15-213/18-243, Fall 2009

Carnegie Mellon

Introduction to Computer Systems

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

Instructors:
Roger B. Dannenberg and Greg Ganger

Carnegie Mellon

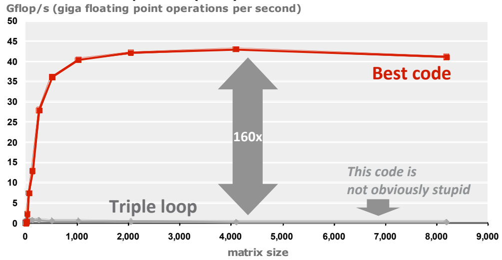
Today

- **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

Carnegie Mellon

Example Matrix Multiplication

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

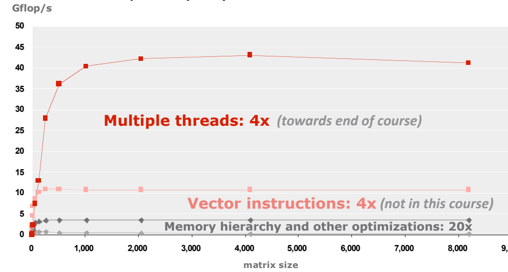


- Standard desktop computer, compiler, using optimization flags
- Both implementations have *exactly* the same operations count ($2n^3$)
- *What is going on?*

Carnegie Mellon

MMM Plot: Analysis

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



- 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*

Carnegie Mellon

Harsh Reality

- *There's more to runtime performance than asymptotic complexity*
- *One can easily loose 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)

Carnegie Mellon

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

-O

- 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

```
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

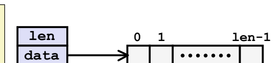
- *If in doubt, the compiler is conservative*
- Operate under fundamental constraints
  - Must not change program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.
- Behavior that may be obvious to the 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

```
/* 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?

```
/* 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;
}
```

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**

```
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;
    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];
}
```

```
    long j;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
        *rowp++ = 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:                                    # loop:
        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 < goot loop
.L7:                                    # done:
        rep ; ret                       #  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*x` → `x << 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**

```
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*

```
/* 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;
```

```
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 mult: i*n*

```
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;
```

```
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

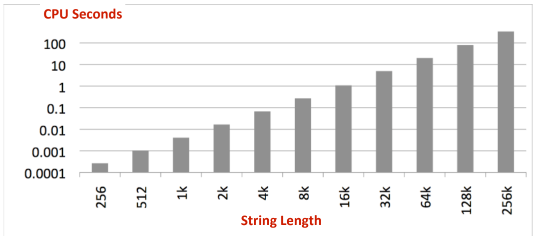
- **Procedure to convert string to lower case**

```
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

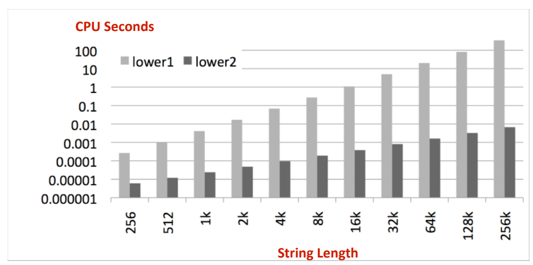
```
void lower(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
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');
}
```

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

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

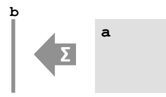
```
int lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```


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:

```
/* 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};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of B:

```
init:  [4,  8,  16]
i = 0: [3,  8,  16]
i = 1: [3, 22,  16]
i = 2: [3, 22, 224]
```


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

```
/* 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};

double B[3] = A+3;

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

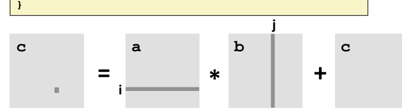
- Memory aliasing: Two different memory references write to the same location
- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- Hard to analyze = compiler cannot figure it out
  - Hence is conservative
- Solution: Scalar replacement in innermost loop
  - Copy memory variables that are reused into local variables
  - Basic scheme:
    - **Load:** t1 = a[i], t2 = b[i+1], ….
    - **Compute:** t4 = t1 * t2; ….
    - **Store:** a[i] = t12, b[i+1] = t7, …

More Difficult Example

- Matrix multiplication: C = A*B + 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];
}
```



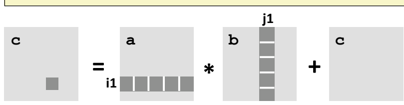
- Which array elements are reused?
- All of them! *But how to take advantage?*

Step 1: Blocking (Here: 2 x 2)

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

```
c = (double *) 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 = (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)
                <body>
}
```

```
<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)]
```

- **Every array element `a[…]`, `b[…]`, `c[…]` 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

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

```
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)

Cycles per element (or per mult)

