Applied Algorithms Lec 6: External Memory and Optimization

Sam McCauley

October 21, 2021

Williams College

Admin

- Office hours changed:
- Sam: Mon 2:30–4, 5–6:30; Wed 2–4
- Chris: Tue 3–5, Wed 8–10
- Updated on website

Questions about Assignment 2?

Matrix Multiplication in External Memory

Compute Product Directly

```
for i = 1 to n:
  for j = 1 to n:
    for k = 1 to n:
      C[i][j] += A[i][k] +
          B[k][j]
```

- Recall: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
- How many I/Os?
- Assume matrices are stored in row-major order.
  - First: assume $3n^2 < M$
  - After $O(n^2/B)$ I/Os, all three matrices are in memory, and don't have any more I/Os.
  - What if $nB > M$?
  - Answer: $O(n^3)$ I/Os. Every inner loop operation requires an I/O for B.

Any ideas for how to improve this?

- One idea: transpose B.
- Another idea: swap the loops!
- `-O3` optimization of `gcc` actually tries to do this automatically (Very cool)

```
for i = 1 to n:
  for k = 1 to n:
    for j = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- This gives us $O(n^3/B)$ I/Os: (assume $B < n$ to make things easier)

Any ideas for how to improve this?

```
for i = 1 to n:
  for k = 1 to n:
    for j = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- This gives us $O(n^3/B)$ I/Os: (assume $B < n$ to make things easier)
- Let's say $A[i][k]$ is a cache miss. No more cache misses until $A[i][k']$ with $k' = k + B$.
- Let's say $B[k][j]$ is a cache miss. No more cache misses until $B[i][j']$ with $j' = j + B$.
- Let's say $C[i][j]$ is a cache miss. No more cache misses until $C[i][j']$ with $j' = j + B$.
- Sum up each

Improvement in practice

I am given two functions for finding the product of two matrices:

```
void MultiplyMatrices_1(int **a, int **b, int **c, int n){
     for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
             for (int k = 0; k < n; k++)
                 c[i][j] = c[i][j] + a[i][k]*b[k][j];
 }

void MultiplyMatrices_2(int **a, int **b, int **c, int n){
     for (int i = 0; i < n; i++)
         for (int k = 0; k < n; k++)
             for (int j = 0; j < n; j++)
                 c[i][j] = c[i][j] + a[i][k]*b[k][j];
}
```

I ran and profiled two executables using `gprof`, each with identical code except for this function. The second of these is significantly (about 5 times) faster for matrices of size 2048 x 2048. Any ideas as to why?

c algorithm matrix matrix-multiplication gprof

share improve this question

edited Sep 13 '11 at 2:47
templatetypedef
295k • 80 • 725 • 933

asked Sep 13 '11 at 0:29
kevlar1818
2,639 • 4 • 19 • 39

add a comment

nswers

active oldest votes

We haven't used the cache yet

- No Ms in any running times—except when the whole problem fits in cache
- Why? All algorithms so far have read the data once and then thrown it away.
- Goal: bring items into cache so that we can perform *many* computations on them before writing them back.
- Note: can't do this with linear scan. $O(n/B)$ is optimal.

Blocking

- Standard technique for improving cache performance of algorithms.
- Remember from before: cache efficiency can get WAY better when the problem fits in cache. Let's find subproblems that can fit in cache.
- Idea: break problems into subproblems of size $O(M)$
  - Can solve in $O(M/B)$ I/Os
  - Efficiently combine them for a cache-efficient solution

Blocked Matrix Multiplication

- Split A, B, and C into blocks of size $M/3$
  - $\sqrt{M/3} \times \sqrt{M/3}$-sized blocks
  - Let's say the number of rows and columns in our blocks is (each) $T = \lfloor \sqrt{M/3} \rfloor$. Assume that T divides n for now.
- Multiply blocks one at a time
- Need some structure to help us make this work

Decomposing matrices into blocks

Classic result: if we treat the blocks as single elements of the matrices, and multiply (and add) them as normal, we obtain the same result as we would have in normal matrix multiplication.

