Sequential Warm-Up

ICS632: Principles of High Performance Computing

Henri Casanova (henric@hawaii.edu)

Fall 2015

Disclaimer

- If you have taken my ICS432 course, this will seem familiar but much quicker
- You may have seen this material in undergraduate courses elsewhere
- Each time I teach this course some students are not familiar with this content, so stop me if it goes too fast

Outline

High Performance Computing and Parallelism

- A main theme in any HPC course is parallel computing
- Parallelism is necessary for performance, and occurs at many levels in current systems
  - Within functional units (e.g., pipelining, vectorization)
  - Across functional units (e.g., add and multiply at the same time)
  - Across processor cores (e.g., using multi-threading)
  - Across processors (e.g., using CPU and GPU simultaneously)
  - Across compute nodes (e.g., using “parallel computing”)
- This is because a single sequential CPU is limited in performance by the laws of physics...

The World’s Fastest Computer

- The Top500 (`http://www.top500.org`) site gives bi-annual lists of the “““““fastest”””” 500 supercomputers
- The November 2014 list’s top machine is: Tianhe-2 (Guangzhou, China)
  - Massive parallelism (3,120,000 cores!)
  - Massive power consumption (17.8 MWatts)
    - Submarine nuclear reactor
  - Theoretical peak performance: 53,902 TFlop/s
    - Flop: Floating Point Operations
- Peak Performance: unachievable Flop number assuming all functional units work full tilt
  - Tianhe-2 achieves 61% of the peak on an “easy” linear system solve benchmark
- Could we build an equivalent sequential computer?

The World’s Fastest Computer

- The Top500 (`http://www.top500.org`) site gives bi-annual lists of the ““““““fastest”””””” 500 supercomputers
- The November 2014 list’s top machine is: Tianhe-2 (Guangzhou, China)
  - Massive parallelism (3,120,000 cores!)
  - Massive power consumption (17.8 MWatts)
    - Submarine nuclear reactor
  - Theoretical peak performance: 53,902 TFlop/s
    - Flop: Floating Point Operations
- Peak Performance: unachievable Flop number assuming all functional units work full tilt
  - Tianhe-2 achieves 61% of the peak on an “easy” linear system solve benchmark
- Could we build an equivalent sequential computer?

A Sequential Supercomputer?

- The goal: peak performance of 5×10^{16} Flop/s
- One flop in one cycle: 5×10^{16} Hz clock rate
  - As we'll see in a bit, we're *far* from this.
- If each operation operates on at least one different byte, we must bring each byte in less than $1/(5 \times 10^{16})$ seconds
- Assuming speed-of-light access, a byte cannot be further than 6×10^{-9} m from the CPU
- Assuming a perfect, spherical computer, this means that each byte of data must occupy less than 3×10^{-33} m^2, i.e., smaller than a carbon atom!

A Sequential Supercomputer?

- The goal: peak performance of 5×10^{16} Flop/s
- One flop in one cycle: 5×10^{16} Hz clock rate
  - As we'll see in a bit, we're *far* from this.
- If each operation operates on at least one different byte, we must bring each byte in less than $1/(5 \times 10^{16})$ seconds
- Assuming speed-of-light access, a byte cannot be further than 6×10^{-9} m from the CPU
- Assuming a perfect, spherical computer, this means that each byte of data must occupy less than 3×10^{-33} m^2, i.e., smaller than a carbon atom!

A Sequential Supercomputer?

- The goal: peak performance of 5×10^{16} Flop/s
- One flop in one cycle: 5×10^{16} Hz clock rate
  - As we'll see in a bit, we're *far* from this.
- If each operation operates on at least one different byte, we must bring each byte in less than $1/(5 \times 10^{16})$ seconds
- Assuming speed-of-light access, a byte cannot be further than 6×10^{-9} m from the CPU
- Assuming a perfect, spherical computer, this means that each byte of data must occupy less than 3×10^{-33} m^2, i.e., smaller than a carbon atom!
- NOT happening... but we try anyway

Moore's Law

- To compute more per time unit use more transistors
- Good news: Moore's Law
  - Transistor density doubles every 24 months
  - Prediction made by Gordon Moore in 1965!

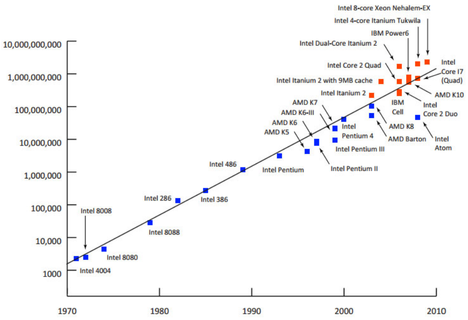
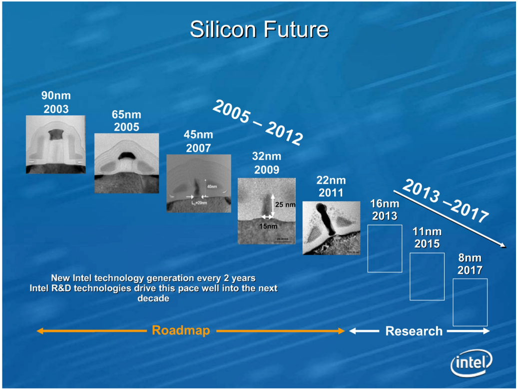


Figure from http://rapidconsultingusa.com

Moore's Law



15 complete processors today could fit on a single 1971 transistor!!

