Speed Optimization Tour with Spherical Geometry

BASIC OPTIMIZATION CONCEPTS

Starting Points for Efficient Code

Premature optimization is the root of nearly all evil.

Donald Knuth

1. Write correct code
2. Optimize ***as little of it*** as possible

- Use the ***right tools*** for the problem, and use the ***tools right***
  - Know your programming language, compiler, and CPU architecture
  - Verify the compiler is doing a ***good enough*** job

Starting Points for Efficient Code

- Write correct code, then optimize.
- Use a top-down approach.
- Know your microprocessor's architecture, compiler, and programming language.
  - Use the right tool for the problem
- Leave assembly language for un-ported designs, interrupt service routines, and frequently used functions.

Do Less Work at Run-Time

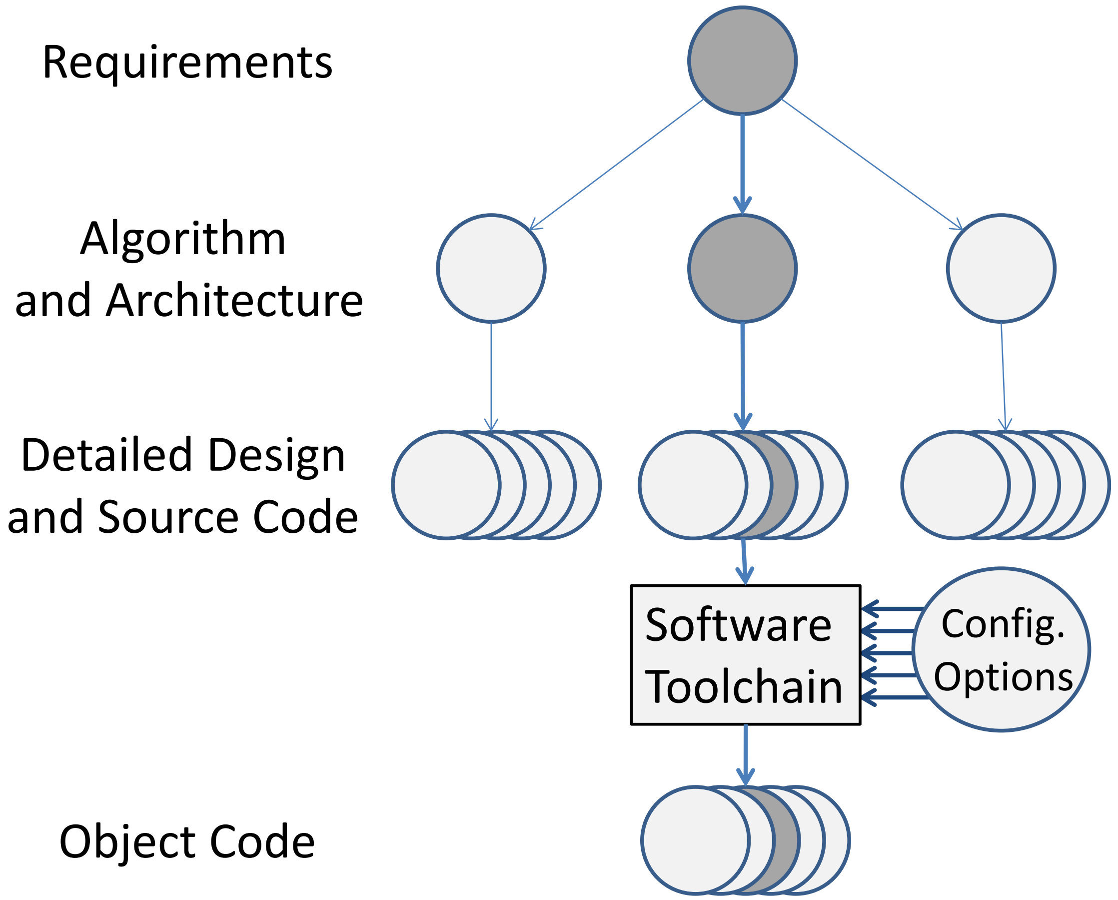
- Fundamental concept: perform less computation at run-time
  - Lazy (or deferred) execution: don't compute data until needed
  - Early decisions: for decisions based on computations, may be able to use intermediate results

- Applied broadly
  - Many algorithms implement these concepts
  - Compilers try to apply these in optimizations passes

- Role of developer
  - Help the compiler apply these concepts
  - Implement concepts directly in source code

Optimization & The Software Development Process

- Process overview
  - Select algorithm and architecture based on requirements and constraints
  - Create detailed design and source code
  - Compile and evaluate object code
  - Optimize by changing
    - Toolchain options
    - Source code
    - Detailed design & source code
    - etc.
- Many possible designs and implementations
- Profiler shows *what* to optimize
- *How* to optimize?
  - Low-level – based on toolchain and source code details
  - High-level – based on algorithms (and source code too!)



Overview of Optimization Process

- *Can optimize for speed or size, and sometimes both*
- Avoid unnecessary work
  - Start at a high level and think about how to minimize how much work must be done
- Do the necessary work quickly
  - Use efficient algorithms
    - E.g. bubble sort vs. quicksort, parallelization
  - Implement using efficient coding practices
    - Make it easy for compiler to create good code
    - Use appropriate techniques for the target processor
      - Fixed vs. floating point, data sizes, …
  - Compile with optimizations turned on
    - e.g. -O3 for speed
- Execute and profile program to find worst parts
- Look at the assembly code – is it good enough?

Your Results Will Vary

- Different programs will have different bottlenecks
- Bottlenecks may depend on input data
- Bottlenecks may move after optimizing the code
- Different processor architectures may create different bottlenecks in a program
- Different compilers may create different bottlenecks in a program
- Different compiler settings may create different bottlenecks in a program

Optimization Risks

- Hard to predict development effort needed
  - Balancing act
    - Pro: expected performance gain
    - Cons: additional development time requires, increased schedule risk
  - Difficulties in prediction
    - How much faster will the code be after this optimization? Will it be fast enough so we can stop optimizing the program?
    - How long will it take to perform this optimization?
    - How many more optimizations will we need?

