

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

Kenjiro Taura

Contents

- 1 Introduction
- 2 Analyzing data access complexity of serial programs
 - Overview
 - Model of a machine
 - An analysis methodology
- 3 Applying the methodology to matrix multiply
- 4 Analyzing merge sort

Contents

- 1 Introduction
- 2 Analyzing data access complexity of serial programs
 - Overview
 - Model of a machine
 - An analysis methodology
- 3 Applying the methodology to matrix multiply
- 4 Analyzing merge sort

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

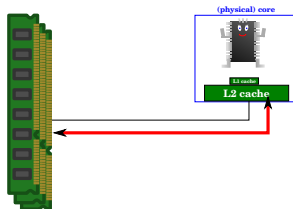
- 1 Introduction
- 2 Analyzing data access complexity of serial programs
 - Overview
 - Model of a machine
 - An analysis methodology
- 3 Applying the methodology to matrix multiply
- 4 Analyzing merge sort

Contents

- 1 Introduction
- 2 Analyzing data access complexity of serial programs
 - Overview
 - Model of a machine
 - An analysis methodology
- 3 Applying the methodology to matrix multiply
- 4 Analyzing merge sort

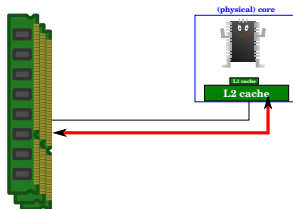
Analyzing data access performance of serial programs

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



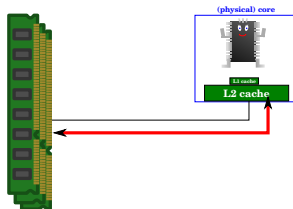
Analyzing data access performance of serial programs

- 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*



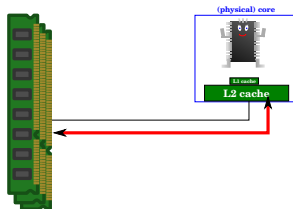
Analyzing data access performance of serial programs

- 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

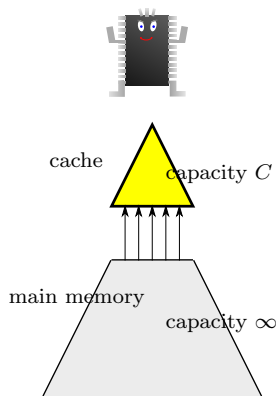


Contents

- 1 Introduction
- 2 Analyzing data access complexity of serial programs
 - Overview
 - Model of a machine
 - An analysis methodology
- 3 Applying the methodology to matrix multiply
- 4 Analyzing merge sort

Model (of a single processor machine)

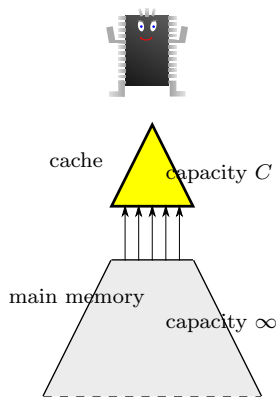
- a machine consists of
 - a processor,
 - a fixed size cache (C words), and
 - an arbitrarily large main memory
- accessing a word not in the cache mandates transferring the word into the cache



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

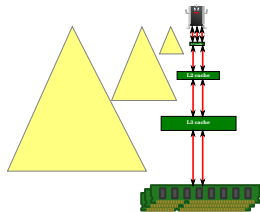


For now, we assume a single processor machine.

