# Applied Algorithms Lec 6: External Memory; Optimization

Sam McCauley

September 23, 2025

Williams College

# Admin

- Next week I will not be here:
  - We will still have lecture (Shikha will teach you about filters, probability and hashing)
  - We will *not* have lab
  - Start Assignment 2 early even though it is due next week!
- Any questions about Assignment 2?
- Today: discuss Assignment 1, external memory practice, discuss other considerations for optimization

Let’s go over Hirshberg’s, and do one (abbreviated) run on the board

# Plots

- I'm asking you to generate plots for Assignment 2

# Plots

- I'm asking you to generate plots for Assignment 2
- You can use whatever software you want

# Plots

- I'm asking you to generate plots for Assignment 2
- You can use whatever software you want
  - `matplotlib` in python is probably the best

# Plots

- I’m asking you to generate plots for Assignment 2
- You can use whatever software you want
    - `matplotlib` in python is probably the best
    - Google Sheets (or whatever) is fine

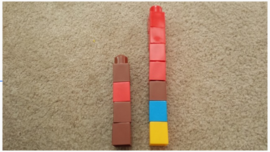
# Plots

- I’m asking you to generate plots for Assignment 2
- You can use whatever software you want
  - `matplotlib` in python is probably the best
  - Google Sheets (or whatever) is fine
- Make sure you label your plot; your axes; give units; make sure the results are legible

# Plots

- I'm asking you to generate plots for Assignment 2
- You can use whatever software you want
  - `matplotlib` in python is probably the best
  - Google Sheets (or whatever) is fine
- Make sure you label your plot; your axes; give units; make sure the results are legible
- Use `includegraphics` to put the plot into latex

# Looking back at Assignment 1



- Quick reminder of Implementation 1 vs Implementation 4
- First extra credit: use external memory model to explain why Implementation 4 was faster
- Second extra credit: how to get $O(2^{n/2})$ *operations*?
  - How many cache misses does this incur?

# What is Slow on Modern Computers Beyond Cache Misses?

# Topic for Today

- We’ve seen: cache misses take a lot of time!

# Topic for Today

- We’ve seen: cache misses take a lot of time!
- Now: What other operations are slow? What can/should we do about it?

# Topic for Today

- We've seen: cache misses take a lot of time!
- Now: What other operations are slow? What can/should we do about it?
    - Any of you have ideas to start?

# Note on time taken for each operation

# Note on time taken for each operation

- Intel used to release information along the lines of: here’s how much time it takes to add two integers; here’s how much time it takes to compare to floats.

# Note on time taken for each operation

- Intel used to release information along the lines of: here’s how much time it takes to add two integers; here’s how much time it takes to compare to floats.

- In the last couple years, Intel has stopped releasing this information!

# Note on time taken for each operation

- Intel used to release information along the lines of: here's how much time it takes to add two integers; here's how much time it takes to compare to floats.

- In the last couple years, Intel has stopped releasing this information!

- Too much else going on for strong conclusions.

# Note on time taken for each operation

- Intel used to release information along the lines of: here’s how much time it takes to add two integers; here’s how much time it takes to compare to floats.
- In the last couple years, Intel has stopped releasing this information!
- Too much else going on for strong conclusions.
- I’ll go over the numbers from a couple years ago anyway; some (very high level) lessons to be learned

# Note on time taken for each operation

- Intel used to release information along the lines of: here's how much time it takes to add two integers; here's how much time it takes to compare to floats.

- In the last couple years, Intel has stopped releasing this information!

- Too much else going on for strong conclusions.

- I'll go over the numbers from a couple years ago anyway; some (very high level) lessons to be learned

- To *know* if something is fast: run an experiment!

# Basic operations (latency)

- Integer add, multiply (bit operations, move, push, pop, etc.)

# Basic operations (latency)

- Integer add, multiply (bit operations, move, push, pop, etc.)
    - fast! 1-2 cycles

# Basic operations (latency)

- Integer add, multiply (bit operations, move, push, pop, etc.)
    - fast! 1-2 cycles
- Divide, modulo

# Basic operations (latency)

- Integer add, multiply (bit operations, move, push, pop, etc.)
    - fast! 1-2 cycles
- Divide, modulo
    - Generally pretty slow; 5-20 cycles

# Basic operations (latency)

- Integer add, multiply (bit operations, move, push, pop, etc.)
    - fast! 1-2 cycles
- Divide, modulo
    - Generally pretty slow; 5-20 cycles
- Float add, multiply?
    - Pretty fast on x86; almost as fast as integers

# Experiments

- Let's run some (really rough) experiments: `timetests.c`
- Unroll loops to minimize loop overhead; compile with optimizations off
- Why is this important? Let's look at the assembly
- Compiler explorer: recent, *super cool* tool to look at assembly for C code
  - `godbolt.org`
  - Awesome for people (like me) who aren't assembly experts but sometimes care about what exactly the computer is doing

# More complicated operations

- Square root?

# More complicated operations

- Square root?
  - fast on our machines! 1-2 cycles

# More complicated operations

- Square root?
  - fast on our machines! 1-2 cycles
- Generating a random number?

