"The Pareto principle (also known as the 80/20 rule, the law of the vital few, or the principle of factor sparsity) states that, for many events, roughly 80% of the effects come from 20% of the causes."

wikipedia

Lecture B.2:
Performance Optimization

CS205: Computing Foundations for Computational Science
Dr. David Sondak
Spring Term 2021

HARVARD
School of Engineering and Applied Sciences

IACS INSTITUTE FOR APPLIED COMPUTATIONAL SCIENCE AT HARVARD UNIVERSITY

Lectures developed by: Dr. Ignacio M. Llorente

Before We Start

Where We Are

Computing Foundations for Computational and Data Science

How to use modern computing platforms in solving scientific problems

Intro: Large-Scale Computational and Data Science

A. Parallel Processing Fundamentals

B. Parallel Computing

- B.1. Foundations of Parallel Computing
- B.2. Performance Optimization
- B.3. Accelerated Computing
- B.4. Shared-memory Parallel Processing
- B.5. Distributed-memory Parallel Processing

C. Parallel Data Processing

Wrap-Up: Advanced Topics

CS205: Contents

APPLICATION SOFTWARE

APPLICATION PARALLELISM

PARALLEL PROGRAM DESIGN

PROGRAMMING MODEL

- Optimization
- OpenACC
- OpenMP
- MPI

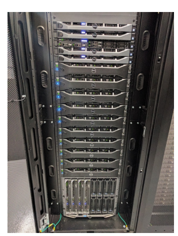
- Spark
- Map-Reduce

PLATFORM

B. BIG COMPUTE

C. BIG DATA

amazon web services



Open Nebula

FAS RC

FAS RC CANNON HARVARD'S LARGEST CLUSTER

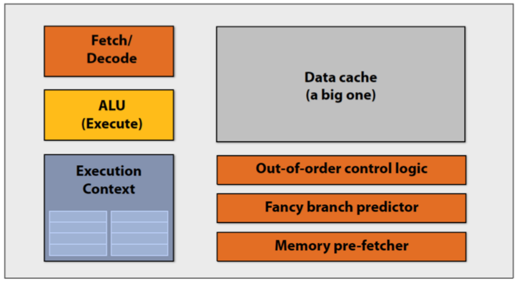
CLOUD COMPUTING

PARALLEL ARCHITECTURES

Context

Single-thread / Single-core Optimization

Before using multiple cores or nodes, let us maximize the performance of the application on a single core



ILP/Data

Context

What is the Goal of Optimization?

- Different kinds of optimization:
  - ✓ Space optimization: Reduce memory use
  - ✓ Time optimization: Reduce execution time
  - ✓ Power optimization: Reduce power usage
  - ✓ …

Roadmap

Performance Optimization

Performance Analysis

Optimization Process

Optimization Techniques

Memory Locality Model

Loop Optimization

Compiler

PERFORMANCE ANALYSIS

Performance Analysis

The Main Questions to Reduce Execution Time

Why is the code inefficient ?

- Processor
- Input/output
- Memory

time

Where is the bottleneck?

profiler

Pareto: 80/20

How can it be improved?

- Optimization techniques
- Accelerators
- Parallel computing

Performance Analysis

WHY? - Execution Time Components

EXECUTION_TIME = CPU_TIME + I/O_TIME + SYSTEM_TIME

Read/write data

Virtual memory (page faults)

System calls

Non-exclusive resources

Optimization

Accelerators

PARALLEL PPROCESSING

Performance Analysis

WHY? - Execution Time Components

time

```
demos% time a.out
real     0m0.064s
user     0m0.001s
sys      0m0.002s
```

Wall-clock time (real time)

User CPU time

System CPU time

Note: There is some shell variability in the output. The example above is from the `bash` shell.

Performance Analysis

WHERE? - Code Profiling

- Identify the program's hotspots:
  - ✓ Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places (Pareto)
  - ✓ Profilers and performance analysis tools can help here
  - ✓ Focus on optimizing the hotspots and ignore those sections of the program that account for little CPU usage

- Identify bottlenecks in the program:
  - ✓ Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
  - ✓ May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas

