CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 24: Introduction to Performance Programming

Instructors: Dan Garcia, Justin Yokota

Agenda

- Introduction
- Timing
- Amdahl's Law
- Case Study: Matrix Multiplication
- Single-threaded improvements

Introduction

- Part 1 of this course: How a computer works
  - C->RISC-V->circuit level
- Part 2: How to optimize a program
  - Pipelining
  - Parallelism
  - Caching/VM
- Today: Optimizations we can make without parallelism

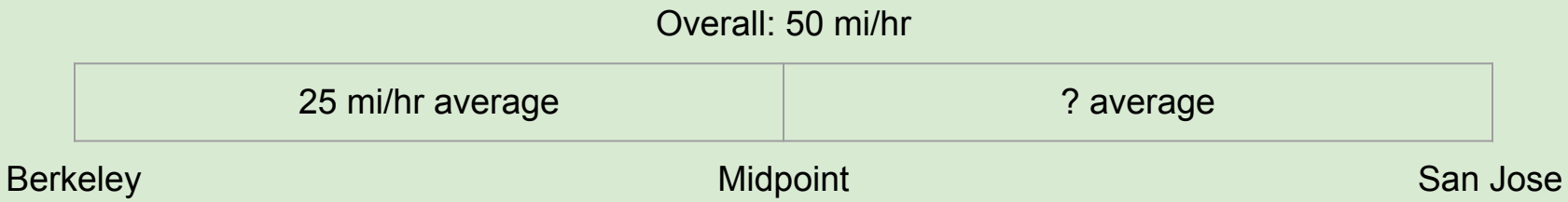
Measuring Runtime: The `<time.h>` library

- Human time is weird
  - Leap years and leap seconds are inconsistent, and hard to work with.
- Most systems use epoch time, defined as the number of seconds since some historical event
- The most common epoch is Unix time, which starts January 1, 1970, 00:00:00 GMT
  - Using this time with a 32-bit signed integer causes an overflow in January 2038, which may cause a large number of bugs in 32-bit programs…
- C has several functions for this, in addition to a `clock()` function, which measures the amount of time from the start of a program
- As a reminder, green slides are not considered in scope, but provide useful background information

Measuring Runtime: The `<time.h>` library

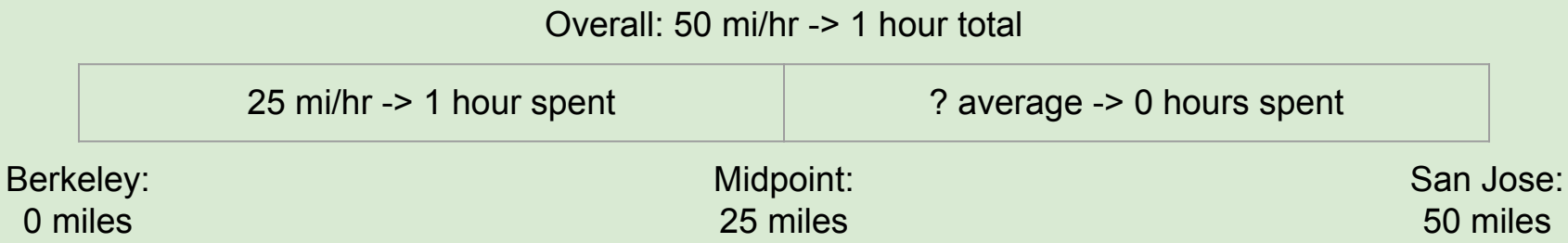
- General approach for measuring a program's runtime:
- Step 1: Before you run the program, call `clock` to get the start time of your program
- Step 2: Run your program
- Step 3: Call `clock` again and take the difference. To get the runtime in seconds, divide by `CLOCKS_PER_SECOND`
  - Make sure you typecast to a double or float; otherwise, your time will be an integer.
- This still ends up rounding results to the nearest millisecond and random lag spikes can occur, so it's sometimes useful to run the same test multiple times and take the average.
- Often useful to time individual parts of your program separately; this lets you see which parts of your code are taking the most runtime