- Bad News: Is Moore's Law on the verge of failing?
  - What's beyond 5nm?
  - Google "End of Moore's law" (end of silicon)

Moore's Law and Clock Rates

- A common but wrong interpretation of Moore's law: clock rate doubles every 24 months
- This looked true for a while, but no longer (see this article)

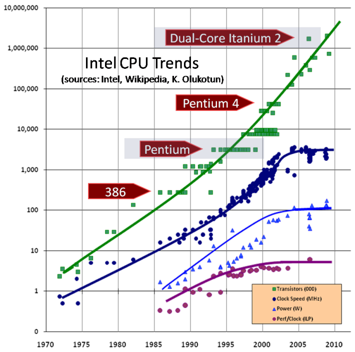
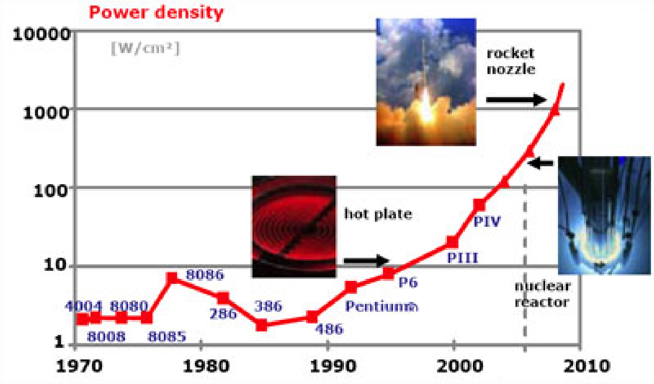


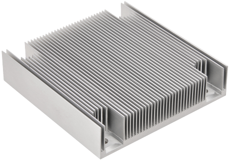
Figure from the "Free Lunch" article by Herb Sutter

If the wrong Moore's Law were true we'd have CPUs with 100+ GHz clock rates by now.

The density / clock rate problem

- Increasing transistor density and/or clock rate leads to power and temperature issues
  - Patterson, 2003: "Can soon put more transistors on a chip than can afford to turn on"





The end of Moore's Law?

- The "good news" of Moore's law may be ending anyway
- Predictions that it will end with 7nm (2020) or 5nm (2022) technology, after which CMOS won't be shrunk further
- What will happen?
  - New technologies
    - Carbon nanotubes
    - Graphene nanoribbons
    - ...
  - New designs
    - 3D chip stacking (already going on for memory, challenging due to heat)
    - Put more functionality on a chip (SoC, already going on)

Parallel vs. Sequential Computing

- Since we cannot build a single processor core with high clock rate, instead we must use multiple cores together to do HPC (within a machine, across multiple machines)
- This raises all sorts of interesting questions, some of which are the topics of this course
- But before we can focus on writing fast parallel programs, we must be able to make its sequential parts fast as well!
- Hence, this Sequential Computing Warm-Up module

Parallel vs. Sequential Computing

- Since we cannot build a single processor core with high clock rate, instead we must use multiple cores together to do HPC (within a machine, across multiple machines)
- This raises all sorts of interesting questions, some of which are the topics of this course
- But before we can focus on writing fast parallel programs, we must be able to make its sequential parts fast as well!
- Hence, this Sequential Computing Warm-Up module

Outline

Performance as Time

- Time between the start and the end of an operation
  - Also called running time, elapsed time, wall-clock time, response time, latency, execution time, flow time, ...
- Most straightforward measure: "My program takes 12.5s on a 2.4GHz Intel Core i7, but 10.1s on a 3.5GHz Intel Xeon"
- Often normalized to some reference time

The UNIX time command

- You can put time in front of any UNIX command you invoke
- When the invoked command completes, time prints out timing (and other) information

% time ls /home/casanova/ -la -R
0.520u 1.570s 0:20.58 10.1% 0+0k 570+105io 0pf+0w

0.520u	0.52 seconds of user time
1.570s	1.57 seconds of system time
0:20.56	20.56 seconds of wall-clock time
10.1%	10.1% of CPU was used
0+0k	memory used (text + data)
570+105io	570 input, 105 output (file system I/O)
0pf+0w	0 page faults and 0 swaps

The UNIX time command (II)

- Wall-clock time $\geq$ User time + System time
  - Because the process spends time in the Ready Queue or is suspended waiting for I/O events
- The time command gives interesting information
  - Also accessible from the `getrusage()` call
- But it has two drawbacks:
  - It has poor resolution ("only" milliseconds)
  - It times the whole code, not just the section of interest
- Another option is `gettimeofday`...

Timing with `gettimeofday`

- Measures the number of microseconds since midnight, Jan 1st 1970, expressed in seconds and microseconds

Example code

```
#include <sys/time.h>
struct timeval start,end;
...
gettimeofday(&start,NULL);
<some work>
gettimeofday(&end,NULL);

printf("Seconds elapsed: %f\n",
   (end.tv_sec*1000000.0 + end.tv_usec - start.tv_sec*1000000.0 - start.tv_usec) /
     1000000.0);
```

- There are timers with better resolution and precision estimates
  - e.g., `clock_gettime()` (nanoseconds!?!)

The `perf` command

- On Linux systems, the `perf` command is used for performance analysis
- Its most basic use is as: `perf stat <command>`
  - Let's try this on some command
- It can also be used to access hardware performance counters
- For instance: `perf stat -e L1-icache-load-misses <command>`
  - Let's try this on some command
- Use `perf list` so see all possible hardware/software events captured by `perf`
- Accessing harwdare counters from code is often done via the portable PAPI library

Performance as Rate

- Used often so that performance can be independent on the "size" of the application
  - e.g., compressing a 1MB file takes 1 minute. compressing a 2MB file takes 2 minutes. The performance is the same.
- Note that one can make the rate high and have terrible wall-clock time (e.g., bad algorithm)
- Millions of instructions / sec (MIPS)
  - MIPS = instruction count / (execution time * 106) = clock rate / (CPI * 106)
  - But Instructions Set Architectures are not equivalent!
    - Which is why CPU clock rate is not a good performance measure
  - May be ok for same program on similar architectures

Performance as Rate (II)

- Millions of floating point operations /sec (MFlops)
- Application-specific:
  - Millions of frames rendered per second
  - Millions of amino-acid compared per second
  - Millions of HTTP requests served per seconds
- Application-specific metrics are often preferable and others may be misleading
  - MFlops can be application-specific though
  - For instance:
    - I want to add two n-element vectors
    - This requires n Floating Point Operations
    - Therefore MFlops is a good measure

Measuring Performance

- Performance measures must be on a dedicated system
  - No other user process running (but for a Shell)
  - "single-user mode" is sometimes used
- In spite of this, performance results should be obtained over multiple trials
  - averages
  - error bars, confidence intervals, etc.
  - even better, clouds of points on graphs
- In your assignments in this course, **always** report results on at least 10 experiments (more if measures are not in a small-ranged distribution)
  - Ideally, use statistical savvy

