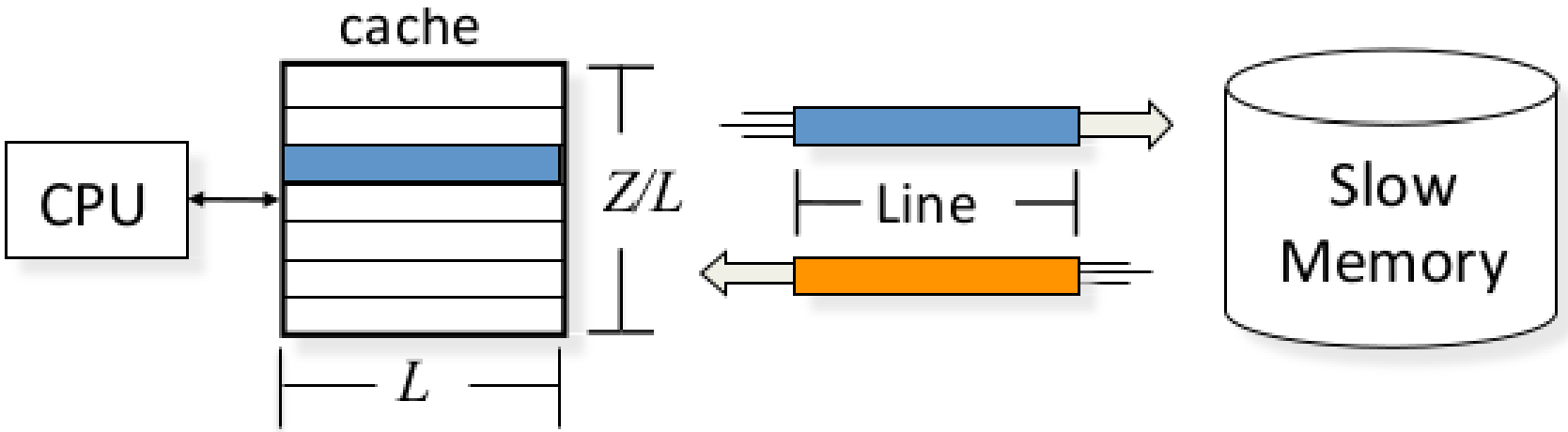
- The processor can only reference words that reside in the cache.
- If the referenced word belongs to a line already in cache, a **cache hit** occurs, and the word is delivered to the processor.

The ideal cache model (2/5)



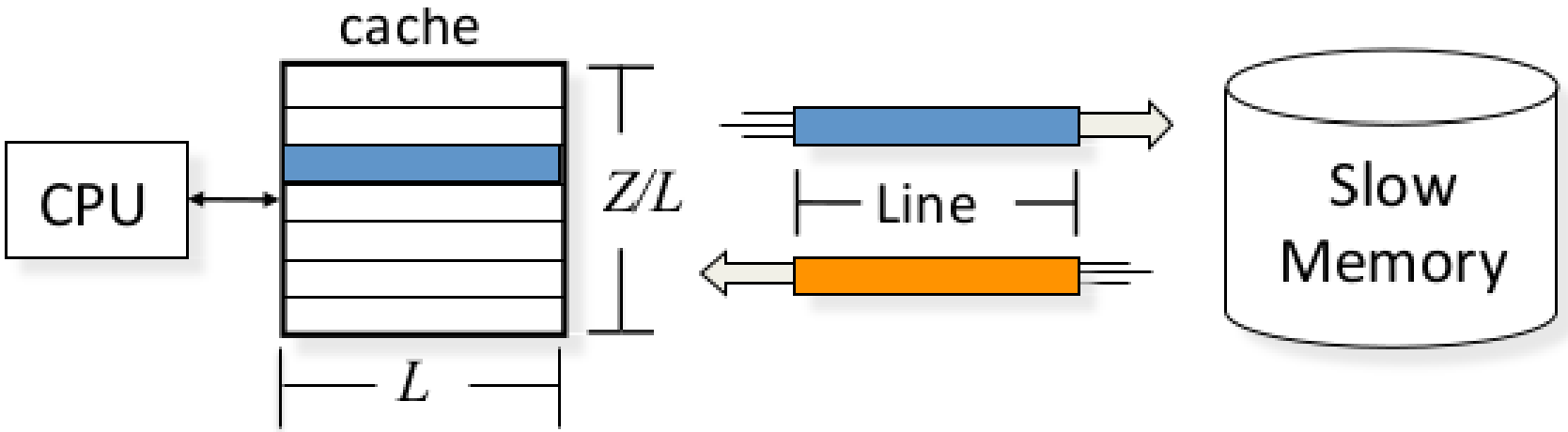
- The processor can only reference words that reside in the cache.
- If the referenced word belongs to a line already in cache, a **cache hit** occurs, and the word is delivered to the processor.
- Otherwise, a **cache miss** occurs, and the line is fetched and installed into the cache.

The ideal cache model (3/5)



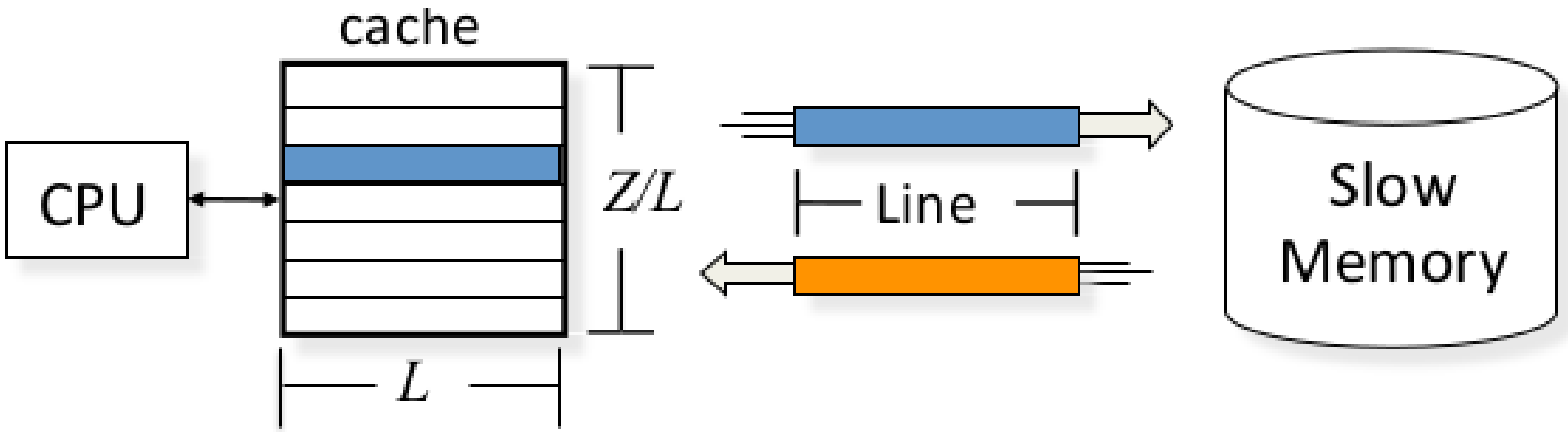
- The ideal cache is **fully associative**: cache lines can be stored anywhere in the cache.

The ideal cache model (3/5)



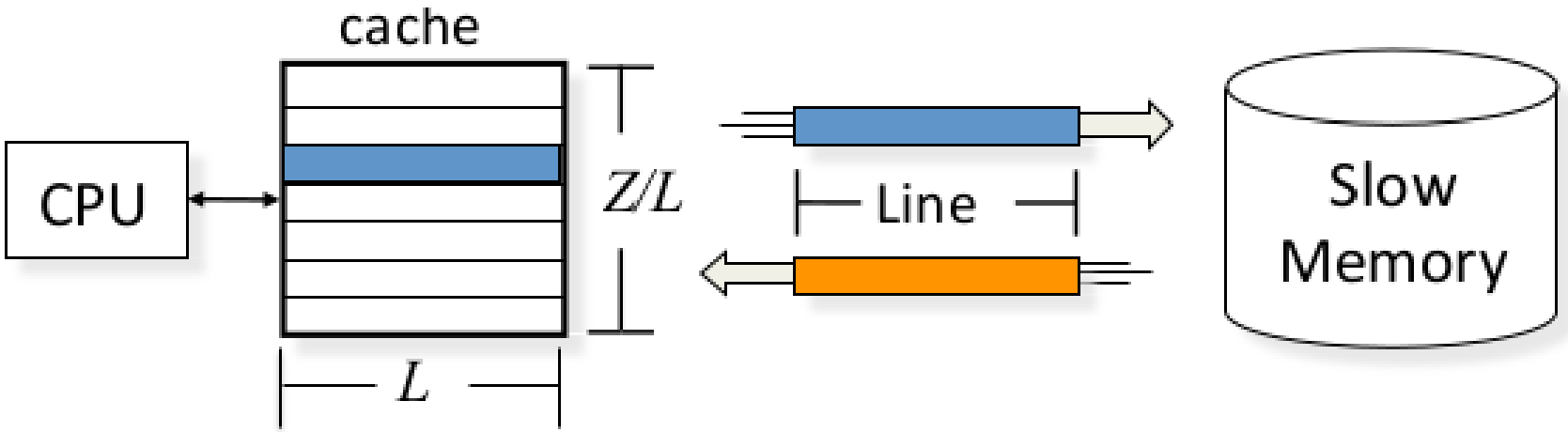
- The ideal cache is **fully associative**: cache lines can be stored anywhere in the cache.
- The ideal cache uses the **optimal off-line strategy of replacement**, that is, replacing the cache line whose next access is furthest in the future

The ideal cache model (3/5)



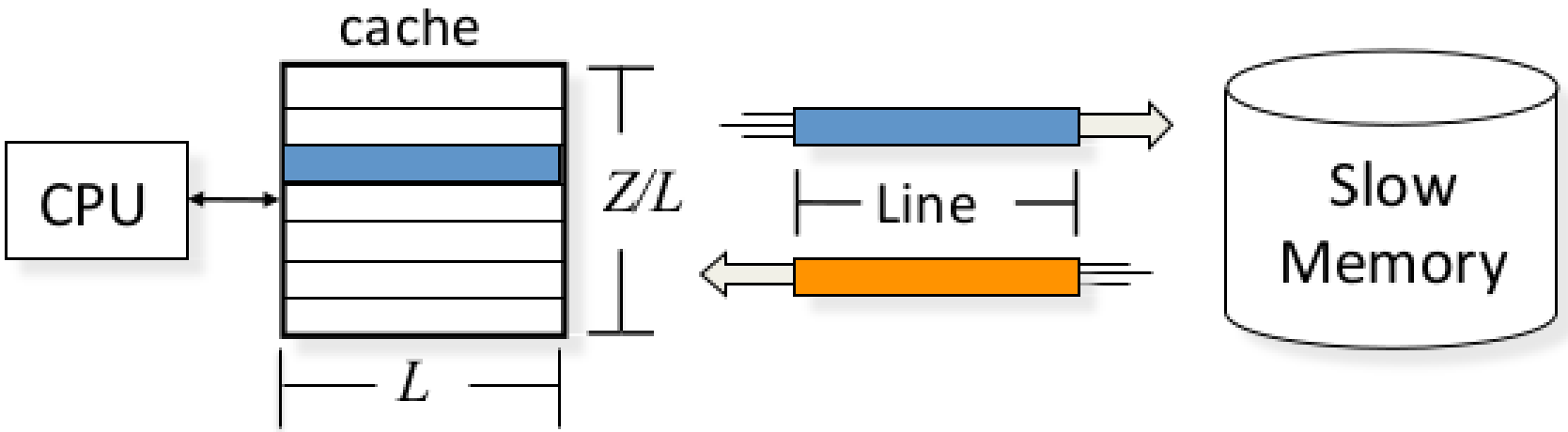
- The ideal cache is **fully associative**: cache lines can be stored anywhere in the cache.
- The ideal cache uses the **optimal off-line strategy of replacement**, that is, replacing the cache line whose next access is furthest in the future
- This strategy exploits temporal locality perfectly.

The ideal cache model (3/5)



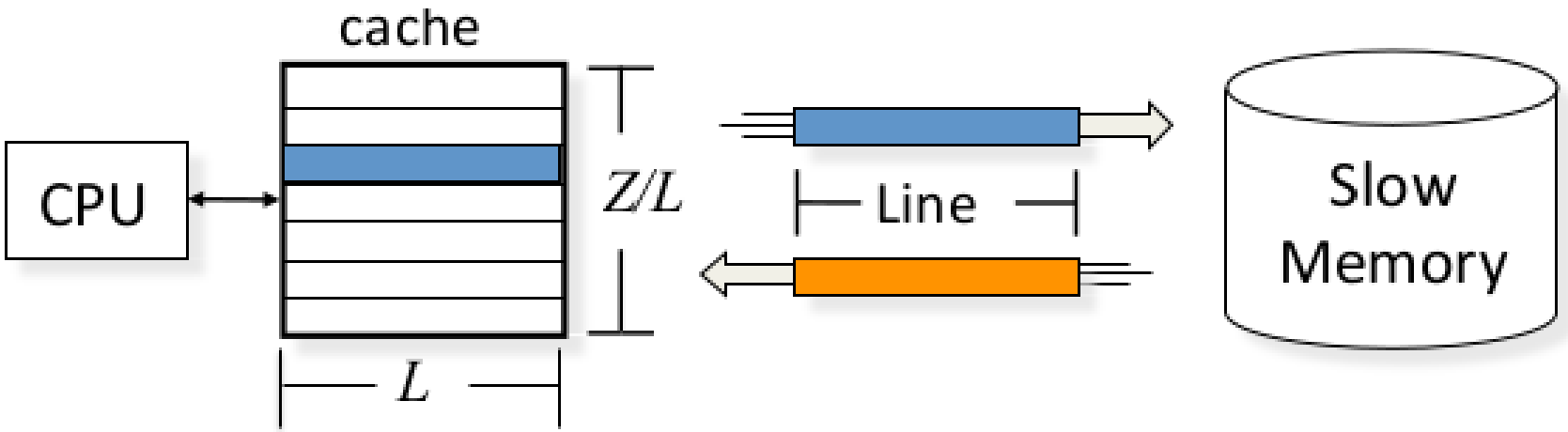
- The ideal cache is **fully associative**: cache lines can be stored anywhere in the cache.
- The ideal cache uses the **optimal off-line strategy of replacement**, that is, replacing the cache line whose next access is furthest in the future
- This strategy exploits temporal locality perfectly.
- While full associativity and the optimal off-line strategy of replacement cannot be implemented, experimental and theoretical results show that they can be approximated in a satisfactory manner.