Amdahl's Law: Analogy



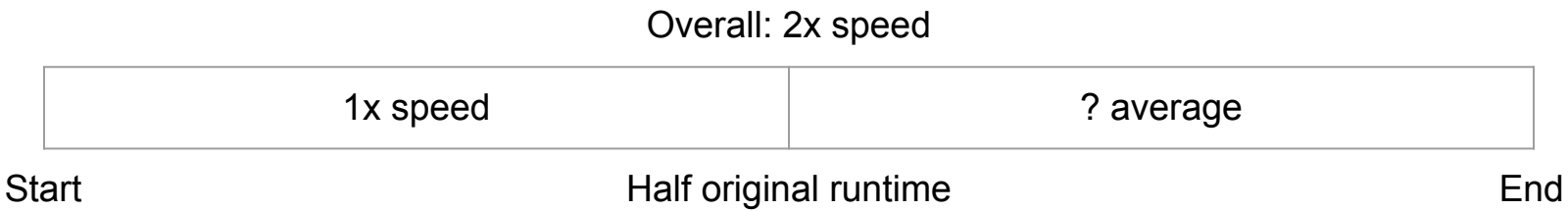
- You're driving from Berkeley to San Jose.
- For the first half of the distance, you average 25 miles/hour.
- How fast do you need to travel for the second half, in order to average 50 miles/hour overall?

Amdahl's Law: Analogy



- Assume Berkeley-San Jose is 50 miles (the math works out regardless of the actual distance, so we can just pick a distance)
- First half: 25 miles at 25 mph = 1 hour
- Overall: 50 miles at 50 mph = 1 hour
- Time left for second half: 25 miles in 0 hours -> We need to travel at ∞ mph

Amdahl's Law



- You're trying to speed up a program.
- The first half of the code can't be sped up.
- How many times faster do you need to make the second half of the code, if you want to overall get a 2x speedup?
  - Infinitely faster!

Amdahl's Law

- AKA the bane of performance programming
- "The maximum speedup we can attain with our code is limited by the fraction that cannot be sped up"
  - If we speed up 95% of our code, we can get at most a 20x speedup
- Almost any optimization we discuss from now on will only affect one portion of your code. If you only focus on one part, you'll eventually be unable to continue optimizing.
- Formal equation: Speedup = 1/((1-F)+F/S), where F is % of code that you speed up and S is how many times faster you make that part.
  - That's annoying to work with; much easier to treat these problems as word problems instead

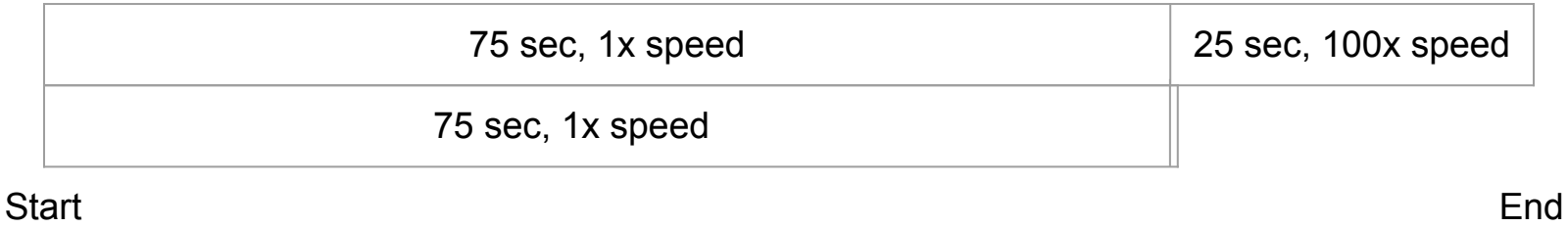
Amdahl's Law: Example

- You have an optimization that speeds up the foo function by 100x
- Unfortunately, the runtime of foo was only 25% of our original code's runtime. How many times faster have we made our code overall?

Amdahl's Law: Example

- Strategy one: Use the formula
- 1/((1-F)+F/S) -> 1/((1-0.25)+0.25/100) = 1/(0.7525) ≈ 1.33x speedup

Amdahl's Law: Example



- Strategy two: Assign values
- Let's say that our original code took 100 seconds to run
- Total runtime is now 75 seconds for part 1, .25 seconds for part 2 = 75.25 seconds
- Speedup = 100 seconds / 75.25 seconds ≈ 1.33x speedup