Gaps between our model and real machines

- hierarchical caches:

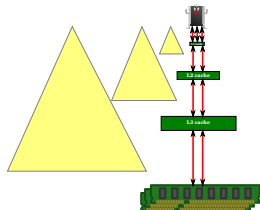
⇒ each level can be easily modeled separately, with caches of various sizes



Gaps between our model and real machines

- hierarchical caches:

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



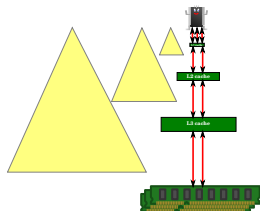
- concurrent accesses:

- 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:

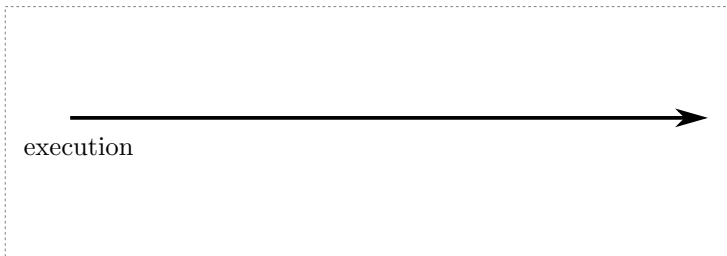
- similarly, this model cannot capture the difference between sequential accesses that can take advantages of the hardware prefetcher and random accesses that cannot

Contents

- 1 Introduction
- 2 Analyzing data access complexity of serial programs
 - Overview
 - Model of a machine
 - An analysis methodology
- 3 Applying the methodology to matrix multiply
- 4 Analyzing merge sort

