Optimizing Program Performance

CMPU 224 – Computer Organization
Jason Waterman
Performance Realities

• There’s more to performance than asymptotic complexity (big-O notation)

• Constant factors matter too!
  • Easily see 10x performance range depending on how code is written
  • Must optimize at multiple levels:
    • algorithm, data representations, procedures, and loops

• Must understand system to optimize performance
  • How programs are compiled and executed
  • How modern processors + memory systems operate
  • How to measure program performance and identify bottlenecks
  • How to improve performance without destroying code modularity and generality
Optimizing Compilers

• Provide efficient mapping of program to machine
  • register allocation
  • code selection and ordering (scheduling)
  • dead code elimination
  • eliminating minor inefficiencies

• Have difficulty overcoming “optimization blockers”
  • potential memory aliasing
  • potential procedure side-effects

• Don’t (usually) improve asymptotic efficiency
  • up to programmer to select best overall algorithm
  • big-O savings are (often) more important than constant factors
    • but constant factors also matter
Limitations of Optimizing Compilers

• Operate under fundamental constraint
  • Must not cause any change in program behavior

• Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  • e.g., Data ranges may be more limited than variable types suggest

• Most analysis is performed only within procedures
  • Whole-program analysis is too expensive in most cases
  • Newer versions of GCC do inter-procedural analysis within individual files
    • But, not between code in different files

• Most analysis is based only on static information
  • Compiler has difficulty anticipating run-time inputs

• When in doubt, the compiler must be conservative
Generally Useful Optimizations

• Optimizations that you or the compiler should do regardless of processor / compiler

• Code Motion
  • Reduce frequency with which computation performed
    • If it will always produce same result
    • Especially moving code out of loop

```c
void set_row(double *a, double *b, long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```
Reduction in Strength

• Replace costly operation with simpler one
• Shift, add instead of multiply or divide
  • $16 \times x \rightarrow x \ll 4$
  • Utility machine dependent
  • Depends on cost of multiply or divide instruction
    • On Intel Nehalem, integer multiply requires 3 CPU cycles
• Recognize sequence of products

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

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

- Reuse portions of expressions
- GCC will do this with `-O1`

```c
/* Sum neighbors of i,j */
up = val[(i-1)*n + j ];
down = val[(i+1)*n + j ];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;
```

```c
long inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

### 3 multiplications: i*n, (i-1)*n, (i+1)*n

```assembly
leaq   1(%rsi), %rax  # i+1
leaq   -1(%rsi), %r8  # i-1
imulq  %rcx, %rsi     # i*n
imulq  %rcx, %rax     # (i+1)*n
imulq  %rcx, %r8      # (i-1)*n
addq   %rdx, %rsi     # i*n+j
addq   %rdx, %rax     # (i+1)*n+j
addq   %rdx, %r8      # (i-1)*n+j
```

### 1 multiplication: i*n

```assembly
imulq  %rcx, %rsi     # i*n
addq   %rdx, %rsi     # i*n+j
movq   %rsi, %rax     # i*n+j
subq   %rcx, %rax     # i*n+j-n
leaq   (%rsi,%rcx), %rcx # i*n+j+n
```
Optimization Blocker #1: Procedure Calls

- Procedure to convert string to lower case

```c
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```
Lower Case Conversion Performance

- Time quadruples when string length doubles
- $O(N^2)$
Convert Loop To Goto Form

- `strlen` executed every iteration

```c
void lower(char *s)
{
    size_t i = 0;
    if (i >= strlen(s))
        goto done;
    loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
    done:
}
```
Calling Strlen

- **strlen performance**
  - Only way to determine length of string is to scan its entire length, looking for null character

- Overall performance for string of length N:
  - N calls to `strlen`
  - Each call to `strlen` iterates over all N characters in the string
  - Overall $O(N^2)$ performance

```c
/* My version of strlen */
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```
Improving Performance

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

```c
void lower(char *s)
{
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```
Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2
Optimization Blocker: Procedure Calls

• Why couldn’t compiler move strlen out of inner loop?
  • Procedure may have side effects
    • May alter some global state each time it is called
  • Function may not return same value for the same given arguments
    • Depends on other parts of global state
    • Procedure lower could interact with strlen

• Warning:
  • Compiler treats procedure call as a black box

• Remedies:
  • Use of inline functions
    • GCC does this with –O1
      • Within single file
  • Do your own code motion
Memory Matters

• Both functions seem to have identical behavior
• Both add twice the value at the location \( yp \) to the location \( xp \)
• \texttt{sum2} has less memory accesses
• The compiler won’t replace \texttt{sum2} for \texttt{sum1}
• What if \( xp \) and \( yp \) point to the same memory location?
  • \texttt{sum1} and \texttt{sum2} will give different answers

