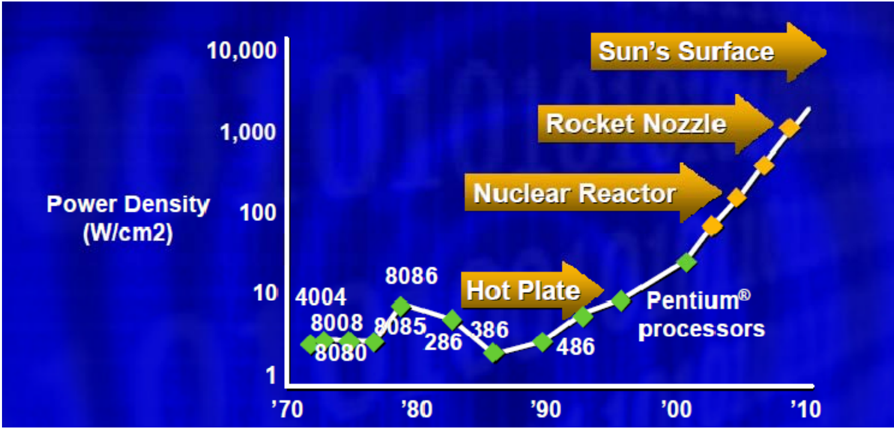
Cache Memories, Cache Complexity

Marc Moreno Maza

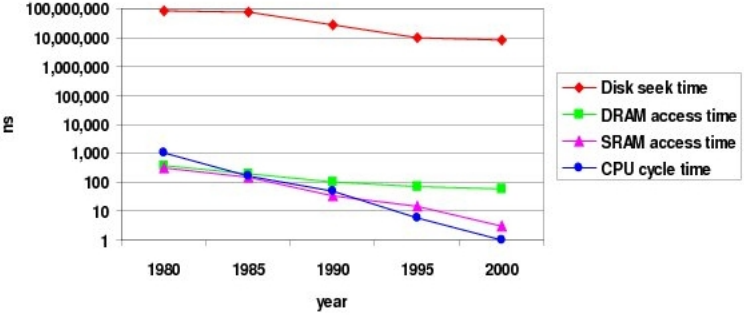
University of Western Ontario, London (Canada)

Applications of Computer Algebra
Session on High-Performance Computer Algebra
Jerusalem College of Technology, July 20, 2017


10,000
1,000
100
10
1
Power Density
(W/cm2)
Sun's Surface
Rocket Nozzle
Nuclear Reactor
Hot Plate
4004
8008
8080
8085
8086
286
386
486
Pentium®
processors
'70
'80
'90
'00
'10

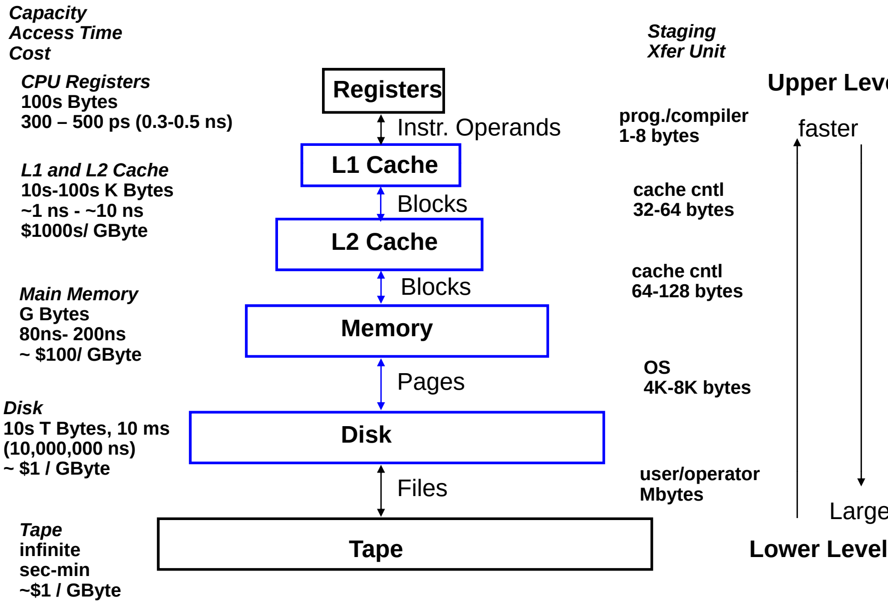
The CPU-Memory Gap

The increasing gap between DRAM, disk, and CPU speeds.