- Impact on code maintainability
  - Code will be used in future
    - Bug fixes, feature changes, feature additions, upgrades
    - Basis for follow-up and evolved products, platform for range of products
  - What if you've forgotten how your optimized code works?
  - What if someone else needs to maintain your optimized code?
  - Optimization often hurts code maintainability
  - Need to optimize in a way which retains maintainability

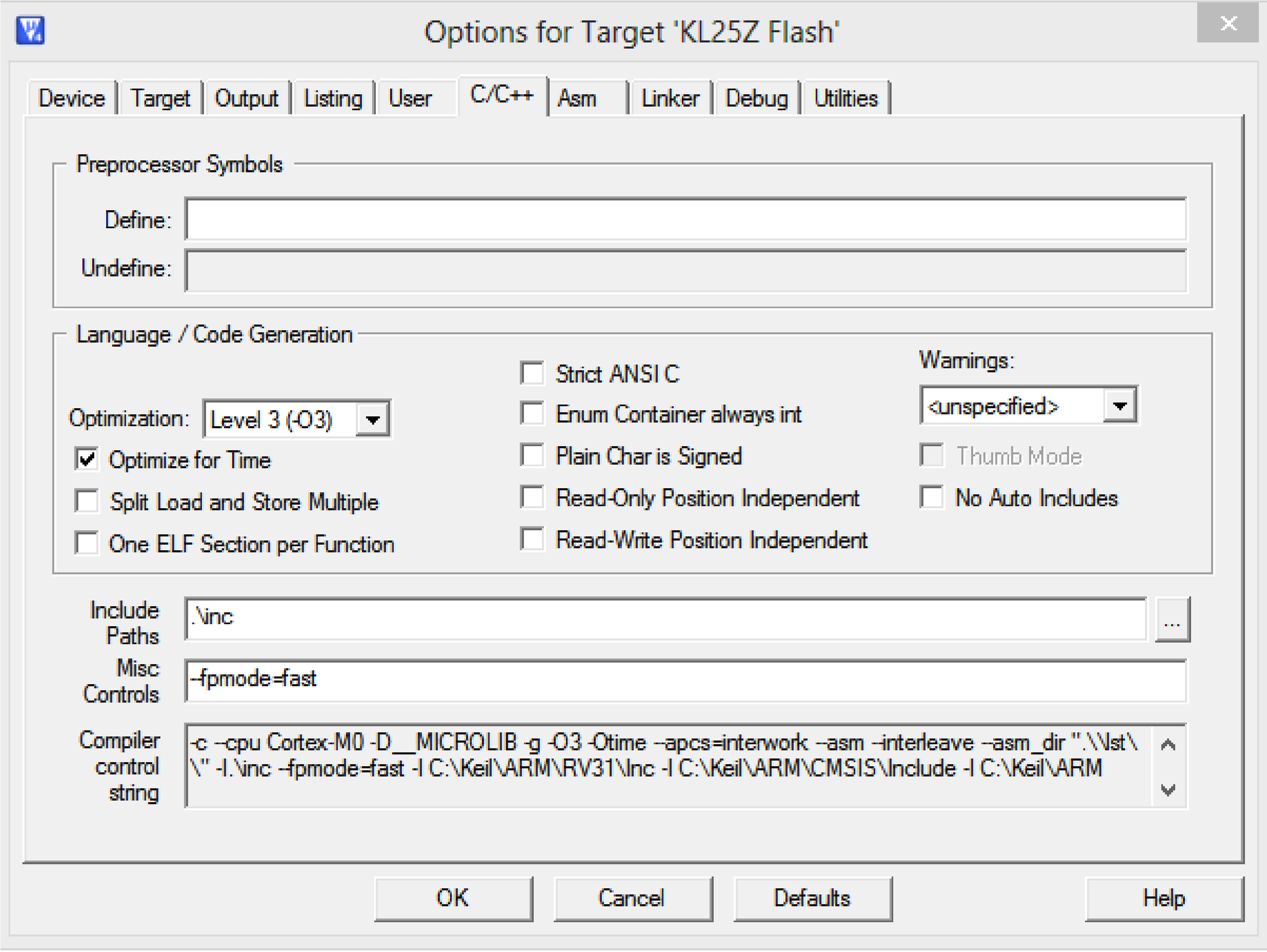
TOOLCHAIN CONFIGURATION

Review of Compiler Stages

- Parser
  - reads in C code,
  - checks for syntax errors,
  - forms intermediate code (tree representation)
- High-Level Optimizer
  - Modifies intermediate code (processor-independent)
- Code Generator
  - Creates assembly code step-by-step from each node of the intermediate code
  - Allocates variable uses to registers
- Low-Level Optimizer
  - Modifies assembly code (parts are processor-specific)
- *Assembler*
  - *Creates object code (machine code)*
- *Linker/Loader*
  - *Creates executable image from object file*

Compiler Optimization Settings

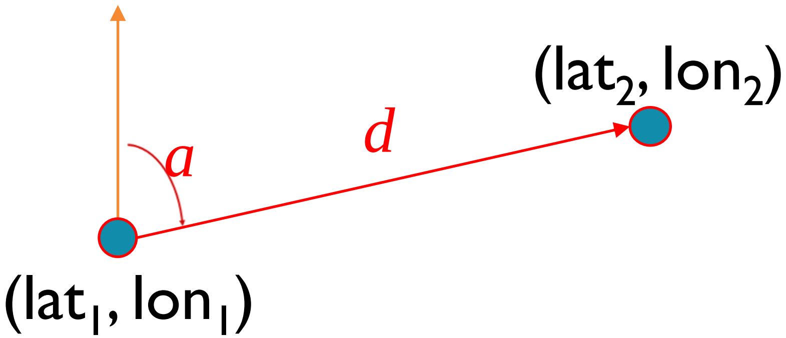
- Select
  - –O3 optimization (maximum)
  - Optimize for speed
- Unselect
  - Strict ANSI C – don't need this!

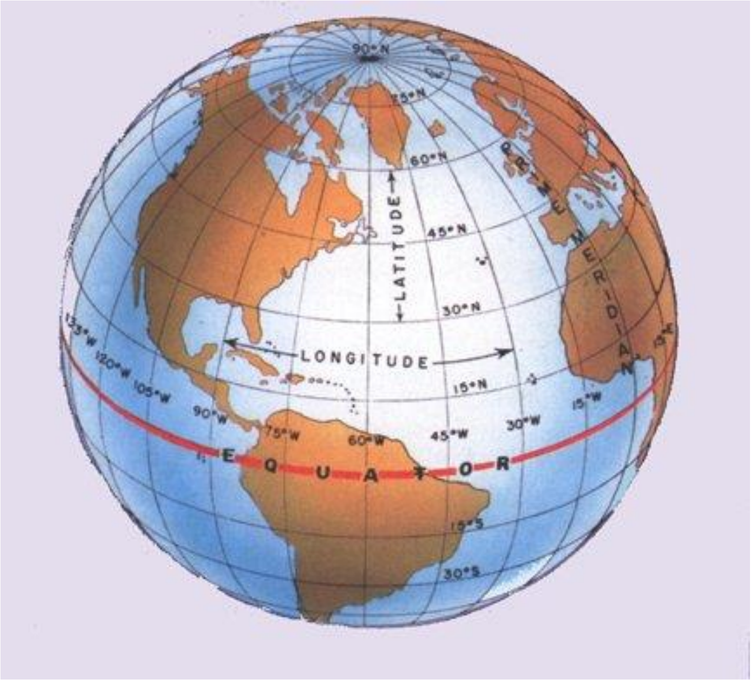


SAMPLE PROGRAM FOR OPTIMIZATION

Example Program: "Nearby Points of Interest"

