How to Write Fast Numerical Code
Spring 2017
*Lecture: Compiler Limitations*

**Instructor:** Markus Püschel
**TA:** Alen Stojanov, Georg Ofenbeck, Gagandeep Singh

---

**Last Time: ILP**

- Latency/throughput (Pentium 4 fp mult: 7/2)

```
        1 d0
      /   
  *     d2
   /   
*     d3
    /   
*     d4
     /   
*     d5
      /   
  *     d6
   /   
  *     d7

Twice as fast
```

```
Last Time: How Many Accumulators?

Latency: 7 cycles

Those have to be independent

Based on this insight: \[ K = \# \text{accumulators} = \text{ceil}(\text{latency}/\text{cycles per issue}) \]

Compiler Limitations

- Associativity law does not hold for floats: illegal transformation
- No good way of handling choices (e.g., number of accumulators)
- More examples of limitations today
Today

- Optimizing compilers and optimization blockers
  - Overview
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Summary


*Part of these slides are adapted from the course associated with this book*

---

Optimizing Compilers

- Always use optimization flags:
  - gcc: *default is no optimization* (-O0)!
  - icc: some optimization is turned on

- Good choices for gcc/icc: -O2, -O3, -march=xxx, -mAVX, -m64
  - Read in manual what they do
  - Understand the differences

- Try different flags and maybe different compilers
Example (On Core 2 Duo)

```c
double a[4][4];
double b[4][4];
double c[4][4];

/* Multiply 4 x 4 matrices c = a*b + c */
void mmm(double *a, double *b, double *c) {
    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

---

<table>
<thead>
<tr>
<th>MMX: Multimedia extension</th>
<th>Intel x86</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE: Streaming SIMD extension</td>
<td>x86-16</td>
<td>8086</td>
</tr>
<tr>
<td></td>
<td></td>
<td>286</td>
</tr>
<tr>
<td>AVX: Advanced vector extensions</td>
<td>x86-32</td>
<td>386</td>
</tr>
<tr>
<td></td>
<td></td>
<td>486</td>
</tr>
<tr>
<td></td>
<td>MMX</td>
<td>Pentium</td>
</tr>
<tr>
<td></td>
<td>SSE</td>
<td>Pentium MMX</td>
</tr>
<tr>
<td></td>
<td>SSE2</td>
<td>Pentium III</td>
</tr>
<tr>
<td></td>
<td>SSE3</td>
<td>Pentium 4</td>
</tr>
<tr>
<td></td>
<td>x86-64 / em64t</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td></td>
<td>SSE4</td>
<td>Core 2 Duo</td>
</tr>
<tr>
<td></td>
<td>AVX</td>
<td>Penryn</td>
</tr>
<tr>
<td></td>
<td>AVX2</td>
<td>Core i7 (Nehalem)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sandy Bridge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Haswell</td>
</tr>
</tbody>
</table>

Prevents use of SSE

Use architecture flags

---

© Markus Püschel  
Computer Science

How to write fast numerical code  
Spring 2017
Optimizing Compilers

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

- Compilers are **not good** at: algorithmic restructuring
  - for example to increase ILP, locality, etc.
  - cannot deal with choices

- 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

- **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
  - Not good at evaluating or dealing with choices
Organization

- Instruction level parallelism (ILP): an example
- Optimizing compilers and optimization blockers
  - Overview
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
- Summary

Code Motion

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

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

- Compiler is likely to do
Strength Reduction

- Replace costly operation with simpler one
- Example: Shift/add instead of multiply or divide \(16 \times x \rightarrow x \ll 4\)
  - Utility machine dependent
- Example: Recognize sequence of products

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

- Compiler is likely to do

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;
```

- In simple cases compiler is likely to do

```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;
```
Organization

- Instruction level parallelism (ILP): an example
- Optimizing compilers and optimization blockers
  - Overview
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Summary

Example: Data Type for Vectors

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

/* retrieve vector element and store at val */
int get_vec_element(vec *v, int 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 t;
    for (i = 0; i < n; i++) {
        get_vec_element(v, i, &t);
        *res += t;
    }
    return res;
}
```

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

Potential big performance loss

Removing Procedure Call

```c
/* sum elements of vector */
double sum_elements_opt(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 in superfast numerical library functions

- Watch your innermost loop!
- Get a feel for overhead versus actual computation being performed

Further Inspection of the Example

```
vector.c    // vector data type
sum.c       // sum
sum_opt.c   // optimized sum
main.c      // timing

$(CC) -c -o vector.o vector.c
$(CC) -c -o sum.o sum.c
$(CC) -c -o main.o main.c
$(CC) -o vector vector.o sum.o main.o