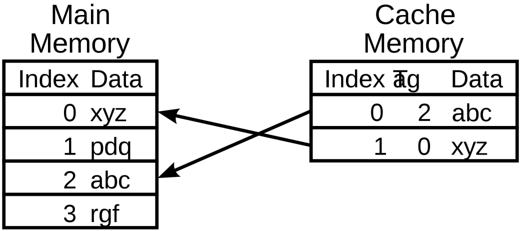


Plan

Plan

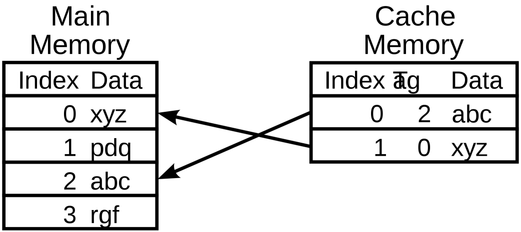

Capacity
Access Time
Cost
CPU Registers
100s Bytes
300 – 500 ps (0.3-0.5 ns)
L1 and L2 Cache
10s-100s K Bytes
~1 ns - ~10 ns
$1000s/ GByte
Main Memory
G Bytes
80ns- 200ns
~ $100/ GByte
Disk
10s T Bytes, 10 ms
(10,000,000 ns)
~ $1 / GByte
Tape
infinite
sec-min
~$1 / GByte
Registers
Instr. Operands
L1 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
faster
Lower Level

CPU Cache (1/3)



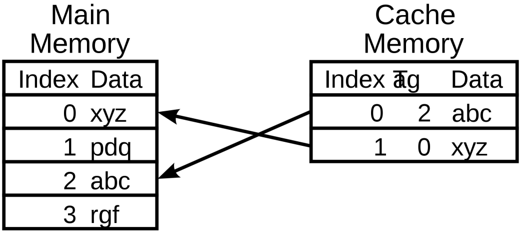
- A CPU cache is an auxiliary memory which is smaller, faster memory than the main memory and which stores copies of the main memory locations that are expectedly frequently used.
- Most modern desktop and server CPUs have at least three independent caches: the data cache, the instruction cache and the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

CPU Cache (2/3)



- Each location in each memory (main or cache) has
  - a datum (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 (3/3)

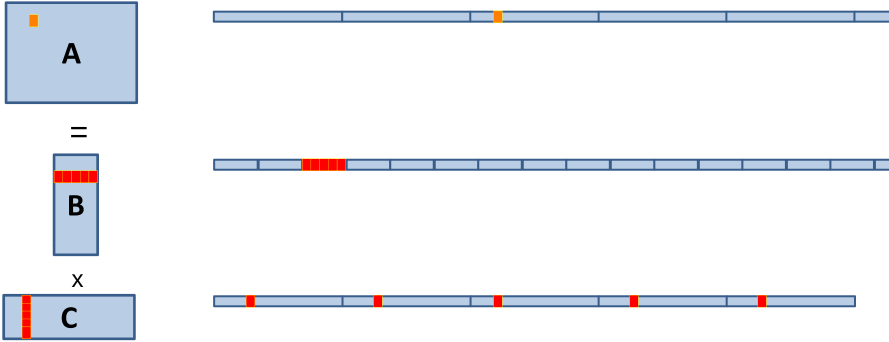


- When the CPU needs to read or write a location, it checks the cache:
  - if it finds it there, we have a cache hit
  - if not, we have a cache miss and (in most cases) the processor needs to create a new entry in the cache.
- Making room for a new entry requires a replacement policy: the Least Recently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