Performance Analysis

WHERE? – Tools for Code Profiling

gprof

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 51.52      2.55     2.55        5   510.04   510.04  USURP_Reg_poll
 29.41      4.01     1.46       34    42.82    42.82  USURP_DMA_write
 11.97      4.60     0.59       14    42.31    42.31  USURP_DMA_read
  4.06      4.80     0.20        1   200.80   200.80  USURP_Finalize
  2.23      4.91     0.11        5    22.09    22.09  localp
  1.22      4.97     0.06        5    12.05    12.05  USURP_Load
  0.00      4.97     0.00       10     0.00     0.00  USURP_Reg_write
  0.00      4.97     0.00        5     0.00     0.00  USURP_Set_clk
  0.00      4.97     0.00        5     0.00   931.73  rcwork
  0.00      4.97     0.00        1     0.00     0.00  USURP_Init
```

% of total running time

Running sum of # of seconds accounted for by this function and those above it

Seconds accounted for by this function.

# of times this function was called

Ave. # of ms spent in this function per call

Ave. # of ms spent in this function and its descendents per call

Performance Analysis

HOW? - Execution Time Components

Processor

- Optimization
- Accelerators
- Parallel programming

Input/output

- Reorganize I/O to reuse data and have a lower number of larger transactions
- Parallelize I/O transactions
- Functions mmap to map files into memory
- Functions madvise to give directions to the OS about the file access pattern

Virtual Memory

- Optimize data structures and memory access patterns to improve data locality

Performance Analysis

WHERE? – Tools for Code Profiling

- Help identify performance problems, answering questions like:

  How many times each method in the code is called?

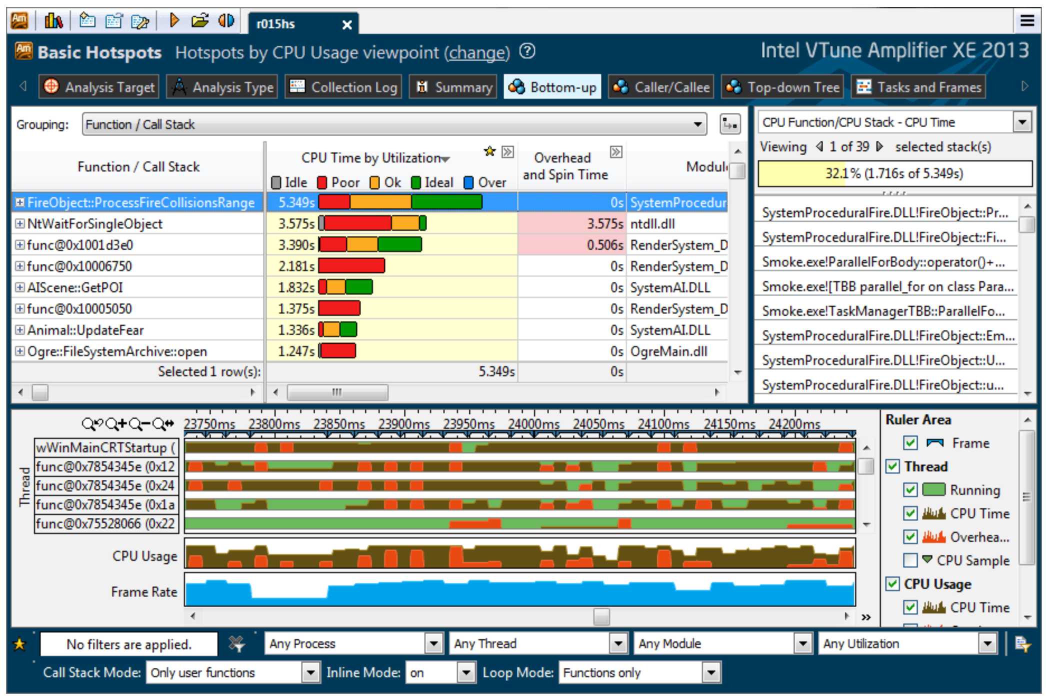
  How long does each of those methods take?

  What uses twenty percent of the total CPU usage of the code?

CLI Tools

