Advanced Systems Lab

Spring 2022

Lecture: Compiler Limitations

Instructor: Markus Püschel, Ce Zhang

TA: Joao Rivera, several more

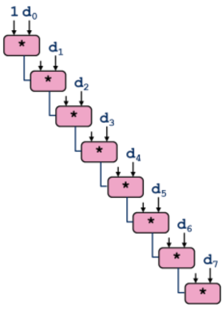
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Last Time: ILP

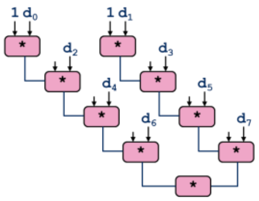
Coffee Lake

	latency	1/tp = gap
FP Add	4	0.5
FP Mul	4	0.5
Int Add	1	0.5
Int Mul	3	1

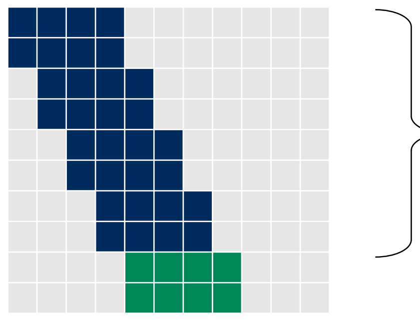
Deep (long) pipelines require ILP



Twice as fast



Last Time: How Many Accumulators?



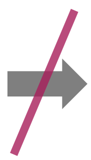
Based on this insight: K = #accumulators = ceil(latency/cycles per issue)
= ceil(latency * throughput)

Coffee Lake, FP mult: K = ceil(4/0.5) = 8

8x speedup

Compiler Limitations

```
void reduce(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*

Experiment: Try different flags and maybe different compilers

Example (On Skylake)

```
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 (gcc):
~1000 cycles

Compiled with -O3 -march=native -fno-tree-vectorize
~100 cycles

Prevents use of vector instructions

MMX:
Multimedia extension

SSE:
Streaming SIMD extension

AVX:
Advanced vector extensions

Intel x86	Processors (subset)
x86-16	8086
	286
x86-32	386
	486
	Pentium
MMX	Pentium MMX
SSE	Pentium III
SSE2	Pentium 4
SSE3	Pentium 4E
x86-64	Pentium 4F
	Core 2
SSE4	*Penryn*
	Core i3/5/7
AVX	*Sandy Bridge*
AVX2	*Haswell*
AVX-512	*Skylake-X*

time ↓

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*

9

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 that would only affect behavior under pathological conditions*

Most analysis is performed only within procedures

- *Whole-program analysis is too expensive in many cases*

Most analysis is based only on *static* information (C/C++)

- *Compiler has difficulty anticipating run-time inputs*
- *Not good at evaluating or dealing with choices*

10

Organization

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

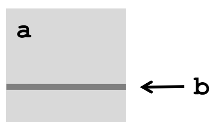
Reduce frequency with which computation is performed

- *If it will always produce same result*
- *Especially moving code out of loop (loop-invariant code motion)*

A form of 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`

- *Benefit is machine dependent*

Example:

```
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

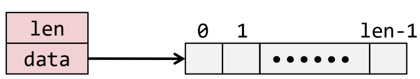
Optimizing compilers and optimization blockers

- *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

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;
}
```

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;
  int 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;
  int 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;
  int 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 E3-1535M (Skylake)
`CC=gcc -w -std=c99 -O3 -march=native`
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: 9.1 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: 4 cycles/add
Atom: 6 cycles/add

```
cat vector.c sum.c > vector_sum.c
$(CC) -c -o vector.o vector_sum.c
$(CC) -c -o main.o main.c
$(CC) -o vector vector.o main.o
```

Xeon: 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*

```
/* 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

```
/* 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

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*

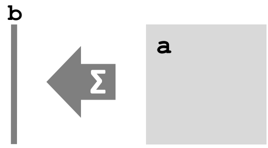
Compiler usually treats procedure call as a black box that cannot be analyzed

