Applied Algorithms Lec 4: Optimation Contd., External Memory Model

Sam McCauley

September 17, 2024

Williams College

Admin

- Homework 1 due Thursday night; office hours Wed 2-5 and Thu 3-5—come with questions (more details next slide)

- Some reading today! Optional/potentially useful for reference. We don't cover the topic in exactly the same way
    - For ex: we'll have $K = 1$; no distribution sort; no B-trees

Homework 1 Updates

- Time data is fixed
- The private time data is a little longer than the one I gave you (so expect somewhat slower absolute times)
- The “Sam” code is pretty fast because it doesn’t binary search
- Hint: what happens if you sort both the left and the right halves?

C Debugging

- Make sure your code editor is happy with your code
- Then: run `gcc` (probably using `make`)
- Then: run `valgrind`
  - Then: `valgrind test.out testData.txt timeData.txt`
  - It's slow. But will (often) literally just tell you every memory-based bug in your program
  - If you run `make clean` and `make debug` so `valgrind` will give you line-by-line pointers
  - To check for memory leaks: `valgrind --leak-check=full test.out testData.txt timeData.txt`
- Then normal debugging with gdb etc.

Revisiting Division

Revisiting Division

- We saw last time: integer division is a little faster than I was saying
- I found some things online saying integer division performance has improved significantly in the last couple years
- Principle does still remain
  - I had a project last winter where an extra division significantly affected performance
  - (And another division that did *not*—in fact the compiler used a fancy math trick to get rid of it!)
- This experience should underline that rules of thumb like “division is slow” is just a direction to look—always back it up with experiments

Note on time taken

Latency vs throughput:

- Latency: time it takes for a sequence of *data-dependent* operations of a given type
- Throughput: time after a previous operation when a new operation of the same type can begin.
- Let's look at an example: `latencythroughput.c`

Bear this distinction in mind when designing experiments!

Optimization Costs so Far

- Remember: code cleanly; let your compiler optimize as much as it can!
- Compiler is great with local optimizations that are obviously correct at compile time

Optimization Costs so Far (with Comments on Compiler)

- Lower cost: expensive vs cheap operations; latency vs throughput; memory allocations themselves; expensive casts
  - A few clock cycles per optimization; compiler is quite good at these
- Moderate cost: branch mispredictions
  - Could be on the order of 20 cycles if branch is difficult to predict!
  - Fewer cycles, but still expensive, if easier to predict
  - Compiler can do this when the improvement is not *algorithmic*
- Let’s talk about the *high* cost: cache efficiency
  - Hardest for the compiler to help with

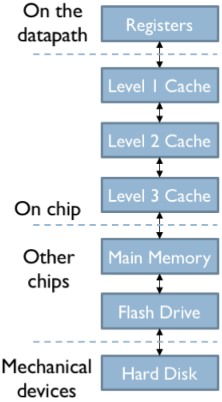
Cache Efficiency

Final major cost: cache misses!

- Data is stored in different places on the computer
- Cost to access the data frequently dominates running time

A Typical Memory Hierarchy

- Everything is a cache for something else…



	Access time	Capacity	Managed By
Registers	1 cycle	1 KB	Software/Compiler
Level 1 Cache	2-4 cycles	32 KB	Hardware
Level 2 Cache	10 cycles	256 KB	Hardware
Level 3 Cache	40 cycles	10 MB	Hardware
Main Memory	200 cycles	10 GB	Software/OS
Flash Drive	10-100us	100 GB	Software/OS
Hard Disk	10ms	1 TB	Software/OS

How caches work

- Can *always assume*: your computer stores data in the optimal(ish) place
- Moves data around in cache lines of $\approx$ 64 bytes
- Modern caches are *very* complicated
- Can be advantages of accessing adjacent cache lines
- Basically: close is good; recent is good; jumping around is bad.
- Examples: `sortedlinkedlist.c` and `unsortedlinkedlist.c`

Code Profiling

Profiling code

- Why not just have your computer tell you what functions are called the most, or keep track of how long they run, or monitor specific high-cost operations like cache misses?
- Lots of such tools! We'll look at a couple of them right now, and use them throughout the class.
    - `gprof`
    - `cachegrind`
    - We won't use `perf` but some people like it
    - We won't use Intel VTune either but seems very cool and powerful
- What do you think some advantages and disadvantages are of using profiling software?

