/INFOMOV/

Optimization & Vectorization

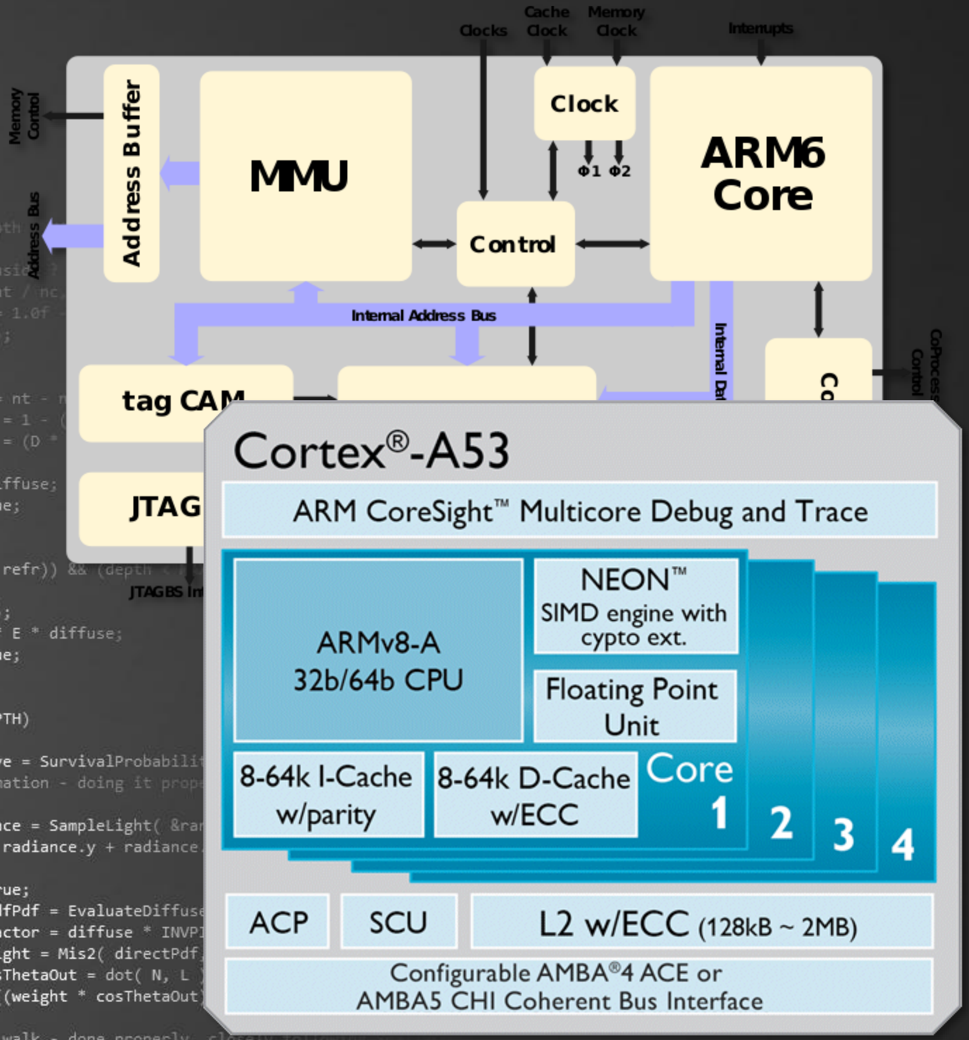
J. Bikker - Sep-Nov 2019 - Lecture 12: “Cache-Oblivious”

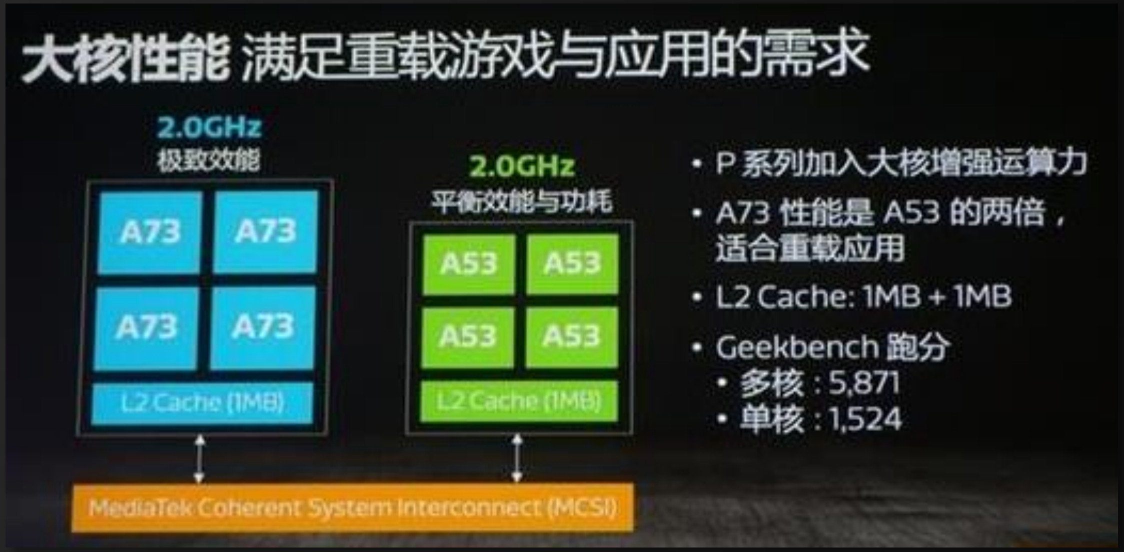
Welcome!

Today’s Agenda:

- Introduction
- The Idealized Cache Model
- Divide and Conquer
- Sorting
- Digest

Introduction





L1$= ?
L2$=?
L3?
L4?
L5?

Introduction

Dealing with Different Architectures

Modern hardware is not uniform

- Number of cache levels
- Cache sizes and cache line size
- Associativity, replacement strategy, bandwidth, latency…

Programs should ideally run for different parameters

- Works if we determine the parameters at runtime
- (or perhaps a few important ones)
- Or we just ignore the details. *(i.e., what we do in practice)*

Programs are executed on unpredictable configurations

- Generic portable software libraries
- Code running in the browser

Introduction

Dealing with Different Architectures

Modern hardware is not uniform

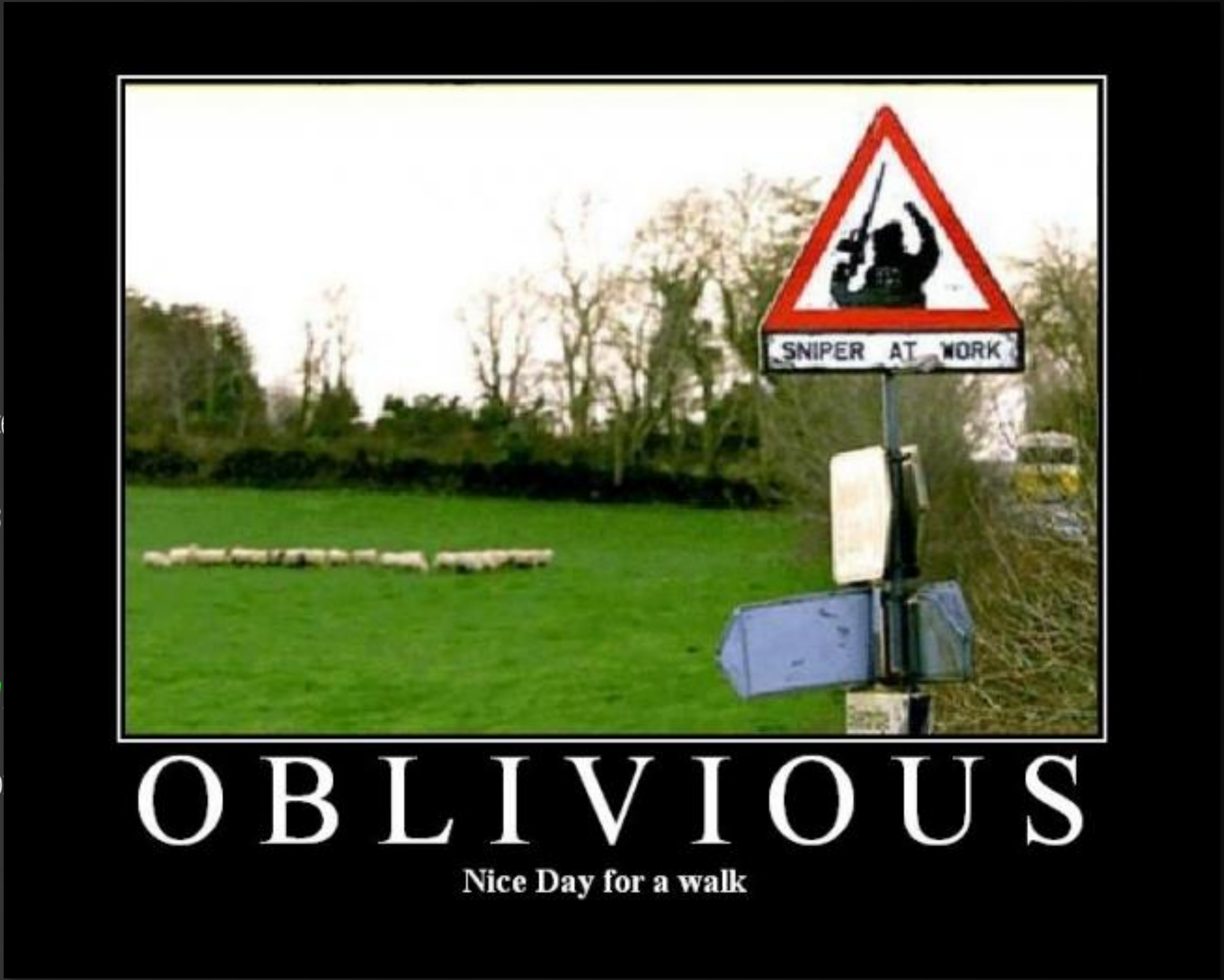
- Number of cache levels
- Cache sizes and cache line size
- Associativity, replacement strategy, bandwidth, lat