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

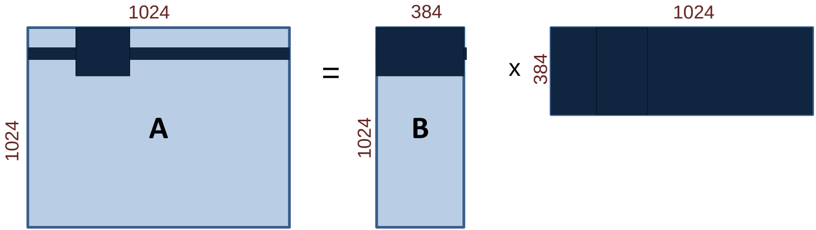


- Contiguous accesses are better:
  - Data fetch as cache line (Core 2 Duo 64 byte per cache line)
  - With contiguous data, a single cache fetch supports 8 reads of doubles.
  - Transposing the matrix C should reduce L1 cache misses!

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



- 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 $= 25,600$.
- 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 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.

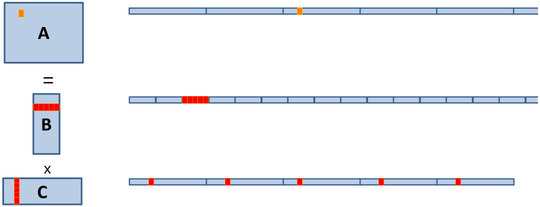
Other performance counters

Hardware count 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.

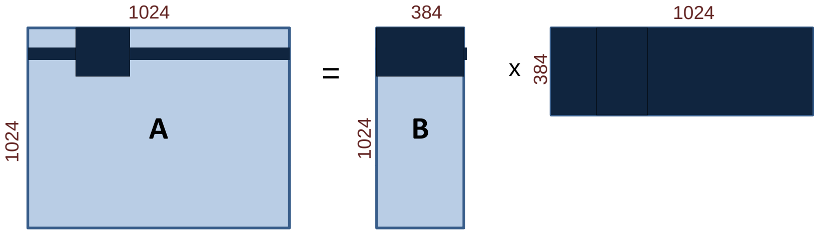
	CPI		L1 Miss Rate		L2 Miss Rate	Percent SSE Instructions	Instructions Retired	
In C	4.78		0.24		0.02	43%	13,137,280,000	
		5x		2x				1x
Transposed	1.13		0.15		0.02	50%	13,001,486,336	
		3x		8x				0.8x
Tiled	0.49		0.02		0	39%	18,044,811,264	

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 one, 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 per flop!**

Analyzing cache misses in the tiled multiplication



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

Counting sort: the algorithm

- *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.

```
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: cache complexity analysis

```
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).

   Total: $3n + 3n/L + 2k/L$ cache misses (worst case).

Counting sort: cache complexity analysis: 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.
3. $n/L + n$ cache misses for the histogram (worst case): `items` accesses are linear but `Count` accesses are potentially random.
4. k/L cache misses for the prefix sum: `Count` accesses are linear.
5. $n/L + n + n$ cache misses for building `Output` (worst case): `items` accesses are linear but `Output` and `Count` accesses are potentially random.

   Total: $3n + 3n/L + 2k/L$ cache misses (worst case).

