PRAGMATIC OPTIMIZATION

IN MODERN PROGRAMMING

A BIT OF COMPILER MAGIC

Created by Marina Kolpakova, Itseez / 2016

OUTLINE

- Pragmatic approach
- Before we start...
- The magic box!
- How to learn optimization?

PRAGMATIC APPROACH

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of non-critical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small inefficiencies, say about 97% of the time; ***premature optimization is the root of all evil****. Yet we should not pass up our opportunities in that critical 3%.“*

-Donald Knuth, Structured Programming With go to Statements

1. **Find what to start from (3%)**
2. **Know when to stop (97%)**

BEFORE WE START...

GETTING FEEDBACK

- **Check wall-time of you application**
  - If a compiler does it right, you will see some uplift
- **Dump an assembly of your code** (or/and IL)
  - Ensure instruction and register scheduling
  - Check for extra operations and register spills
- **See compiler optimization report**
  - All the compilers have some support for it
  - Some of them are able to generate very detailed reports about loop unrolling, auto-vectorization, VLIW slots scheduling, etc

CONSIDERED METRICS

Wall(-clock) time

is a human perception of the span of time from the start to the completion of a task.

Power consumption

is the electrical energy which is consumed to complete a task.

Processor time (or runtime)

is the total execution time during which a processor was dedicated to a task (i.e. executes instructions of that task).

ASSEMBLY

Assembler is a must-have to check the compiler
but it is rarely used to write low-level code.

```
$ gcc code.c -S -o asm.s
```

- Assembly writing is the least portable optimization
- Inline assembly limits compiler optimizations
- Assembly does not give overwhelming speedup nowadays
- Sometimes it is needed to overcome compiler bugs and optimization limitations

INTERMEDIATE LANGUAGE

Intermediate representation (IR) is a source code representation in some abstract Instruction Set Architecture (ISA), which is close to a classic RISC

High Level IR

is close to the source and can be easily generated from the source code. Some code optimizations are possible. It is not very suitable for target machine optimization.

Low Level IR

is close to the target machine and used for machine-dependent optimizations: register allocation, instruction selection, peephole optimization.

GETTING IR

GENERIC and GIMPLE

GNU Compiler Collection

```
-fdump-tree-all -fdump-tree-optimized -fdump-tree-ssa
-fdump-rtl-all
```

LLVM IL

clang and other LLWM based compilers

```
-emit-llvm
```

CIL (C Intermediate Language)

Visual Studio cl.exe

GETTING IR

```
$ clang -Os -S -emit-llvm test.c -o test.ll
$ cat test.ll
```

```
for( ; j <= width - 4; j += 4 )
{
    uchar t0 = tab[src[j]];
    uchar t1 = tab[src[j+1]];
    dst[j] = t0;
    dst[j+1] = t1;
    t0 = tab[src[j+2]];
    t1 = tab[src[j+3]];
    dst[j+2] = t0;
    dst[j+3] = t1;
}
```

GETTING IR

```
.lr.ph4:                              ; preds = %0, %.lr.ph4
  %indvars.iv5 = phi i64 [ %indvars.iv.next6, %.lr.ph4 ], [ 0, %0 ]
  %6 = getelementptr inbounds i8* %src, i64 %indvars.iv5
  %7 = load i8* %6, align 1, !tbaa !1
  %8 = zext i8 %7 to i64
  %9 = getelementptr inbounds i8* %tab, i64 %8
  %10 = load i8* %9, align 1, !tbaa !1
  %11 = or i64 %indvars.iv5, 1
  %12 = getelementptr inbounds i8* %src, i64 %11
  // even more code here
  %30 = load i8* %29, align 1, !tbaa !1
  %31 = getelementptr inbounds i8* %dst, i64 %19
  store i8 %24, i8* %31, align 1, !tbaa !1
  %32 = getelementptr inbounds i8* %dst, i64 %25
  store i8 %30, i8* %32, align 1, !tbaa !1
  %indvars.iv.next6 = add nuw nsw i64 %indvars.iv5, 4
  %33 = trunc i64 %indvars.iv.next6 to i32
  %34 = icmp sgt i32 %33, %1
  br i1 %34, label %..preheader_crit_edge, label %.lr.ph4
```

Compiler don't care how many variables are used in code, register allocation is done after IR rotations.