Outline

Sequential Code Optimization

- You have a piece of sequential code, and you want to make it run faster (without considering parallelization, for now)
- Here is what you can do:
  - Buy better hardware (but clock rate is limited!)
  - Use what you learned in Algorithms and Data Structures classes
    - Know your complexities!
  - Perform *code optimization*

Sequential Code Optimization

- You have a piece of sequential code, and you want to make it run faster (without considering parallelization, for now)
- Here is what you can do:
  - Buy better hardware (but clock rate is limited!)
  - Use what you learned in Algorithms and Data Structures classes
    - Know your complexities!
  - Perform *code optimization*

Sequential Code Optimization

- You have a piece of sequential code, and you want to make it run faster (without considering parallelization, for now)
- Here is what you can do:
  - Buy better hardware (but clock rate is limited!)
  - Use what you learned in Algorithms and Data Structures classes
    - Know your complexities!
  - Perform *code optimization*

Sequential Code Optimization

- You have a piece of sequential code, and you want to make it run faster (without considering parallelization, for now)
- Here is what you can do:
  - Buy better hardware (but clock rate is limited!)
  - Use what you learned in Algorithms and Data Structures classes
    - Know your complexities!
  - Perform *code optimization*

Sequential Code Optimization

- You have a piece of sequential code, and you want to make it run faster (without considering parallelization, for now)
- Here is what you can do:
  - Buy better hardware (but clock rate is limited!)
  - Use what you learned in Algorithms and Data Structures classes
    - Know your complexities!
  - Perform *code optimization*

Code Optimization

- There is a wealth of age-old code optimization techniques
- Most of them straightforward to apply but reducing code readability
- Most of them make a LOT of sense when thinking about the assembly code, and plain odd for somebody who doesn't know about assembly code
- Let's review some typical examples...

Loop unrolling

Original code

```
for (k = 0; k < 100; k++) {
  sum += f(k);
}
```

Optimized code

```
for (k = 0; k < 100; k+=4) {
  sum += f(k);
  sum += f(k+1);
  sum += f(k+2);
  sum += f(k+3);
}
```

Why is the optimized code faster?

Loop unrolling

Original code

```
for (k = 0; k < 100; k++) {
  sum += f(k);
}
```

Optimized code

```
for (k = 0; k < 100; k+=4) {
  sum += f(k);
  sum += f(k+1);
  sum += f(k+2);
  sum += f(k+3);
}
```

Faster because the optimized code performs 4 times fewer comparisons `k < 100`

- Branch instructions are known to hurt performance due to pipeline stalls on RISC processors
- But unrolling too much will hurt instruction cache hit rate

Using Pointers

Original code

```
double a[N][N];
for (k = 0; k < N; k++) {
  a[i][k] = 2;
}
```

Optimized code

```
double a[N][N];
double *ptr = &(a[i][0]);
for (k = 0; k < N; k++) {
  *ptr = 2;
  ptr++;
}
```

Why is the optimized code faster?

Using Pointers

Original code

```
double a[N][N];
for (k = 0; k < N; k++) {
  a[i][k] = 2;
}
```

Optimized code

```
double a[N][N];
double *ptr = &(a[i][0]);
for (k = 0; k < N; k++) {
  *ptr = 2;
  ptr++;
}
```

Faster because the optimized code does not do complex address computations

- `&(a[i][k]) = &(a[0][0]) + sizeof(double)*N*i + sizeof(double)*k`
- Several additions and multiplications
- Gets worse with higher numbers of dimensions

Code Motion

Original code

```
int sum = 0;
for (k = 0; k < N*N+4*N+12; k++) {
  sum += k;
}
```

Optimized code

```
int sum = 0;
int bound = N*N+4*N+12;
for (k = 0; k < bound; k++) {
  sum += k;
}
```

The optimized code is faster because it computes `N*N+4*N+12` only once

Inlining

Original code

```
int sum = 0;
for (k = 0; k < N; k++) {
  sum += cube(i);
}
...
int cube(n) { return (n*n*n); }
```

Optimized code

```
int sum = 0;
for (k = 0; k < N; k++) {
  sum += i*i*i;
}
...
int cube(n) { return (n*n*n); }
```

The optimized code is faster because it places no function call

- Function calls involve many operations on the runtime stack, saving/restoring register values.

Instruction Scheduling

Original code

```
a++;
b++;
c *= 3;
d *= 4;
```

Optimized code

```
a++;
c *= 3;
b++;
d *= 4;
```

Why is the optimized code faster?

Instruction Scheduling

Original code

```
a++;
b++;
c *= 3;
d *= 4;
```

Optimized code

```
a++;
c *= 3;
b++;
d *= 4;
```

- The optimized code is faster because the CPU can do an addition and a multiplication at the same time
- A typical trick is to overlap memory load/stores with computation

Software Pipelining

- Instruction scheduling used in the context of loops
- Requires to reason on the assembly code
- Basic technique:
  - Unroll the loop
  - Reorder instructions so that they can happen at the same time
    - e.g., one add, one multiply, one store
- Let's see an example...

Software Pipelining Example

Source code

```
for (i=0; i < n; i++)
  prod *= a[i]
```

Assembly code for the loop body

```
// Assume that initially R0 contains the address of a[0]
// Assume the R2 corresponds to prod
mov R1, [R0]
<memory stall> // Wasted CPU cycle
mul R2, R1
add R0, 4
```

Software Pipelining Example

4-way unrolled loop with allocated registers

```
mov R1, [R0]
<memory stall>
mul R2, R1
add R0, 4

mov R3, [R0]
<memory stall>
mul R2, R3
add R0, 4

mov R4, [R0]
<memory stall>
mul R2, R4
add R0, 4

mov R5, [R0]
<memory stall>
mul R2, R5
add R0, 4
```

- Let's assume one memory stall per memory operation
- New registers are used for "independent" computations
- The CPU can add, multiply, and load from RAM at the same time
  - Pipelined, multi-issue CPU
  - But this only works for independent computations
- Let's see how many cycles this execution takes...