- Find distance and bearing from current position to closest of a fixed set of positions
- Positions are described as coordinates on the surface of the Earth (latitude, longitude)





$$d = \mathrm{acos}((\sin(lat_1) * \sin(lat_2) + (\cos(lat_1) * \cos(lat_2) * \cos(lon_2 - lon_1)) * 6371$$

$$a = atan2(\cos(lat_1) * \sin(lat_2) - \sin(lat_1) * \cos(lat_2) * \cos(lon_2 - lon_1)\,,\ \sin(lon_2 - lon_1) * \cos(lat_2)) * \frac{180}{\pi}$$

Core Code: Calculate Distance

NC STATE UNIVERSITY

```
float Calc_Distance( PT_T * p1,  const PT_T * p2) {
// calculates distance in kilometers between locations
  return acos(sin(p1->Lat*PI/180)*
             sin(p2->Lat*PI/180) +
               cos(p1->Lat*PI/180)*cos(p2->Lat*PI/180)*
               cos(p2->Lon*PI/180 - p1->Lon*PI/180)) * 6371;
}
```

Core Code: Calculate Bearing

```
float Calc_Bearing( PT_T * p1,  const PT_T * p2){
// calculates bearing from p1 to p2 in degrees
  float angle = atan2(
       sin(p1->Lon*(PI/180) - p2->Lon*(PI/180))*
      cos(p2->Lat*(PI/180)),
       cos(p1->Lat*(PI/180))*sin(p2->Lat*(PI/180)) -
       sin(p1->Lat*(PI/180))*cos(p2->Lat*(PI/180))*
      cos(p1->Lon*(PI/180) - p2->Lon*(PI/180))
       ) * (180/PI);

  if (angle < 0.0)
       angle += 360;
  return angle;
}
```

Find Nearest Point

```
void Find_Nearest_Waypoint(
float cur_pos_lat, float cur_pos_lon,
float * distance, float * bearing,
char  * * name) {
  // cur_pos_lat, cur_pos_lon: deg.
  // distance: km
  // bearing: degrees

  // Initialization code removed

  while (strcmp(waypoints[i].Name,
    "END"))
  {
    d = Calc_Distance(&ref,
        &(waypoints[i]) );
    b = Calc_Bearing(&ref,
        &(waypoints[i]) );

    // remember closest waypoint
    if (d<closest_d) {
      closest_d = d;
      closest_i = i;
    }
    i++;
  }
  d = Calc_Distance(&ref,
      &(waypoints[closest_i]) );
  b = Calc_Bearing(&ref,
      &(waypoints[closest_i]) );
…
}
```

Data: Point Table

```
typedef struct {
  float Lat;
  float Lon;
  char Name[24];
} PT_T;

const PT_T waypoints[] = {
  //   Lat          Lon           Name
  {    56.07553,    152.57224,    "ALBATROSS BNK"     },
  {    51.15329,    -179.0052,    "AMCHITKA"   },
  {    59.38128,    153.35352,    "AUGUSTINE ISLAND, AK"     },
  {    22.02867,    94.058737,    "BAY CAMPECHE"      },
  {    57.07501,    177.75757,    "BERING SEA"  },
  {    30.09335,    88.773624,    "BILOXI"     },
  {    60.84875,    146.88753,    "BLIGH REEF LIGHT, AK"     },
……………… // many entries deleted
  {    19.87879,    85.064566,    "YUCATAN CHNL"      },
  {    0       ,    0        ,    "END"  },     };
```

EXAMPLE EVALUATION OF COMPILER'S ACTUAL OPTIMIZATIONS

Sample Code

```
#define PI 3.14159265
float Calc_Distance( PT_T * p1,  const PT_T * p2) {
// calculates distance in kilometers between locations
  return acos(sin(p1->Lat*PI/180)*
                sin(p2->Lat*PI/180) +
                  cos(p1->Lat*PI/180)*cos(p2->Lat*PI/180)*
                  cos(p2->Lon*PI/180 - p1->Lon*PI/180)) * 6371;
}
```

Operation	Count in Source Code
Arc Cosine	1
Sine	2
Cosine	3
Floating-Point Multiply	10
Floating-Point Add	1
Floating-Point Subtract	1
Floating-Point Divide	6

Evaluation Environment

- MDK-ARM
- armcc v5
- Optimization
  - -O3
  - Speed

Examine Object Code (v1)

- Compile with armcc with maximum optimization, for speed
- Examine .txt file
- Observations
  - No conditional branching in function, just subroutine calls
  - Code makes 31 calls (BL), but only expected only 18 based on source code
  - Lots of double precision math routines called
    - __aeabi_dmul
    - __aeabi_ddiv
  - Assembly code listing is long (150 lines), tedious to examine

```
Calc_Distance PROC
        PUSH     {r4-r7,lr}
        SUB      sp,sp,#0x5c
        MOV      r4,r0
        MOV      r5,r1
        LDR      r0,[r4,#4]
        BL       __aeabi_f2d
        MOV      r6,r0
        LDR      r2,|L1.328|
        LDR      r3,|L1.332|
        BL       __aeabi_dmul
        MOVS     r2,#0
        LDR      r3,|L1.336|
        STR      r1,[sp,#4]
        STR      r0,[sp,#0]
        BL       __aeabi_ddiv
        STR      r1,[sp,#0x14]
        STR      r0,[sp,#0x10]
        LDR      r0,[r5,#4]
```

Examine Calls in Object Code

- Use search tool to extract function calls from source code for clarity
  - grep BL geometry.txt
- Observations
  - Code makes 31 calls (BL), but only expected only 18 based on source code
  - Lots of double precision (DP) math routines called (__aeabi_dmul, __aeabi_ddiv)
  - Type conversion routines called (__aeabi_f2d)
- What's happening?
  - *Library trig functions acos, sin, cos expect double precision arguments*
  - *C language performs automatic promotions of variables*

```
BL __aeabi_f2d
BL __aeabi_dmul
BL __aeabi_dmul
BL __aeabi_f2d
BL __aeabi_dmul
BL __aeabi_dmul
BL __aeabi_dsub
BL ||cos||
BL __aeabi_f2d
BL __aeabi_dmul
BL __aeabi_dmul
BL ||cos||
BL __aeabi_f2d
BL __aeabi_dmul
BL __aeabi_dmul
BL ||cos||
BL __aeabi_dmul
BL __aeabi_dmul
BL __aeabi_f2d
BL __aeabi_dmul
BL __aeabi_dmul
```

```
BL ||sin||
BL __aeabi_f2d
BL __aeabi_dmul
BL __aeabi_dmul
BL ||sin||
BL __aeabi_dmul
BL __aeabi_dadd
BL acos
BL __aeabi_d2f
BL __aeabi_fmul
```

Math.h Functions and Automatic Promotions

- Standard math routines usually accept double-precision inputs and return double-precision outputs.
- That double-precision return value will force all other operands in the expression to be promoted to doubles
  - Example: `return ((unsigned char) (3.5*(sin(x/f)+1.0)));`
    - x is unsigned int
    - f is float
    - 3.5 and 1.0 are loaded as doubles
  - The multiply and addition are promoted to double precision
  - We cast the result to an 8-bit integral value, discarding the fraction and most of the integer portion
  - A single precision float or even fixed-point sin() would be even faster
- It is likely that only single-precision is needed.
  - Cast to single-precision if accuracy and overflow conditions are satisfied

