Introduction to Computer Systems
15-213/18-243, spring 2009
10th Lecture, Oct. 1st

Instructors:
Roger B. Dannenberg and Greg Ganger
Today

- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism
Example Matrix Multiplication

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz
Gflop/s (giga floating point operations per second)

- Standard desktop computer, compiler, using optimization flags
- Both implementations have exactly the same operations count (2n³)
- What is going on?
MMM Plot: Analysis

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz
Gflop/s

Multiple threads: 4x  (towards end of course)

Vector instructions: 4x  (not in this course)

Memory hierarchy and other optimizations: 20x

- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice

- Effect: more instruction level parallelism, better register use, less L1/L2 cache misses, less TLB misses
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, \textit{default is no optimization} (-O0)!
- Good choices for gcc: -O2, -O3, -march=xxx, -m64
- Try different flags and maybe different compilers
Example

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
  - 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

- **Program optimization**
  - Overview
  - **Removing unnecessary procedure calls**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism
Example: Data Type for Vectors

/* 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;
}
Example: Summing Vector Elements

```c
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double val;

    for (i = 0; i < n; i++) {
        get_vec_element(v, i, &val);
        *res += val;
    }
    return res;
}
```

```c
/* 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;
}
```

Bound check unnecessary in `sum_elements` Why?

Overhead for every fp +:
- One fct call
- One <
- One >=
- One ||
- One memory variable access

Slowdown: probably 10x or more
Removing Procedure Call

/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double val;

    for (i = 0; i < n; i++) {
        get_vec_element(v, i, &val);
        *res += val;
    }
    return res;
}

/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double *data = get_vec_start(v);

    for (i = 0; i < n; i++)
        *res += data[i];
    return res;
}
Removing Procedure Calls

- Procedure calls can be very expensive
- Bound checking can be very expensive
- Abstract data types can easily lead to inefficiencies
  - Usually avoided for in superfast numerical library functions

- Watch your innermost loop!

- Get a feel for overhead versus actual computation being performed
Today

■ Program optimization
  ▪ Overview
  ▪ Removing unnecessary procedure calls
  ▪ Code motion/precomputation
  ▪ Strength reduction
  ▪ Sharing of common subexpressions
  ▪ Optimization blocker: Procedure calls
  ▪ Optimization blocker: Memory aliasing
  ▪ Out of order processing: Instruction level parallelism
Code Motion

- Reduce frequency with which computation is performed
  - If it will always produce same result
  - Especially moving code out of loop
- Sometimes also called precomputation

```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];
```
Compiler-Generated Code Motion

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];
}

Where are the FP operations?

set_row:
    xorl %r8d, %r8d           #  j = 0
    cmpq %rcx, %r8            #  j:n
    jge .L7                   #  if >= goto done
    movq %rcx, %rax           #  n
    imulq %rdx, %rax          #  n*i outside of inner loop
    leaq (%rdi,%rax,8), %rdx  #  rowp = A + n*i*8
.L5:
    movq (%rsi,%r8,8), %rax   #  t = b[j]
    incq %r8                  #  j++
    movq %rax, (%rdx)         #  *rowp = t
    addq $8, %rdx             #  rowp++
    cmpq %rcx, %r8            #  j:n
    jl .L5                    #  if < goto loop
.L7:
    rep ; ret                 #  done:
    # return
Today

- **Program optimization**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - **Strength reduction**
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism
Strength Reduction

- Replace costly operation with simpler one
- Example: Shift/add instead of multiply or divide
  \[ 16 \times x \quad \rightarrow \quad x \ll 4 \]
  - Benefits are machine dependent
  - Depends on cost of multiply or divide instruction
  - On Pentium IV, integer multiply requires 10 CPU cycles

- Example: Recognize sequence of products

```plaintext
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

- **Program optimization**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - **Sharing of common subexpressions**
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism
Share Common Subexpressions

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

3 mults: \( i*n, (i-1)*n, (i+1)*n \)

```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;
```

1 mult: \( i*n \)

```c
int 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;
```

```
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
```

```
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

- **Program optimization**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - **Optimization blocker: Procedure calls**
  - Optimization blocker: Memory aliasing
  - Out of order processing: 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');
}
```

*Extracted from 213 lab submissions, Fall 1998*
Performance

- Time quadruples when double string length
- Quadratic performance
Why is That?

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

■ String length is called in every iteration!
    ▪ And strlen is $O(n)$, so lower is $O(n^2)$

/* 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/precomputation