Xeon: 7.2 cycles/add
Atom: 28 cycles/add

$(CC) -c -o vector.o vector.c
$(CC) -c -o sum_opt.o sum_opt.c
$(CC) -c -o main.o main.c
$(CC) -o vector vector.o sum_opt.o main.o

Xeon: 2.4 cycles/add
Atom: 6 cycles/add

$(CC) -c -o vector.o vector.c sum.c
$(CC) -c -o main.o main.c
$(CC) -o vector vector.o main.o

Xeon: 2.4 cycles/add
Atom: 6 cycles/add
```

What's happening here?
Function Inlining

- Compilers may be able to do function inlining
  - Replace function call with body of function
  - Usually requires that source code is compiled together

Enables other optimizations

**Problem:** performance libraries distributed as binary

```c
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double t;
    for (i = 0; i < n; i++) {
        get_vec_element(v, i, &t);
        *res += t;
    }
    return res;
}
```

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

Organization

- Instruction level parallelism (ILP): an example

- Optimizing compilers and optimization blockers
  - Overview
    - Code motion
    - Strength reduction
    - Sharing of common subexpressions
    - Removing unnecessary procedure calls
    - **Optimization blocker: Procedure calls**
    - Optimization blocker: Memory aliasing
    - Summary
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');
}
```

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

Optimization Blocker #2: Procedure Calls

- Move call to strlen outside of loop
- Form of code motion/precomputation

```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');
}
```
Optimization Blocker: Procedure Calls

- Why couldn’t compiler move `strlen` out of inner loop?
  - Procedure may have side effects

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

- In this case the compiler may actually do it if `strlen` is recognized as built-in function whose properties are known

---

Organization

- Instruction level parallelism (ILP): an example
- Optimizing compilers and optimization blockers
  - Overview
    - Removing unnecessary procedure calls
    - Code motion
    - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
  - Optimization blocker: Procedure calls
  - *Optimization blocker: Memory aliasing*
  - Summary
Optimization Blocker: Memory Aliasing

/* Sums rows of n x n matrix a and stores in vector b */
void sum_rows1(double *a, double *b, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}

- Code updates $b[i]$ (= memory access) on every iteration

Optimization Blocker: Memory Aliasing

/* Sums rows of n x n matrix a and stores in vector b */
void sum_rows2(double *a, double *b, int n) {
    int 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;
    }
}

Does compiler optimize this?
No!
Why?
Reason: Possible Memory Aliasing

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

```c
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, 22, 16]`
- `i = 2: [3, 22, 224]`

Removing Aliasing

```c
/* Sums rows of n x n matrix a and stores in vector b */
void sum_rows2(double *a, double *b, int n) {
    int 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;
    }
}
```

- Scalar replacement:
  - Copy array elements *that are reused* into temporary variables
  - Perform computation on those variables
  - Enables register allocation and instruction scheduling
  - Assumes no memory aliasing (otherwise incorrect)
**Optimization Blocker: Memory Aliasing**

- **Memoryaliasing:**
  - 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:* \( t_1 = a[i], t_2 = b[i+1], \ldots \)
    - *Compute:* \( t_4 = t_1 \times t_2, \ldots \)
    - *Store:* \( a[i] = t_1^2, b[i+1] = t_7, \ldots \)

**Example: MMM**

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

```c
void mmm(double const * A, double const * B, double * C, size_t N) {
    for (size_t k6 = 0; k6 < N; k6++)
        for (size_t i5 = 0; i5 < N; i5++)
            for (size_t j7 = 0; j7 < N; j7++)
}
```

```c
void mmm(double const * A, double const * B, double * C, size_t N) {
    for (size_t i21 = 0; i21 < N; i21+=2)
        for (size_t j23 = 0; j23 < N; j23+=2)
            for (size_t k22 = 0; k22 < N; k22+=2)
                for (size_t kk25 = 0; kk25 < 2; kk25++)
                    for (size_t ii24 = 0; ii24 < 2; ii24++)
                        for (size_t jj26 = 0; jj26 < 2; jj26++)
                            C[N*i21 + N*i24 + j23 + jj26] = C[N*i21 + N*i24 + j23 + jj26] +
}
```

- **tile each loop (= blocking MMM)**
- **unroll inner three loops**
how the reuse becomes apparent (every element used twice)

```c
void mmm(double const * A, double const * B, double * C, size_t N) {
    for (size_t i21 = 0; i21 < N; i21+=2 )
        for( size_t j23 = 0; j23 < N; j23+=2 ) {
        }
}
```

unroll inner three loops

Now the reuse becomes apparent (every element used twice)

