Program Optimization

Spring 2012

Instructors:
Aykut & Erkut Erdem

Acknowledgement: The course slides are adapted from the slides prepared by R.E. Bryant, D.R. O’Hallaron, G. Kesden and Markus Püschel of Carnegie-Mellon Univ.
Today

- Overview
- Program optimization
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
- Exploiting Instruction-Level Parallelism
Harsh Reality

- *There’s more to runtime performance than asymptotic complexity*

- *One can easily lose 10x, 100x in runtime or even more*

- **What matters:**
  - Constants (100n and 5n is both O(n), but ....)
  - Coding style (unnecessary procedure calls, unrolling, reordering, ...)
  - Algorithm structure (locality, instruction level parallelism, ...)
  - Data representation (complicated structs or simple arrays)
Harsh Reality

- Must optimize at multiple levels:
  - Algorithm
  - Data representations
  - Procedures
  - Loops

- Must understand system to optimize performance
  - How programs are compiled and executed
    - Execution units, memory hierarchy
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
Optimizing Compilers

- Use optimization flags, **default is no optimization** (-O0)!
- Good choices for gcc: -O2, -O3, -march=xxx, -m64
- Try different flags and maybe different compilers
Example

```c
double a[4][4];
double b[4][4];
double c[4][4]; // set to zero

/* Multiply 4 x 4 matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                c[i*4+j] += a[i*4 + k]*b[k*4 + j];
}
```

- Compiled without flags:
  ~1300 cycles
- Compiled with `-O3 -m64 -march=... -fno-tree-vectorize`
  ~150 cycles
- Core 2 Duo, 2.66 GHz
Optimizing Compilers

- Compilers are **good** at: mapping program to machine instructions
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies

- Compilers are **not good** at: improving 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

- Compilers are **not good** at: overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects
Limitations of Optimizing Compilers

- *If in doubt, the compiler is conservative*
- Operate under fundamental constraints
  - Must not change program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.
- 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
- Most analysis is based only on *static* information
  - Compiler has difficulty anticipating run-time inputs
Today

■ Overview

■ **Program optimization**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing

■ **Exploiting Instruction-Level Parallelism**
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];
}
```

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

long j;
long ni = n*i;
double *rowp = a+ni;
for (j = 0; j < n; j++)
    *rowp++ = b[j];

set_row:
    testq %rcx, %rcx
    jle .L4
    movq %rcx, %rax
    imulq %rdx, %rax
    leaq (%rdi,%rax,8), %rdx
    movl $0, %r8d
    .L3:
    movq (%rsi,%r8,8), %rax
    movq %rax, (%rdx)
    addq $1, %r8
    addq $8, %rdx
    cmpq %r8, %rcx
    jg .L3
    rep ; ret

Where are the FP operations?
Today

- Overview

- Program optimization
  - Code motion/precomputation
  - **Strength reduction**
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing

- Exploiting Instruction-Level Parallelism
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++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];

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

- Overview

- Program optimization
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing

- Exploiting Instruction-Level Parallelism
Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```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
```
Today

- Overview
- **Program optimization**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - **Optimization blocker: Procedure calls**
  - Optimization blocker: Memory aliasing
- **Exploiting Instruction-Level Parallelism**
**Optimization Blocker #1: Procedure Calls**

**Procedure to Convert String to Lower Case**

```c
void lower(char *s)
{
    int 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 double string length
- Quadratic performance
Convert Loop To Goto Form

```c
void lower(char *s)
{
    int 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:
}
```

- `strlen` executed every iteration
Calling Strlen

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

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

- **Overall performance, string of length N**
  - N calls to strlen
  - Require times N, N-1, N-2, ..., 1
  - Overall O(N²) performance
Improving Performance