```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');
}

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');
}
Performance

- Lower2: Time doubles when double string length
- Linear performance
Optimization Blocker: Procedure Calls

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

- Compiler usually treats procedure call as a black box that cannot be analyzed
  - Consequence: conservative in optimizations

- Remedies:
  - Inline the function if possible
  - 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

- **Program optimization**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - **Optimization blocker: Memory aliasing**
  - Out of order processing: Instruction level parallelism
Optimization Blocker: Memory Aliasing

/* Sums rows of n x n matrix a and stores 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] (= memory access) on every iteration
- Why couldn’t compiler optimize this away?
Reason

- If memory is accessed, compiler assumes the possibility of side effects

Example:

```c
/* Sums rows of n x n matrix a and stores 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);

Value of B:

<table>
<thead>
<tr>
<th>i</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>[4, 8, 16]</td>
</tr>
<tr>
<td>i = 0</td>
<td>[3, 8, 16]</td>
</tr>
<tr>
<td>i = 1</td>
<td>[3, 22, 16]</td>
</tr>
<tr>
<td>i = 2</td>
<td>[3, 22, 224]</td>
</tr>
</tbody>
</table>
Removing Aliasing

/* Sums rows of n x n matrix a and stores 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

- **Scalar replacement:**
  - Copy array elements that are reused into temporary variables
  - Assumes no memory aliasing (otherwise possibly incorrect)
Unaliased Version When Aliasing Happens

```c
/* 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;
    }
}
```

double A[9] =
{ 0,  1,  2,
  4,  8, 16},
32, 64, 128};
sum_rows1(A, B, 3);

Value of B:
init: [4, 8, 16]
i = 0: [3, 8, 16]
i = 1: [3, 27, 16]
i = 2: [3, 27, 224]

- Aliasing still creates interference
- Result different than before
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: \( t1 = a[i] \), \( t2 = b[i+1] \), ....
  - Compute: \( t4 = t1 \ast t2 \); ....
  - Store: \( a[i] = t12 \), \( b[i+1] = t7 \), ...
More Difficult Example

- Matrix multiplication: \( C = A \times B + C \)

```c

c = (double *) calloc(sizeof(double), n*n);

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

- Which array elements are reused?
- All of them! *But how to take advantage?*
Step 1: Blocking (Here: 2 x 2)

- Blocking, also called tiling = partial unrolling + loop exchange
  - Assumes associativity (= compiler will never do it)

```c
double *c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=2)
        for (j = 0; j < n; j+=2)
            for (k = 0; k < n; k+=2)
                for (i1 = i; i1 < i+2; i1++)
                    for (j1 = j; j1 < j+2; j1++)
                        for (k1 = k; k1 < k+2; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```
Step 2: Unrolling Inner Loops

```c
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=2)
        for (j = 0; j < n; j+=2)
            for (k = 0; k < n; k+=2)
                <body>
                \[ c[i*n + j] = a[i*n + k]*b[k*n + j] + a[i*n + k+1]*b[(k+1)*n + j] + c[i*n + j] \]
                \[ c[(i+1)*n + j] = a[(i+1)*n + k]*b[k*n + j] + a[(i+1)*n + k+1]*b[(k+1)*n + j] + c[(i+1)*n + j] \]
                \[ c[i*n + (j+1)] = a[i*n + k]*b[k*n + (j+1)] + a[i*n + k+1]*b[(k+1)*n + (j+1)] + c[i*n + (j+1)] \]
                \[ c[(i+1)*n + (j+1)] = a[(i+1)*n + k]*b[k*n + (j+1)] + a[(i+1)*n + k+1]*b[(k+1)*n + (j+1)] + c[(i+1)*n + (j+1)] \]
            <body>

- Every array element \( a[\ldots], b[\ldots], c[\ldots] \) used twice
- Now scalar replacement can be applied
Today

- **Program optimization**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - **Out of order processing: Instruction level parallelism**
Example: Compute Factorials

```c
int rfact(int n)
{
    if (n <= 1)
        return 1;
    return n * rfact(n-1);
}
```

```c
int fact(int n)
{
    int i;
    int result = 1;
    for (i = n; i > 0; i--)
        result = result * i;
    return result;
}
```

- **Machines**
  - Intel Pentium 4 Nocona, 3.2 GHz
    - Fish Machines
  - Intel Core 2, 2.7 GHz

- **Compiler Versions**
  - GCC 3.4.2 (current on Fish machines)