GCC FEEDBACK OPTIONS

- Enables optimization information printing

```
-fopt-info
-fopt-info-<optimized/missed/note/all>
-fopt-info-all-
<ipa/loop/inline/vec/optall>
-fopt-info=filename
```

- Controls the amount of debugging output the scheduler prints on targets that use instruction scheduling

```
-fopt-info -fsched-verbose=n
```

- Controls the amount of output from auto-vectorizer

```
-ftree-vectorizer-verbose=n
```

GCC FEEDBACK OPTIONS

- Outputs all optimization info to stderr.

```
gcc -O3 -fopt-info
```

- Outputs missed optimization report from all the passes to missed.txt

```
gcc -O3 -fopt-info-missed=missed.txt
```

- Outputs information about missed optimizations as well as optimized locations from all the inlining passes to inline.txt.

```
gcc -O3 -fopt-info-inline-optimized-missed=inline.txt
```

GCC FEEDBACK EXAMPLE

```
1.cc:193:9: note: loop vectorized
1.cc:193:9: note: loop versioned for vectorization because of possib
le aliasing
1.cc:193:9: note: loop peeled for vectorization to enhance alignment
1.cc:96:9: note: loop vectorized
1.cc:96:9: note: loop peeled for vectorization to enhance alignment
1.cc:51:9: note: loop vectorized
1.cc:51:9: note: loop peeled for vectorization to enhance alignment
1.cc:193:9: note: loop with 7 iterations completely unrolled
1.cc:32:13: note: loop with 7 iterations completely unrolled
1.cc:96:9: note: loop with 15 iterations completely unrolled
1.cc:51:9: note: loop with 15 iterations completely unrolled
1.cc:584:9: note: loop vectorized
1.cc:584:9: note: loop versioned for vectorization because of possib
le aliasing
1.cc:584:9: note: loop peeled for vectorization to enhance alignment
1.cc:482:9: note: loop vectorized
1.cc:482:9: note: loop peeled for vectorization to enhance alignment
1.cc:463:5: note: loop vectorized
1.cc:463:5: note: loop versioned for vectorization because of possib
le aliasing
1.cc:463:5: note: loop peeled for vectorization to enhance alignment
```

THE MAGIC BOX!

FILLING/COPYING MEMORY BLOCKS

Compiler automatically uses the library functions `memset` and `memcpy` to initialize and copy memory blocks

```
static char a[100000];
static char b[100000];

static int at(int idx, char val)
{
 if (idx>=0 && idx<100000)
  a[idx] = val;
}

int main()
{
 int i;
 for (i=0; i<100000; ++i) a[i]=42;
 for (i=0; i<100000; ++i) at(i,-1);
 for (i=0; i<100000; ++i) b[i] = a[i];
}
```

FILLING/COPYING MEMORY BLOCKS

Compiler knows what you mean

```
main:
.LFB1:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_def_cfa_offset 16
  movl  $100000, %edx
  movl  $42, %esi
  movl  $a, %edi
  call  memset
  movl  $100000, %edx
  movl  $255, %esi
  movl  $a, %edi
  call  memset
  movl  $100000, %edx
  movl  $a, %esi
  movl  $b, %edi
  call  memcpy
  addq  $8, %rsp
  .cfi_def_cfa_offset 8
  ret
```

FILLING/COPYING MEMORY BLOCKS

The same picture, if the code is compiled for ARM target

```
main:
  ldr r3, .L3
  mov r1, #42
  stmfd sp!, {r4, lr}
  add r3, pc, r3
  movw  r4, #34464
  movt  r4, 1
  mov r0, r3
  mov r2, r4
  bl   memset(PLT)
  mov r2, r4
  mov r1, #255
  bl   memset(PLT)
  mov r2, r4
  mov r3, r0
  ldr r0, .L3+4
  mov r1, r3
  add r0, pc, r0
  add r0, r0, #1792
  bl   memcpy(PLT)
  ldmfd sp!, {r4, pc}
```