MDK Floating Point Math Library

- mathlib includes two versions of each function
  - double precision: sin()
  - single precision: sinf()

- Two methods to use
  - Write/change source code to call the single precision version
  - Pass argument **--fpmode=fast** to compiler to replace double precision with single precision,
    - Be sure to set optimization as high as possible

Default Type for Float Literals

```
#define PI 3.14159265
// Lat, Lon fields are single precision floats

float Calc_Distance( PT_T * p1,  const PT_T * p2) {
// calculates distance in kilometers between locations
  return acosf(sinf(p1->Lat*PI/180)*
               sinf(p2->Lat*PI/180) +
                 cosf(p1->Lat*PI/180)*cosf(p2->Lat*PI/180)*
                 cosf(p2->Lon*PI/180 - p1->Lon*PI/180)) * 6371;
}
```

Object Code

```
;;;19     float Calc_Distance( PT_T * p1,  const PT_T * p2) {
000000  b5f0       PUSH      {r4-r7,lr}
000002  b08f       SUB       sp,sp,#0x3c
000004  4604       MOV       r4,r0
000006  460d       MOV       r5,r1
;;;21       return acosf(sinf(p1->Lat*PI/180)*…
000008  6860       LDR       r0,[r4,#4]  ; Load p1->Lat (SP)
00000a  f7fffffe   BL        __aeabi_f2d ; Convert to DP
00000e  4606       MOV       r6,r0
000010  4a4b       LDR       r2,|L1.320| ; Load a constant
000012  4b4c       LDR       r3,|L1.324| ; Load another constant
000014  f7fffffe   BL        __aeabi_dmul; DP multiply
```

- p1->Lat is SP, but promoted to DP, returned in r0, r1
- A DP constant is loaded into r2 and r3
- DP multiply performed on argument 1 (in r0, r1) and argument 2 (in r2, r3)
- Why is PI represented as a double?
- *C standard states floating point literals are interpreted as double precision*

Floating Point Types and Literals

- Single-precision (SP) vs. double-precision (DP) vs. long double
  - ANSI C: single, double and long double sizes are implementation-dependent
  - ANSI/IEEE 754-1985, *Standard for Binary Floating Point Arithmetic*
    - Single precision: 32 bits
    - Double precision: 64 bits
    - Long double precision: 96 or 128 bits (architecture-dependent)
  - Single-precision is probably adequate

- Use the smallest adequate data type, or else…
  - Conversions without an FPU are very slow
  - Extra space is used
  - C standard allows compiler to convert automatically, slowing down code more

- Type suffixes for floating-point literals (ANSI C)
  - f/F type: single precision: 3.14f, 3.14F
  - l/L type: long double precision: 3.14l, 3.14L
  - **default: double precision: 3.14**

- What to do
  - Use single-precision floating point specifier "f" when possible
  - May be able to change compiler's default type for float literals to single-precision

Use Single-Precision Trig Functions

```
#define PI 3.14159265

float Calc_Distance( PT_T * p1,  const PT_T * p2) {
// calculates distance in kilometers between locations
  return acosf(sinf(p1->Lat*PI/180)*
             sinf(p2->Lat*PI/180) +
               cosf(p1->Lat*PI/180)*cosf(p2->Lat*PI/180)*
               cosf(p2->Lon*PI/180 - p1->Lon*PI/180)) * 6371;
}
```

Examine Calls in Object Code (v2)

- Good news
  - Code now is calling single-precision trig functions (e.g. cosf)
  - Code makes 24 calls now
- Bad news
  - Code is still making 6 more calls than expected

```
1.  BL __aeabi_fmul
2.  BL __aeabi_fmul
3.  BL __aeabi_fmul
4.  BL __aeabi_fmul
5.  BL __aeabi_fsub
6.  BL cosf
7.  BL __aeabi_fmul
8.  BL __aeabi_fmul
9.  BL cosf
10. BL __aeabi_fmul
11. BL __aeabi_fmul
12. BL cosf
13. BL __aeabi_fmul
14. BL __aeabi_fmul
15. BL __aeabi_fmul
16. BL __aeabi_fmul
17. BL sinf
18. BL __aeabi_fmul
19. BL __aeabi_fmul
20. BL sinf
21. BL __aeabi_fmul
22. BL __aeabi_fadd
23. BL acosf
24. BL __aeabi_fmul
```

Summary of Calls in v2 Code

Operation	Count in Source Code	Library Routine	Count in Object Code
Arc Cosine	1	acosf	1
Sine	2	sinf	2
Cosine	3	cosf	3
FP Multiply	10	__aeabi_fmul	16
FP Add	1	__aeabi_fadd	1
FP Subtract	1	__aeabi_fsub	1
FP Divide	6	__aeabi_fdiv	0

- We missing 6 fdiv calls
- We have 6 extra fmul calls
- Is this a coincidence? Let's examine the object code

Examine Object Code (v2)