Programs should ideally run for different parameters

- Works if we determine the parameters at runtime
- (or perhaps a few important ones)
- Or we just ignore the details. *(i.e., what we do in*

Programs are executed on unpredictable configuratio

- Generic portable software libraries
- Code running in the browser



Introduction

a **cache-oblivious algorithm** is an algorithm designed to *take advantage of a CPU cache without having the size of the cache (or the length of the cache lines, etc.) as an explicit parameter.*

An **optimal cache-oblivious algorithm** is a cache-oblivious algorithm that *uses the cache optimally.*

A cache-oblivious algorithm is effective on *all levels of the memory hierarchy, simultaneously.*

Can we get the benefits of cache-aware code without knowing the details of the cache?

Introduction

People

Cache-Oblivious Algorithms. Harald Prokop, Master thesis, MIT, 1999.
Cache-Oblivious Algorithms. Frigo, Leierson, Prokop, Ramachandran, 1999.
Cache Oblivious Distribution Sweeping. Brodal, Stølting. Lecture notes, 2002.
Cache-Oblivious Algorithms and Data Structures. Brodal, SWAT 2004.

Introduction

Cache-oblivious data structures and algorithms:

Optimizing an application *without knowing hardware details.*

Today's Agenda:

- Introduction
- The Idealized Cache Model
- Divide and Conquer
- Sorting
- Digest

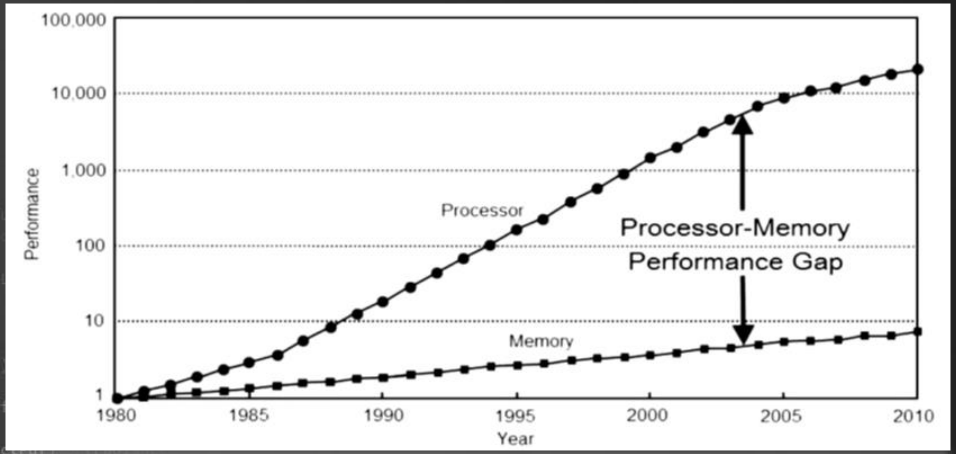
Cache Model

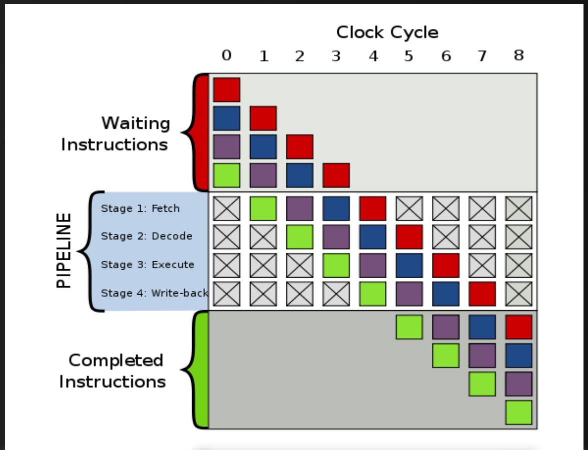
Previously in INFOMOV:

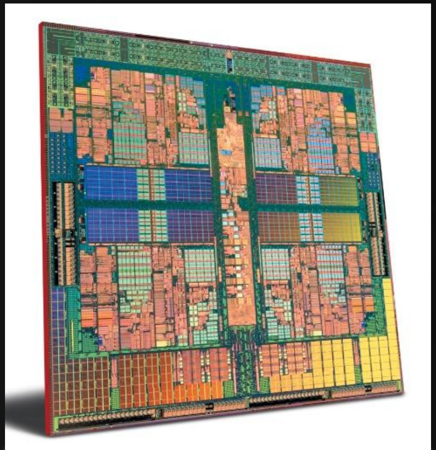
Estimating algorithm cost:

t

1. Algorithmic Complexity : $O(N)$, $O(N^2)$, $O(N \log N)$, ...
2. Cyclomatic Complexity* (or: Conditional Complexity)
3. Amdahl’s Law / Work-Span Model
4. Cache Effectiveness





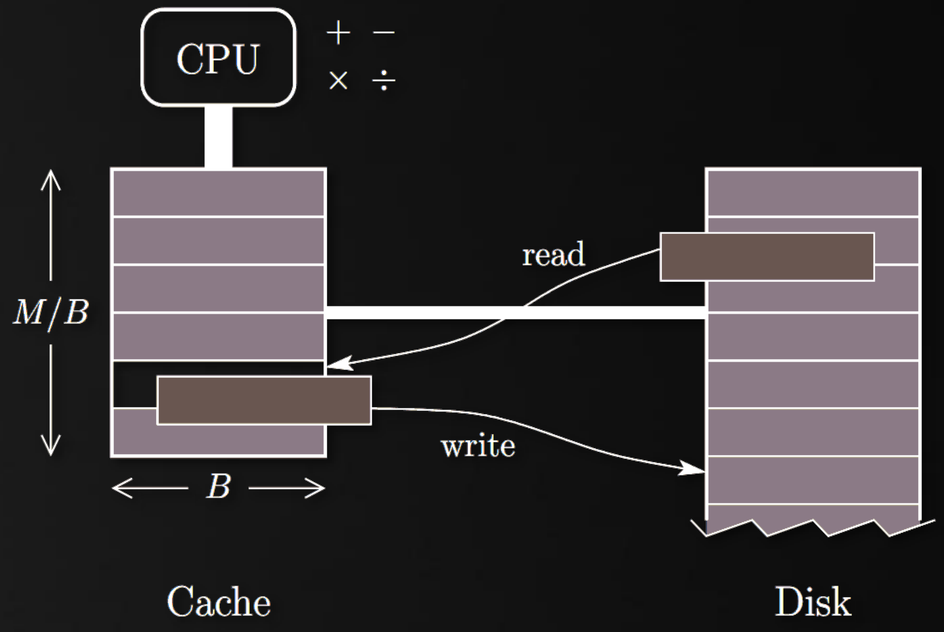


Cache Model

The External-Memory Model

Assumptions*:

- Transfers happen in blocks of B elements.
- The cache stores M elements, in M/B blocks.
- The block count is substantial.
- A cache miss results in transfer of 1 block. If the cache was full, a second transfer occurs (eviction).



The complexity of an algorithm is (solely) measured as the number of cache misses.