<table>
<thead>
<tr>
<th>Cycles per element (or per mult)</th>
<th>Machine</th>
<th>Nocona</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfact</td>
<td>15.5</td>
<td>6.0</td>
<td></td>
</tr>
<tr>
<td>fact</td>
<td>10.0</td>
<td>3.0</td>
<td></td>
</tr>
</tbody>
</table>

Something changed from Pentium 4 to Core: Details later
Optimization 1: Loop Unrolling

- Compute more values per iteration
- Does not help here
- Why? Branch prediction – details later

```c
int fact_u3a(int n)
{
    int i;
    int result = 1;
    
    for (i = n; i >= 3; i-=3) {
        result =
            result * i * (i-1) * (i-2);
    }
    for (; i > 0; i--)
        result *= i;
    return result;
}
```

<table>
<thead>
<tr>
<th></th>
<th>Nocona</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfact</td>
<td>15.5</td>
<td>6.0</td>
</tr>
<tr>
<td>fact</td>
<td>10.0</td>
<td>3.0</td>
</tr>
<tr>
<td>fact_u3a</td>
<td>10.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Optimization 2: Multiple Accumulators

```c
int fact_u3b(int n)
{
    int i;
    int result0 = 1;
    int result1 = 1;
    int result2 = 1;

    for (i = n; i >= 3; i-=3) {
        result0 *= i;
        result1 *= (i-1);
        result2 *= (i-2);
    }
    for (; i > 0; i--)
        result0 *= i;
    return result0 * result1 * result2;
}
```

- That seems to help. Can one get even faster?
- Explanation: instruction level parallelism – details later

<table>
<thead>
<tr>
<th>Cycles per element (or per mult)</th>
<th>Nocona</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfact</td>
<td>15.5</td>
<td>6.0</td>
</tr>
<tr>
<td>fact</td>
<td>10.0</td>
<td>3.0</td>
</tr>
<tr>
<td>fact_u3a</td>
<td>10.0</td>
<td>3.0</td>
</tr>
<tr>
<td>fact_u3b</td>
<td>3.3</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Modern CPU Design

**Instruction Control**
- Instruction Cache
  - Instructions
  - Address
- Instruction Decode
  - Instructions
  - Operations
- Fetch Control
  - Instructions
  - Address
- Retirement Unit
  - Register File
  - Prediction OK?
  - Register Updates
- Functional Units
  - Integer/Branch
  - General Integer
  - FP Add
  - FP Mult/Div
  - Load
  - Store
  - Data Cache
  - Addr.
  - Data
  - Operation Results

**Execution**
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
Pentium 4 Nocona 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/SSE3 unit
  1 FP move (does all conversions)

- 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>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>36/106</td>
<td>36/106</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>32/46</td>
<td>32/46</td>
</tr>
</tbody>
</table>
Latency versus Throughput

- Last slide: latency cycles/issue
  - Integer Multiply
    - Latency: 10 cycles
    - Cycles per Issue: 1 cycle

  ![Diagram](image)

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

- Major problem for fast execution: **Keep pipelines filled**
Hard Bounds

- Latency and throughput of instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>36/106</td>
<td>36/106</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>32/46</td>
<td>32/46</td>
</tr>
</tbody>
</table>

- How many cycles at least if
  - Function requires n int mults?
  - Function requires n float adds?
  - Function requires n float ops (adds and mults)?
Performance in Numerical Computing

- Numerical computing = computing dominated by floating point operations
- Example: Matrix multiplication

- Performance measure:
  Floating point operations per second (flop/s)
  - Counting only floating point adds and mults
  - Higher is better
  - Like inverse runtime

- Theoretical scalar (no vector SSE) peak performance on fish machines?
  - 3.2 Gflop/s = 3200 Mflop/s. Why?
# Nocona vs. Core 2

## Nocona (3.2 GHz) (Saltwater fish machines)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>36/106</td>
<td>36/106</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>32/46</td>
<td>32/46</td>
</tr>
</tbody>
</table>

## Core 2 (2.7 GHz) (Recent Intel microprocessors)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>18/50</td>
<td>18/50</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>18/32</td>
<td>18/32</td>
</tr>
</tbody>
</table>
Instruction Control

- **Grabs instruction bytes from memory**
  - Based on current PC + predicted targets for predicted branches
  - Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

- **Translates instructions into micro-operations** (for CISC style CPUs)
  - Micro-op = primitive step required to perform instruction
  - Typical instruction requires 1–3 operations

- **Converts register references into tags**
  - Abstract identifier linking destination of one operation with sources of later operations
Translating into Micro-Operations

