Lecture 22: Cache-Oblivious Algorithms

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

November 30, 2020

Overview

- Administrivia
- Designing Algorithms in Real Life
- Cache-Aware and Cache-Oblivious Algorithms
- Conclusion
- Acknowledgements

Rate this course!

Please log in to

https://evaluate.uwaterloo.ca/

from *November 24th until December 7th* and provide us with your evaluation and feedback on the course!

- This would really help me figuring out what worked and what didn't for the course
- And whether I should put memes or gifs into my slides...
- Teaching this course is also a learning experience for me :)

Research Opportunities at UW!

Consider doing a URA, URF or USRA with a U Waterloo faculty!
See research openings at:

- Undergraduate Research Assistanship (URA):

  `https://cs.uwaterloo.ca/computer-science/current-undergraduate-students/research-opportunities/undergraduate-research-assistantship-ura-program`

- Undergraduate Research Fellowship (URF):

  `https://grec.cs.uwaterloo.ca/`

- Undergraduate Research Internship (URI):

  `https://cs.uwaterloo.ca/current-undergraduate-students/research-opportunities/undergraduate-research-internship-uri-program`

- For Canadians, please check out NSERC's USRA:

  `https://cs.uwaterloo.ca/usra`

Memory Hierarchy

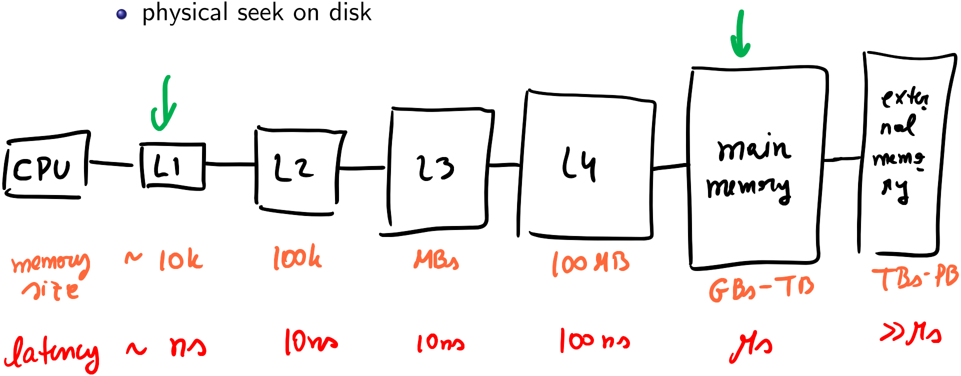
- Not all memory created equal

Memory Hierarchy

- Not all memory created equal
- In real life, need to account for our use of memory - it can be the bottleneck!

Memory Hierarchy

- Not all memory created equal
- In real life, need to account for our use of memory - it can be the bottleneck!
- Bigger memory usually have slower latency
  - distance travel to cache
  - physical seek on disk



Memory Hierarchy

- Not all memory created equal
- In real life, need to account for our use of memory - it can be the bottleneck!
- Bigger memory usually have slower latency
  - distance travel to cache
  - physical seek on disk
- Bandwidth (rate of transmission) is usually matched across memories

Memory Hierarchy

- Not all memory created equal
- In real life, need to account for our use of memory - it can be the bottleneck!
- Bigger memory usually have slower latency
  - distance travel to cache
  - physical seek on disk
- Bandwidth (rate of transmission) is usually matched across memories
- Memory architects *group* (or *block*) memory to decrease latency!
  - when fetching word of data, get entire *line/block* containing it
  - amortize latency over the whole line of memory

$$\frac{\text{latency}}{\text{block size}} + \frac{1}{\text{bandwidth}}$$

$\uparrow$ $\nearrow$

set these to be equal

(adjusting block size usually)