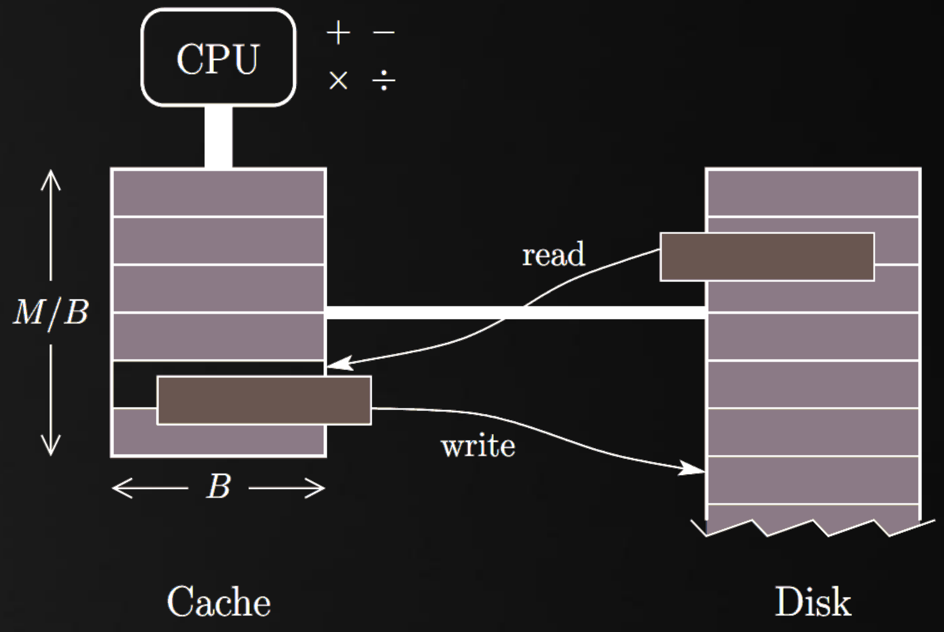
*: Cache-Oblivious Algorithms. Prokop, 1999. MIT Master Thesis.
For a digest, read: http://erikdemaine.org/papers/BRICS2002/paper.pdf

Cache Model

The Cache-Oblivious Model

Assumptions*:

- ~~Transfers happen in blocks of B elements.~~
- ~~The cache stores M elements, in M/B blocks.~~
- The block count is substantial.
- A cache miss results in transfer of 1 block. If the cache was full, a second transfer occurs (eviction).
- The cache is fully associative.
- The replacement policy is optimal.



*: Cache-Oblivious Algorithms. Prokop, 1999. MIT Master Thesis.
For a digest, read: http://erikdemaine.org/papers/BRICS2002/paper.pdf

Cache Model

The Cache-Oblivious Model

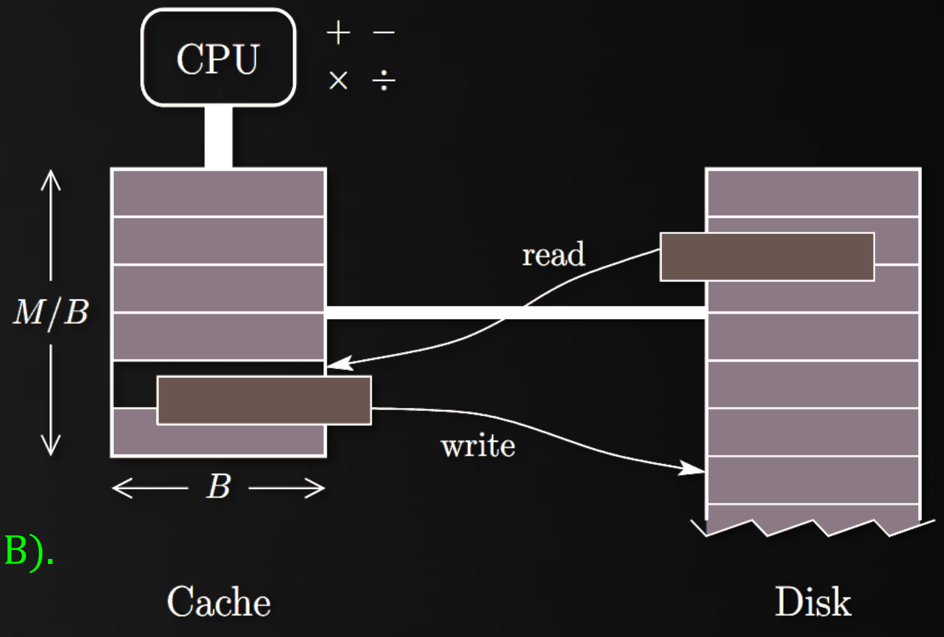
Example:

Calculating the sum of an array of N integers has an algorithmic complexity $O(N)$.

In the external-memory model, the complexity is:
$\lceil N/B \rceil$ *(i.e.: ceil(M/B)).*
(note: this assumes alignment, which requires knowledge about B).

The cache-oblivious algorithm cannot assume specific values for M or B. We therefore get: $\lceil N/B \rceil + 1$.

(note: one extra block, because of alignment)
(note: we do use B in the analysis, but not in the algorithm.)
(note: the complexity is identical to $\lceil N/B \rceil$ for $N = \infty$.)

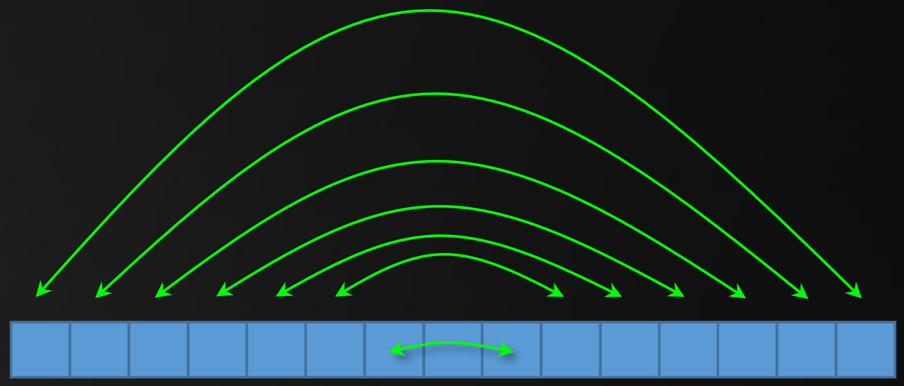


Cache Model

The Cache-Oblivious Model

And now for an actually useful example…

```
void Reverse( int* values, int N )
{
   // ...?
}
```



- Easy to do with a temporary array.
- Cache-oblivious algorithm*:

```
for( int i = 0; i < N/2; i++) { swap( values[i], values[N-1-i] );
```

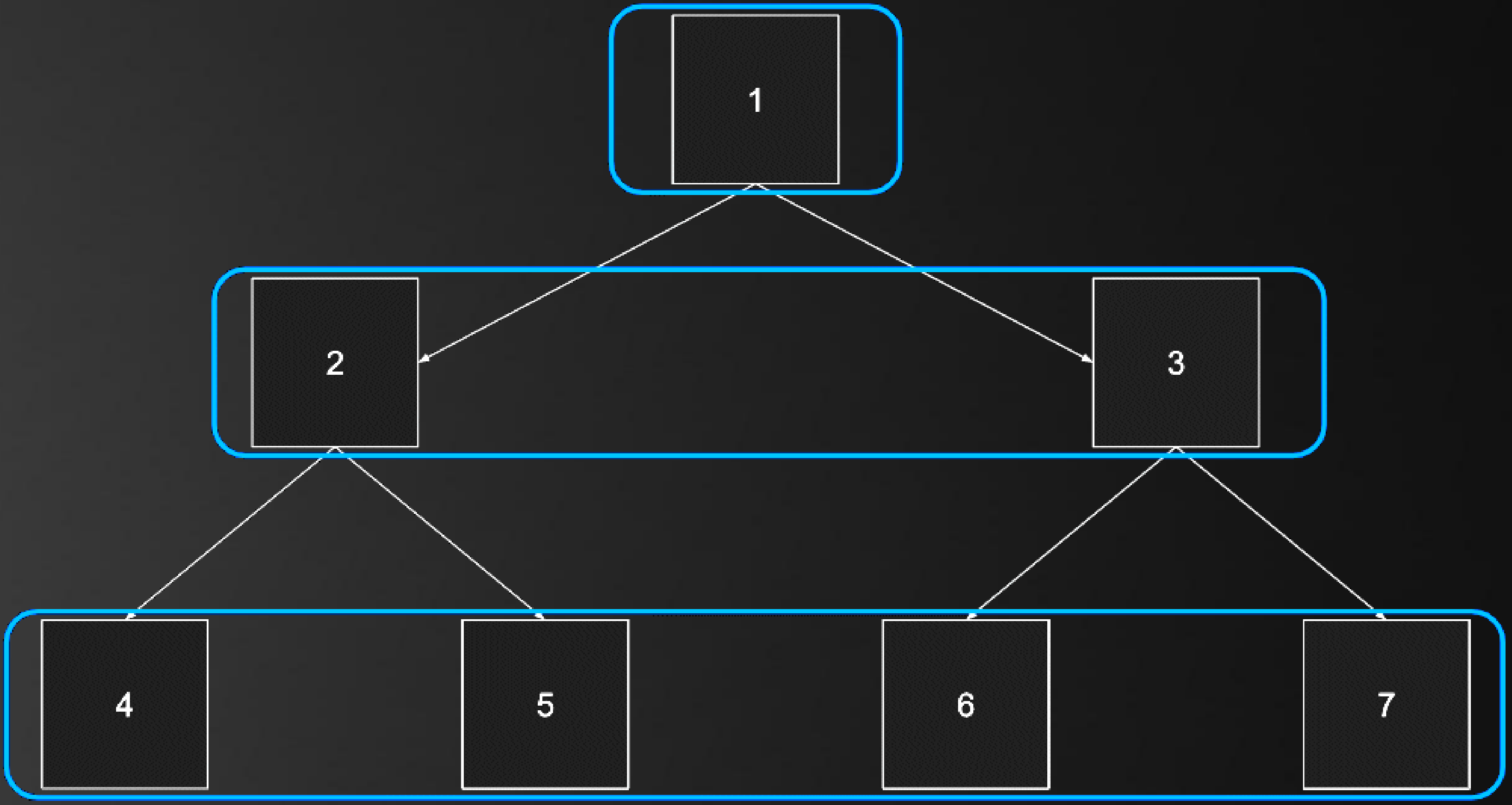
(note: requires as many block access as a single scan.)