Goal: Each operation utilizes single functional unit

- Requires: Load, integer arithmetic, store

```plaintext
load 8(%rbx,%rdx,4) ➔ temp1
imulq %rax, temp1 ➔ temp2
store temp2, 8(%rbx,%rdx,4)
```

- Exact form and format of operations is trade secret
Traditional View of Instruction Execution

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq %rax, %rbx</td>
<td># I1</td>
</tr>
<tr>
<td>andq %rbx, %rdx</td>
<td># I2</td>
</tr>
<tr>
<td>mulq %rcx, %rbx</td>
<td># I3</td>
</tr>
<tr>
<td>xorq %rbx, %rdi</td>
<td># I4</td>
</tr>
</tbody>
</table>

- **Imperative View**
  - Registers are fixed storage locations
    - Individual instructions read & write them
  - Instructions must be executed in specified sequence to guarantee proper program behavior
Dataflow View of Instruction Execution

- **Functional View**
  - View each write as creating new instance of value
  - Operations can be performed as soon as operands available
  - No need to execute in original sequence

```plaintext
addq %rax, %rbx  # I1
andq %rbx, %rdx  # I2
mulq %rcx, %rbx  # I3
xorq %rbx, %rdi  # I4
```
Example Computation

```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;
}
```


- **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 operators on vectors or lists
- Length = $n$
- In our case: $CPE = \text{cycles per OP}$ (gives hard lower bound)
- $T = CPE \times n + \text{Overhead}$

![Graph showing linear relationship between cycles and number of elements]

- $vsum1: \text{Slope} = 4.0$
- $vsum2: \text{Slope} = 3.5$
x86-64 Compilation of Combine4

```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;
}
```

### Inner Loop (Case: Integer Multiply)

L33: # Loop:

- `movl (%eax,%edx,4), %ebx` # temp = d[i]
- `incl %edx` # i++
- `imull %ebx, %ecx` # x *= temp
- `cmpl %esi, %edx` # i:length
- `jl L33` # if < goto Loop

### Cycles per element (or per OP)

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</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! Hence,**
  - Performance: determined by latency of OP!

---

**Cycles per element (or per OP)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
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>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Helps integer sum
- Others don’t improve. *Why?*
  - Still sequential dependency

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

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

```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;
}
```
Effect of Reassociation

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2-ra</td>
<td>1.56</td>
<td>5.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency
  
  \[
x = x \text{ OP } (d[i] \text{ OP } d[i+1]);
  \]

- Why is that? (next slide)
Reassociated Computation

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:
  \[\text{CPE} = D/2\]
- Measured CPE slightly worse for FP

\[x = x \text{ OP } (d[i] \text{ OP } d[i+1]);\]
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>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2-ra</td>
<td>1.56</td>
<td>5.0</td>
</tr>
<tr>
<td>unroll2-sa</td>
<td>1.50</td>
<td>5.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
x_0 &= x_0 \text{ OP } d[i]; \\
x_1 &= x_1 \text{ OP } d[i+1];
\end{align*}
\]
Separate Accumulators

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

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

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

- Case
  - Intel Nocona (Saltwater fish machines)
  - FP Multiplication
  - Theoretical Limit: 2.00

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K 1 2 3 4 6 8 10 12</td>
</tr>
<tr>
<td></td>
<td>1 7.00 7.00 7.01 7.00</td>
</tr>
<tr>
<td></td>
<td>2 3.50 3.50 3.50</td>
</tr>
<tr>
<td></td>
<td>3 2.34</td>
</tr>
<tr>
<td></td>
<td>4 2.01 2.00</td>
</tr>
<tr>
<td></td>
<td>6 2.00</td>
</tr>
<tr>
<td></td>
<td>8 2.01</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

Accumulators
Unrolling & Accumulating: Intel FP +

- **Case**
  - Intel Nocona (Saltwater fish machines)
  - FP Addition
  - Theoretical Limit: 2.00

<table>
<thead>
<tr>
<th>FP +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K 1 2 3 4 6 8 10 12</td>
</tr>
<tr>
<td></td>
<td>5.00 5.00 5.02 5.00</td>
</tr>
<tr>
<td>1</td>
<td>2.50 2.51 2.51</td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>3</td>
<td>2.01 2.00</td>
</tr>
<tr>
<td>4</td>
<td>2.00</td>
</tr>
<tr>
<td>6</td>
<td>2.00 1.99</td>
</tr>
<tr>
<td>8</td>
<td>2.01</td>
</tr>
<tr>
<td>10</td>
<td>2.00</td>
</tr>
<tr>
<td>12</td>
<td>2.00</td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Intel Int *

