How to Write Fast Numerical Code

Spring 2017

Lecture: Compiler Limitations

Instructor: Markus Püschel

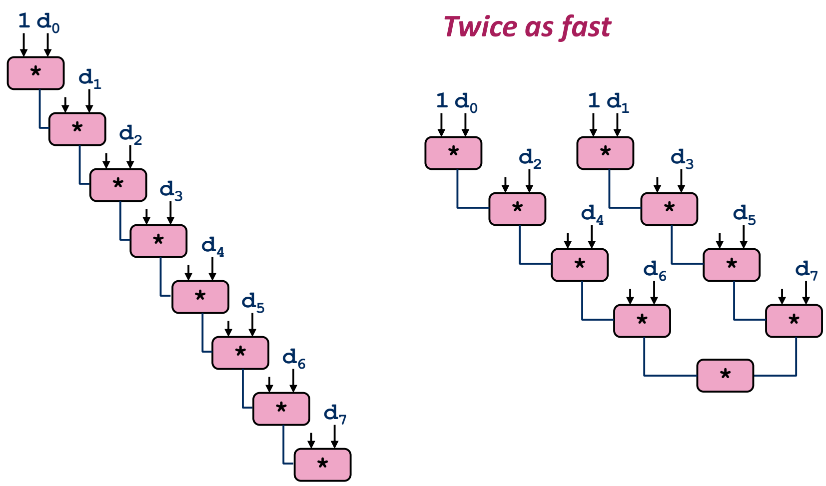
TA: Alen Stojanov, Georg Ofenbeck, Gagandeep Singh

ETH

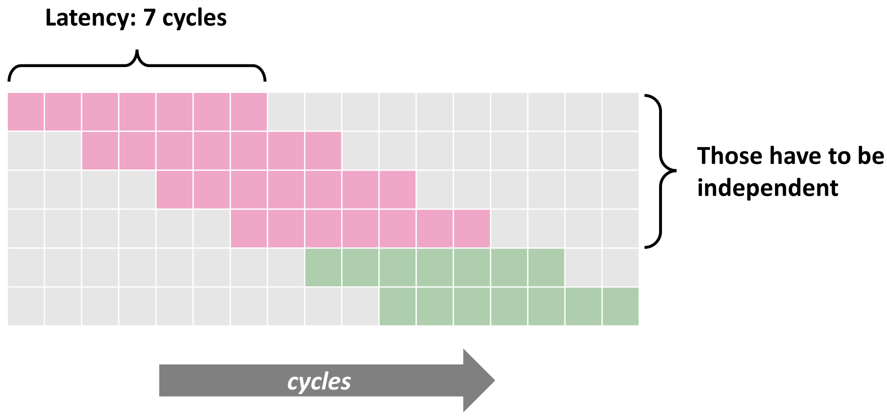
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Last Time: ILP

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



Last Time: How Many Accumulators?



Based on this insight: **K = #accumulators = ceil(latency/cycles per issue)**

Compiler Limitations

```
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;
}
```

```
void unroll2_sa(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;
}
```

- **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

*Chapter 5 in **Computer Systems: A Programmer's Perspective**, 2nd edition, Randal E. Bryant and David R. O'Hallaron, Addison Wesley 2010*
Part of these slides are adapted from the course associated with this book

Optimizing Compilers

-O

- **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)

```
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];
}
```