The ideal cache model (4/5)



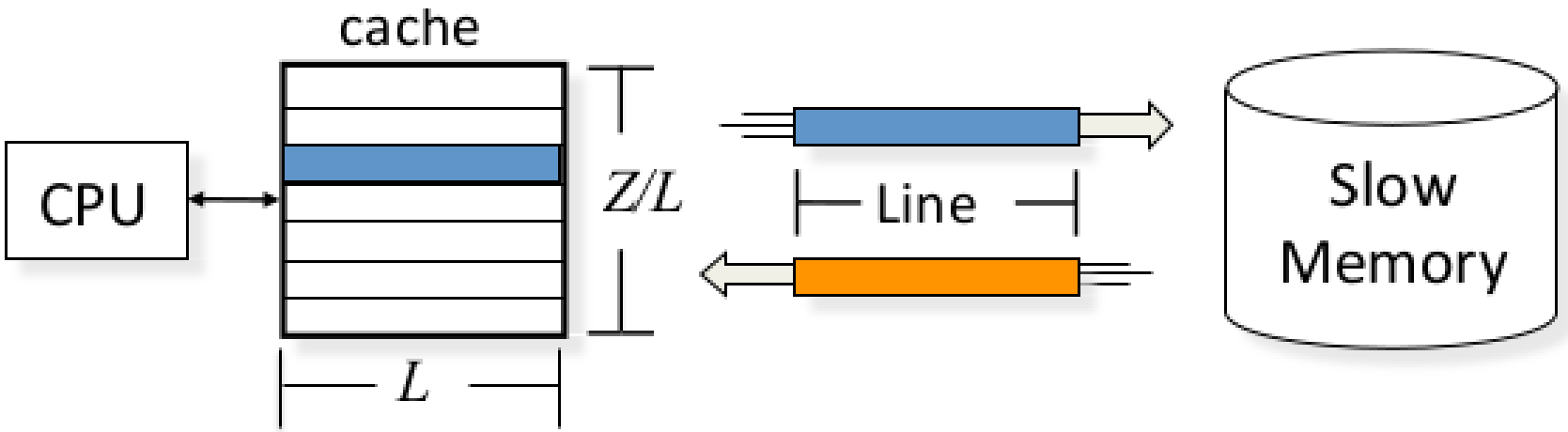
- For an algorithm with an input of size n, the ideal-cache model uses two complexity measures:

The ideal cache model (4/5)



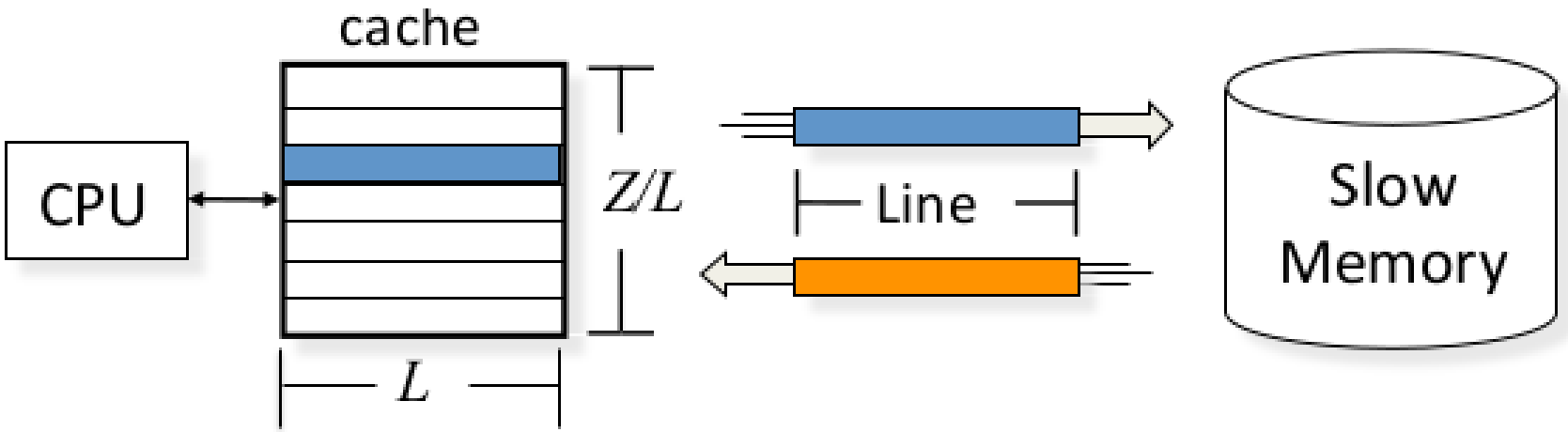
- For an algorithm with an input of size n, the ideal-cache model uses two complexity measures:
  - the **work complexity** $W(n)$, which is its conventional running time in a RAM model.

The ideal cache model (4/5)



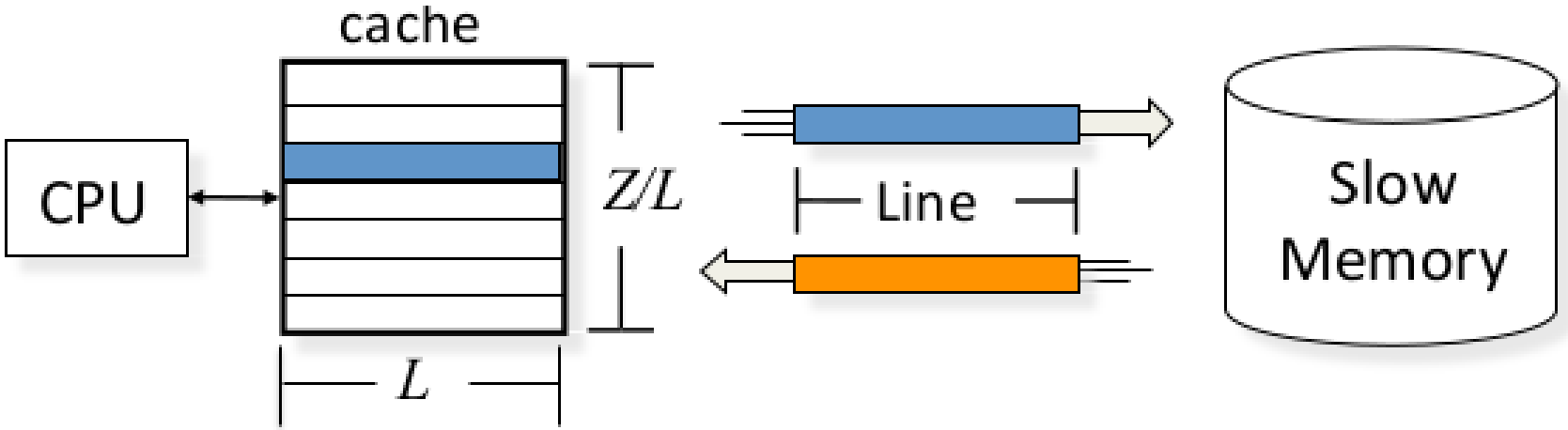
- For an algorithm with an input of size n, the ideal-cache model uses two complexity measures:
  - the **work complexity** $W(n)$, which is its conventional running time in a RAM model.
  - the **cache complexity** $Q(n; Z, L)$, the number of cache misses it incurs (as a function of the size Z and line length L of the ideal cache).

The ideal cache model (4/5)



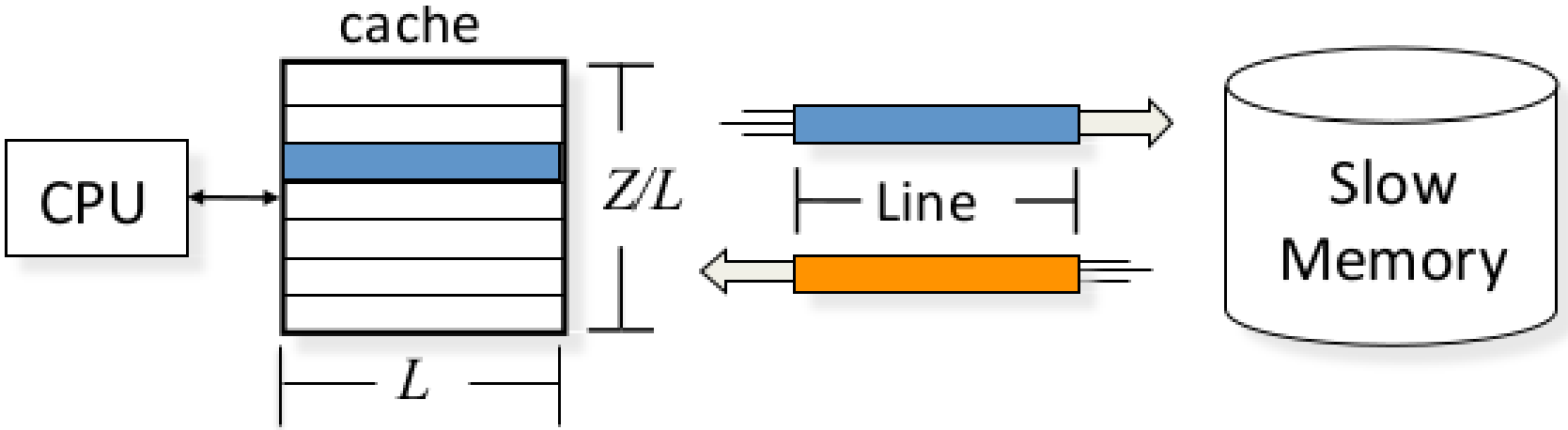
- For an algorithm with an input of size n, the ideal-cache model uses two complexity measures:
  - the **work complexity** $W(n)$, which is its conventional running time in a RAM model.
  - the **cache complexity** $Q(n; Z, L)$, the number of cache misses it incurs (as a function of the size Z and line length L of the ideal cache).
  - When Z and L are clear from context, we simply write $Q(n)$ instead of $Q(n; Z, L)$.

The ideal cache model (5/5)



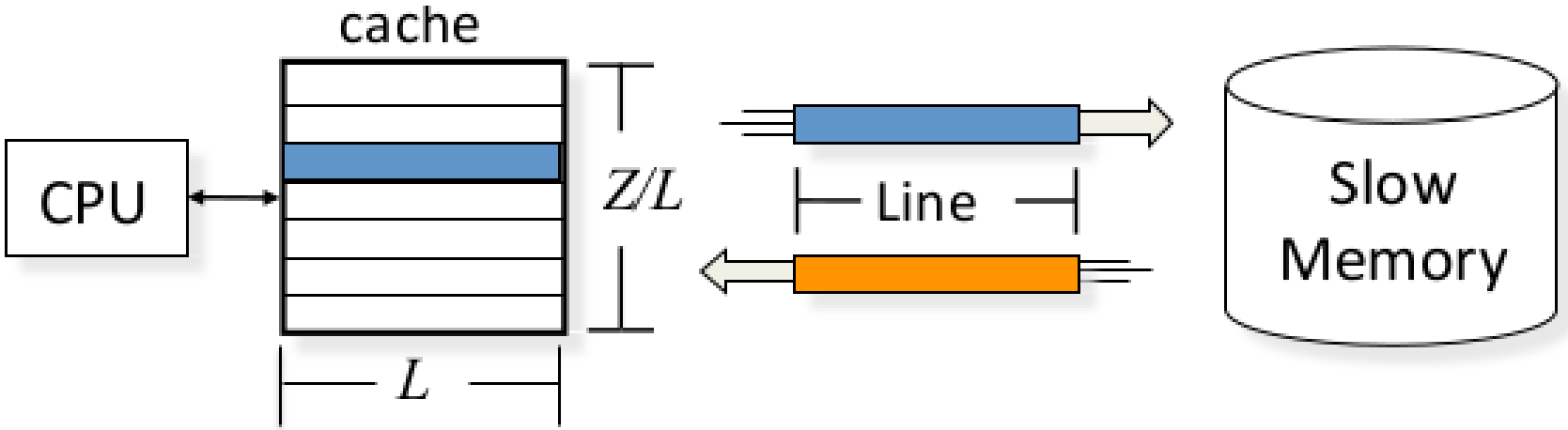
- An algorithm is said to be **cache aware** if its behavior (and thus performances) can be tuned (and thus depend on) on the particular cache size and line length of the targeted machine.

The ideal cache model (5/5)



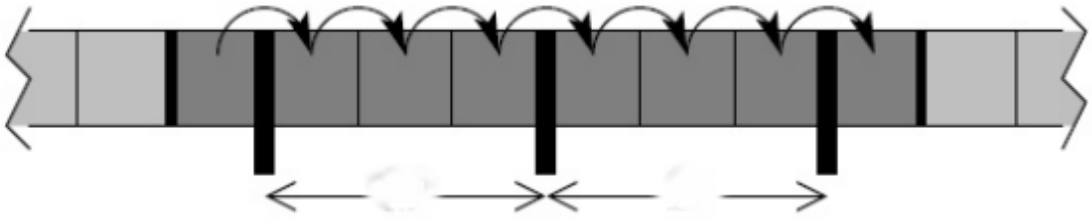
- An algorithm is said to be **cache aware** if its behavior (and thus performances) can be tuned (and thus depend on) on the particular cache size and line length of the targeted machine.
- Otherwise the algorithm is **cache oblivious**.

The ideal cache model (5/5)



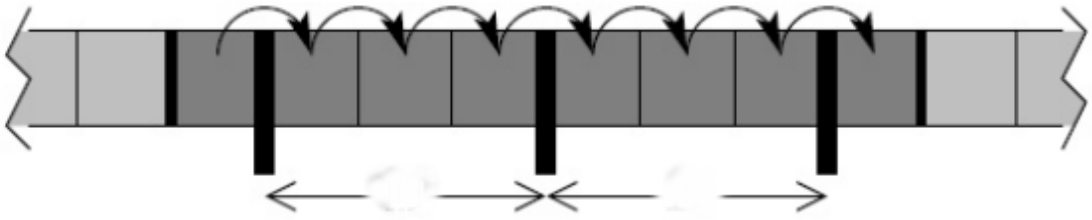
- An algorithm is said to be **cache aware** if its behavior (and thus performances) can be tuned (and thus depend on) on the particular cache size and line length of the targeted machine.
- Otherwise the algorithm is **cache oblivious**.
- Cache oblivious naturally performs well on hierarchical memories.

Scanning



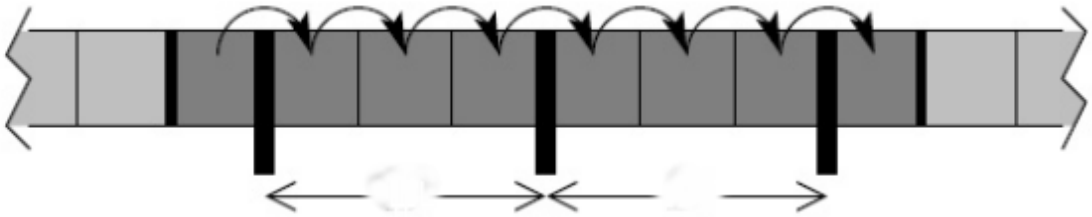
- Scanning n words stored in a contiguous segment of memory with cache-line size L costs at most $\lceil n/L \rceil + 1$ cache misses.

Scanning



- Scanning n words stored in a contiguous segment of memory with cache-line size L costs at most $\lceil n/L \rceil + 1$ cache misses.
- If this vector of n words is aligned in memory, then this estimate is simply $\lceil n/L \rceil$.

Scanning



- Scanning n words stored in a contiguous segment of memory with cache-line size L costs at most $\lceil n/L \rceil + 1$ cache misses.
- If this vector of n words is aligned in memory, then this estimate is simply $\lceil n/L \rceil$.

Proof.

- Let (q, r) be the quotient and remainder in the integer division of n by L.
- Let u (resp. w) be the total number of words stored in cache-lines fully (not fully) used by those n consecutive words. Thus, we have $n = u + w$. Three cases arise.
  1. if $w = 0$ then $(q, r) = (\lfloor n/L \rfloor, 0)$ and the scanning costs exactly q; thus the conclusion is clear since $\lceil n/L \rceil = \lfloor n/L \rfloor$ in this case.
  2. if $0 < w < L$ then $(q, r) = (\lfloor n/L \rfloor, w)$ and the scanning cost is at most $q + 2$; the conclusion is clear since $\lceil n/L \rceil = \lfloor n/L \rfloor + 1$ in this case.
  3. if $L \leq w < 2L$ then $(q, r) = (\lfloor n/L \rfloor, w - L)$ and the scanning cost is at most $q + 1$; the conclusion is clear again.

Adding vectors

- Consider $m \geq 2$ vectors $V_1, \ldots, V_m$ of size $n \geq 1$ aligned in memory.

Adding vectors

- Consider $m \geq 2$ vectors $V_1, \ldots, V_m$ of size $n \geq 1$ aligned in memory.
- Consider $m-1$ scalars $\alpha_1, \ldots, \alpha_{m-1}$, stored in a contiguous segment of memory in $m-1$ words.

Adding vectors

- Consider $m \geq 2$ vectors $V_1, \ldots, V_m$ of size $n \geq 1$ aligned in memory.
- Consider $m-1$ scalars $\alpha_1, \ldots, \alpha_{m-1}$, stored in a contiguous segment of memory in $m-1$ words.
- Assume that the ideal cache has at least $\lceil m/L \rceil + 4$ cache-lines.

Adding vectors

- Consider $m \geq 2$ vectors $V_1, \ldots, V_m$ of size $n \geq 1$ aligned in memory.
- Consider $m-1$ scalars $\alpha_1, \ldots, \alpha_{m-1}$, stored in a contiguous segment of memory in $m-1$ words.
- Assume that the ideal cache has at least $\lceil m/L \rceil + 4$ cache-lines.
- Then, computing the linear combination $\alpha_1 V_1 + \cdots + \alpha_{m-1} V_{m-1}$ and writing it to V_m can be done in no more cache misses than those required for scanning $V_1, \ldots, V_m, \alpha_1, \ldots, \alpha_{m-1}$,

Adding vectors

- Consider $m \geq 2$ vectors $V_1, \ldots, V_m$ of size $n \geq 1$ aligned in memory.
- Consider $m-1$ scalars $\alpha_1, \ldots, \alpha_{m-1}$, stored in a contiguous segment of memory in $m-1$ words.
- Assume that the ideal cache has at least $\lceil m/L \rceil + 4$ cache-lines.
- Then, computing the linear combination $\alpha_1 V_1 + \cdots + \alpha_{m-1} V_{m-1}$ and writing it to V_m can be done in no more cache misses than those required for scanning $V_1, \ldots, V_m, \alpha_1, \ldots, \alpha_{m-1}$,
- thus, within $m\lceil n/L \rceil + \lceil m/L \rceil + 1$ cache misses.

