Code Optimization Techniques

1

Code optimization for performance

- A quick look at some techniques that can improve the performance of your code
- Rewrite code to minimize processor cycles
  - But do not mess up the correctness!
  - Reduce number of instructions executed
  - Reduce the “complexity” of instructions
    - In real processors, different arithmetic operations can take different times
- Locality
  - Will improve memory performance

2

Recall CPU time model

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

CPU = IC * CPI * Clk

3

Summary: Memory Access time optimization

- If each access to memory leads to a cache hit then time to fetch from memory is one cycle
  - Program performance is good!
- If each access to memory leads to a cache miss then time to fetch from memory is much larger than 1 cycle
  - Program performance is bad!
- Design Goal:

How to arrange data/instructions so that we have as few cache misses as possible.

4

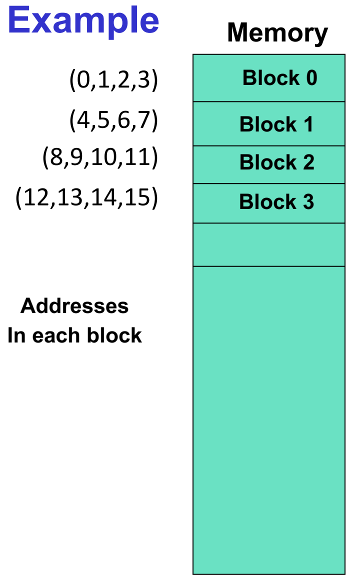
Exercise: Improve Cache Hit Rate by rewriting the code

- Assume array A[16]
- Assume cache block size = 4
- Assume total cache size= 2 blocks
- Assume

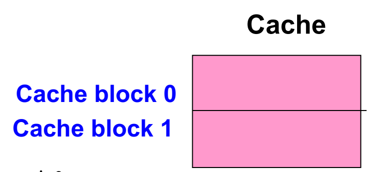
```
i=0;
while (i<16){
        A[i]= A[i] *10;
        i= i+4; }
i=1;
while (i<16){
        A[i] = A[i]*20;
        i=i+4; }
i=2;
while (i<16){
        A[i]=A[i]*30;
        i= i+4;}
i=3;
while (i<16){
        A[i]= A[i]*40;
        i= i+4;}
…
```

5

Example



4 Cache blocks
4 bytes in each block
Mem Block *x* maps into cache block (*y* mod 2)




```
i=0;
while (i<16){
        A[i]= A[i] *10;
        i= i+4; }
i=1;
while (i<16){
        A[i] = A[i]*20;
        i=i+4; }
i=2;
while (i<16){
        A[i]=A[i]*30;
        i= i+4;}
i=3;
while (i<16){
        A[i]= A[i]*40;
        i= i+4;}
…
```

Cache accesses:
Miss
Miss
Miss
Miss
Miss
….

6

Code with Improved locality

Cache accesses:

Miss
Hit
Hit
Hit
Miss
Hit
Hit
Hit
Miss
....

```
i=0;
while (i<16){
        A[i]= A[i] *10;
        A[i+1] = A[i+1]*20;
        A[i+2] = A[i+2]*30;
        A[i+3] = A[i+3]*40
```

7

Reducing CPU Time: Who can 'change' each parameter

- CPU time = IC * CPI * Clk
- Clock: completely under HW control
- IC: programmer and compiler
- CPI: compiler and HW
- ....so what does a compiler do?

8

Compiler Tasks

- 1. Code Translation
  - Source language → target language

    FORTRAN → C

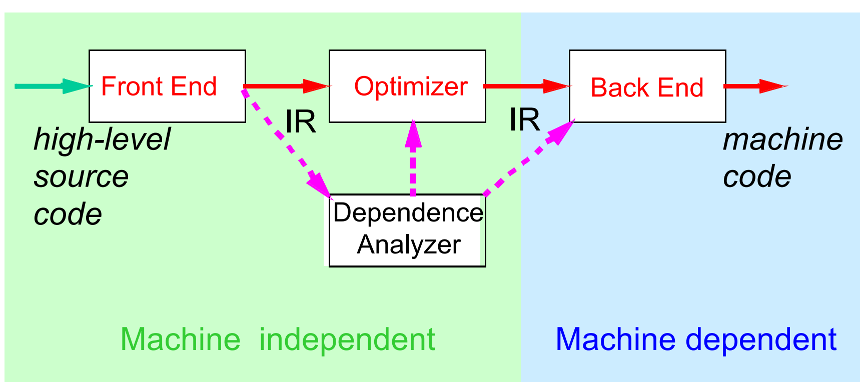
    C → MIPS, x86, PowerPC or Alpha machine code

    MIPS binary → x86 binary

- 2. Code Optimization
  - Code runs faster
  - Match dynamic code behavior to static machine structure

9

Compiler Structure



(IR= intermediate representation)

10

Compiler Front End tasks

- Lexical Analysis
  - Misspelling an identifier, keyword, or operator
    - e.g. lex

  done by a finite state machine (i.e., *deterministic finite automata*)!