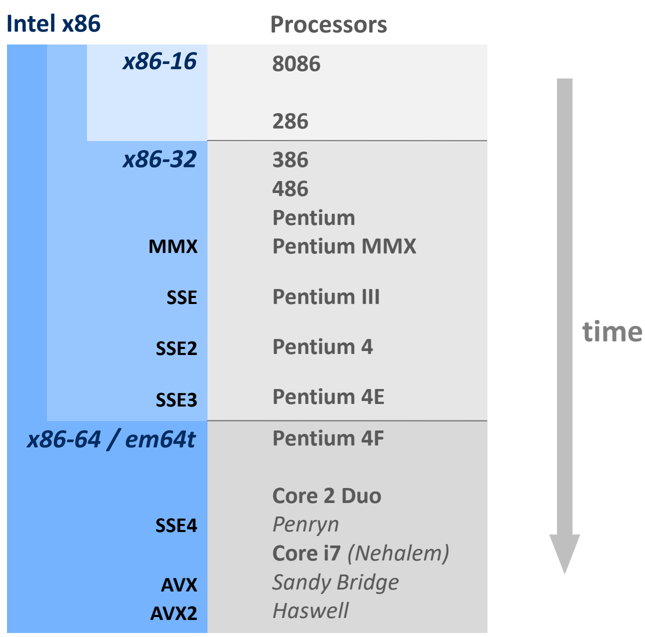
- **Compiled without flags:**
  ~1300 cycles
- **Compiled with -O3 -m64 -march=… -fno-tree-vectorize**
  ~150 cycles

Prevents use of SSE

MMX:
Multimedia extension

SSE:
Streaming SIMD extension

AVX:
Advanced vector extensions



Use architecture flags

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

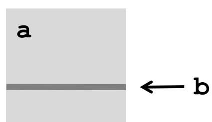
Organization

- 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**

```
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];
}
```




```
  int j;
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni+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*x → x << 4`
  - Utility machine dependent
- **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;
}
```

- **Compiler is likely to do**

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;
```

*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;
```

- **In simple cases compiler is likely to do**

Organization

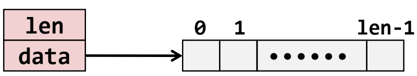
- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
  - Overview
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - ***Removing unnecessary procedure calls***
  - Optimization blocker: Procedure calls
  - Optimization blocker: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

Compiler is likely to do

15

Example: Data Type for Vectors

```
/* 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;
}
```

16

Example: Summing Vector Elements

```
/* 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;
}
```

```
/* 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

```
/* 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;
}
```

```
/* 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
```

Intel Xeon E3-1285L v3 (Haswell)
`CC=gcc -w -O3 -std=c99 -march=core-avx2`
Intel Atom D2550
`CC=gcc -w -std=c99 -O3 -march=atom`

```
$(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

```c
/* 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;
}
```

insert

```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;
}
```

- **Enables other optimizations**
- ***Problem:* performance libraries distributed as binary**

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

Compiler is likely to do

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');
}
```

$O(n^2)$ instead of $O(n)$

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

$O(n)$

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**
- **Form of code motion/precomputation**

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

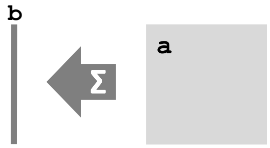
- **Instruction level parallelism (ILP): an example**
- **Optimizing compilers and optimization blockers**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
  - Optimization blocker: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

Compiler is likely to do

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];
  }
}
```

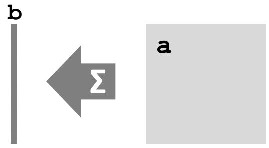


- **Code updates `b[i]` (= memory access) on every iteration**

Optimization Blocker: Memory Aliasing

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_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];
  }
}
```




```
/* 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:**

```
/* 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];
  }
}
```

```
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, 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

- ***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, …*

31

Example: MMM

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

```
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[N*i5 + j7] = C[N*i5 + j7] + A[N*i5 + k6] * B[j7 + N*k6]; }
```

tile each loop (= blocking MMM)

```
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*ii24 + j23 + jj26] = C[N*i21 + N*ii24 + j23 + jj26] +
                A[N*i21 + N*ii24 + k22 + kk25] * B[j23 + jj26 + N*k22 + N*kk25]; }
```

unroll inner three loops

32

unroll inner three loops

Now the reuse becomes apparent
(every elements used twice)