```
gperftools, valgrind, gprof...
```

GUI Tools



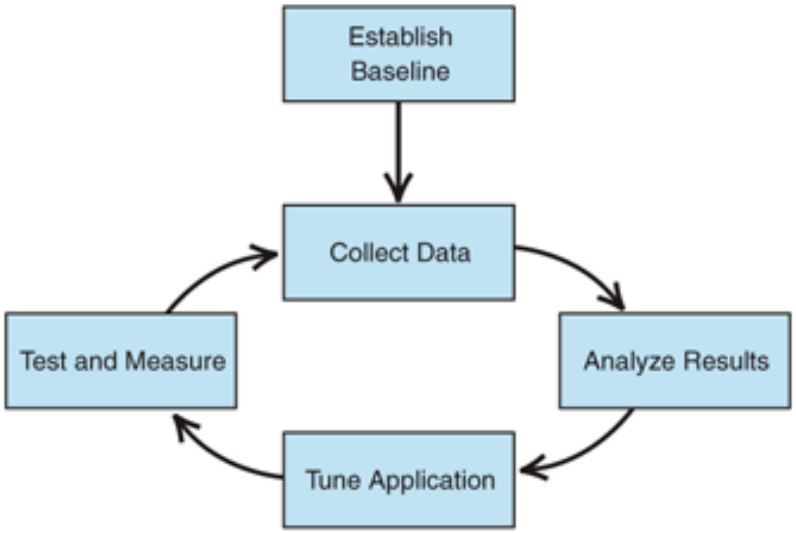
Optimization Process

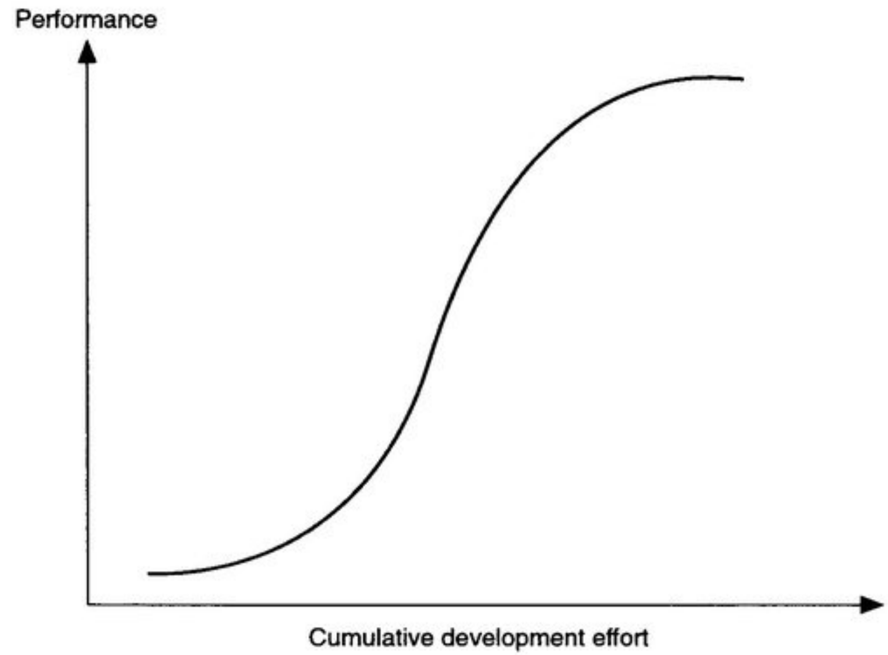
Previous Steps

1. Analyze execution time and consider Amdahl's law
2. Pick the right algorithms: Consider design for few operations and numerical complexity
3. Pick the right data structures: Consider design for locality
4. Establish baseline with no optimization (performance / results)
5. Turn on profile to figure out program hot spots
6. Start tuning process with focus on hot spots

Optimization Process

Continuous Process





OPTIMIZATION TECHNIQUES

Optimization Techniques

Optimizations Are Code Transformations

- Aimed at **achieving assembly-code performance**
  - ✓ Clean, modular, high-level source code
  - ✓ Can’t change meaning of program to behavior not allowed by source

- **Who does the work?**
  - ✓ Transformed by compiler (with our advice)
  - ✓ Transformed explicitly by developer

Optimization Techniques

Basic Techniques

Inlining

- Replace a function call with the body of the function

Constant Propagation