Memory Hierarchy

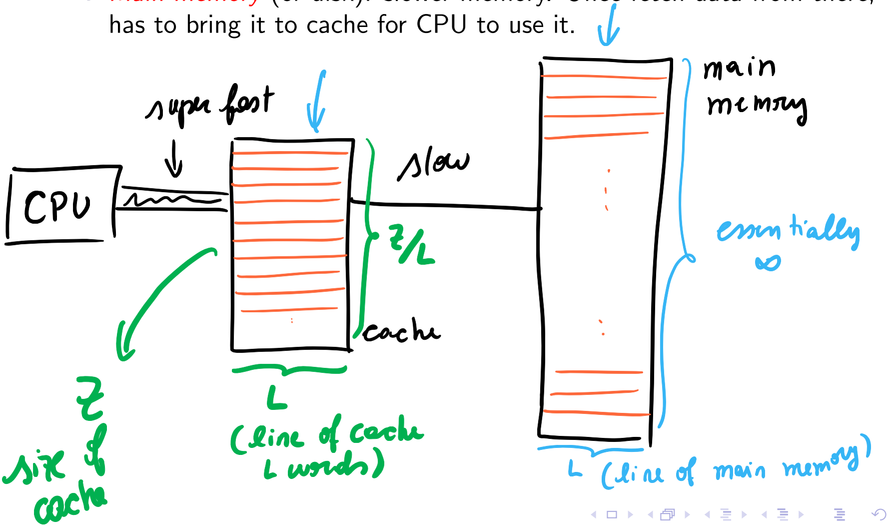
- Not all memory created equal
- In real life, need to account for our use of memory - it can be the bottleneck!
- Bigger memory usually have slower latency
  - distance travel to cache
  - physical seek on disk
- Bandwidth (rate of transmission) is usually matched across memories
- Memory architects *group* (or *block*) memory to decrease latency!
  - when fetching word of data, get entire *line/block* containing it
  - amortize latency over the whole line of memory

$$\frac{\text{latency}}{\text{block size}} + \frac{1}{\text{bandwidth}}$$

- Good algorithms must have:
  - *Spatial locality*: use all elements in same memory line
  - *Temporal locality*: re-use lines in cache before moving on

External Memory Model [Aggarwal and Vitter 1988]

- For simplicity, just 2 levels:
    - *Cache*: the fast memory (CPU has instant access to any word in it)
    - *Main memory* (or disk): slower memory. Once fetch data from there, has to bring it to cache for CPU to use it.



External Memory Model [Aggarwal and Vitter 1988]

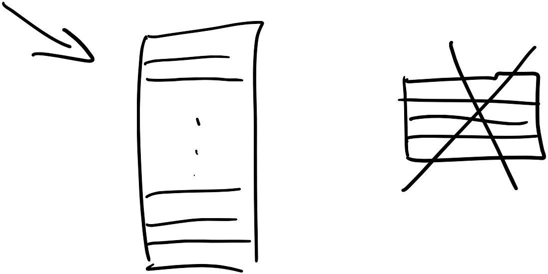
- For simplicity, just 2 levels:
  - *Cache*: the fast memory (CPU has instant access to any word in it)
  - *Main memory* (or disk): slower memory. Once fetch data from there, has to bring it to cache for CPU to use it.
- Algorithm has to read and write lines of memory!

External Memory Model [Aggarwal and Vitter 1988]

- For simplicity, just 2 levels:
  - *Cache*: the fast memory (CPU has instant access to any word in it)
  - *Main memory* (or disk): slower memory. Once fetch data from there, has to bring it to cache for CPU to use it.
- Algorithm has to read and write lines of memory!
- Complexity Measures:
  - *Work*: number of *computational* steps needed from start to end.
  - *Cache Misses*: number of cache misses (memory transfers from disk to cache) during entire computation

Cache-Oblivious Model [Frigo et al 1999]

- Ideal cache model:
  - word size known to everyone in advance
  - two-level memory
  - Memory partitioned in lines of L words each
  - Cache size is Z
  - *Tall Cache Assumption*: $Z = \Omega(L^2)$
  - Eviction algorithm: *farthest in the future*[1]



---

[1] Can be replaced by LRU efficiently.

Cache-Oblivious Model [Frigo et al 1999]

- Ideal cache model:
    - word size known to everyone in advance
    - two-level memory
    - Memory partitioned in lines of L words each
    - Cache size is Z
    - *Tall Cache Assumption*: $Z = \Omega(L^2)$
    - Eviction algorithm: *farthest in the future*[1]
- *Cache-Oblivious Algorithm:* doesn't know Z, L

---

[1] Can be replaced by LRU efficiently.