Execution of Unrolled Loop

Execution in 12 cycles

```
mov R1, [R0]
<memory stall>
mul R2, R1          add R0, 4

mov R3, [R0]
<memory stall>
mul R2, R3          add R0, 4

mov R4, [R0]
<memory stall>
mul R2, R4          add R0, 4

mov R5, [R0]
<memory stall>
mul R2, R5          add R0, 4
```

- Each multiplication can be done concurrently with the "add R0,4" instruction
- But we can do much better by issuing instructions out of order...

Out-of-order execution

cycle at which instruction is issued ↓

RAM	Adder	Multiplier
mov R1, [R0]	add R0, 4	<idle>
mov R3, [R0]	add R0, 4	<idle>
mov R4, [R0]	add R0, 4	mul R2, R1
mov R5, [R0]	add R0, 4	mul R2, R3
mov R1, [R0]	add R0, 4	mul R2, R4
mov R3, [R0]	add R0, 4	mul R2, R5
mov R4, [R0]	add R0, 4	mul R2, R1
mov R5, [R0]	add R0, 4	mul R2, R3
...	...	...

- We can hide all memory stalls and at each cycle we:
  - Load a new item from RAM with the current loop counter
  - Increment the loop counter for the next iteration
  - Multiply the item from 2 iterations ago

Software Pipelining Example

- We found a repeating pattern that gives us a new instruction order to execute on our CPU
- In the original ordering, we only multiply and add at the same time, so that we compute one iteration in 3 cycles
- But we found an ordering in which we compute one iteration per cycle

Software Pipeline is Difficult

- In the previous example, we found a straightforward repeating pattern, that gave us an instruction order
  - Accelerating the code by ≈ 3
- Full-fledge software pipelining is difficult (NP-hard)
  - How to use as few registers as possible?
  - Difficult to do multiple nested loops
  - Memory stalls are not all equal
  - Multiplication and Addition are not necessarily 1-cycle operations
- In practice, there are known good cases and heuristics, and a huge (a bit old, definitely theoretical) literature

Optimizations Galore

- There are dozens of known optimizations
  - Common sub-expression elimination
  - Strength reduction (e.g., replace "*" by "+", replace "*" by shifts)
  - Dead code elimination
  - Constant propagation
  - Software pipelining
  - ...
- A key concern: optimizing for locality

Outline

Memory as a Bottleneck

- A *performance bottleneck* is a limiting factor
  - If you magically make it faster, then the whole program goes faster
- Many, many programs are memory bound
  - i.e., memory access is the bottleneck
- Consider this code: `A[i] = B[j] + C[k]`
- There are 3 memory accesses (two loads, one store) and one addition
- This line of code is memory-bound on most machines
  - In the 70's, everything was balanced (n cycles to execute an instruction, n cycles to bring in a word from memory)
  - CPUs have gotten 1,000x faster, RAM has gotten 10x faster (and 1,000,000x larger)

Memory as a Bottleneck on my Laptop

Simple loops repeated 32*10000 times, N=100000

```
// 1 +, 3 refs:
for (i=0; i < N; i+=32)                                    10.41 sec
  A[i] = B[i] + C[i];

// 1 +, 8 *, 3 refs:
for (i=0; i < N; i+=32)                                    10.62 sec
  A[i] = B[i]*B[i]*B[i]*B[i]*B[i] + C[i]*C[i]*C[i]*C[i]*C[i];

// 1 +, 9 *, 1 ref
for (i=0; i < N; i+=32)                                    5.76 sec
  A[i] = A[i]*A[i]*A[i]*A[i]*A[i] + A[i]*A[i]*A[i]*A[i]*A[i]*A[i];

// TONS + and *, 1 ref
for (i=0; i < N; i+=32)                                    12.21 sec
  A[i] = (int)log(log(log((double)A[i])));
```

Memory as a Bottleneck on my Laptop

Simple loops repeated 32*10000 times, N=100000

// 1 +, 3 refs:

```
for (i=0; i < N; i+=32)
  A[i] = B[i] + C[i];
```

10.41 sec

// 1 +, 8 ∗, 3 refs:

```
for (i=0; i < N; i+=32)
  A[i] = B[i]*B[i]*B[i]*B[i]*B[i] + C[i]*C[i]*C[i]*C[i]*C[i];
```

10.62 sec

// 1 +, 9 ∗, 1 ref

```
for (i=0; i < N; i+=32)
  A[i] = A[i]*A[i]*A[i]*A[i]*A[i] + A[i]*A[i]*A[i]*A[i]*A[i]*A[i];
```

5.76 sec

// TONS + and ∗, 1 ref

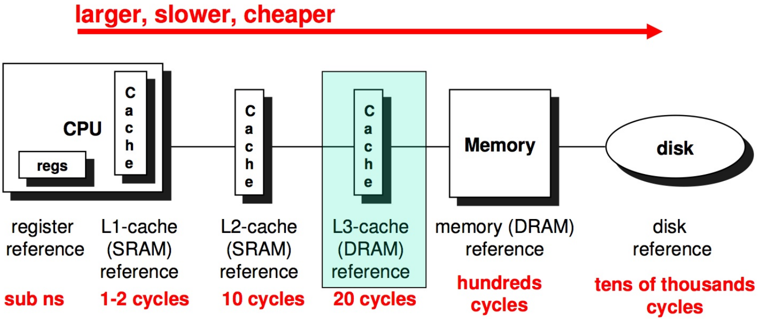
```
for (i=0; i < N; i+=32)
  A[i] = (int)log(log(log((double)A[i])));
```

12.21 sec

- Bottom line: computation is (almost) free and for many applications the CPU is starved for data

Memory Hierarchy

- We cope with the memory wall with a hierarchy of caches
  - When fetching a byte from RAM, grab a whole *cache line* around the byte and copy it all the way up to the L1 cache



Locality

- The memory hierarchy works because of locality that occurs in (useful) code
  - **Temporal locality**: When a memory location is accessed, it will be accessed again soon
  - **Spatial locality**: When a memory location is accessed, locations next to it will be accessed soon
- One must maximize locality to maximize performance
  - The compiler can help... a bit
- Keeping a mental picture of the memory layout of the application and reasoning about locality is difficult
  - cache-aware algorithms
  - cache-oblivious algorithms
- On a multi-core architecture, one must also think of which caches are shared/private, which is often very complex