FUNCTION BODY INLINING

Replaces functional call to function body itself

Enables all further optimizations!

```
int square(int x)
{
  return x*x;
}

for (int i = 0; i < len; i++)
  arr[i] = square(i);
```

Becomes

```
for (int i = 0; i < len; i++)
  arr[i] = i*i;
```

```
.L2:
  add x2, x4, :lo12:.LANCHOR0
  mov x1, 34464
  mul w3, w0, w0
  movk  x1, 0x1, lsl 16
  str w3, [x2,x0,lsl 2]
  add x0, x0, 1
  cmp x0, x1
  bne .L2
```

```
gcc -march=armv8-a+nosimd -fstrict-aliasing -O3 -fopt-info 1.c -S -o 1.s
```

AUTO-VECTORIZATION

- Machine code generation that takes an advantage of vector instructions.
- Most of all modern architectures have vector extensions as a co-processor or as dedicated pipes
  - MMX, SSE, SSE2, SSE4, AVX, AVX-512
  - AltiVec, VSX
  - ASIMD (NEON), MSA
- Enabled by inlining, unrolling, fusion, software pipelining, inter-procedural optimization, and other machine independent transformations.

FUNCTION BODY INLINING

Let's compile the previous example with vector extension enabled **`-march=armv8-a+simd`**

```
int square(int x)
{
  return x*x;
}

for (int i = 0; i < len; i++)
  arr[i] = square(i);
```

Becomes

```
for (int i = 0; i < len; i++)
  arr[i] = i*i;
```

```
 add x0, x0, :lo12:.LANCHOR0
 movi  v2.4s, 0x4
 ldr q0, [x1]
 add x1, x0, 397312
 add x1, x1, 2688
.L2:
 mul v1.4s, v0.4s, v0.4s
 add v0.4s, v0.4s, v2.4s
 str q1, [x0],16
 cmp x0, x1
 bne .L2
```

It is vectorized because of possibility to inline function call

AUTO-VECTORIZATION

```
void vectorizeMe(float *a, float *b, int len)
{
  int i;
  for (i = 0; i < len; i++)
    a[i] += b[i];
}
```

```
$ gcc -march=armv7-a -mfpu=neon-vfpv4 -mfloat-abi=softfp -mthumb -O3 -fopt-info-missed 1.c -S -o 1.s
```

AUTO-VECTORIZATION

NEON does not support full IEEE 754, so gcc cannot vectorize the loop, what it told us

```
.L3:
  fldmias r1!, {s14}
  flds  s15, [r0]
  fadds s15, s15, s14
  fstmias r0!, {s15}
  cmp r0, r2
  bne .L3
```

```
1.c:64:3: note: not vectorized:
relevant stmt not supported:_13 =_9+_12;
```

AUTO-VECTORIZATION

But armv8-a does support, let's check it!

```
gcc -march=armv7-a -mfpu=neon-vfpv4 -mfloat-abi=softfp -mthumb -O3 -fopt-info-missed 1.c -S -o 1.s
```

```
.L6:
  ldr q1, [x3],16
  add w6, w6, 1
  ldr q0, [x7],16
  cmp w6, w4
  fadd  v0.4s, v0.4s, v1.4s
  str q0, [x8],16
  bcc .L6
```

```
1.c:64:3: note: loop vectorized
```

FULL OPTIMIZER'S REPORT

```
1.c:66:3: note: loop vectorized
1.c:66:3: note: loop versioned for vectorization because of possible
 aliasing
1.c:66:3: note: loop peeled for vectorization to enhance alignment
1.c:66:3: note: loop with 3 iterations completely unrolled
1.c:61:6: note: loop with 3 iterations completely unrolled
```

Compiler versions the loop to allow optimized paths in case of aligned and non-aliasing pointers

KEYWORDS

Lit's follow the advice to put some keywords

```
void vectorizeMe(float* __restrict a_,
    float* __restrict b_, int len)
{
  int i;
  float *a = __builtin_assume_aligned(a_, 16);
  float *b = __builtin_assume_aligned(b_, 16);
  for (i = 0; i < len; i++)
    a[i] += b[i];
}
```

```
1.c:66:3: note: loop vectorized
1.c:66:3: note: loop with 3 iterations
completely unrolled
```