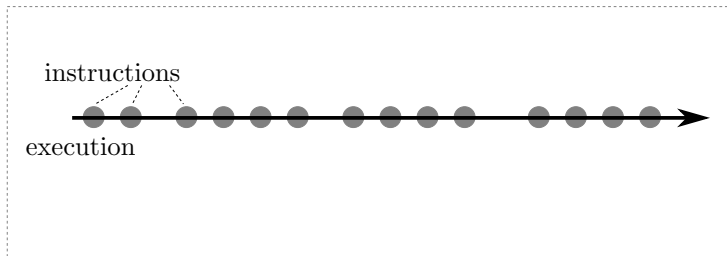
Terminologies

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



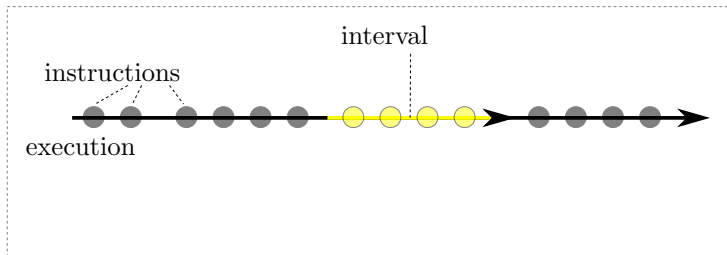
Terminologies

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



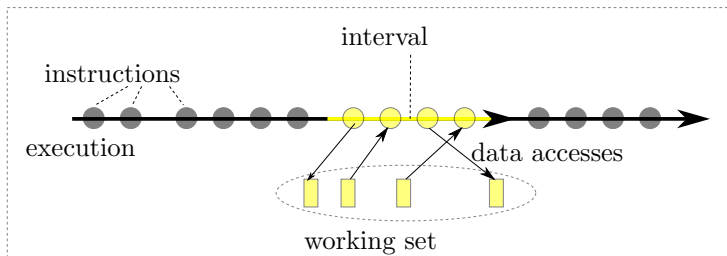
Terminologies

- 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

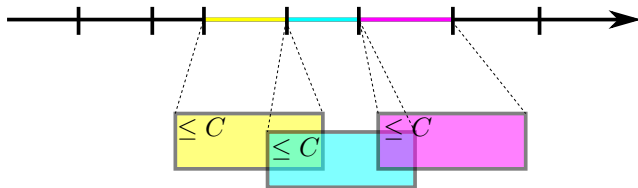


A basic methodology

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.,

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



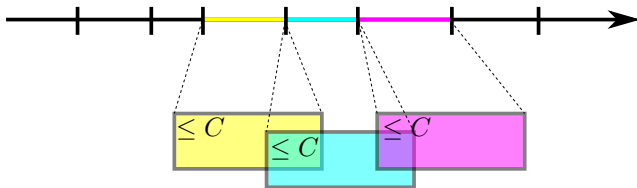
A basic methodology

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.,

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

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



A basic methodology

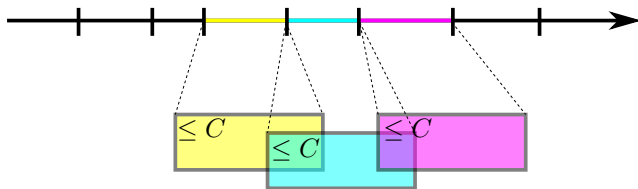
to calculate the amount of data transferred in an execution,

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

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

- 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

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

Remarks

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

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

- 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

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

- 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

- 1 Introduction
- 2 Analyzing data access complexity of serial programs
 - Overview
 - Model of a machine
 - An analysis methodology
- 3 Applying the methodology to matrix multiply
- 4 Analyzing merge sort

Applying the methodology

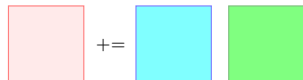
- we will apply the methodology to some of the algorithms we have studied so far
- the key is to *find subproblems (intervals) that fit in the cache*

Analyzing the triply nested loop

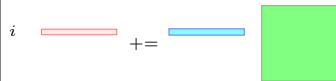
```
1 for (i = 0; i < n; i++) {  
2   for (j = 0; j < n; j++) {  
3     for (k = 0; k < n; k++) {  
4       C(i,j) += A(i,k) * B(k,j);  
5     }  
6   }  
7 }
```

- key question: *which iteration fits in the cache?*

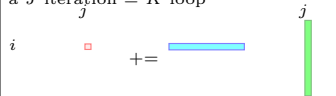
I loop (= entire computation)



an *I* iteration = *J* loop



a *J* iteration = *K* loop

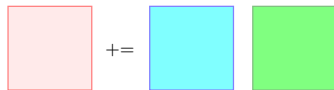


Working sets

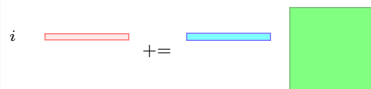
```
for (i = 0; i < n; i++) {  
  for (j = 0; j < n; j++) {  
    for (k = 0; k < n; k++) {  
      C(i,j) += A(i,k) * B(k,j);  
    }  
  }  
}
```

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

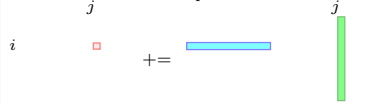
I loop (= entire computation)



an I iteration = J loop

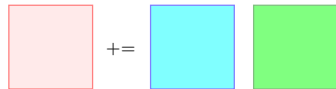


a J iteration = K loop

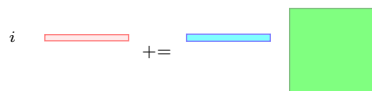


Cases to consider

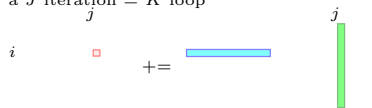
I loop (= entire computation)



an I iteration = J loop



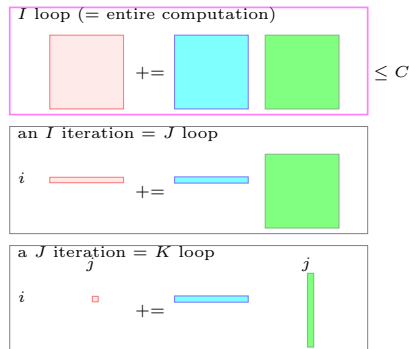
a J iteration = K loop



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

Cases to consider

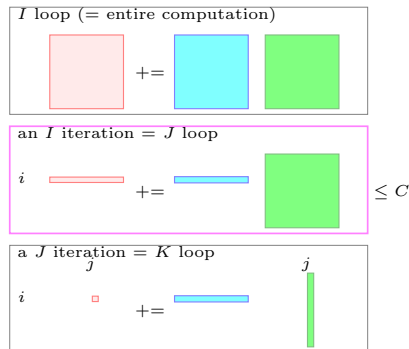
- Case 1: the three matrices all fit in the cache ($3n^2 \leq C$)



