Analyzing Data Access of Algorithms and How to Make Them Cache-Friendly?

Kenjiro Taura

Contents

Contents

Motivation

- we've learned data access is an important factor that determines performance of your program

Motivation

- we've learned data access is an important factor that determines performance of your program
- it is thus clearly important to analyze how "good" your algorithm is, in terms of data access performance

Motivation

- we've learned data access is an important factor that determines performance of your program
- it is thus clearly important to analyze how "good" your algorithm is, in terms of data access performance
- we routinely analyze computational complexity of algorithms to predict or explain algorithm performance, but it ignores the differing costs of accessing memory hierarchy (all memory accesses are $O(1)$)

Motivation

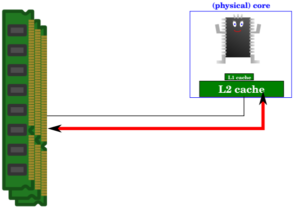
- we've learned data access is an important factor that determines performance of your program
- it is thus clearly important to analyze how "good" your algorithm is, in terms of data access performance
- we routinely analyze computational complexity of algorithms to predict or explain algorithm performance, but it ignores the differing costs of accessing memory hierarchy (all memory accesses are $O(1)$)
- we like to do *something analogous for data access*

Contents

Contents

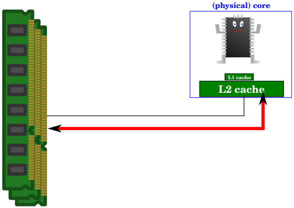
Analyzing data access performance of serial programs

- we like to analyze data access cost of serial programs (on pencils and papers)



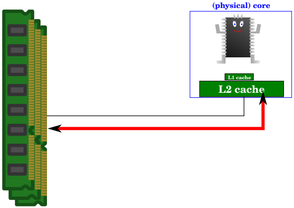
Analyzing data access performance of serial programs

- we like to analyze data access cost of serial programs (on pencils and papers)
- for this purpose, we calculate the amount of *data transferred between levels of memory hierarchy*



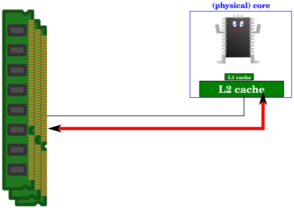
Analyzing data access performance of serial programs

- we like to analyze data access cost of serial programs (on pencils and papers)
- for this purpose, we calculate the amount of *data transferred between levels of memory hierarchy*
- in practical terms, this is a proxy of *cache misses*



Analyzing data access performance of serial programs

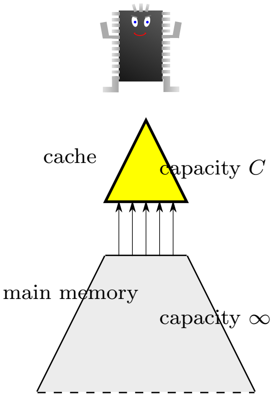
- we like to analyze data access cost of serial programs (on pencils and papers)
- for this purpose, we calculate the amount of *data transferred between levels of memory hierarchy*
- in practical terms, this is a proxy of *cache misses*
- the analysis assumes a simple two level memory hierarchy



Contents

Model (of a single processor machine)

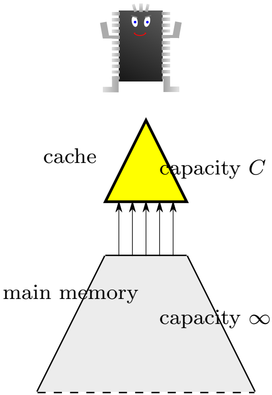
- a machine consists of
  - a processor,
  - a fixed size cache (C words), and
  - an arbitrarily large main memory
  - accessing a word not in the cache mandates transferring the word into the cache



For now, we assume a single processor machine.

Model (of a single processor machine)

- a machine consists of
  - a processor,
  - a fixed size cache (C words), and
  - an arbitrarily large main memory
  - accessing a word not in the cache mandates transferring the word into the cache
- the cache holds the most recently accessed C words; $\approx$
  - line size = single word (whatever is convenient)
  - fully associative
  - LRU replacement

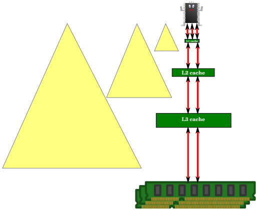


For now, we assume a single processor machine.

Gaps between our model and real machines

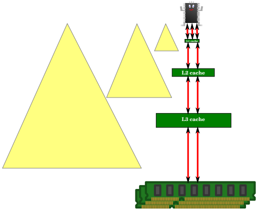
- hierarchical caches:

⇒ each level can be easily modeled separately, with caches of various sizes



Gaps between our model and real machines

- hierarchical caches:

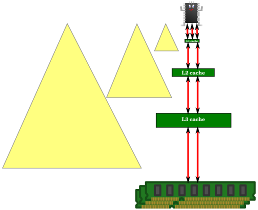
  ⇒ each level can be easily modeled separately, with caches of various sizes

- concurrent accesses:
  - the model only counts the amount of data transferred
  - in practice the cost heavily depends on how many concurrent accesses you have
  - this model cannot capture the difference between link list traversal and random array access

Gaps between our model and real machines

- hierarchical caches:

  ⇒ each level can be easily modeled separately, with caches of various sizes

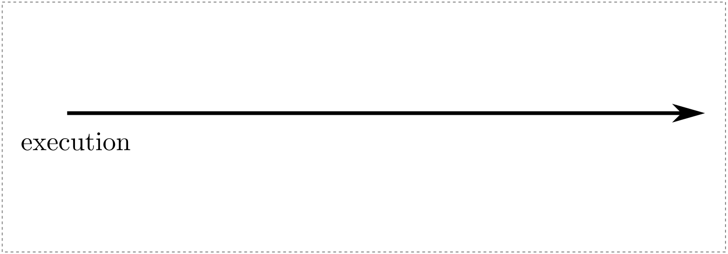


- concurrent accesses:
  - the model only counts the amount of data transferred
  - in practice the cost heavily depends on how many concurrent accesses you have
  - this model cannot capture the difference between link list traversal and random array access
- prefetch:
  - similarly, this model cannot capture the difference between sequential accesses that can take advantages of the hardware prefetcher and random accesses that cannot

Contents

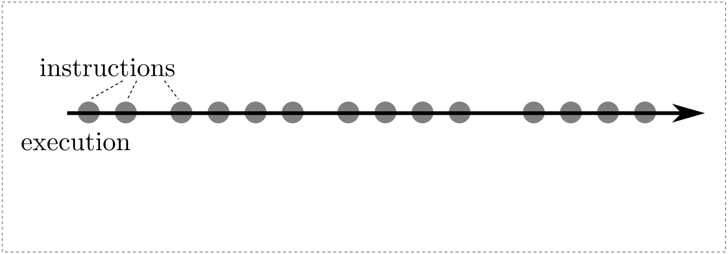
Terminologies

- an *execution* of a program is the sequence of instructions



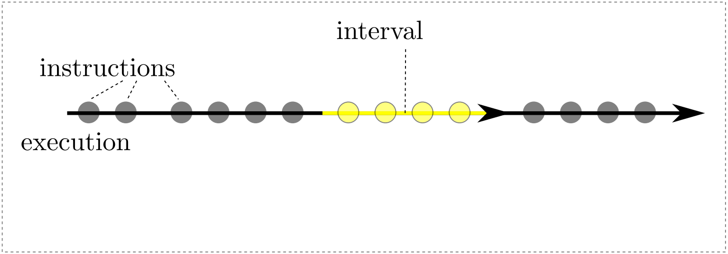
Terminologies

- an *execution* of a program is the sequence of instructions



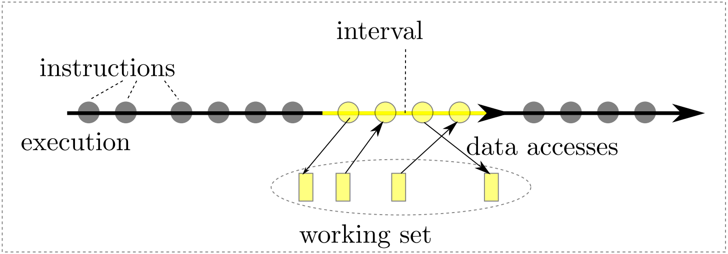
Terminologies

- an *execution* of a program is the sequence of instructions
- an *interval* in an execution is a consecutive sequence of instructions in the execution



Terminologies

- an *execution* of a program is the sequence of instructions
- an *interval* in an execution is a consecutive sequence of instructions in the execution
- the *working set* of an interval is the number of *distinct words* accessed by the interval

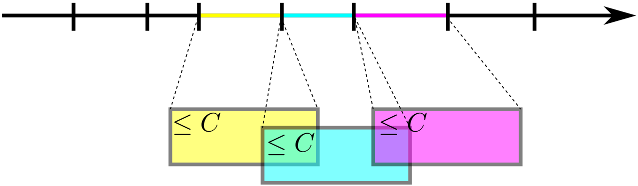


A basic methodology

to calculate the amount of data transferred in an execution,

1. split an execution into intervals each of which fits in the cache; i.e.,

$$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$



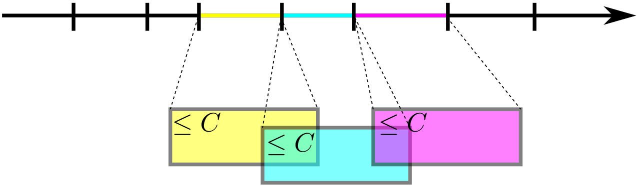
A basic methodology

to calculate the amount of data transferred in an execution,

1. split an execution into intervals each of which fits in the cache; i.e.,

   $$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$

2. then, each interval transfers at most C words, because each word misses at most only once in the interval



A basic methodology

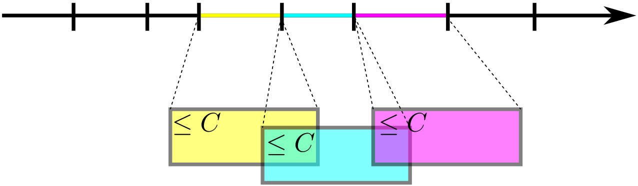
to calculate the amount of data transferred in an execution,

1. split an execution into intervals each of which fits in the cache; i.e.,

$$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$

2. then, each interval transfers at most C words, because each word misses at most only once in the interval
3. therefore,

$$\text{data transferred} \leq \sum_{I:\text{all intervals}} \text{working set size of } I$$



Remarks

- the condition $(*)$ is important to bound data transfers from above

$$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$

Remarks

- the condition $(*)$ is important to bound data transfers from above

$$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$

- each word in an interval misses at most only once, because
  - the cache is LRU, and
  - the condition $(*)$ guarantees that each word is never evicted within the interval

Remarks

- the condition $(*)$ is important to bound data transfers from above

$$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$

- each word in an interval misses at most only once, because
  - the cache is LRU, and
  - the condition $(*)$ guarantees that each word is never evicted within the interval
- in practical terms, an essential step to analyze data transfer is to *identify the largest intervals that fits in the cache*