```c
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');
}
```

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion
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
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure lower could interact with strlen

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

- **Remedies:**
  - Use of inline functions
    - GCC does this with –O2
    - See web aside ASM:OPT
  - Do your own code motion

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

- Overview
- **Program optimization**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - **Optimization blocker: Memory aliasing**
- Exploiting Instruction-Level Parallelism
Memory Matters

/* Sum rows is of n X n matrix a and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}

# sum_rows1 inner loop
.L53:
    addsd (%rcx), %xmm0  # FP add
    addq $8, %rcx
    decq %rax
    movsd %xmm0, (%rsi,%r8,8)  # FP store
    jne .L53

- Code updates b[i] on every iteration
- Why couldn’t compiler optimize this away?
Memory Aliasing (Simple Example)

```c
void twiddle1(int *xp, int *yp) {
    *xp += *yp;
    *xp += *yp;
}

void twiddle2(int *xp, int *yp) {
    *xp += 2 * *yp;
}
```

What if `xp` and `yp` point to the same address?

```c
twiddle1:
    *xp += *yp;  // *xp = 2 + 2 = 4
    *xp += *yp;  // *xp = 4 + 4 = 8

twiddle2:
    *xp += 2 * (*yp);  // *xp = 2 + 2*2 = 6
```

```c
void twiddle1(int *xp, int *yp) {
    *xp += *yp;
    *xp += *yp;
}

void twiddle2(int *xp, int *yp) {
    *xp += 2 * *yp;
}
```

```c
i=2, j=2;
xp = &i; yp = &j;

*xp += *yp;  // *xp = 2 + 2 = 4
*xp += *yp  // *xp = 4 + 2 = 6

*xp += 2 * (*yp);  // *xp = 2 + 2*2 = 6
```

What if `xp` and `yp` point to the same address?

```c
int i=2;
xp = yp = &i;

twiddle1:
    *xp += *yp;  // *xp = 2 + 2 = 4
    *xp += *yp;  // *xp = 4 + 4 = 8

twiddle2:
    *xp += 2 * (*yp);  // *xp = 2 + 2*2 = 6
```
Memory Aliasing

```c
/* Sum rows is of n X n matrix a 
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

double A[9] =
{ 0, 1, 2,
  4, 8, 16},
32, 64, 128;


sum_rows1(A, B, 3);

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

Value of B:

| i = 0: [3, 8, 16] |
| i = 1: [3, 22, 16] |
| i = 2: [3, 22, 224] |
Removing Aliasing

/* Sum rows is of n X n matrix a
   and store in vector b */
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;
    }
}

# sum_rows2 inner loop
.L66:
    addsd (%rcx), %xmm0   # FP Add
    addq $8, %rcx
    decq %rax
    jne .L66

- No need to store intermediate results
Optimization Blocker: Memory Aliasing

- Memory aliasing: Two different memory references write to the same location
- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- Hard to analyze = compiler cannot figure it out
  - Hence is conservative

Solution: Scalar replacement in innermost loop
- Copy memory variables that are reused into local variables
- Basic scheme:
  - Load: \( t1 = a[i], t2 = b[i+1], \ldots \)
  - Compute: \( t4 = t1 \times t2; \ldots \)
  - Store: \( a[i] = t12, b[i+1] = t7, \ldots \)
Today

- Overview
- Program optimization
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
- Exploiting Instruction-Level Parallelism
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 have dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic
Benchmark Example: Data Type for Vectors

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

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

- **Data Types**
  - Use different declarations for `data_t`
    - `int`
    - `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

\[
CPE = \text{slope of line}
\]

\[
T = CPE \times n + \text{Overhead}
\]

\[
v\text{sum1: Slope} = 4.0
\]

\[
v\text{sum2: Slope} = 3.5
\]
Benchmark Performance

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;
    }
}

<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>29.0</td>
<td>29.2</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>

Compute sum or product of vector elements
Basic Optimizations

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;
}

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

<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</td>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

- Eliminates sources of overhead in loop
Modern CPU Design

**Instruction Control**

- **Fetch Control**
- **Instruction Decode**
- **Instruction Cache**
- **Address**
- **Instructions**
- **Operations**
- **Retirement Unit**
- **Register File**
- **Register Updates**
- **Prediction OK?**