*: Programming Pearls, 2nd edition. Jon Bentley, 2000.

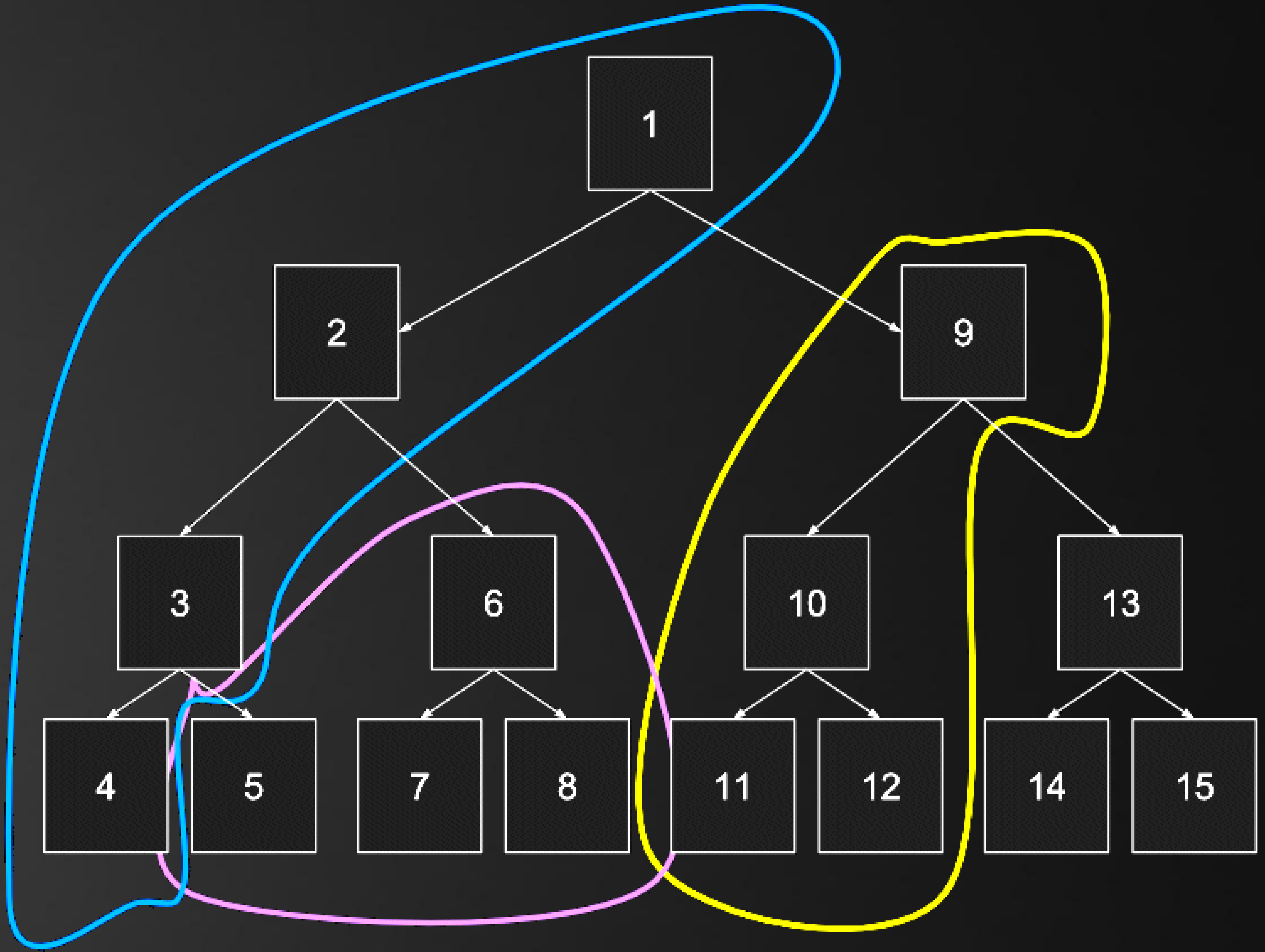
Today's Agenda:

- Introduction
- The Idealized Cache Model
- Divide and Conquer
- Sorting
- Digest

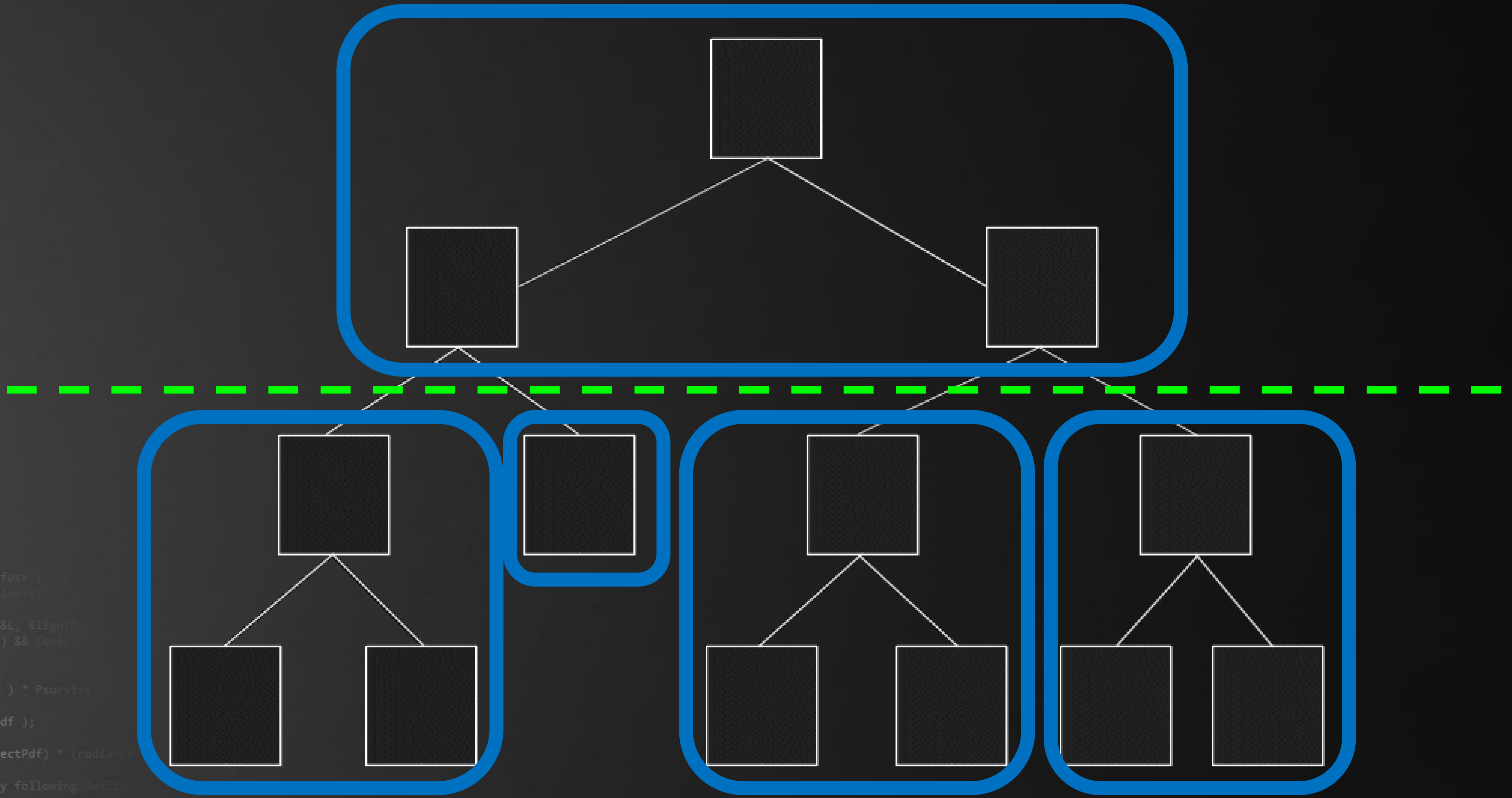
Tree



Tree



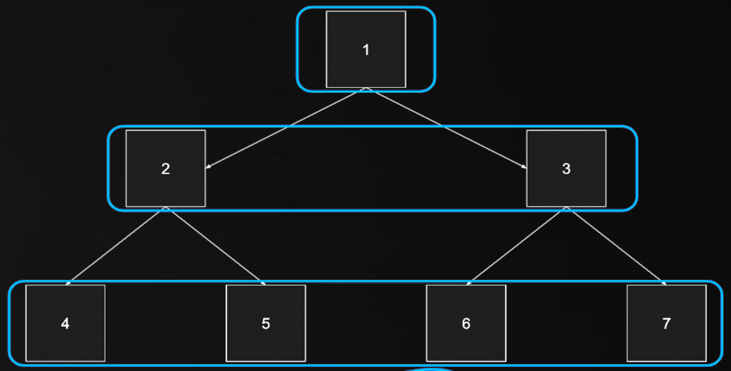
Tree



Tree

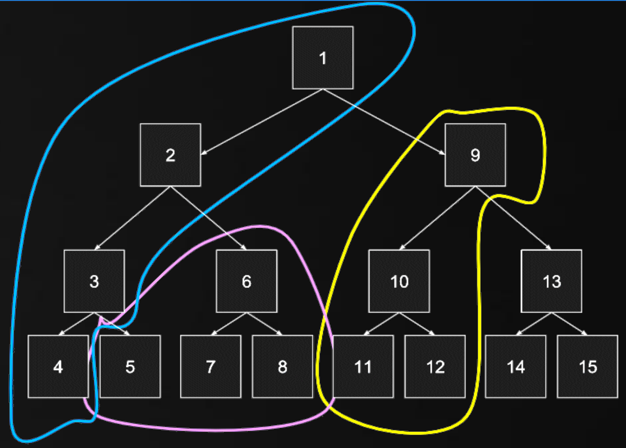
Comparisons

Breadth-first tree:



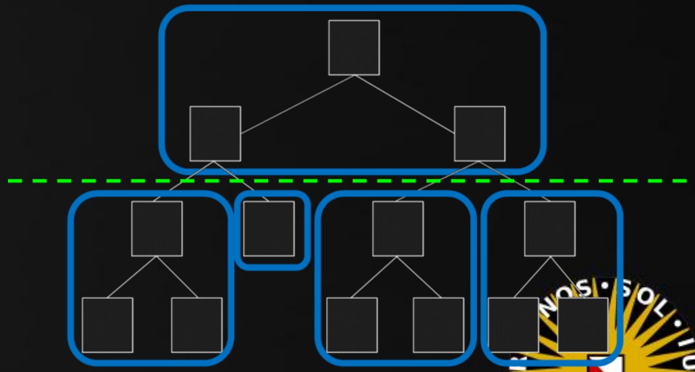
Going down in the tree, every step will access a different block. Expected accesses is $\log_2 N$. (e.g. 16 for N=65536)

Depth-first tree:



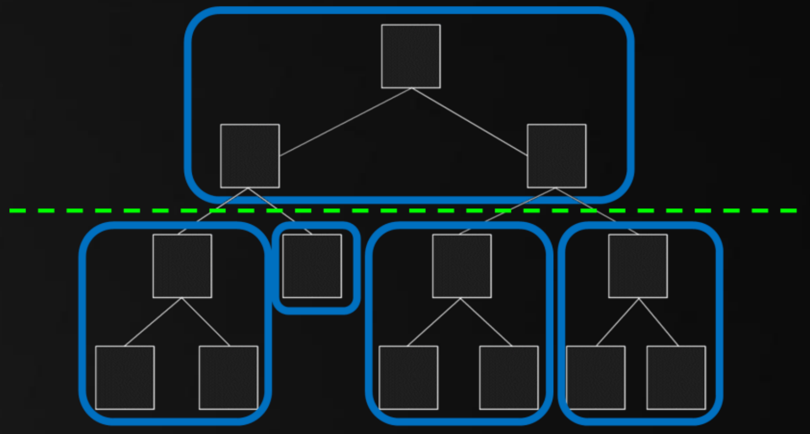
Although left branches are efficient, every right branch requires a different block.

Cache-oblivious layout:



$\frac{\log_2 N}{\log_2 B} = \log_B N$. (e.g. 4 for N=65536, B=16)

