Last Time: ILP

<table>
<thead>
<tr>
<th>Haswell</th>
<th>latency</th>
<th>(1/tp = \text{gap})</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP Mul</td>
<td>5</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:

$$K = \#\text{accumulators} = \text{ceil}(\text{latency}/\text{cycles per issue})$$

$$= \text{ceil}(\text{latency} \times \text{throughput})$$

Haswell, FP mult:

$$K = \text{ceil}(5/0.5) = 10$$

10x speedup

Those have to be independent

Question From Last Time

Can I use FMAs for the adds for further speedup?

**Yes:** using intrinsics

### Haswell

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>$1/\text{tp} = \text{gap}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP FMA</td>
<td>5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Can I use FMAs for the adds for further speedup?

**Yes:** using intrinsics

<table>
<thead>
<tr>
<th>Function</th>
<th>Latency [cyc/ops]</th>
<th>$1/\text{tp} = \text{gap}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_double (1,1)</td>
<td><strong>2.95522</strong></td>
<td>1</td>
</tr>
<tr>
<td>ADD_double (2,2)</td>
<td>1.49528</td>
<td>1</td>
</tr>
<tr>
<td>ADD_double (3,3)</td>
<td><strong>1.0046</strong></td>
<td>0.5</td>
</tr>
<tr>
<td>ADD_double (4,4)</td>
<td>1.00578</td>
<td>0.5</td>
</tr>
<tr>
<td>FMA_double (1,1)</td>
<td><strong>4.92087</strong></td>
<td>1</td>
</tr>
<tr>
<td>FMA_double (2,2)</td>
<td>2.46956</td>
<td>1</td>
</tr>
<tr>
<td>FMA_double (3,3)</td>
<td>1.65881</td>
<td>0.5</td>
</tr>
<tr>
<td>FMA_double (4,4)</td>
<td>1.24497</td>
<td>0.5</td>
</tr>
<tr>
<td>FMA_double (6,6)</td>
<td>0.843982</td>
<td>0.5</td>
</tr>
<tr>
<td>FMA_double (8,8)</td>
<td>0.64387</td>
<td>0.5</td>
</tr>
<tr>
<td>FMA_double (10,10)</td>
<td><strong>0.520438</strong></td>
<td>0.5</td>
</tr>
<tr>
<td>FMA_double (12,12)</td>
<td>0.51887</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Another 2x speedup

Compiler doesn’t do
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

```c
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;
}
```

```c
void unroll2_sa(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    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;
}
```

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 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 (gcc):
~1300 cycles

Compiled with -O3 -m64 -march=... -fno-tree-vectorize
~150 cycles

Prevents use of SSE
### 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 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

- 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)

- 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];
  }
  
  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 \( 16 \times x \rightarrow x \ll 4 \)
  - Benefit is machine dependent

- Example:

  ```c
  int ni = 0;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      a[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

In simple cases compiler is likely to do:

```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;
```

```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;
```

3 multis: \(i^n, (i-1)^n, (i+1)^n\)
1 mult: \(i^n\)

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

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

Example: Summing Vector Elements

```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;
}

/* 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++) {
        t = 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;
    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;
}
```

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

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

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

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

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

  \[O(n^2) \text{ instead of } O(n)\]

### Improving Performance

- 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

```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];
    }
}
```

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

Value of B:
- init: [4, 8, 16]
- i = 0: [3, 8, 16]
- i = 1: [3, 22, 16]
- i = 2: [3, 22, 224]
Removing Aliasing

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 register allocation and instruction scheduling

```
/* 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;
    }
}
```

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: \( t_1 = a[i], t_2 = b[i+1] \), ...
    - Compute: \( t_4 = t_1 \times t_2 \), ...
    - Store: \( a[i] = t_{12}, b[i+1] = t_7 \), ...

Example: MMM

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

```c
void mmm(double * A, double * 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++ )
}
```

Tile each loop (= blocking MMM)

```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 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 + ii24 + jj26] * B[j23 + jj26 + N*k22 + N*jj26];
}
```

Unroll inner three loops

```c
void mmm(double * A, double * 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 + N + N + j23 + 1] =
            }
}
```

Now the reuse becomes apparent (every elements used twice)
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 )
            for( size_t k22 = 0; k22 < N; k22+=2 ) {
                C[i21*N + j23 + 1] = (C[i21*N + j23 + 1] + A[i21*N + k22] * B[j23 + N*k22 + 1]);
            }
}
```

All high performance libraries are written in this style!

Example
Effect on Runtime?

Intel Core i7-2600 (Sandy Bridge)
compiler: icc 12.1
flags: -O3 -no-vec -no-ipo -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>
</tbody>
</table>

As usual, unrolling by itself does nothing
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
    - 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 lose 10x, 100x in runtime 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 (simple data structure) if possible**

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