Machine	Nocona	Core
rfact	15.5	6.0
fact	10.0	3.0

Something changed from Pentium 4 to Core: Details later

Optimization 1: Loop Unrolling

```
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;
}
```

Cycles per element (or per mult)

Machine	Nocona	Core 2
rfact	15.5	6.0
fact	10.0	3.0
fact_u3a	10.0	3.0

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

Optimization 2: Multiple Accumulators

```
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;
}
```

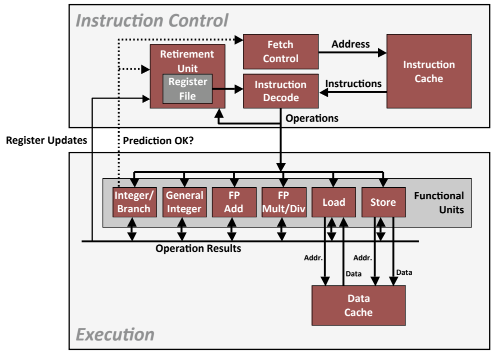
Cycles per element (or per mult)

Machine	Nocona	Core 2
rfact	15.5	6.0
fact	10.0	3.0
fact_u3a	10.0	3.0
fact_u3b	3.3	1.0

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

Carnegie Mellon

Modern CPU Design



43

Carnegie Mellon

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**

44

Carnegie Mellon

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**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	5	1
Integer Multiply	10	1
Integer/Long Divide	**36/106**	**36/106**
Single/Double FP Multiply	7	2
Single/Double FP Add	5	2
Single/Double FP Divide	**32/46**	**32/46**

45

Carnegie Mellon

Latency versus Throughput

- **Last slide:**

	latency	*cycles/issue*
Integer Multiply	10	1

- **Consequence:**
  - How fast can 10 independent int mults be executed?
    `t1 = t2*t3; t4 = t5*t6; …`
  - How fast can 10 sequentially dependent int mults be executed?
    `t1 = t2*t3; t4 = t5*t1; t6 = t7*t4; …`
- **Major problem for fast execution: Keep pipelines filled**

46

Carnegie Mellon

Hard Bounds

- **Latency and throughput of instructions**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	5	1
Integer Multiply	10	1
Integer/Long Divide	**36/106**	**36/106**
Single/Double FP Multiply	7	2
Single/Double FP Add	5	2
Single/Double FP Divide	**32/46**	**32/46**

- **How many cycles at least if**
  - Function requires n int mults?
  - Function requires n float adds?
  - Function requires n float ops (adds and mults)?

47

Carnegie Mellon

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?***

48

Nocona vs. Core 2

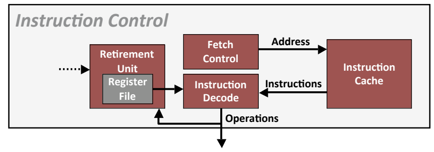
- **Nocona (3.2 GHz) (Saltwater fish machines)**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	5	1
Integer Multiply	10	1
Integer/Long Divide	36/106	36/106
Single/Double FP Multiply	**7**	**2**
Single/Double FP Add	**5**	**2**
Single/Double FP Divide	32/46	32/46

- **Core 2 (2.7 GHz) (Recent Intel microprocessors)**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	5	1
Integer Multiply	3	1
Integer/Long Divide	18/50	18/50
Single/Double FP Multiply	**4/5**	**1**
Single/Double FP Add	**3**	**1**
Single/Double FP Divide	18/32	18/32

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

```
imulq %rax, 8(%rbx,%rdx,4)
```

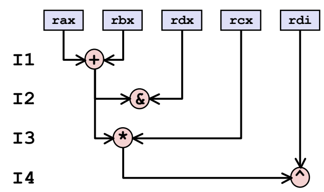
- **Goal: Each operation utilizes single functional unit**
  - Requires: Load, integer arithmetic, store

```
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

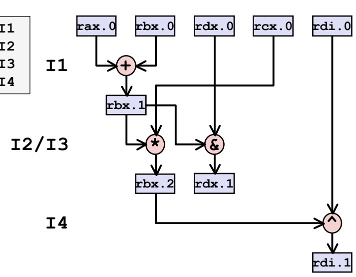
```
addq %rax, %rbx  # I1
andq %rbx, %rdx  # I2
mulq %rcx, %rbx  # I3
xorq %rbx, %rdi  # I4
```



- **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

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



- **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

Example Computation

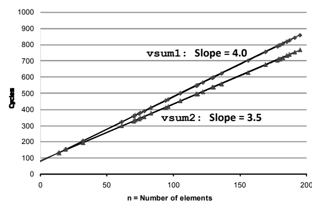
```
void combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

`d[1] OP d[2] OP d[3] OP … OP d[length-1]`

- **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 = cycles per OP (gives hard lower bound)
- T = CPE*n + Overhead



x86-64 Compilation of Combine4

```
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;
}
```