Tree

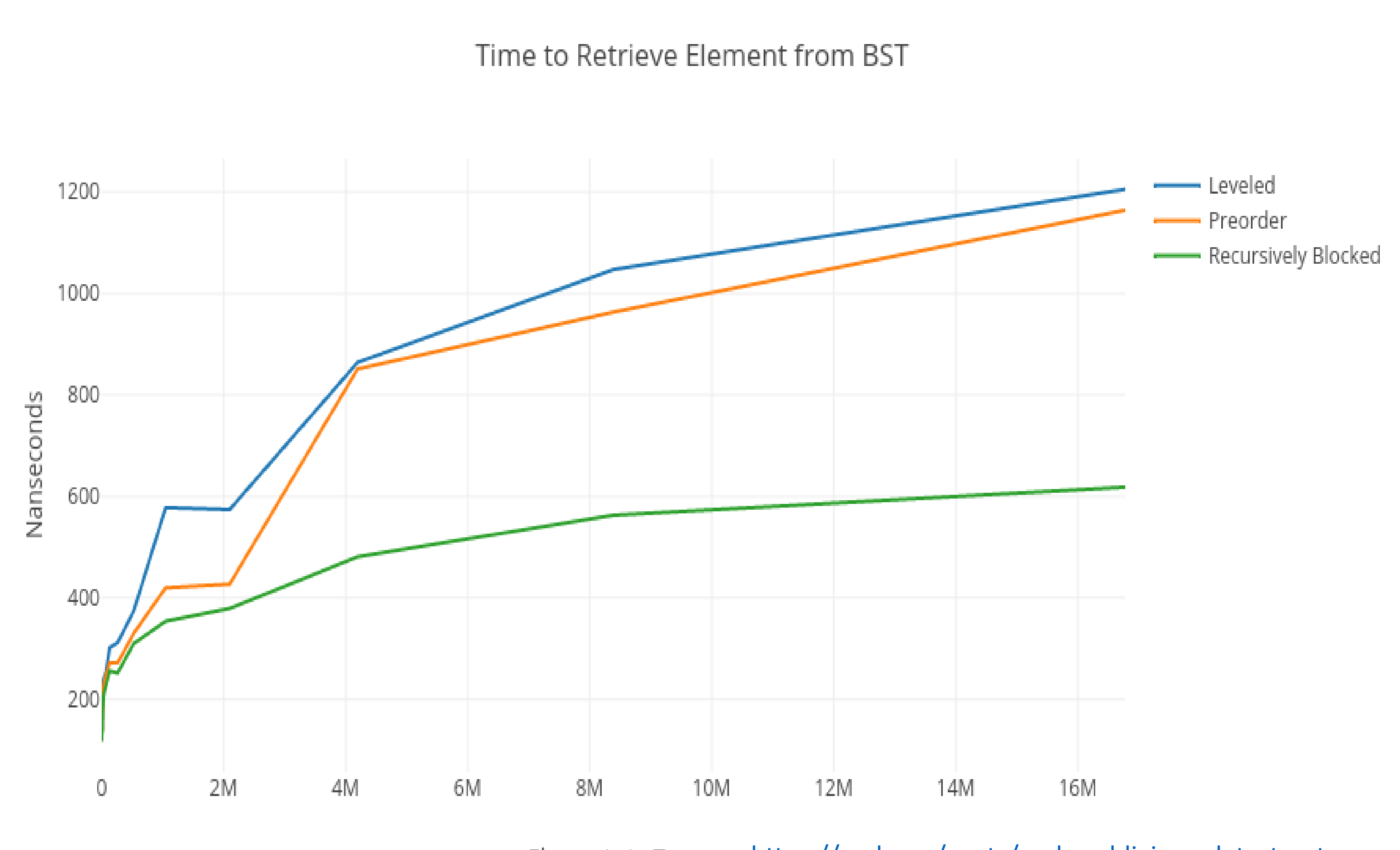


The Cache-Oblivious Tree

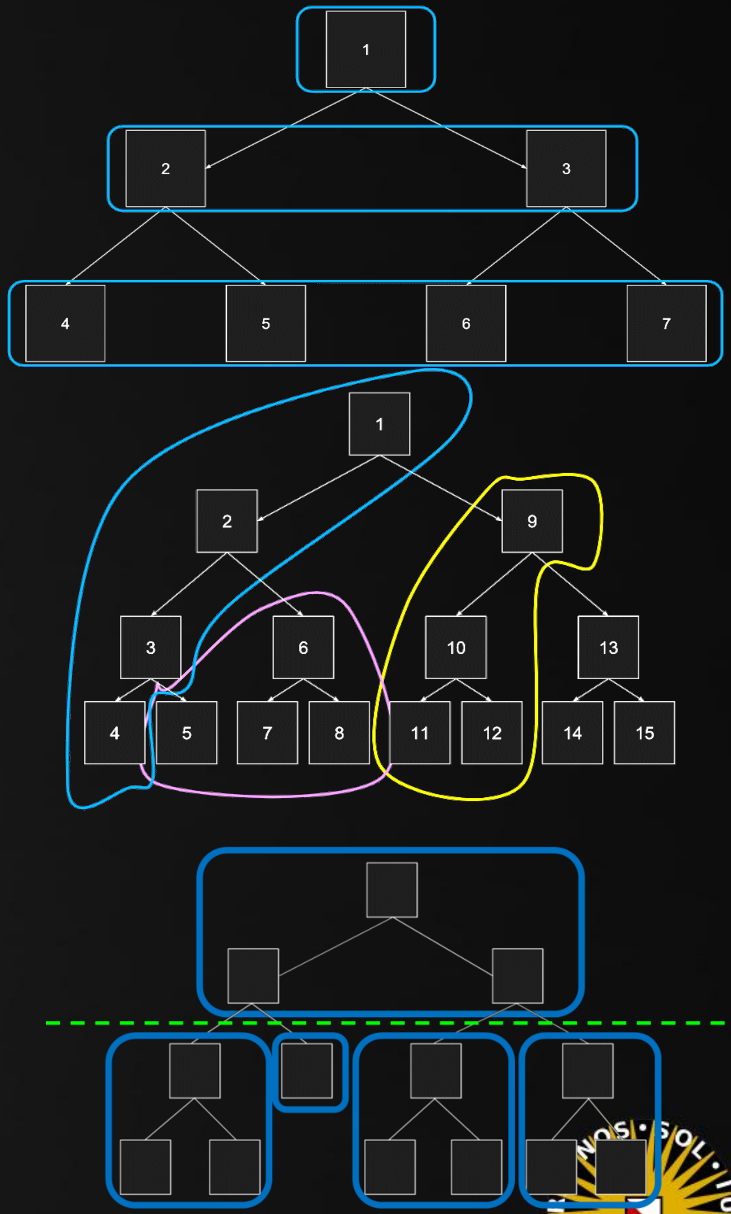
Algorithm:

1. Split the tree vertically, at level $\frac{1}{2}\log(N)$.
   (where N is the number of leaf nodes)
2. The top now contains $\sqrt{N}$ elements.
3. Produce five subtrees and process these recursively.

Tree



https://rcoh.me/posts/cache-oblivious-datastructures

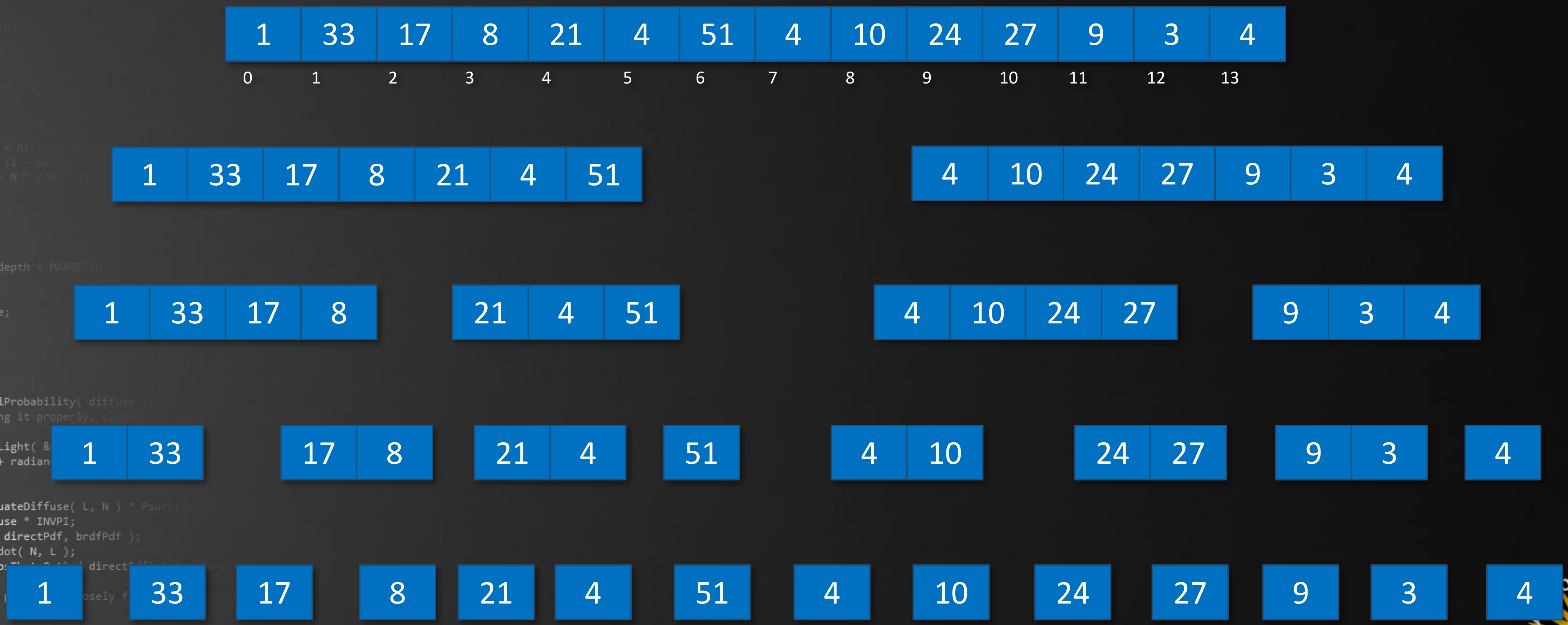


Today's Agenda:

- Introduction
- The Idealized Cache Model
- Divide and Conquer
- Sorting
- Digest

Sort

MergeSort



Sort

MergeSort

1 33 17 8 21 4 51 4 10 24 27 9 3 4

1 33 | 8 17 | 4 21 | 51 | 4 10 | 24 27 | 3 9 | 4

1 8 17 33 | 4 21 51 | 4 10 24 27 | 3 4 9

Merging two buffers A[] and B[] to C[]:

```
*C = *A < *B ? *A++ : *B++
```

Sort

MergeSort

1 33 17 8 21 4 51 4 10 24 27 9 3 4

MergeSort reaches optimal algorithmic complexity if we merge more than 2 streams at a time*.

The optimal number of streams is cache-dependent, namely: M/B.

Recall:

M=cache size,
B=block size.

For 32KB L1$:

M=32768,
B=64,

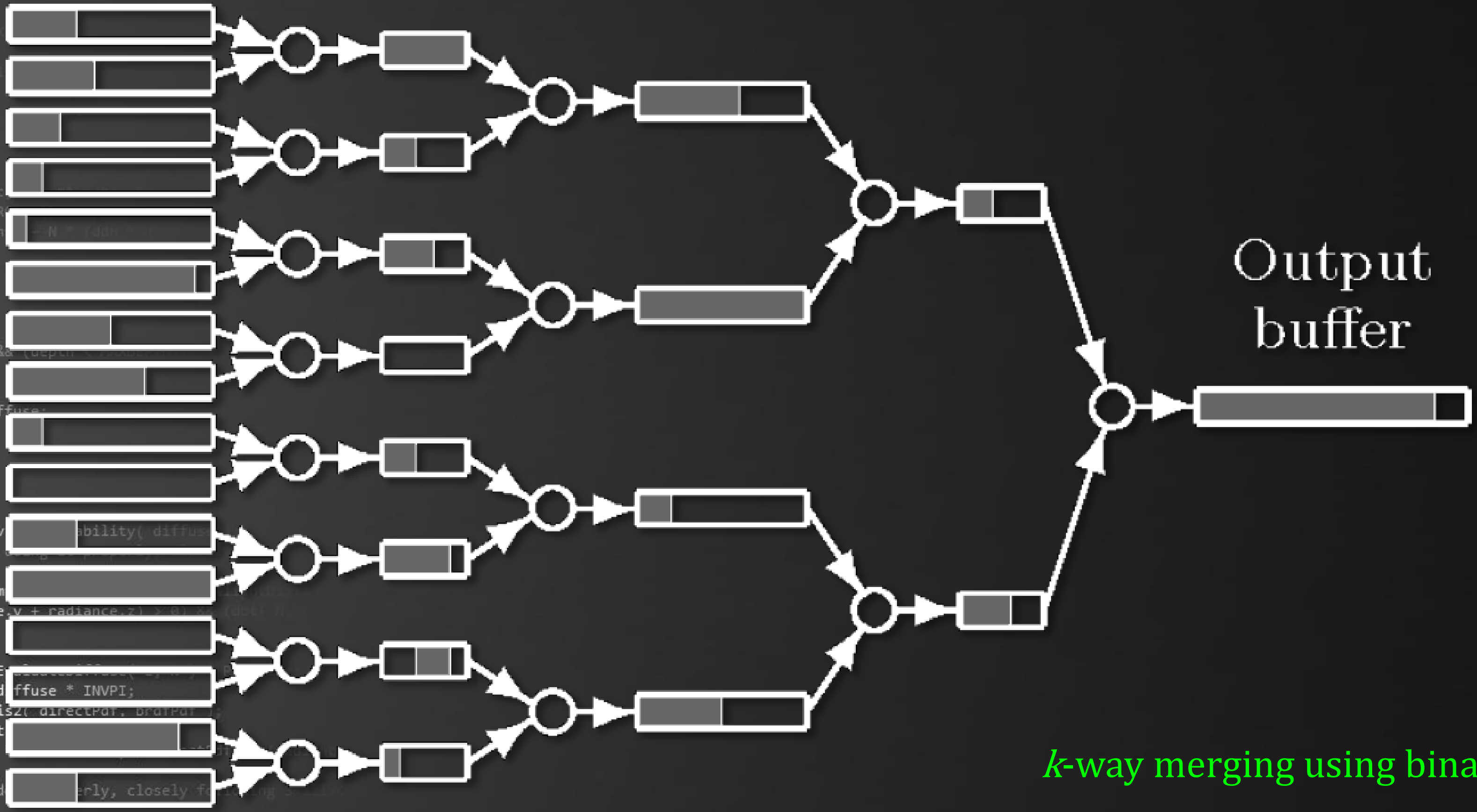
➔ 512-way.