- Syntax Analysis
  - Grammar errors, such as mismatched parentheses
  - Define syntax using *Context Free Grammar*…then build parser

  e.g. yacc

- Semantic Analysis
  - Type checking, check formal and actual arguments to function match, etc.
- code generation…*you've been doing this for C to LC3!!*
  - to target ISA or intermediate code, llvm-code

11

Code Optimization

- After front end analysis ⟹ an executable program P
- P has some performance T(P)
  - T(P) = IC*CPI* Clock
- Goal: Improve T(P)
  - Reduce time
  - How ? Reduce CPI and/or IC
- Rewrite/transform P to equivalent program Q such that
  1. T(Q) < T(P) and
  2. Q and P are equivalent, i.e, do exactly the same thing
     - For all inputs, Q and P produce the same result and compute the same function

12

Formal Model for Code Optimization ?

•Is it a hack job or is there a formal model underlying the various transformations that can help with designing a tool to optimize code ?

- *Need to make sure that transformed code is correct and does not change semantics of the original program.*

•Power of abstraction…..

•Graph theory: model program as a graph (Program dependence graph)

- Model data and control dependencies
- Code transformation = graph transformation

13

The Program Dependence Graph

- How to represent control and data flow of a program ?
- The Program Dependence Graph (PDG) is the intermediate (abstract) representation of a program designed for use in optimizations
- It consists of two important graphs:
  - Control Dependence Graph captures control flow and control dependence
  - Data Dependence Graph captures data dependences
- Analogous to a flow-chart of the program
  - Formal model for flow charts!

14

Definition: Control Flow Graph

A control flow graph CFG = (N_c ; E_c ; T_c) consists of

- N_c, a set of nodes. A node represents a straight-line sequence of operations with no intervening control flow i.e. a basic block.
- $E_c \subseteq N_c \times N_c \times Labels$, a set of *labeled* edges.

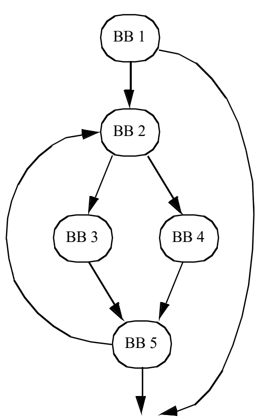
- Example: the code below has two basic blocks

```
ADD R0, R0, #0
BRn here1
```

```
LDR R1, R0, #0
ADD R2, R1, R2
BRzp here2
```

15

Control Flow Graph




```
main:
        addi r2,  r0,  A        |
        addi r3,  r0,  B        |
        addi r4,  r0,  C        |  BB 1
        addi r5,  r0,  N        |
        add  r10, r0,  r0       |
        bge  r10, r5,  end      |
loop:
        lw   r20,  0(r2)        |
        lw   r21,  0(r3)        |  BB 2
        bge  r20, r21, T1       |
        sw   r21,  0(r4)        |  BB 3
        b    T2                 |
T1:
        sw   r20,  0(r4)        |  BB 4
T2:
        addi r10, r10, 1        |
        addi r2,  r2,  4        |
        addi r3,  r3,  4        |  BB 5
        addi r4,  r4,  4        |
        blt  r10, r5,  loop     |
end:
```

16

Data Dependence Graph

- Within each basic block capture the data dependencies between instructions
  - follow the data in the registers
  - Value computed in a register is needed by an instruction in the future
  - Ex:
    - Value computed by LDR is needed by next instruction
    - But no dependence between AND and the other instructions
- Can capture these dependencies using a graph
  - Nodes are instructions and edges dependencies
- Data dependencies important in
  - Scheduling instructions
  - Parallelizing the code

```
LDR R1, R0, #0

ADD R2, R1, R2
AND R3, R3, #4
BRzp here2
```

17

Program behaviour ?