Contents

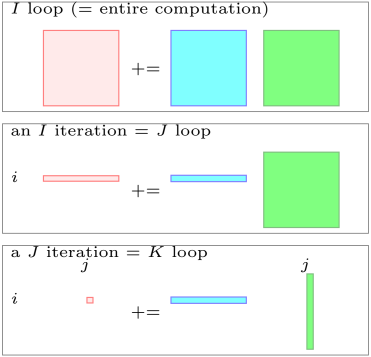
Applying the methodology

- we will apply the methodology to some of the algorithms we have studied so far
- the key is to *find subproblems (intervals) that fit in the cache*

Analyzing the triply nested loop

```
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(k,j);
    }
  }
}
```

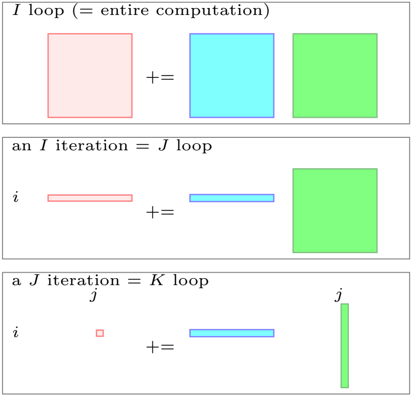
- perform n^3 FMAs on $3n^2$ words
- key question: *which iteration fits in the cache?*



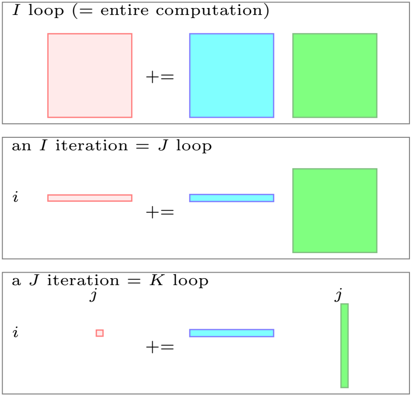
Working sets

```
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(k,j);
    }
  }
}
```

level	working set size
I loop	$3n^2$
J loop	$2n + n^2$
K loop	$1 + 2n$
K iteration	3

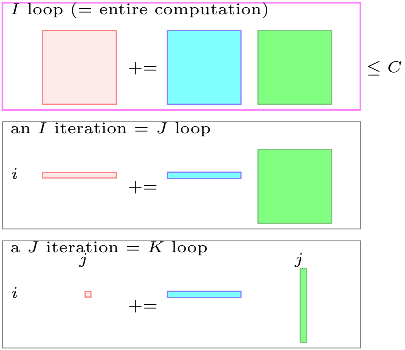


Cases to consider



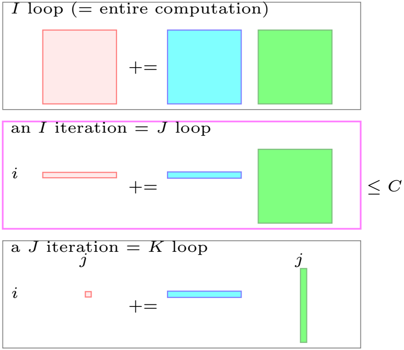
Cases to consider

- Case 1: working set of the entire I-loop fits in the cache ($3n^2 \leq C$)



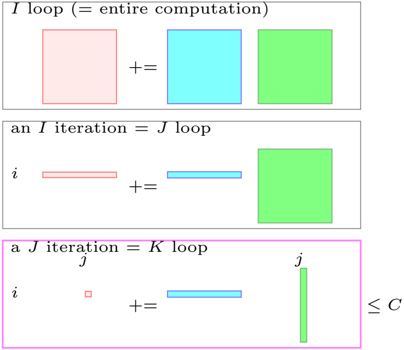
Cases to consider

- Case 1: working set of the entire I-loop fits in the cache ($3n^2 \leq C$)
- Case 2: working set of a single I-iteration (= J-loop) fits in the cache ($2n + n^2 \leq C$)



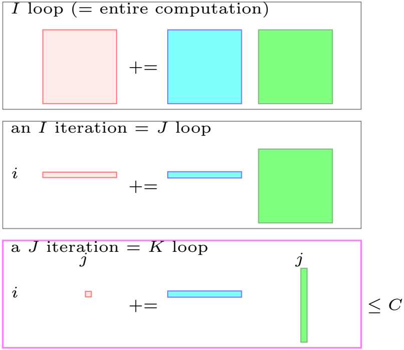
Cases to consider

- Case 1: working set of the entire I-loop fits in the cache ($3n^2 \leq C$)
- Case 2: working set of a single I-iteration ($= J$-loop) fits in the cache ($2n + n^2 \leq C$)
- Case 3: working set of a single J-iteration ($= K$-loop) fits in the cache ($1 + 2n \leq C$)



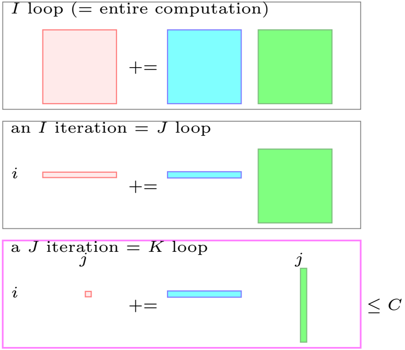
Cases to consider

- Case 1: working set of the entire I-loop fits in the cache ($3n^2 \leq C$)
- Case 2: working set of a single I-iteration (= J-loop) fits in the cache ($2n + n^2 \leq C$)
- Case 3: working set of a single J-iteration (= K-loop) fits in the cache ($1 + 2n \leq C$)
- Case 4: none of the above ($1 + 2n > C$)