(in this case, MergeSort requires $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ transactions.)

*: The input/output complexity of sorting and related problems. Aggarval & Vitter, 1988.

Sort

FunnelSort (the "lazy" variety)



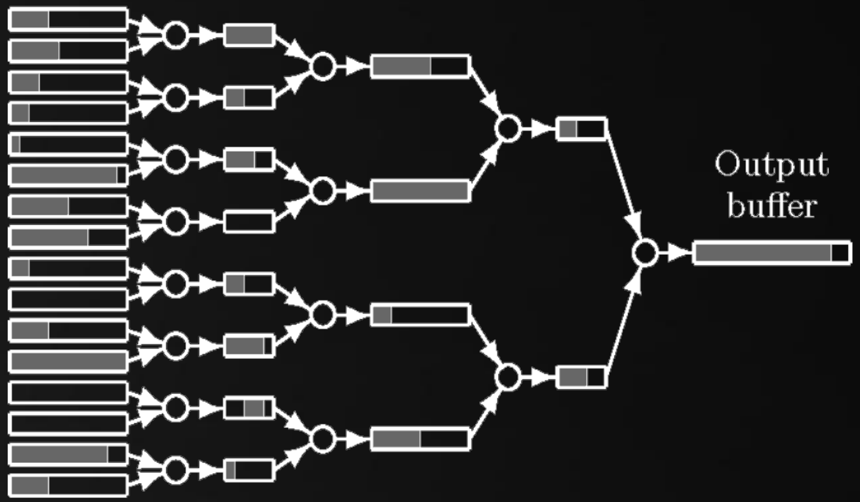

```
void Fill(v)
{
   while (!v.full())
   {
      if (v.left.empty())
         Fill(v.left)
      if (v.right.empty())
         Fill(v.right)
      Merge()
   }
}
```

k-way merging using binary merging with cyclic buffers.

Figure from: Engineering a Cache-Oblivious Sorting Algorithm. Brodal et al., 2007.

Sort

FunnelSort (the “lazy” variety)



How:

- Split the input into $N^{\frac{1}{3}}$ (“cube root”) sets of $N^{\frac{2}{3}}$ elements.

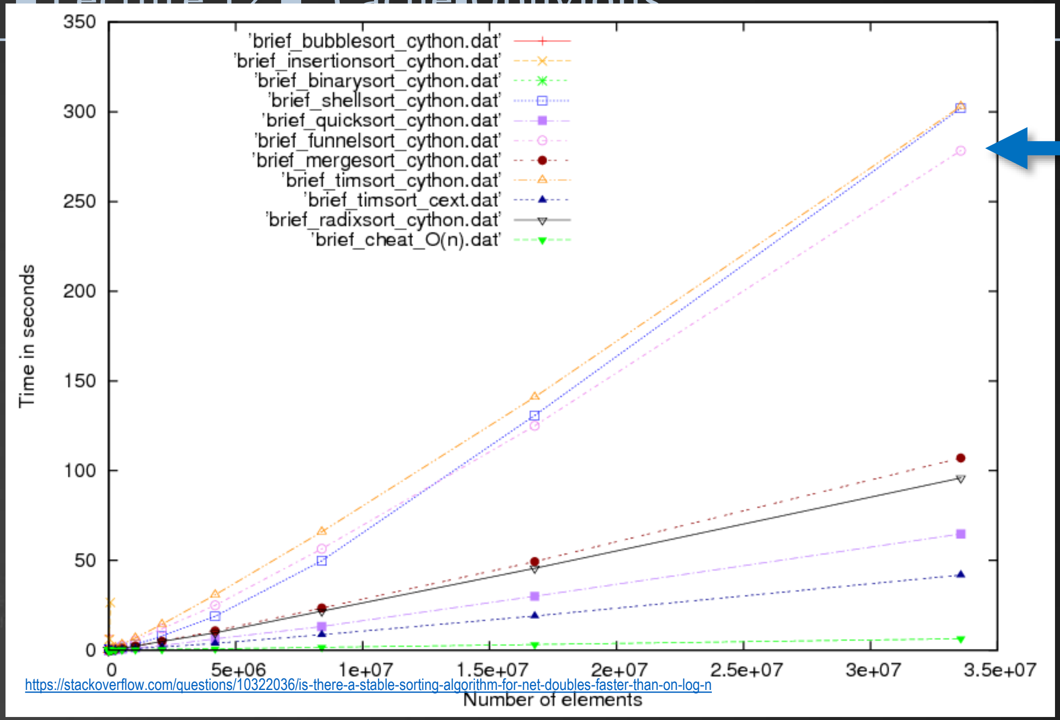
  (so: 1000 becomes 10 sets of 100;
  512 becomes 8 sets of 64, 8 becomes 2 sets of 4.)

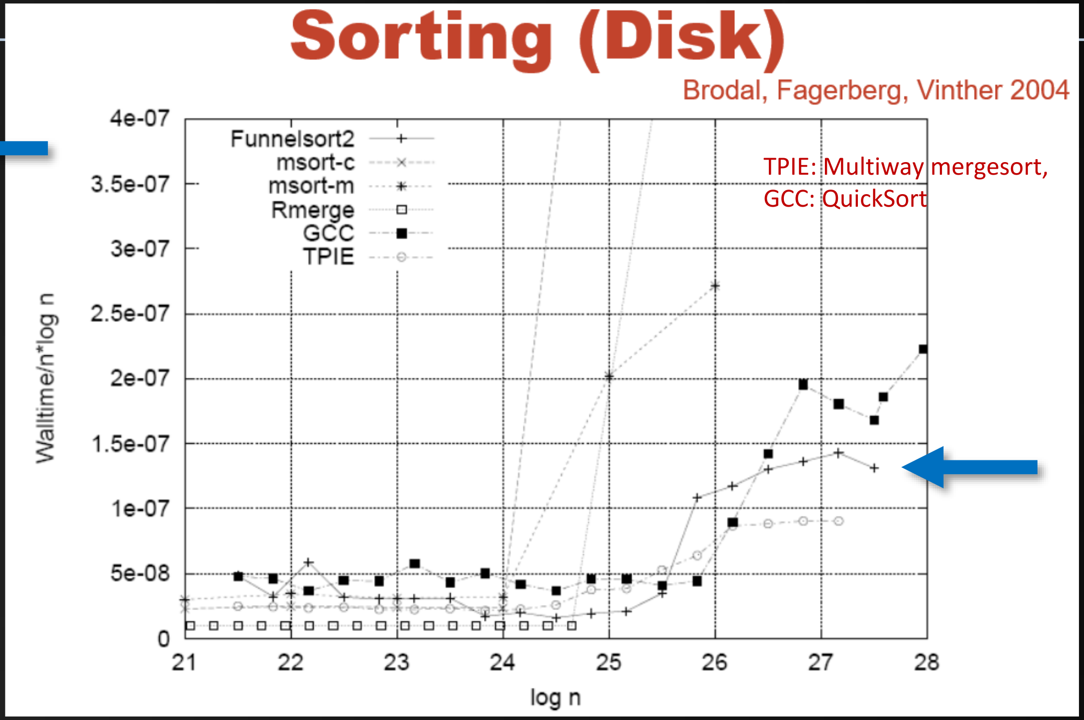
- Recurse.