- Model as program dependence graph!
- What is a correct execution ?
  - Execution will only follow valid paths in the program dependence graph!
    - **IF code is written correctly, then force the program to only follow paths in the dependence graph!**
- connection to Software security/correctness
  - Only execute along paths in the graph = program cannot execute any malicious code

18

Formal Definition/Model: Code Optimization

• First goal: make sure that transformed code is correct and does not change semantics of the original program.

•model program as a graph (Program dependence graph)

- Model data and control dependencies

•Any transformation should give us a homomorphic graph

- Recall concept of Isomorphism/Homomorphism Discrete Structures courses !!!

•Bad news: checking graph isomorphism is NP-complete !

- Therefore … ???

19

Compiler optimizations

•Use 'heuristics' to solve the difficult problem

•All 'useful' compilers have code optimizers built into them

- Optimize time….
- other metrics: power ? Code size?
  - Why ?

•Machine dependent optimizations

- Need to know something about the processor details before we can optimize

•Machine independent optimizations

- These are independent of processor specifics
- These are what we will cover

20

Machine Dependent Optimizations

These need some knowledge of the processor

- Register Allocation

- Instruction Scheduling

- Peephole Optimizations

21

Instruction Scheduling

- Given a source program P, schedule the instructions so as to minimize the overall execution time on the functional units in the target machine
  - This is where processors with parallelism introduce complexity into the scheduling process
  - Schedule parallel instructions

- the instruction flow (data dependencies) are again modeled as a graph !

- Finding a schedule with minimum execution time is an NP-complete problem
  - Need fast and effective heuristics
  - You will cover schedulers in Operating Systems course

23

Register Allocation

- Storing and accessing variables from registers is much faster than accessing data from memory.
  - Variables ought to be stored in registers
- It is useful to store variables as long as possible, once they are loaded into registers
- Registers are bounded in number
  - "register-sharing" is needed over time.
  - Some variables have to be 'flushed' to memory
  - Reading from memory takes longer
- how important is Register allocation to performance?
  - **efficient register allocators improved performance 25%**
  - Poor allocation means repeatedly reading variables from memory

24

Register Allocation

```
{ …
      i=10;
      x= y +i;
      while (i<100) {
            a = a*100
            b = b + 100
            i++;
      }
```

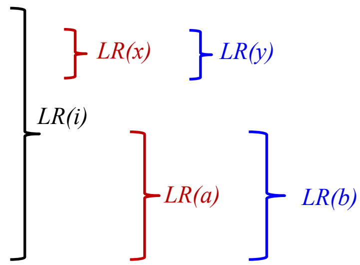
Each variable gets placed into a register, ex: LDR R0, R5, #-3 puts local var with offset -3 into R0

- Suppose you have 3 registers available…and 5 variables
- should you place a and b into same register ?
- Can you place x and a into same register ?

25

Register Allocation

```
{ …
      i=10;
      x= y +i;
      while (i<100){
            a = a*100
            b = b +100
            i++;
      }
```



- "live range" *LR(j)* for each variable *j* – where is it accessed
- Do live ranges of x and a "interfere" : LR(x) & LR(a) = 0?
- Do live ranges of a and b interfere ? : LR(a) & LR(b) = 0?
- If ranges interfere, then assign to different registers

26

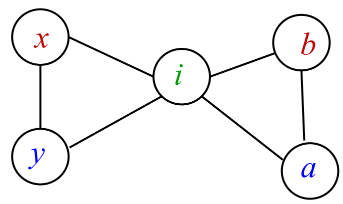
Register Allocation: Problem Formulation and Solution

- Determine live ranges for each variable, and determine conflicts/interference between variables/live ranges
  - Using dataflow analysis compute live ranges for each variable
- How do we model the register allocation problem?
  - Power of abstraction!!
- Formulate the problem of assigning variables to registers as a graph problem: The Graph coloring problem !
  - Nodes in graph = number of variables (live ranges)
  - Edges in graph = edge between x,y if live ranges x,y interfere
  - Can we color the graph with K colors ?
  - Number of colors = Number of registers!
  - Use application domain (Instruction execution) to define the priority function

27

Example:

- Graph theory & CS – it is every &#@&@ place!
  - My curriculum advice (that nobody takes…except 3): take a graph theory course!