**Execution**

- **Functional Units**
  - **Load**
  - **Store**
  - **Integer/Branch**
  - **General Integer**
  - **FP Add**
  - **FP Mult/Div**
- **Operation Results**
- **Addr.**
- **Data**
- **Data Cache**
Latency versus Throughput

- **Example:**
  
<table>
<thead>
<tr>
<th>Integer Multiply</th>
<th>latency</th>
<th>cycles/issue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Consequence:**
  - How fast can 10 independent int mults be executed?
    \[ t_1 = t_2 \times t_3; \quad t_4 = t_5 \times t_6; \quad \ldots \]
  - How fast can 10 sequentially dependent int mults be executed?
    \[ t_1 = t_2 \times t_3; \quad t_4 = t_5 \times t_1; \quad t_6 = t_7 \times t_4; \quad \ldots \]

- **Major problem for fast execution:** Keep pipelines filled
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 CPUs since about 1998 are superscalar.
- Intel: since Pentium Pro
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>
x86-64 Compilation of Combine4

- Inner Loop (Case: Integer Multiply)

```assembly
.L519:
    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>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Add</th>
<th>Mult</th>
<th>Add</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine4</td>
<td>3.0</td>
<td>5.0</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>3.0</td>
<td>5.0</td>
<td>3.0</td>
<td>5.0</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

void unroll2a_combine(vec_ptr v, data_t *dest)
{
  int length = vec_length(v);
  int limit = length-1;
  data_t *d = get_vec_start(v);
  data_t x = IDENT;
  int 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;
}

- Perform 2x more useful work per iteration
Effect of Loop Unrolling

<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>Operation</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

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

```c
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int 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;
}
```

- Can this change the result of the computation?
- Yes, for FP. *Why?*
# Effect of Reassociation

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></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>Unroll 2x, 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 dependency
    
    ```
    x = x OP (d[i] OP d[i+1]);
    ```
  - Why is that? (next slide)
Reassociated Computation

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

- **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

```c
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    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++) {
        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></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>Unroll 2x, reassociate</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Unroll 2x Parallel 2x</td>
<td>1.5</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>

- 2x speedup (over Combine4) for Int *, FP +, FP *
  - Breaks sequential dependency in a “cleaner,” more obvious way

```plaintext
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```
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?
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 Nehelam
- 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>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>5.00</td>
</tr>
<tr>
<td>2</td>
<td>2.50</td>
</tr>
<tr>
<td>3</td>
<td></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 Nehelam
  - 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>K 1</td>
<td>2.00 2.00</td>
</tr>
<tr>
<td>K 2</td>
<td>1.50</td>
</tr>
<tr>
<td>K 3</td>
<td>1.00</td>
</tr>
<tr>
<td>K 4</td>
<td>1.00</td>
</tr>
<tr>
<td>K 6</td>
<td>1.00</td>
</tr>
<tr>
<td>K 8</td>
<td>1.03</td>
</tr>
<tr>
<td>K 10</td>
<td></td>
</tr>
<tr>
<td>K 12</td>
<td></td>
</tr>
</tbody>
</table>
## Achievable Performance

<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>1.00</td>
<td>1.00</td>
<td>Add</td>
<td>1.00</td>
</tr>
<tr>
<td>Mult</td>
<td>1.00</td>
<td>1.00</td>
<td>Mult</td>
<td>1.00</td>
</tr>
<tr>
<td>Scalar Optimum</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</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>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar Optimum</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Vector Optimum</td>
<td>0.25</td>
<td>0.53</td>
<td>0.53</td>
<td>0.57</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>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Vec Throughput Bound</td>
<td>0.25</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
</tr>
</tbody>
</table>

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

- Section 5.14 – Identifying and Eliminating Performance Bottlenecks
  - Program Profiling
  - Using a Profiler to Guide Optimization
  - Amdahl’s Law
Getting High Performance

- **Good compiler and 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 (not covered)
  - Make code cache friendly (covered in course before)