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 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
```

- Goal: after preprocessing, `Count` and `Output` incur **cold misses only**.
- To this end we choose a parameter m (more on this later) such that
  1. a key in the range $[ih, (i+1)h - 1]$ is always before a 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`
2. **Key observation:** `bucketedinput` is traversed regularly by segment.
3. Hence, $2n/L + m + m/L$ caches misses to compute `bucketedinput`

   Preprocessing: $3n/L + 3m/L + m$ 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,
   - $m/L + n/L$ 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.
- 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;

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

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

Cache friendly counting sort: experimental results

- 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-oblivious counting sort (preprocessing + 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)

Plan

The (Z, L) ideal cache model (1/4)

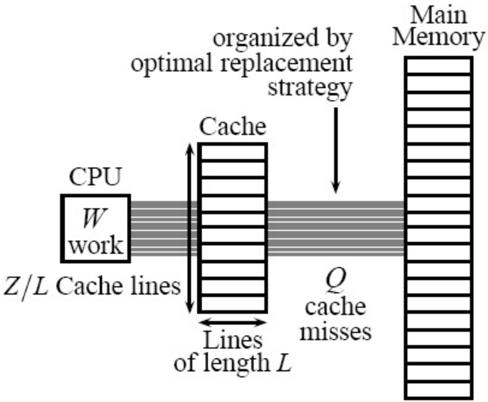


Figure 1: The ideal-cache model

The (Z, L) ideal cache model (2/4)

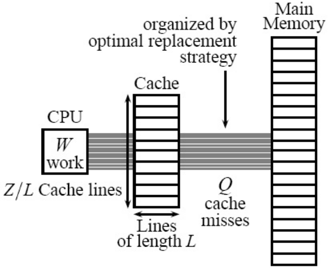


Figure 1: The ideal-cache model

- 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 (Z, L) ideal cache model (3/4)

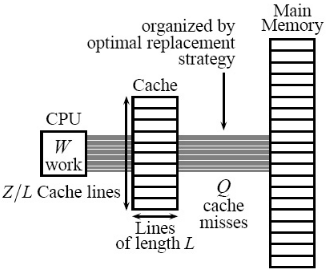


Figure 1: The ideal-cache model

- 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 into the cache.

The (Z, L) ideal cache model (4/4)

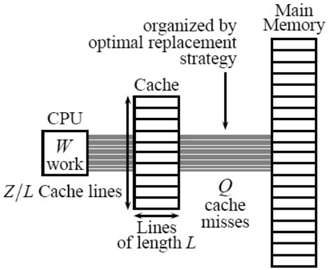


Figure 1: The ideal-cache model

- 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 replacing** the cache line whose next access is furthest in the future, and thus it exploits temporal locality perfectly.

Cache complexity

- For an algorithm with an input of size n, he 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 $Q(n; Z, L)$.
- 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 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];
```

- Assuming $Z \geq 3L$, computing each `C[i][j]` incurs $O(1 + n/L)$ caches misses.
- If Z large enough, say $Z \in \Omega(n)$ then the 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

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

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.
- 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, $s \in \Theta(\sqrt{Z})$.
- 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

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

Plan

Scanning

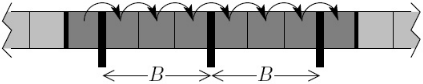


Figure 2. Scanning an array of N elements arbitrarily aligned with blocks may cost one more memory transfer than $\lceil N/B \rceil$.

Scanning n elements stored in a contiguous segment (= cache lines) of memory costs at most $\lceil n/L \rceil + 1$ cache misses.

Indeed:

- In the above, $N = n$ and $B = L$. The main issue here is alignment.
- Let (q, r) be the quotient and remainder in the integer division of n by L. Let u (resp. w) be # words in a fully (not fully) used cache line.
- If $w = 0$ then $r = 0$ and the conclusion is clear.
- If $w < L$ then $r = w$ and the conclusion is clear again.

Array reversal

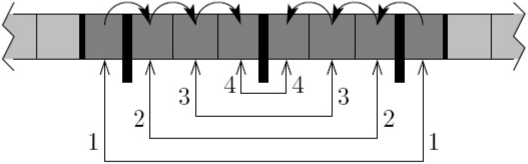


Figure 3. Bentley's reversal of an array.

Reversing an array of n elements stored in a contiguous segment (= cache lines) of memory costs at most $\lceil n/L \rceil + 1$ cache misses, provided that $Z \geq 2L$ holds. Exercise!

Median and selection (1/8)

- A selection algorithm is an algorithm for finding the k-th smallest number in a list. This includes the cases of finding the minimum, maximum, and median elements.
- A worst-case linear algorithm for the general case of selecting the k-th largest element was published by Blum, Floyd, Pratt, Rivest, and Tarjan in their 1973 paper *Time bounds for selection*, sometimes called BFPRT.
- The principle is the following:
  - Find a *pivot* that allows splitting the list into two parts of nearly equal size such that
  - the search can continue in one of them.