- This idea is used in recursive matrix multiplication
- And Strassen's algorithm for matrix multiplication

Decomposing matrices into blocks

Example: Recall how to multiply 2x2 matrices:

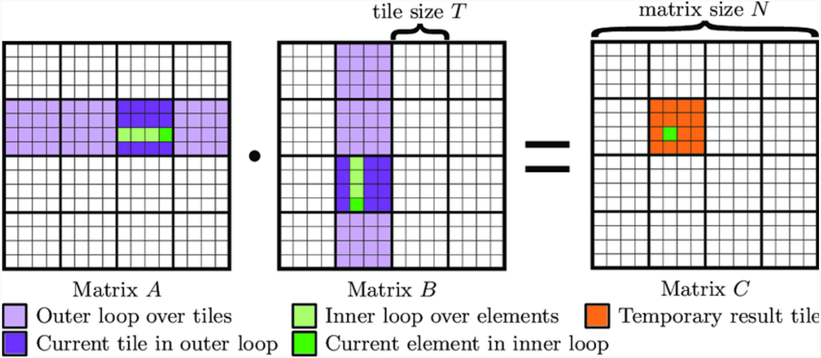
$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}$$

$$\begin{bmatrix} 17 & 15 & 20 & 4 \\ 15 & 3 & 20 & 8 \\ 1 & 10 & 15 & 2 \\ 3 & 19 & 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 & 9 & 1 \\ 4 & 6 & 11 & 2 \\ 13 & 18 & 8 & 20 \\ 3 & 11 & 18 & 9 \end{bmatrix} =$$

$$\begin{bmatrix} \begin{bmatrix} 17 & 15 \\ 15 & 3 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 20 & 4 \\ 20 & 8 \end{bmatrix} \cdot \begin{bmatrix} 13 & 8 \\ 3 & 11 \end{bmatrix} & \begin{bmatrix} 17 & 15 \\ 15 & 3 \end{bmatrix} \cdot \begin{bmatrix} 9 & 1 \\ 11 & 2 \end{bmatrix} + \begin{bmatrix} 20 & 4 \\ 20 & 8 \end{bmatrix} \cdot \begin{bmatrix} 8 & 20 \\ 18 & 9 \end{bmatrix} \\ \begin{bmatrix} 1 & 10 \\ 3 & 19 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 15 & 2 \\ 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 13 & 8 \\ 3 & 11 \end{bmatrix} & \begin{bmatrix} 1 & 10 \\ 3 & 19 \end{bmatrix} \cdot \begin{bmatrix} 9 & 1 \\ 11 & 2 \end{bmatrix} + \begin{bmatrix} 15 & 2 \\ 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 8 & 20 \\ 18 & 9 \end{bmatrix} \end{bmatrix}$$

Blocked Matrix Multiplication

- Decompose matrix into blocks of length T (recall that $T^2 \leq M/3$)
- Do a normal $n/T \times n/T$ matrix multiplication



Blocked Matrix Multiplication Pseudocode

```
MatrixMultiply(A, B, C, n, T):
   for i = 1 to n/T:
     for k = 1 to n/T:
       for j = 1 to n/T:
        A’ = TxT matrix with upper left corner A[Ti][Tk]
        B’ = TxT matrix with upper left corner B[Tk][Tj]
        C’ = TxT matrix with upper left corner C[Ti][Tj]
        BlockMultiply(A’, B’, C’, T)

BlockMultiply(A, B, C, n):
   for i = 1 to n:
      for k = 1 to n:
          for j = 1 to n:
            C[i][j] += A[i][k] + B[k][j]
```

Let’s analyze the cost of this algorithm in the EM model

Analysis

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses. If $B^2 = O(M)$ then we can simplify this to $O(M/B)$. (Called the "tall cache assumption.")
- `BlockMultiply` only accesses elements of A', B', C'. Since all three matrices are in cache, it requires zero additional cache misses
- Therefore, our total running time is the number of loop iterations times the cost of a loop. This is $O((n/T)^3 \cdot (T^2/B)) = O((n/\sqrt{M})^3 \cdot (M/B)) = O(n^3/B\sqrt{M})$.

Implementation questions!

- What do we do if n is not divisible by T?
  - Easy answer: pad it out! Doesn't change asymptotics.
  - Can carefully make it work without padding as well
- How do we figure out M? We don't have a two-level cache and we're ignoring that space is used for other programs, other variables, etc.
  - Experiment! Try different values of M and see what's fastest on a particular machine.
- Is blocking actually worthwhile?
  - Yes; it is used all the time to speed up programs with poor cache performance.
  - (Not a panacea; some programs (like linear scan, binary search) can't be blocked.)

Sorting in External Memory

What about algorithms we know?