Locality

- The memory hierarchy works because of locality that occurs in (useful) code
  - **Temporal locality**: When a memory location is accessed, it will be accessed again soon
  - **Spatial locality**: When a memory location is accessed, locations next to it will be accessed soon
- One must maximize locality to maximize performance
  - The compiler can help... a bit
- Keeping a mental picture of the memory layout of the application and reasoning about locality is difficult
  - cache-aware algorithms
  - cache-oblivious algorithms
- On a multi-core architecture, one must also think of which caches are shared/private, which is often very complex

Locality and 2-D Array Initialization

i-j loop

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    A[i][j] = 42;
  }
}
```

j-i loop

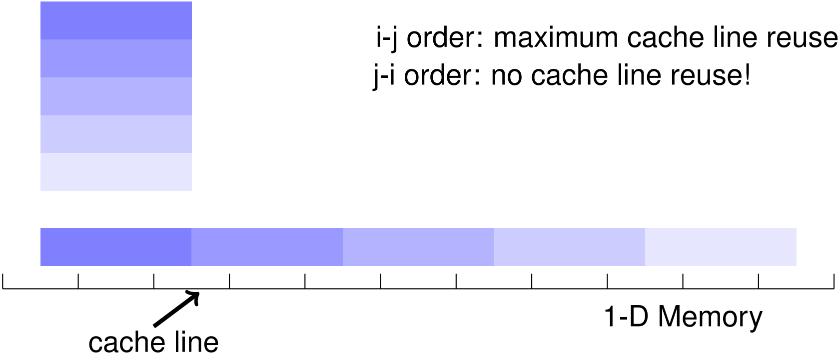
```
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i++) {
    A[i][j] = 42;
  }
}
```

Which code is faster?

1-D Memory

- Non-1-D data structures are mapped to the 1-D memory
- C stores arrays in row-major fashion:

i-j order: maximum cache line reuse
j-i order: no cache line reuse!



Counting Cache Misses

- Let's consider the 2-D array initialization code
- Assume each element is 1 byte, the array has N rows and M columns, and each cache line is L bytes
- How many cache misses for the i-j order?
- How many cache misses for the j-i order?

Counting Cache Misses

- Let's consider the 2-D array initialization code
- Assume each element is 1 byte, the array has N rows and M columns, and each cache line is L bytes
- How many cache misses for the i-j order?

  $M \times N/L$ (one miss every L accesses)
- How many cache misses for the j-i order?

  $M \times N$ (maximized!)

Counting Cache Misses

- Let's consider the 2-D array initialization code
- Assume each element is 1 byte, the array has N rows and M columns, and each cache line is L bytes
- How many cache misses for the i-j order?

  $M \times N/L$ (one miss every L accesses)
- How many cache misses for the j-i order?

  $M \times N$ (maximized!)

- This is all about spatial locality (if $L = 1$, no difference)

Revisiting Memory Wall Example: NO LOCALITY

Simple loops repeated 32*10000 times, N=100000

```
// 1 +, 3 refs:
for (i=0; i < N; i+=32)                                   10.41 sec
  A[i] = B[i] + C[i];

// 1 +, 8 *, 3 refs:
for (i=0; i < N; i+=32)                                   10.62 sec
  A[i] = B[i]*B[i]*B[i]*B[i]*B[i] + C[i]*C[i]*C[i]*C[i]*C[i];

// 1 +, 9 *, 1 ref
for (i=0; i < N; i+=32)                                   5.76 sec
  A[i] = A[i]*A[i]*A[i]*A[i]*A[i] + A[i]*A[i]*A[i]*A[i]*A[i]*A[i];

// TONS + and *, 1 ref
for (i=0; i < N; i+=32)                                   12.21 sec
  A[i] = (int)log(log(log((double)A[i])));
```

Memory Wall Example Revisited: LOCALITY

Simple loops repeated 10000 times, N=100000

```
// 1 +, 3 refs:
for (i=0; i < N; i++)                                      2.78 sec (was 10.41 sec)
  A[i] = B[i] + C[i];

// 1 +, 8 *, 3 refs:
for (i=0; i < N; i++)                                      5.67 sec (was 10.62 sec)
  A[i] = B[i]*B[i]*B[i]*B[i]*B[i] + C[i]*C[i]*C[i]*C[i]*C[i];

// 1 +, 9 *, 1 ref
for (i=0; i < N; i++)                                      4.08 sec (was 5.76 sec)
  A[i] = A[i]*A[i]*A[i]*A[i]*A[i] + A[i]*A[i]*A[i]*A[i]*A[i]*A[i];

// TONS + and *, 1 ref
for (i=0; i < N; i++)                                      11.88 sec (was 12.21 sec)
  A[i] = (int)log(log(log((double)A[i])));
```

Memory Wall Example Revisited: LOCALITY

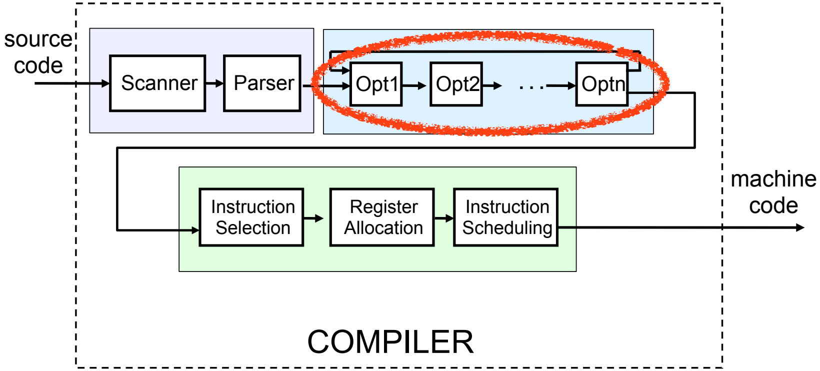
Simple loops repeated 10000 times, N=100000

```
// 1 +, 3 refs:
for (i=0; i < N; i++)                                      2.78 sec (was 10.41 sec)
  A[i] = B[i] + C[i];

// 1 +, 8 *, 3 refs:
for (i=0; i < N; i++)                                      5.67 sec (was 10.62 sec)
  A[i] = B[i]*B[i]*B[i]*B[i]*B[i] + C[i]*C[i]*C[i]*C[i]*C[i];