Adding vectors

- Consider $m \geq 2$ vectors $V_1, \ldots, V_m$ of size $n \geq 1$ aligned in memory.
- Consider $m-1$ scalars $\alpha_1, \ldots, \alpha_{m-1}$, stored in a contiguous segment of memory in $m-1$ words.
- Assume that the ideal cache has at least $\lceil m/L \rceil + 4$ cache-lines.
- Then, computing the linear combination $\alpha_1 V_1 + \cdots + \alpha_{m-1} V_{m-1}$ and writing it to V_m can be done in no more cache misses than those required for scanning $V_1, \ldots, V_m, \alpha_1, \ldots, \alpha_{m-1}$,
- thus, within $m\lceil n/L \rceil + \lceil m/L \rceil + 1$ cache misses.

Proof.

- We first load $\alpha_1, \ldots, \alpha_{m-1}$ into the cache, thus using at most $\lceil m/L \rceil + 1$ cache-lines.
- In the pseudo-code below, vector indexing starts at 0.
  1. For b with $0 \leq b \leq \lfloor n/L \rfloor$, for each j with $1 \leq j < m$, for each i with $0 \leq i < L$ do:
     1. $k := b * L + i$,
     2. if $k < n$ then $V_m[k] := V_m[k] + \alpha_j V_j[k]$
- Use the optimal replacement policy and the fact that vectors are aligned in memory

Outline

Counting sort: the algorithm

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- *Counting sort* takes as input a collection of n items, each of which known by a key in the range $0 \cdots k$.

Counting sort: the algorithm

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- *Counting sort* takes as input a collection of n items, each of which known by a key in the range $0 \cdots k$.
- The algorithm computes a *histogram* of the number of times each key occurs.

Counting sort: the algorithm

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- *Counting sort* takes as input a collection of n items, each of which known by a key in the range $0 \cdots k$.
- The algorithm computes a *histogram* of the number of times each key occurs.
- Then performs a *prefix sum* to compute positions in the output.

Counting sort: cache complexity analysis with short explanations

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

1. n/L to compute k.

Counting sort: cache complexity analysis with short explanations

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

1. n/L to compute k.
2. k/L cache misses to initialize `Count`.

Counting sort: cache complexity analysis with short explanations

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

1. n/L to compute k.
2. k/L cache misses to initialize `Count`.
3. $n/L + n$ cache misses for the histogram (worst case).

Counting sort: cache complexity analysis with short explanations

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

1. n/L to compute k.
2. k/L cache misses to initialize `Count`.
3. $n/L + n$ cache misses for the histogram (worst case).
4. k/L cache misses for the prefix sum.

Counting sort: cache complexity analysis with short explanations

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

1. n/L to compute k.
2. k/L cache misses to initialize `Count`.
3. $n/L + n$ cache misses for the histogram (worst case).
4. k/L cache misses for the prefix sum.
5. $n/L + n + n$ cache misses for building `Output` (worst case).

Counting sort: cache complexity analysis with short explanations

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

1. n/L to compute k.
2. k/L cache misses to initialize `Count`.
3. $n/L + n$ cache misses for the histogram (worst case).
4. k/L cache misses for the prefix sum.
5. $n/L + n + n$ cache misses for building `Output` (worst case).
6. Total: $3n+3n/L + 2k/L$ cache misses (worst case).

Counting sort: cache complexity analysis with detailed explanations

1. n/L to compute k: this can be done by traversing the `items` linearly.

Counting sort: cache complexity analysis with detailed explanations

1. n/L to compute k: this can be done by traversing the `items` linearly.
2. k/L cache misses to initialize `Count`: this can be done by traversing the `Count` linearly (with a stride of 1).

Counting sort: cache complexity analysis with detailed explanations

1. n/L to compute k: this can be done by traversing the `items` linearly.
2. k/L cache misses to initialize `Count`: this can be done by traversing the `Count` linearly (with a stride of 1).
3. $n/L + n$ cache misses for the histogram (worst case): accesses in `items` are linear but accesses in `Count` are potentially random.

Counting sort: cache complexity analysis with detailed explanations

1. n/L to compute k: this can be done by traversing the `items` linearly.
2. k/L cache misses to initialize `Count`: this can be done by traversing the `Count` linearly (with a stride of 1).
3. $n/L + n$ cache misses for the histogram (worst case): accesses in `items` are linear but accesses in `Count` are potentially random.
4. k/L cache misses for the prefix sum: accesses in `Count` are linear.

Counting sort: cache complexity analysis with detailed explanations

1. n/L to compute k: this can be done by traversing the `items` linearly.
2. k/L cache misses to initialize `Count`: this can be done by traversing the `Count` linearly (with a stride of 1).
3. $n/L + n$ cache misses for the histogram (worst case): accesses in `items` are linear but accesses in `Count` are potentially random.
4. k/L cache misses for the prefix sum: accesses in `Count` are linear.
5. $n/L + n + n$ cache misses for building `Output` (worst case): accesses in `items` are linear but accesses in `Output` and `Count` are potentially random.

Counting sort: cache complexity analysis with detailed explanations

1. n/L to compute k: this can be done by traversing the `items` linearly.
2. k/L cache misses to initialize `Count`: this can be done by traversing the `Count` linearly (with a stride of 1).
3. $n/L + n$ cache misses for the histogram (worst case): accesses in `items` are linear but accesses in `Count` are potentially random.
4. k/L cache misses for the prefix sum: accesses in `Count` are linear.
5. $n/L + n + n$ cache misses for building `Output` (worst case): accesses in `items` are linear but accesses in `Output` and `Count` are potentially random.
6. Total: $3n$+$3n/L + 2k/L$ cache misses (worst case).

Counting sort has a poor spatial locality

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- For n large enough: $Q(n; Z, L) = 3n + 3n/L + 2k/L$ cache misses (worst case).

Counting sort has a poor spatial locality

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- For n large enough: $Q(n; Z, L) = 3n + 3n/L + 2k/L$ cache misses (worst case).
- The possibly random distribution of the input values creates possibly many non-cold misses, see counting_sort.pdf for an animation.

How to fix the poor data locality of counting sort?

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- Recall that our worst case is $3n+3n/L+2k/L$ cache misses.

How to fix the poor data locality of counting sort?

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- Recall that our worst case is $3n+3n/L+2k/L$ cache misses.
- The troubles come from the irregular accesses which experience capacity misses and conflict misses.

How to fix the poor data locality of counting sort?

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- Recall that our worst case is $3n+3n/L+2k/L$ cache misses.
- The troubles come from the irregular accesses which experience capacity misses and conflict misses.
- To solve this problem, we preprocess the input so that counting sort is applied in succession to several smaller input item sets with smaller value ranges.

How to fix the poor data locality of counting sort?

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- Recall that our worst case is $3n+3n/L+2k/L$ cache misses.
- The troubles come from the irregular accesses which experience capacity misses and conflict misses.
- To solve this problem, we preprocess the input so that counting sort is applied in succession to several smaller input item sets with smaller value ranges.
- To put it simply, so that k and n are small enough for `Output` and `Count` to incur cold misses only.

Counting sort: bukecting the input

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

- Intention: after preprocessing, the arrays `Count` and `Output` incur **cold misses only**, , see counting_sort_bucket.pdf for an animation.

Counting sort: bukecting the input

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

- Intention: after preprocessing, the arrays `Count` and `Output` incur **cold misses only**, , see counting_sort_bucket.pdf for an animation.
- To this end we choose a parameter m (more on this later) such that, after preprocessing:

Counting sort: bukecting the input

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

- Intention: after preprocessing, the arrays `Count` and `Output` incur **cold misses only**, , see counting_sort_bucket.pdf for an animation.
- To this end we choose a parameter m (more on this later) such that, after preprocessing:
  1. any key in the range $[ih, (i+1)h-1]$ is always $\leq$ any key in the range $[(i+1)h, (i+2)h-1]$, for $i = 0 \cdots m-2$, with $h = k/m$,

Counting sort: bukecting the input

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

- Intention: after preprocessing, the arrays `Count` and `Output` incur **cold misses only**, , see counting_sort_bucket.pdf for an animation.
- To this end we choose a parameter m (more on this later) such that, after preprocessing:
  1. any key in the range $[ih, (i+1)h-1]$ is always $\leq$ any key in the range $[(i+1)h, (i+2)h-1]$, for $i = 0 \cdots m-2$, with $h = k/m$,
  2. `bucketsize` and m cache-lines from `bucketedinput` all fit in cache. That is, counting cache-lines, $mL + m \leq Z$.

Counting sort: cache complexity with bukecting

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

1. $3m/L + n/L$ caches misses to compute `bucketsize`

Counting sort: cache complexity with bukecting

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

1. $3m/L + n/L$ caches misses to compute `bucketsize`
2. **Key observation:** `bucketedinput` is traversed regularly by segment.

Counting sort: cache complexity with bukecting

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

1. $3m/L + n/L$ caches misses to compute `bucketsize`
2. **Key observation:** `bucketedinput` is traversed regularly by segment.
3. Hence, $2n/L + m + m/L$ caches misses to compute `bucketedinput`

Counting sort: cache complexity with bukecting

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

1. $3m/L + n/L$ caches misses to compute `bucketsize`
2. **Key observation:** `bucketedinput` is traversed regularly by segment.
3. Hence, $2n/L + m + m/L$ caches misses to compute `bucketedinput`
4. Preprocessing: $3n/L + 4m/L + m$ cache misses.

Counting sort: cache complexity with bukecting: explanations

1. $3m/L + n/L$ caches misses to compute `bucketsize`:
   - ↳ m/L to set each cell of `bucketsize` to zero,

Counting sort: cache complexity with bukecting: explanations

1. $3m/L + n/L$ caches misses to compute `bucketsize`:
   - ↳ m/L to set each cell of `bucketsize` to zero,
   - ↳ $m/L + n/L$ for the first `for loop`,

Counting sort: cache complexity with bukecting: explanations

1. $3m/L + n/L$ caches misses to compute `bucketsize`:
   - ↳ m/L to set each cell of `bucketsize` to zero,
   - ↳ $m/L + n/L$ for the first `for loop`,
   - ↳ m/L for the second `for loop`.

Counting sort: cache complexity with bukecting: explanations

1. $3m/L + n/L$ caches misses to compute `bucketsize`:
   - ↳ m/L to set each cell of `bucketsize` to zero,
   - ↳ $m/L + n/L$ for the first `for loop`,
   - ↳ m/L for the second `for loop`.
2. **Key observation:** `bucketedinput` is traversed regularly by segment:

Counting sort: cache complexity with bukecting: explanations

1. $3m/L + n/L$ caches misses to compute `bucketsize`:
   - ↳ m/L to set each cell of `bucketsize` to zero,
   - ↳ $m/L + n/L$ for the first `for loop`,
   - ↳ m/L for the second `for loop`.
2. **Key observation:** `bucketedinput` is traversed regularly by segment:

   - ↳ So writing `bucketedinput` means writing (in a linear traversal) m consecutive arrays, of possibly different sizes, but with total size n.

Counting sort: cache complexity with bukecting: explanations

1. $3m/L + n/L$ caches misses to compute `bucketsize`:
   - ↳ m/L to set each cell of `bucketsize` to zero,
   - ↳ $m/L + n/L$ for the first `for loop`,
   - ↳ m/L for the second `for loop`.
2. **Key observation:** `bucketedinput` is traversed regularly by segment:

   - ↳ So writing `bucketedinput` means writing (in a linear traversal) m consecutive arrays, of possibly different sizes, but with total size n.
   - ↳ Thus, because of possible misalignments between those arrays and their cache-lines, this writing procedure can yield $n/L + m$ cache misses (and not just n/L).

Counting sort: cache complexity with bukecting: explanations

1. $3m/L + n/L$ caches misses to compute `bucketsize`:
   - ↳ m/L to set each cell of `bucketsize` to zero,
   - ↳ $m/L + n/L$ for the first `for loop`,
   - ↳ m/L for the second `for loop`.
2. **Key observation:** `bucketedinput` is traversed regularly by segment:
   - ↳ So writing `bucketedinput` means writing (in a linear traversal) m consecutive arrays, of possibly different sizes, but with total size n.
   - ↳ Thus, because of possible misalignments between those arrays and their cache-lines, this writing procedure can yield $n/L + m$ cache misses (and not just n/L).
3. Hence, $2n/L + m + m/L$ caches misses to compute `bucketedinput`:

Counting sort: cache complexity with bukecting: explanations

1. $3m/L + n/L$ caches misses to compute `bucketsize`:
   - ↳ m/L to set each cell of `bucketsize` to zero,
   - ↳ $m/L + n/L$ for the first `for loop`,
   - ↳ m/L for the second `for loop`.
2. **Key observation:** `bucketedinput` is traversed regularly by segment:
   - ↳ So writing `bucketedinput` means writing (in a linear traversal) m consecutive arrays, of possibly different sizes, but with total size n.
   - ↳ Thus, because of possible misalignments between those arrays and their cache-lines, this writing procedure can yield $n/L + m$ cache misses (and not just n/L).
3. Hence, $2n/L + m + m/L$ caches misses to compute `bucketedinput`:
   - ↳ n/L to read the items,

Counting sort: cache complexity with bukecting: explanations

1. $3m/L + n/L$ caches misses to compute `bucketsize`:
   - ↳ m/L to set each cell of `bucketsize` to zero,
   - ↳ $m/L + n/L$ for the first `for loop`,
   - ↳ m/L for the second `for loop`.
2. **Key observation:** `bucketedinput` is traversed regularly by segment:
   - ↳ So writing `bucketedinput` means writing (in a linear traversal) m consecutive arrays, of possibly different sizes, but with total size n.
   - ↳ Thus, because of possible misalignments between those arrays and their cache-lines, this writing procedure can yield $n/L + m$ cache misses (and not just n/L).
3. Hence, $2n/L + m + m/L$ caches misses to compute `bucketedinput`:
   - ↳ n/L to read the items,
   - ↳ $n/L + m$ to write `bucketedinput`,

Counting sort: cache complexity with bukecting: explanations

1. $3m/L + n/L$ caches misses to compute `bucketsize`:
   - ↳ m/L to set each cell of `bucketsize` to zero,
   - ↳ $m/L + n/L$ for the first `for loop`,
   - ↳ m/L for the second `for loop`.
2. **Key observation:** `bucketedinput` is traversed regularly by segment:
   - ↳ So writing `bucketedinput` means writing (in a linear traversal) m consecutive arrays, of possibly different sizes, but with total size n.
   - ↳ Thus, because of possible misalignments between those arrays and their cache-lines, this writing procedure can yield $n/L + m$ cache misses (and not just n/L).
3. Hence, $2n/L + m + m/L$ caches misses to compute `bucketedinput`:
   - ↳ n/L to read the items,
   - ↳ $n/L + m$ to write `bucketedinput`,
   - ↳ m/L to load `bucketsize`.

Cache friendly counting sort: complete cache complexity analysis

- **Assumption:** the preprocessing creates buckets of average size n/m.

Cache friendly counting sort: complete cache complexity analysis

- **Assumption:** the preprocessing creates buckets of average size n/m.
- After preprocessing, counting sort is applied to each bucket whose values are in a range $[ih, (i+1)h-1]$, for $i = 0 \cdots m-1$.

Cache friendly counting sort: complete cache complexity analysis

- **Assumption:** the preprocessing creates buckets of average size n/m.
- After preprocessing, counting sort is applied to each bucket whose values are in a range $[ih, (i+1)h-1]$, for $i = 0 \cdots m-1$.
- To be cache-friendly, this requires, for $i = 0 \cdots m-1$, $h + |\{\texttt{key} \in [ih, (i+1)h-1]\}| < Z$ and $m < Z/(1+L)$. These two are very realistic assumption considering today's cache size.

Cache friendly counting sort: complete cache complexity analysis

- **Assumption:** the preprocessing creates buckets of average size n/m.
- After preprocessing, counting sort is applied to each bucket whose values are in a range $[ih, (i+1)h-1]$, for $i = 0 \cdots m-1$.
- To be cache-friendly, this requires, for $i = 0 \cdots m-1$, $h + |\{\texttt{key} \in [ih, (i+1)h-1]\}| < Z$ and $m < Z/(1+L)$. These two are very realistic assumption considering today's cache size.
- And the total complexity becomes;

$$\begin{aligned} Q_{\text{total}} &= Q_{\text{preprocessing}} + Q_{\text{sorting}} \\ &= Q_{\text{preprocessing}} + m\, Q_{\text{sortingofonebucket}} \\ &= Q_{\text{preprocessing}} + m\,(3\tfrac{n}{m\,\mathbf{L}} + 3\tfrac{n}{mL} + 2\tfrac{k}{mL}) \\ &= Q_{\text{preprocessing}} + 6n/L + 2k/L \\ &= 3n/L + 4m/L + m + 6n/L + 2k/L \\ &= 9n/L + 4m/L + m + 2k/L \end{aligned}$$

Instead of $3n$+$3n/L + 2k/L$ for the naive counting sort.

