Performance is

- Constant factors matter too (e.g. $O(n^2)$, $O(n^3)$)!
- Even exact operation count does not predict performance
  - Easily see 10:1 performance gain depending on how code is written
  - Remember loop interchange in matrix multiplication?
- Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs compiled and executed (optimization levels)
  - 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 – top level of memory hierarchy
  - Code selection and ordering (scheduling) – data/control dependences
  - Dead code elimination – reduce number of instructions executed
  - Eliminating minor inefficiencies
- 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
- Have difficulty overcoming “optimization blockers”
  - Potential memory aliasing
  - Potential procedure side-effects

Limitations of Optimizing Compilers

- Operate under fundamental constraint
  - Must not cause any change in program behavior
  - Often prevents optimizations even if it only affects pathological cases
    (often need programmers’ intervention)
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles to a compiler
  - 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 (time and space) in most cases
- Most analysis is based only on static (i.e. compile time) information
  - Compiler has difficulty anticipating run-time inputs (e.g. pointers’ targets)
- When in doubt, the compiler must be conservative to guarantee correctness

Outline

- Overview
- Generally Useful Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploiting Instruction-Level Parallelism
- Dealing with Conditionals
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

Reduction in Strength

- Replace costly operation with simpler one
- shift, add instead of multiply or divide
  \( 16 \times n \leftrightarrow n \leq 4 \)
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
  - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

Outline

- Overview
- Generally Useful Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
  - Exploiting Instruction-Level Parallelism
  - Dealing with Conditionals

Compiler-Generated Code Motion

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

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

Share Common Subexpressions

- Reuse portions of expressions
- Compilers not very sophisticated in exploiting arithmetic properties
- Example: reduce 8 instructions to 5 instructions (1/8 = 12.5% reduction)

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

Optimization Blocker #1: Procedure Calls
Calling Strlen

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

- Overall performance, string of length N
  - N calls to `strlen`
  - Require times N
  - Overall $O(N^2)$ performance

Optimization Blocker: Procedure Calls

- Why couldn’t compiler move `strlen` out of inner loop?
  - Procedure may have side effects
    - Alter global state each time called
    - Function may not return same value for given arguments
      - Depending on other parts of global state
    - Procedure `lower` could interact with `strlen`

  **Warning:**
  - Compiler treats function call as black box
  - Weak optimizations near them

  **Remedies:**
  - Use of inlined functions
    - GCC does this with -O2
  - Do your own code motion

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

```
void lower2(char *s)
{
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

```
int len = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '0') {
        s++;
        length++;
    }
    return length;
}
```

Memory Matters

- `X86-64` has 16 floating-point registers `%xmm0 - %xmm15`
  - Each %xmm register is 128-bit wide and can hold 4 32-bit floating point numbers
- `X86-64` has 16 general-purpose registers, with additional `%r8 - %r15`

- Code updates `b[1]` on every iteration
  - Why couldn’t compiler optimize this away?

```
void sum_rows1_double(double *a, double *b, long n) {
    long i;
    double sum = 0;
    for (i = 0; i < n; i++) {
        sum += a[i] + b[i];
    }
    return sum;
}
```
Memory Aliasing

- Code updates $b[i]$ on every iteration
- Must consider possibility that these updates will affect program behavior

Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references specify same location
- Easy to have happened 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
- Avoid using pointers if you can!!

Exploiting Instruction-Level Parallelism (ILP)

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel (i.e. instruction-level parallelism, ILP)
- Performance limited by data dependencies
- Simple transformations can have dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

Removing Aliasing

```c
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```
### Benchmark Example: Data Type for Vectors

```c
/* data structure for vectors */
typedef struct{
  int len;
  double *data;
} vec;
```

```c
/* retrieve vector element and store at val */
double get_vec_element(vec *vec, int idx, double *val)
{
  if (idx < 0 || idx > vec->len)
    return 0;
  *val = vec->data[idx];
  return 1;
}
```

### 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;
}
```

### Benchmark Performance

### Basic Optimizations

```c
void combine2(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    i = i OP d[i];
  *dest = t;
}
```