`__restrict` and `__builtin_assume_aligned` keywords only eliminate some loop versioning, but are not very useful from the performance perspective nowadays

SCALARIZATION

***Scalarization** replaces branchy code with a branchless analogy, usually to allow auto-vectorization of a loop body*

```c
int branchy(int i)
{
  if (i > 4 && i < 42)
    return 1;
  return 0;
}
int branchless(int i)
{
  return (((unsigned)i) - 5 > 36);
}
```

```
branchy:
  sub w0, w0, #5
  cmp w0, 36
  cset  w0, ls
  ret
branchless:
  sub w0, w0, #5
  cmp w0, 36
  cset  w0, hi
  ret
```

```
gcc -march=armv8-a+simd -O3 1.c -S -o 1.s
```

Both snippets are compiled to the same instructions!

UNSWITCHING

***Unswitching** moves loop-invariant conditions out of its body*

```
gcc -march=armv8-a+simd -O3 1.c -S -o 1.s
```

```
for (int i = 0; i < len; i++) {
  if (a > 32)
    arr[i] = a;
  else
    arr[i] = 42;
}
```

Becomes

```
if (a > 32) {
  for (int i = 0; i < len; i++)
    arr[i] = a;
} else {
  for (int i = 0; i < len; i++)
    arr[i] = 42;
}
```

```
  cmp w2, 32
  bgt .L6
  mov x2, 0
  mov w3, 42
.L4:
  str w3, [x0,x2,lsl 2]
  add x2, x2, 1
  cmp w1, w2
  bgt .L4
.L1:
  ret
.L6:
  mov x3, 0
.L3:
  str w2, [x0,x3,lsl 2]
  add x3, x3, 1
  cmp w1, w3
  bgt .L3
```

LOOP-INDUCTION VARIABLES

Replacing address arithmetics with pointer arithmetics

```
gcc -march=armv7-a -mfpu=neon -mfloat-abi=softfp -O1 1.c -S -o 1.s
```

```c
void function(int* arr, int len)
{
  for (int i = 0; i < len; i++)
    arr[i] = 1;
}
```

```
  mov r3, r0
  add r0, r0, r1, lsl #2
  movs  r2, #1
.L3:
  str r2, [r3], #4
  cmp r3, r0
  bne .L3
```

```c
void function(int* arr, int len)
{
  for (int* p = arr; p < arr + len; p++)
      *p = 1;
}
```

```

  add r1, r0, r1, lsl #2
  movs  r3, #1
.L8:
  str r3, [r0], #4
  cmp r1, r0
  bhi .L8
```

Most hand-written pointer optimizations do not make sense with usage optimization levels higher than O0.

STRENGTH REDUCTION

Replaces complex expressions with a simpler analogy

```c
double usePow(double x)
{
  return pow(x, 2.0);
}
```

```
usePow:
  fmdrr d16, r0, r1
  fmuld d16, d16, d16
  fmrrd r0, r1, d16
  bx  lr
```

```c
float usePowf(float x)
{
  return powf(x, 2.f);
}
```

```
usePowf:
  fmsr  s15, r0
  fmuls s15, s15, s15
  fmrs  r0, s15
  bx  lr
```

```
gcc -march=armv7-a -mfpu=vfpv4 -mfloat-abi=softfp -mthumb -O3 1.c -S -o 1.s
```

STRENGTH REDUCTION (ADVANCED)