- *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

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



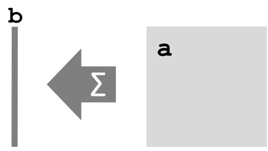
Code updates `b[i]` (= memory access) on every iteration

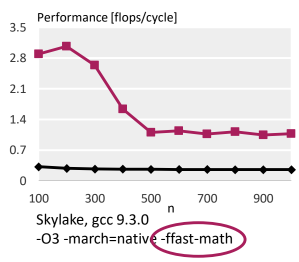

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows1(double *a, double *b, int n) {
  int i, j;

  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;
  }
}
```





learn about this flag

Why does the compiler not optimize as shown?

Reason: Possible Memory Aliasing

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

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

Example:

```
double A[9] =
  { 1, 2, 3,
    2, 4, 6,
    3, 6, 9 };

double B[3] = A+3;

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

```
Start:
A → 1 2 3
B → 2 4 6
    3 6 9
```

```
i=0:
1 2 3          1 2 3
0 4 6 •••••••  6 4 6
3 6 9          3 6 9
```

```
i=1:
1 2 3   1 2 3   1  2 3   1  2 3
6 0 6   6 6 6   6 12 6   6 18 6
3 6 9   3 6 9   3  6 9   3  6 9
```

```
i=2:
1  2 3          1  2  3
6 18 0 •••••••  6 18 18
3  6 9          3  6  9
```

```
Result:
6 18 18 ≠ 6 12 18
```


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:

- *Assumes no memory aliasing (otherwise likely an incorrect transformation)*
- *Copy array elements **that are reused** into temporary variables*
- *Perform computation on those variables*
- *Enables better register allocation and instruction scheduling by the compiler*

Optimization Blocker: Memory Aliasing

Memory aliasing: Two different memory references write to the same location

Easy to have happen in C
- *Address arithmetic is allowed*
- *Direct access to storage structures*

Hard to analyze = compiler cannot figure it out
- *Hence is conservative*

Prevents many performance optimizations

Solution: ***Scalar replacement*** (by programmer) 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, …`

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 k = 0; k < N; k++ )
    for( size_t i = 0; i < N; i++ )
      for( size_t j = 0; j < N; j++ )
        C[N*i + j] = C[N*i + j] + A[N*i + k] * B[j + N*k]; }
```

tile each loop (= blocking MMM)

```
void mmm(double const * A, double const * B, double * C, size_t N) {

  for( size_t i = 0; i < N; i+=2 )
    for( size_t j = 0; j < N; j+=2 )
      for( size_t k = 0; k < N; k+=2 )
        for( size_t kk = 0; kk < 2; kk++ )
          for( size_t ii = 0; ii < 2; ii++ )
            for( size_t jj = 0; jj < 2; jj++ )
              C[N*i + N*ii + j + jj] = C[N*i + N*ii + j + jj] +
                A[N*i + N*ii + k + kk] * B[j + jj + N*k + N*kk]; }
```

unroll inner three loops

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 i = 0; i < N; i+=2 )
    for( size_t j = 0; j < N; j+=2 )
      for( size_t k = 0; k < N; k+=2 ) {
        C[N*i + j]         = C[N*i + j] + A[N*i + k] * B[j + N*k];
        C[N*i + j + 1]     = C[N*i + j + 1] + A[N*i + k] * B[j + N*k + 1];
        C[N*i + N + j]     = C[N*i + N + j] + A[N*i + N + k] * B[j + N*k];
        C[N*i + N + j + 1] = C[N*i + N + j + 1] + A[N*i + N + k] * B[j + N*k + 1];
        C[N*i + j]         = C[N*i + j] + A[N*i + k + 1] * B[j + N*k + N];
        C[N*i + j + 1]     = C[N*i + j + 1] + A[N*i + k + 1] * B[j + N*k + N + 1];
        C[N*i + N + j]     = C[N*i + N + j] + A[N*i + N + k + 1] * B[j + N*k + N];
        C[N*i + N + j + 1] = C[N*i + N + j + 1] + A[N*i + N + k + 1] * B[j + N*k + N + 1];
      }
}
```