# More complicated operations

- Square root?
  - fast on our machines! 1-2 cycles
- Generating a random number?
  - Pretty slow

# More complicated operations

- Square root?
  - fast on our machines! 1-2 cycles
- Generating a random number?
  - Pretty slow
- memory allocation in bytes? `timetests2.c`

# More complicated operations

- Square root?
  - fast on our machines! 1-2 cycles
- Generating a random number?
  - Pretty slow
- memory allocation in bytes? `timetests2.c`
- memory allocation in megabytes? (So: fewer allocations, but each of larger size, for same total size)

# More complicated operations

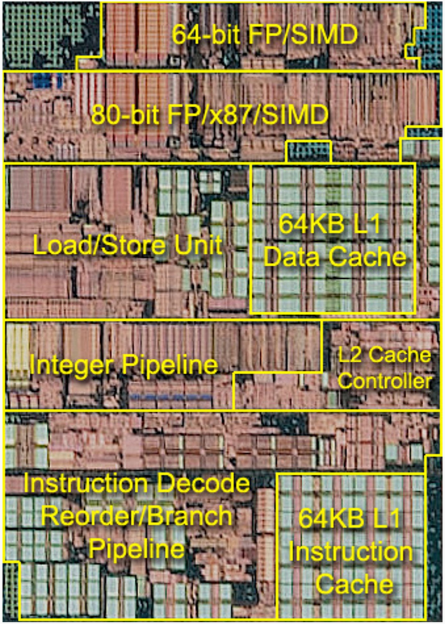
- Square root?
  - fast on our machines! 1-2 cycles
- Generating a random number?
  - Pretty slow
- memory allocation in bytes? `timetests2.c`
- memory allocation in megabytes? (So: fewer allocations, but each of larger size, for same total size)
- how does it grow as we increase the number of operations?
  - Cache efficiency is the problem here, not the memory call itself
  - (For what it's worth: `malloc` really is $O(1)$)

# Latency and Throughput

Latency vs throughput:

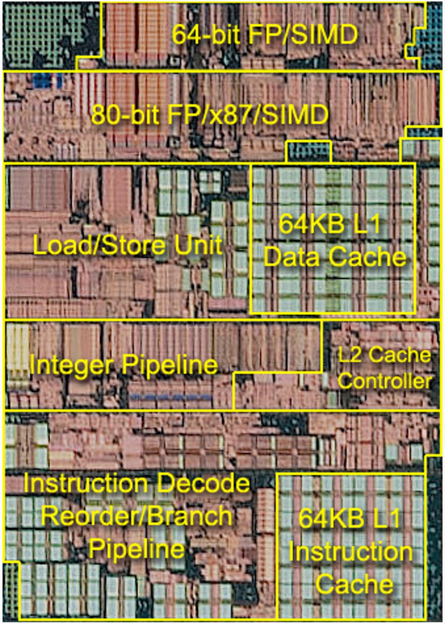
- Latency: time it takes for a sequence of *data-dependent* operations of a given type
- Throughput: time after a previous operation when a new operation of the same type can begin.
- Let’s look at an example: `latencythroughput2.c`
- When designing code, be careful of data dependencies; they can significantly affect running time
- (As an aside: this is one reason why statements like “additions take 1 clock cycle” aren’t really possible anymore)

# Modern processors



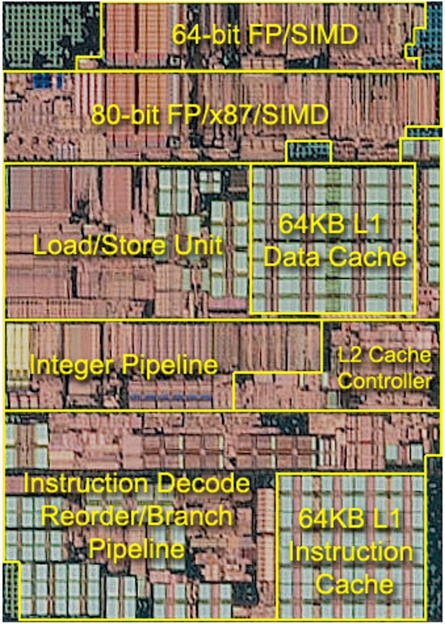

- Lots going on

# Modern processors




- Lots going on
- Moving things around takes more time than processing

# Modern processors




- Lots going on
- Moving things around takes more time than processing
- Cache costs are the most extreme example of this! But not the *only* example.

# Casts and moving data around

- Casts can be expensive if they require moving the data into another part of the processor!

# Casts and moving data around

- Casts can be expensive if they require moving the data into another part of the processor!
- (Can be free if they don't)

# Branch mispredictions, etc.

- Instructions need to be moved into the CPU

# Branch mispredictions, etc.

- Instructions need to be moved into the CPU
- Modern CPUs predict what instructions will be next; move while completing other operations

# Branch mispredictions, etc.

- Instructions need to be moved into the CPU
- Modern CPUs predict what instructions will be next; move while completing other operations
- What if the CPU gets it wrong?