```c
void sum1(long *xp, long *yp) {
    *xp += *yp;
    *xp += *yp;
}

void sum2(long *xp, long *yp) {
    *xp += 2 * *yp;
}
```

```assembly
sum1:
    movq (%rdi), %rax
    addq (%rsi), %rax
    movq %rax, (%rdi)
    addq (%rsi), %rax
    movq %rax, (%rdi)
    ret

sum2:
    movq (%rsi), %rax
    addq %rax, %rax
    addq %rax, (%rdi)
    ret
```
Optimization Blocker: Memory Aliasing

- **Aliasing**
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

```c
void sum2(long *xp, long *yp) {
    *xp += 2 * *yp;
}

void sum3(long *xp, long *yp) {
    long sum;
    sum += *yp;
    sum += *yp;
    *xp = sum;
}
```

```assembly
sum2:
    movq (%rsi), %rax
    addq %rax, %rax
    addq %rax, (%rdi)
    ret

sum3:
    movq (%rsi), %rax
    addq %rax, %rax
    addq %rax, (%rdi)
    ret
```
Benchmark Example: Data Type for Vectors

/* data structure for vectors */
typedef struct {
    size_t len;
    data_t *data;
} vec;

/* retrieve vector element and store at val */
/* return 1 if successful, 0 otherwise */
int get_vec_element(*vec v, size_t idx, data_t *val) {
    if (idx >= v->len) {
        return 0;
    }
    *val = v->data[idx];
    return 1;
}

• Data Types
  • Use different declarations for data_t
  • int
  • long
  • float
  • double
Benchmark Computation

```c
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

- **Data Types**
  - Use different declarations for `data_t`
    - `int`
    - `long`
    - `float`
    - `double`

- **Operations**
  - Use different definitions of `OP` and `IDENT`
    - `+ / 0`
    - `* / 1`
Cycles Per Element (CPE)

• Convenient way to express performance of program that operates on vectors or lists
• Length = n
• In our case: CPE = cycles per OP
• T = CPE*n + Overhead
  • CPE is slope of line

![Graph showing cycles per element (CPE) for two operations, psum1 and psum2, with slopes of 9.0 and 6.0 respectively.](image)
void combine1(vec_ptr v, data_t *dest) {
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}

Compute sum or product of vector elements

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>22.68</td>
<td>20.02</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
</tr>
</tbody>
</table>
Basic Optimizations

- Move `vec_length` out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

```c
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
    {
        t = t OP d[i];
    }
    *dest = t;
}
```
Effect of Basic Optimizations

```c
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
</tbody>
</table>

- Eliminates sources of overhead in loop
Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
Modern CPU Design

Execution

Instruction Control

Register Updates

Prediction OK?

Operation Results

Data Cache

Functional Units

Branch Arith Arith Arith Load Store

Operation Results

Addr. Addr. Data Data

Retirement Unit Register File

Fetch Control Instruction Decode

Address Instructions Operations

Instruction Cache
Superscalar Processor

• Definition: A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

• Benefit: without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have

• Most modern CPUs are superscalar
  • Intel: since Pentium (1993)
Pipelined Functional Units

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

```c
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>a*b</td>
<td>a*c</td>
<td></td>
<td></td>
<td>p1*p2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td>a*b</td>
<td>a*c</td>
<td></td>
<td></td>
<td>p1*p2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>a*b</td>
<td>a*c</td>
<td></td>
<td></td>
<td></td>
<td>p1*p2</td>
<td></td>
</tr>
</tbody>
</table>
Haswell CPU

• 8 Total Functional Units
• Multiple instructions can execute in parallel
  2 load, with address computation
  1 store, with address computation
  4 integer
  2 FP multiply
  1 FP add
  1 FP divide

• Some instructions take > 1 cycle, but can be pipelined

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>3-30</td>
<td>3-30</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>3-15</td>
<td>3-15</td>
</tr>
</tbody>
</table>
x86-64 Compilation of Combine4

• Inner Loop (Case: Integer Multiply)

```assembly
.L519:    # Loop:
imull (%rax,%rdx,4), %ecx # t = t * d[i]
addq $1, %rdx        # i++
cmpq %rdx, %rbp     # Compare length:i
jg .L519            # If >, goto Loop
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>
Combine4 = Serial Computation (OP = *)

• Computation (length=8)
  \(((1 \times d[0]) \times d[1]) \times d[2] \times d[3] \times d[4] \times d[5] \times d[6] \times d[7])

• Sequential dependence
  • Performance: determined by latency of OP
Loop Unrolling (2x1)

- Perform 2x more useful work per iteration