3 registers = 3 colors

x,and b assigned to same register since no edge

y and a assigned to same register since no edge

i cannot be assigned to same register as any of the others

28

Machine Dependent Optimizations

- Need thorough knowledge of the architecture AND algorithms
- New architectures introduce new challenges…
  - Multi-core, Multi-threaded, Embedded (need to optimize for power consumption), Security to enforce software security)
  - Compiling for FPGA co-processors to accelerate (ex: AWS, Microsoft)
- Compiling for Security – leverage FPGAs & extra HW to place verification and encryption circuits
- Compiling for power optimization
  - control memory power using compiler….layout the data so we can switch off memory modules

My Research Areas

- Machine dependent optimizations done by a compiler writer….
  - Huge demand in industry….23 requests (>150K) in 2022 from alumni
  - But few CS students want to study this stuff ☹ …and, this is not our focus for now!

29

Our focus in this course: Machine Independent Optimizations

- As SW developers, these should be a ‘default’ when you write code…
  - THIS is what separates you from those who take a single programming course and claim they know CS!!
- How does it work: a large ‘menu’ of optimization techniques
  - Some dependent on general architecture
    - Ex: Pipelined processors and loop unrolling
  - We cover a small sample that works on all processors

30

Some Machine-Independent Optimizations

- Some easy ones: Dataflow Analysis and Optimizations
  - Constant folding, Copy propagation etc.
  - Elimination of common subexpression
  - Dead code elimination
- Code motion
- Strength reduction
- Function/Procedure inlining
- Improving memory locality

31

Code-Optimizing Transformations

- *Constant folding*

(1 + 2) ⇒ 3

(100 > 0) ⇒ true

This saves one instruction – *reduce IC*

32

Code-Optimizing Transformations

- *Copy propagation*

```
x  =  b + c                x  = b + c
z  =  y * x    ⇒           z  = y * (b + c)
```

Why does this make a difference: Recall how code is generated..

(b+c) is stored into a temp register R0 and then STR R0, R5, #-2 to store local var x.
Code generated for the 2nd statement **z = y*x** is:

```
LDR R0, R5, #-2    ; Load x into R0
LDR R1, R5, #-3    ; load y into R1
MUL R2, R0, R1     ; multiply x,y and store into R2
Replace above with
LDR R1, R5, #-3    ; load y into R1
MUL R2, R0,R1      ; multiply with value (b+c) stored in R0
```

This saves one memory access..reduces IC and CPI

33

Code-Optimizing Transformations

- Common subexpression – reduce instruction count

```
x  =  b * c + 4         ⇒        t  = b * c
z  =  b * c -  1                 x  = t + 4
                                 z  = t – 1
```

  - 2 mult, 1 add, 1 sub replaced by
  - 1 mult, 1 add, 1 sub

  - Reduces IC

34

Code-Optimizing Transformations

- Dead code elimination

```
x  =  1
x  =  b + c
```

or if x is not referred to at all

Saves one instruction…reduce IC

35

Code Optimization Example

```
x = 1
y = a * b + 3
z = a * b + x + z + 2
x = 3
```

propagation →

```
x = 1
y = a * b + 3
z = a * b + 1 + z + 2
x = 3
```

↓ constant folding

```
x = 1
y = a * b + 3
z = a * b + 3 + z
x = 3
```

← dead code elimination

```
y = a * b + 3
z = a * b + 3 + z
x = 3
```

↓ common subexpression

```
t = a * b + 3
y = t
z = t + z
x = 3
```

Original: 2 Mult, 4 Add, 7 Read/Write Mem

New: 1 Mult, 2 Add, 5 Read/Write

36

Code Motion

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop
- Move code between blocks
  - eg. move loop invariant computations outside of loops
- What does this reduce ?
  - Number of times x/y is computed…reduce IC

```
while ( i < 100 ) {
  *p = x / y + i
   i = i + 1
  }
```

→

```
t = x / y
while ( i < 100 ) {
  *p = t + i
   i = i + 1
}
```

37

Strength Reduction

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  16*x --> x << 4

  - Utility is machine dependent
  - Depends on cost of multiply or divide instruction
  - On Pentium x86, integer multiply only requires 4 CPU cycles
- Recognize sequence of products

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

→

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

39

Function Inlining

- What happens on a function call ?
  - How are function calls implemented on the machine ?
  - Is function call = one subroutine call ?
- Function call in C = number of instructions in machine code
  - Create activation records, allocate memory
  - Manipulate stack and frame pointers
- What happens if we replace function call with body of function
  - i.e., Inline the function