Cache-Oblivious Model [Frigo et al 1999]

- Ideal cache model:
  - word size known to everyone in advance
  - two-level memory
  - Memory partitioned in lines of L words each
  - Cache size is Z
  - *Tall Cache Assumption*: $Z = \Omega(L^2)$
  - Eviction algorithm: *farthest in the future*[1]
- *Cache-Oblivious Algorithm:* doesn't know Z, L
- Whenever it fetches word from disk, disk will send the line containing it (like in real world)

---

[1] Can be replaced by LRU efficiently.

Why Cache-Oblivious Algorithms?

- Why should we care about this model?
    - Simpler code
    - Algorithm automatically “tunes” itself during execution
    - It is cool
    - Works for the many levels of memory hierarchy (all with their own parameters of Z and L)

Example: Median Finding

Algorithm: input Array of length n

① think of array partitioned into $\frac{n}{5}$ blocks of 5 elements each

② find median of each block (say by sorting)

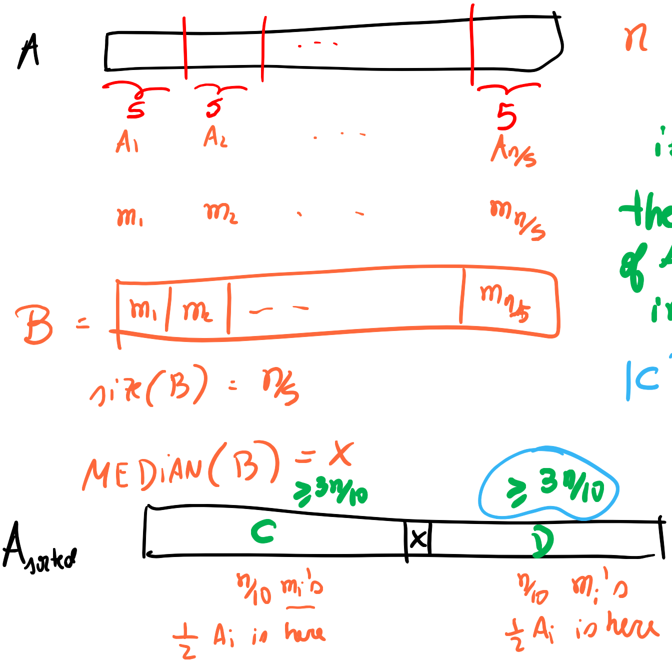
③ Find median of medians, call it x

④ Divide array into two $\begin{cases} \leq x \\ > x \end{cases}$

Recurse on array with most # elements.

Total work: $O(n)$

Memory used: ??



if $|C| > |D|$ then median of A must be in C

$|C| \leq \frac{7n}{10}$ elements

Example: Median Finding

$MT(n)$ ← # memory transfers that our algorithm will do on input size n

Algoritm, with memory handling:

① nothing to do (we are just thinking)

② find median of each block : can do this with linear scan through array ∴ $O(n/L + 1)$

③ find median of medians:
- write each median to main memory $O(n/L + 1)$
- recurse with our median finding algorithm $MT(n/5)$

④ Partition array into $\leq x$, $> x$
- do this with a linear scan $O(n/L + 1)$
- write arrays to memory $O(n/L + 1)$

⑤ Recurse $MT(7n/10)$

Example: Median Finding

Picture

Example: Median Finding - Recursion analysis

$$MT(n) = \underline{MT(n/5)} + \underline{MT\left(\frac{7n}{10}\right)} + \underline{O\left(n/L + 1\right)}$$

Base case: $MT(O(1)) = O(1)$ (i.e. ignoring cache)

$$MT(n) \geq \# \underbrace{\text{leaves in recursion}}_{L(n)}$$

$$\underline{L(n)} = \underline{L(n/5)} + \underline{L\left(\frac{7n}{10}\right)} \rightarrow \underline{n^{\alpha} = (n/5)^{\alpha} + \left(\frac{7}{10}n\right)^{\alpha}}$$

$$\Rightarrow \alpha \approx 0.83$$

$$\therefore MT(n) \geq n^{0.83} \underbrace{= \omega(n/L)} \text{ if } \underline{L = \omega(n^{0.17})}$$