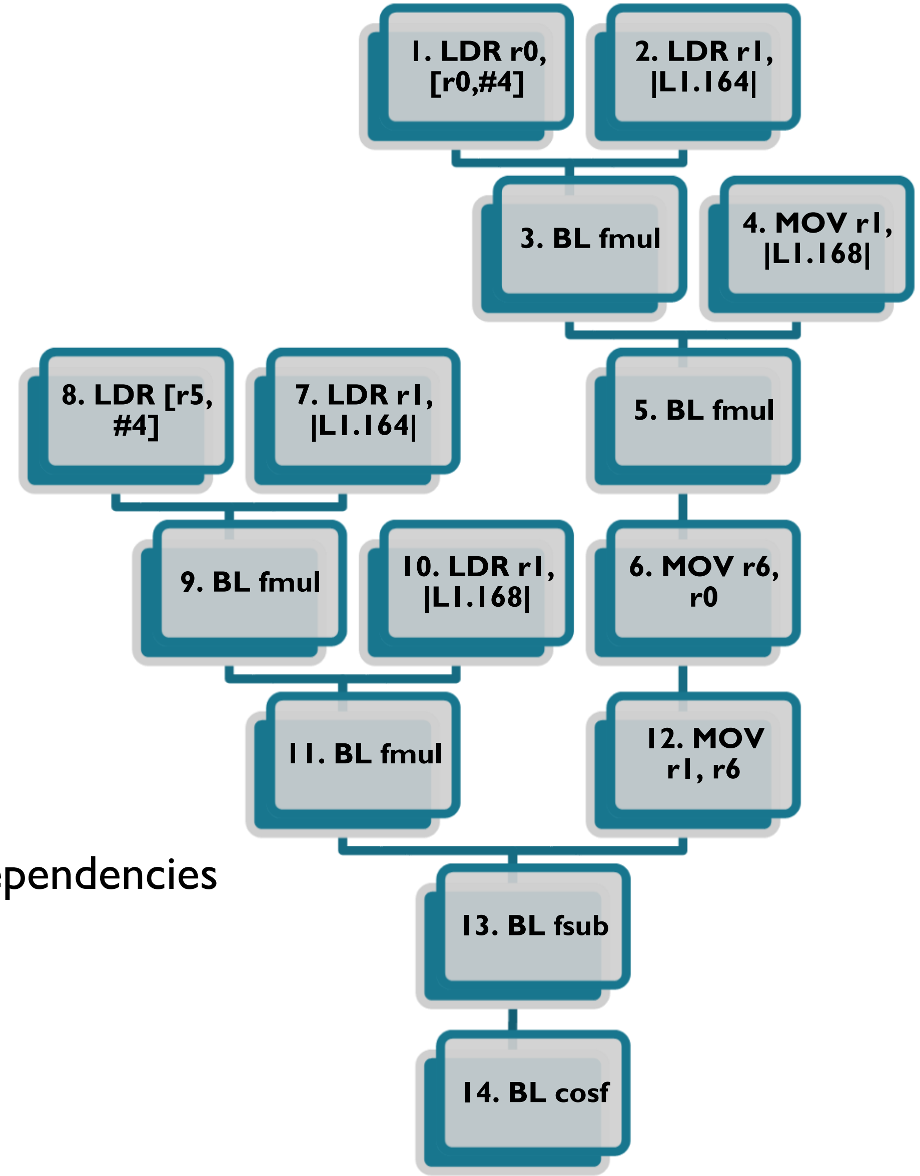
```
acosf(sinf(p1->Lat*PI/180)*sinf(p2->Lat*PI/180) +
         cosf(p1->Lat*PI/180)*cosf(p2->Lat*PI/180)*
         cosf(p2->Lon*PI/180 - p1->Lon*PI/180)) * 6371;
```

```
LDR     r0,[r0,#4]          LDR     r0,[r5,#4]
LDR     r1,|L1.164|         BL      __aeabi_fmul
BL      __aeabi_fmul        LDR     r1,|L1.168|
LDR     r1,|L1.168|         BL      __aeabi_fmul
BL      __aeabi_fmul        MOV     r1,r6
MOV     r6,r0               BL      __aeabi_fsub
LDR     r1,|L1.164|         BL      cosf
```

- Let's examine a portion of the code to understand what's happening
- Start with first call to cosf in object code and work backwards to see the data it uses (data flow analysis)
- Remember argument and return passing conventions
  - Arguments go in registers r0, r1, r2, r3, then stack
  - Return value comes back in register r0
- cosf has one argument, __aeabi_fmul has two

Data Flow Graph to BL cosf Instruction

```
1.  LDR    r0,[r0,#4]
2.  LDR    r1,|L1.164|
3.  BL     __aeabi_fmul
4.  LDR    r1,|L1.168|
5.  BL     __aeabi_fmul
6.  MOV    r6,r0
7.  LDR    r1,|L1.164|
8.  LDR    r0,[r5,#4]
9.  BL     __aeabi_fmul
10. LDR    r1,|L1.168|
11. BL     __aeabi_fmul
12. MOV    r1,r6
13. BL     __aeabi_fsub
14. BL     cosf
```



- How is the argument to BL cosf computed?
  - DFG shows how data flows among instructions, shows true data dependencies
  - -> Only multiplies are used to compute argument (no divides)
- Compiler's implementation eliminates division
  - Source code: `p2->Lon*PI/180`
  - Compiler's approach: `p2->Lon*PI*0.005555`

Optimization: Compile-Time Evaluation

```
return acosf(sinf(p1->Lat*PI/180)*sinf(p2->Lat*PI/180) +
                cosf(p1->Lat*PI/180)*cosf(p2->Lat*PI/180)*
                cosf(p2->Lon*PI/180 - p1->Lon*PI/180)) * 6371;

return acosf(sinf(p1->Lat*PI*0.005556)*sinf(p2->Lat*PI*0.005556) +
                cosf(p1->Lat*PI*0.005556)*cosf(p2->Lat*PI*0.005556)*
                cosf(p2->Lon*PI*0.005556 - p1->Lon*PI*0.005556)) *
                6371;
```

- PI/180 or PI*0.005556 can be evaluated at compile-time
  - Compiler should be able to optimize out this operation
- Why doesn't it? C operator precedence rules
  - * and / are same level of precedence, and are evaluated left to right
- Try putting parentheses around PI/180 term in source code

Examine Calls in Object Code (v3)

- Good news
  - No more calls to division functions
  - 10 calls to fmul
  - 18 calls total

- Should we expect the compiler to do better?

```
1. BL __aeabi_fmul
2. BL __aeabi_fmul
3. BL __aeabi_fsub
4. BL cosf
5. BL __aeabi_fmul
6. BL cosf
7. BL __aeabi_fmul
8. BL cosf
9. BL __aeabi_fmul
10.BL __aeabi_fmul
11.BL __aeabi_fmul
12.BL sinf
13.BL __aeabi_fmul
14.BL sinf
15.BL __aeabi_fmul
16.BL __aeabi_fadd
17.BL acosf
18.BL __aeabi_fmul
```

Optimization: Reuse of Data

```
#define PI 3.14159265f
float Calc_Distance( PT_T * p1,  const PT_T * p2) {
// calculates distance in kilometers between locations
  return acos(sin(p1->Lat*(PI/180))*
             sin(p2->Lat*(PI/180)) +
               cos(p1->Lat*(PI/180))*cos(p2->Lat*(PI/180))*
               cos(p2->Lon*(PI/180) - p1->Lon*(PI/180))) *
              6371;
}
```

- Repeated calculations
  - p1->Lat*(PI/180) is used twice
  - p2->Lat*(PI/180) is used twice
- Opportunity for optimization – "common subexpression elimination"

Does Compiler Do It?

- Are the results reused?
  - Need to examine the object code
  - Code has ten calls to __aebi_fmul, implying no reuse
- Why no reuse?
  - Does compiler assume that memory pointed to by p1, p2 may have changed between these calls?

- Try giving function a local copy of data to work with: p1Lat, p2Lat

```
float Calc_Distance( PT_T * p1,  const PT_T * p2) {
  float p1Lat = p1->Lat;
  float p2Lat = p2->Lat;
  return acosf(sinf(p1Lat*(PI/180))*sinf(p2Lat*(PI/180)) +
               cosf(p1Lat*(PI/180))*cosf(p2Lat*(PI/180))*
               cosf(p2->Lon*(PI/180) - p1->Lon*(PI/180))) * 6371;
}
```

Examine Calls in Object Code (v4)

- Good news
  - No more calls to division functions
  - 8 calls to fmul
  - 16 calls total

- Should we expect the compiler to do better?

```
1. BL __aeabi_fmul
2. BL __aeabi_fmul
3. BL __aeabi_fsub
4. BL cosf
5. BL __aeabi_fmul
6. BL cosf
7. BL __aeabi_fmul
8. BL cosf
9. BL __aeabi_fmul
10.BL __aeabi_fmul
11.BL sinf
12.BL sinf
13.BL __aeabi_fmul
14.BL __aeabi_fadd
15.BL acosf
16.BL __aeabi_fmul
```

DON’T HANDCUFF THE COMPILER

Approach

- Which optimizations is the compiler capable of?
- What might stop the compiler from applying them?
- How can we tell if the compiler applied them?
- How can we modify the source code to help the compiler optimize?