41

Function Call/Return

- Instructions to Push arguments to stack
- Instructions to Push frame pointer, return addr.
- Execute instructions of function
- Instructions to Pop return value, reset frame pointer, pop return address
- The bookkeeping instructions are essentially an "overhead"
  - They do not do the work of the function
- What happens if we replace function call with body of function ?
  - **Inline the function**
  - **Remove the function call and return overhead instructions**
  - **...**reduce IC

42

Function Inlining

```
…                                   int myfunc(int m,n)
x= myfunc(i,j)          {
…                                            return(m+n);}
```

After inlining:

```
…
x = m+n
…..
```

- Improves performance
  - Removes bookkeeping instructions
- but tradeoff with code readability
  - and code size

43

Code Optimization Techniques: Part 2

44

Machine-Independent Code Optimizations

- Dataflow Analysis and Optimizations
  - Constant folding, Copy propagation etc.
  - Elimination of common subexpression
  - Dead code elimination
- Code motion
- Strength reduction
- Function/Procedure inlining
- Improving memory locality
- Example/Exercise: An example is posted for you to try rewriting code

45

Locality

- Recall Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - Temporal locality: Recently referenced items are likely to be referenced in the near future.
  - Spatial locality: Items with nearby addresses tend to be referenced close together in time.

- *Being able to look at code and get a qualitative sense of its locality is a key skill for a professional software developer.*

46

Locality and performance

- Recall: Memory = Cache + Main memory
  - Cache contains small number of bytes
- Recall: cache is arranged as a set of blocks
  - Can only fetch block at a time
- ExampleAssume each cache block has 4 words
  - If you fetch a block with addresses {0,1,2,3}
  - If four successive instructions use locations 0,1,2,3 then we only have one cache miss (first time to fetch block into cache)
  - If four successive instructions use locations 0,4,8,12 then each time we have to fetch a new cache block
    - Each memory access is an access to main memory
- Goal: have locality in memory accesses

47

Exercise: Improve Cache Hit Rate by rewriting the code

- Assume array A[16]
- Assume cache block size = 4
- Assume total cache size= 2 blocks

- Array access pattern:

A[0], A[4], A[8], A[12], A[1], A[5], A[9],…

```
i=0;
while (i<16){
        A[i]= A[i] *10;
        i= i+4; }
i=1;
while (i<16){
        A[i] = A[i]*20;
        i=i+4; }
i=2;
while (i<16){
        A[i]=A[i]*30;
        i= i+4;}
i=3;
while (i<16){
        A[i]= A[i]*40;
        i= i+4;}
…
```

48

Example

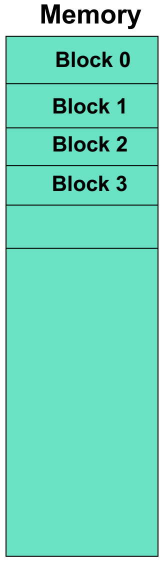
Memory

(0,1,2,3)

(4,5,6,7)

(8,9,10,11)

(12,13,14,15)



Addresses In each block

2 Cache blocks, 4 bytes in each block

Mem Block *x* maps into cache block (*y* mod 2)

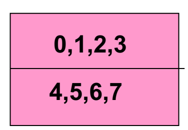
Access pattern:

0,4,8,12,1,5,9,13,2,…

Cache

Cache block 0

Cache block 1




```
i=0;
while (i<16){
        A[i]= A[i] *10;
        i= i+4; }
i=1;
while (i<16){
        A[i] = A[i]*20;
        i=i+4; }
i=2;
while (i<16){
        A[i]=A[i]*30;
        i= i+4;}
i=3;
while (i<16){
        A[i]= A[i]*40;
        i= i+4;}
…
```

Cache accesses:

Miss

Miss

Miss

Miss

Miss

….

49

Code with Improved locality

```
i=0;
while (i<16){
        A[i]= A[i] *10;
        A[i+1] = A[i+1]*20;
        A[i+2] = A[i+2]*30;
        A[i+3] = A[i+3]*40 ; }
```

Cache accesses:

Miss
Hit
Hit
Hit
Miss
Hit
Hit
Hit
Miss
….

Array access pattern:

A[0], A[1], A[2], A[3], A[4], A[5], A[6],A[7],….