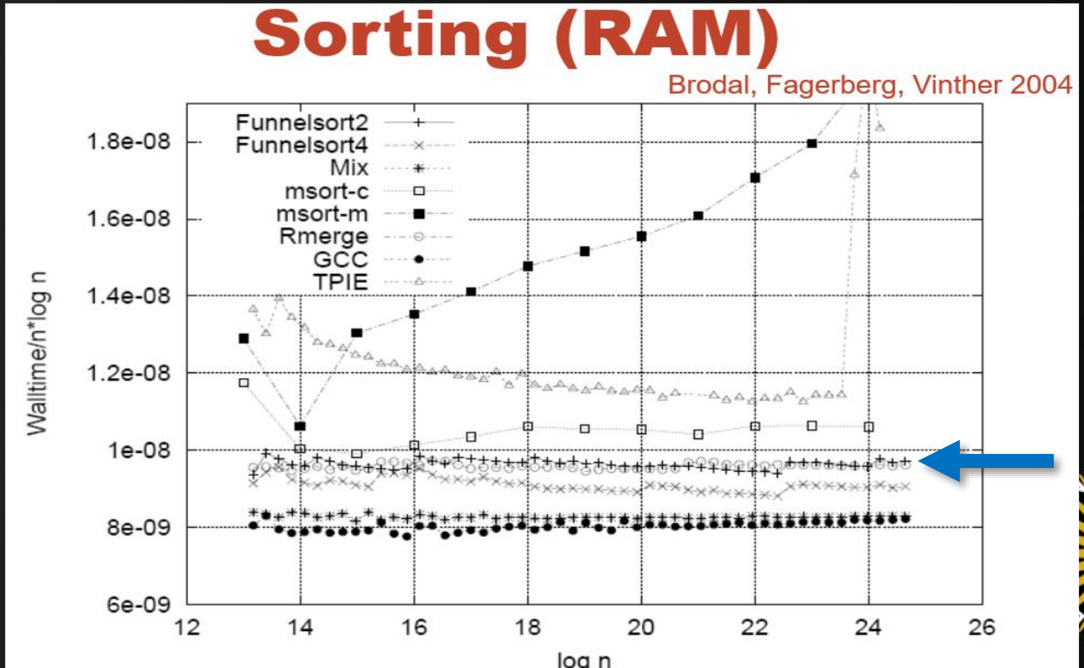
- Merge the $N^{\frac{1}{3}}$ sorted sequences using an $\mathrm{k} = N^{\frac{1}{3}}$ merger.

- The *k*-merger suspends work whenever there is sufficient output.

Sort







Funnelsort works “as advertised” when I/O is expensive.

Today's Agenda:

- Introduction
- The Idealized Cache Model
- Divide and Conquer
- Sorting
- Digest

Digest

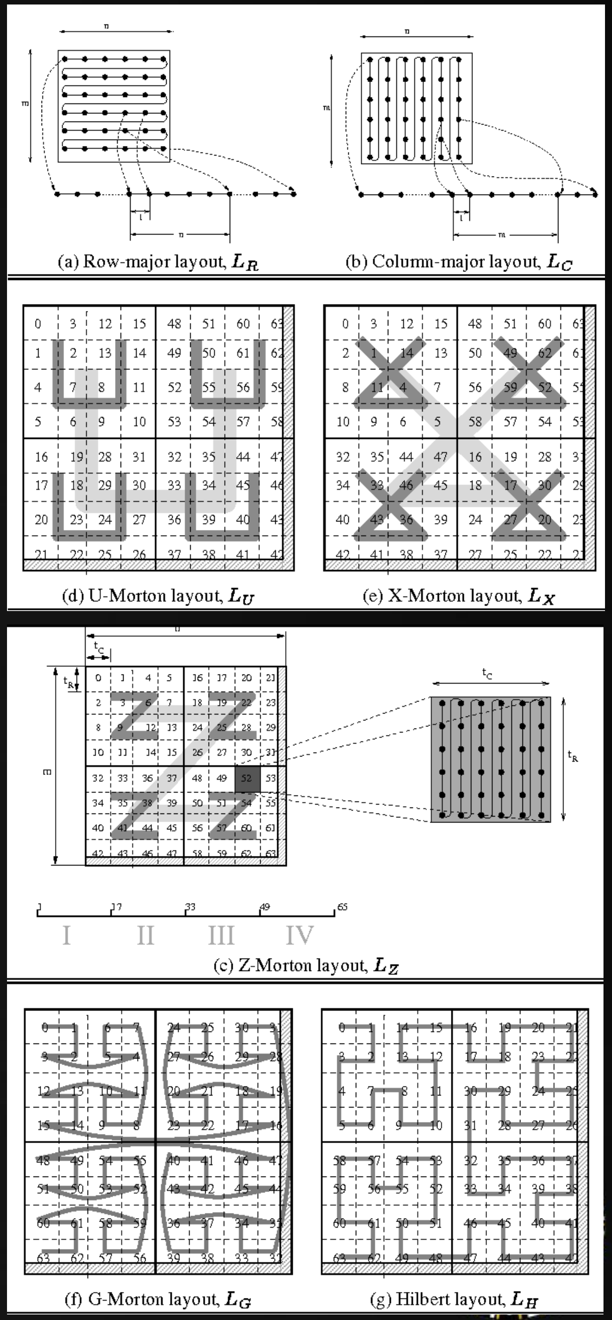
Cache-Oblivious Concepts

Data structures:

1. Linear array – operated on using a scan.
   (works for the most basic cases, but also Bentley’s Reverse)

2. Recursive subdivision
   (not discussed in this lecture, but covered before)

3. Cache-Oblivious tree layout
   (I wish I knew about that one before)



(a) Row-major layout, L_R
(b) Column-major layout, L_C
(d) U-Morton layout, L_U
(e) X-Morton layout, L_X
(c) Z-Morton layout, L_Z
(f) G-Morton layout, L_G
(g) Hilbert layout, L_H

Digest

Cache-Oblivious Concepts

Algorithms:

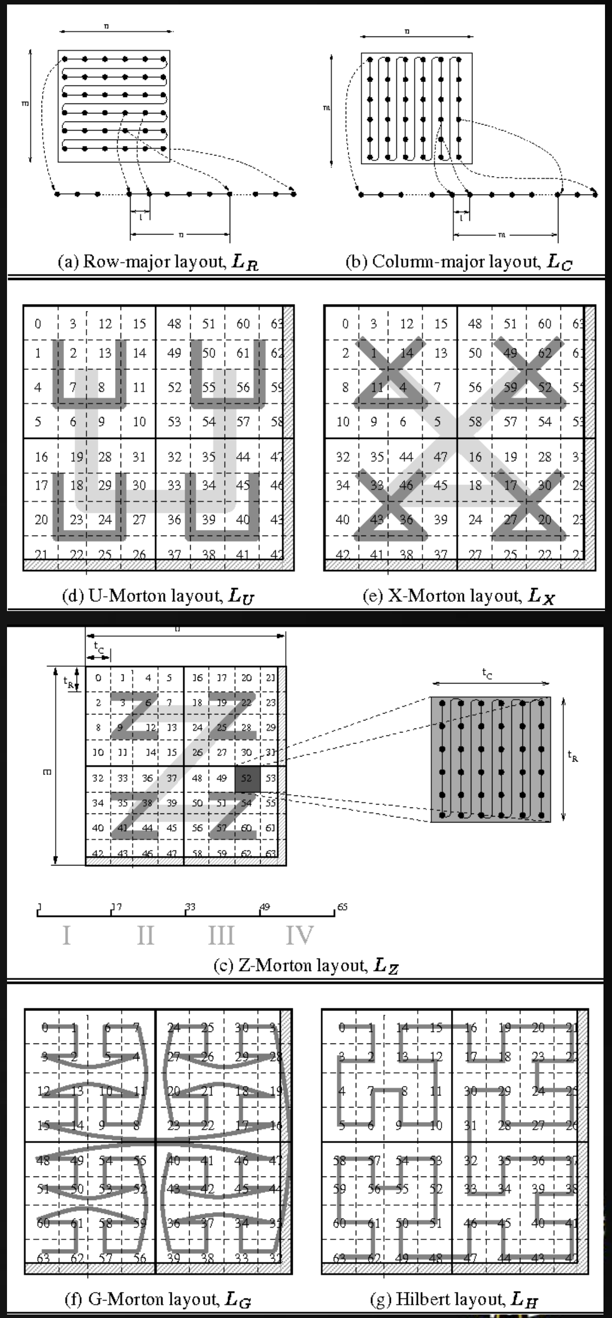
- Often trivially following from data structures.
- Sorting only fast for expensive I/O.

Note the overlap with:

- Data oriented design
- Data-parallel algorithms
- Streaming algorithms

(although there are differences too)

And appreciate the attention to memory cost.



(a) Row-major layout, L_R (b) Column-major layout, L_C

(d) U-Morton layout, L_U (e) X-Morton layout, L_X

(c) Z-Morton layout, L_Z

(f) G-Morton layout, L_G (g) Hilbert layout, L_H

Digest

Cache-Oblivious Concepts

Original question:

Can we get the benefits of cache-aware code without knowing the details of the cache?

IMHO:

- Yes, to some extend.
- But we were not really taking into account cache size anyway
- Nor the specifics of the eviction policy
- And it seems silly not to anticipate a reasonable ‘B’ (e.g. for alignment)

Digest

Cache-Oblivious Concepts

Further reading

“& Cache-Oblivious Algorithms (Updated)”
qstuff.blogspot.com/2010/06/cache-oblivious-algorithms.html

Cache-Oblivious R-Trees:
www.win.tue.nl/~mdberg/Papers/co-rtree.pdf

Cache-Oblivious hashing:
https://www.itu.dk/people/pagh/papers/cohash.pdf

Cache-Oblivious FFT:
https://www.csd.uwo.ca/~moreno/CS433-CS9624/Resources/Implementing_FFTs_in_Practice.pdf

Cache-Oblivious mesh layouts (and other graphics-related CO topics):
http://gamma.cs.unc.edu/COL/

Today’s Agenda:

- Introduction
- The Idealized Cache Model
- Divide and Conquer
- Sorting
- Digest

/INFOMOV/

END of “Low Level”

next lecture: “Snippets & Multi-Threading”