Let's look at more complex expression.

```c
float useManyPowf(
  float a, float b,
  float c, float d,
  float e, float f,
  float x)
{
 return
  a * powf(x,5.f)+
  b * powf(x,4.f)+
  c * powf(x,3.f)+
  d * powf(x,2.f)+
  e * x         +
  f;
}
```

```asm
useManyPowf:
 push  {r3, lr}
 flds  s17, [sp, #56]
 fmsr  s24, r1
 movs  r1, #0
 fmsr  s22, r0
 movt  r1, 16544
 fmrs  r0, s17
 fmsr  s21, r2
 fmsr  s20, r3
 flds  s19, [sp, #48]
 flds  s18, [sp, #52]
 bl   powf(PLT)
 mov r1, #1082130432
 fmsr  s23, r0
 fmrs  r0, s17
 bl   powf(PLT)
```

```asm
 movs  r1, #0
 movt  r1, 16448
 fmsr  s16, r0
 fmrs  r0, s17
 bl   powf(PLT)
 fmuls s16, s16, s24
 vfma.f32  s16, s23, s22
 fmsr  s15, r0
 vfma.f32  s16, s15, s21
 fmuls s15, s17, s17
 vfma.f32  s16, s20, s15
 vfma.f32  s16, s19, s17
 fadds s15, s16, s18
 fldmfdd sp!, {d8-d12}
 fmrs  r0, s15
 pop {r3, pc}
```

GCC was able partly reduce the complexity using `vfma`

HORNER'S RULE

```
float applyHornerf(float a, float b, float c,
                   float d, float e, float f, float x)
{
  return ((((a * x + b) * x + c) * x + d) * x + e) * x + f;
}
```

```
gcc -march=armv7-a -mfpu=vfpv4 -mfloat-abi=softfp -mthumb -O3 1.c -S -o 1.s
```

```
applyHornerf:
  flds   s15, [sp, #8]
  fmsr   s11, r0
  fmsr   s12, r1
  flds   s14, [sp]
  vfma.f32  s12, s11, s15
  fmsr   s11, r2
  flds   s13, [sp, #4]
  vfma.f32  s11, s12, s15
  fcpys s12, s11
  fmsr   s11, r3
  vfma.f32  s11, s12, s15
  vfma.f32  s14, s11, s15
  vfma.f32  s13, s14, s15
  fmrs   r0, s13
  bx   lr
```

CASE STUDY: FLOATING POINT

```
double power( double d, unsigned n)
{
  double x = 1.0;
  for (unsigned j = 1; j<=n; j++, x *= d) ;
  return x;
}
int main ()
{
  double a = 1./0x80000000U, sum = 0.0;
  for (unsigned i=1; i<= 0x80000000U; i++)
    sum += power( i*a, i % 8);
  printf ("sum = %g\n", sum);
}
```

flags-dp.c

OPTIMIZATION LEVELS

1. Compile it **without** optimization

```
$ gcc -std=c99 -Wall -O0 flags-dp.c -o flags-dp
$ time ./flags-dp
sum = 7.29569e+08
real   0m24.550s
```

2. Compile it with **O1**: **~3.26 speedup**

```
$ gcc -std=c99 -Wall -O1 flags-dp.c -o flags-dp
$ time ./flags-dp
sum = 7.29569e+08
real   0m7.529s
```

OPTIMIZATION LEVELS

3. Compile it with **O2: ~1.24 speedup**

```
$ gcc -std=c99 -Wall -O2 flags-dp.c -o flags-dp
$ time ./flags-dp
sum = 7.29569e+08
real   0m6.069s
```

4. Compile it with **O3: ~1.00 speedup**

```
$ gcc -std=c99 -Wall -O3 flags-dp.c -o flags-dp
$ time ./flags-dp
sum = 7.29569e+08
real   0m6.067s
```

Total speedup is ~4.05

CASE-STUDY: INTEGER

```c
int power( int d, unsigned n)
{
  int x = 1;
  for (unsigned j = 1; j<=n; j++, x*=d) ;
  return x;
}
int main ()
{
  int64_t sum = 0;
  for (unsigned i=1; i<0x80000000U; i++)
    sum += power( i, i % 8);
  printf ("sum = %ld\n", sum);
}
```

flags.c

OPTIMIZATION LEVELS

1. Compile it **without** optimization

```
$ gcc -std=c99 -Wall -O0 flags.c -o flags
$ time ./flags
sum = 2882318614050897 91
real   0m18.750s
```

2. Compile it with **O1**: **~2.64 speedup**

```
$ gcc -std=c99 -Wall -O1 flag.c -o flags
$ time ./flags
sum = 2882318614050897 91
real   0m7.092s
```

OPTIMIZATION LEVELS

3. Compile it with **O2:~0.97 speedup**

```
$ gcc -std=c99 -Wall -O2 flags.c -o flags
$ time ./flags
sum = 288231861405089791
real   0m7.300s
```

4. Compile it with **O3: ~1.00 speedup**

```
$ gcc -std=c99 -Wall -O3 flags.c -o flags
$ time ./flags
sum = 288231861405089791
real   0m7.082s
```

WHY THERE IS NO IMPROVEMENT?