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

Cases to consider

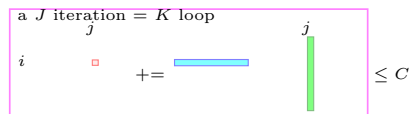
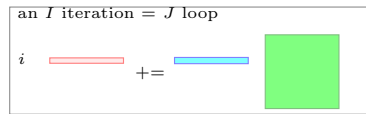
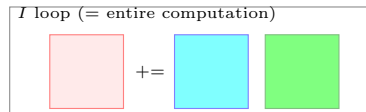
- Case 1: the three matrices all fit in the cache ($3n^2 \leq C$)
- Case 2: a single i iteration (\approx a matrix) fits in the cache ($2n + n^2 \leq C$)



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

Cases to consider

- Case 1: the three matrices all fit in the cache ($3n^2 \leq C$)
- Case 2: a single i iteration (\approx a matrix) fits in the cache ($2n + n^2 \leq C$)
- Case 3: a single j iteration (\approx two vectors) fits in the cache ($1 + 2n \leq C$)

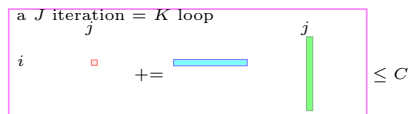
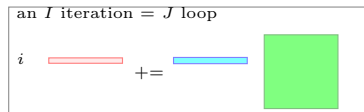
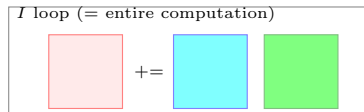


Goal: for each case, bound from the above $R(n)$, the traffic between memory and cache for $n \times n$ matrix-multiply

Cases to consider

- Case 1: the three matrices all fit in the cache ($3n^2 \leq C$)
- Case 2: a single i iteration (\approx a matrix) fits in the cache ($2n + n^2 \leq C$)
- Case 3: a single j iteration (\approx two vectors) fits in the cache ($1 + 2n \leq C$)
- Case 4: none of the above ($1 + 2n > C$)

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

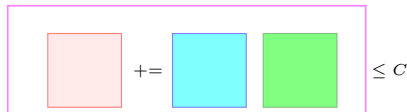


Case 1 ($3n^2 \leq C$)

- 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

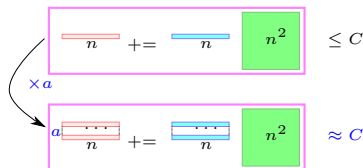


Case 2 ($2n + n^2 \leq C$)

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

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

- each such set of iterations transfer $\leq C$ words, so

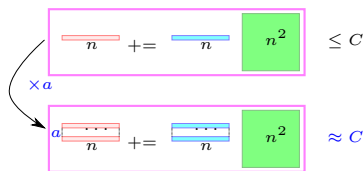


$$R(n) \leq \frac{n}{a}C = \frac{n}{a}(n^2 + 2an) = \left(\frac{1}{a} + \frac{2}{n}\right)n^3$$

interpretation: each element of B is reused a times in the cache; each element in A or C many ($\propto n$) 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 each i -iteration, B stays in the cache for any number of a iterations
- so, in this case too,



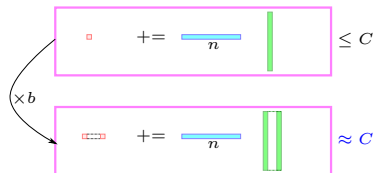
$$R(n) \leq 3n^2 = \frac{3}{n} \cdot n^3$$