50

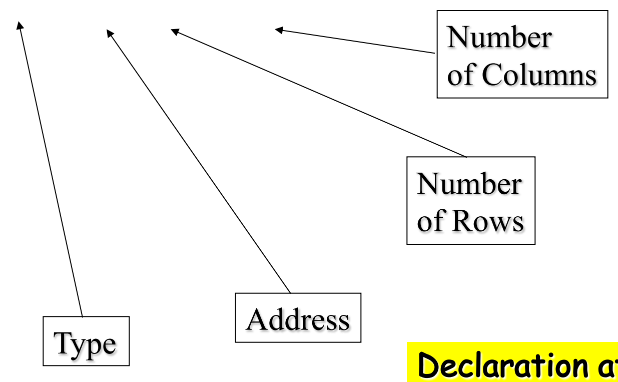
Arrays and Memory organization… . .

- Let's use array data structures to guide our discussions
- Recall: accesses to cache better than accesses to main memory/disk
- Recall: Multidimensional Arrays

51

Declaration

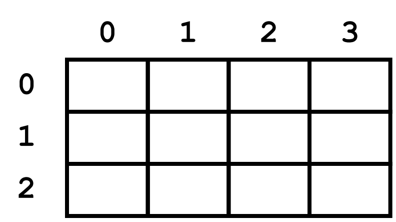
```
int ia[3][4];
```



Declaration at compile time i.e. size must be known

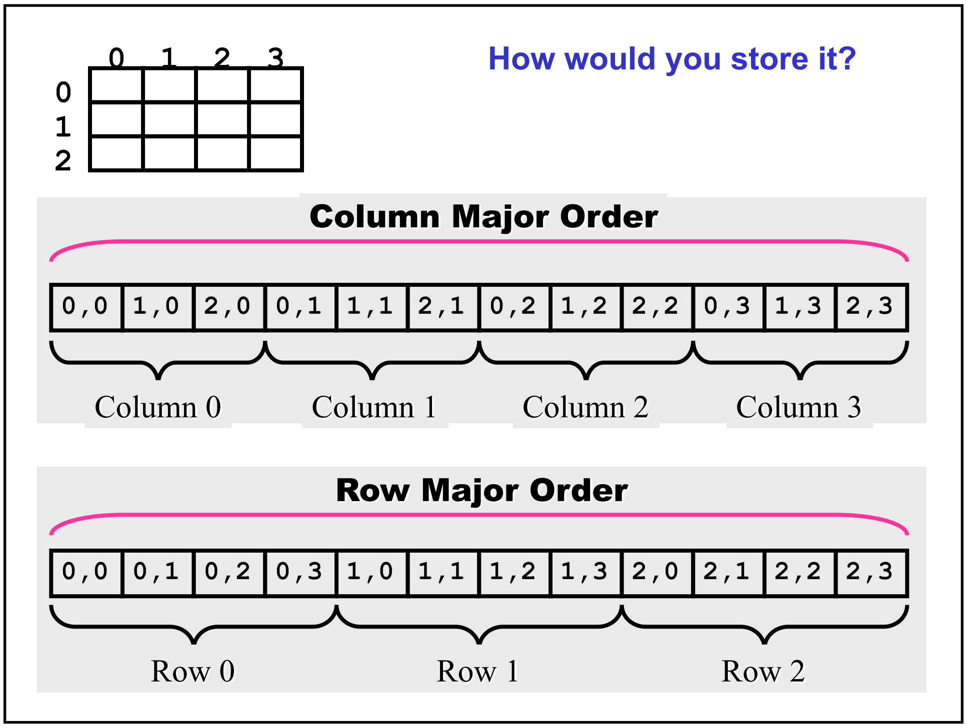
52

How does a two dimensional array work?

