Last Time: ILP

Coffee Lake

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>1/throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP Add</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>FP Mul</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>Int Add</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>Int Mul</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Deep (long) pipelines require ILP

Twice as fast
Last Time: How Many Accumulators?

Based on this insight:

\[ K = \#\text{accumulators} = \lceil \text{latency/cycles per issue} \rceil = \lceil \text{latency} \times \text{throughput} \rceil \]

Coffee Lake, FP mult:

\[ K = \lceil 4/0.5 \rceil = 8 \]

8x speedup

Those have to be independent

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

Experiment: Try different flags and maybe different compilers
### Example (On Skylake)

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

/* Multiply 4 x 4 matrices c = a*b + c */
void mm(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

---

### Intel x86 Processes (subset)

<table>
<thead>
<tr>
<th>Processor</th>
<th>1978</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86-16</td>
<td>8086</td>
</tr>
<tr>
<td>x86-32</td>
<td>386</td>
</tr>
<tr>
<td>x86-64</td>
<td>486</td>
</tr>
<tr>
<td>MMX</td>
<td>Pentium</td>
</tr>
<tr>
<td>SSE</td>
<td>Pentium MMX</td>
</tr>
<tr>
<td>SSE2</td>
<td>Pentium III</td>
</tr>
<tr>
<td>SSE3</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>SSE4</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td>AVX</td>
<td>Pentium 4F</td>
</tr>
<tr>
<td>AVX2</td>
<td>Core 2</td>
</tr>
<tr>
<td>AVX-512</td>
<td>Penryn</td>
</tr>
<tr>
<td></td>
<td>Core i3/5/7</td>
</tr>
<tr>
<td></td>
<td>Sandy Bridge</td>
</tr>
<tr>
<td></td>
<td>Haswell</td>
</tr>
<tr>
<td></td>
<td>Skylake-X</td>
</tr>
<tr>
<td></td>
<td>Ice Lake</td>
</tr>
<tr>
<td></td>
<td>Golden Cove</td>
</tr>
</tbody>
</table>

**Use architecture flags**
Optimizing Compilers

Compilers are *good* at: mapping program to machine
- register allocation
- 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 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
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

```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, usually don’t do yourself
Strength Reduction

Replace costly operation with simpler one

Example: Shift/add instead of multiply or divide $16 \times x \rightarrow x \ll 4$

- Benefit is machine dependent

Example:

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

Compiler is likely to do, usually don’t do yourself

Share Common Subexpressions

Reuse portions of expressions

Compilers often not very sophisticated in exploiting arithmetic properties

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

```c
/* Sum neighbors of i, j */
up = val[(i-1)*n + j ];
down = val[(i+1)*n + j ];
left = val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

In simple cases compiler is likely to do, usually don’t do yourself
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
Usually don’t do yourself: it may prevent other compiler optimizations

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**

```c
/* 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

```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;
}
```

**Removing Procedure Call**

```c
/* 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;
}
```

```c
/* 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

Enables other optimizations

Problem:
- code size can increase dramatically
- performance libraries distributed as binary

Optimization Blocker: 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;
}
```

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

Prevents change of string s

O(n)
Improving Performance

```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');
}
```

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

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

Example:

```
Start:
A → 1 2 3
  1 2 3
  0 4 6
  3 6 9
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
  6 0 6
  3 6 9
```

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

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

Result B:
```
 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

One solution: Scalar replacement (by programmer) 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 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] += A[N*i + k] * B[j + N*k];
}
```

tile each loop (= blocking MMM)

```c
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] += 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 element used twice)

```c
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 ) {
        }
}
```
```c
void mm(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];
                t5 =  B[j + N*k];
                t4 =  B[j + N*k + 1];
                t9 =  C[N*i + j];
                t8 =  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;
                t10 = t5 * t3;
                t11 = t5 * t2;
                t11 = t6 * t11;
                t8 = t8 + t12;
                t12 = t6 * t8;
                t9 = t9 + t12;
                t10 = t4 * t11;
                t11 = t4 * t8;
                t11 = t11 + t12;
                C[N*i + j] = t8;
                C[N*i + j + 1] = t10;
                C[N*i + N + j] = t10;
                C[N*i + N + j + 1] = t11;
            }
    }
```

**All high performance libraries are written in this style!**

**Example**

Even better: SSA style (later)

### Effect on Runtime (Shown in Cycles)?

**Intel Xeon E-2176M (Coffee Lake)**

**compiler:** gcc 9.4.0

**flags:** -O3 -ffast-math -march=native

<table>
<thead>
<tr>
<th></th>
<th>N = 4</th>
<th>N = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triple loop</td>
<td>181</td>
<td>2.2M</td>
</tr>
</tbody>
</table>

*As usual, unrolling by itself does nothing*
Effect on Runtime (Shown in Cycles)?

<table>
<thead>
<tr>
<th></th>
<th>N = 4</th>
<th>N = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triple loop</td>
<td>181</td>
<td>2.2M</td>
</tr>
<tr>
<td>Six-fold loop</td>
<td>158</td>
<td>2.4M</td>
</tr>
<tr>
<td>+ Inner three unrolled</td>
<td>160</td>
<td>2.4M</td>
</tr>
<tr>
<td>+ Scalar replacement</td>
<td>90</td>
<td>1.5M</td>
</tr>
</tbody>
</table>

30–45% speedup for smallish sizes

and we did not experiment yet with the block size (here 2 x 2) ...

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

```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
- 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 lose 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