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

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

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

- each such set of iterations transfer $\leq C$ words, so



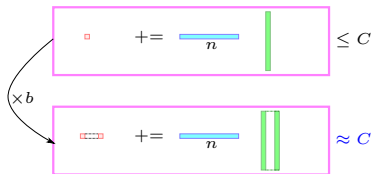
$$R(n) \leq \frac{n^2}{b}C = \frac{n^2}{b}(n + b(n+1)) = \left(\frac{1}{b} + 1 + \frac{1}{n}\right)n^3$$

interpretation: each element in B is never reused; each element in A b times; each element 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
- therefore

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



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

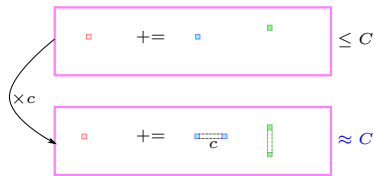
- the maximum number of k -iterations that fit in the cache is:

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

- each such set of iterations transfer $\leq C$ words, so

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

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

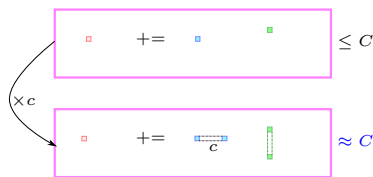


A Remark

- similarly, the element of C stays in the cache for any number of c iterations
- so

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

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
$1 + 2n > C$	$n^2 + 2n^3$	$\approx 1/2$

So how to improve it?

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

So how to improve it?

- in general, the traffic increases when *the same amount of computation has a large working set*
- to reduce the traffic, you arrange the computation (order subcomputations) so that *you do a lot of computation on the same amount of data*

So how to improve it?

- in general, the traffic increases when *the same amount of computation has a large working set*
- to reduce the traffic, you arrange the computation (order subcomputations) so that *you do 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

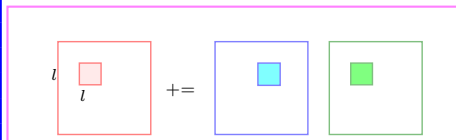
So how to improve it?

- in general, the traffic increases when *the same amount of computation has a large working set*
- to reduce the traffic, you arrange the computation (order subcomputations) so that *you do 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 =  $\sqrt{C}$  - 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,

```
1  l =  $\sqrt{C}$  - small constant;  
2  for (ii = 0; ii < n; ii += l)  
3      for (jj = 0; jj < n; jj += l)  
4          for (kk = 0; kk < n; kk += l)  
5              /* block below misses each */  
6              /* element at most once */  
7              for (i = ii; i < ii + l; i++)  
8                  for (j = jj; j < jj + l; j++)  
9                      for (k = kk; k < kk + l; k++)  
10                         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

$$\leq 3l^2 \cdot \left(\frac{n}{l}\right)^3 = \frac{3}{l}n^3 = \frac{3}{\sqrt{C}}n^{3^{10}}$$

```
1  l =  $\sqrt{C}$  - small constant;  
2  for (ii = 0; ii < n; ii += l)  
3    for (jj = 0; jj < n; jj += l)  
4      for (kk = 0; kk < n; kk += l)  
5        /* block below misses each */  
6        /* element at most once */  
7        for (i = ii; i < ii + l; i++)  
8          for (j = jj; j < jj + l; j++)  
9            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	$\frac{3}{\sqrt{C}}n^3$	$\frac{\sqrt{C}}{3}$

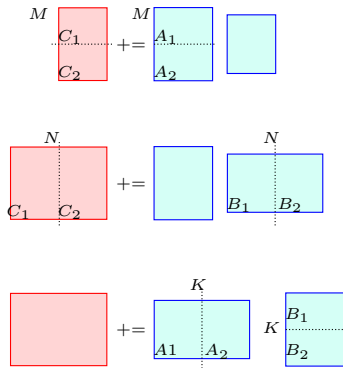
- assume a word = 4 bytes (**float**)