# Branch mispredictions, etc.

- Instructions need to be moved into the CPU
- Modern CPUs predict what instructions will be next; move while completing other operations
- What if the CPU gets it wrong?
- “Branch misprediction:” 10-20 cycles to fetch the new instructions from memory

# Branch mispredictions, etc.

- Instructions need to be moved into the CPU
- Modern CPUs predict what instructions will be next; move while completing other operations
- What if the CPU gets it wrong?
- “Branch misprediction:” 10-20 cycles to fetch the new instructions from memory
- Can have similar issues with calling non-inlined functions (compiler is very good at avoiding this)

# Branch predictors

- CPU keeps track of your branches as it runs
  - Divides into four categories of how likely it is to be taken

# Branch predictors

- CPU keeps track of your branches as it runs
  - Divides into four categories of how likely it is to be taken
- `gcc` also predicts your branches during compilation

# Branch predictors

- CPU keeps track of your branches as it runs
    - Divides into four categories of how likely it is to be taken
- `gcc` also predicts your branches during compilation
- Can also give preprocessor directives about branches. Can be slightly helpful (one of the *absolute last* things you should do for optimization)

# Branch predictors

- CPU keeps track of your branches as it runs
  - Divides into four categories of how likely it is to be taken
- `gcc` also predicts your branches during compilation
- Can also give preprocessor directives about branches. Can be slightly helpful (one of the *absolute last* things you should do for optimization)
  - Example:

```
int i = !time(NULL);
if (__builtin_expect(i, 0))
  printf("It's not 1970!");
return 0;
```

# Branch predictors

- CPU keeps track of your branches as it runs
  - Divides into four categories of how likely it is to be taken
- `gcc` also predicts your branches during compilation
- Can also give preprocessor directives about branches. Can be slightly helpful (one of the *absolute last* things you should do for optimization)
  - Example:

```
int i = !time(NULL);
if (__builtin_expect(i, 0))
  printf("It's not 1970!");
return 0;
```

  - Predictors are so good that this rarely helps

# Avoiding branch mispredictions

```c
int max(int a, int b) {
   int diff = a - b;
   int dsgn = diff >> 31;
   return a - (diff & dsgn);
}
```

```c
int swap(int a, int b) {
   a = a ^ b;
   b = a ^ b;
   a = a ^ b;
}
```

- Avoid branches (ifs, etc.) by refactoring when possible

# Avoiding branch mispredictions

```c
int max(int a, int b) {
   int diff = a - b;
   int dsgn = diff >> 31;
   return a - (diff & dsgn);
}
```

```c
int swap(int a, int b) {
   a = a ^ b;
   b = a ^ b;
   a = a ^ b;
}
```

- Avoid branches (ifs, etc.) by refactoring when possible
- Crazy tricks often not worth it nowadays—true in general; though some exceptions (again, check this only at the very end of optimizing; only for crucial operations)

# Avoiding branch mispredictions

```
int max(int a, int b) {
   int diff = a - b;
   int dsgn = diff >> 31;
   return a - (diff & dsgn);
}
```

```
int swap(int a, int b) {
   a = a ^ b;
   b = a ^ b;
   a = a ^ b;
}
```

- cmov operations help a lot in modern processors; compilers are great at avoiding expensive branches

# Avoiding branch mispredictions

```
int max(int a, int b) {
   int diff = a - b;
   int dsgn = diff >> 31;
   return a - (diff & dsgn);
}
```

```
int swap(int a, int b) {
   a = a ^ b;
   b = a ^ b;
   a = a ^ b;
}
```

- cmov operations help a lot in modern processors; compilers are great at avoiding expensive branches
- If you do create a branch, ask yourself how easy it is to predict!

# Avoiding branch mispredictions

```
int max(int a, int b) {
   int diff = a - b;
   int dsgn = diff >> 31;
   return a - (diff & dsgn);
}
```

```
int swap(int a, int b) {
   a = a ^ b;
   b = a ^ b;
   a = a ^ b;
}
```

- cmov operations help a lot in modern processors; compilers are great at avoiding expensive branches
- If you do create a branch, ask yourself how easy it is to predict!
- Thought question: in Implementation 4, were some of the binary search branches easy to predict?

# Avoiding branch mispredictions

```
int max(int a, int b) {
   int diff = a - b;
   int dsgn = diff >> 31;
   return a - (diff & dsgn);
}
```

```
int swap(int a, int b) {
   a = a ^ b;
   b = a ^ b;
   a = a ^ b;
}
```

- cmov operations help a lot in modern processors; compilers are great at avoiding expensive branches
- If you do create a branch, ask yourself how easy it is to predict!
- Thought question: in Implementation 4, were some of the binary search branches easy to predict?
- Only way to be sure is to experiment

# Avoiding branch mispredictions

```
int max(int a, int b) {
   int diff = a - b;
   int dsgn = diff >> 31;
   return a - (diff & dsgn);
}
```

```
int swap(int a, int b) {
   a = a ^ b;
   b = a ^ b;
   a = a ^ b;
}
```

- `cmov` operations help a lot in modern processors; compilers are great at avoiding expensive branches
- If you do create a branch, ask yourself how easy it is to predict!
- Thought question: in Implementation 4, were some of the binary search branches easy to predict?
- Only way to be sure is to experiment
- `branchpredictions.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?
- 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` (needs root access to do many of the interesting things)—but can do cachegrind-like things *without simulation*
    - 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?