ASSEMBLY

Optimization level: -O1

```
  movl   $1, %r8d
  movl   $0, %edx
.L9:
  movl   %r8d, %edi
  movl   %r8d, %esi
  andl   $7, %esi
  je   .L10
  movl   $1, %ecx
  movl   $1, %eax
.L8:                    @
  addl   $1, %eax       @
  imull  %edi, %ecx     @
  cmpl   %eax, %esi     @
  jae  .L8
  jmp  .L7
.L10:
  movl   $1, %ecx
.L7:
  movslq   %ecx, %rcx
  addq   %rcx, %rdx
  addl   $1, %r8d
  jns  .L9
  @ printing is here
```

Optimization level: -O2

```
  movl     $1, %esi
    movl     $1, %r8d
    xorl     %edx, %edx
    .p2align 4,,10
    .p2align 3
.L10:
    movl     %r8d, %edi
    andl     $7, %edi
    je   .L11
    movl     $1, %ecx
    movl     $1, %eax
    .p2align 4,,10
    .p2align 3
.L9:
    addl     $1, %eax
    imull    %esi, %ecx
    cmpl     %eax, %edi
    jae .L9
    movslq   %ecx, %rcx
```

```
.L8:
    addl     $1, %r8d
    addq     %rcx, %rdx
    testl    %r8d, %r8d
    movl     %r8d, %esi
    jns  .L10
    subq     $8, %rsp
    @ printing is here
    ret
.L11:
    movl     $1, %ecx
    jmp .L8
```

Compiler overdone with branch twiddling and alignment

HELPING A COMPILER

Compiler usually applies optimization to the inner loops. In this example number of iterations in the inner loop depends on a value of an induction variable of the outer loop.

Let's help the compiler (flags-dp-tuned.c)

HELPING A COMPILER

```
/*power function is not changed*/
int main ()
{
  double a = 1./0x80000000U, s = -1.;
  for (double i=0; i<=0x80000000U-8; i += 8)
  {
    s+=power((i+0)*a,0);s+=power((i+1)*a,1);
    s+=power((i+2)*a,2);s+=power((i+3)*a,3);
    s+=power((i+4)*a,4);s+=power((i+5)*a,5);
    s+=power((i+6)*a,6);s+=power((i+7)*a,7);
  }
  printf ("sum = %g\n", s);
}
```

```
$ gcc -std=c99 -Wall -O3 flags-dp-tuned.c -o flags-dp-tuned
$ time ./flags-dp-tuned
sum = 7.29569e+08
real   0m2.448s
```

Speedup is ~2.48x.

HELPING A COMPILER

Let's try it for integers (flags-tuned.c)

```c
/*power function is not changed*/
int main ()
{
  int64_t sum = -1;
  for (unsigned i=0; i<=0x80000000U-8; i += 8)
  {
    sum+=power(i+0, 0); sum+=power(i+1,1);
    sum+=power(i+2, 2); sum+=power(i+3,3);
    sum+=power(i+4, 4); sum+=power(i+5,5);
    sum+=power(i+6, 6); sum+=power(i+7,7);
  }
  printf ("sum = %ld\n", sum);
}
```

```
$ gcc -std=c99 -Wall -O3 flags-tuned.c -o flags-tuned
$ time ./flags-tuned
sum = 2882318614050897 91
real   0m1.286s
```

Speedup is ~5.5x.

HOW TO LEARN OPTIMIZATION?

HOW TO LEARN OPTIMIZATION?