// 1 +, 9 *, 1 ref
for (i=0; i < N; i++)                                      4.08 sec (was 5.76 sec)
  A[i] = A[i]*A[i]*A[i]*A[i]*A[i] + A[i]*A[i]*A[i]*A[i]*A[i]*A[i];

// TONS + and *, 1 ref
for (i=0; i < N; i++)                                      11.88 sec (was 12.21 sec)
  A[i] = (int)log(log(log((double)A[i])));
```

- Memory accesses are much cheaper thanks to caches
- But they are still expensive!

Outline

The Compiler



Compiler Optimization

- In the 60's and 70's, developers did a lot of by-hand optimization
  - Making the code much less readable
- Decades of compiler-driven optimization research
- Modern compilers can do a lot for you, with some very tricky optimizations
  - Especially vendor-provided compilers
- Most compilers provide several levels of optimization
  - -O0, -O1, -O2, -O3, -O4/-fast, etc.
  - Pre-packages sets of optimizations (let's look at gcc)
- "Interesting" to look at the generated assembly!
- The highest levels of optimization can break code
- There are options to optimize for code size

The Compiler Cannot Do Everything

Aliasing

```
void foo(int *q, int *p) {
  *q = 3;
  *p++;
  *q *= 4;  // q can be equal to p
}
```

Aliasing

```
a[i] = b[i] + c;
a[i+1] = b[i+1] * d;   // What if &(b[i+1]) == &(a[i])?
```

Function Call

```
//function  f may have side effects
for (i = 0; i < f(n) ; i++) {
  sum += i;
}
```

Locality?

The Memory-Wall Example and the Compiler

Version	No locality		Locality	
	-O0	-O4	-O0	-O4
1 +, 3 refs	10.41	5.67	2.78	0.62
1 +, 8 *, 3 refs	10.62	5.69	5.67	2.52
1 +, 9 *, 1 ref	5.76	2.41	4.08	1.55
TONS + and *, 1 ref	12.21	12.29	11.88	12.24

Wall-clock times (sec)

The Array Initialization Example and the Compiler

Version	-O0	-O4
Locality	3.32	0.65
No Locality	12.45	12.14

Wall-clock times (sec)

The Array Initialization Example and the Compiler

Version	-O0	-O4
Locality	3.32	0.65
No Locality	12.45	12.14

Wall-clock times (sec)

The compiler doesn't do (all) locality optimizations for you

The Beaten-to-Death Matrix-Multiplication

Aliasing

```
for (i=0; i < N; i++) {
  for (j=0; j < N; j++) {
    for (k=0; k < N; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

- There are 3!=6 possible ordering of the loops
- Each ordering leads to some number of cache misses
- Counting the exact number of cache misses is tedious
  - Requires assumptions about cache characteristics
  - Requires error-free discrete math
- We don’t need cache miss counts, just the best ordering
- Typical approach: think about the inner loop only

The Beaten-to-Death Matrix-Multiplication

Aliasing

```
for (i=0; i < N; i++) {
  for (j=0; j < N; j++) {
    for (k=0; k < N; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

ordering	C[i][j]	A[i][k]	B[k][j]
i-j-k	cst	seq	str
i-k-j	seq	cst	seq
j-i-k	cst	seq	str
j-k-i	str	str	cst
k-i-j	seq	cst	seq
k-j-i	str	str	cst

- Constant: does not depend in inner loop index
- Sequential: one cache miss every N/L accesses
- Strided: one cache miss every access

The Beaten-to-Death Matrix-Multiplication

Aliasing

```
for (i=0; i < N; i++) {
  for (j=0; j < N; j++) {
    for (k=0; k < N; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

ordering	C[i][j]	A[i][k]	B[k][j]
i-j-k	cst	seq	str
i-k-j	seq	cst	seq
j-i-k	cst	seq	str
j-k-i	str	str	cst
k-i-j	seq	cst	seq
k-j-i	str	str	cst

- Constant: does not depend in inner loop index
- Sequential: one cache miss every N/L accesses
- Strided: one cache miss every access

Sequential Warmup "Experience"

- We have a short programming assignment ...

Matrix Multiplication

- Matrix multiplication doesn’t have to be 3 nested loops!
  - Because of associativity and commutativity, we have n^3 computations that can be done in any order
- Researchers have looked at matrix-multiplication from a much broader perspective
- Consider an $n \times n$ matrix multiplication with a single level of cache of size M

Theorem (Upper bound)

The number of memory<->cache transfers is at most

$$3.46\left(\frac{n^3}{\sqrt{M}}\right) .$$

Matrix Multiplication

Theorem (Lower bound - Irony et al., 2004)

The number of memory<->cache transfers is at least

$$0.35\left(\frac{n^3}{\sqrt{M}}\right) - M\ .$$

- Bounds are useful to assess the performance of practical algorithms
- Let's go through the method for obtaining such a result
  - Classical result, but re-explained nicely by Prof. Julien Langou at a Dagstuhl Seminar in July 2015, whose slides I used as an inspiration

What are the fundamental operations

- The “operations” for a matrix multiplication are
  - **Read** an element from memory into cache
  - **Update** an element (of C) in cache
  - **Write** an element from cache to memory
  - **Delete** an element from cache
- We assume that we manage the cache from the algorithm
  - In practice, the cache is hardware-managed
  - But the goal is to compute a lower bound assuming we can control everything, so that in practice any algorithm will be worse

Algorithm Segment

- Any algorithm proceeds as a sequence of operations:

...	
Read	a_{13}
Read	b_{31}
Read	c_{11}
Update	c_{11}
Read	a_{24}
Write	c_{11}
Delete	c_{11}, a_{13}, b_{42}
...	