# gprof

- Older command line tool

# gprof

- Older command line tool
- Uses sampling to collect data

# `gprof`

- Older command line tool
- Uses sampling to collect data
- Designed to talk with `gcc` using `-pg` flag

# `gprof`

- Older command line tool
- Uses sampling to collect data
- Designed to talk with `gcc` using `-pg` flag
- Gives information about time as well as the call graph

# `gprof`

- Older command line tool
- Uses sampling to collect data
- Designed to talk with `gcc` using `-pg` flag
- Gives information about time as well as the call graph
- *Quite* limited. But in some circumstances gives good advice.
  - Recursion; function-level resolution; cannot optimize; overhead; sampling problems

# Profilers examples: gprof

- Compile with `-pg` option; then run normally; then run gprof on the executable

# 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

# 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

# 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
- Gets confusing with recursive calls

# 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
- Gets confusing with recursive calls
- I may ask you to use this, but be aware that it's useful sometimes at best

# 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
- Gets confusing with recursive calls
- I may ask you to use this, but be aware that it's useful sometimes at best
- Let's look at what it says about Assignment 1

# `callgrind` and `cachegrind`

- Features of `valgrind`

# `callgrind` and `cachegrind`

- Features of `valgrind`
- `callgrind` gives `gprof`-like profiling

# `callgrind` and `cachegrind`

- Features of `valgrind`
- `callgrind` gives `gprof`-like profiling
- `cachegrind` helps determine the cost of *moving* data: cache misses, branch mispredictions, etc.

# `callgrind` and `cachegrind`

- Features of `valgrind`
- `callgrind` gives `gprof`-like profiling
- `cachegrind` helps determine the cost of *moving* data: cache misses, branch mispredictions, etc.
- Essentially runs the program on a simulated virtual machine

# `callgrind` and `cachegrind`

- Features of `valgrind`
- `callgrind` gives `gprof`-like profiling
- `cachegrind` helps determine the cost of *moving* data: cache misses, branch mispredictions, etc.
- Essentially runs the program on a simulated virtual machine
- Gives detailed information about costs you could not otherwise get, but VERY slow.

# `callgrind` and `cachegrind`

# `callgrind` and `cachegrind`

- Let's run cachegrind on the basic backtracking version of Assignment 2

# `callgrind` and `cachegrind`

- Let's run cachegrind on the basic backtracking version of Assignment 2
- For course-grained results, compile normally; then:

```
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes
    [executable]
```

# `callgrind` and `cachegrind`

- Let's run cachegrind on the basic backtracking version of Assignment 2
- For course-grained results, compile normally; then:

```
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes
    [executable]
```

- (Turns on the simulation of the cache, *and* the simulation of branch mispredictions. Can also run one or the other if you want.)

# `callgrind` and `cachegrind`

- Let's run cachegrind on the basic backtracking version of Assignment 2
- For course-grained results, compile normally; then:

```
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes
    [executable]
```

- (Turns on the simulation of the cache, *and* the simulation of branch mispredictions. Can also run one or the other if you want.)
- After doing the above, can get more detail using:

```
cg_annotate [cachegrind file]
```

# `callgrind` and `cachegrind`

- Let's run cachegrind on the basic backtracking version of Assignment 2
- For course-grained results, compile normally; then:

```
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes
    [executable]
```

- (Turns on the simulation of the cache, *and* the simulation of branch mispredictions. Can also run one or the other if you want.)
- After doing the above, can get more detail using:

```
cg_annotate [cachegrind file]
```

- If you instead compile with debug information on, outputs a line-by-line analysis

# Optimization

With these costs in mind, how does one write code that runs quickly?

# 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?

[1] Of course, we know that we're never setting any values to 0 before checking them, but the compiler doesn't check for that.

# 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?

[1]Of course, we know that we're never setting any values to `0` before checking them, but the compiler doesn't check for that.

# 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*:[1] we're changing the array, which could change the first `0`.

[1]Of course, we know that we're never setting any values to `0` before checking them, 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;
}
```

# 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

# 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?

# 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)

# Theme of user optimizations vs compiler optimizations

- The compiler will do the best optimizations it can that work for *all* code

# Theme of user optimizations vs compiler optimizations

- The compiler will do the best optimizations it can that work for *all* code
- Bear in mind: only common optimizations are implemented

# Theme of user optimizations vs compiler optimizations

- The compiler will do the best optimizations it can that work for *all* code
- Bear in mind: only common optimizations are implemented
- Opportunities for you: what do you know about your data, and about your methodology, that allows for further efficiency?

# Loop Unrolling

- Classic technique to improve loop efficiency

# Loop Unrolling

- Classic technique to improve loop efficiency
- What are the costs of each iteration of a simple for loop?