Profilers examples: gprof

- Compile with `-pg` option; then run normally; then run gprof on the executable
- Gives information about what calls what and how much time is in each
- Not perfect, but gives us some information, especially for simpler programs
    - Can see if one function is called a LOT
    - Can see if one function is only ever called by one other function
    - (Can be issues with optimizations, especially `-O3`)
- I may ask you to use this, but be aware of its limitations
- Let's run gprof on my `twotowers.c`

Profilers examples: cachegrind

- Compile with debugging info on `-g` AND optimizations on
    - What does this entail immediately?
- Then `valgrind --tool=cachegrind [your program]` wefoiaewfjoiafwej
- Use `--branch-sim=yes` for branch prediction statistics.
    - Very oversimplified unfortunately
- Outputs number of cache misses for instructions, then data, then combined
- Simulates a simple cache (based on your machine) with separate L1 caches for instructions and data, and unified L2 and (if on machine) L3 caches
- Does L1 misses vs last level (L3) misses
- Virtual machine: not 100% accurate; *slow*
- Let's do `sortedlinkedlist.c` and `unsortedlinkedlist.c`; plus `branchpredictions.c`

Reading cachegrind output

- I, I1, LLi, etc.: *instruction* misses
- D, D1: first level of cache
- LL: last layer of cache
- Run `cg_annotate cachegrind.out.118717` (the last number will change based on which cachegrind run you are referring to) for function-by-function and line-by-line stats
  - Extremely wide output; probably want to pipe to a file
- Let’s do `cg_annotate` for our run of `unsortedlinkedlist.c`

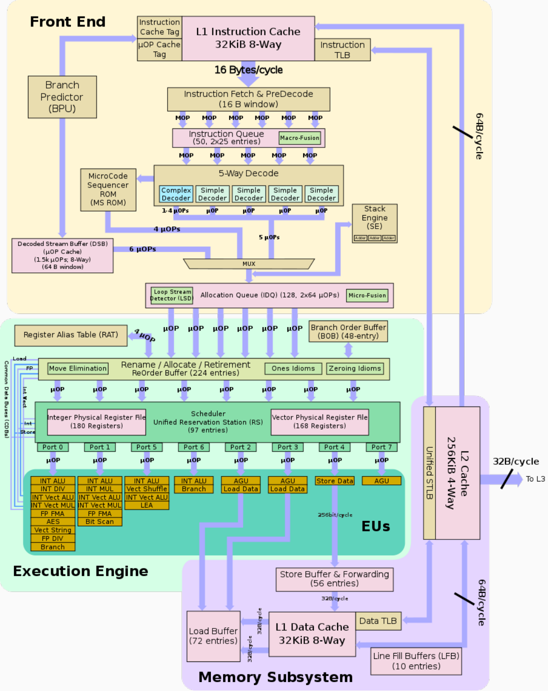
Real-World Caching

- We looked at a simple, constructed example where caching is easy to reason about
- But: bear in mind that modern caches are very complicated; interact nontrivially with other costs (branch mispredictions; expensive operations; etc.)
- I should at least mention *prefetching*: if your computer thinks it can get a head start on fetching your data, it will
  - Can also instruct the compiler to do this manually (another very fine-grained optimization)
- Model things the best you can, but always use experiments when you’re not sure

Costs of Computation

Designing Experiments

- As said before: make sure macroscopic time
- Try to avoid loop overhead when possible
- Be careful that the code *actually does* what you’re testing
- Careful: need to make sure the compiler does not optimize out your test!
- Compiler explorer; or compile using `gcc -S --verbose-asm`


Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
64B/cycle
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
Stack Engine (SE)
Adder
1-4 µOPs
µOP
4 µOPs
5 µOPs
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
6 µOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
Register Alias Table (RAT)
4 µOP
Branch Order Buffer (BOB) (48-entry)
Load
FP
Common Data Buses (CDBs)
Int Vect
Int
Store
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
FP FMA
AES
Vect String
FP DIV
Branch
INT MUL
Bit Scan
Vect Shuffle
LEA
AGU
Load Data
Store Data
256bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 256KiB 4-Way
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
32B/cycle
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
64B/cycle
Memory Subsystem

Optimization Conclusions

- Different places where we can incur costs (in increasing order of cost):
  - Operations
  - Branches and moving around instructions
  - Cache misses

- Determining costs is a matter of experimentation on modern machines!
  - Rarely perfect!

- Theme throughout class: design different experiments to test different aspects of code performance.