Cases to consider

- Case 1: working set of the entire I-loop fits in the cache ($3n^2 \leq C$)
- Case 2: working set of a single I-iteration (= J-loop) fits in the cache ($2n + n^2 \leq C$)
- Case 3: working set of a single J-iteration (= K-loop) fits in the cache ($1 + 2n \leq C$)
- Case 4: none of the above ($1 + 2n > C$)

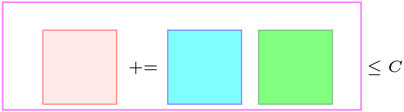


Goal: for each case, bound $R(n)$ from the above, the traffic between memory and cache for $n \times n$ matrix-multiply

Case 1 ($3n^2 \leq C$)

- trivially, each element misses the cache only once. thus,

$$R(n) \leq 3n^2 = \frac{3}{n} \cdot n^3$$

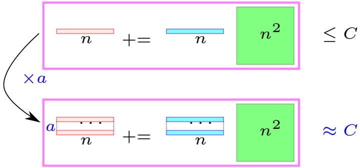


- interpretation: each element of A, B, and C are reused n times

Case 2 ($2n + n^2 \leq C$)

- the maximum number of I-iterations whose working set fit in the cache is:

$$a \approx \frac{C - n^2}{2n}$$



- in a iterations, each element misses only once, so

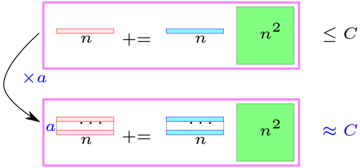
$$R(n) \leq= \frac{n}{a}(n^2 + 2an) = \left(\frac{1}{a} + \frac{2}{n}\right) n^3$$

- interpretation: each element of B is reused a times in the cache; each element in A or C n times

A Remark

- for a particular access pattern of the matrix multiplication, a better bound can be obtained

- as we know all elements of B are accessed in the same order in each I-iteration, B stays in the cache for any number of a iterations



- so, in this case too, each element misses only once throughout the entire computation

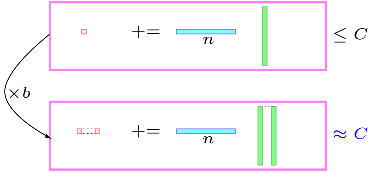
$$\therefore R(n) \le 3n^2 = \frac{3}{n} \cdot n^3$$

- note that this analysis is specific to matrix-matrix multiplication

Case 3 ($1 + 2n \leq C$)

- the maximum number of j-iterations that fit in the cache is:

$$b \approx \frac{C - n}{n + 1}$$



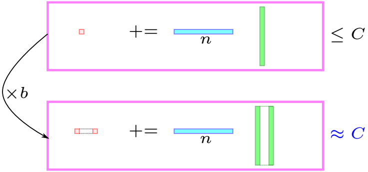
- in b iterations, each element misses only once, so

$$R(n) \leq \frac{n^2}{b}(n+b(n+1)) = \left(\frac{1}{b} + 1 + \frac{1}{n}\right)n^3$$

interpretation: each element in B is never reused; each element in A b times; each clement in C many ($\propto n$) times

A Remark

- a similar argument shows that an entire row of A stays in the cache for any number of b iterations
- so each element misses only once in the entire J-loop (an iteration of the I-loop)

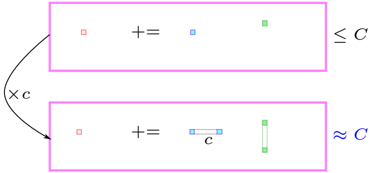


$$\therefore R(n) \leq (2n + n^2)n = n^3 + 2n^2$$

Case 4 $(1 + 2n > C)$

- the maximum number of K-iterations that fit in the cache is:

$$c \approx \frac{C-1}{2}$$



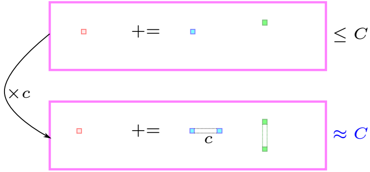
- in each c iterations, each element misses only once, so

$$R(n) \leq \frac{n^3}{c}(2c+1) = \left(2 + \frac{1}{c}\right) n^3$$

interpretation: each element of B or A never reused; each element of C reused c times

A Remark

- similarly, the element of C stays in the cache for any number of c iterations
- so, each element misses only once in the entire k-loop (an iteration of the j-loop)



$$R(n) \le (2n+1)n^2 = 2n^3 + n^2$$

interpretation: each element of B or A never reused; each element of C reused n times

Summary

- $n^3/R(n)$ or the number of FMAs per memory accesses (cache miss)

condition	$R(n)$	$n^3/R(n)$
$2n + n^2 \leq C$	$3n^2$	$n/3$
$1 + 2n \leq C$	$2n^2 + n^3$	≈ 1
$1 + 2n > C$	$n^2 + 2n^3$	$\approx 1/2$

Can we do better for large matrices?