Amdahl's Law in the Real World

- Excerpts from FY 2021 US Federal budget:
  - "The Budget proposes to eliminate nearly $2 million for duplicative Government-funded online English-language learning programs"
  - "The Budget invests $15.1 billion in DOD's tactical fighter programs, continuing the procurement of F-35A stealth fighters and [other stealth fighters]"
- Eliminating small items tends to be easy. Cutting large items tend to be hard.
- Cut of $2 million affects budget 7,500x less than $15,100 million fighter jet budget
- Citation: BUDGET-2021-BUD.pdf (govinfo.gov)

Amdahl's Law: Consequences

- In order to properly speed up your code, you need to know which parts of your code are taking the runtime
- Test your code to help analyze where your runtime's going
  - Check multiple sizes, check repeatedly (since there might be variation between runs)
  - Check each component independently
    - Our autograders will only give you overall speedup, which doesn't help much in determining where you can speed things up further.

Amdahl's Law: Consequences

- In order to properly speed up your code, you need to know which parts of your code are taking the runtime
- Test your code to help analyze where your runtime's going
  - Check multiple sizes, check repeatedly (since there might be variation between runs)
  - Check each component independently
    - Our autograders will only give you overall speedup, which doesn't help much in determining where you can speed things up further.

Case Study: Matrix Multiplication

- Given 2 matrices A, B, return AB
- For simplicity, assume for now:
  - Each matrix is square
  - Each matrix is row-major
  - Matrix dimensions are multiples of some large power of 2
  - Strassen Algorithm doesn't exist. We're stuck with basic $O(n^3)$ matrix multiplication
- Matrix multiply forms the basis of many complex algorithms (ex. neural networks, component analysis)

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

How many times faster is C's version of matrix multiply, compared to Python's version of matrix multiply?

- <2x faster
- 2-10x faster
- 10-100x faster
- 100-1000x faster
- >1000x faster

How many times faster is C's version of matrix multiply, compared to Python's version of matrix multiply?

- <2x faster
- 2-10x faster
- 10-100x faster
- 100-1000x faster
- >1000x faster

How many times faster is C's version of matrix multiply, compared to Python's version of matrix multiply?

- <2x faster
- 2-10x faster
- 10-100x faster
- 100-1000x faster
- >1000x faster

Version 1: Matrix Multiply in C


```
jusym@JY MINGW64 ~/Documents/cs61csp23/Parallelism
$ gcc -o matmul.exe matmul.c

jusym@JY MINGW64 ~/Documents/cs61csp23/Parallelism
$ ./compareruntimes 100
C Runtime: 0.002000 seconds
Python Runtime: 0.0891256332397461 seconds

jusym@JY MINGW64 ~/Documents/cs61csp23/Parallelism
$ ./compareruntimes 200
C Runtime: 0.016000 seconds
Python Runtime: 0.6416308879852295 seconds

jusym@JY MINGW64 ~/Documents/cs61csp23/Parallelism
$ ./compareruntimes 300
C Runtime: 0.053000 seconds
Python Runtime: 2.25158429145813 seconds
```

Version 1: Matrix Multiply in C

- Just by switching from Python to C, we get a ~40x speedup
  - This is actually better than it used to be; Python 3.11 came out last semester, and sped up Python by a LOT.
  - Downside: It is hard to write code in C.
- Is this enough?
  - NO. WE NEED MOAR SPEED!
  - Also, Python's numpy library (which uses a C library) still beats us by a factor of 10
- How do we improve our runtime?

Operation Time Costs

- As a general rule of thumb, the runtime cost of operations gets broken down as follows:
- File operations
  - Extremely slow (retrieving from disk, so it takes a long time)
  - Also includes prints and other I/O, so it's a good idea to avoid printing lots of data when testing runtime.
- Memory operations
  - ~100x faster than file operations (accessing RAM)
  - Can be improved through caching, but memory operations still take 10-100 clock cycles
- Branches and Jumps
  - Slower than most operations due to hazards (potentially 5 clock cycles)
  - Can be improved through loop unrolling
- Arithmetic operations
  - Fastest operations (1 cycle ideally, though some operations take longer)

Operation Time Costs in Matrix Multiply

- File operations
  - $\Theta(N^2)$ operations, no real way to optimize this
- Memory operations
  - $\Theta(N^3)$ operations; can be reduced by constant factors
- Branches, Jumps, and Arithmetic operations
  - $\Theta(N^3)$ operations; can be reduced by constant factors
- Since the number of memory operations and Branch/Jump/Arithmetic operations are similar, memory operations will likely take the majority of runtime. Thus saving memory operations will have a larger impact on our runtime, even if it increases calculation steps.