```
for(int x = 0; x < 1000; x++){
  total += array[x];
}
```

# Loop Unrolling

```
for(int x = 0; x < 1000; x++){
  total += array[x];
}
```

- Need to do a branch every loop

# Loop Unrolling

```
for(int x = 0; x < 1000; x++){
  total += array[x];
}
```

- Need to do a branch every loop
- Instruction pointer jump every loop (cost of “jumping back” varies; outside scope of course)

# Loop Unrolling

```
for(int x = 0; x < 1000; x++){
   total += array[x];
}
```

- Need to do a branch every loop
- Instruction pointer jump every loop (cost of “jumping back” varies; outside scope of course)
- Need to compare every loop

# Loop Unrolling

```
for(int x = 0; x < 1000; x++){
   total += array[x];
}
```

- Need to do a branch every loop
- Instruction pointer jump every loop (cost of “jumping back” varies; outside scope of course)
- Need to compare every loop
- Need to increment every loop

# Unrolled Loop

```
for(int x = 0; x < 1000; x+=5){
   total += array[x];
   total += array[x+1];
   total += array[x+2];
   total += array[x+3];
   total += array[x+4];
}
```

- In short: repeat body of the loop multiple times.
- What does this gain us?

# Unrolled Loop

```
for(int x=0; x < 1000; x++){
   total += array[x];
}
```

- Branch every loop
- Instruction pointer jump every loop
- Compare every loop
- Increment every loop