- so far, we've discussed the traffic of the straightforward triply-nested loop
- its conclusion can be summarized as
  - if a single matrix is larger than B, access to B almost induces a cache miss
  - can we do better for large matrices?

condition	$R(n)$	$n^3/R(n)$
$2n + n^2 \leq C$	$3n^2$	$n/3$
$1 + 2n \leq C$	$2n^2 + n^3$	≈ 1
$1 + 2n > C$	$n^2 + 2n^3$	$\approx 1/2$

A general strategy

- in general, the traffic increases when *the same amount of computation has a large working set*

A general strategy

- in general, the traffic increases when *the same amount of computation has a large working set*
- assuming the entire computaiton you want to perform is given/fixed, to reduce the traffic, you arrange *order* subcomputations so that *each subcomputation does a lot of computation on the same amount of data*

A general strategy

- in general, the traffic increases when *the same amount of computation has a large working set*
- assuming the entire computaiton you want to perform is given/fixed, to reduce the traffic, you arrange *order* subcomputations so that *each subcomputation does a lot of computation on the same amount of data*
- the notion is so important that it is variously called
    - compute/data ratio,
    - flops/byte,
    - compute intensity, or
    - arithmetic intensity

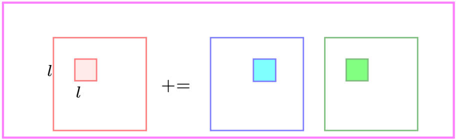
A general strategy

- in general, the traffic increases when *the same amount of computation has a large working set*
- assuming the entire computaiton you want to perform is given/fixed, to reduce the traffic, you arrange *order* subcomputations so that *each subcomputation does a lot of computation on the same amount of data*
- the notion is so important that it is variously called
  - compute/data ratio,
  - flops/byte,
  - compute intensity, or
  - arithmetic intensity
- the key is to identify the unit of computation (task) whose compute intensity is high (compute-intensive task)

Cache blocking (tiling)

- for matrix multiplication, let l be the maximum number that satisfies $2l + l^2 \leq C$ (i.e., $l \approx \sqrt{C}$) and form a subcomputation that performs a $(l \times l)$ matrix multiplication
- ignoring remainder iterations, it looks like:

```
l = √C − small constant;
for (ii = 0; ii < n; ii += l)
 for (jj = 0; jj < n; jj += l)
  for (kk = 0; kk < n; kk += l)
   /* block below misses each
   element at most once */
   for (i = ii; i < ii + l; i++)
    for (j = jj; j < jj + l; j++)
     for (k = kk; k < kk + l; k++)
       A(i,j) += B(i,k) * C(k,j);
```



Cache blocking (tiling)

- each block:
  - performs l^3 FMAs and
  - touches $3l^2$ distinct words,

```
l = √C – small constant;
for (ii = 0; ii < n; ii += l)
 for (jj = 0; jj < n; jj += l)
  for (kk = 0; kk < n; kk += l)
   /* block below misses each
   element at most once */
   for (i = ii; i < ii + l; i++)
    for (j = jj; j < jj + l; j++)
     for (k = kk; k < kk + l; k++)
      A(i,j) += B(i,k) * C(k,j);
```

Cache blocking (tiling)

- each block:
  - performs l^3 FMAs and
  - touches $3l^2$ distinct words,
- so, the traffic of the entire computation

$$\leq 3l^2 \cdot \left(\frac{n}{l}\right)^3 = \frac{3}{l}n^3 = \frac{3}{\sqrt{C}}n^3$$

```
l = √C – small constant;
for (ii = 0; ii < n; ii += l)
 for (jj = 0; jj < n; jj += l)
  for (kk = 0; kk < n; kk += l)
   /* block below misses each
   element at most once */
   for (i = ii; i < ii + l; i++)
    for (j = jj; j < jj + l; j++)
     for (k = kk; k < kk + l; k++)
      A(i,j) += B(i,k) * C(k,j);
```

Effect of cache blocking

condition	$R(n)$	$n^3/R(n)$
$2n + n^2 \leq C$	$3n^2$	$n/3$
$1 + 2n \leq C$	$2n^2 + n^3$	~ 1
$1 + 2n > C$	$n^2 + 2n^3$	$\sim 1/2$
blocking	$\dfrac{3}{\sqrt{C}} n^3$	$\dfrac{\sqrt{C}}{3}$

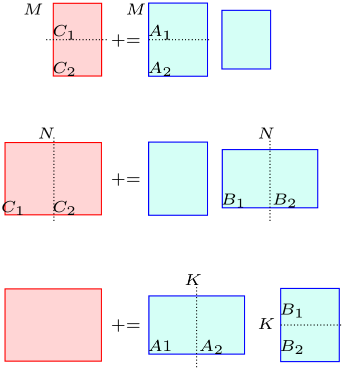
- assume a word = 4 bytes (`float`)

bytes	C	l	$n^3/R(n)$
32K	8K	≈ 90.50	30.16
256K	64K	≈ 256	85.33
3MB	768K	≈ 886	295.60

Recursive blocking

- the tiling technique just mentioned needs to determine l for a particular size (= level)
- it may have to do this at all levels (12 deep nested loop)?
- we also (for the sake of simplicity) assumed all matrices are square
- for generality, portability, simplicity, recursive blocking may apply

Recursively blocked matrix multiply




```
gemm(A, B, C) {
  if ((M, N, K) = (1, 1, 1)) {
    c11 += a11 * b11;
  } else if (max(M, N, K) = M) {
    gemm(A1, B, C1);
    gemm(A2, B, C2);
  } else if (max(M, N, K) = N) {
    gemm(A, B1, C1);
    gemm(A, B1, C2);
  } else { /* max(M, N, K) = K */
    gemm(A1, B1, C);
    gemm(A2, B2, C);
  }
}
```

- it divides flops into two
- it divides two of the three matrices, along *the longest* axis

Settings

- a single word = a single floating point number
- cache size $= C$ words

Settings

- a single word = a single floating point number
- cache size $= C$ words
- let $R(M, N, K)$ be the number of words transferred between cache and memory when multiplying $M \times K$ and $K \times N$ matrices (the cache is initially empty)