Optimization 1: Loop Unrolling and Function Inlining

- Main idea: Since branches and jumps take a long time, reduce the number of jumps/branches needed to run a program
- Function inlining: Replace a function call with the body of that function
- Loop unrolling: Increase the number of steps done per iteration of a loop

```
int f(int i) {return i*i;}

for(int i = 0; i < max; i++)
{
    arr[i] = f(i);
}
```

```
int i = 0;
for(; i < max/4; i+=4) {
    arr[i] = i * i;
    arr[i+1] = (i+1) * (i+1);
    arr[i+2] = (i+2) * (i+2);
    arr[i+3] = (i+3) * (i+3);
}
for(; i < max; i++) {
    arr[i] = i * i;
}
```

Optimization 1: Loop Unrolling and Function Inlining

- Main idea: Since branches and jumps take a long time, reduce the number of jumps/branches needed to run a program
- Function inlining: Replace a function call with the body of that function
  - Significant reductions in runtime due to not having to set up the stack, deal with calling convention, etc.
  - Can also request the compiler inline a function for you with the `inline` keyword; i.e. `inline int f(int i)`
- Loop unrolling: Increase the number of steps done per iteration of a loop
  - Reduces runtime due to fewer branches, and can have a surprisingly large impact
  - Often done automatically by the gcc compiler, but manually unrolling can improve runtime over the automated system

Loop Unrolling and Function Inlining Limitations

- Generally causes your code file/resulting executable to be larger
- If your loop is too large, you end up exceeding your branch immediate, and take a large penalty to runtime
- The `inline` keyword is only a *suggestion* to the compiler; gcc is perfectly free to ignore the keyword entirely.
- Loop unrolling is already done by some compilers, so speedup may be minimal/hard to predict
- Major: Loop unrolling makes your code much harder to read and modify. Thus it's generally best to only unroll at the end, or to save a rolled version of your code for later use.

Optimization 2: Variable Caching

- Main idea: Save commonly used values in variables, instead of forcing recomputations
- Additionally, save commonly used variables in registers, instead of saving them on the stack. The `register` keyword can be used to request a variable gets put in a register
  - x86 is the assembly language commonly used by PCs, and has fewer registers than RISC-V does. Thus, most variables get stored in the stack, instead of in registers.

```
for(int i = 0; i < max/4;
i++)
{
    f(i);
}
```

```
register int i = 0;
register int maxoverfour =
max/4;
for(; i < maxoverfour; i++) {
    f(i);
}
```

Variable Caching Limitations

- Limited by the number of registers available
- Since different CPUs can have different registers, this makes your performance more dependent on the architecture than before, which means:
  - You need to know how the architecture works to optimize your code
  - Your code will work slower on a different architecture
- As before, the compiler tends to do some of this already, so getting speedup can be inconsistent. The `register` keyword is also only a suggestion, and the compiler is free to ignore the keyword.

Optimization 3: Data Caching

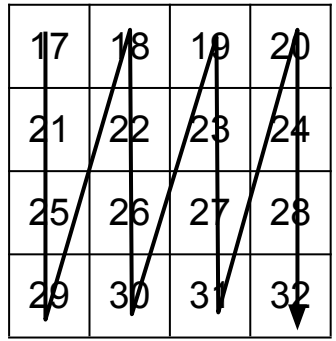
- Variable caching: Save commonly used variables in registers
- Accessing main memory takes hundreds of clock cycles
- Data caching: Save commonly used data "closer" to the CPU, so we can access it in fewer cycles (~10 cycles)
- Even better: Predict what data you need and bring that data closer to the CPU before we even access it
- The exact mechanics of this are complicated enough that we have a full lecture sequence on them: see Caches
- For today: accessing adjacent memory addresses will be on average faster than accessing nonadjacent memory addresses