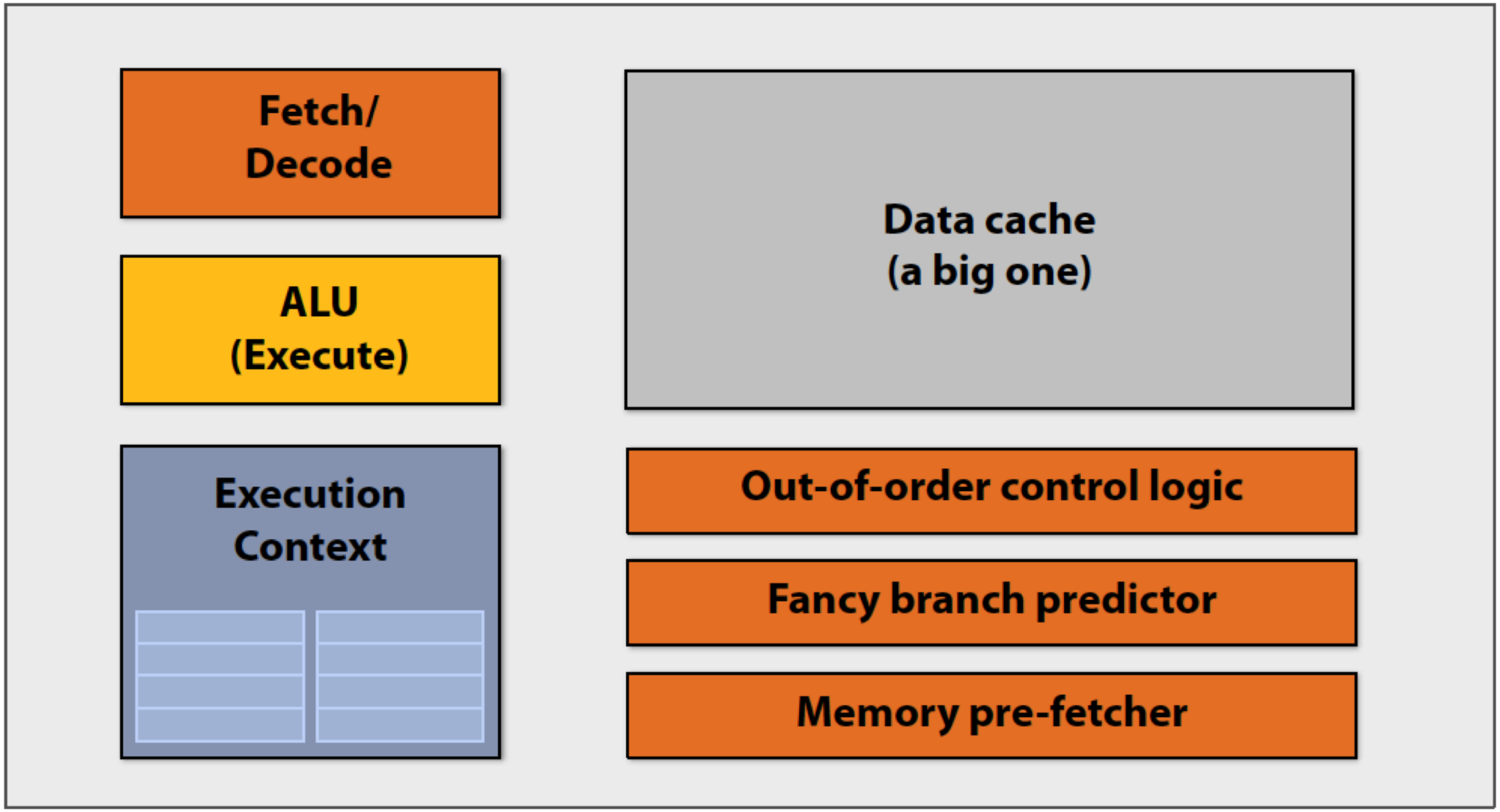
- If value of variable is known to be a constant, replace use of variable with constant

Dead-Code Elimination

- If side effect of a statement can never be observed, can eliminate the statement

Optimization Techniques

Single-core Execution Time



Optimization Techniques

Single-core Execution Time

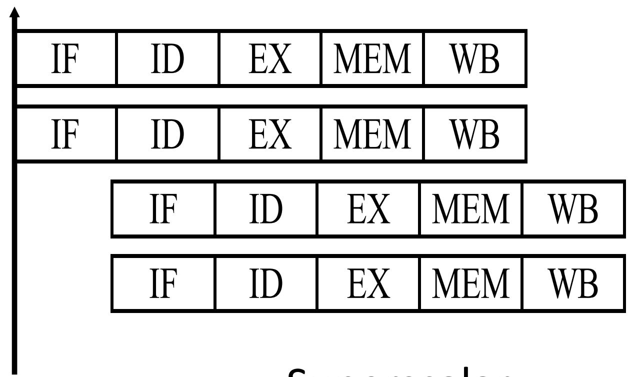
CPUs Use Two Main Techniques for Performance

- Instruction Level Parallelism (Superscalar and Pipelining)
  - ✓ Superscalar processors have multiple "functional units" that can run in parallel
  - ✓ Pipelining is a form of parallelism, like an assembly line in a factory
- Caches (Memory Hierarchy)
  - ✓ Small amount of fast memory where values are "cached" in hope of reusing recently used or nearby data

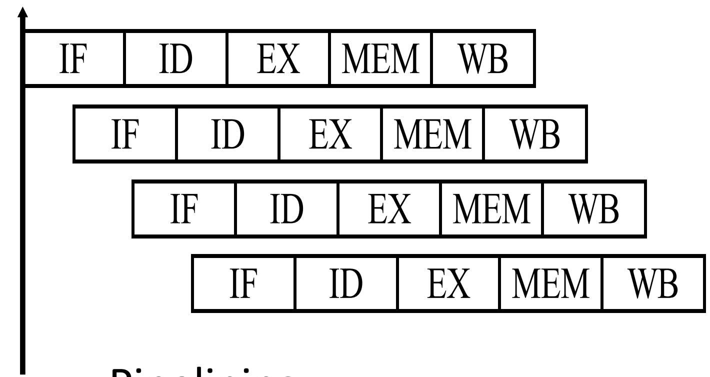
Optimization Techniques

Single-core Execution Time

CPU Architectures Try to Exploit Instruction Parallelism



Superscalar



Pipelining

AIM: Improve ILP, for example by avoiding conditional branches

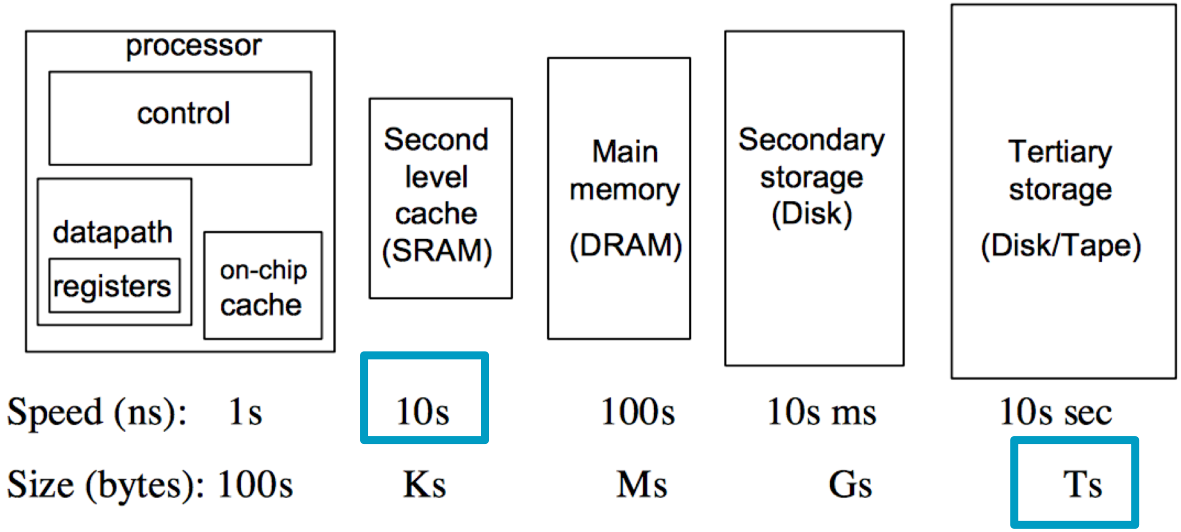
```
int x;
for (x = 0; x < 100; x++{
  delete(x);
}
```