External Memory Model

Measuring cache misses

- Takeaway from today's examples: cache performance is often *more important* than number of operations
- But algorithmic analysis measures number of operations
- Can we algorithmically examine the cache performance of a program?
- Yes: with the external memory model

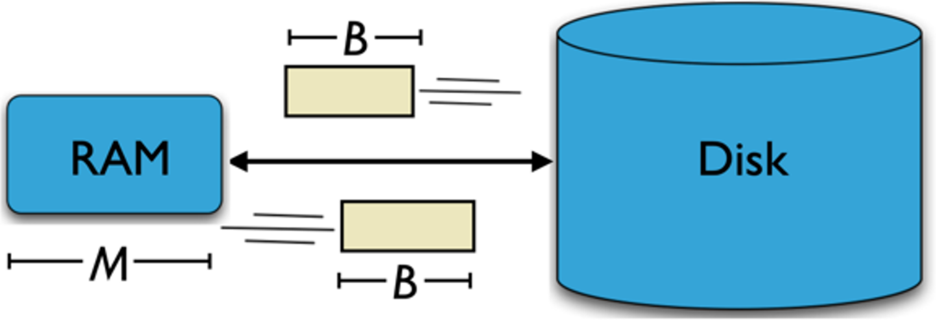
What do we want out of this model?

- Simple, but able to capture major performance considerations
- Parameters for the model? How can we make it universal across computers that may have very different cache parameters?
    - Answer: we'll use parameters. (The exact size of cache, and a cache line, can *drastically* affect algorithmic performance.)
- Do we want asymptotics? Worst case?
    - Yes!

External memory model basics

- Cache of size M
- Cache line of size B
- Computation is free: *only* count number of "cache misses." Can perform arbitrary computation on items in cache.
- We will say something like "$O(n/B)$ cache misses" rather than "$O(n)$ operations" to emphasize the model.

External Memory Model Basics



Transferring *B* *consecutive* items to/from the disk costs 1. Can only store *M* things in cache.

Memory Evictions

- Can only hold M items in cache!

- So when we bring B in, need to write B items back to disk. (We can bring them in later if we need them again)

- Assume that the computer does this optimally.
  - Reasonable; it's really good at it. Very cool algorithms behind this!

Vocabulary

- “Cache” of size *M*; “disk” of unlimited size
- With the cost of one “cache miss” can bring in *B* consecutive items
    - (Sometimes called “memory access” or “I/Os” but I will try not to use those terms.)
- These *B* items are called a “block” or a “cache line”.

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B+1$?
- Answer: $O(n)$
- The external memory model predicts the real-world slowdown of this process.
- (Actual performance is *worse* in this case: we get a slowdown of $\approx$ 30, whereas the number of nodes in a cache line is 8. I imagine that this is due to prefetching; seem to be some further optimizations internally.)

Finding the minimum element in an array

- How many cache misses in the external memory model?

- Answer: $O(n/B)$

Binary search?

- What is the recurrence for binary search in terms of number of operations?
- What is the recurrence for binary search in terms of the number of cache misses?
- Each recursive call takes 1 cache miss.
- Base case: can perform *all* operations on B items with only 1 cache miss
- Total: $O(\log_2(n/B))$ cache misses.

Fitting in Cache

- If you have a sequence of operations on a dataset of size at most M, there is no further cost so long as they all stay in cache!

- $O(M/B)$ to load the items into cache, then all computation is free

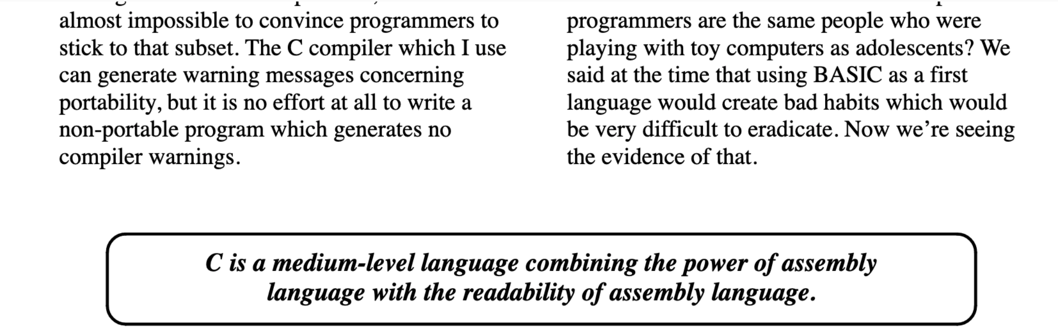
- Real-world time: what if instead of a linked list of 100 million items, we repeatedly access a linked list of 100 thousand items?

Why does the external memory model make sense?

- Simple model that captures *one level* of the memory hierarchy
- Idea: usually one level has by far the largest cost.
  - Small programs may be dominated by L1 cache misses
  - Larger programs it may be by L3 cache misses
- External memory model zooms in on one crucial level of the memory hierarchy (with particular B, M); gives asymptotics for how well we do on that level.