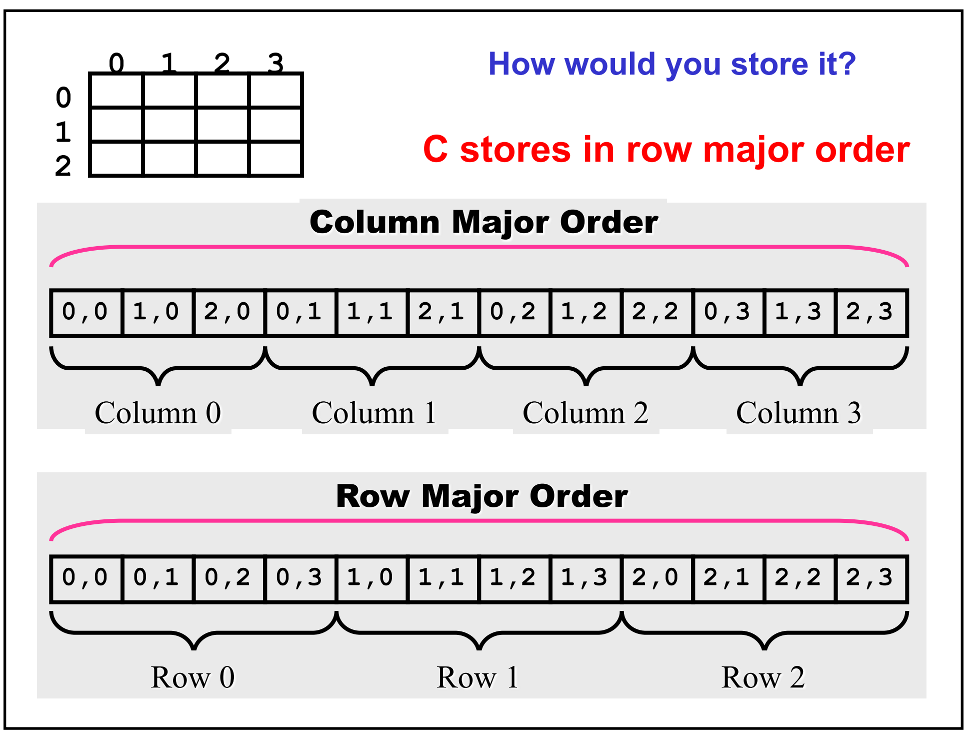


How would you store it?

53


0 1 2 3
0
1
2
How would you store it?
Column Major Order
0,0 1,0 2,0 0,1 1,1 2,1 0,2 1,2 2,2 0,3 1,3 2,3
Column 0
Column 1
Column 2
Column 3
Row Major Order
0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3
Row 0
Row 1
Row 2

54


0 1 2 3
0
1
2
How would you store it?
C stores in row major order
Column Major Order
0,0 1,0 2,0 0,1 1,1 2,1 0,2 1,2 2,2 0,3 1,3 2,3
Column 0
Column 1
Column 2
Column 3
Row Major Order
0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3
Row 0
Row 1
Row 2

55

Question to ask: Locality of Access

- How are elements in the array accessed in your program ?
  - Row major or column major or other ?
  - How would you iterate over the 2-D array to maintain locality ?

56

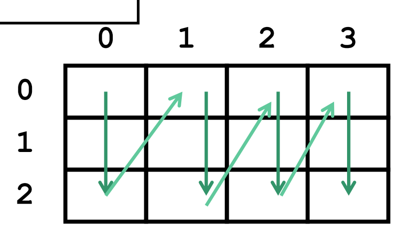
Locality Example

- Question: Does this function have good locality?

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum
}
```

Column-major
Access pattern



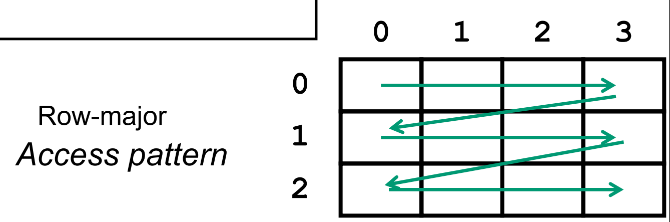
57

Locality Example

- Question: Does this function have good locality?

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum
}
```



Improving Memory Access Times (Cache Performance) by Compiler Optimizations

- McFarling [1989] improve perf. By rewriting the software
- Instructions
  - Reorder procedures in memory so as to reduce cache misses
  - Code Profiling to look at cache misses(using tools they developed)
- Data
  - *Merging Arrays*: improve spatial locality by single array of compound elements vs. 2 arrays
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Blocking*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

Compiler optimizations – merging arrays

- This works by improving spatial locality
- For example, some programs may reference multiple arrays of the same size at the same time
  - Could be bad – not enough locality
    - Accesses may interfere with one another in the cache – conflict misses
- A solution: Generate a single, compound array/struct…

60

Merging Arrays Example

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of stuctures */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];


Reducing conflicts between val & key;
  improve spatial locality
```

61

Compiler optimizations – loop interchange

- Some programs have nested loops that access memory in non-sequential order
  - Simply changing the order of the loops may make them access the data *in* sequential order…
- What's an example of this?
  - Recall: C stores 2-D arrays in row-major format

62

Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
      for (i = 0; i < 5000; i = i+1)
            x[i][j] = 2 * x[i][j];
```

63

Loop Interchange Example

```
/* After */
for (k = 0; k < 100; k = k+1)
 for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
         x[i][j] = 2 * x[i][j];
```

```
Sequential accesses instead of striding
  through memory every 100 words;
  improved spatial locality
```