Counting sort: experimentation

- Experimentation on an *Intel(R) Core(TM) i7 CPU @ 2.93GHz*. It has L2 cache of 8MB.

n	classical counting sort	cache-friendly counting sort (bucketing + sorting)
100000000	13.74	4.66 (= 3.04 + 1.62)
200000000	30.20	9.93 (= 6.16 + 3.77)
300000000	50.19	16.02 (= 9.32 + 6.70)
400000000	71.55	22.13 (= 12.50 +9.63)
500000000	94.32	28.37 (= 15.71 + 12.66)
600000000	116.74	34.61 (= 18.95 + 15.66)

Counting sort: experimentation

- Experimentation on an *Intel(R) Core(TM) i7 CPU @ 2.93GHz*. It has L2 cache of 8MB.
- CPU times in seconds for both classical and cache-friendly counting sort algorithm.

n	classical counting sort	cache-friendly counting sort (bucketing + sorting)
100000000	13.74	4.66 (= 3.04 + 1.62)
200000000	30.20	9.93 (= 6.16 + 3.77)
300000000	50.19	16.02 (= 9.32 + 6.70)
400000000	71.55	22.13 (= 12.50 +9.63)
500000000	94.32	28.37 (= 15.71 + 12.66)
600000000	116.74	34.61 (= 18.95 + 15.66)

Counting sort: experimentation

- Experimentation on an *Intel(R) Core(TM) i7 CPU @ 2.93GHz*. It has L2 cache of 8MB.
- CPU times in seconds for both classical and cache-friendly counting sort algorithm.
- The keys are random machine integers in the range $[0, n]$.

n	classical counting sort	cache-friendly counting sort (bucketing + sorting)
100000000	13.74	4.66 (= 3.04 + 1.62)
200000000	30.20	9.93 (= 6.16 + 3.77)
300000000	50.19	16.02 (= 9.32 + 6.70)
400000000	71.55	22.13 (= 12.50 +9.63)
500000000	94.32	28.37 (= 15.71 + 12.66)
600000000	116.74	34.61 (= 18.95 + 15.66)

Cache-friendly counting sort: extension to sample sort

1. Split the input array into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$ and sort those subarrays recursively.

Cache complexity analysis of Sample sort

- Step 1 costs $\sqrt{n}Q(\sqrt{n})$, Step 4 (expectedly) costs $\sqrt{n}Q(\sqrt{n})$ also and Steps 2, 3, 5 cost $\Theta(n/L)$. Thus, we have:

$$Q(n) = \begin{cases} n/L & \text{if} \quad n < Z \quad \text{(base case)} \\ 2\sqrt{n}Q(\sqrt{n}) + \Theta(n/L) & \text{if} \quad n \geq Z \quad \text{(recurrence)} \end{cases}$$

- This yields $Q(n) \in \Theta(\frac{n}{L}\log_Z(n))$.

Cache-friendly counting sort: extension to sample sort

1. Split the input array into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$ and sort those subarrays recursively.
2. Choose $m := \sqrt{n} - 1$ "good" pivot values $p_1 \leq p_2 \leq \cdots \leq p_m$.

Cache complexity analysis of Sample sort

- Step 1 costs $\sqrt{n}Q(\sqrt{n})$, Step 4 (expectedly) costs $\sqrt{n}Q(\sqrt{n})$ also and Steps 2, 3, 5 cost $\Theta(n/L)$. Thus, we have:

$$Q(n) = \begin{cases} n/L & \text{if} \quad n < Z \quad \text{(base case)} \\ 2\sqrt{n}Q(\sqrt{n}) + \Theta(n/L) & \text{if} \quad n \geq Z \quad \text{(recurrence)} \end{cases}$$

- This yields $Q(n) \in \Theta(\frac{n}{L}\log_Z(n))$.

Cache-friendly counting sort: extension to sample sort

1. Split the input array into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$ and sort those subarrays recursively.
2. Choose $m := \sqrt{n} - 1$ "good" pivot values $p_1 \leq p_2 \leq \cdots \leq p_m$.
3. Distribute subarrays into buckets $B_1, \ldots, B_{m+1}$ according to pivots. Bucket B_i has size $n_i \simeq \sqrt{n}$, expectedly.

Cache complexity analysis of Sample sort

- Step 1 costs $\sqrt{n}Q(\sqrt{n})$, Step 4 (expectedly) costs $\sqrt{n}Q(\sqrt{n})$ also and Steps 2, 3, 5 cost $\Theta(n/L)$. Thus, we have:

$$Q(n) = \begin{cases} n/L & \text{if} \quad n < Z \quad \text{(base case)} \\ 2\sqrt{n}Q(\sqrt{n}) + \Theta(n/L) & \text{if} \quad n \geq Z \quad \text{(recurrence)} \end{cases}$$

- This yields $Q(n) \in \Theta(\frac{n}{L}\log_Z(n))$.

Cache-friendly counting sort: extension to sample sort

1. Split the input array into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$ and sort those subarrays recursively.
2. Choose $m := \sqrt{n} - 1$ "good" pivot values $p_1 \leq p_2 \leq \cdots \leq p_m$.
3. Distribute subarrays into buckets $B_1, \ldots, B_{m+1}$ according to pivots. Bucket B_i has size $n_i \simeq \sqrt{n}$, expectedly.
4. Recursively sort the buckets

Cache complexity analysis of Sample sort

- Step 1 costs $\sqrt{n}Q(\sqrt{n})$, Step 4 (expectedly) costs $\sqrt{n}Q(\sqrt{n})$ also and Steps 2, 3, 5 cost $\Theta(n/L)$. Thus, we have:

$$Q(n) = \begin{cases} n/L & \text{if} \quad n < Z \quad \text{(base case)} \\ 2\sqrt{n}Q(\sqrt{n}) + \Theta(n/L) & \text{if} \quad n \geq Z \quad \text{(recurrence)} \end{cases}$$

- This yields $Q(n) \in \Theta(\frac{n}{L}\log_Z(n))$.

Cache-friendly counting sort: extension to sample sort

1. Split the input array into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$ and sort those subarrays recursively.
2. Choose $m := \sqrt{n} - 1$ "good" pivot values $p_1 \leq p_2 \leq \cdots \leq p_m$.
3. Distribute subarrays into buckets $B_1, \ldots, B_{m+1}$ according to pivots. Bucket B_i has size $n_i \simeq \sqrt{n}$, expectedly.
4. Recursively sort the buckets
5. Copy-concatenate the buckets back to the input array.

Cache complexity analysis of Sample sort

- Step 1 costs $\sqrt{n}Q(\sqrt{n})$, Step 4 (expectedly) costs $\sqrt{n}Q(\sqrt{n})$ also and Steps 2, 3, 5 cost $\Theta(n/L)$. Thus, we have:

$$Q(n) = \begin{cases} n/L & \text{if} \quad n < Z \quad \text{(base case)} \\ 2\sqrt{n}Q(\sqrt{n}) + \Theta(n/L) & \text{if} \quad n \geq Z \quad \text{(recurrence)} \end{cases}$$

- This yields $Q(n) \in \Theta(\frac{n}{L}\log_Z(n))$.

Outline

Matrix transposition: various algorithms (1/2)

- **Matrix transposition problem:** Given an $m \times n$ matrix A stored in a row-major layout, compute and store A^T into an $n \times m$ matrix B also stored in a row-major layout.

Matrix transposition: various algorithms (1/2)

- **Matrix transposition problem:** Given an $m \times n$ matrix A stored in a row-major layout, compute and store A^T into an $n \times m$ matrix B also stored in a row-major layout.
- We shall describe a recursive cache-oblivious algorithm which uses $\Theta(mn)$ work and incurs $\Theta(1 + mn/L)$ cache misses, which is optimal.

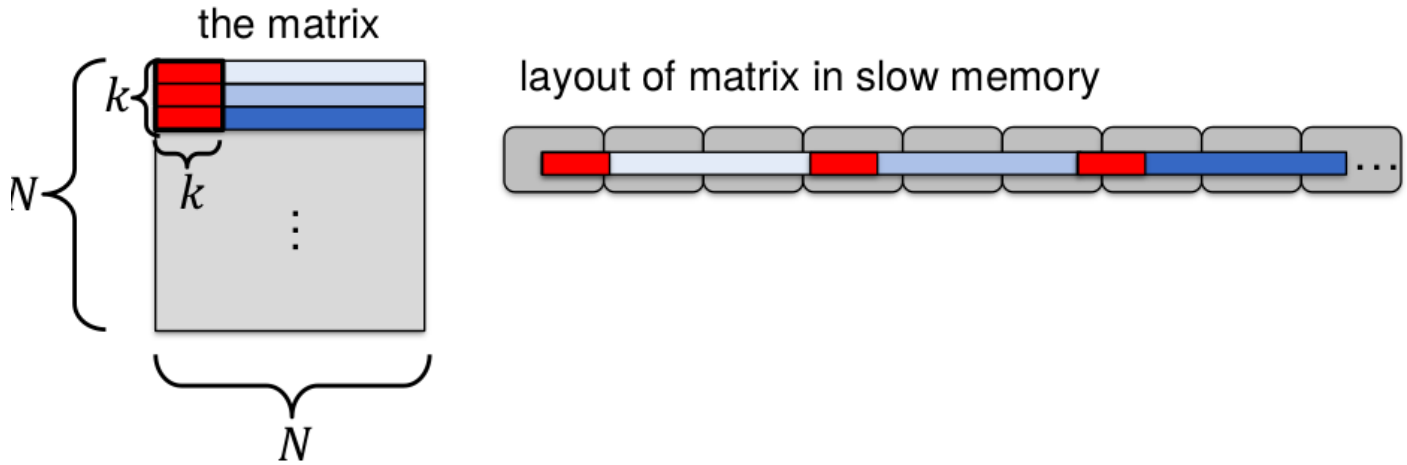
Matrix transposition: various algorithms (1/2)

- **Matrix transposition problem:** Given an $m \times n$ matrix A stored in a row-major layout, compute and store A^T into an $n \times m$ matrix B also stored in a row-major layout.
- We shall describe a recursive cache-oblivious algorithm which uses $\Theta(mn)$ work and incurs $\Theta(1 + mn/L)$ cache misses, which is optimal.
- The straightforward algorithm employing doubly nested loops incurs $\Theta(mn)$ cache misses on one of the matrices when $m \gg Z/L$ and $n \gg Z/L$.

Matrix transposition: various algorithms (1/2)

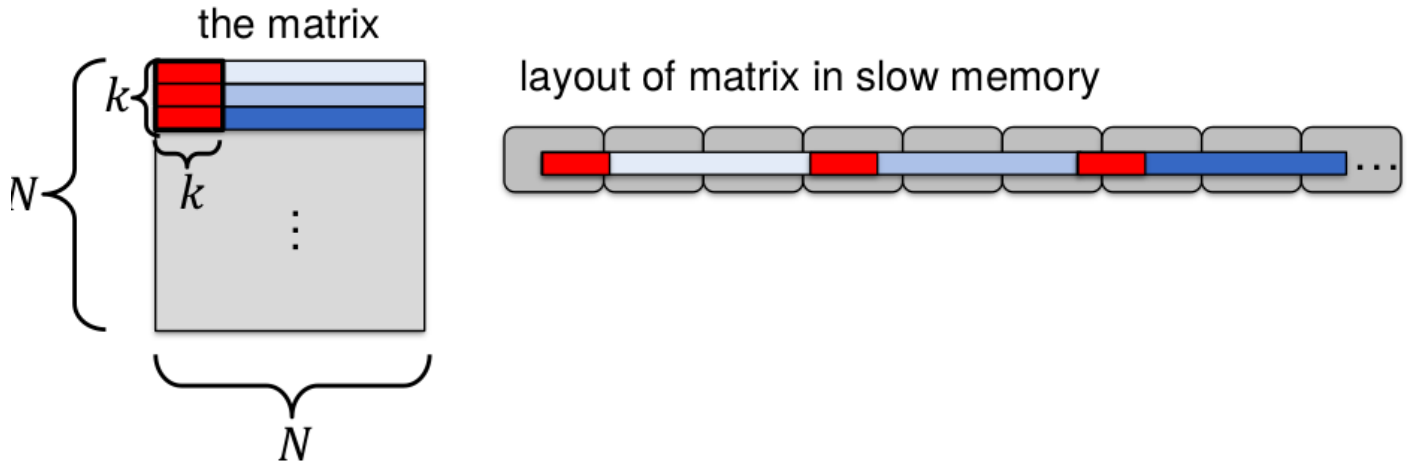
- **Matrix transposition problem:** Given an $m \times n$ matrix A stored in a row-major layout, compute and store A^T into an $n \times m$ matrix B also stored in a row-major layout.
- We shall describe a recursive cache-oblivious algorithm which uses $\Theta(mn)$ work and incurs $\Theta(1 + mn/L)$ cache misses, which is optimal.
- The straightforward algorithm employing doubly nested loops incurs $\Theta(mn)$ cache misses on one of the matrices when $m \gg Z/L$ and $n \gg Z/L$.
- We will also study an apparently good algorithm and use complexity analysis to show that it is even worse than the straightforward algorithm.

Matrix transposition: various algorithms (2/2)



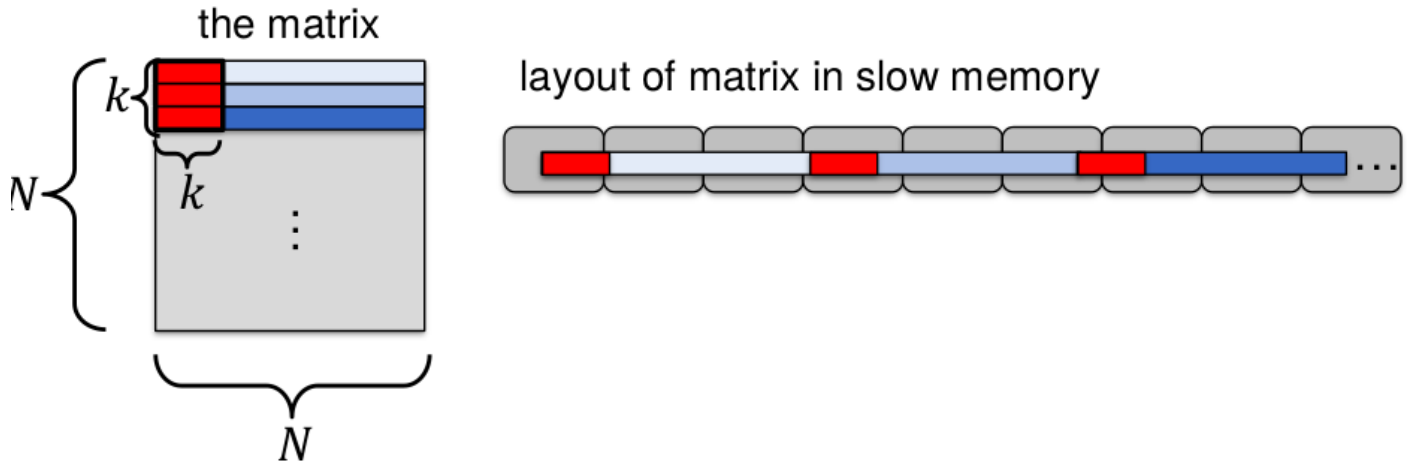
- Assume that multi-dimensional arrays (and in particular dense rectangular matrices) are stored in memory using a row-major layout.