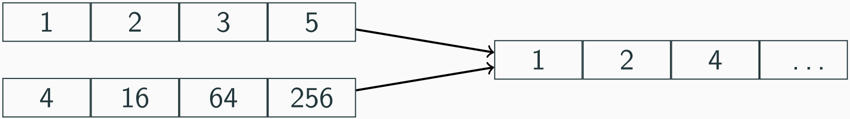
- How long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $\frac{n}{B} \log_2 \frac{n}{B}$.
- How about quicksort?
- Essentially same; partition is $O(n/B)$; total is $\frac{n}{B} \log_2 \frac{n}{B}$.
- Heapsort is $n \log_2 n/B$ unless we're careful...
- Can we do better?

Using the cache

- Blocking? A little unclear. (We'll come back to this.)
- Does anyone know the sorting lower bound? Where does $n \log n$ come from?
- Answer: each time you compare two numbers, can only have two outcomes.
- Each time we bring a cache line into cache, how many more things can we compare it to?

Merge sort reminder

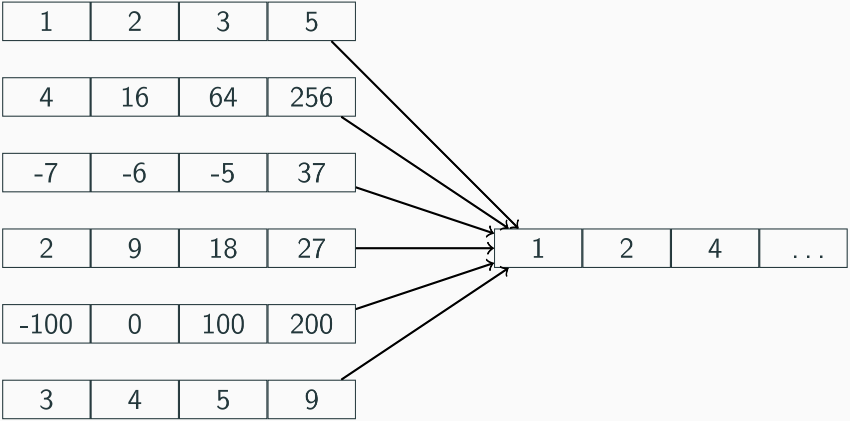
- Divide array into two equal parts
- Recursively sort both parts
- Merge them in $O(n)$ time (and $O(n/B)$ cache misses)



M/B-way merge sort

- Divide array into M/B equal parts
- Recursively sort all M/B parts
- Merge all M/B arrays in $O(n)$ time (and $O(n/B)$ cache misses)

Diagram of M/B-way merge sort



More Detail on merges

- Keep B slots for each array in cache. (M/B arrays so this fits!)
- When all B slots are empty for the array, take B more items from the array in cache.
- Example on board

Analysis

- Divide array into M/B parts; combine in $O(N/B)$ cache misses.
- Recursion:

$$T(N) = \frac{M}{B} T(N/(M/B)) + O(N/B)$$

$$T(B) = O(1)$$

- Solves to $O(\frac{n}{B} \log_{M/B} n/B)$ cache misses
- Optimal!

Useful?

- Can be useful if your data is VERY large
- Distribution sort: similar idea, but with Quicksort instead of Mergesort
- Another method is most popular in practice: Timsort

Timsort

- We won't go over in detail
- Idea: one cache-efficient pass over the array using $O(n/B)$ cache misses that tries to sort things as much as possible
- Then, a super optimized merge sort
- Used in Python, Java, Rust, Android

External Memory Sorting

- M/B way merge sort is most efficient
- Timsort is very popular in practice; uses a simpler blocking approach to stay cache-friendly.

Optimization (And Assignment 1 Review)

Plan for this topic

- First, talk about how various techniques can make code more efficient
    - ...or less efficient
- Focus on loops, and on compiler options
- Then, look back a bit at Assignment 1. Talk about various strategies, and what some final products looked like
    - May continue this a bit Thursday if we run out of time

Taking out expensive operations

```
for(int i = 0; i < strlen(str1); i++){
    str1[i] = 'a';
}
```

- What's wrong with this code? How long does it take?
- Does the compiler optimize this out?

- It can't: we're changing the array, which could change its length. (Of course, we know that we're never setting any values to 0, but the compiler doesn't check for that.)

More subtle issues

```c
int len = strlen(str1);
for(int i=0; i < len; i++){
   str1[i] = str1[0];
}
```

```c
int len = strlen(str1);
int start = str1[0];
for(int i=0; i < len; i++){
   str1[i] = start;
}
```

- Version on the right runs 2-3x faster even with optimizations on
- Why is that?
- Don't need to look up value! (Compiler doesn't know it doesn't change after the first iteration)