- Let us define a segment as a sequence of instructions that performs a total of M memory operations (reads or writes)

Algorithm Segment

- For a segment let us define:
  - R_a: number of reads for elements of matrix A
  - W_a: number of writes for elements of matrix A
  - M_a: number of elements of matrix A at the beginning of the segment
  - N_a: number of elements of matrix A at the end of the segment
- We have the above for matrices B and C as well
- Deletes are free so we don't need to count them
- **Objective**: maximize the number of updates in a segment
  - So as to maximize data reuse

Optimization Problem

Problem

Maximize the # of multiplications, subject to:

$$R_a + R_b + R_c + W_a + W_b + W_c = M \quad (1)$$

$$M_a + M_b + M_c \leq M \quad (2)$$

$$N_a + N_b + N_c \leq M \quad (3)$$

All variables are ≥ 0 (4)

Optimization Problem

Problem

Maximize the # of multiplications, subject to:

$$R_a + R_b + R_c + W_a + W_b + W_c = M \quad (1)$$

$$M_a + M_b + M_c \leq M \quad (2)$$

$$N_a + N_b + N_c \leq M \quad (3)$$

$$\textit{All variables are} \geq 0 \quad (4)$$

- Constraint (1): Total number of I/O operations in a segment
- Constraints (2)-(3): Cache capacity before/after the segment
  - We don't care about the middle of the segment to simplify!!
- Constraint (4): Obvious nonnegativity
- These constraints hold for **any matrix-multiplication** algorithm

Element updates

- **Question:** How many updates can we do in a segment?
- To perform the k-th update of c_{ij} we need to have c_{ij}, a_{ik}, and $b_{k,j}$ in cache
- Here comes in a geometric argument: the Loomis-Whitney inequality
  - Let $V \in Z^3$ be a finite set, and let V_x, V_y, and V_z be orthogonal projections of V onto the coordinate planes. Then $|V| \leq \sqrt{|V_x| \cdot |V_y| \cdot |V_z|}$
  - Given V_a, V_b, and V_c, elements of A, B, and C in cache, we can perform at most $\sqrt{|V_a| \cdot |V_b| \cdot |V_c|}$ updates
- This is the true insight behind the method
- We can now reason on V_a, V_b, and V_c...

Elements in cache

- The maximum number of elements of A in cache during a segment execution, V_a, satisfies $V_a \leq M_a + R_a$
  - The ones at the beginning + the ones read
- The maximum number of elements of B in cache during a segment execution, V_b, satisfies $V_b \leq M_b + R_b$
- The maximum number of elements of C in cache during a segment execution, V_c, satisfies $V_c \leq N_c + W_c$
  - The ones at the end of the segment + the ones that were written
  - Note that this is "cleverly" different, but valid
- Therefore, the maximum number of updates that can be performed during a segment is at most $\sqrt{(M_a + R_a)(M_b + R_b)(N_c + W_c)}$

Updated Optimization Problem

Problem

Maximize $\sqrt{(M_a + R_a)(M_b + R_b)(N_c + W_c)}$, *subject to*

$$
\begin{aligned}
R_a + R_b + R_c + W_a + W_b + W_c &= M && (5)\\
M_a + M_b + M_c &\leq M && (6)\\
N_a + N_b + N_c &\leq M && (7)\\
\textit{All variables are} \geq 0 & && (8)
\end{aligned}
$$

- The objective is to maximize the maximum potential number of updates (Loomis-Whitney inequality)
- The solution of the problem will provide us with a lower bound
  - No algorithm can do better in practice

Updated Optimization Problem

Problem

Maximize $\sqrt{(M_a + R_a)(M_b + R_b)(N_c + W_c)}$, *subject to*

$$R_a + R_b + R_c + W_a + W_b + W_c = M \quad (9)$$

$$M_a + M_b + M_c \leq M \quad (10)$$

$$N_a + N_b + N_c \leq M \quad (11)$$

$$\textit{All variables are} \geq 0 \quad (12)$$

- The variables in blue above do not appear in the objective function, and $N_c \leq M$ always hold
- We can thus set these blue variables to zero and set N_c to M

Updated Optimization Problem

Problem

Maximize $\sqrt{(M_a + R_a)(M_b + R_b)(M + W_c)}$, *subject to*

$$R_a + R_b + W_c = M \tag{13}$$

$$M_a + M_b \leq M \tag{14}$$

$$\textit{All variables are} \geq 0 \tag{15}$$

- Each variable is bounded by M, therefore :

$$\sqrt{(M_a + R_a)(M_b + R_b)(M + W_c)} \leq \sqrt{2 \cdot M \cdot 2 \cdot M \cdot 2 \cdot M} = 2\sqrt{2}M^{3/2}$$

- Upper bound on the number of updates in a segment: $2\sqrt{2}M^{3/2}$

Completing the proof

- We need to perform a total of n^3 updates, so we have:

$$\#\text{segments} \geq \lfloor \frac{n^3}{2\sqrt{2}M^{3/2}} \rfloor$$

- Since each segment performs M reads/writes, we have:

$$\#\text{reads/writes} \geq \lfloor \frac{n^3}{2\sqrt{2}M^{3/2}} \rfloor \cdot M$$

- Removing the floor gives us the theorem:

$$\#\text{reads/writes} \geq \frac{1}{2\sqrt{2}} \frac{n^3}{\sqrt{M}} - M$$

Wait, is this a theory course???

- Not at all!!!
- But I'll show a few standard proof techniques every now and then because I find them interesting
- HPC is a domain in which there are tons of theoreticians, and tons of practitioners

- At the Dagstuhl seminar I was at in July 2014, Prof. Julien Langou then proceeded to explain how they improved the bound to $\frac{2n^3}{\sqrt{M}} - 2M$
  - With not too complicated techniques (consider segments of size αM, add more constraints not just begin/end, etc.)

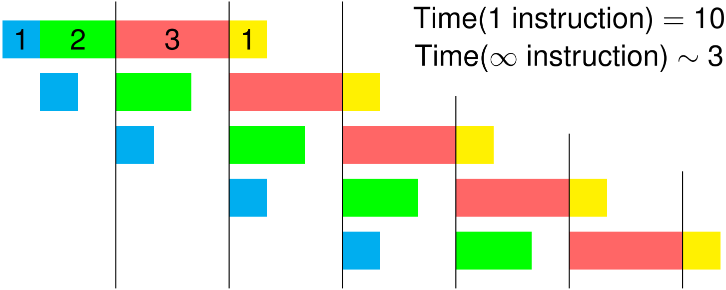
Outline

Sequential?