Median and selection (2/8)

```
select(L,k)
{
if (L has 10 or fewer elements)
{
    sort L
    return the element in the kth position
}

partition L into subsets S[i] of five elements each
    (there will be n/5 subsets total).

for (i = 1 to n/5) do
    x[i] = select(S[i],3)

M = select({x[i]}, n/10)

partition L into L1<M, L2=M, L3>M
if (k <= length(L1))
    return select(L1,k)
else if (k > length(L1)+length(L2))
    return select(L3,k-length(L1)-length(L2))
else return M
```

Median and selection (3/8)

For an input list of n elements, the number $T(n)$ of comparisons satisfies

$$T(n) \leq 12n/5 + T(n/5) + T(7n/10).$$

- We always throw away either `L3` (the values greater than `M`) or `L1` (the values less than `M`). Suppose we throw away `L3`.
- Among the $n/5$ values `x[i]`, $n/10$ are larger than `M`, since `M` was defined to be the median of these values.
- For each `i` such that `x[i]` is larger than `M`, two other values in `S[i]` are also larger than `x[i]`
- So `L3` has at least $3n/10$ elements. By a symmetric argument, `L1` has at least $3n/10$ elements.
- Therefore the final recursive call is on a list of at most $7n/10$ elements and takes time at most $T(7n/10)$.

Median and selection (4/8)

How to solve

$$T(n) \leq 12n/5 + T(n/5) + T(7n/10)?$$

- We "try" $T(n) \leq c\,n$ by induction. The substitution gives

  $$T(n) \leq n\,(12/5 + 9c/10).$$

  From $n(12/5 + 9c/10) \leq c\,n$ we derive $c \leq 24$.
- The tree-based method also brings $T(n) \leq 24n$.
- The same tree-expansion method then shows that, more generally, if $T(n) \leq cn + T(an) + T(bn)$, where $a + b < 1$, the total time is $c(1/(1 - a - b))n$.
- With a lot of work one can reduce the number of comparisons to $2.95n$ [D. Dor and U. Zwick, *Selecting the Median*, 6th SODA, 1995].

Median and selection (5/8)

In order to analyze its cache complexity, let us review the algorithm and consider an array instead of a list.

Step 1: Conceptually partition the array into $n/5$ quintuplets of five adjacent elements each.

Step 2: Compute the median of each quintuplet using $O(1)$ comparisons.

Step 3: Recursively compute the median of these medians (which is not necessarily the median of the original array).

Step 4: Partition the elements of the array into three groups, according to whether they equal, or strictly less or strictly greater than this median of medians.

Step 5: Count the number of elements in each group, and recurse into the group that contains the element of the desired rank.

Median and selection (6/8)

To make this algorithm cache-oblivious, we specify how each step works in terms of memory layout and scanning. We assume that $Z \geq 3L$.

Step 1: Just conceptual; no work needs to be done.

Step 2: requires two parallel scans, one reading the 5 element arrays at a time, and the other writing a new array of computed medians, incurring $\Theta(1 + n/L)$.

Step 3: Just a recursive call on size $n/5$.

Step 4: Can be done with three parallel scans, one reading the array, and two others writing the partitioned arrays, incurring again $\Theta(1 + n/L)$.

Step 5: Just a recursive call on size $7n/10$.

This leads to

$$Q(n) \leq Q(n/5) + Q(7n/10) + \Theta(1 + n/L).$$

Median and selection (7/8)

How to solve

$$Q(n) \leq Q(n/5) + Q(7n/10) + \Theta(1 + n/L)?$$

The unknown is what is the **base-case**?

- Suppose the base case is $Q(0(1)) \in O(1)$.
- Following *Master Theorem* proof the number of leaves $L(n) = n^c$ satisfies in $N(n) = N(n/5) + N(7n/10), N(1) = 1$, which brings
$$\left(\frac{1}{5}\right)^c + \left(\frac{7}{10}\right)^c = 1$$
leading to $c \simeq 0.8397803$.
- Since each leaf incurs a constant number of cache misses we have $Q(n) \in \Omega(n^c)$, which could be larger or smaller than $\Theta(1 + n/L)$ ...