Matrix transposition: various algorithms (2/2)



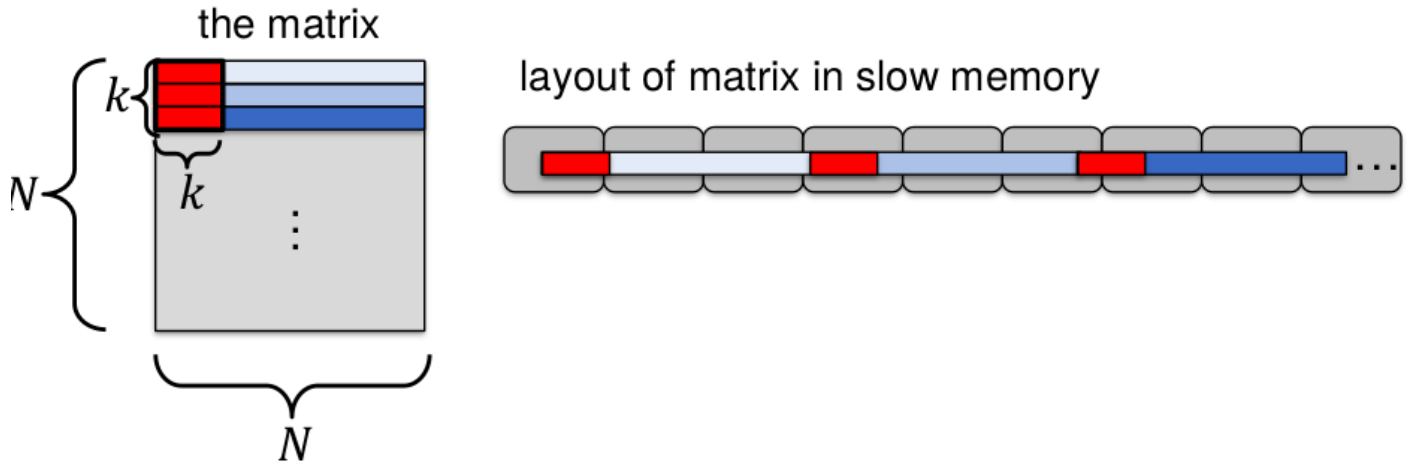
- Assume that multi-dimensional arrays (and in particular dense rectangular matrices) are stored in memory using a row-major layout.
- Assume that each array coefficient is stored on a single word.

Matrix transposition: various algorithms (2/2)



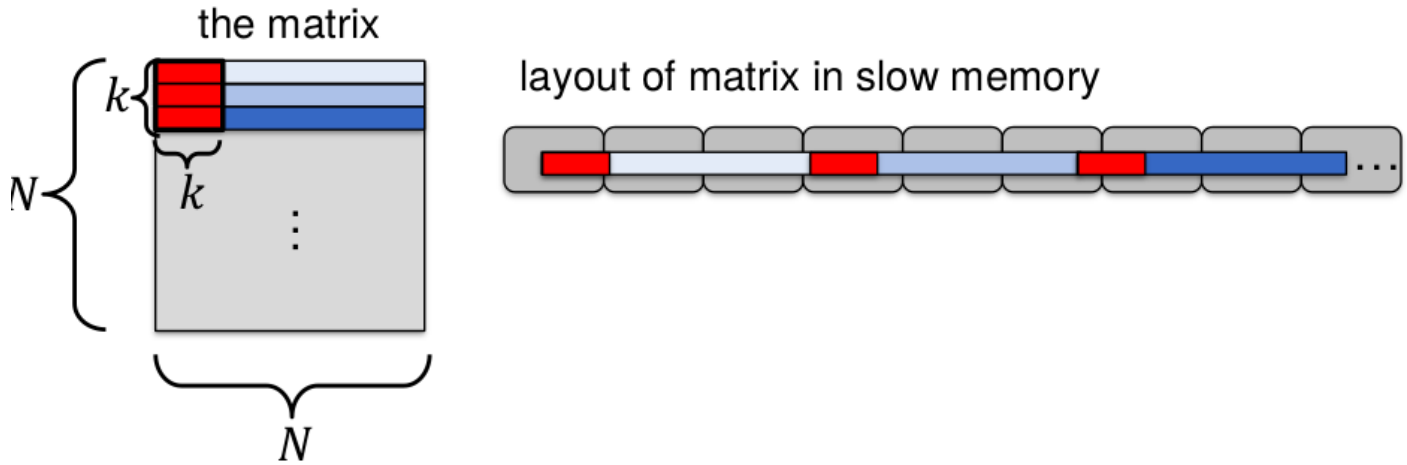
- Assume that multi-dimensional arrays (and in particular dense rectangular matrices) are stored in memory using a row-major layout.
- Assume that each array coefficient is stored on a single word.
- Therefore, reading a $k \times k$ block may incur $k(\lceil k/L \rceil + 1)$ caches misses.

Matrix transposition: various algorithms (2/2)



- Assume that multi-dimensional arrays (and in particular dense rectangular matrices) are stored in memory using a row-major layout.
- Assume that each array coefficient is stored on a single word.
- Therefore, reading a $k \times k$ block may incur $k(\lceil k/L \rceil + 1)$ caches misses.
- In this exercise sheet, determine the cache complexity of the proposed algorithms for transposing a square matrix of order n. Assume n large (say $n > Z$) and n is a power of 2.

Matrix transposition: various algorithms (2/2)



- Assume that multi-dimensional arrays (and in particular dense rectangular matrices) are stored in memory using a row-major layout.
- Assume that each array coefficient is stored on a single word.
- Therefore, reading a $k \times k$ block may incur $k(\lceil k/L \rceil + 1)$ caches misses.
- In this exercise sheet, determine the cache complexity of the proposed algorithms for transposing a square matrix of order n. Assume n large (say $n > Z$) and n is a power of 2.
- Algo 1: $\Theta(n^2)$. Algo 2: $\Theta(\log_2(\frac{n}{Z})\,\frac{n^2}{L})$. Algo 3: $\Theta(n^2/L)$. Proofs and precise estimates below.

Matrix transposition: a first divide-and-conquer (1/4)

- For simplicity, assume that our input matrix A is square of order n and that n is a power of 2, say $n = 2^k$.

Matrix transposition: a first divide-and-conquer (1/4)

- For simplicity, assume that our input matrix A is square of order n and that n is a power of 2, say $n = 2^k$.
- We divide A into four square quadrants of order $n/2$ and we have

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \Rightarrow {}^tA = \begin{pmatrix} {}^tA_{1,1} & {}^tA_{2,1} \\ {}^tA_{1,2} & {}^tA_{2,2} \end{pmatrix}.$$

Matrix transposition: a first divide-and-conquer (1/4)

- For simplicity, assume that our input matrix A is square of order n and that n is a power of 2, say $n = 2^k$.
- We divide A into four square quadrants of order $n/2$ and we have

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \Rightarrow {}^tA = \begin{pmatrix} {}^tA_{1,1} & {}^tA_{2,1} \\ {}^tA_{1,2} & {}^tA_{2,2} \end{pmatrix}.$$

- This observation yields an "in-place" algorithm:

Matrix transposition: a first divide-and-conquer (1/4)

- For simplicity, assume that our input matrix A is square of order n and that n is a power of 2, say $n = 2^k$.
- We divide A into four square quadrants of order $n/2$ and we have

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \Rightarrow {}^tA = \begin{pmatrix} {}^tA_{1,1} & {}^tA_{2,1} \\ {}^tA_{1,2} & {}^tA_{2,2} \end{pmatrix}.$$

- This observation yields an "in-place" algorithm:
  1. If $n = 1$ then return A.

Matrix transposition: a first divide-and-conquer (1/4)

- For simplicity, assume that our input matrix A is square of order n and that n is a power of 2, say $n = 2^k$.
- We divide A into four square quadrants of order $n/2$ and we have

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \Rightarrow {}^tA = \begin{pmatrix} {}^tA_{1,1} & {}^tA_{2,1} \\ {}^tA_{1,2} & {}^tA_{2,2} \end{pmatrix}.$$

- This observation yields an "in-place" algorithm:
  1. If $n = 1$ then return A.
  2. If $n > 1$ then

Matrix transposition: a first divide-and-conquer (1/4)

- For simplicity, assume that our input matrix A is square of order n and that n is a power of 2, say $n = 2^k$.
- We divide A into four square quadrants of order $n/2$ and we have

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \Rightarrow {}^tA = \begin{pmatrix} {}^tA_{1,1} & {}^tA_{2,1} \\ {}^tA_{1,2} & {}^tA_{2,2} \end{pmatrix}.$$

- This observation yields an "in-place" algorithm:
  1. If $n = 1$ then return A.
  2. If $n > 1$ then
     1. recursively compute ${}^tA_{1,1}, {}^tA_{2,1}, {}^tA_{1,2}, {}^tA_{2,2}$ in place as

$$\begin{pmatrix} {}^tA_{1,1} & {}^tA_{1,2} \\ {}^tA_{2,1} & {}^tA_{2,2} \end{pmatrix}$$

Matrix transposition: a first divide-and-conquer (1/4)

- For simplicity, assume that our input matrix A is square of order n and that n is a power of 2, say $n = 2^k$.
- We divide A into four square quadrants of order $n/2$ and we have

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \Rightarrow {}^tA = \begin{pmatrix} {}^tA_{1,1} & {}^tA_{2,1} \\ {}^tA_{1,2} & {}^tA_{2,2} \end{pmatrix}.$$

- This observation yields an "in-place" algorithm:
  1. If $n = 1$ then return A.
  2. If $n > 1$ then
     1. recursively compute ${}^tA_{1,1}, {}^tA_{2,1}, {}^tA_{1,2}, {}^tA_{2,2}$ in place as
     $$\begin{pmatrix} {}^tA_{1,1} & {}^tA_{1,2} \\ {}^tA_{2,1} & {}^tA_{2,2} \end{pmatrix}$$
     2. exchange ${}^tA_{1,2}$ and ${}^tA_{2,1}$.

Matrix transposition: a first divide-and-conquer (1/4)

- For simplicity, assume that our input matrix A is square of order n and that n is a power of 2, say $n = 2^k$.
- We divide A into four square quadrants of order $n/2$ and we have

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \Rightarrow {}^tA = \begin{pmatrix} {}^tA_{1,1} & {}^tA_{2,1} \\ {}^tA_{1,2} & {}^tA_{2,2} \end{pmatrix}.$$

- This observation yields an "in-place" algorithm:
  1. If $n = 1$ then return A.
  2. If $n > 1$ then
     1. recursively compute ${}^tA_{1,1}, {}^tA_{2,1}, {}^tA_{1,2}, {}^tA_{2,2}$ in place as
     $$\begin{pmatrix} {}^tA_{1,1} & {}^tA_{1,2} \\ {}^tA_{2,1} & {}^tA_{2,2} \end{pmatrix}$$
     2. exchange ${}^tA_{1,2}$ and ${}^tA_{2,1}$.
- What is the number $M(n)$ of memory accesses to A, performed by this algorithm on an input matrix A of order n?

Matrix transposition: a first divide-and-conquer (2/4)

- $M(n)$ satisfies the following recurrence relation

$$M(n) = \begin{cases} 0 & \text{if} \quad n = 1 \\ 4M(n/2) + 2(n/2)^2 & \text{if} \quad n > 1. \end{cases}$$

Matrix transposition: a first divide-and-conquer (2/4)

- $M(n)$ satisfies the following recurrence relation

$$M(n) = \begin{cases} 0 & \text{if} \quad n = 1 \\ 4M(n/2) + 2(n/2)^2 & \text{if} \quad n > 1. \end{cases}$$

- Unfolding the tree of recursive calls or using the *Master's Theorem*, one obtains:

$$M(n) = 2(n/2)^2 \ \log_2(n).$$

Matrix transposition: a first divide-and-conquer (2/4)

- $M(n)$ satisfies the following recurrence relation

$$M(n) = \begin{cases} 0 & \text{if} \quad n = 1 \\ 4M(n/2) + 2(n/2)^2 & \text{if} \quad n > 1. \end{cases}$$

- Unfolding the tree of recursive calls or using the *Master's Theorem*, one obtains:

$$M(n) = 2(n/2)^2 \ \log_2(n).$$

- This is worse than the straightforward algorithm (which employs doubly nested loops). Indeed, for this latter, we have $M(n) = n^2 - n$. Explain why!

Matrix transposition: a first divide-and-conquer (2/4)

- $M(n)$ satisfies the following recurrence relation

$$M(n) = \begin{cases} 0 & \text{if} \quad n = 1 \\ 4M(n/2) + 2(n/2)^2 & \text{if} \quad n > 1. \end{cases}$$

- Unfolding the tree of recursive calls or using the *Master's Theorem*, one obtains:

$$M(n) = 2(n/2)^2 \ \log_2(n).$$

- This is worse than the straightforward algorithm (which employs doubly nested loops). Indeed, for this latter, we have $M(n) = n^2 - n$. Explain why!
- Despite of this negative result, we shall analyze the cache complexity of this first divide-and-conquer algorithm. Indeed, it provides us with an easy training exercise

Matrix transposition: a first divide-and-conquer (2/4)

- $M(n)$ satisfies the following recurrence relation

$$M(n) = \begin{cases} 0 & \text{if} \quad n = 1 \\ 4M(n/2) + 2(n/2)^2 & \text{if} \quad n > 1. \end{cases}$$

- Unfolding the tree of recursive calls or using the *Master's Theorem*, one obtains:

$$M(n) = 2(n/2)^2 \ \log_2(n).$$

- This is worse than the straightforward algorithm (which employs doubly nested loops). Indeed, for this latter, we have $M(n) = n^2 - n$. Explain why!
- Despite of this negative result, we shall analyze the cache complexity of this first divide-and-conquer algorithm. Indeed, it provides us with an easy training exercise
- We shall study later a second and efficiency-optimal divide-and-conquer algorithm, whose cache complexity analysis is more involved.

Matrix transposition: a first divide-and-conquer (3/4)

- We shall determine $Q(n)$ the number of cache misses incurred by our first divide-and-conquer algorithm on a (Z, L)-ideal cache machine.

Matrix transposition: a first divide-and-conquer (3/4)

- We shall determine $Q(n)$ the number of cache misses incurred by our first divide-and-conquer algorithm on a (Z, L)-ideal cache machine.
- For n small enough, the entire input matrix or the entire block (input of some recursive call) fits in cache and incurs only the cost of a scanning. Because of possible misalignment, that is, $n(\lceil n/L \rceil + 1)$.

Matrix transposition: a first divide-and-conquer (3/4)

- We shall determine $Q(n)$ the number of cache misses incurred by our first divide-and-conquer algorithm on a (Z, L)-ideal cache machine.
- For n small enough, the entire input matrix or the entire block (input of some recursive call) fits in cache and incurs only the cost of a scanning. Because of possible misalignment, that is, $n(\lceil n/L \rceil + 1)$.
- **Important:** For simplicity, some authors write n/L instead of $\lceil n/L \rceil$. This can be dangerous.

Matrix transposition: a first divide-and-conquer (3/4)

- We shall determine $Q(n)$ the number of cache misses incurred by our first divide-and-conquer algorithm on a (Z, L)-ideal cache machine.
- For n small enough, the entire input matrix or the entire block (input of some recursive call) fits in cache and incurs only the cost of a scanning. Because of possible misalignment, that is, $n(\lceil n/L \rceil + 1)$.
- **Important:** For simplicity, some authors write n/L instead of $\lceil n/L \rceil$. This can be dangerous.
- **However:** these simplifications are fine for asymptotic estimates, keeping in mind that n/L is a rational number satisfying

$$n/L - 1 \leq \lfloor n/L \rfloor \leq n/L \leq \lceil n/L \rceil \leq n/L + 1.$$

Thus, for a fixed L, the functions $\lfloor n/L \rfloor$, n/L and $\lceil n/L \rceil$ are asymptotically of the same order of magnitude.

Matrix transposition: a first divide-and-conquer (3/4)

- We shall determine $Q(n)$ the number of cache misses incurred by our first divide-and-conquer algorithm on a (Z, L)-ideal cache machine.
- For n small enough, the entire input matrix or the entire block (input of some recursive call) fits in cache and incurs only the cost of a scanning. Because of possible misalignment, that is, $n(\lceil n/L \rceil + 1)$.
- **Important:** For simplicity, some authors write n/L instead of $\lceil n/L \rceil$. This can be dangerous.
- **However:** these simplifications are fine for asymptotic estimates, keeping in mind that n/L is a rational number satisfying

$$n/L - 1 \leq \lfloor n/L \rfloor \leq n/L \leq \lceil n/L \rceil \leq n/L + 1.$$

  Thus, for a fixed L, the functions $\lfloor n/L \rfloor$, n/L and $\lceil n/L \rceil$ are asymptotically of the same order of magnitude.
- We need to translate "for n small enough" into a formula. We claim that there exists a real constant $\alpha > 0$ s.t. for all n and Z we have

$$n^2 < \alpha Z \quad \Rightarrow \quad Q(n) \leq n^2/L + n.$$

Matrix transposition: a first divide-and-conquer (4/4)

- $Q(n)$ satisfies the following recurrence relation

$$Q(n) = \begin{cases} n^2/L + n & \text{if} \quad n^2 < \alpha Z \quad \text{(base case)} \\ 4Q(n/2) + \frac{n^2}{2L} + n & \text{if} \quad n^2 \geq \alpha Z \quad \text{(recurrence)} \end{cases}$$

Matrix transposition: a first divide-and-conquer (4/4)

- $Q(n)$ satisfies the following recurrence relation

$$Q(n) = \begin{cases} n^2/L + n & \text{if} \quad n^2 < \alpha Z \quad (\text{base case}) \\ 4Q(n/2) + \frac{n^2}{2L} + n & \text{if} \quad n^2 \geq \alpha Z \quad (\text{recurrence}) \end{cases}$$

- Indeed, **exchanging 2 blocks** amount to $2((n/2)^2/L + n/2)$ accesses.