- Not much is really sequential in today's computers
- For decades the way to achieve performance of sequential programs has been to do things in parallel under the hood
- We've seen this with instruction scheduling, software pipelining
  - Concurrent use of CPU and memory
  - Concurrent use of ALU components
- So although you've always thought of your programs as sequential, in fact parallelism has been going on all along at a low level

Pipelining

- If one has a sequence of tasks to perform, and each task consists of the same n steps where each step is performed by a different piece of hardware
- Then one can do pipelining:



$$\text{Time}(1 \text{ instruction}) = 10$$
$$\text{Time}(\infty \text{ instruction}) \sim 3$$

RISC Processors

- If all pipeline stages take the same time, then we have a "balanced" pipeline, which is more efficient
- This is how modern (RISC) processors work, with each instruction consisting of a sequence of operations that take (if all goes well) 1 clock cycle
  - e.g., instruction fetch, instruction decode, instruction execute, memory load/store, register write-back
  - Some instruction are idle during some stages
  - Pipeline stalls can be inserted
  - Most real-world pipelines are much longer than 5 stages

Pipelines Function Units

- Some operations (e.g., instruction execute) are too expensive to do in one clock cycle
  - e.g., numerical operations like multiplications
- These are pipelined as well

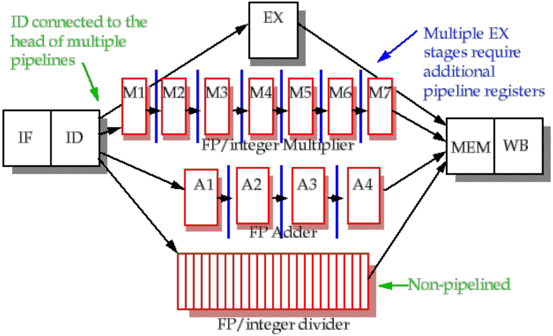


Figure from Computer Architecture: Quantitative Approach, by Hennessy and Patterson.

Vector Processing

- A vector unit is a functional unit that can do concurrent element-wise operations on special *vector registers*
  - Thanks to multiple functional units
  - Obviously very useful for, say, linear algebra
- So-called SIMD: Single Instruction Multiple Data
- In the 70's there were "vector supercomputers"
- Vector unit has made its way into the mainstream
- "Vectorizing compilers" used to be fancy
- Modern compilers to loop vectorization as a matter of course (e.g., gcc -O3)
- Let's see it in action for a simple example...

Loop Vectorization in Action

Simple vectorizable loop

```
#define N (1024*1024*1024)
char A[N],B[N];
for (int i=0; i < N; i++)
      A[i] = A[i] * B[i];
```

Without Vectorization

```
% gcc -O1 loop.c -o loop; ./loop;
```

1.55 sec

With Vectorization

```
% gcc -O1 -ftree-vectorize loop.c -o loop; ./loop;
```

0.96 sec

Outline

Bottom Line of Sequential Code Optimization

- Use the highest level of compiler optimization
- Use a profiler to find out the most time-consuming portions of the code
  - Accelerating by 100x a portion of the code that's responsible for 10% of the execution time accelerates the whole code by only 9.9%
- Find and remove known optimization blockers
- Do not optimize your code by hand into oblivion, but deal with locality

- The above approach works reasonably well, but of course more sophistication can take performance higher
- Question: Can a human handle more sophistication?

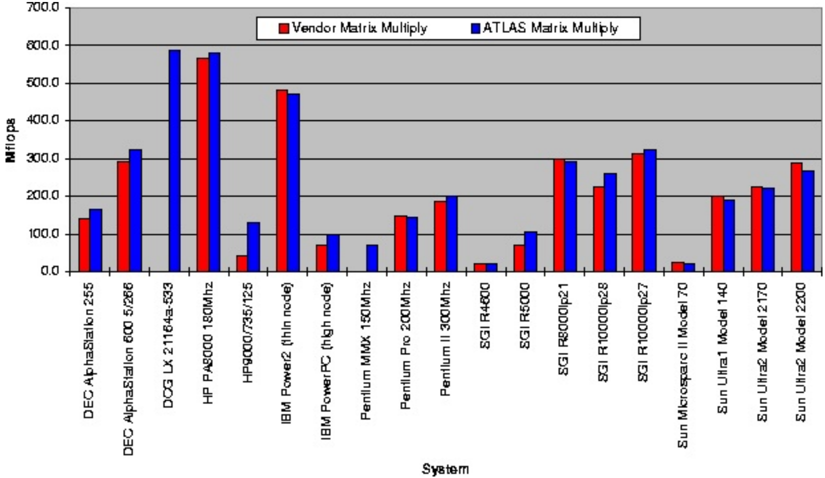
Code Optimization is *Optimization*

- Code optimization is difficult
  - Tons of possible compiler optimizations
    - with options (e.g., by how much to unroll a loop?)
  - Tons of possible hand optimizations
    - e.g., rearranging computation for locality
    - e.g., rearranging the data structures for locality
- This is an *optimization problem*
  - Objective: minimize wall-clock time
  - Search space: all possible correct executables
  - Search strategy: use a computer program to generate a large number of (non-stupid) executables and select the fastest one
- Given a computer, solve the optimization problem (which can take hours) and get a high-performance implementation

The ATLAS Project

- ATLAS is a software that you can download and run on most platforms
- It runs for a while (perhaps a couple of hours) and generates a .c file that implements matrix multiplication!
  - Does some pruning of the search space
- ATLAS optimizes for
  - Instruction cache reuse
  - Instruction scheduling
  - Reducing loop overhead
  - Cache reuse
  - ...
- One of the first AutoTuning approaches

The ATLAS Result from 1997



End of Warm-Up

- Optimizing sequential code is by no means straightforward
- But it can be entertaining/frustrating
  - Some of you may recall ICS432 final projects
- But we have a large body of knowledge
  - A lot of which is implemented in the compiler
  - But compiler optimization is mysterious (even magical) unless you really know your architecture and your assembly
- Commonly used kernels have been optimized by others before you, so don't waste your time
  - DO NOT implement your own matrix-matrix multiplication
  - Unless it's for a programming assignment ☺
- We have a warm-up programming assignment...