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



```
1  gemm(A, B, C) {  
2    if ((M, N, K) = (1, 1, 1)) {  
3       $c_{11} += a_{11} * b_{11};$   
4    } else if (max(M, N, K) = M) {  
5      gemm( $A_1$ , B,  $C_1$ );  
6      gemm( $A_2$ , B,  $C_2$ );  
7    } else if (max(M, N, K) = N) {  
8      gemm(A,  $B_1$ ,  $C_1$ );  
9      gemm(A,  $B_1$ ,  $C_2$ );  
10   } else { /* max(M, N, K) = K */  
11     gemm( $A_1$ ,  $B_1$ , C);  
12     gemm( $A_2$ ,  $B_2$ , C);  
13   }  
14 }
```

- 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:

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

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

- to avoid making analysis tedious, assume all matrices are “**nearly square**”

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

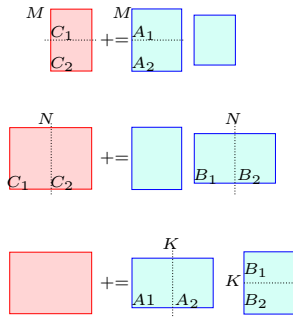
The largest subproblem that fits in the cache

- 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:

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

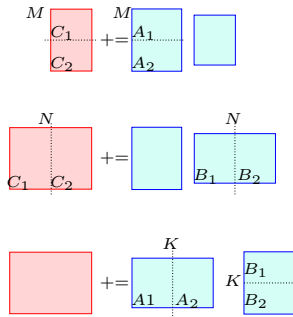
Analyzing cases that do not fit in the cache

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



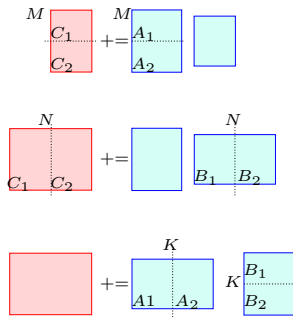
Analyzing cases that do not fit in the cache

- 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

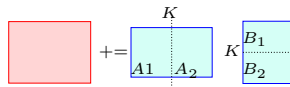
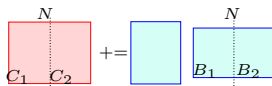
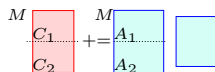
- when $MK + KN + MN > C$, the interval doing `gemm(A,B,C)` is two subintervals, each of which does `gemm` for slightly smaller matrices
- in the “nearly square” assumption, the working set becomes $\leq 1/4$ when we divide 3 times
- to make math simpler, we take it that the working set becomes $\leq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

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

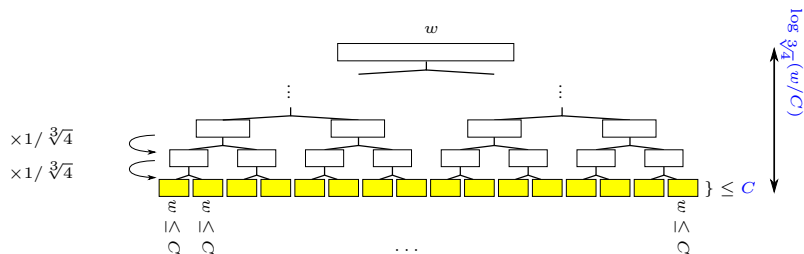


- we have:

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

- 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”

```
1  gemm(A, B, C) {  
2    if (A, B, C together fit in the cache) {  
3      for (i, j, k) ∈ [0..M] × [0..N] × [0..K]  
4        cij += aik * bkj;  
5    } else if (max(M, N, K) = M) {  
6      gemm(A1, B, C1);  
7      gemm(A2, B, C2);  
8    } else if (max(M, N, K) = N) {  
9      gemm(A, B1, C1);  
10     gemm(A, B1, C2);  
11   } else { /* max(M, N, K) = K */  
12     gemm(A1, B1, C);  
13     gemm(A2, B2, C);  
14   }  
15 }
```