*Optimization is a **craft** rather than a **science**.*

Practice more

Do not make practical knowledge too theoretical

Look, what other people do

Find use-cases of different approaches and techniques

Dig into an architecture

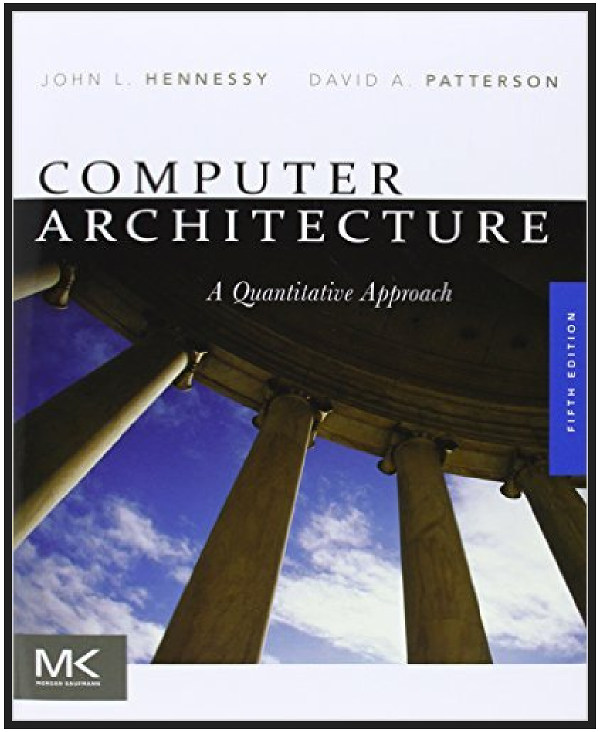
Hardware evolves rapidly, hence today's devices obsolete in a wink. **Comprehensive knowledge helps to see beforehand**

KNOWLEDGE WHICH IS REQUIRED

1. **The code**
   - The problem, it solves
   - The algorithm, it implements
   - The algorithmic complexity
2. **The compiler**
   - Compilation trajectory
   - Compiler's capabilities and obstacles
3. **The platform**
   - Architecture capabilities

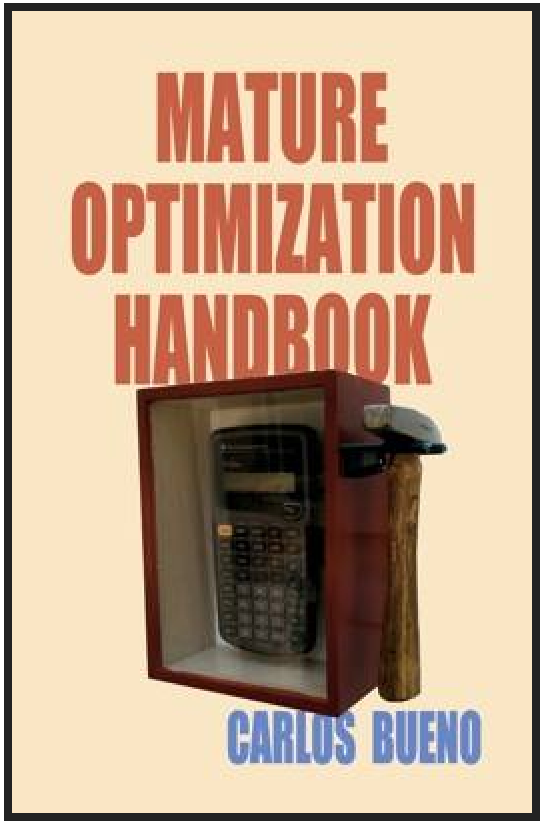
     $\mathrm{I}_{\mathrm{nstruction}}\ \mathrm{S}_{\mathrm{et}}\ \mathrm{A}_{\mathrm{rchitecture}}$
   - Micro-architecture specifics