Example: Median Finding - Recursion analysis

$$MT(n) = MT(n/5) + MT\left(\frac{7n}{10}\right) + \boxed{O(n/L + 1)}$$

Base case : $MT(O(L)) = O(1)$ (i.e. using cache)

$\Rightarrow$ # leaves now is $L(n) = (n/L)^{\alpha} = o(n/L)$

Cost at each level decreases geometrically (practice problem)

$\therefore$ $MT(n)$ dominated by root cost

$$\therefore \boxed{MT(n) = O(n/L + 1)}$$

- Administrivia
- Designing Algorithms in Real Life
- Cache-Aware and Cache-Oblivious Algorithms
- Conclusion
- Acknowledgements

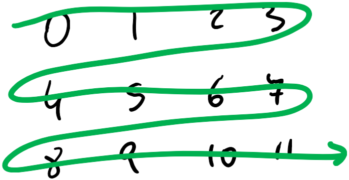
Cache-Aware Algorithm: Matrix Multiplication

Input : A, B $n \times n$ matrices

Output : $C = AB$

How to store matrices?

- major-row order: $A_{ij} > A_{i'j'}$ if $\begin{cases} i < i' \end{cases}$ or $\begin{cases} i = i' \\ j < j' \end{cases}$



0	1	2	3
4	5	6	7
8	9	10	11

Cache-Aware Algorithm: Matrix Multiplication

- divide matrices A, B, C into blocks (until they fit into cache) then multiply blocks
- let s be size of the blocks (parameter that must be set by algorithm)

```
BLOCK-MULT_s(A, B, C, n)
    for i ← 1 to n/s
        for j ← 1 to n/s
            for k ← 1 to n/s
                MULT_s(A_ik, B_kj, C_ij)
```

ordinary MaMu

$AB = C$

$O(n^3)$ time

Cache-Aware Algorithm: Matrix Multiplication

- assume $s \mid n$ (otherwise more code to write)
- $s \times s$ matrices take $O(s^2/L)$ cache lines
- from tall cache assumption $(Z = \Omega(L^2))$

  $s = O(\sqrt{Z})$ to minimize

  cache complexity

$$s = \sqrt{Z/3} \longmapsto \underbrace{3}_{A,B,C} \times \underbrace{\sqrt{\frac{Z}{3}} \cdot \sqrt{\frac{Z}{3}}}_{\text{size of matrix}} = Z$$

Cache-Aware Algorithm: Matrix Multiplication

- each call to MULT run with

$$O\left(\frac{Z}{L}\right) = O\left(\frac{s^2}{L}\right) \text{ cache misses}$$

- $i, j, k \in \{1, \ldots, n/s\}$ $\qquad n/\sqrt{Z} = n/s$

algorithm has

$$\underbrace{O\left(\left(\frac{n}{\sqrt{Z}}\right)^3 \cdot \frac{Z}{L}\right)}_{O\left(\frac{n^3}{L\sqrt{Z}}\right)} \text{ cache misses}$$

Cache-Aware Algorithm: Matrix Multiplication

Cache-Aware Algorithm: Matrix Multiplication

Cache-Oblivious Algorithm: Matrix Multiplication

- use regular MaMu algorithm
- total work : $O(n^3)$
- matrices are given in major-row

Remark: using Straßen's algorithm $3 \rightarrow \lg 7$

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \qquad A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$

$$C_{12} = A_{11} B_{12} + A_{12} B_{22}$$

$$\boxed{C_{11} \leftarrow C_{11} + A_{11} \cdot B_{11}}$$

if A_{11}, B_{11}, C_{11} don't fit in cache we recurse.