Matrix transposition: a first divide-and-conquer (4/4)

- $Q(n)$ satisfies the following recurrence relation

$$Q(n) = \begin{cases} n^2/L + n & \text{if} \quad n^2 < \alpha Z \quad \text{(base case)} \\ 4Q(n/2) + \frac{n^2}{2L} + n & \text{if} \quad n^2 \geq \alpha Z \quad \text{(recurrence)} \end{cases}$$

- Indeed, **exchanging 2 blocks** amount to $2((n/2)^2/L + n/2)$ accesses.
- Unfolding the recurrence relation k times (using an induction) yields

$$Q(n) = 4^k\, Q(\frac{n}{2^k}) + k\,\frac{n^2}{2L} + (2^k - 1)n.$$

Matrix transposition: a first divide-and-conquer (4/4)

- $Q(n)$ satisfies the following recurrence relation

$$Q(n) = \begin{cases} n^2/L + n & \text{if} \quad n^2 < \alpha Z \quad \text{(base case)} \\ 4Q(n/2) + \frac{n^2}{2L} + n & \text{if} \quad n^2 \geq \alpha Z \quad \text{(recurrence)} \end{cases}$$

- Indeed, **exchanging 2 blocks** amount to $2((n/2)^2/L + n/2)$ accesses.
- Unfolding the recurrence relation k times (using an induction) yields

$$Q(n) = 4^k\, Q(\frac{n}{2^k}) + k\,\frac{n^2}{2L} + (2^k - 1)n.$$

- The minimum k for reaching the base case satisfies $\frac{n^2}{4^k} = \alpha Z$, that is, $4^k = \frac{n^2}{\alpha Z}$, that is, $k = \log_4(\frac{n^2}{\alpha Z})$. This implies $2^k = \frac{n}{\sqrt{\alpha Z}}$ and thus

$$\begin{aligned} Q(n) &\leq \frac{n^2}{\alpha Z}\,(\alpha Z/L + \sqrt{\alpha Z}) + \log_4(\frac{n^2}{\alpha Z})\,\frac{n^2}{2L} + \frac{n}{\sqrt{\alpha Z}}\,n \\ &\leq n^2/L + 2\frac{n^2}{\sqrt{\alpha Z}} + \log_4(\frac{n^2}{\alpha Z})\,\frac{n^2}{2L}. \end{aligned}$$

A matrix transposition cache-oblivious algorithm (1/2)

- If $n \geq m$, the Rec-Transpose algorithm partitions

$$A = (A_1 \ A_2) \ , \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

and recursively executes Rec-Transpose(A_1, B_1) and Rec-Transpose(A_2, B_2).

A matrix transposition cache-oblivious algorithm (1/2)

- If $n \geq m$, the Rec-Transpose algorithm partitions

$$A = (A_1 \ A_2) \ , \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

and recursively executes Rec-Transpose(A_1, B_1) and Rec-Transpose(A_2, B_2).

- If $m > n$, the Rec-Transpose algorithm partitions

$$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \ , \quad B = (B_1 \ B_2)$$

and recursively executes Rec-Transpose(A_1, B_1) and Rec-Transpose(A_2, B_2).

A matrix transposition cache-oblivious algorithm (2/2)

- Recall that the matrices are stored in row-major layout.

A matrix transposition cache-oblivious algorithm (2/2)

- Recall that the matrices are stored in row-major layout.
- Let α be a constant sufficiently small such that the following two conditions hold:

A matrix transposition cache-oblivious algorithm (2/2)

- Recall that the matrices are stored in row-major layout.
- Let α be a constant sufficiently small such that the following two conditions hold:
  - (i) two sub-matrices of size $m \times n$ and $n \times m$, where $\max\{m, n\} \leq \alpha L$, fit in cache

A matrix transposition cache-oblivious algorithm (2/2)

- Recall that the matrices are stored in row-major layout.
- Let α be a constant sufficiently small such that the following two conditions hold:
  - (i) two sub-matrices of size $m \times n$ and $n \times m$, where $\max\{m, n\} \leq \alpha L$, fit in cache
  - (ii) even if each row starts at a different cache line.

A matrix transposition cache-oblivious algorithm (2/2)

- Recall that the matrices are stored in row-major layout.
- Let α be a constant sufficiently small such that the following two conditions hold:
  - (i) two sub-matrices of size $m \times n$ and $n \times m$, where $\max\{m, n\} \leq \alpha L$, fit in cache
  - (ii) even if each row starts at a different cache line.
- We distinguish three cases for the input matrix A:

A matrix transposition cache-oblivious algorithm (2/2)

- Recall that the matrices are stored in row-major layout.
- Let α be a constant sufficiently small such that the following two conditions hold:
  - (i) two sub-matrices of size $m \times n$ and $n \times m$, where $\max\{m, n\} \leq \alpha L$, fit in cache
  - (ii) even if each row starts at a different cache line.
- We distinguish three cases for the input matrix A:
  - ↳ Case I: $\max\{m, n\} \leq \alpha L$.

A matrix transposition cache-oblivious algorithm (2/2)

- Recall that the matrices are stored in row-major layout.
- Let α be a constant sufficiently small such that the following two conditions hold:
  - (i) two sub-matrices of size $m \times n$ and $n \times m$, where $\max\{m, n\} \leq \alpha L$, fit in cache
  - (ii) even if each row starts at a different cache line.
- We distinguish three cases for the input matrix A:
  - ↳ Case I: $\max\{m, n\} \leq \alpha L$.
  - ↳ Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.

A matrix transposition cache-oblivious algorithm (2/2)

- Recall that the matrices are stored in row-major layout.
- Let α be a constant sufficiently small such that the following two conditions hold:
  - (i) two sub-matrices of size $m \times n$ and $n \times m$, where $\max\{m, n\} \leq \alpha L$, fit in cache
  - (ii) even if each row starts at a different cache line.
- We distinguish three cases for the input matrix A:
  - ↳ Case I: $\max\{m, n\} \leq \alpha L$.
  - ↳ Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.
  - ↳ Case III: $m, n > \alpha L$.

Case I: $\max\{m, n\} \leq \alpha L$.

- Both matrices fit in $O(1) + 2mn/L$ lines.

Case I: $\max\{m, n\} \leq \alpha L$.

- Both matrices fit in $O(1) + 2mn/L$ lines.
- From the choice of α, the number of lines required for the entire computation is at most Z/L.

Case I: $\max\{m, n\} \leq \alpha L$.

- Both matrices fit in $O(1) + 2mn/L$ lines.
- From the choice of α, the number of lines required for the entire computation is at most Z/L.
- Thus, no cache lines need to be evicted during the computation. Hence, it feels like we are simply scanning A and B.

Case I: $\max\{m, n\} \leq \alpha L$.

- Both matrices fit in $O(1) + 2mn/L$ lines.
- From the choice of α, the number of lines required for the entire computation is at most Z/L.
- Thus, no cache lines need to be evicted during the computation. Hence, it feels like we are simply scanning A and B.
- Therefore $Q(m, n) \in O(1 + mn/L)$.

Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.

- Consider $n \leq \alpha L < m$. The REC-TRANSPOSE algorithm divides the greater dimension m by 2 and recurses.

Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.

- Consider $n \leq \alpha L < m$. The REC-TRANSPOSE algorithm divides the greater dimension m by 2 and recurses.
- At some point in the recursion, we have $\alpha L/2 \leq m \leq \alpha L$ and the whole computation fits in cache. At this point:

Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.

- Consider $n \leq \alpha L < m$. The REC-TRANSPOSE algorithm divides the greater dimension m by 2 and recurses.
- At some point in the recursion, we have $\alpha L/2 \leq m \leq \alpha L$ and the whole computation fits in cache. At this point:
  - ↳ the input array resides in contiguous locations, requiring at most $\Theta(1 + nm/L)$ cache misses

Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.

- Consider $n \leq \alpha L < m$. The Rec-Transpose algorithm divides the greater dimension m by 2 and recurses.
- At some point in the recursion, we have $\alpha L/2 \leq m \leq \alpha L$ and the whole computation fits in cache. At this point:
  - ↳ the input array resides in contiguous locations, requiring at most $\Theta(1 + nm/L)$ cache misses
  - ↳ the output array consists of nm elements in n rows, where in the **worst case** every row starts at a different cache line, leading to at most $\Theta(n + nm/L)$ cache misses.

Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.

- Consider $n \leq \alpha L < m$. The REC-TRANSPOSE algorithm divides the greater dimension m by 2 and recurses.
- At some point in the recursion, we have $\alpha L/2 \leq m \leq \alpha L$ and the whole computation fits in cache. At this point:
  - ↳ the input array resides in contiguous locations, requiring at most $\Theta(1 + nm/L)$ cache misses
  - ↳ the output array consists of nm elements in n rows, where in the **worst case** every row starts at a different cache line, leading to at most $\Theta(n + nm/L)$ cache misses.
- Since $m/L \in [\alpha/2, \alpha]$, the **total** cache complexity for this base case is $\Theta(1 + n)$, yielding the recurrence (where the resulting $Q(m, n)$ is a **worst case estimate**)

$$Q(m,n) = \begin{cases} \Theta(1+n) & \text{if } m \in [\alpha L/2, \alpha L] \ , \\ 2Q(m/2,n) + O(1) & \text{otherwise ;} \end{cases}$$

whose solution satisfies $Q(m, n) = \Theta(1 + mn/L)$.

Case III: $m, n > \alpha L$.

- As in Case II, at some point in the recursion both n and m fall into the range $[\alpha L/2, \alpha L]$.

Case III: $m, n > \alpha L$.

- As in Case II, at some point in the recursion both n and m fall into the range $[\alpha L/2, \alpha L]$.
- The whole problem fits into cache and can be solved with at most $\Theta(m + n + mn/L)$ cache misses.

Case III: $m, n > \alpha L$.

- As in Case II, at some point in the recursion both n and m fall into the range $[\alpha L/2, \alpha L]$.
- The whole problem fits into cache and can be solved with at most $\Theta(m + n + mn/L)$ cache misses.
- The **worst case cache miss estimate** satisfies the recurrence

$$Q(m,n) = \begin{cases} \Theta(m+n+mn/L) & \text{if } m,n \in [\alpha L/2, \alpha L] \ , \\ 2Q(m/2,n) + O(1) & \text{if } m \geq n \ , \\ 2Q(m,n/2) + O(1) & \text{otherwise;} \end{cases}$$

whose solution is $Q(m,n) = \Theta(1 + mn/L)$.

Case III: $m, n > \alpha L$.

- As in Case II, at some point in the recursion both n and m fall into the range $[\alpha L/2, \alpha L]$.
- The whole problem fits into cache and can be solved with at most $\Theta(m + n + mn/L)$ cache misses.
- The **worst case cache miss estimate** satisfies the recurrence

$$Q(m,n) = \begin{cases} \Theta(m+n+mn/L) & \text{if } m,n \in [\alpha L/2, \alpha L] \ , \\ 2Q(m/2,n) + O(1) & \text{if } m \geq n \ , \\ 2Q(m,n/2) + O(1) & \text{otherwise;} \end{cases}$$

  whose solution is $Q(m,n) = \Theta(1 + mn/L)$.
- **Therefore, the Rec-Transpose algorithm has optimal cache complexity.**

Case III: $m, n > \alpha L$.

- As in Case II, at some point in the recursion both n and m fall into the range $[\alpha L/2, \alpha L]$.
- The whole problem fits into cache and can be solved with at most $\Theta(m + n + mn/L)$ cache misses.
- The **worst case cache miss estimate** satisfies the recurrence

$$Q(m,n) = \begin{cases} \Theta(m+n+mn/L) & \text{if } m,n \in [\alpha L/2, \alpha L] \ , \\ 2Q(m/2,n) + O(1) & \text{if } m \geq n \ , \\ 2Q(m,n/2) + O(1) & \text{otherwise;} \end{cases}$$

  whose solution is $Q(m,n) = \Theta(1 + mn/L)$.
- **Therefore, the Rec-Transpose algorithm has optimal cache complexity.**
- Indeed, for an $m \times n$ matrix, the algorithm must write to mn distinct elements, which occupy at least $\lceil mn/L \rceil$ cache lines.

Tuned cache-oblivious square matrix transposition

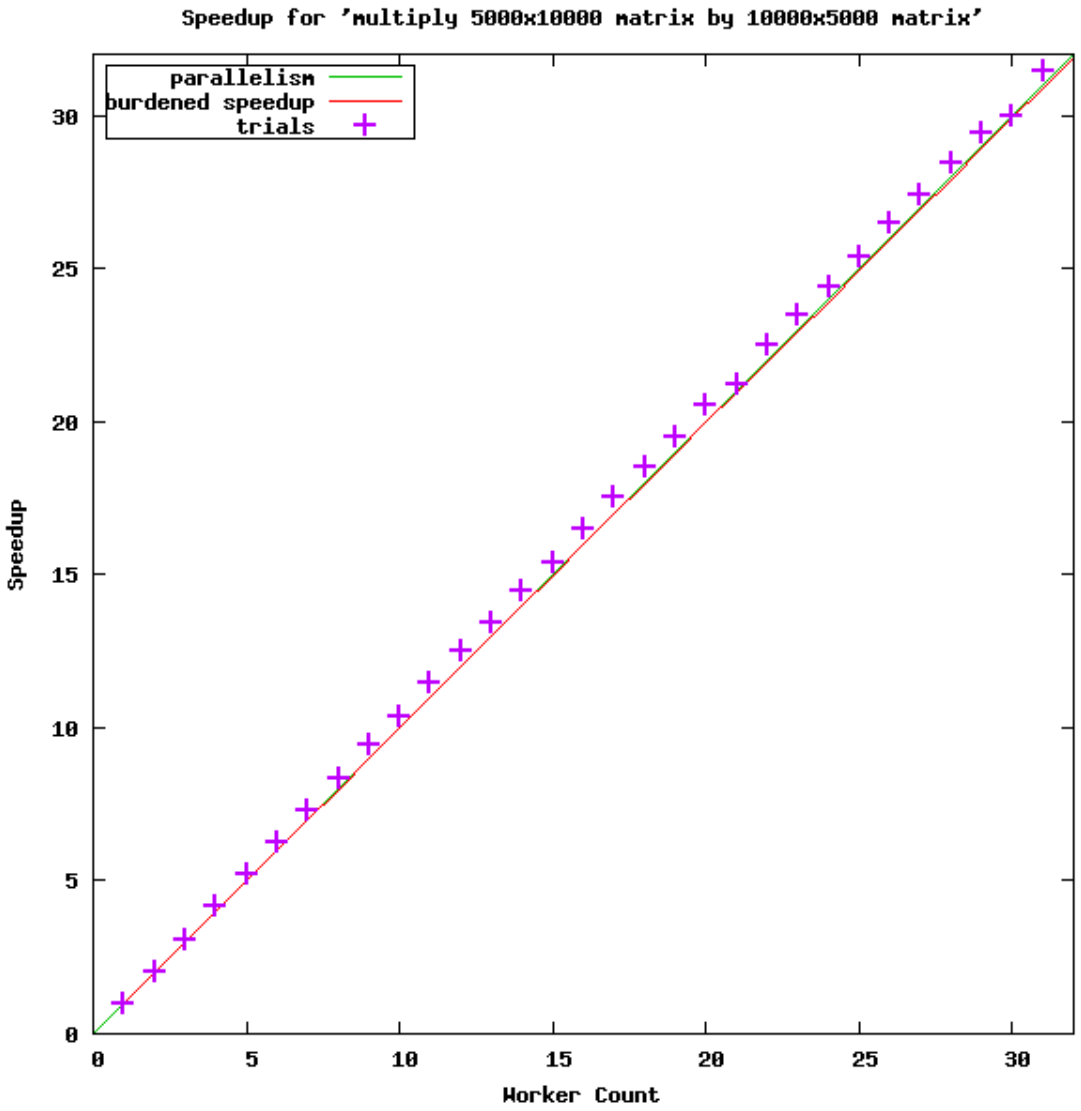
```
void DC_matrix_transpose(int *A, int lda, int i0, int i1,
    int j0, int dj0, int j1 /*, int dj1 = 0 */){
    const int THRESHOLD = 16; // tuned for the target machine
  tail:
    int di = i1 - i0, dj = j1 - j0;
    if (dj >= 2 * di && dj > THRESHOLD) {
        int dj2 = dj / 2;
        cilk_spawn DC_matrix_transpose(A, lda, i0, i1, j0, dj0, j0 + dj2);
        j0 += dj2; dj0 = 0; goto tail;
    } else if (di > THRESHOLD) {
        int di2 = di / 2;
        cilk_spawn DC_matrix_transpose(A, lda, i0, i0 + di2, j0, dj0, j1);
        i0 += di2; j0 += dj0 * di2; goto tail;
    } else {
        for (int i = i0; i < i1; ++i) {
            for (int j = j0; j < j1; ++j) {
                int x = A[j * lda + i];
                A[j * lda + i] = A[i * lda + j];
                A[i * lda + j] = x;
            }
            j0 += dj0;
        }
    }
}
```

Tuned cache-oblivious matrix transposition benchmarks

size	Naive	Cache-oblivious	ratio
5000x5000	126	79	1.59
10000x10000	627	311	2.02
20000x20000	4373	1244	3.52
30000x30000	23603	2734	8.63
40000x40000	62432	4963	12.58

- `Intel(R) Xeon(R) CPU E7340 @ 2.40GHz`
- L1 data 32 KB, L2 4096 KB, cache line size 64bytes
- **Both codes run on 1 core**
- The improvement comes simply from an **optimal memory access pattern.**

Tuned cache-oblivious matrix multiplication



Outline

Cache complexity of the naive matrix multiplication

```
// A is stored in ROW-major and B in COLUMN-major
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        for(k=0; k < n; k++)
            C[i][j] += A[i][k] * B[j][k];
```

- Recall the tall cache assumption, that is, $Z \in \Omega(L^2)$.