```
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 ) {
        C[N*i21 + j23] = C[N*i21 + j23] + A[N*i21 + k22] * B[j23 + N*k22];
        C[N*i21 + j23 + 1] = C[N*i21 + j23 + 1] + A[N*i21 + k22] * B[j23 + N*k22 + 1];
        C[N*i21 + N + j23] = C[N*i21 + N + j23] + A[N*i21 + N + k22] * B[j23 + N*k22];
        C[N*i21 + N + j23 + 1] = C[N*i21 + N + j23 + 1] + A[N*i21 + N + k22] * B[j23 + N*k22 + 1];
        C[N*i21 + j23] = C[N*i21 + j23] + A[N*i21 + k22 + 1] * B[j23 + N*k22 + N];
        C[N*i21 + j23 + 1] = C[N*i21 + j23 + 1] + A[N*i21 + k22 + 1] * B[j23 + N*k22 + N + 1];
        C[N*i21 + N + j23] = C[N*i21 + N + j23] + A[N*i21 + N + k22 + 1] * B[j23 + N*k22 + N];
        C[N*i21 + N + j23 + 1] =
          C[N*i21 + N + j23 + 1] + A[N*i21 + N + k22 + 1] * B[j23 + N*k22 + N + 1];
      }
}
```


unroll inner three loops

Now the reuse becomes apparent
(every elements used twice)

```
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 ) {
        C[N*i21 + j23] = C[N*i21 + j23] + A[N*i21 + k22] * B[j23 + N*k22];
        C[N*i21 + j23 + 1] = C[N*i21 + j23 + 1] + A[N*i21 + k22] * B[j23 + N*k22 + 1];
        C[N*i21 + N + j23] = C[N*i21 + N + j23] + A[N*i21 + N + k22] * B[j23 + N*k22];
        C[N*i21 + N + j23 + 1] = C[N*i21 + N + j23 + 1] + A[N*i21 + N + k22] * B[j23 + N*k22 + 1];
        C[N*i21 + j23] = C[N*i21 + j23] + A[N*i21 + k22 + 1] * B[j23 + N*k22 + N];
        C[N*i21 + j23 + 1] = C[N*i21 + j23 + 1] + A[N*i21 + k22 + 1] * B[j23 + N*k22 + N + 1];
        C[N*i21 + N + j23] = C[N*i21 + N + j23] + A[N*i21 + N + k22 + 1] * B[j23 + N*k22 + N];
        C[N*i21 + N + j23 + 1] =
          C[N*i21 + N + j23 + 1] + A[N*i21 + N + k22 + 1] * B[j23 + N*k22 + N + 1];
      }
}
```

scalar replacement

```
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 ) {

        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_12 = t0_7 * t0_3;
        t0_8 = t0_8 + t0_12;
        t0_12 = t0_7 * t0_2;
        t0_9 = t0_9 + t0_12;
        t0_12 = t0_5 * t0_3;
        t0_10 = t0_10 + t0_12;
        t0_12 = t0_5 * t0_2;
        t0_11 = t0_11 + t0_12;
        t0_12 = t0_6 * t0_1;
        t0_8 = t0_8 + t0_12;
        t0_12 = t0_6 * t0_0;
        t0_9 = t0_9 + t0_12;
        t0_12 = t0_4 * t0_1;
        t0_10 = t0_10 + t0_12;
        t0_12 = t0_4 * t0_0;
        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;
      }
}
```

load (first 12 statements), *compute* (next 16 statements), *store* (last 4 statements)

Effect on Runtime?

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

	N = 4	N = 100
Triple loop	202	2.3M

Effect on Runtime?

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

	N = 4	N = 100
Triple loop	202	2.3M
Six-fold loop	144	2.3M
+ Inner three unrolled	166	2.4M
+ scalar replacement	106	1.6M

Can Compiler Remove Aliasing?

```
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:

```
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, /Qalias-args-` (function arguments only)
- **For one loop: pragma**

```
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`)**

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

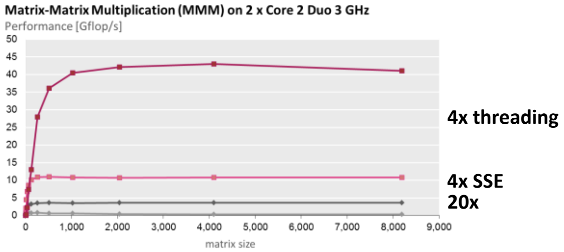

Organization

- 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***

Compiler is likely to do

Summary

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



- **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