WHAT SHOULD THE COMPILER BE ABLE TO DO?

Scalar Optimizations

- What should you expect your compiler to be able to do?

- Machine-Independent (MI)
  - Eliminate code with no effect
  - Move operation to where it executes less often
  - Specialize computations
  - Eliminate redundant computations

- Machine-Dependent (MD)
  - Take advantage of special hardware features
  - Manage or hide latency
  - Manage limited machine resources

MI: Eliminate code with no effect

- Unreachable code
  - Examine CFG for nodes without predecessors
- Useless code (based on data-flow graph)
  - Mark *critical operations*
    - sets return value for procedure
    - input/output statement
    - modifies non-local data
  - Find operations which define data (operands) used by these critical operations
  - Repeat until set of critical operations doesn't grow
  - Delete remaining operations
- Useless control flow (based on CFG)
  - Fold redundant branches (both from BB A to BB B)
  - Remove empty BBs
  - Combine BBs (jump from A to B)
  - Branch hoisting – replace jump to empty block with a jump to its successor
- Simplification of algebraic identities
  - x*1 → x
  - x+0 → x

MI: Specialize computations

- Operator strength reduction
  - Replace multiplication with addition or shifting, division with subtraction or shifting
- Constant propagation
  - If a variable has a known value at given point in program, may be able to specialize operations based on this knowledge
  - e.g. perform calculations at compile time rather than run time
  - e.g. for (i=0; i<10; i++) is a top-test loop, but don't need test on first entry
- Peephole optimization
  - Recognize patterns of assembly instructions which can be replaced with a faster set
    - e.g. addressing modes

MI: Enabling Transformations

- Goal is to make code more amenable to *other* optimizations
- Loop unrolling
  - replicate loop body
  - adjust loop control code
  - *also reduces loop overhead, useful for short loop bodies*
- Loop unswitching
  - hoist loop-invariant control-flow operations out of loop
  - can be hard to determine if control-flow really is loop-invariant
- Renaming
  - value numbering gives independent name to each value defined
  - optimizer can recognize already-computed values which are still live

$a \leftarrow x + y$	$a_0 \leftarrow x_0 + y_0$
$b \leftarrow x + y$	$b_0 \leftarrow x_0 + y_0$
$a \leftarrow 17$	$a_1 \leftarrow 17$
$c \leftarrow x + y$	$c_0 \leftarrow x_0 + y_0$

Machine-Dependent Optimizations

- Take advantage of special instructions, addressing modes and hardware features
  - Pre-fetch, predicted branch, load bypassing cache
  - Conditional instruction execution
  - Advanced addressing modes
  - Pre-ALU barrel shifter
- Manage or hide latency (pipeline, memory system, ALU)
  - Large memory latency
  - Rearrange loop iteration order for cache locality
  - Change data layout to improve cache locality
- Manage limited machine resources
  - Register allocation

WHAT COULD STOP THE COMPILER FROM OPTIMIZING?

Excessive Variable Scope

- Avoid declaring variables as globals or statics when they could be locals (automatics or parameters)
- Globals and statics are allocated permanent storage in memory, not reusable stack space
- Compiler assumes that any function may access a global variable
  - Function must write back any globals it has modified **before calling a subroutine**
  - Function must reload any globals to use **after calling the routine**

Automatic Promotions in Arithmetic Expressions

- How are expressions with mixed data types evaluated?

```
float f;
char c;
int r;

r = f * c;
```

- ANSI C Standard for conversions
  - If either operand is a long double, promote the other to a long double
  - Else if either is a double, promote the other to a double
  - Else if either is a float, promote the other to a float
  - Else if either is a unsigned long int, promote the other to a unsigned long int
  - Else if either is a long int, promote the other to a long int
  - Else if either is an unsigned int, promote the other to an unsigned int
  - Else both are promoted to int: short, char, bit field
  - *Special rules for dealing with signed/unsigned differences left out*

Resulting Object Code


```
float f;
char c;
int r;

r = f * c;
```

- `Call routine to convert c to float`
- `Call routine to perform floating-point multiply with f`
- `Call routine to convert result from float to integer`
- `Store result in r`

- Time and code space overhead of conversion routines
- Avoid mixed type expressions

ANSI C Standard for Argument Promotions

- Integral function arguments smaller than an int for non-prototyped functions are promoted to ints
  - Extra time converting to int
  - Extra space on stack
- So prototype all functions
  - Function:

```
int Find_Average(char a, char b, char c,
  char d) {
. . .
}
```

  - Correct, complete prototype:

```
int Find_Average(char a, char b, char c,
  char d);
```

  - Parameter names are good documentation

- Where should the prototype go?
  - If program is broken into modules, put prototype in header (.h) file
  - Otherwise put prototype near top of C code file, before the function is called
- Another reason to prototype – some compilers won't promote arguments to use registers rather than stack if the function isn't prototyped

Precedence and Order of Evaluation

- How is a = b+c*d-e%f/g; evaluated?
- Order is based upon operator precedence and associativity
- Repeat
  - Evaluate the highest precedence operator
  - If multiple operators of the same precedence, apply associativity

Example: a = b + c * d – e % f / g;

1 c*d
2 e%f
3 /g
4 b+
5 –

Type	Operator	Associativity
Primary Expression	() [] . -> ++(post) - -(post)	left to right
Unary	* & + - ! ~ ++(pre) - -(pre) (typecast) sizeof()	right to left
Binary	* / %	left to right
	+ -	
	>> <<	
	< > <= >=	
	== !-	
	&	
	^	
	|	
	&&	
	||	
Ternary	?:	right to left
Assignment	= += -= *= /= %= >>= << = &= ^= |=	right to left
Comma	,	left to right

Functions

- Make local functions static, keep in same module (source file)
  - Allows more aggressive function inlining
  - Only functions in same module can be inlined
- Prototype your functions, or else the compiler may promote all arguments to ints or doubles

DATA REUSE

Data Reuse

```
float Calc_Bearing( PT_T * p1,  const PT_T * p2){
        float angle = atan2(
                sin(p1->Lon*(PI/180) - p2->Lon*(PI/180))*
       cos(p2->Lat*(PI/180)),
                cos(p1->Lat*(PI/180))*sin(p2->Lat*(PI/180)) -
                sin(p1->Lat*(PI/180))*cos(p2->Lat*(PI/180))*
       cos(p1->Lon*(PI/180) - p2->Lon*(PI/180))
                ) * (180/PI);
        if (angle < 0.0)
                angle += 360;
        return angle;
}
```