Cache complexity of the naive matrix multiplication

```
// A is stored in ROW-major and B in COLUMN-major
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        for(k=0; k < n; k++)
            C[i][j] += A[i][k] * B[j][k];
```

- Recall the tall cache assumption, that is, $Z \in \Omega(L^2)$.
- If the 3 matrices fit in cache, say $3n^2 \leq Z$ holds, then all cache misses are cold and we have $Q(n, Z, L) \in O(1 + n^2/L)$.

Cache complexity of the naive matrix multiplication

```
// A is stored in ROW-major and B in COLUMN-major
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        for(k=0; k < n; k++)
            C[i][j] += A[i][k] * B[j][k];
```

- Recall the tall cache assumption, that is, $Z \in \Omega(L^2)$.
- If the 3 matrices fit in cache, say $3n^2 \leq Z$ holds, then all cache misses are cold and we have $Q(n, Z, L) \in O(1 + n^2/L)$.
- If Z is large enough, precislely if $Z \in \Omega(n)$ holds, then Row i of A will be remembered for its entire involvement in computing row i of C.
- For Column j of B to be remembered when necessary, one needs $Z \in \Omega(n^2)$ in which case the whole computation fits in cache. Therefore, we have:

$$Q(n, Z, L) = \begin{cases} O(1 + n^2/L) & \text{if} \quad 3n^2 \leq Z, \\ O(n + n^3/L) & \text{if} \quad Z < n^2. \end{cases}$$

A cache-aware matrix multiplication algorithm (1/2)

```
// A, B and C are in row-major storage
for(i =0; i < n/s; i++)
    for(j =0; j < n/s; j++)
        for(k=0; k < n/s; k++)
            blockMult(A,B,C,i,j,k,s);
```

- Each matrix $M \in \{A, B, C\}$ consists of $(n/s) \times (n/s)$ submatrices M_{ij} (the blocks), each of which has size $s \times s$, where s is a tuning parameter.

A cache-aware matrix multiplication algorithm (1/2)

```
// A, B and C are in row-major storage
for(i =0; i < n/s; i++)
    for(j =0; j < n/s; j++)
        for(k=0; k < n/s; k++)
            blockMult(A,B,C,i,j,k,s);
```

- Each matrix $M \in \{A, B, C\}$ consists of $(n/s) \times (n/s)$ submatrices M_{ij} (the blocks), each of which has size $s \times s$, where s is a tuning parameter.
- Assume s divides n to keep the analysis simple.

A cache-aware matrix multiplication algorithm (1/2)

```
// A, B and C are in row-major storage
for(i =0; i < n/s; i++)
    for(j =0; j < n/s; j++)
        for(k=0; k < n/s; k++)
            blockMult(A,B,C,i,j,k,s);
```

- Each matrix $M \in \{A, B, C\}$ consists of $(n/s) \times (n/s)$ submatrices M_{ij} (the blocks), each of which has size $s \times s$, where s is a tuning parameter.
- Assume s divides n to keep the analysis simple.
- `blockMult(A,B,C,i,j,k,s)` computes $C_{ij} = A_{ik} \times B_{kj}$ using the naive algorithm

A cache-aware matrix multiplication algorithm (2/2)

```
// A, B and C are in row-major storage
for(i =0; i < n/s; i++)
    for(j =0; j < n/s; j++)
        for(k=0; k < n/s; k++)
            blockMult(A,B,C,i,j,k,s);
```

- Choose s to be the largest value such that three $s \times s$ submatrices simultaneously fit in cache, that is, $Z \in \Theta(s^2)$, that is, $s \in \Theta(\sqrt{Z})$.

A cache-aware matrix multiplication algorithm (2/2)

```
// A, B and C are in row-major storage
for(i =0; i < n/s; i++)
    for(j =0; j < n/s; j++)
        for(k=0; k < n/s; k++)
            blockMult(A,B,C,i,j,k,s);
```

- Choose s to be the largest value such that three $s \times s$ submatrices simultaneously fit in cache, that is, $Z \in \Theta(s^2)$, that is, $s \in \Theta(\sqrt{Z})$.
- An $s \times s$ submatrix is stored on $\Theta(s + s^2/L)$ cache lines.

A cache-aware matrix multiplication algorithm (2/2)

```
// A, B and C are in row-major storage
for(i =0; i < n/s; i++)
    for(j =0; j < n/s; j++)
        for(k=0; k < n/s; k++)
            blockMult(A,B,C,i,j,k,s);
```

- Choose s to be the largest value such that three $s \times s$ submatrices simultaneously fit in cache, that is, $Z \in \Theta(s^2)$, that is, $s \in \Theta(\sqrt{Z})$.
- An $s \times s$ submatrix is stored on $\Theta(s + s^2/L)$ cache lines.
- Thus `blockMult(A,B,C,i,j,k,s)` runs within $\Theta(s + s^2/L)$ cache misses.

A cache-aware matrix multiplication algorithm (2/2)

```
// A, B and C are in row-major storage
for(i =0; i < n/s; i++)
    for(j =0; j < n/s; j++)
        for(k=0; k < n/s; k++)
            blockMult(A,B,C,i,j,k,s);
```

- Choose s to be the largest value such that three $s \times s$ submatrices simultaneously fit in cache, that is, $Z \in \Theta(s^2)$, that is, $s \in \Theta(\sqrt{Z})$.
- An $s \times s$ submatrix is stored on $\Theta(s + s^2/L)$ cache lines.
- Thus `blockMult(A,B,C,i,j,k,s)` runs within $\Theta(s + s^2/L)$ cache misses.
- Initializing the n^2 elements of C amounts to $\Theta(1 + n^2/L)$ caches misses. Therefore we have

$$\begin{aligned} Q(n, Z, L) &\in \Theta(1 + n^2/L + (n/\sqrt{Z})^3(\sqrt{Z} + Z/L)) \\ &\in \Theta(1 + n^2/L + n^3/Z + n^3/(L\sqrt{Z})). \end{aligned}$$

A cache-oblivious matrix multiplication algorithm (1/3)

- We describe and analyze a cache-oblivious algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix cache-obliviously using

A cache-oblivious matrix multiplication algorithm (1/3)

- We describe and analyze a cache-oblivious algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix cache-obliviously using
  - $\Theta(mnp)$ **work** and incurring

A cache-oblivious matrix multiplication algorithm (1/3)

- We describe and analyze a cache-oblivious algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix cache-obliviously using
  - $\Theta(mnp)$ **work** and incurring
  - $\Theta(m + n + p + (mn + np + mp)/L + mnp/(L\sqrt{Z}))$ **cache misses.**

A cache-oblivious matrix multiplication algorithm (1/3)

- We describe and analyze a cache-oblivious algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix cache-obliviously using
  - $\Theta(mnp)$ **work** and incurring
  - $\Theta(m + n + p + (mn + np + mp)/L + mnp/(L\sqrt{Z}))$ **cache misses.**
- This straightforward divide-and-conquer algorithm contains **no voodoo parameters** (tuning parameters) and it uses cache optimally.

A cache-oblivious matrix multiplication algorithm (1/3)

- We describe and analyze a cache-oblivious algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix cache-obliviously using
  - $\Theta(mnp)$ **work** and incurring
  - $\Theta(m + n + p + (mn + np + mp)/L + mnp/(L\sqrt{Z}))$ **cache misses.**
- This straightforward divide-and-conquer algorithm contains **no voodoo parameters** (tuning parameters) and it uses cache optimally.
- Intuitively, this algorithm uses the cache effectively, because once a subproblem fits into the cache, its smaller subproblems can be solved in cache with no further cache misses.

A cache-oblivious matrix multiplication algorithm (1/3)

- We describe and analyze a cache-oblivious algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix cache-obliviously using
  - $\Theta(mnp)$ **work** and incurring
  - $\Theta(m + n + p + (mn + np + mp)/L + mnp/(L\sqrt{Z}))$ **cache misses.**
- This straightforward divide-and-conquer algorithm contains **no voodoo parameters** (tuning parameters) and it uses cache optimally.
- Intuitively, this algorithm uses the cache effectively, because once a subproblem fits into the cache, its smaller subproblems can be solved in cache with no further cache misses.
- These results require the tall-cache assumption for matrices stored in row-major layout format,

A cache-oblivious matrix multiplication algorithm (1/3)

- We describe and analyze a cache-oblivious algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix cache-obliviously using
  - $\Theta(mnp)$ **work** and incurring
  - $\Theta(m + n + p + (mn + np + mp)/L + mnp/(L\sqrt{Z}))$ **cache misses.**
- This straightforward divide-and-conquer algorithm contains **no voodoo parameters** (tuning parameters) and it uses cache optimally.
- Intuitively, this algorithm uses the cache effectively, because once a subproblem fits into the cache, its smaller subproblems can be solved in cache with no further cache misses.
- These results require the tall-cache assumption for matrices stored in row-major layout format,
- This assumption can be relaxed for certain other layouts, see (Frigo et al. 1999).

A cache-oblivious matrix multiplication algorithm (1/3)

- We describe and analyze a cache-oblivious algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix cache-obliviously using
  - $\Theta(mnp)$ **work** and incurring
  - $\Theta(m + n + p + (mn + np + mp)/L + mnp/(L\sqrt{Z}))$ **cache misses.**
- This straightforward divide-and-conquer algorithm contains **no voodoo parameters** (tuning parameters) and it uses cache optimally.
- Intuitively, this algorithm uses the cache effectively, because once a subproblem fits into the cache, its smaller subproblems can be solved in cache with no further cache misses.
- These results require the tall-cache assumption for matrices stored in row-major layout format,
- This assumption can be relaxed for certain other layouts, see (Frigo et al. 1999).
- The case of Strassen's algorithm is also treated in (Frigo et al. 1999).

A cache-oblivious matrix multiplication algorithm (2/3)

- To multiply an $m \times n$ matrix A and an $n \times p$ matrix B, the REC-MULT algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \tag{1}$$

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \tag{2}$$

$$A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} . \tag{3}$$

A cache-oblivious matrix multiplication algorithm (2/3)

- To multiply an $m \times n$ matrix A and an $n \times p$ matrix B, the REC-MULT algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \tag{1}$$

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \tag{2}$$

$$A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} . \tag{3}$$

- In case (1), we have $m \geq \max\{n, p\}$. Matrix A is split horizontally, and both halves are multiplied by matrix B.

A cache-oblivious matrix multiplication algorithm (2/3)

- To multiply an $m \times n$ matrix A and an $n \times p$ matrix B, the REC-MULT algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \tag{1}$$

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \tag{2}$$

$$A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} . \tag{3}$$

- In case (1), we have $m \geq \max\{n, p\}$. Matrix A is split horizontally, and both halves are multiplied by matrix B.
- In case (2), we have $n \geq \max\{m, p\}$. Both matrices are split, and the two halves are multiplied.

A cache-oblivious matrix multiplication algorithm (2/3)

- To multiply an $m \times n$ matrix A and an $n \times p$ matrix B, the REC-MULT algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \tag{1}$$

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \tag{2}$$

$$A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} . \tag{3}$$

- In case (1), we have $m \geq \max\{n, p\}$. Matrix A is split horizontally, and both halves are multiplied by matrix B.
- In case (2), we have $n \geq \max\{m, p\}$. Both matrices are split, and the two halves are multiplied.
- In case (3), we have $p \geq \max\{m, n\}$. Matrix B is split vertically, and each half is multiplied by A.

A cache-oblivious matrix multiplication algorithm (2/3)

- To multiply an $m \times n$ matrix A and an $n \times p$ matrix B, the REC-MULT algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \tag{1}$$

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \tag{2}$$

$$A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} . \tag{3}$$

- In case (1), we have $m \geq \max\{n, p\}$. Matrix A is split horizontally, and both halves are multiplied by matrix B.
- In case (2), we have $n \geq \max\{m, p\}$. Both matrices are split, and the two halves are multiplied.
- In case (3), we have $p \geq \max\{m, n\}$. Matrix B is split vertically, and each half is multiplied by A.
- The base case occurs when $m = n = p = 1$.

A cache-oblivious matrix multiplication algorithm (3/3)

- let $\alpha > 0$ be the largest constant sufficiently small that three submatrices of sizes $m' \times n'$, $n' \times p'$, and $m' \times p'$ all fit completely in the cache, whenever $\max\{m', n', p'\} \leq \alpha\sqrt{Z}$ holds.

A cache-oblivious matrix multiplication algorithm (3/3)

- let $\alpha > 0$ be the largest constant sufficiently small that three submatrices of sizes $m' \times n'$, $n' \times p'$, and $m' \times p'$ all fit completely in the cache, whenever $\max\{m', n', p'\} \leq \alpha\sqrt{Z}$ holds.

- We distinguish four cases depending on the initial size of the matrices.

A cache-oblivious matrix multiplication algorithm (3/3)

- let $\alpha > 0$ be the largest constant sufficiently small that three submatrices of sizes $m' \times n'$, $n' \times p'$, and $m' \times p'$ all fit completely in the cache, whenever $\max\{m', n', p'\} \leq \alpha\sqrt{Z}$ holds.

- We distinguish four cases depending on the initial size of the matrices.
  - ↳ Case I: $m, n, p > \alpha\sqrt{Z}$.

A cache-oblivious matrix multiplication algorithm (3/3)

- let $\alpha > 0$ be the largest constant sufficiently small that three submatrices of sizes $m' \times n'$, $n' \times p'$, and $m' \times p'$ all fit completely in the cache, whenever $\max\{m', n', p'\} \leq \alpha\sqrt{Z}$ holds.

- We distinguish four cases depending on the initial size of the matrices.
  - Case I: $m, n, p > \alpha\sqrt{Z}$.
  - Case II: $(m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z})$ or $(n \leq \alpha\sqrt{Z}$ and $m, p > \alpha\sqrt{Z})$ or $(p \leq \alpha\sqrt{Z}$ and $m, n > \alpha\sqrt{Z})$.

A cache-oblivious matrix multiplication algorithm (3/3)

- let $\alpha > 0$ be the largest constant sufficiently small that three submatrices of sizes $m' \times n'$, $n' \times p'$, and $m' \times p'$ all fit completely in the cache, whenever $\max\{m', n', p'\} \leq \alpha\sqrt{Z}$ holds.

- We distinguish four cases depending on the initial size of the matrices.
  - Case I: $m, n, p > \alpha\sqrt{Z}$.
  - Case II: $(m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z})$ or $(n \leq \alpha\sqrt{Z}$ and $m, p > \alpha\sqrt{Z})$ or $(p \leq \alpha\sqrt{Z}$ and $m, n > \alpha\sqrt{Z})$.
  - Case III: $(n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z})$ or $(m, p \leq \alpha\sqrt{Z}$ and $n > \alpha\sqrt{Z})$ or $(m, n \leq \alpha\sqrt{Z}$ and $p > \alpha\sqrt{Z})$.