```c
void unroll2a_combine(vec_ptr v, data_t *dest) {
    long length = vec_length(v);
    long limit = length - 1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```
Effect of Loop Unrolling

- Helps integer add
  - Achieves latency bound
- Others don’t improve. *Why?*
  - Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```
Loop Unrolling with Reassociation (2x1a)

Can this change the result of the computation?
Yes, for floating point numbers. Why?
• Floating point numbers are not associative in all cases!

```c
void unroll2aa_combine(vec_ptr v, data_t *dest) {
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before
```
x = (x OP d[i]) OP d[i+1];
```
Effect of Reassociation

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency

\[
x = x \text{ OP } (d[i] \text{ OP } d[i+1]);
\]

- Why is that? (next slide)

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- 2 func. units for FP *
- 2 func. units for load
Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```

- What changed:
  - Ops in the next iteration can be started early (no dependency)
Loop Unrolling with Separate Accumulators (2x2)

void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}

• Different form of reassociation
Effect of Separate Accumulators

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td>Unroll 2x2</td>
<td>0.81</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Int + makes use of two load units
  
  \[ x_0 = x_0 \text{ OP } d[i]; \]
  
  \[ x_1 = x_1 \text{ OP } d[i+1]; \]

- 2x speedup (over unroll2) for Int *, FP +, FP *
Separate Accumulators

x0 = x0 OP d[i];
x1 = x1 OP d[i+1];

- **What changed:**
  - Two independent “streams” of operations

- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be \((N/2+1)\)*D cycles:
    \[
    \text{CPE} = \frac{D}{2}
    \]
  - CPE matches prediction!

**What Now?**
Unrolling & Accumulating

• Idea
  • Can unroll to any degree L
  • Can accumulate K results in parallel
  • L must be multiple of K

• Limitations
  • Diminishing returns
    • Cannot go beyond throughput limitations of execution units
  • Large overhead for short lengths
    • Finish off iterations sequentially
Unrolling & Accumulating: Double *

- Case
  - Intel Haswell
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 0.50 (Issue: 1, Capacity 2)

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>K</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5.01</td>
</tr>
<tr>
<td>2</td>
<td>2.51</td>
</tr>
<tr>
<td>3</td>
<td>1.67</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Int +

- Case
  - Intel Haswell
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 0.50

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1.27</td>
</tr>
<tr>
<td>2</td>
<td>0.81</td>
</tr>
<tr>
<td>3</td>
<td>0.74</td>
</tr>
<tr>
<td>4</td>
<td>0.69</td>
</tr>
<tr>
<td>6</td>
<td>0.56</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Achievable Performance

- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>0.54</td>
<td>1.01</td>
<td>1.01</td>
<td>0.52</td>
</tr>
<tr>
<td>Mult</td>
<td>1.01</td>
<td></td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
</tr>
</tbody>
</table>
Programming with AVX2 (Advanced Vector Extensions)

- YMM Registers: 16 total, each 32 bytes
  - 32 single-byte integers
  - 16 16-bit integers
  - 8 32-bit integers
  - 8 single-precision floats
  - 4 double-precision floats
  - 1 single-precision float
  - 1 double-precision float
SIMD (Single Instruction Multiple Data) Operations

• SIMD Operations: Single Precision

\[ \text{vaddsd} \quad \%y\text{mm}0, \%y\text{mm}1, \%y\text{mm}1 \]

• SIMD Operations: Double Precision

\[ \text{vaddpd} \quad \%y\text{mm}0, \%y\text{mm}1, \%y\text{mm}1 \]
Using Vector Instructions

- Make use of AVX Instructions
  - Parallel operations on multiple data elements
  - See Web Aside OPT:SIMD on CS:APP web page

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar Best</td>
<td>0.54</td>
<td>1.01</td>
</tr>
<tr>
<td>Vector Best</td>
<td>0.06</td>
<td>0.24</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>0.50</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>Vec Throughput Bound</td>
<td>0.06</td>
<td>0.12</td>
</tr>
</tbody>
</table>
Getting High Performance

• Good compiler and flags
• Don’t do anything silly
  • Watch out for hidden algorithmic inefficiencies
  • Write compiler-friendly code
    • Watch out for optimization blockers: procedure calls & memory references
    • Look carefully at innermost loops (where most work is done)
• Tune code for machine
  • Exploit instruction-level parallelism
  • Make code cache friendly
I DON'T ALWAYS OPTIMIZE MY CODE

BUT WHEN I DO, I TAKE ADVANTAGE OF THE UNDERLYING HARDWARE