- Code may have *common sub-expressions* which perform *same operations* on *same input data*
  - Waste of computation
- Compiler should be able to recognize these, delete extra computations, reuse original result
- *Does it?*

Expression	Occurrences
p1->Lon*(PI/180)	2
p2->Lon*(PI/180)	2
p2->Lat*(PI/180)	3
p1->Lat*(PI/180)	2

PRECOMPUTATION OF RUN-TIME INVARIANT DATA

Run-Time Invariant Data

- Don’t waste the program’s time computing results which can never change at run-time
  - 2 + 2 will always be 4…
- Two approaches to eliminating these calculations from run-time
  - Rely on the compiler
    - Constant propagation: Optimization method which propagates constant values (known at compile time) to eliminate computation
    - May need to modify source code to help compiler optimize
  - Use custom tool before compiler
    - Generate final values automatically

Optimization Possible by Compiler

```
#define PI 3.14159265
float Calc_Distance( PT_T * p1,  const PT_T * p2) {
// calculates distance in kilometers between locations
  return acos(sin(p1->Lat*PI/180)*
              sin(p2->Lat*PI/180) +
               cos(p1->Lat*PI/180)*cos(p2->Lat*PI/180)*
               cos(p2->Lon*PI/180 - p1->Lon*PI/180)) * 6371;
}
```

- PI/180
  - Both operands are constants, known at compile time
  - Will always have same result
- Optimization
  - Compiler instead uses a multiplication by 0.017453293
  - Will delete six divisions per call to Calc_Distance

Optimization Requiring Source Code Modification

```
const PT_T waypoints[] = {
  //  Lat          Lon          Name
  {   56.07553,   152.57224,  "ALBATROSS BNK"   },
…
}
return acos(sin(p1->Lat*PI/180)*sin(p2->Lat*PI/180) +
          cos(p1->Lat*PI/180)*cos(p2->Lat*PI/180)*
          cos(p2->Lon*PI/180 - p1->Lon*PI/180)) * 6371;
```

- Why all the multiplication by PI/180?
  - Points coordinates are stored in degrees
  - Trig functions use radians for arguments or return values
- Store the point coordinates in radians instead of degrees
  - Modify source code so functions use radians
  - Helpful to use a spreadsheet or other tool to automate data conversion process

Optimization Requiring Source Code Modification

```
void Find_Nearest_Waypoint(float cur_pos_lat, float cur_pos_lon,
  float * distance, float * bearing, char  * * name) {
  …
  while (strcmp(waypoints[i].Name, "END")) {
    d = Calc_Distance(&ref, &(waypoints[i]) );
    b = Calc_Bearing(&ref, &(waypoints[i]) );
    …
  }
```

- Observations
  - Computing distance, bearing between one **arbitrary** point and one **known** point
  - Known point coordinates are constants stored in table
  - Don’t need to recompute sine or cosine of a constant, can apply constant propagation
- Optimization
  - Precompute the run-time invariant values based on the known point (2^{nd} argument)
  - Again, helpful to use a spreadsheet or other tool to automate data conversion process

Manual Constant Propagation (1)

```
float Calc_Distance( PT_T * p1,  const PT_T * p2) {
// calculates distance in kilometers between locations
  return acos(sin(p1->Lat*PI/180)*
              sin(p2->Lat*PI/180) +
              cos(p1->Lat*PI/180)*cos(p2->Lat*PI/180)*
              cos(p2->Lon*PI/180 - p1->Lon*PI/180)) * 6371;
}
```

- p2->Lat and p2->Lon are constants
- How far can we propagate those constants in the functions? See red code above
- What needs to be added to the point structure and table?
  - p2->Lon*PI/180
  - sin(p2->Lat*PI/180)
  - cos(p2->Lat*PI/180)

Manual Constant Propagation (2)

```
float Calc_Bearing( PT_T * p1,  const PT_T * p2){
  float angle = atan2(sin(p1->Lon*(PI/180) -
      p2->Lon*(PI/180))*cos(p2->Lat*(PI/180)),
      cos(p1->Lat*(PI/180))*sin(p2->Lat*(PI/180)) -
      sin(p1->Lat*(PI/180))*cos(p2->Lat*(PI/180))*
      cos(p1->Lon*(PI/180) - p2->Lon*(PI/180))
      ) * (180/PI);
```

- p2->Lat and p2->Lon are constants
- How far can we propagate those constants in the functions? See red code above
- What needs to be added to the point structure and table?
  - p2->Lon*PI/180
  - sin(p2->Lat*PI/180)
  - cos(p2->Lat*PI/180)
- Same as for previous function (Calc_Distance)

Improved Point Structure and Table

```
typedef struct {
  float Lat;
  float SinLat;
  float CosLat;
  float Lon;
  char Name[24];
} PT_T;
const PT_T waypoints[] = {
//  Lat        sin(Lat)      cos(Lat)      Lon        Name
  { 0.97860611, 0.829720137, 0.558179626, 2.66284884, "ALBATROSS BNK"},
  { 0.89273591, 0.778790853, 0.627283673, -3.12413936, "AMCHITKA" },
  { 1.03637651, 0.860564286, 0.50934184, 2.67646241, "AUGUSTINE IS, AK"},
  { 0.38432150, 0.374930219, 0.927053036, 1.64148216,"BAY CAMPECHE"     },
…
```

- Code is much faster
- Table is slightly larger: two more floats per point (40 instead of 32 bytes)

LESS COMPUTATION AT RUN-TIME

Example: Find the Nearest Point

```
void Find_Nearest_Point( . . . ) {
. . .
     while (strcmp(points[i].Name, "END")) {
          d = Calc_Distance (&ref, &(points[i]) );
               if (d>closest_d) {
                    closest_d = d;
                    closest_i = i;
               }
               i++;
          }
```

- Calc_Distance is called on every point, returning closest_d (distance in km)
- This distance is used in two ways
  - To identify closest point
  - Returned to calling function
- Can we split these up?