Cache-Oblivious Algorithm: Matrix Multiplication

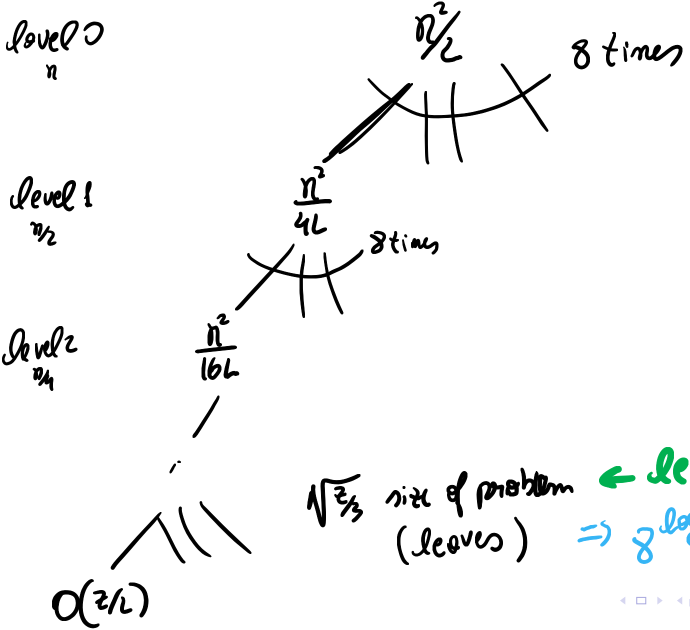
$$MT(n) \le 8\,MT(n/2) + \underbrace{O\left(\begin{array}{c}\text{\# cache misses}\\ \text{to put everything together}\end{array}\right)}_{O(n^2/L)}$$

linear scan
to add the
multiplied blocks

Base case: $$\underset{}{MT\left(\sqrt{Z/3}\right)} = \underbrace{O(Z/L)}_{}$$

\# cache misses
to bring matrices
to cache

Cache-Oblivious Algorithm: Matrix Multiplication



$\frac{n^2}{2}$

$\frac{2n^2}{2}$ inc. geom.

$\frac{4n^2}{2}$

← level $\log(n/\sqrt{Z})$

$\Rightarrow 8^{\log(n/\sqrt{Z})}$ leaves

Cache-Oblivious Algorithm: Matrix Multiplication

$$O\Big(\underbrace{8^{\log(n/\sqrt{Z})}}_{\#\text{ leaves}} \cdot \underbrace{Z/L}_{\#\text{ cache misses per leaf}}\Big) = O\left(\left(\frac{n}{\sqrt{Z}}\right)^3 \cdot \frac{Z}{L}\right)$$

$$= O\left(\frac{n^3}{L \cdot \sqrt{Z}}\right).$$

Cache-Oblivious Algorithm: Matrix Multiplication

Cache-Oblivious Algorithm: Matrix Multiplication

Cache-Oblivious Algorithm: Matrix Multiplication

Conclusion

- We saw today how memory management affects execution of our algorithms

Conclusion

- We saw today how memory management affects execution of our algorithms
- Cache-Oblivious model: design algorithms with asymptotically optimal use of memory.
  - Simpler code
  - Efficient use of memory
  - Algorithm automatically “tunes” itself during execution
  - It is cool
  - Works for the many levels of memory hierarchy (all with their own parameters of Z and L)

Conclusion

- We saw today how memory management affects execution of our algorithms
- Cache-Oblivious model: design algorithms with asymptotically optimal use of memory.
    - Simpler code
    - Efficient use of memory
    - Algorithm automatically "tunes" itself during execution
    - It is cool
    - Works for the many levels of memory hierarchy (all with their own parameters of Z and L)
- Matrix Multiplication in cache-oblivious format

Acknowledgement

- Lecture based largely on:
  - [Frigo et al 1999]
  - 6.046 lecture notes

    `http://stellar.mit.edu/S/course/6/sp15/6.046J/courseMaterial/topics/topic2/lectureNotes/L23_-_Cache_Oblivious_I/L23_-_Cache_Oblivious_I.pdf`

References I

Aggarwal, A. and Vitter, J.S., 1988.
The input/output complexity of sorting and related problems.
Communications of the ACM, 31(9), pp. 1116-1127.

Frigo, M., Leiserson, C.E., Prokop, H. and Ramachandran, S., 1999.
Cache-oblivious algorithms.
In 40th Annual Symposium on Foundations of Computer Science (pp. 285-297). IEEE.