Median and selection (8/8)

How to solve

$$Q(n) \leq Q(n/5) + Q(7n/10) + \Theta(1 + n/L)?$$

- Fortunately, we have a better **base-case**: $Q(0(L)) \in O(1)$.
- Indeed, once the problem fits into $O(1)$ cache-lines, all five steps incur only a constant number of cache misses.
- Thus we have only $(n/L)^c$ leaves in the recursion tree.
- In total, these leaves incur $O((n/L)^c) = o(n/L)$ cache misses.
- In fact, the cost per level decreases geometrically from the root, so the total cost is the cost of the root. Finally we have

$$Q(n) \in \Theta(1 + n/L)$$

Plan

Matrix transposition: various algorithms

- **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 shall start with an apparently good algorithm and use complexity analysis to show that it is even worse than the straightforward 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.
  2. If $n > 1$ then
     2.1 recursively compute ${}^tA_{1,1}, {}^tA_{2,1}, {}^tA_{1,2}, {}^tA_{2,2}$ in place as

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

     2.2 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

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

- 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.
- 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

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 (more details in class) 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).
- 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.
- 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.
- 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.
- 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 $[\alpha L/2, \alpha L]$.
- 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.

Plan

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:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \quad (1)$$

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

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

- 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.

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

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

- 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)

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

- 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 $n, p > \alpha\sqrt{Z}$.
- 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

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

  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: $(n, p \leq \alpha\sqrt{Z}$ 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

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

  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.
- 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

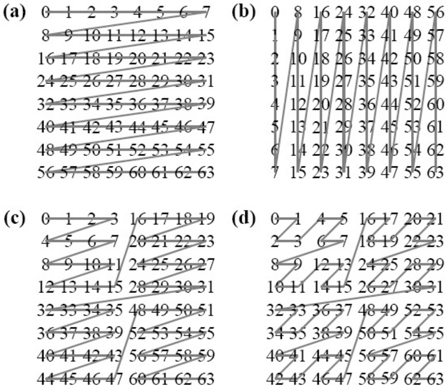


Figure 2: Layout of a 16×16 matrix in **(a)** row major, **(b)** column major, **(c)** 4×4-blocked, and **(d)** bit-interleaved layouts.

Plan

Tuned cache-oblivious square matrix transposition

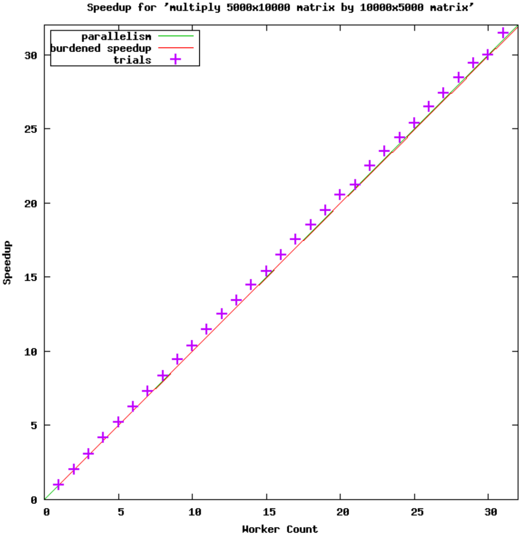
```
void DC_matrix_transpose(int *A, int lda, int i0, int i1,
    int j0, int dj0, int j1 /*, int dj1 = 0 */){
    const int THRESHOLD = 16; // tuned for the target machine
  tail:
    int di = i1 - i0, dj = j1 - j0;
    if (dj >= 2 * di && dj > THRESHOLD) {
        int dj2 = dj / 2;
        cilk_spawn 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 ration comes simply from an **optimal memory access pattern.**

Tuned cache-oblivious matrix multiplication



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.