A practical remark

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

```
1  gemm(A, B, C) {  
2    if (A, B, C together fit in the cache) {  
3      for (i, j, k) ∈ [0..M] × [0..N] × [0..K]  
4        cij += aik * bkj;  
5    } else if (max(M, N, K) = M) {  
6      gemm(A1, B, C1);  
7      gemm(A2, B, C2);  
8    } else if (max(M, N, K) = N) {  
9      gemm(A, B1, C1);  
10     gemm(A, B2, C2);  
11   } else { /* max(M, N, K) = K */  
12     gemm(A1, B1, C);  
13     gemm(A2, B2, C);  
14   }  
15 }
```

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

```
1  gemm(A, B, C) {  
2    if (A, B, C together fit in the cache) {  
3      for (i, j, k)  $\in [0..M] \times [0..N] \times [0..K]$   
4         $c_{ij} += a_{ik} * b_{kj};$   
5    } else if (max(M, N, K) = M) {  
6      gemm(A1, B, C1);  
7      gemm(A2, B, C2);  
8    } else if (max(M, N, K) = N) {  
9      gemm(A, B1, C1);  
10     gemm(A, B1, C2);  
11   } else { /* max(M, N, K) = K */  
12     gemm(A1, B1, C);  
13     gemm(A2, B2, C);  
14   }  
15 }
```


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

```
1  gemm(A, B, C) {  
2    if (A, B, C together fit in the cache) {  
3      for (i, j, k)  $\in [0..M] \times [0..N] \times [0..K]$   
4         $c_{ij} += a_{ik} * b_{kj}$ ;  
5    } else if (max(M, N, K) = M) {  
6      gemm(A1, B, C1);  
7      gemm(A2, B, C2);  
8    } else if (max(M, N, K) = N) {  
9      gemm(A, B1, C1);  
10     gemm(A, B1, C2);  
11   } else { /* max(M, N, K) = K */  
12     gemm(A1, B1, C);  
13     gemm(A2, B2, C);  
14   }  
15 }
```

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

Contents

- 1 Introduction
- 2 Analyzing data access complexity of serial programs
 - Overview
 - Model of a machine
 - An analysis methodology
- 3 Applying the methodology to matrix multiply
- 4 Analyzing merge sort

Review: (serial) merge sort

```
1  /* sort a..a_end and put the result into
2     (i) a (if dest = 0)
3     (ii) t (if dest = 1) */
4  void ms(elem * a, elem * a_end,
5          elem * t, int dest) {
6      long n = a_end - a;
7      if (n == 1) {
8          if (dest) t[0] = a[0];
9      } else {
10         /* split the array into two */
11         long nh = n / 2;
12         elem * c = a + nh;
13         /* sort 1st half */
14         ms(a, c, t, 1 - dest);
15         /* sort 2nd half */
16         ms(c, a_end, t + nh, 1 - dest);
17         elem * s = (dest ? a : t);
18         elem * d = (dest ? t : a);
19         /* merge them */
20         merge(s, s + nh,
21              s + nh, s + n, d);
22     }
23 }
```

```
1  /* merge a_beg ... a_end
2     and b_beg ... b_end
3     into c */
4  void
5  merge(elem * a, elem * a_end,
6        elem * b, elem * b_end,
7        elem * c) {
8      elem * p = a, * q = b, * r = c;
9      while (p < a_end && q < b_end) {
10         if (*p < *q) { *r++ = *p++; }
11         else { *r++ = *q++; }
12     }
13     while (p < a_end) *r++ = *p++;
14     while (q < b_end) *r++ = *q++;
15 }
```

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

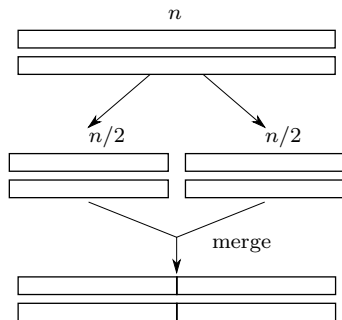
- merge sorting n elements takes *two* arrays of n elements each, and touch all elements of them \Rightarrow *the working set is $2n$ words*
- thus, it fits in the cache when $2n \leq C$