```
for(int x=0; x<1000; x+=5){
   total += array[x];
   total += array[x+1];
   total += array[x+2];
   total += array[x+3];
   total += array[x+4];
}
```

- Branch every 5 loops
- Instruction pointer jump every 5 loops
- Compare every 5 loops
- Increment every 5 loops (?) Need to do some *extra* additions however

# What did we need to know to make this substitution?

- Needed array size to be a multiple of 5

# What did we need to know to make this substitution?

- Needed array size to be a multiple of 5

- Can get around this with some extra work

# Disadvantages of Loop Unrolling?

- Seems like we break even at worst?

# Disadvantages of Loop Unrolling?

- Seems like we break even at worst?
- But: Loop unrolling increases code size

# Disadvantages of Loop Unrolling?

- Seems like we break even at worst?

- But: Loop unrolling increases code size

- Can hurt performance if important parts of code no longer fit in cache

# Disadvantages of Loop Unrolling?

- Seems like we break even at worst?
- But: Loop unrolling increases code size
- Can hurt performance if important parts of code no longer fit in cache
- *Fetching instructions* can require cache misses!

# Disadvantages of Loop Unrolling?

- Seems like we break even at worst?
- But: Loop unrolling increases code size
- Can hurt performance if important parts of code no longer fit in cache
- *Fetching instructions* can require cache misses!
  - We saw this in the output of `cachegrind`

# Automatic loop unrolling?

- Why can't `gcc` unroll our loops?

# Automatic loop unrolling?

- Why can't `gcc` unroll our loops?
- It can!

# Automatic loop unrolling?

- Why can't `gcc` unroll our loops?
- It can!
- Need to turn on specifically (not enabled at *any* optimization level)

`-funroll-loops`
Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. `-funroll-loops` implies `-frerun-cse-after-loop`. This option makes code larger, and may or may not make it run faster.

`-funroll-all-loops`
Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly. `-funroll-all-loops` implies the same options as `-funroll-loops`,

# Automatic loop unrolling?

- Why can't `gcc` unroll our loops?
- It can!
- Need to turn on specifically (not enabled at *any* optimization level)

`-funroll-loops`
: Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. `-funroll-loops` implies `-frerun-cse-after-loop`. This option makes code larger, and may or may not make it run faster.

`-funroll-all-loops`
: Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly. `-funroll-all-loops` implies the same options as `-funroll-loops`,

- `-O3` does a specific kind of unrolling of nested loops

# Compiler optimizations?

- We've stumbled upon a classic (and thematic) problem in optimization: time vs space of the machine code itself

# Compiler optimizations?

- We've stumbled upon a classic (and thematic) problem in optimization: time vs space of the machine code itself

- Many optimizations of code reduce the number of operations (or their total time), but increase the size of the code itself—potentially leading to cache misses

# Revisiting compiler flags

- `-O0`: No optimizations
- `-O1`: Some optimizations; may take longer to compile than `-O0`
- `-O2`: Turns on “nearly all” optimizations *that do not involve a space-time tradeoff*
- `-O3`: More optimizations. May lead to larger final programs
- `-Ofast`: Even more optimizations. Most notable is reordering floating point operations (can lead to correctness issues)

# Optimizations and this course

- Our projects generally involve really small programs. This is why the very optimized versions (using -O3) tend to work well for your code.

# Optimizations and this course

- Our projects generally involve really small programs. This is why the very optimized versions (using -O3) tend to work well for your code.
- Not advised in general

# Optimizations and this course

- Our projects generally involve really small programs. This is why the very optimized versions (using `-O3`) tend to work well for your code.
- Not advised in general
- Example: Gentoo user manual. (Gentoo is a linux distribution in which *all* software is compiled from scratch. So this is advice for people compiling large software like the linux kernel, chromium, libreoffice, etc. (as well as, of course, very small utilities like git))