```
int x;
for (x = 0; x < 100; x += 2 ) {
  delete(x);
  delete(x + 1);
}
```

Optimization Techniques

Uniprocessor Cost: Memory Hierarchy

Memory Hierarchy Tries to Exploit Memory Access Locality



Speed of cache with the capacity of the disk

AIM: Improve degree of memory access locality

- Spatial locality: Accessing data nearby previous accesses (low strides)
- Temporal locality: Reusing an item that was previously accessed

MEMORY LOCALITY MODEL

Memory Locality Model

Simple Model for Temporal Locality

Simple Model (Temporal Locality)

Consider two types of memory (fast and slow) over which we have complete control:

- m = words read from slow memory
- t_m = slow memory access time
- f = number of flops
- t_f =time per flop

$$\text{time} = ft_f + mt_m = ft_f\left(1 + \frac{t_m/t_f}{q}\right) = ft_f\left(1 + \frac{b}{q}\right)$$

Relevant Ratios

- Machine balance: $b = t_m/t_f$ (smaller is better)
- Algorithm computational intensity: $q = f/m$ (larger is better)

Ideal time $= ft_f\left(1+\epsilon\right)$, ϵ is zero when all data is in fast memory.

Memory Locality Model

Example of Application of Memory Model

Simple Example of Memory Model

- Assume $t_f = 10^{-10}$ sec (0.1 ns, 10 Gflop/s => 1 Intel i9-7900X CPU core)
- Assume slow memory speed is $t_m = 10$ ns
- Assume the function `h` takes `h` flops => $f = hn$ for an array of size `n`.
- Assume array X is in slow memory => $m = n$