Question about External Memory Model Basics?

Joke to break up the material



almost impossible to convince programmers to stick to that subset. The C compiler which I use can generate warning messages concerning portability, but it is no effort at all to write a non-portable program which generates no compiler warnings.

programmers are the same people who were playing with toy computers as adolescents? We said at the time that using BASIC as a first language would create bad habits which would be very difficult to eradicate. Now we're seeing the evidence of that.

C is a medium-level language combining the power of assembly language with the readability of assembly language.

Joke to break up the material

Matrix Multiplication in External Memory

Matrix Multiplication Reminder

- Given two $n \times n$ matrices A, B
- Want to compute their product C:
- $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

Example:

$$\begin{bmatrix} 1 & 2 \\ 8 & -1 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ -2 & 7 \end{bmatrix} = \begin{bmatrix} -2 & 17 \\ 18 & 17 \end{bmatrix}$$

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 does this take?
- Assume matrices are stored in row-major order.
  - First: assume $M < n^2$ Then all fits in cache; $O(n^2/B)$ I/Os
  - What if $M > n^2$?
  - Answer: $O(n^3)$ I/Os. Every operation requires an I/O for B.

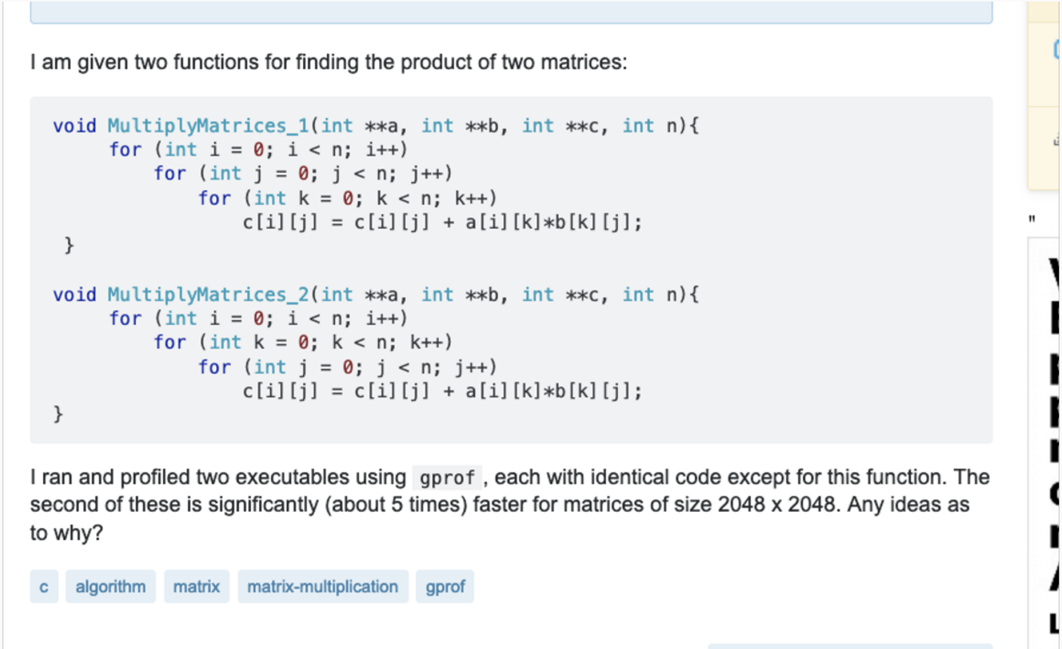
Any ideas for how to improve this?

- One idea: transpose *B* (store in column-major order)
  - A good idea; works well! A bit nontrivial, especially if you want the transposition to be cache-efficient
- Another idea: swap the loops! How many cache misses is 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 $O(n^3/B)$ cache misses. Is this actually worth doing?

Yep!



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?

We haven't used the cache yet

- No *M*s in any running times
- 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. But we did do this with `smallunsortedlinkedlist.c`

Blocking

- Standard technique for improving cache performance of algorithms.
- Idea: break problems into subproblems of size $O(M)$
  - Can solve any such problem 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 $M/3$
    - $\sqrt{M/3} \times \sqrt{M/3}$ matrices
    - Really want blocks with size $T = \lfloor \sqrt{M/3} \rfloor$. Assume that T divides n for now so there's no rounding

- Multiply blocks one at a time

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:

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

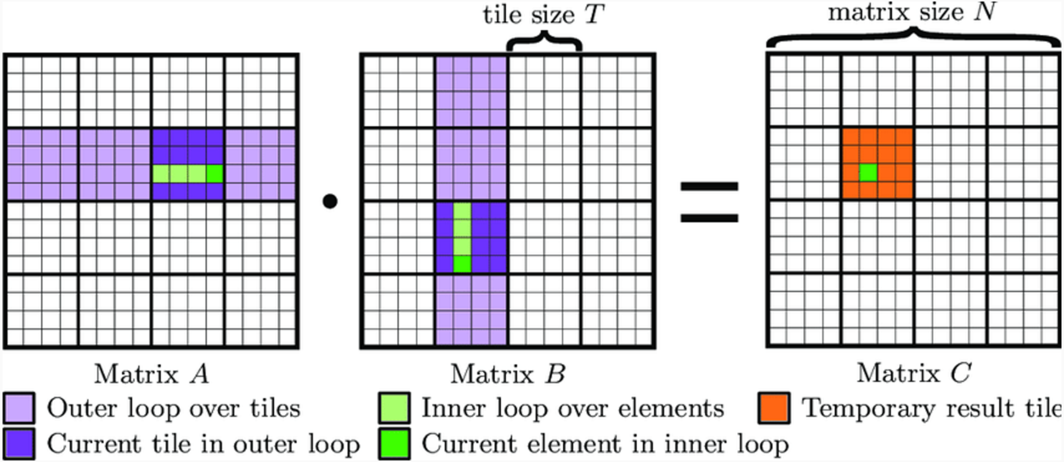
We can use this principle to multiply two larger matrices.

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

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

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

Let’s analyze the cost of this algorithm in the EM model together on the board!

Analysis

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses. If $B = O(T)$ then we can just write $O(T^2/B)$; let's assume this for simplicity.

- `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?

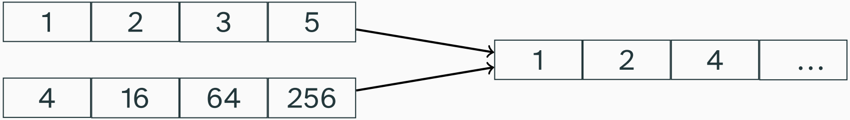
- How long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $n/B \log_2 n/B$.
- How about quicksort?
- Essentially same; partition is $O(n/B)$; total is $n/B \log_2 n/B$.
- Heapsort is $n \log_2 n/B$ unless we're careful...
- Can we do better?

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

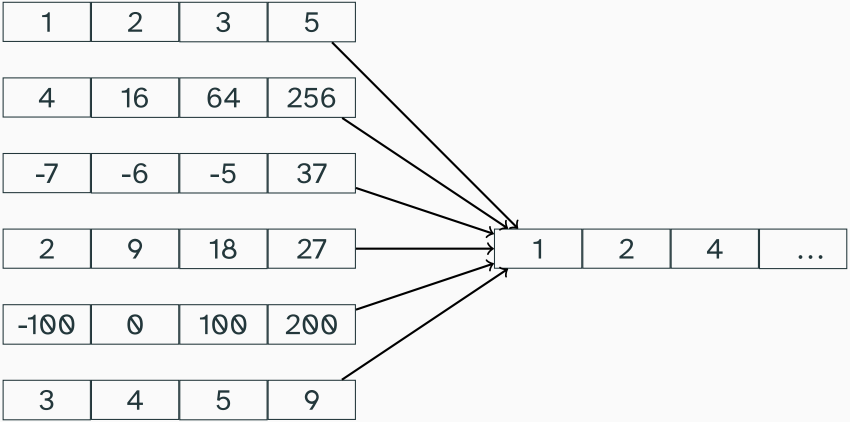
- Divide array into two equal parts
- Recursively sort both parts
- Merge them in $O(n)$ time (and $O(n/B)$ cache misses)



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



1	2	3	5
4	16	64	256
-7	-6	-5	37
2	9	18	27
-100	0	100	200
3	4	5	9

1	2	4	...

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:

$$T(N) = T(N/(M/B)) + O(N/B)T(B) = O(1)$$

- 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

- Developed to be the sorting method for python
- Now also used in Java, Rust
- Keeps cache in mind, but focuses more on taking advantage of easy patterns in data

Blocking revisited: run generation

- Basic idea: sort all M-sized subarrays. That would give us sorted subarrays of length M to start out with

- This is wasteful, as we empty out cache between each subarray

- Timsort starts with “run generation”: a greedy version of this that uses the same cache for as long as possible. Always outputs sorted runs of length at least M; can be MUCH longer

Timsort after run generation

- First, run generation
- Then, super optimized (2-way) merge sort
- Insertion sort on any very small arrays that are encountered (size < 64)

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.