Settings

- a single word = a single floating point number
- cache size $= C$ words
- let $R(M, N, K)$ be the number of words transferred between cache and memory when multiplying $M \times K$ and $K \times N$ matrices (the cache is initially empty)
- let $R(w)$ be the maximum number of words transferred for any matrix multiply of up to w words in total:

$$R(w) \equiv \max_{MK+KN+MN \leq w} R(M, N, K)$$

we want to bound $R(w)$ from the above

Settings

- a single word = a single floating point number
- cache size = C words
- let $R(M, N, K)$ be the number of words transferred between cache and memory when multiplying $M \times K$ and $K \times N$ matrices (the cache is initially empty)
- let $R(w)$ be the maximum number of words transferred for any matrix multiply of up to w words in total:

$$R(w) \equiv \max_{MK+KN+MN \le w} R(M, N, K)$$

  we want to bound $R(w)$ from the above
- to avoid making analysis tedious, assume all matrices are "nearly square"

$$\max(M, N, K) \le 2 \min(M, N, K)$$

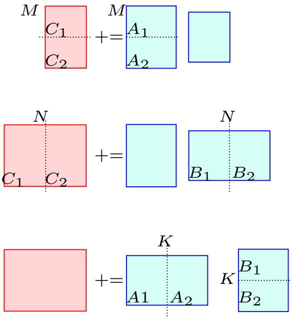
The largest subproblem that fits in the cache

- the working set of `gemm(A,B,C)` is $(MK + KN + MN)$ (words)
- it fits in the cache if this is $\leq C$
- thus we have:

$$\therefore R(w) \leq C \quad (w \leq C)$$

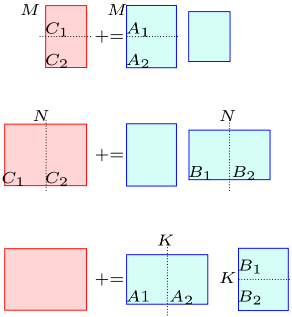
Analyzing cases that do not fit in the cache

- when $MK + KN + MN > C$, the interval doing `gemm(A,B,C)` is two subintervals, each of which does `gemm` for slightly smaller matrices



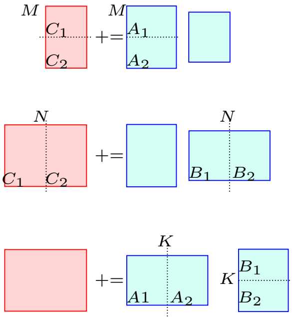
Analyzing cases that do not fit in the cache

- when $MK + KN + MN > C$, the interval doing `gemm(A,B,C)` is two subintervals, each of which does `gemm` for slightly smaller matrices
- in the "nearly square" assumption, the working set becomes $\leq 1/4$ when we divide 3 times



Analyzing cases that do not fit in the cache

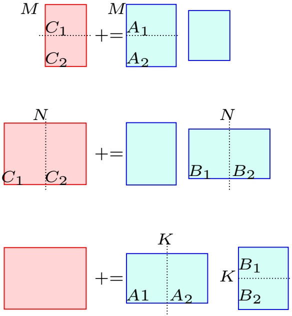
- when $MK + KN + MN > C$, the interval doing `gemm(A,B,C)` is two subintervals, each of which does `gemm` for slightly smaller matrices
- in the "nearly square" assumption, the working set becomes $\leq 1/4$ when we divide 3 times
- to make math simpler, we take it that the working set becomes $\leq \dfrac{1}{\sqrt[3]{4}} (= 2^{-2/3})$ of the original size on each recursion. i.e.,



Analyzing cases that do not fit in the cache

- when $MK + KN + MN > C$, the interval doing `gemm(A,B,C)` is two subintervals, each of which does `gemm` for slightly smaller matrices
- in the "nearly square" assumption, the working set becomes $\leq 1/4$ when we divide 3 times
- to make math simpler, we take it that the working set becomes $\leq \frac{1}{\sqrt[3]{4}} (= 2^{-2/3})$ of the original size on each recursion. i.e.,

$$\therefore R(w) \leq 2R(w/\sqrt[3]{4}) \quad (w > C)$$



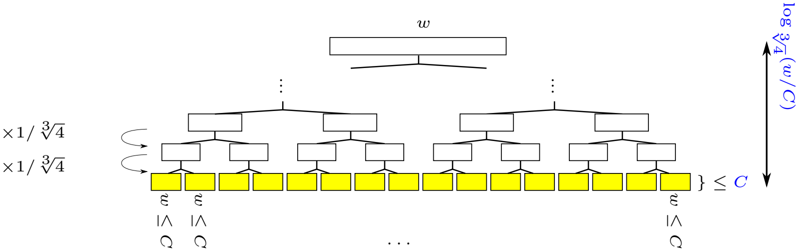
Combined

- we have:

$$R(w) \leq \begin{cases} w & (w \leq C) \\ 2R(w/\sqrt[3]{4}) & (w > C) \end{cases}$$

- when $w > C$, it takes up to $d \approx \log_{\sqrt[3]{4}}(w/C)$ recursion steps until the working set becomes $\leq C$
- the whole computation is essentially 2^d consecutive intervals, each transferring $\leq C$ words

Illustration



$$\begin{aligned} \therefore R(w) &< 2^d \cdot C \\ &= 2^{\log_{\sqrt[3]{4}}(w/C)} \cdot C \\ &= C\left(\frac{w}{C}\right)^{\frac{1}{\log \sqrt[3]{4}}} \\ &= \frac{1}{\sqrt{C}} w^{3/2} \end{aligned}$$