```
s=0;
for (int i = 0; i < n; i++) {
  s = s + h(X[i]);
}
```

$$\text{time} = \underbrace{hn}_{f} \underbrace{(0.1)}_{t_f} + \underbrace{n}_{m} \underbrace{(10)}_{t_m}$$

$$\text{machine balance} = b = \frac{t_m}{t_f} = \frac{10}{0.1} = 100$$

$$\text{computational intensity} = q = \frac{f}{m} = \frac{hn}{m} = h$$

$$\text{performance} = \frac{f}{\text{time}} = \frac{q}{10 + 0.1q}$$

As `q` increases it reaches a peak of 10 Gflop/s.

Memory Locality Model

Example of Application of Memory Model

Some Examples of q (Computational Intensity)

- Matrix-vector multiply: $m=3n+n^2$ data, $f=2n^2$ flops

```
s=0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    Y[i] = Y[i] + A[i,j]*X[j];
}
```

Assumption: Fast memory (cache) not big enough to store matrix A but it is big enough to store X and Y.

- Read in n components of x = n +
- Read in n components of y = n +
- Read in n^2 components of A = n^2 +
- Write out n components of y = n

Adding them up gives $3n+n^2$.

$$q = \frac{f}{m} = \frac{2n^2}{3n+n^2} = \frac{2}{3/n+1}$$

$q \approx 2$ for large n.

Memory Locality Model

Example of Application of Memory Model

Some Examples of q (Computational Intensity)

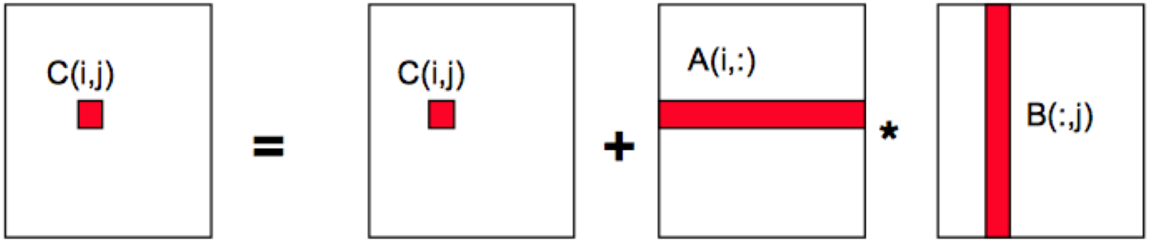
- Matrix-matrix multiply: $m = n^3 + 3n^2$ data, $2n^3 - n^2$ flops

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

Assumption: Fast memory (cache) not big enough to store matrices A/B

Breakout Room!

Verify the formula for m and calculate q for large n.

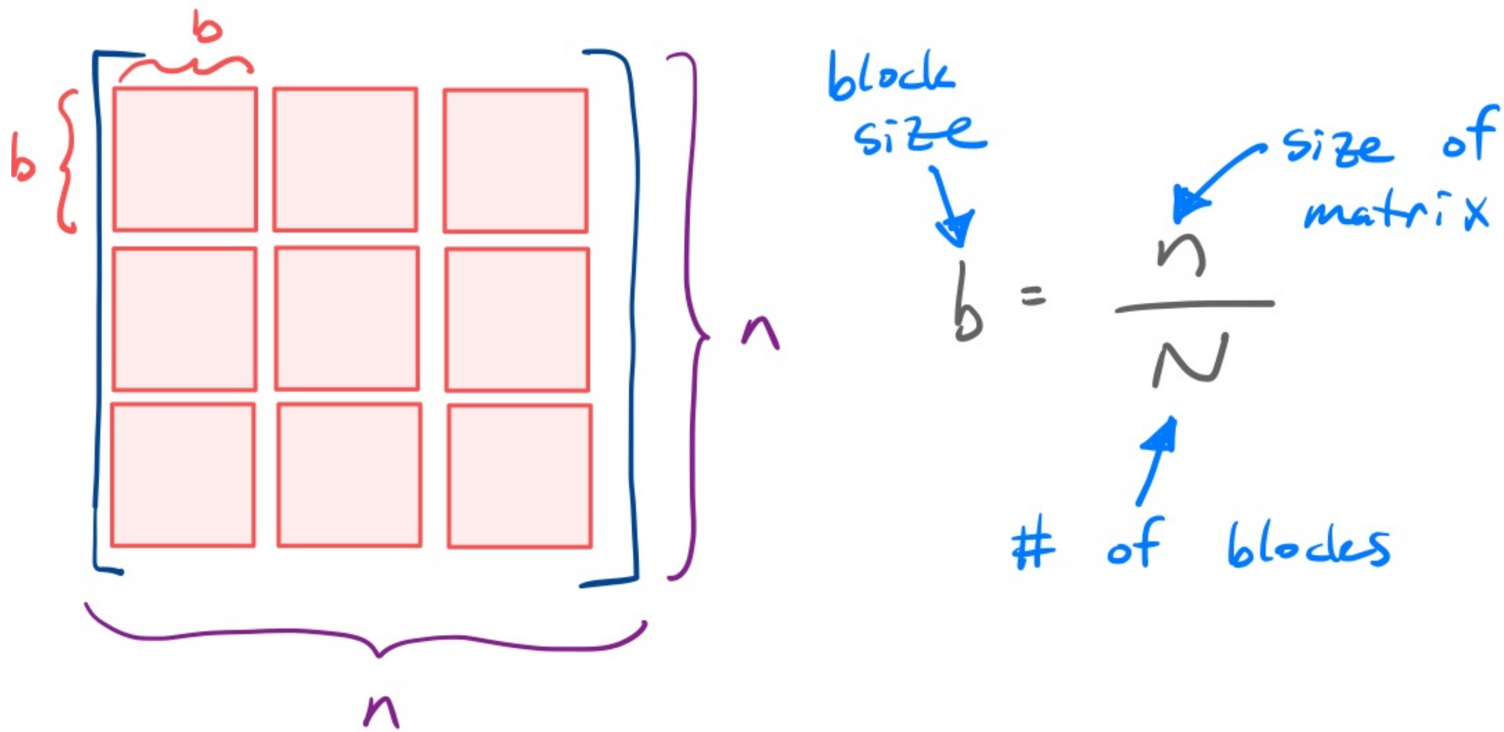


Memory Locality Model

Matrix Blocking

Some Examples of q (Computational Intensity)

- Matrix-matrix multiply (blocked/tiled): Consider A,B,C to be N by N matrices of b by b subblocks where b=n/N is called the blocksize



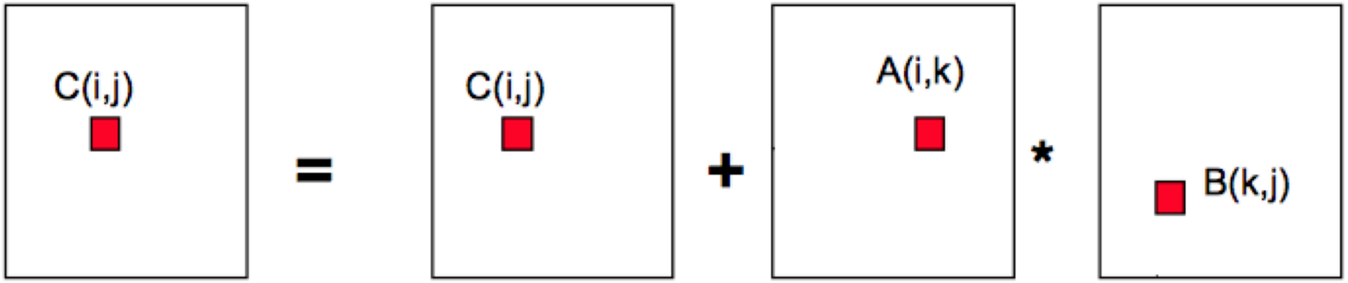
Memory Locality Model

Matrix Blocking

Some Examples of q (Computational Intensity)

- Matrix-matrix multiply (blocked/tiled): Consider A,B,C to be N by N matrices of b by b subblocks where b=n/N is called the blocksize