```c
void combine3(vec_ptr v, data_t *dest)
{
  int i;
  int 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)
{
  int i;
  int 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;
}
```

### 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

### Benchmark Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine1</td>
<td>29.0</td>
<td>27.4</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>

### Effect of Basic Optimizations

- Move `vec_length` out of loop – **code motion**
- Avoid bounds check on each cycle in `get_vec_element`
- Use `get_vec_start` instead of `get_vec_element`
- Move it outside of loop (code motion)
- Accumulate in temporary
- Eliminates sources of overhead in loop
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 scheduled/run dynamically.

**Benefit:**
- Without programming effort, superscalar processors can take advantage of the instruction level parallelism (ILP) that most programs have
- Most CPUs since about 1998 are superscalar.
- Intel: since Pentium Pro

Intel Nehalem CPU

- Multiple instructions can execute in parallel
  - 1 load, with address computation
  - 1 store, with address computation
  - 2 simple integer (one may be branch)
  - 1 complex integer (multiply/divide)
  - 1 FP Multiply
  - 1 FP Add
- 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>11–21</td>
<td>11–21</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>4/5</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>10–23</td>
<td>10–23</td>
</tr>
</tbody>
</table>

**Combine4 = Serial Computation (OP = *)**

- Computation (length=8)
  ```
  (((((1 * d[0]) * d[1]) * d[2]) * d[3]) * d[4]) * d[5]) * d[6]) * d[7])
  ```
- Sequential dependence
  - Performance: determined by latency of OP
  - Critical path determines the performance

Modern CPU Design

**x86-64 Compilation of Combine4**

- Inner Loop (Case: Integer Multiply)

```c
void combine4(vec_ptr v, data_t *dest) {
  int i;
  int length = vec_length(v);
  int limit = length-1;
  data_t t = get_vec_start(v);
  data_t s = [IDB];
  for (i = 0; i < length; i++) {
    t = t * d[i];
  }
  for (i = 0; i < length; i++) {
    t = t + d[i];
  }
  for (i = 0; i < length; i++) {
    t = t - d[i];
  }
  for (i = 0; i < length; i++) {
    t = t / d[i];
  }
  *dest = t;
}
```

**Loop Unrolling**

```c
void unroll2_combine(vec_ptr v, data_t *dest) {
  int i;
  int length = vec_length(v);
  int limit = length-1;
  data_t t = get_vec_start(v);
  data_t s = [IDB];
  for (i = 0; i < length; i++) {
    t = x OP d[i];
  }
  for (i = 0; i < length; i++) {
    t = x OP d[i];
  }
  *dest = x;
}
```

**Benefit:**
- With programming effort, superscalar processors can take advantage of the instruction level parallelism (ILP) that most programs have
- Without programming effort, superscalar processors can take advantage of the instruction level parallelism (ILP) that most programs have
- Most CPUs since about 1998 are superscalar.
- Intel: since Pentium Pro

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

**Benefit:**
- Without programming effort, superscalar processor can take advantage of the instruction level parallelism (ILP) that most programs have
- Most CPUs since about 1998 are superscalar.
- Intel: since Pentium Pro

Intel Nehalem CPU

- Multiple instructions can execute in parallel
  - 1 load, with address computation
  - 1 store, with address computation
  - 2 simple integer (one may be branch)
  - 1 complex integer (multiply/divide)
  - 1 FP Multiply
  - 1 FP Add
- 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>11–21</td>
<td>11–21</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>4/5</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>10–23</td>
<td>10–23</td>
</tr>
</tbody>
</table>

**Combine4 = Serial Computation (OP = *)**

- Computation (length=8)
  ```
  (((((1 * d[0]) * d[1]) * d[2]) * d[3]) * d[4]) * d[5]) * d[6]) * d[7])
  ```
- Sequential dependence
  - Performance: determined by latency of OP
  - Critical path determines the performance

**Modern CPU Design**

**x86-64 Compilation of Combine4**

- Inner Loop (Case: Integer Multiply)

```c
void combine4(vec_ptr v, data_t *dest) {
  int i;
  int length = vec_length(v);
  int limit = length-1;
  data_t t = get_vec_start(v);
  data_t s = [IDB];
  for (i = 0; i < length; i++) {
    t = t * d[i];
  }
  for (i = 0; i < length; i++) {
    t = t + d[i];
  }
  for (i = 0; i < length; i++) {
    t = t - d[i];
  }
  for (i = 0; i < length; i++) {
    t = t / d[i];
  }
  *dest = t;
}
```

**Loop Unrolling**

```c
void unroll2_combine(vec_ptr v, data_t *dest) {
  int i;
  int length = vec_length(v);
  int limit = length-1;
  data_t t = get_vec_start(v);
  data_t s = [IDB];
  for (i = 0; i < length; i++) {
    t = x OP d[i];
  }
  for (i = 0; i < length; i++) {
    t = x OP d[i];
  }
  *dest = x;
}
```
Effect of Loop Unrolling

<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>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

- Helps integer multiply
  - below latency bound
  - Compiler does clever optimization
- Others don’t improve. Why?
  - Still sequential dependency

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

Effect of Re-association

<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>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Reassociate</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential
  - x = x OP (d[i] OP d[i+1]);
  - Why is that? (next slide)

Re-associated Computation

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

- What changed:
  - Ops in the next iteration can be started early (no dependency)
- Overall Performance
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles:
  - CPE = D/2
  - Measured CPE slightly worse for FP mult

Loop Unrolling with Separate Accumulators