- Case
  - Intel Nocona (Saltwater fish machines)
  - Integer Multiplication
  - Theoretical Limit: 1.00

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

**Case**
- Intel Nocona (Saltwater fish machines)
- Integer addition
- Theoretical Limit: 1.00 (unrolling enough)

<table>
<thead>
<tr>
<th>Int +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K 1 2 3 4 6 8 10 12</td>
</tr>
<tr>
<td>1</td>
<td>2.20 1.50 1.10 1.03</td>
</tr>
<tr>
<td>2</td>
<td>1.50 1.10 1.03</td>
</tr>
<tr>
<td>3</td>
<td>1.34</td>
</tr>
<tr>
<td>4</td>
<td>1.09 1.03</td>
</tr>
<tr>
<td>6</td>
<td>1.01</td>
</tr>
<tr>
<td>8</td>
<td>1.01</td>
</tr>
<tr>
<td>10</td>
<td>1.04</td>
</tr>
<tr>
<td>12</td>
<td>1.11</td>
</tr>
</tbody>
</table>
### FP *:

Nocona versus Core 2

#### Machines
- Intel Nocona
  - 3.2 GHz
- Intel Core 2
  - 2.7 GHz

#### Performance
- Core 2 lower latency & fully pipelined (1 cycle/issue)

<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>7.00</td>
</tr>
<tr>
<td>2</td>
<td>3.50</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>

<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>4.00</td>
</tr>
<tr>
<td>2</td>
<td>2.00</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>
### Nocona vs. Core 2 Int *

#### Performance
- Newer version of GCC does reassociation
- Why for int’s and not for float’s?

<table>
<thead>
<tr>
<th>Int *</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>10.00</td>
</tr>
<tr>
<td>2</td>
<td>5.00</td>
</tr>
<tr>
<td>3</td>
<td>3.33</td>
</tr>
<tr>
<td>4</td>
<td>2.50</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>

<table>
<thead>
<tr>
<th>Int *</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>3.00</td>
</tr>
<tr>
<td>2</td>
<td>1.50</td>
</tr>
<tr>
<td>3</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>1.00</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>
### Intel vs. AMD FP

#### Machines
- Intel Nocona
  - 3.2 GHz
- AMD Opteron
  - 2.0 GHz

#### Performance
- AMD lower latency & better pipelining
- But slower clock rate

### Unrolling Factor L

<table>
<thead>
<tr>
<th>K</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.00</td>
<td>7.00</td>
<td>7.01</td>
<td>7.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.50</td>
<td>3.50</td>
<td>3.50</td>
<td>2.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.34</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2.01</td>
<td>2.00</td>
<td>2.00</td>
<td>2.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2.00</td>
<td>2.01</td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2.01</td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>K</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td>4.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.34</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Intel vs. AMD

**Int *\**

#### Performance

- AMD multiplier much lower latency
- Can get high performance with less work
- Doesn’t achieve as good an optimum

<table>
<thead>
<tr>
<th>K</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
<td>10.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5.00</td>
<td>5.01</td>
<td>5.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3.33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2.50</td>
<td>2.51</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.67</td>
<td>1.67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.09</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.14</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>K</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.00</td>
<td>3.00</td>
<td>3.00</td>
<td>3.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.33</td>
<td>2.0</td>
<td>1.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.75</td>
<td>1.38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.50</td>
<td>1.50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.33</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Intel vs. AMD Int

#### Performance
- AMD gets below 1.0
- Even just with unrolling

#### Explanation
- Both Intel & AMD can “double pump” integer units
- Only AMD can load two elements / cycle

<table>
<thead>
<tr>
<th>K</th>
<th>Int +</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.20</td>
<td>1.50</td>
<td>1.10</td>
<td>1.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.50</td>
<td>1.10</td>
<td>1.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1.34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.09</td>
<td></td>
<td>1.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>1.01</td>
<td></td>
<td>1.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>1.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>K</th>
<th>Int +</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.32</td>
<td>1.50</td>
<td>0.75</td>
<td>0.63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.50</td>
<td>0.83</td>
<td>0.63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td></td>
<td>0.63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>0.83</td>
<td></td>
<td>0.67</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>0.63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.85</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Can We Go Faster?

- Yes, SSE!
  - But not in this class
  - 18-645
Summary

- **Optimization comes from many directions:**
  - Algorithm design: huge potential
  - Optimizing compilers: effective but conservative
  - Manual tuning: many techniques
  - Parallel computation: we'll talk about this later

- **Understanding processors, memory, and compilers**