```
for (i = 0; i < N; ++i) {
  for (j = 0; j < N; ++j) {
  {read block C[i,j] into fast memory}
    for (k = 0; k < N; ++k) {
      {read block A[i,k] into fast memory}
      {read block B[k,j] into fast memory}
      C[i,j] += A [i,k] * B[k,j]; {do a matrix multiply on blocks}
{write block C[i,j] back to slow memory}
```



Memory Locality Model

Matrix Blocking

Some Examples of q (Computational Intensity)

- Matrix-matrix multiply (blocked/tiled): Consider A,B,C to be N by N matrices of b by b subblocks where b=n/N is called the blocksize

$m = N n^2$ (read each block of B N^3 times ($N^3 * (n/N)^2$)
$+ N n^2$ (read each block of A N^3 times)
$+ 2 n^2$ (read and write each block of C once)
$= (2N + 2) n^2$

$f = 2n^3$
$q = f / m \approx n/N = b$ for large n

- So we can improve performance by increasing the blocksize b
- Can be much faster than matrix-vector multiply (q=2)
- Limit: All three blocks from A,B,C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large: $3 b^2 <= M$, so q ~= b <= sqrt(M/3)
    - M = size of fast memory
- Theorem (Hong, Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to q =O(sqrt(M))

LOOP OPTIMIZATION

Loop Optimization

Loop Interchange

Loop Interchange

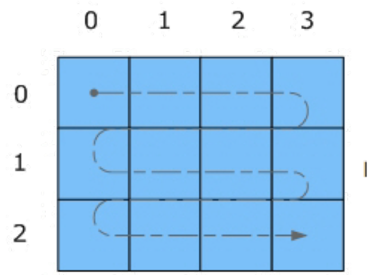
- Process of exchanging the order of two iteration variables used by a nested loop to improve spatial locality

```
for (int j = 0; j < n; j++) {
  for (int i = 0; i < n; i++)
  a[i] = a[i] + b[i,j] * c[i];
}
```

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++)
  a[i] = a[i] + b[i,j] * c[i];
}
```

stride-n

stride-1



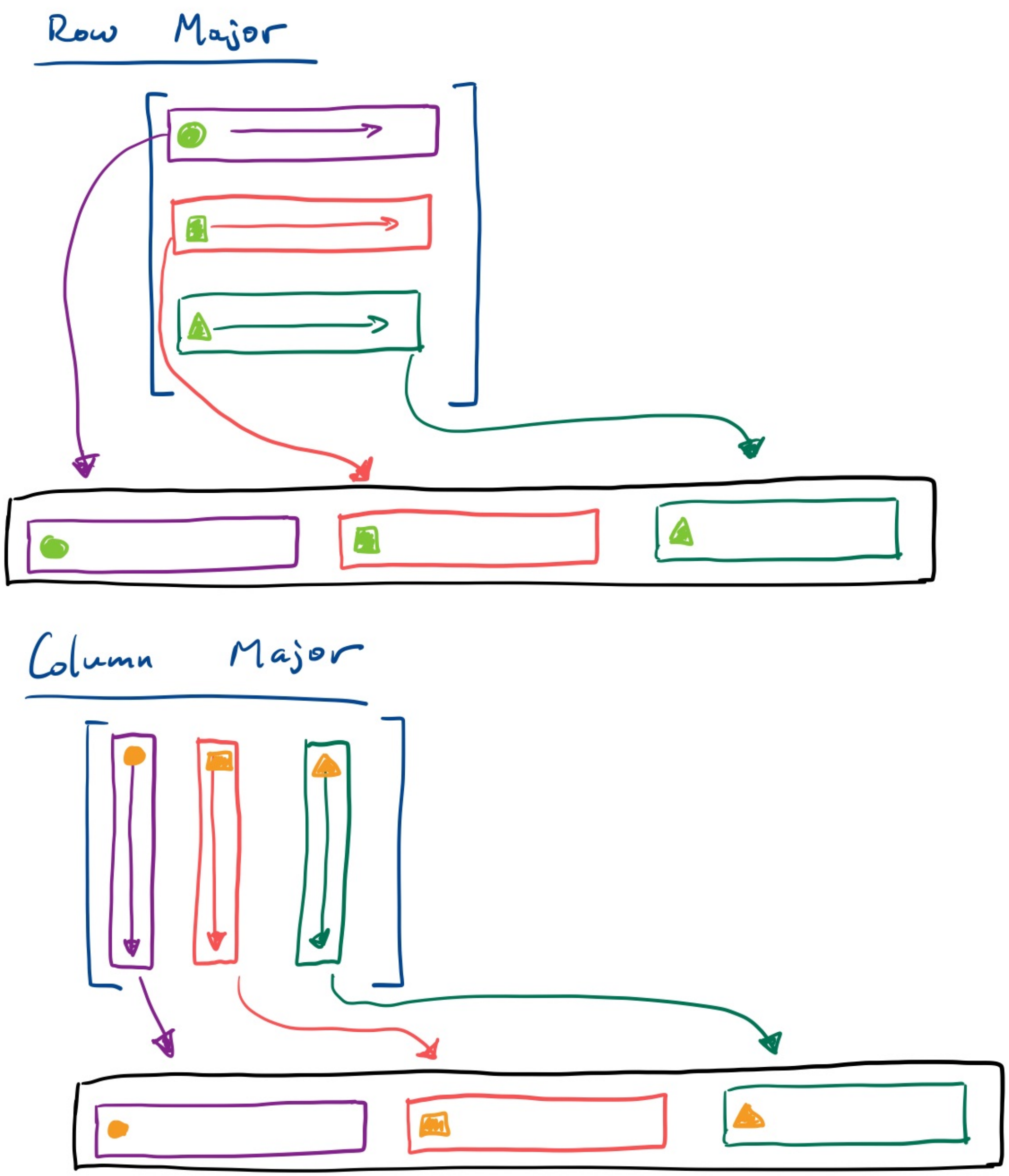
b[1,1], b[1,2], b[1,3]...