A cache-oblivious matrix multiplication algorithm (3/3)

- let $\alpha > 0$ be the largest constant sufficiently small that three submatrices of sizes $m' \times n'$, $n' \times p'$, and $m' \times p'$ all fit completely in the cache, whenever $\max\{m', n', p'\} \leq \alpha\sqrt{Z}$ holds.

- We distinguish four cases depending on the initial size of the matrices.
  - Case I: $m, n, p > \alpha\sqrt{Z}$.
  - Case II: $(m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z})$ or $(n \leq \alpha\sqrt{Z}$ and $m, p > \alpha\sqrt{Z})$ or $(p \leq \alpha\sqrt{Z}$ and $m, n > \alpha\sqrt{Z})$.
  - Case III: $(n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z})$ or $(m, p \leq \alpha\sqrt{Z}$ and $n > \alpha\sqrt{Z})$ or $(m, n \leq \alpha\sqrt{Z}$ and $p > \alpha\sqrt{Z})$.
  - Case IV: $m, n, p \leq \alpha\sqrt{Z}$.

A cache-oblivious matrix multiplication algorithm (3/3)

- let $\alpha > 0$ be the largest constant sufficiently small that three submatrices of sizes $m' \times n'$, $n' \times p'$, and $m' \times p'$ all fit completely in the cache, whenever $\max\{m', n', p'\} \leq \alpha\sqrt{Z}$ holds.

- We distinguish four cases depending on the initial size of the matrices.
  - Case I: $m, n, p > \alpha\sqrt{Z}$.
  - Case II: $(m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z})$ or $(n \leq \alpha\sqrt{Z}$ and $m, p > \alpha\sqrt{Z})$ or $(p \leq \alpha\sqrt{Z}$ and $m, n > \alpha\sqrt{Z})$.
  - Case III: $(n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z})$ or $(m, p \leq \alpha\sqrt{Z}$ and $n > \alpha\sqrt{Z})$ or $(m, n \leq \alpha\sqrt{Z}$ and $p > \alpha\sqrt{Z})$.
  - Case IV: $m, n, p \leq \alpha\sqrt{Z}$.
- Similarly to matrix transposition, $Q(m, n, p)$ is a **worst case cache miss estimate**.

Case I: $m, n, p > \alpha\sqrt{Z}$. (1/2)

$$Q(m,n,p) = \tag{4}$$
$$\begin{cases} \Theta((mn+np+mp)/L) & \text{if } m,n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m/2,n,p) + O(1) & \text{ow. if } m \geq n \text{ and } m \geq p \ , \\ 2Q(m,n/2,p) + O(1) & \text{ow. if } n > m \text{ and } n \geq p \ , \\ 2Q(m,n,p/2) + O(1) & \text{otherwise .} \end{cases}$$

- The base case arises as soon as all three submatrices fit in cache:

Case I: $m, n, p > \alpha\sqrt{Z}$. (1/2)

$$Q(m,n,p) = \tag{4}$$
$$\begin{cases} \Theta((mn+np+mp)/L) & \text{if } m,n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m/2,n,p)+O(1) & \text{ow. if } m \geq n \text{ and } m \geq p \ , \\ 2Q(m,n/2,p)+O(1) & \text{ow. if } n > m \text{ and } n \geq p \ , \\ 2Q(m,n,p/2)+O(1) & \text{otherwise .} \end{cases}$$

- The base case arises as soon as all three submatrices fit in cache:
  - ↳ The total number of cache lines used by the three submatrices is $\Theta((mn+np+mp)/L)$.

Case I: $m, n, p > \alpha\sqrt{Z}$. (1/2)

$$Q(m,n,p) = \tag{4}$$
$$\begin{cases} \Theta((mn+np+mp)/L) & \text{if } m,n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m/2,n,p)+O(1) & \text{ow. if } m \geq n \text{ and } m \geq p \ , \\ 2Q(m,n/2,p)+O(1) & \text{ow. if } n > m \text{ and } n \geq p \ , \\ 2Q(m,n,p/2)+O(1) & \text{otherwise .} \end{cases}$$

- The base case arises as soon as all three submatrices fit in cache:
  - The total number of cache lines used by the three submatrices is $\Theta((mn+np+mp)/L)$.
  - The only cache misses that occur during the remainder of the recursion are the $\Theta((mn+np+mp)/L)$ cache misses required to bring the matrices into cache.

Case I: $m, n, p > \alpha\sqrt{Z}$. (2/2)

$$Q(m,n,p) = \begin{cases} \Theta((mn+np+mp)/L) & \text{if } m,n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \text{ ,} \\ 2Q(m/2,n,p) + O(1) & \text{ow. if } m \geq n \text{ and } m \geq p \text{ ,} \\ 2Q(m,n/2,p) + O(1) & \text{ow. if } n > m \text{ and } n \geq p \text{ ,} \\ 2Q(m,n,p/2) + O(1) & \text{otherwise .} \end{cases}$$

- In the recursive cases, when the matrices do not fit in cache, we pay for the cache misses of the recursive calls, plus $O(1)$ cache misses for the overhead of manipulating submatrices.

Case I: $m, n, p > \alpha\sqrt{Z}$. (2/2)

$$Q(m,n,p) = \begin{cases} \Theta((mn+np+mp)/L) & \text{if } m,n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m/2,n,p) + O(1) & \text{ow. if } m \geq n \text{ and } m \geq p \ , \\ 2Q(m,n/2,p) + O(1) & \text{ow. if } n > m \text{ and } n \geq p \ , \\ 2Q(m,n,p/2) + O(1) & \text{otherwise .} \end{cases}$$

- In the recursive cases, when the matrices do not fit in cache, we pay for the cache misses of the recursive calls, plus $O(1)$ cache misses for the overhead of manipulating submatrices.
- The solution to this recurrence is

$$Q(m,n,p) = \Theta(mnp/(L\sqrt{Z})).$$

Case I: $m, n, p > \alpha\sqrt{Z}$. (2/2)

$$Q(m,n,p) = \begin{cases} \Theta((mn+np+mp)/L) & \text{if } m,n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m/2,n,p) + O(1) & \text{ow. if } m \geq n \text{ and } m \geq p \ , \\ 2Q(m,n/2,p) + O(1) & \text{ow. if } n > m \text{ and } n \geq p \ , \\ 2Q(m,n,p/2) + O(1) & \text{otherwise .} \end{cases}$$

- In the recursive cases, when the matrices do not fit in cache, we pay for the cache misses of the recursive calls, plus $O(1)$ cache misses for the overhead of manipulating submatrices.
- The solution to this recurrence is

$$Q(m,n,p) = \Theta(mnp/(L\sqrt{Z})).$$

- Indeed, for the base-case $m, m, p \in \Theta(\alpha\sqrt{Z})$.

Case II: $(m \leq \alpha\sqrt{Z})$ and $(n, p > \alpha\sqrt{Z})$.

- Here, we shall present the case where $m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$.

Case II: $(m \leq \alpha\sqrt{Z})$ and $(n, p > \alpha\sqrt{Z})$.

- Here, we shall present the case where $m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$.
- The REC-MULT algorithm always divides n or p by 2 according to cases (2) and (3).

Case II: $(m \leq \alpha\sqrt{Z})$ and $(n, p > \alpha\sqrt{Z})$.

- Here, we shall present the case where $m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$.
- The REC-MULT algorithm always divides n or p by 2 according to cases (2) and (3).
- At some point in the recursion, both n and p are small enough that the whole problem fits into cache.

Case II: $(m \leq \alpha\sqrt{Z})$ and $(n, p > \alpha\sqrt{Z})$.

- Here, we shall present the case where $m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$.
- The REC-MULT algorithm always divides n or p by 2 according to cases (2) and (3).
- At some point in the recursion, both n and p are small enough that the whole problem fits into cache.
- The number of cache misses can be described by the recurrence

$$Q(m,n,p) = \begin{cases} \Theta(1+n+m+np/L) & \text{if } n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m,n/2,p)+O(1) & \text{otherwise if } n \geq p \ , \\ 2Q(m,n,p/2)+O(1) & \text{otherwise ;} \end{cases} \tag{5}$$

whose solution is $Q(m,n,p) = \Theta(np/L + mnp/(L\sqrt{Z}))$.

Case II: $(m \leq \alpha\sqrt{Z})$ and $(n, p > \alpha\sqrt{Z})$.

- Here, we shall present the case where $m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$.
- The REC-MULT algorithm always divides n or p by 2 according to cases (2) and (3).
- At some point in the recursion, both n and p are small enough that the whole problem fits into cache.
- The number of cache misses can be described by the recurrence

$$Q(m,n,p) = \begin{cases} \Theta(1+n+m+np/L) & \text{if } n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m,n/2,p)+O(1) & \text{otherwise if } n \geq p \ , \\ 2Q(m,n,p/2)+O(1) & \text{otherwise ;} \end{cases} \tag{5}$$

  whose solution is $Q(m,n,p) = \Theta(np/L + mnp/(L\sqrt{Z}))$.
- Indeed, in the base case: $mnp/(L\sqrt{Z}) \leq \alpha np/L$.

Case II: $(m \leq \alpha\sqrt{Z})$ and $(n, p > \alpha\sqrt{Z})$.

- Here, we shall present the case where $m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$.
- The REC-MULT algorithm always divides n or p by 2 according to cases (2) and (3).
- At some point in the recursion, both n and p are small enough that the whole problem fits into cache.
- The number of cache misses can be described by the recurrence

$$Q(m,n,p) = \begin{cases} \Theta(1+n+m+np/L) & \text{if } n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m,n/2,p)+O(1) & \text{otherwise if } n \geq p \ , \\ 2Q(m,n,p/2)+O(1) & \text{otherwise ;} \end{cases} \tag{5}$$

whose solution is $Q(m,n,p) = \Theta(np/L + mnp/(L\sqrt{Z}))$.

- Indeed, in the base case: $mnp/(L\sqrt{Z}) \leq \alpha np/L$.
- The term $\Theta(1+n+m)$ appears because of the row-major layout.

Case III: $(n, p \leq \alpha\sqrt{Z} \text{ and } m > \alpha\sqrt{Z})$

- In each of these cases, one of the matrices fits into cache, and the others do not.

Case III: $(n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z})$

- In each of these cases, one of the matrices fits into cache, and the others do not.
- Here, we shall present the case where $n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z}$.

Case III: $(n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z})$

- In each of these cases, one of the matrices fits into cache, and the others do not.
- Here, we shall present the case where $n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z}$.
- The Rec-Mult algorithm always divides m by 2 according to case (1).

Case III: $(n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z})$

- In each of these cases, one of the matrices fits into cache, and the others do not.
- Here, we shall present the case where $n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z}$.
- The Rec-Mult algorithm always divides m by 2 according to case (1).
- At some point in the recursion, m falls into the range $\alpha\sqrt{Z}/2 \leq m \leq \alpha\sqrt{Z}$, and the whole problem fits in cache.

Case III: $(n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z})$

- In each of these cases, one of the matrices fits into cache, and the others do not.
- Here, we shall present the case where $n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z}$.
- The REC-MULT algorithm always divides m by 2 according to case (1).
- At some point in the recursion, m falls into the range $\alpha\sqrt{Z}/2 \leq m \leq \alpha\sqrt{Z}$, and the whole problem fits in cache.
- The number cache misses can be described by the recurrence

$$Q(m, n, p) = \begin{cases} \Theta(1 + m) & \text{if } m \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m/2, n, p) + O(1) & \text{otherwise ;} \end{cases} \tag{6}$$

whose solution is $Q(m, n, p) = \Theta(m + mnp/(L\sqrt{Z}))$.

Case III: $(n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z})$

- In each of these cases, one of the matrices fits into cache, and the others do not.
- Here, we shall present the case where $n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z}$.
- The Rec-Mult algorithm always divides m by 2 according to case (1).
- At some point in the recursion, m falls into the range $\alpha\sqrt{Z}/2 \leq m \leq \alpha\sqrt{Z}$, and the whole problem fits in cache.
- The number cache misses can be described by the recurrence

$$Q(m, n, p) = \begin{cases} \Theta(1 + m) & \text{if } m \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m/2, n, p) + O(1) & \text{otherwise ;} \end{cases} \tag{6}$$

  whose solution is $Q(m, n, p) = \Theta(m + mnp/(L\sqrt{Z}))$.
- Indeed, in the base case: $mnp/(L\sqrt{Z}) \leq \alpha\sqrt{Z}m/L$; moreover $Z \in \Omega(L^2)$ (tall cache assumption).

Case IV: $m, n, p \leq \alpha\sqrt{Z}$.

- From the choice of α, all three matrices fit into cache.

Case IV: $m, n, p \leq \alpha\sqrt{Z}$.

- From the choice of α, all three matrices fit into cache.
- The matrices are stored on $\Theta(1 + mn/L + np/L + mp/L)$ cache lines.

Case IV: $m, n, p \leq \alpha\sqrt{Z}$.

- From the choice of α, all three matrices fit into cache.
- The matrices are stored on $\Theta(1 + mn/L + np/L + mp/L)$ cache lines.
- Therefore, we have $Q(m, n, p) = \Theta(1 + (mn + np + mp)/L)$.

Typical memory layouts for matrices

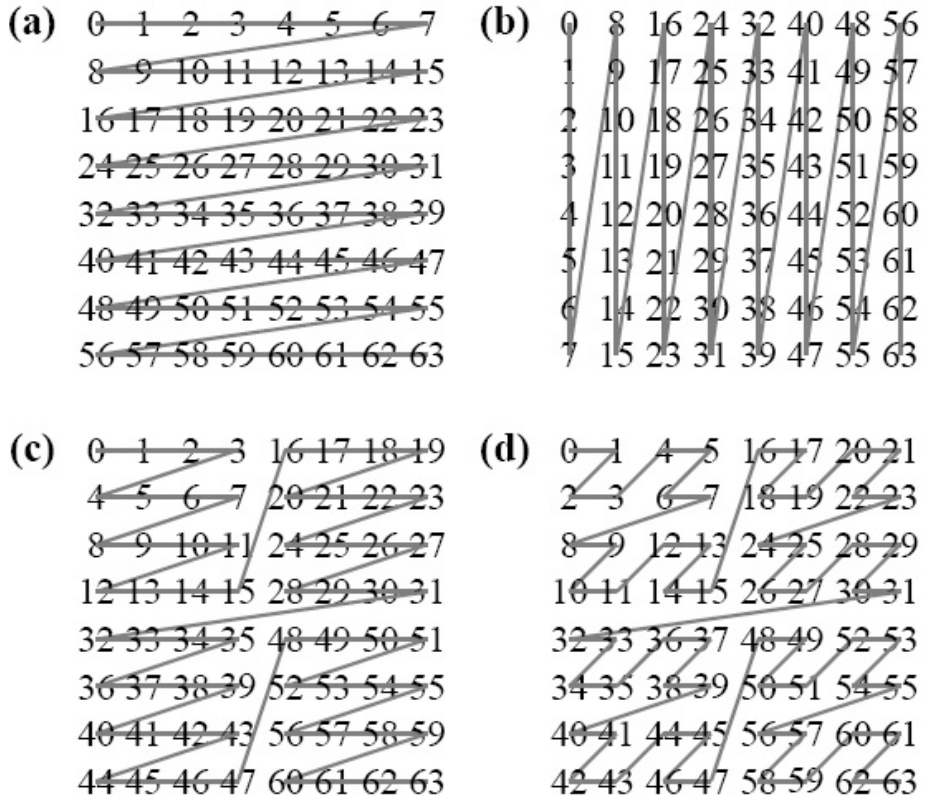


Figure 2: Layout of a 16×16 matrix in **(a)** row major, **(b)** column major, **(c)** 4×4-blocked, and **(d)** bit-interleaved layouts.

Acknowledgements and references

Acknowledgements.

- Charles E. Leiserson (MIT) and Matteo Frigo (Intel) for providing me with the sources of their article *Cache-Oblivious Algorithms*.
- Charles E. Leiserson (MIT) and Saman P. Amarasinghe (MIT) for sharing with me the sources of their course notes and other documents.

References.

- *Cache-Oblivious Algorithms* by Matteo Frigo, Charles E. Leiserson, Harald Prokop and Sridhar Ramachandran.
- *Cache-Oblivious Algorithms and Data Structures* by Erik D. Demaine.

References

[1] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[2] J. E. Savage. *Models of computation - exploring the power of computing*. Addison-Wesley, 1998. ISBN: 978-0-201-89539-1.

[3] M. L. Scott. *Programming Language Pragmatics (3. ed.)* Academic Press, 2009. ISBN: 978-0-12-374514-9.

[4] A. Williams. *C++ concurrency in action: practical multithreading; 1st ed.* Shelter Island, NY: Manning Publ., 2012. URL: https://cds.cern.ch/record/1483005.