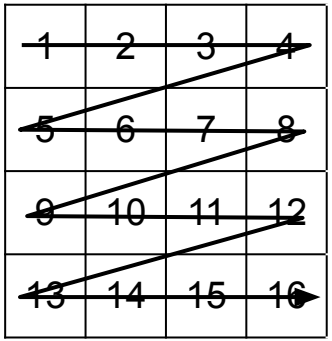
Data Caching Example

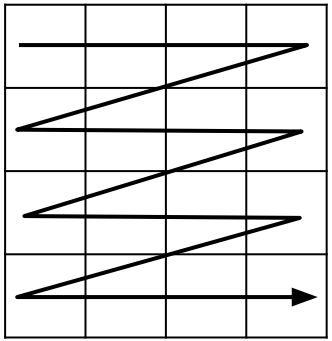

```
Test 91: 0.018000 seconds for adjacent, 0.187000 seconds for nonadjacent
Test 92: 0.015000 seconds for adjacent, 0.138000 seconds for nonadjacent
Test 93: 0.017000 seconds for adjacent, 0.168000 seconds for nonadjacent
Test 94: 0.034000 seconds for adjacent, 0.240000 seconds for nonadjacent
Test 95: 0.015000 seconds for adjacent, 0.274000 seconds for nonadjacent
Test 96: 0.019000 seconds for adjacent, 0.167000 seconds for nonadjacent
Test 97: 0.009000 seconds for adjacent, 0.143000 seconds for nonadjacent
Test 98: 0.011000 seconds for adjacent, 0.173000 seconds for nonadjacent
Test 99: 0.014000 seconds for adjacent, 0.221000 seconds for nonadjacent
Average adjacent test time: 0.016570 seconds
Average nonadjacent test time: 0.168620 seconds
```

Data Caching for Matrix Multiplication

- Assuming we do a naive matrix multiplication, data gets accessed in row-major order for two matrices, and column-major order in one matrix
- Thus, really fast to access two matrices, really slow to access the third.
- Optimization: Transpose the third matrix before starting calculations, so we get fast accesses on all matrices

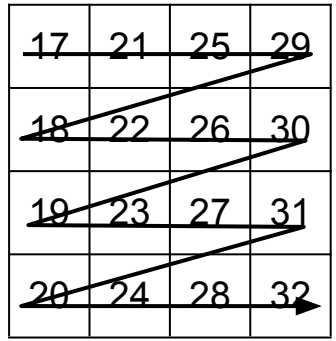


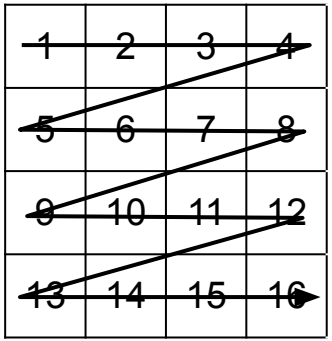


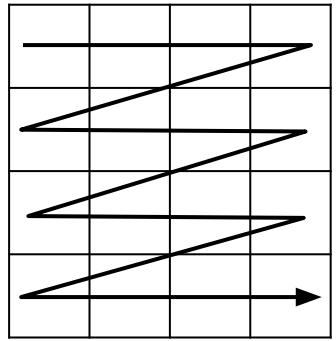


Data Caching for Matrix Multiplication

- Assuming we do a naive matrix multiplication, data gets accessed in row-major order for two matrices, and column-major order in one matrix
- Thus, really fast to access two matrices, really slow to access the third.
- Optimization: Transpose the third matrix before starting calculations, so we get fast accesses on all matrices







Data Caching for Matrix Multiply

- Optimization: Transpose the third matrix before starting calculations, so we get fast accesses on all matrices
- Transposing adds $\Theta(N^2)$ slow memory accesses, but converts $\Theta(N^3)$ slow memory accesses into fast memory accesses. Thus, this should be faster for large matrices, but slower for small matrices.
- Any way to get the best of both worlds?
  - Decide on a threshold; if the size of the matrix is smaller than the threshold, use the naive approach, and if the size of the matrix is larger than the threshold, use the transpose approach
- How to decide threshold?
  - Run tests on a representative machine to find where the break-even point is.

How many times faster is this optimized matrix multiply, compared to the matrix multiply from earlier?

- <2x faster
- 2-10x faster
- 10-100x faster
- 100-1000x faster
- >1000x faster

How many times faster is this optimized matrix multiply, compared to the matrix multiply from earlier?

- <2x faster
- 2-10x faster
- 10-100x faster
- 100-1000x faster
- >1000x faster

How many times faster is this optimized matrix multiply, compared to the matrix multiply from earlier?

- <2x faster
- 2-10x faster
- 10-100x faster
- 100-1000x faster
- >1000x faster