row-major order

✓ Good data spatial locality!
But, ruins the reuse of a(i) and c(i) in the inner loop, as it introduces two extra loads (for a(i) and for c(i)) and one extra store (for a(i)) during each iteration.

Loop Optimization

A Comment on row-major vs. column-major ordering



Loop Optimization

Loop Reversal

Loop Reversal

- Reverses the order in which values are assigned to the index variable

```
for (int j = 0; j < n; j++) {
  for (int i = 0; i < n; i++)
  a[i] = a[i] + b[i,j] * c[i];
}
```

stride-n

```
for (int j = 0; j < n; j++) {
  for (int i = 0; i < n; i++)
  a[i] = a[i] + b[j,i] * c[i];
}
```

stride-1

✓ No loop interchange!
✓ Programmer should change the way to store array data

Loop Optimization

Loop-Invariant Code Motion

Loop-Invariant Code Motion

- If result of a statement or expression does not change during loop, and it has no externally-visible side effect (!), can hoist its computation before loop

```
for (int i = 0; i < n; i++)
{
  x = y + z;
  a[i] = 6 * i + x * x;
}
```

→

```
x = y + z;
t = x * x;
for (int i = 0; i < n; i++)
{
  a[i] = 6 * i + t;
}
```

Loop Optimization

Strength Reduction

Strength Reduction

- Replace expensive operations (*,/) by cheap ones (+,−) via dependent induction variable

```
c = 7;
for (i = 0; i < N; i++){
  y[i] = c * i;
}
```

→

```
c = 7;
k = 0;
for (i = 0; i < N; i++{
  y[i] = k;
  k = k + c;
}
```

Loop Optimization

Unrolling

Loop Unrolling

- Branches are expensive – unroll loop to avoid them

```
int x;
for (x = 0; x < 100; x++{
  delete(x);
}
```

→

```
int x;
for (x = 0; x < 100; x += 5 ) {
  delete(x);
  delete(x + 1);
  delete(x + 2);
  delete(x + 3);
  delete(x + 4);
}
```

- Gets rid of 3/4 of conditional branches!
- Increase instruction parallelism

Loop Optimization

Loop Fission and Fusion

Loop Fission and Fusion

- Break loop into several loops, or merge multiple loops

```
int i, a[100], b[100];
for (i = 0; i < 100; i++){
  a[i] = 1;
  b[i] = 2;
}
```

```
int i, a[100], b[100];
for (i = 0; i < 100; i++){
  a[i] = 1;
}
for (i = 0; i < 100; i++){
  b[i] = 2;
}
```

- Reduce control and branches
- Can improve data temporal locality
- Improve instruction parallelism

- Can improve instruction temporal locality

Trial and Error!

COMPILER

Compiler

Generic Optimization Options - gcc

Option	Level	Execution Time	Code Size	Memory Usage	Compile Time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

Summary

The Optimization Process

Compile without optimization flags
Baseline execution time
Baseline results

↓

Compile with generic optimization flags
Get execution time (speedup)
Compare results (aggressive options)

↓

Use specific optimization flags
Get execution time (speedup)
Compare results (aggressive options)

↓

Adapt code to improve ILP and data locality
Get execution time (speedup)
Compare results (aggressive options)

↓

Change algorithm
Get execution time (speedup)
Compare results (aggressive options)

Next Steps

- Lab session this week (need it for the homework):
  - I4. Performance Optimization on AWS
  - I5. OpenACC on AWS (**request access to GPU-based instances!)**
- Get ready for **next lecture**:
  - B.3. Accelerated computing

Questions

Performance Optimization