```c
void mmm(double const * A, double const * B, double * C, size_t N) {
    for (size_t i21 = 0; i21 < N; i21+=2 )
        for( size_t j23 = 0; j23 < N; j23+=2 ) {
        }
}
```

unroll inner three loops

scalar replacement

Now the reuse becomes apparent (every element used twice)
```c
void mm(double const * A, double const * B, double * C, size_t N) {
    for (size_t i21 = 0; i21 < N; i21+=2 )
        for (size_t j23 = 0; j23 < N; j23+=2 )
            for (size_t k22 = 0; k22 < N; k22+=2 ) {
                double t0_0, t0_1, t0_2, t0_3, t0_4, t0_5, t0_6, t0_7, t0_8, t0_9, t0_10, t0_11, t0_12;
                t0_7 = A[N*i21 + k22];
                t0_6 = A[N*i21 + k22 + 1];
                t0_5 = A[N*i21 + N + k22];
                t0_4 = A[N*i21 + N + k22 + 1];
                t0_3 = B[j23 + N*k22];
                t0_2 = B[j23 + N*k22 + 1];
                t0_1 = B[j23 + N*k22 + N];
                t0_0 = B[j23 + N*k22 + N + 1];
                t0_8 = C[N*i21 + j23];
                t0_9 = C[N*i21 + j23 + 1];
                t0_10 = C[N*i21 + N + j23];
                t0_11 = C[N*i21 + N + j23 + 1];
                t0_8 = t0_8 + t0_12;
                t0_9 = t0_9 + t0_12;
                t0_10 = t0_10 + t0_12;
                t0_11 = t0_11 + t0_12;
                C[N*i21 + j23] = t0_8;
                C[N*i21 + j23 + 1] = t0_9;
                C[N*i21 + N + j23] = t0_10;
                C[N*i21 + N + j23 + 1] = t0_11;
            }
}
```

### Effect on Runtime?

<table>
<thead>
<tr>
<th></th>
<th>N = 4</th>
<th>N = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triple loop</td>
<td>202</td>
<td>2.3M</td>
</tr>
</tbody>
</table>

Intel Core i7-2600 (Sandy Bridge)
 compiler: icc 12.1
 flags: -O3 -no-vec -no-ip -no-ip
Effect on Runtime?

Intel Core i7-2600 (Sandy Bridge)
compiler: icc 12.1
flags: -O3 -no-vec -no-ip -no-ip

<table>
<thead>
<tr>
<th></th>
<th>N = 4</th>
<th>N = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triple loop</td>
<td>202</td>
<td>2.3M</td>
</tr>
<tr>
<td>Six-fold loop</td>
<td>144</td>
<td>2.3M</td>
</tr>
<tr>
<td>+ Inner three unrolled</td>
<td>166</td>
<td>2.4M</td>
</tr>
<tr>
<td>+ scalar replacement</td>
<td>106</td>
<td>1.6M</td>
</tr>
</tbody>
</table>

Can Compiler Remove Aliasing?

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

Potential aliasing: Can compiler do something about it?

Compiler can insert runtime check:

```c
if (a + n < b || b + n < a)
    /* further optimizations may be possible now */
    ...
else
    /* aliased case */
    ...
```
Removing Aliasing With Compiler

- Globally with compiler flag:
  - `-fno-alias`, `/Oa`
  - `-fargument-noalias`, `/Oalias-args` (function arguments only)

- For one loop: pragma

```c
void add(float *a, float *b, int n) {
    #pragma ivdep
    for (i = 0; i < n; i++)
        a[i] = a[i] + b[i];
}
```

- For specific arrays: restrict (needs compiler flag `-restrict`, `/Qrestrict`)

```c
void add(float *restrict a, float *restrict b, int n) {
    for (i = 0; i < n; i++)
        a[i] = a[i] + b[i];
}
```

Organization

- Instruction level parallelism (ILP): an example

- Optimizing compilers and optimization blockers
  - Overview
  - Removing unnecessary procedure calls
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - **Summary**
Summary

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

  ![Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz](image)

  - **What matters besides operation count:**
    - Code style (unnecessary procedure calls, unrolling + ..., reordering, ...)
    - Algorithm structure (instruction level parallelism, locality, ...)
    - Data representation (complicated structs or simple arrays)

Summary: Optimize at Multiple Levels

- **Algorithm:**
  - Evaluate different algorithm choices
  - Restructuring may be needed (ILP, locality)

- **Data representations:**
  - Careful with overhead of complicated data types
  - Best are arrays

- **Procedures:**
  - Careful with overhead
  - They are black boxes for the compiler

- **Loops:**
  - Often need to be restructured (ILP, locality)
  - Unrolling often necessary to enable other optimizations
  - Watch the innermost loop bodies
Numerical Functions

- Use arrays (simple data structure) if possible
- Unroll to some extent
  - To make ILP explicit
  - To enable scalar replacement and hence register allocation for variables that are reused