```
void unroll2a_combine(vect_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int 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++) { x = x OP d[i]; }
    *dest = x;
}
```

Separate Accumulators

- What changed:
  - Two independent "streams" of operations
- Overall Performance
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles:
  - CPE = D/2
  - CPE matches prediction!

What Now?
Effect of Separate Accumulators

- 2x speedup (over unroll2) for Int *, FP +, FP *
  - Breaks sequential dependency in a "cleaner," more obvious way

\[x_0 = x_0 \text{ OP } d[i];
\]
\[x_1 = x_1 \text{ OP } d[i+1];\]

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 functional units
    - Large overhead for short lengths
    - Finish off iterations sequentially

Unrolling & Accumulating: Double *

- Case
  - Intel Nehalem (Shark machines)
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 1.00

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1.00</td>
</tr>
<tr>
<td>12</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Accumulators

- K streams

Unrolling & Accumulating: Int +

- Case
  - Intel Nehalem (Shark machines)
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 1.00

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1.00</td>
</tr>
<tr>
<td>12</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Accumulators

- K streams

Review: Achievable Performance

- Limited only by throughput of functional units
- Up to 29X improvement over original, unoptimized code

Using Vector Instructions

- Make use of Intel SSE (Streaming SIMD Extension) Instructions
- Parallel operations on multiple data elements

The more we know about the machine, the higher the performance we could get for our programs!!
What About Branches?

- **Challenge**
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

```
0489F3: movl $0x1,%ecx
0489F6: movl %edx,%edx
0489F9a: cmpq %esi,%edx
0489Fc: jnl 8048a25
8048a00: imull (%eax,%edx,4),%ecx
```

- When encounters a **conditional branch**, cannot reliably determine where to continue fetching

Outline

- **Overview**
  - Generally Useful Optimizations
    - Code motion/precomputation
    - Strength reduction
    - Sharing of common subexpressions
    - Removing unnecessary procedure calls
  - Optimization Blockers
    - Procedure calls
    - Memory aliasing
  - Exploiting Instruction-Level Parallelism
- **Dealing with Conditionals**

Modern CPU Design

- **Instruction Control**
  - Fetch Control
  - Address
  - Instruction Cache
- **Register Update**
- **Execution**
  - Data Cache
  - Instruction Cache
  - Fetch Control
  - Address
  - Instructions
  - Operations

Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
  - **Branch Not-Taken**: Continue with next instruction in sequence
  - **Branch Taken**: Transfer control to branch target
  - Cannot resolve until outcome determined by branch/integer unit

```
8048a25: cmpq %edi,%edx
8048a27: jl 8048a20
8048a29: movl 0x0(%ebp),%eax
8048a2c: leal 0x0(%ebp),%esp
8048a2f: movl %ecx,%eax
```

Branch Prediction

- **Idea**
  - Guess which way branch will go
  - Begin executing instructions at predicted position
  - But don’t actually modify register or memory data

```
8048a25: cmpq %esi,%edx
8048a27: jl 8048a20
8048a29: movl 0x0(%ebp),%eax
8048a2c: leal 0x0(%ebp),%esp
8048a2f: movl %ecx,%eax
```

Branch Prediction Through Loop

```
80488b1: movl (%eax,%edx,4),%eax
80488b4: addl %eax,%edi
80488b6: incl %edx
80488b7: cmpq %esi,%edx
80488b9: jl 80488b1
```

- Predict Taken (OK)
- Predict Taken (Oops)
- Read invalid location
- Executed
- Fetched
- Assume vector length = 100
Effect of Branch Prediction — Bound Check

- **Loops**
  - Typically, only miss when hit loop end

- **Checking code**
  - Array bound check - reliably predicts that error won’t occur

```
void combine4b(vec_ptr v, data_t* dest)
{
  long int i;
  long int length = vec_length(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
  { if (i >= 0 && i < v->len) {
      t = t OP v->data[i];
    }
  }
  *dest = t;
}
```

---

**Branch Misprediction Invalidation**

Assume vector length = 100

- Predict Taken (OK)

- Predict Taken (Oops)

Invalidate

- Performance Cost
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

---

**Branch Misprediction Recovery**

- Definitively not taken

- Good compiler and appropriate flags
  - Usually not easy to find all appropriate flags

- Don’t do anything stupid
  - 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
  - Avoid unpredictable branches
  - Make code cache friendly
  - May need to profile the program execution to collect runtime information for performance tuning