Result

- we have:

$$R(w) \leq \frac{1}{\sqrt{C}} w^{3/2}$$

- for square $(n \times n)$ matrices $(w = 3n^2)$,

$$\therefore R(n) = R(3n^2) = \frac{3\sqrt{3}}{\sqrt{C}} n^3$$

- the same as the blocking we have seen before (not surprising), but we achieved this for all cache levels

A practical remark

- in practice we stop recursion when the matrices become "small enough"

```
gemm(A, B, C) {
  if (A, B, C together fit in the cache) {
    for (i, j, k) ∈ [0..M] × [0..N] × [0..K]
      c_ij += a_ik * b_kj;
  } else if (max(M, N, K) = M) {
    gemm(A_1, B, C_1);
    gemm(A_2, B, C_2);
  } else if (max(M, N, K) = N) {
    gemm(A, B_1, C_1);
    gemm(A, B_1, C_2);
  } else { /* max(M, N, K) = K */
    gemm(A_1, B_1, C);
    gemm(A_2, B_2, C);
  }
}
```

A practical remark

- in practice we stop recursion when the matrices become "small enough"
- but how small is small enough?

```
gemm(A, B, C) {
  if (A, B, C together fit in the cache) {
    for (i, j, k) ∈ [0..M] × [0..N] × [0..K]
      c_ij += a_ik * b_kj;
  } else if (max(M, N, K) = M) {
    gemm(A_1, B, C_1);
    gemm(A_2, B, C_2);
  } else if (max(M, N, K) = N) {
    gemm(A, B_1, C_1);
    gemm(A, B_1, C_2);
  } else { /* max(M, N, K) = K */
    gemm(A_1, B_1, C);
    gemm(A_2, B_2, C);
  }
}
```

A practical remark

- in practice we stop recursion when the matrices become "small enough"
- but how small is small enough?
- when the threshold $\leq$ level x cache, the analysis holds for all levels x and lower

```
gemm(A, B, C) {
  if (A, B, C together fit in the cache) {
    for (i, j, k) ∈ [0..M] × [0..N] × [0..K]
      c_ij += a_ik * b_kj;
  } else if (max(M, N, K) = M) {
    gemm(A_1, B, C_1);
    gemm(A_2, B, C_2);
  } else if (max(M, N, K) = N) {
    gemm(A, B_1, C_1);
    gemm(A, B_1, C_2);
  } else { /* max(M, N, K) = K */
    gemm(A_1, B_1, C);
    gemm(A_2, B_2, C);
  }
}
```

A practical remark

- in practice we stop recursion when the matrices become "small enough"
- but how small is small enough?
- when the threshold ≤ level x cache, the analysis holds for all levels x and lower

```
gemm(A, B, C) {
  if (A, B, C together fit in the cache) {
    for (i, j, k) ∈ [0..M] × [0..N] × [0..K]
      c_ij += a_ik * b_kj;
  } else if (max(M, N, K) = M) {
    gemm(A_1, B, C_1);
    gemm(A_2, B, C_2);
  } else if (max(M, N, K) = N) {
    gemm(A, B_1, C_1);
    gemm(A, B_1, C_2);
  } else { /* max(M, N, K) = K */
    gemm(A_1, B_1, C);
    gemm(A_2, B_2, C);
  }
}
```

- on the other hand, we like to make it large, to reduce control overhead

Contents

Review: (serial) merge sort

```
/* sort a..a_end and put the result into
   (i)  a (if dest = 0)
   (ii) t (if dest = 1) */
void ms(elem * a, elem * a_end,
        elem * t, int dest) {
  long n = a_end - a;
  if (n == 1) {
    if (dest) t[0] = a[0];
  } else {
    /* split the array into two */
    long nh = n / 2;
    elem * c = a + nh;
    /* sort 1st half */
    ms(a, c,     t,      1 - dest);
    /* sort 2nd half */
    ms(c, a_end, t + nh, 1 - dest);
    elem * s = (dest ? a : t);
    elem * d = (dest ? t : a);
    /* merge them */
    merge(s,      s + nh,
          s + nh, s + n, d);
  }
}
```

```
/* merge a_beg ... a_end
     and b_beg ... b_end
    into c */
void
merge(elem * a, elem * a_end,
      elem * b, elem * b_end,
      elem * c) {
  elem * p = a, * q = b, * r = c;
  while (p < a_end && q < b_end) {
    if (*p < *q) { *r++ = *p++; }
    else { *r++ = *q++; }
  }
  while (p < a_end) *r++ = *p++;
  while (q < b_end) *r++ = *q++;
}
```

note: as always, actually switch to serial sort below a threshold (not shown in the code above)

Memory $\leftrightarrow$ cache transfer in merge sort (1) base case

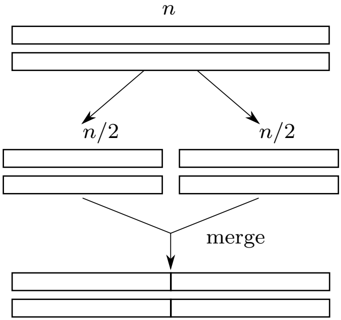
- merge sorting n elements takes *two* arrays of n elements each, and touch all elements of them $\Rightarrow$ *the working set is $2n$ words*
- thus, it fits in the cache when $2n \leq C$

$$\therefore R(n) \leq 2n \quad (2n \leq C)$$

Memory ↔ cache transfer in merge sort (2)