RECOMMENDED LITERATURE



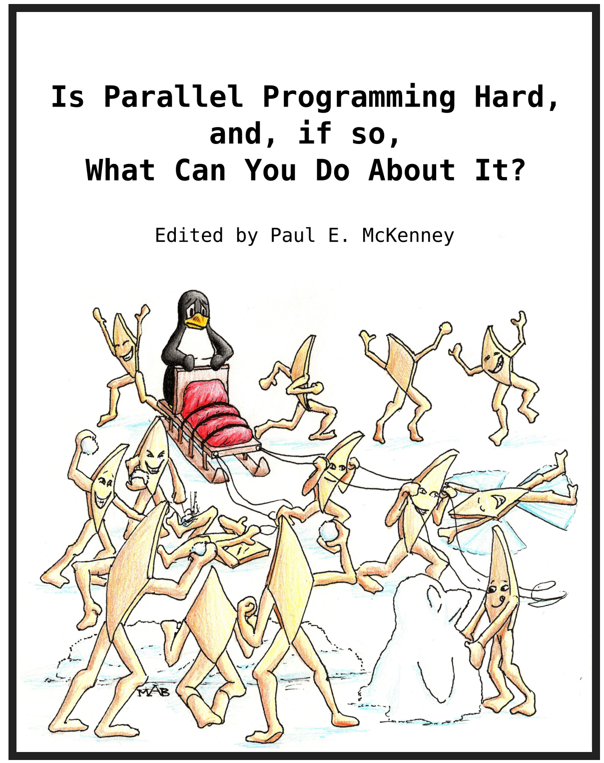
Computer Architecture, Fifth Edition:
A Quantitative Approach
by John L. Hennessy and David A. Patterson.

RECOMMENDED LITERATURE



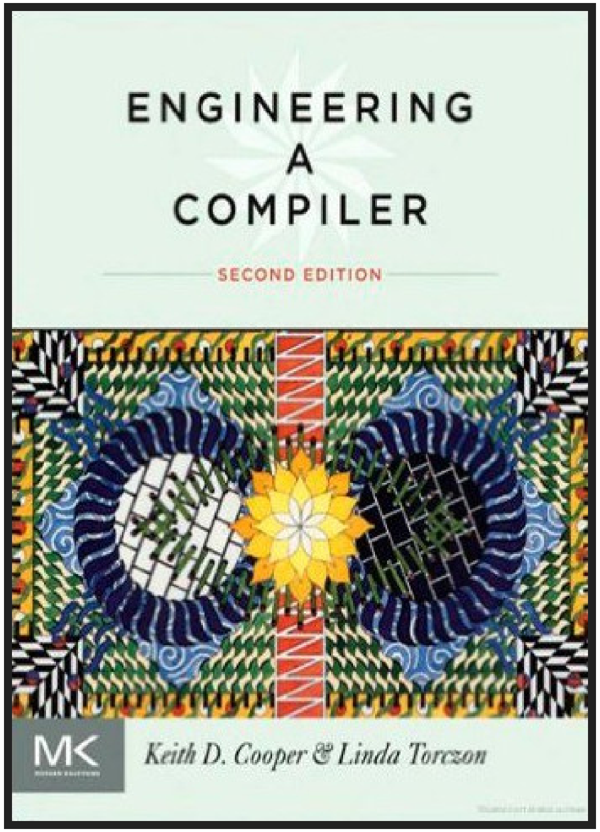
The Mature Optimization Handbook
by Carlos Bueno

RECOMMENDED LITERATURE



Is Parallel Programming Hard, And, If So, What Can You Do About It?
by Paul E. McKenney

RECOMMENDED LITERATURE



Engineering a Compiler
by Keith Cooper and Linda Torczon

SUMMARY

- Compiler detects `memcpy`/`memset`, even if they are implement manually, and call library function instead. Library functions are usually more efficient.
- `__restrict` and `__builtin_assume_aligned` keywords only eliminate some loop versioning.
- For typical constructs a compiler usually does better job.
- Most hand-written pointer optimizations do not make sense with usage of optimization levels higher than O0.
- Function body inlining and loop unswitching enables all other optimizations.
- Compiler optimization is a multi-phase iterative process

SUMMARY

- Knowledge about the code, the compiler and the platform is a must-have.
- The main task of an optimizer is finding the bottleneck.
- Optimizer's mastership is to know where to stop.
- Stick to the high-to-low approach.
- Practice, look what others do, dig into an architecture.
- Express your intentions to the compiler clearly.
- Learn a compiler.
- A compiler is not aware of your program semantics.
- **Write pure code!**

THE END

MARINA KOLPAKOVA / 2016