31

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 i = 0; i < N; i+=2 )
    for( size_t j = 0; j < N; j+=2 )
      for( size_t k = 0; k < N; k+=2 ) {
        C[N*i + j]         = C[N*i + j] + A[N*i + k] * B[j + N*k];
        C[N*i + j + 1]     = C[N*i + j + 1] + A[N*i + k] * B[j + N*k + 1];
        C[N*i + N + j]     = C[N*i + N + j] + A[N*i + N + k] * B[j + N*k];
        C[N*i + N + j + 1] = C[N*i + N + j + 1] + A[N*i + N + k] * B[j + N*k + 1];
        C[N*i + j]         = C[N*i + j] + A[N*i + k + 1] * B[j + N*k + N];
        C[N*i + j + 1]     = C[N*i + j + 1] + A[N*i + k + 1] * B[j + N*k + N + 1];
        C[N*i + N + j]     = C[N*i + N + j] + A[N*i + N + k + 1] * B[j + N*k + N];
        C[N*i + N + j + 1] = C[N*i + N + j + 1] + A[N*i + N + k + 1] * B[j + N*k + N + 1];
      }
}
```

scalar replacement

32

```
void mmm(double const * A, double const * B, double * C, size_t N) {

  for( size_t i = 0; i < N; i+=2 )
    for( size_t j = 0; j < N; j+=2 )
      for( size_t k = 0; k < N; k+=2 ) {

        double t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12;

        t7 =  A[N*i + k];
        t6 =  A[N*i + k + 1];
        t5 =  A[N*i + N + k];
        t4 =  A[N*i + N + k + 1];
        t3 =  B[j + N*k];
        t2 =  B[j + N*k + 1];
        t1 =  B[j + N*k + N];
        t0 =  B[j + N*k + N + 1];
        t8 =  C[N*i + j];
        t9 =  C[N*i + j + 1];
        t10 = C[N*i + N + j];
        t11 = C[N*i + N + j + 1];
        t12 = t7 * t3;
        t8  = t8 + t12;
        t12 = t7 * t2;
        t9  = t9 + t12;
        t12 = t5 * t3;
        t10 = t10 + t12;
        t12 = t5 * t2;
        t11 = t11 + t12;
        t12 = t6 * t1;
        t8  = t8 + t12;
        t12 = t6 * t0;
        t9  = t9 + t12;
        t12 = t4 * t1;
        t10 = t10 + t12;
        t12 = t4 * t0;
        t11 = t11 + t12;
        C[N*i + j]         = t8;
        C[N*i + j + 1]     = t9;
        C[N*i + N + j]     = t10;
        C[N*i + N + j + 1] = t11;
      }
}
```

load (annotates the loads of A, B and C)

compute (annotates the multiply/add statements)

store (annotates the stores to C)

All high performance libraries are written in this style!
[Example](#)

Even better: SSA style (later)

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

As usual, unrolling by itself does nothing

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

30–40% speedup

and we did not experiment yet with the block size …

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

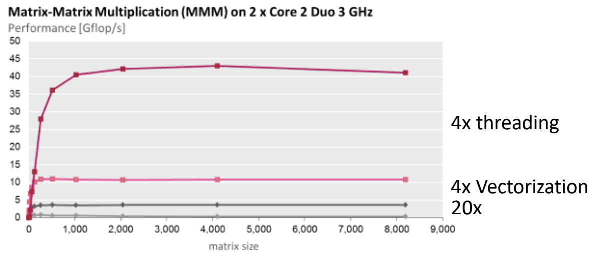
Optimizing compilers and optimization blockers

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

Compiler is likely to do

Summary

One can easily loose 10x or even more



What matters besides operation count:

- *Code style (unnecessary procedure calls, no aliasing, scalar replacement, …)*
- *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, avoid linked data structures, if possible

Unroll to some extent

- *To restructure computation to make ILP explicit*
- *To enable scalar replacement and hence better register allocation for variables that are reused*