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

Memory \leftrightarrow cache transfer in merge sort (2)

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

```
1 long nh = n / 2;  
2 /* sort 1st half */  
3 ms(a, c, t, 1 - dest);  
4 /* sort 2nd half */  
5 ms(c, a_end, t + nh, 1 - dest);  
6 ...  
7 /* merge them */  
8 merge(s, s + nh,  
9       s + nh, s + n, d);
```

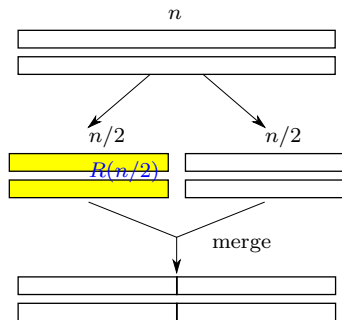


$$\therefore R(n) \leq 2R(n/2) + 2n \quad (n > C/2)$$

Memory \leftrightarrow cache transfer in merge sort (2)

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

```
1 long nh = n / 2;  
2 /* sort 1st half */  
3 ms(a, c, t, 1 - dest);  
4 /* sort 2nd half */  
5 ms(c, a_end, t + nh, 1 - dest);  
6 ...  
7 /* merge them */  
8 merge(s, s + nh,  
9       s + nh, s + n, d);
```

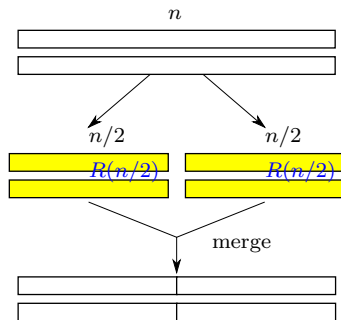


$$\therefore R(n) \leq 2R(n/2) + 2n \quad (n > C/2)$$

Memory \leftrightarrow cache transfer in merge sort (2)

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

```
1 long nh = n / 2;  
2 /* sort 1st half */  
3 ms(a, c, t, 1 - dest);  
4 /* sort 2nd half */  
5 ms(c, a_end, t + nh, 1 - dest);  
6 ...  
7 /* merge them */  
8 merge(s, s + nh,  
9       s + nh, s + n, d);
```

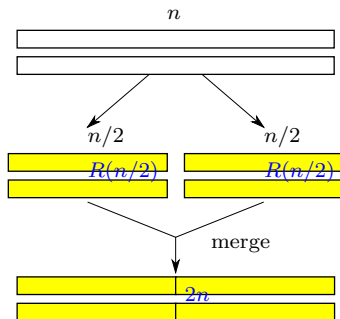


$$\therefore R(n) \leq 2R(n/2) + 2n \quad (n > C/2)$$

Memory \leftrightarrow cache transfer in merge sort (2)

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

```
1 long nh = n / 2;  
2 /* sort 1st half */  
3 ms(a, c, t, 1 - dest);  
4 /* sort 2nd half */  
5 ms(c, a_end, t + nh, 1 - dest);  
6 ...  
7 /* merge them */  
8 merge(s, s + nh,  
9       s + nh, s + n, d);
```



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

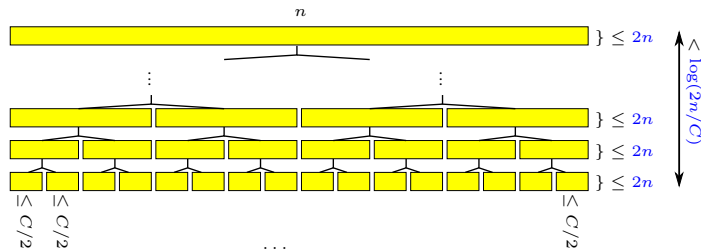
Combined

- so far we have:

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

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

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



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 $(\log \frac{2n}{C}) \Rightarrow$ *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