Distance Calculation

```
float Calc_Distance( PT_T * p1,  const PT_T * p2) {
  // calculates distance in kilometers between locations
  return acos(p1->SinLat * p2->SinLat + p1->CosLat * p2->CosLat
              *cos(p2->Lon - p1->Lon)) * 6371;
}
```

- Distance is acos(big_expression)*6371
- What is 6371?
  - Scaling factor to convert radians to km
  - 6371 km = Earth's radius = circumference/2π radians
- Can compare angle rather than km
  - Angle is still proportional to distance between points

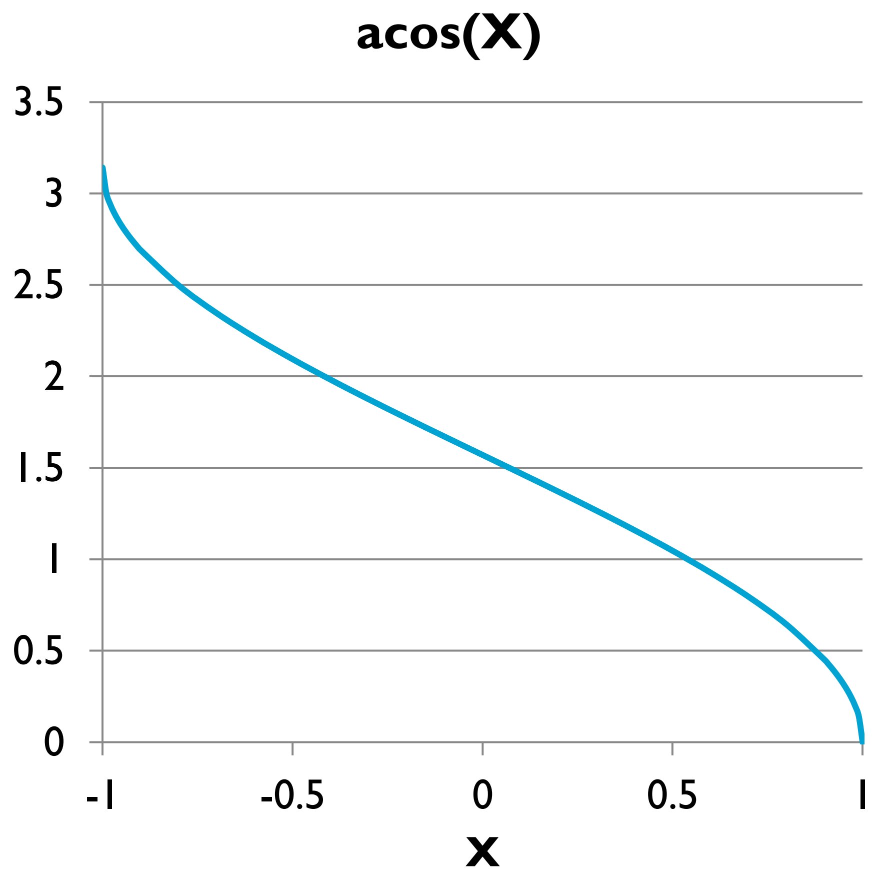
Optimized Code

```
float Calc_Distance_in_Radians( PT_T * p1,  const PT_T * p2) {
  // calculates distance in radians between locations
  return acos(p1->SinLat * p2->SinLat + p1->CosLat * p2->CosLat
              *cos(p2->Lon - p1->Lon)); // no *6371 here
}
void Find_Nearest_Point( . . . ) {
     while (strcmp(points[i].Name, "END")) {
          d = Calc_Distance_in_Radians(&ref, &(points[i]) );
          . . .
     }
     *distance = d*6371;
     . . .
}
```

- Eliminates N_{Points}-1 floating point multiplies

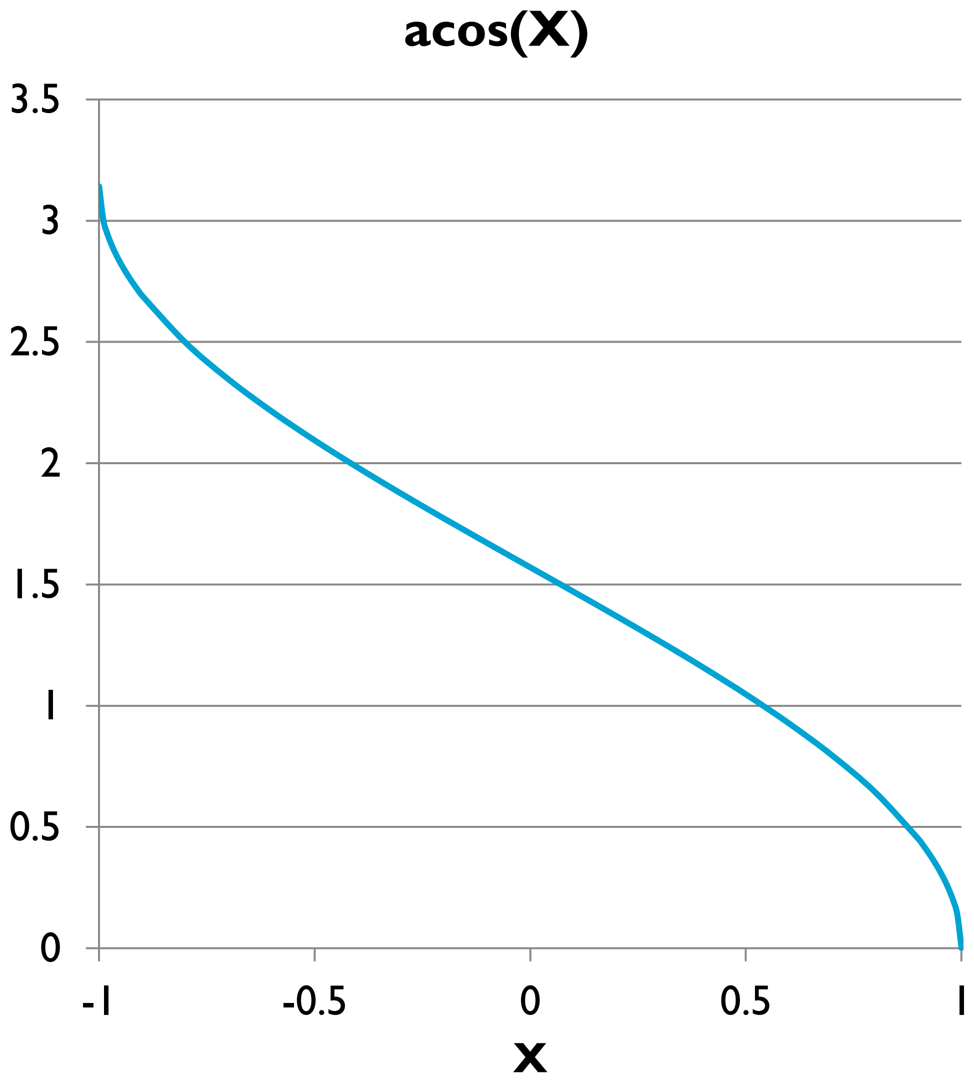
Taking it Further

```
float Calc_Distance_in_Radians( PT_T * p1,  const PT_T * p2) {
  // calculates distance in radians between locations
  return acos(p1->SinLat * p2->SinLat +
              p1->CosLat * p2->CosLat
               *cos(p2->Lon - p1->Lon));
}
```



- How is acos related to its argument X?
- Can we make distance comparisons *without* using acos?
- Just call acos once – to compute the distance to the closest point

Taking it Further




```
float Calc_Proximity ( PT_T * p1,
const PT_T * p2) {
        return (
          p1->SinLat * p2->SinLat +
          p1->CosLat * p2->CosLat
           *cos(p2->Lon - p1->Lon));
}
```

- **acos** always decreases as input X increases
- Nearest point will have **minimum** distance and **maximum** X
- So search for point with **maximum** argument to acos function
- After finding nearest point (max X), compute distance_km = acos(X) * 6371