- when $n > C/2$, the whole computation is two recursive calls + merging two results

```
long nh = n / 2;
/* sort 1st half */
ms(a, c,      t,       1 - dest);
/* sort 2nd half */
ms(c, a_end, t + nh, 1 - dest);
   ...
/* merge them */
merge(s,       s + nh,
      s + nh, s + n, d);
```

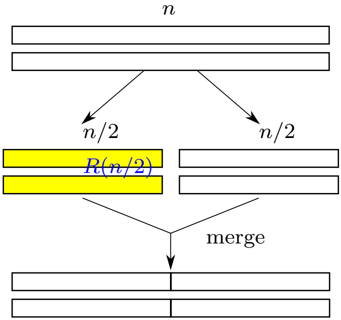


$$\therefore R(n) \leq 2R(n/2) + 2n \quad (n > C/2)$$

Memory ↔ cache transfer in merge sort (2)

- when $n > C/2$, the whole computation is two recursive calls + merging two results

```
long nh = n / 2;
/* sort 1st half */
ms(a, c,      t,       1 - dest);
/* sort 2nd half */
ms(c, a_end, t + nh, 1 - dest);
   ...
/* merge them */
merge(s,      s + nh,
      s + nh, s + n, d);
```

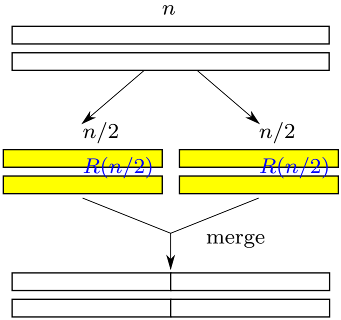


$$\therefore R(n) \leq 2R(n/2) + 2n \quad (n > C/2)$$

Memory $\leftrightarrow$ cache transfer in merge sort (2)

- when $n > C/2$, the whole computation is two recursive calls + merging two results

```
long nh = n / 2;
/* sort 1st half */
ms(a, c,      t,       1 - dest);
/* sort 2nd half */
ms(c, a_end, t + nh, 1 - dest);
   ...
/* merge them */
merge(s,       s + nh,
        s + nh, s + n, d);
```

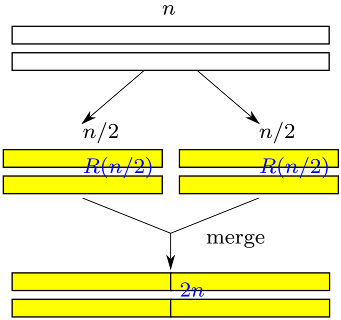


$$\therefore R(n) \leq 2R(n/2) + 2n \quad (n > C/2)$$

Memory ↔ cache transfer in merge sort (2)

- when $n > C/2$, the whole computation is two recursive calls + merging two results

```
long nh = n / 2;
/* sort 1st half */
ms(a, c,      t,      1 - dest);
/* sort 2nd half */
ms(c, a_end, t + nh, 1 - dest);
   ...
/* merge them */
merge(s,      s + nh,
      s + nh, s + n, d);
```



$$\therefore R(n) \leq 2R(n/2) + 2n \quad (n > C/2)$$

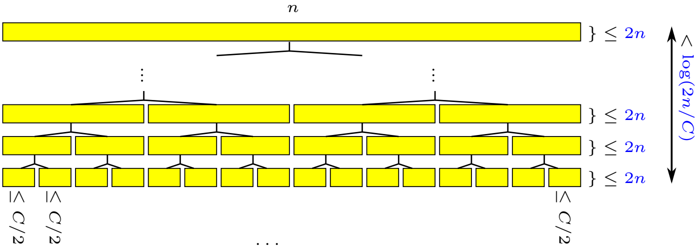
Combined

- so far we have:

$$R(n) \le \begin{cases} 2n & (n \le C/2) \\ 2R(n/2) + 2n & (n > C/2) \end{cases}$$

- for $n > C/2$, it takes at most $d \approx \log \frac{2n}{C}$ divide steps until it becomes $\le C/2$
- thus,

$$R(n) \le 2n \cdot d = 2n \log \frac{2n}{C}$$



Improving merge sort

- so what can we do to improve this?

$$R(n) \leq 2n \log \frac{2n}{C}$$

Improving merge sort

- so what can we do to improve this?

$$R(n) \leq 2n \log \frac{2n}{C}$$

- there are not much we can do to improve a single merge ($\because$ *each element of arrays is accessed only once*)

Improving merge sort

- so what can we do to improve this?

$$R(n) \leq 2n \log \frac{2n}{C}$$

- there are not much we can do to improve a single merge ($\because$ *each element of arrays is accessed only once*)
- so the hope is to reduce the number of steps $\left(\log \frac{2n}{C}\right) \Rightarrow$ *multi-way merge*

Summary

- understanding and assessing data access performance (e.g., cache misses) is important

Summary

- understanding and assessing data access performance (e.g., cache misses) is important
- I hope I have taught that it's a subject of a rigid analysis, *not a black art*

Summary

- understanding and assessing data access performance (e.g., cache misses) is important
- I hope I have taught that it's a subject of a rigid analysis, *not a black art*
- the key for the assessment/analysis is *to identify a unit of computation that fits in the cache*, not to microscopically follow the state of the cache

Summary

- understanding and assessing data access performance (e.g., cache misses) is important
- I hope I have taught that it's a subject of a rigid analysis, *not a black art*
- the key for the assessment/analysis is *to identify a unit of computation that fits in the cache*, not to microscopically follow the state of the cache
- the key to achieve good cache performance is to keep *the compute intensity of cache-fitting computation* high