64

Compiler optimizations – loop fusion

- This one's pretty obvious once you hear what it is…
- Seeks to take advantage of:
  - Programs that have separate sections of code that access the same arrays in different loops
    - Especially when the loops use common data
  - The idea is to "fuse" the loops into one common loop
- What's the target of this optimization?
  - Locality – reduce memory access times
  - IC – by reducing number of branches
    - Important in pipelined processors

65

Loop Fusion Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
      a[i][j] = 1/b[i][j] * c[i][j];

for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
      d[i][j] = a[i][j] + c[i][j];
```

Move inside first loop

66

Loop Fusion Example

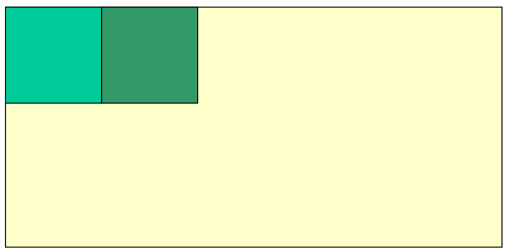
```
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {   a[i][j] = 1/b[i][j] * c[i][j];
      d[i][j] = a[i][j] + c[i][j];}
```

2 misses per access to a & c vs. one miss per access; improve spatial locality & temporal locality

67

And last (but most important?): general concept of Memory Blocking.

- Can you keep locality in all memory operations
- This is probably the most "famous" of compiler optimizations to improve cache performance
- common concept: blocking
  - *Rewrite code to process blocks of data at a time*
  - Size of block = ???? Size of cache block!!



68

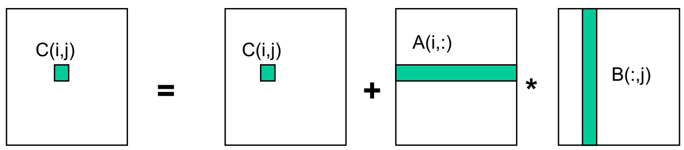
Compiler optimizations – blocking

- Tries to reduce misses by improving temporal locality and spatial locality
- To get a handle on this, you have to work through code on your own
- this is used mainly with arrays!
- Simplest case??
  - Row-major access

69

Naïve Matrix Multiply

{implements C = C + A*B}
for i = 1 to n
 {read row i of A into fast memory}
  for j = 1 to n
   {read C(i,j) into fast memory}
   {read column j of B into fast memory; note column major access!}
   for k = 1 to n
    C(i,j) = C(i,j) + A(i,k) * B(k,j)
   {write C(i,j) back to slow memory}



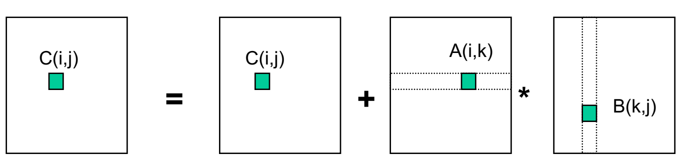
Good locality in access to matrix A; poor locality in access to B

70

70

Blocked (Tiled) Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where
 b= N/m is called the block size
  for i = 1 to N *increment by block size*
   for j = 1 to N *increment by block size*
    {read block of C(i,j) into fast memory}
    for k = 1 to N *increment by block size*
     {read block of A(i,k) into fast memory}
     {read block of B(k,j) into fast memory}
      C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
    {write block C(i,j) back to slow memory}



Work these details out….need it for the project!

71

Code Optimization and Compilers

- Modern compilers provide a menu of code optimization features
  - Inlining, strength reduction, register allocation, loop optimizations, etc.
- Some provide default optimization levels
  - Example: gcc -03 test.c
- Bottom Line: Everyone wants to run optimized code
  - It's about being smart with your solution!
- Have we seen everything there is to code optimization?....not by a long shot !!
  - Lots and lots more optimization techniques
    - The "cooler" ones need architecture knowledge

72

Example of Code Optimization: (Final) Project 6

- Topic: Code Performance Optimization
  - Given code for Image operations, rewrite the code to make it run faster.
    - Use only techniques covered in class.
    - *No collaboration….and no posting on Chegg.com*
- Description will be posted today and code on last day of classes and due official final exam date: **Dec. 15th midnight**.
  - Should take you 10-12 hours to complete
- Requires:
  - Code rewriting & Report writing: summarize your experiments, explain why the code ran faster (or slower).
- Very Important: Grade will depend on your analysis – simply turning in code (with documentation) that runs faster but no report will result in 0 points!

73