# Gentoo optimization advice

- `-O1`: the most basic optimization level. The compiler will try to produce faster, smaller code without taking much compilation time. It is basic, but it should get the job done all the time.
- `-O2`: A step up from `-O1`. The *recommended* level of optimization unless the system has special needs. `-O2` will activate a few more flags in addition to the ones activated by `-O1`. With `-O2`, the compiler will attempt to increase code performance without compromising on size, and without taking too much compilation time. SSE or AVX may be utilized at this level but no YMM registers will be used unless `-ftree-vectorize` is also enabled.
- `-O3`: the highest level of optimization possible. It enables optimizations that are expensive in terms of compile time and memory usage. Compiling with `-O3` is not a guaranteed way to improve performance, and in fact, in many cases, can slow down a system due to larger binaries and increased memory usage. `-O3` is also known to break several packages. Using `-O3` is not recommended. However, it also enables `-ftree-vectorize` so that loops in the code get vectorized and will use AVX YMM registers.

# One more common optimization with a time-space tradeoff

- We’ve talked about how costly it is to call a function

# One more common optimization with a time-space tradeoff

- We’ve talked about how costly it is to call a function
- Well, most of the time, we don’t really *need* function calls at all, do we? If the function doesn’t call another function, can just put the code for the function directly into the code

# One more common optimization with a time-space tradeoff

- We’ve talked about how costly it is to call a function
- Well, most of the time, we don’t really *need* function calls at all, do we? If the function doesn’t call another function, can just put the code for the function directly into the code
- Called *function inlining*

# One more common optimization with a time-space tradeoff

- We've talked about how costly it is to call a function
- Well, most of the time, we don't really *need* function calls at all, do we? If the function doesn't call another function, can just put the code for the function directly into the code
- Called *function inlining*
- Tradeoff?

# Function Inlining

- Can do it yourself. May not be a good idea. (Makes code harder to read.)

# Function Inlining

- Can do it yourself. May not be a good idea. (Makes code harder to read.)
- `gcc` will judge each function for you and inline it if `gcc` thinks it's a good idea (flag to get `gcc` to do this is `-finline-functions`; it is turned on with `-O2`)

# Function Inlining

- Can do it yourself. May not be a good idea. (Makes code harder to read.)
- `gcc` will judge each function for you and inline it if `gcc` thinks it's a good idea (flag to get `gcc` to do this is `-finline-functions`; it is turned on with `-O2`)
- Can use `inline` keyword. `gcc` will try particularly hard to inline it for you, and if it can't will tell you if you have `-Winline` flag on
  - Can use `__inline__`; does the same thing. Some compilers may like this better
  - Probably want to always use `static inline`

# Function Inlining

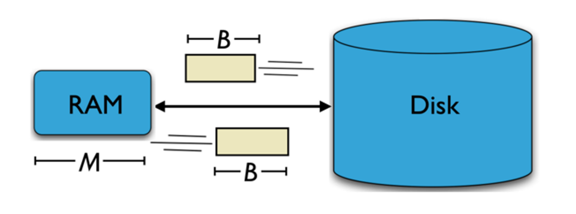
- Can do it yourself. May not be a good idea. (Makes code harder to read.)
- `gcc` will judge each function for you and inline it if `gcc` thinks it's a good idea (flag to get `gcc` to do this is `-finline-functions`; it is turned on with `-O2`)
- Can use `inline` keyword. `gcc` will try particularly hard to inline it for you, and if it can't will tell you if you have `-Winline` flag on
  - Can use `__inline__`; does the same thing. Some compilers may like this better
  - Probably want to always use `static inline`
- Can also use `__attribute__((always_inline))` which really forces it to inline even if optimizations are turned off

# One more optimization flag

- `-march=native`
- tells `gcc` to use instructions specific to this processor. May increase speed
- Only disadvantage: your compiled binary may not run on other computers unless they have an identical processor (this is not a problem for us!)

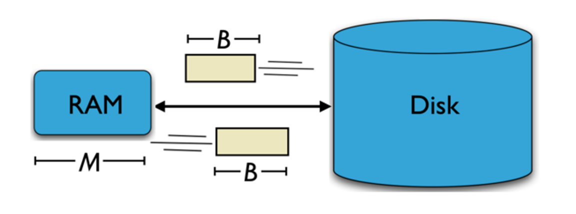
# Sorting in External Memory

# External Memory Model Reminder

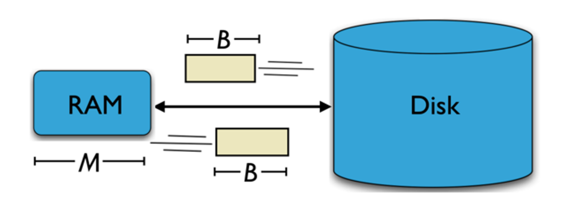



- Cache line of size *B*
- Cache can hold at most *M* elements
- Computing on elements in cache is free! Only cost is to bring things in and out of cache

# External Memory Model Reminder

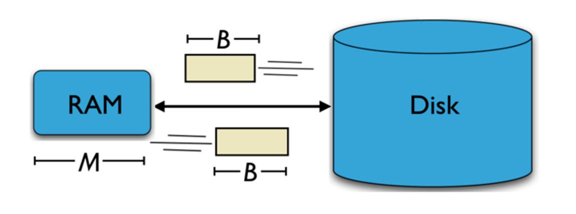