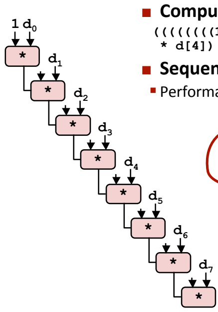
Cycles per element (or per OP)

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

- 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
```


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! Hence,
  - Performance: determined by latency of OP!

Cycles per element (or per OP)

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

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

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

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

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


Loop Unrolling with Reassociation

```
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

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
bound	1.0	1.0	2.0	2.0

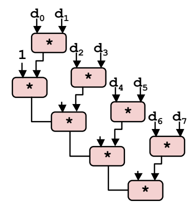
- **Nearly 2x speedup for Int *, FP +, FP ***
  - Reason: Breaks sequential dependency

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

  - Why is that? (next slide)

Reassociated Computation

```
x = x 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

Loop Unrolling with Separate Accumulators

```
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

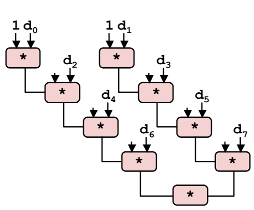
Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
unroll2-sa	1.50	5.0	2.5	3.5
bound	1.0	1.0	2.0	2.0

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

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


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: **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

Carnegie Mellon

Unrolling & Accumulating: Intel FP *

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

Accumulators

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	7.00	7.00		7.01		7.00		
2		3.50		3.50		3.50		
3			2.34					
4				2.01		2.00		
6					2.00			2.01
8						2.01		
10							2.00	
12								2.00

Carnegie Mellon

Unrolling & Accumulating: Intel FP +

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

FP +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.00	5.00		5.02		5.00		
2		2.50		2.51		2.51		
3			2.00					
4				2.01		2.00		
6					2.00			1.99
8						2.01		
10							2.00	
12								2.00

Carnegie Mellon

Unrolling & Accumulating: Intel Int *

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

Int *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	10.00	10.00		10.00		10.01		
2		5.00		5.01		5.00		
3			3.33					
4				2.50		2.51		
6					1.67			1.67
8						1.25		
10							1.09	
12								1.14

Carnegie Mellon

Unrolling & Accumulating: Intel Int +

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

Int +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.20	1.50		1.10		1.03		
2		1.50		1.10		1.03		
3			1.34					
4				1.09		1.03		
6					1.01			1.01
8						1.03		
10							1.04	
12								1.11

Carnegie Mellon

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	7.00	7.00		7.01		7.00		
2		3.50		3.50		3.50		
3			2.34					
4				2.01		2.00		
6					2.00			2.01
8						2.01		
10							2.00	
12								2.00

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	4.00	4.00		4.00		4.01		
2		2.00		2.00		2.00		
3			1.34					
4				1.00		1.00		
6					1.00			1.00
8						1.00		
10							1.00	
12								1.00

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)

Carnegie Mellon

Int *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	10.00	10.00		10.00		10.01		
2		5.00		5.01		5.00		
3			3.33					
4				2.50		2.51		
6					1.67			1.67
8						1.25		
10							1.09	
12								1.14

Int *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	3.00	1.50		1.00		1.00		
2		1.50		1.00		1.00		
3			1.00					
4				1.00		1.00		
6					1.00			1.00
8						1.00		
10							1.00	
12								1.33

Nocona vs. Core 2 Int *

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

Intel vs. AMD FP *

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	7.00	7.00		7.01		7.00		
2		3.50		3.50		3.50		
3			2.34					
4				2.01		2.00		
6					2.00			2.01
8						2.01		
10							2.00	
12								2.00

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	4.00	4.00		4.00		4.01		
2		2.00		2.00		2.00		
3			1.34					
4				1.00		1.00		
6					1.00			1.00
8						1.00		
10							1.00	
12								1.00

- **Machines**
  - Intel Nocona
    - 3.2 GHz
  - AMD Opteron
    - 2.0 GHz
- **Performance**
  - AMD lower latency & better pipelining
  - But slower clock rate

Intel vs. AMD Int *

Int *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	10.00	10.00		10.00		10.01		
2		5.00		5.01		5.00		
3			3.33					
4				2.50		2.51		
6					1.67			1.67
8						1.25		
10							1.09	
12								1.14

Int *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	3.00	3.00		3.00		3.00		
2		2.33		2.0		1.35		
3			2.00					
4				1.75		1.38		
6					1.50			1.50
8						1.75		
10							1.30	
12								1.33

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

Intel vs. AMD Int +

Int +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.20	1.50		1.10		1.03		
2		1.50		1.10		1.03		
3			1.34					
4				1.09		1.03		
6					1.01			1.01
8						1.03		
10							1.04	
12								1.11

Int +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.32	1.50		0.75		0.63		
2		1.50		0.83		0.63		
3			1.00					
4				1.00		0.63		
6					0.83			0.67
8						0.63		
10							0.60	
12								0.85

- **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

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**