# External Memory Model Reminder



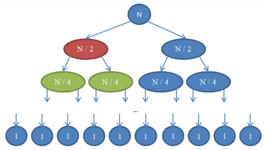

- Let's say I want to find the maximum element in an array using a linear scan. How much time does that take?

# External Memory Model Reminder



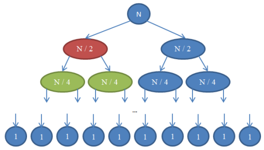

- Let's say I want to find the maximum element in an array using a linear scan. How much time does that take?
    - $O(n/B)$ cache misses: if I have a cache miss accessing $A[i]$, my next cache miss is accessing $A[i+B]$.

# What about algorithms we know?



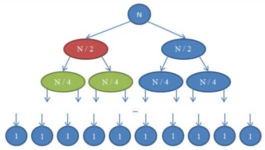

- In pairs: how long does Mergesort take in external memory?

# What about algorithms we know?



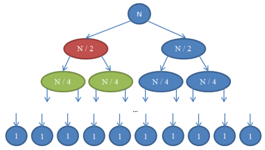

- In pairs: how long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $\mathrm{O}(n/B \log_2 n/B)$.

# What about algorithms we know?



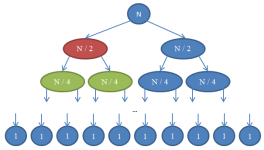

- In pairs: how long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $\mathrm{O}(n/B \log_2 n/B)$.
- How about quicksort?

# What about algorithms we know?



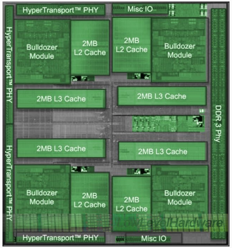

- In pairs: how long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $\mathrm{O}(n/B \log_2 n/B)$.
- How about quicksort?
- Essentially same; partition is $O(n/B)$; total is $O(n/B \log_2 n/B)$.

# What about algorithms we know?



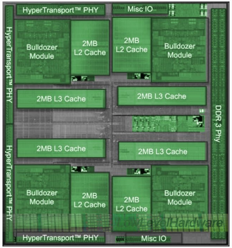

- In pairs: how long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $\mathrm{O}(n/B \log_2 n/B)$.
- How about quicksort?
- Essentially same; partition is $O(n/B)$; total is $O(n/B \log_2 n/B)$.
- Seems pretty good! Can we do better?

# Using the cache




- Blocking? A little unclear. (We'll come back to this.)

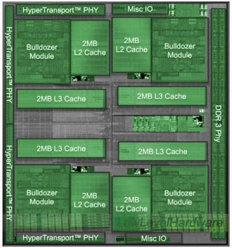
# Using the cache




- Blocking? A little unclear. (We'll come back to this.)

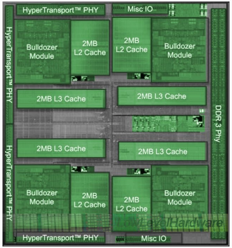
- Does anyone know the sorting lower bound? Where does $n \log n$ come from?

# 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.

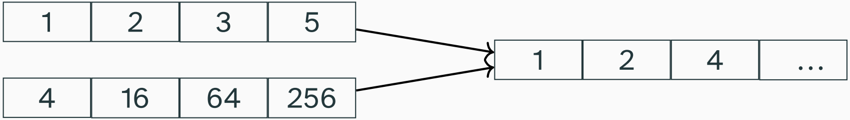
# Using the cache




- 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

# $M/B$-way merge sort

- Divide array into $M/B$ equal parts

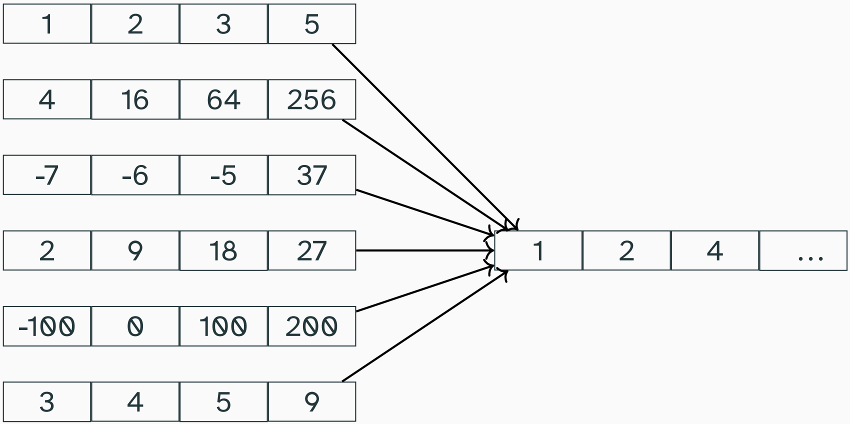
# $M/B$-way merge sort

- Divide array into $M/B$ equal parts
- Recursively sort all $M/B$ parts

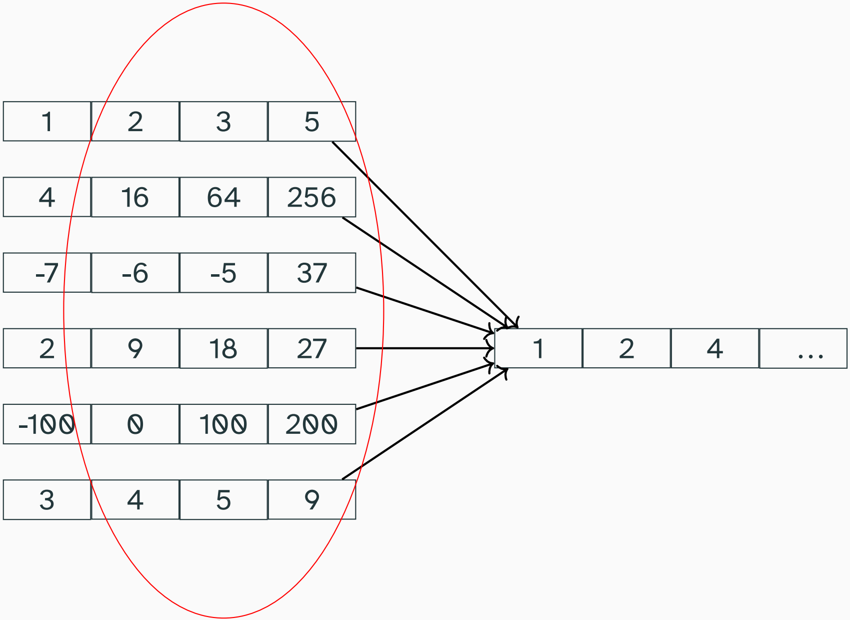
# $M/B$-way merge sort

- 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)

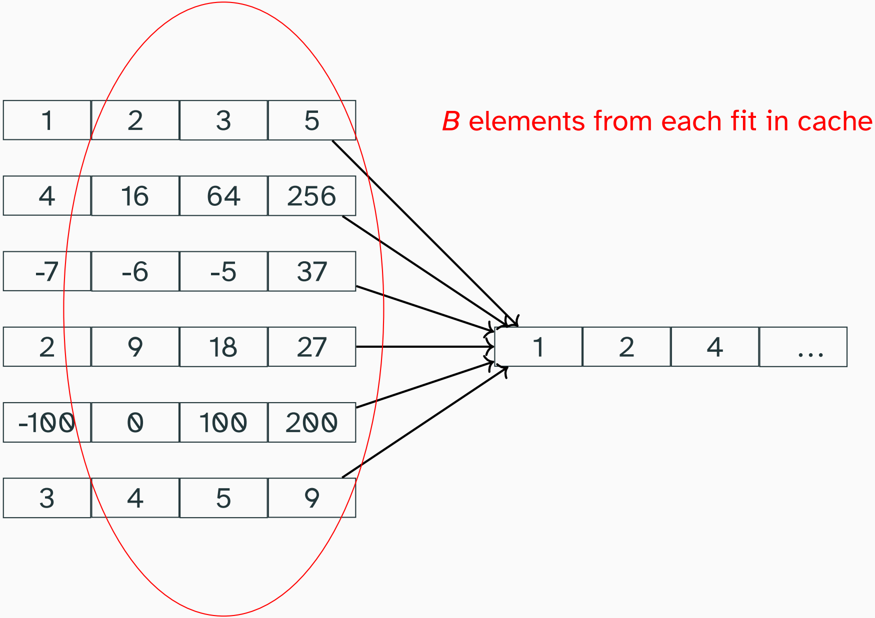
# Diagram of $M/B$-way merge sort



# Diagram of $M/B$-way merge sort



# 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!)

# 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.

# 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.

# Analysis

- Divide array into $M/B$ parts; combine in $O(N/B)$ cache misses.
- Recursion:
$$T(N) = T(N/(M/B)) + O(N/B)$$
$T(B) = O(1)$

# 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 $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ cache misses (use recursion tree method)

# 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 $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ cache misses (use recursion tree method)
- Optimal!

# Useful?

- Can be useful if your data is VERY large

# Useful?

- Can be useful if your data is VERY large
- Distribution sort: similar idea, but with Quicksort instead of Mergesort

# 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: Powersort

# 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: Powersort
- New; invented in 2018 to improve on Timsort

# 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: Powersort
- New; invented in 2018 to improve on Timsort
- Merges a “stack” of runs. Somewhat similar to $M/B$-way merge sort, achieves strong cache efficiency in practice.

# 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: Powersort
- New; invented in 2018 to improve on Timsort
- Merges a “stack” of runs. Somewhat similar to $M/B$-way merge sort, achieves strong cache efficiency in practice.
- Takes advantage of already-sorted portions of the array

# 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: Powersort
- New; invented in 2018 to improve on Timsort
- Merges a “stack” of runs. Somewhat similar to $M/B$-way merge sort, achieves strong cache efficiency in practice.
- Takes advantage of already-sorted portions of the array
- When you call sort in python, it is either Timsort or Powersort