last time (1)

out-of-order execution
- new hazards: computing old value of register while reading new value?

architectural versus physical registers

register renaming
- preprocessing step for instructions
- new physical register for each version of architectural register
- separate names for inputs and outputs
- might also do other instruction translations here

instruction dispatch
- queue of waiting renamed instructions
- each physical register: ready or not ready yet
- run instructions whose inputs are ready
- different types of 'slots' for instructions

last time (2)

data flow diagrams

reassociation

- (A op B) op (C op D) often faster than (A op (B op (C op D)))
- more operations can be done in parallel

multiple accumulators

- reassocation for loops
- use two or more accumulators (sums, products)
- combine at end

quiz Q1 (1)

I messed up

version A, B meant to be cache blocked versions

...

but I wrote `i = 0` instead of `i = ii`

(also missed A[i+1] in version D)

quiz Q1 version A

(with i = ii, j = jj correction)

```
for (int ii = 0; ii < N; ii += 2) {
    for (int i = ii; i < ii + 2 && i < N; i += 1) {
        for (int jj = 0; jj < N; jj += 2) {
            for (int j = jj; j < jj + 2 && j < i; j += 1) {
                A[i] += B[i * N + j] * C[j * N + i];
            }
        }
    }
}
```

does (i, j) = (0, 0), then (0, 1), (0, 2), …

then (1, 0), (1, 1), (1, 2), …

$\rightarrow$ same order as original!

quiz Q1 version B

(with i = ii, j = jj correction)

```
for (int ii = 0; ii < N; ii += 2) {
    for (int jj = j; jj < N; jj += 2) {
        for (int i = ii; i < ii + 2 && i < N; i += 1) {
            for (int j = jj; j < jj + 2 && j < i; j += 1) {
                A[i] += B[i * N + j] * C[j * N + i];
            }
        }
    }
}
```

(0, 0), (0, 1), (1, 0), (1, 1), (0, 2), (0, 3), (1, 2), (1, 3), ...

→ more spatial locality in A!

quiz Q1 version C

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

(0, 0), (0, 1), (0, 2), (0, 3), ...

(1, 0), (1, 1), (1, 2), (1, 3), ...

$\rightarrow$ same order of i, j as original

so same data cache accesses (assuming i, j, etc. in regs)

quiz Q1 version D

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

iterating in i, not j

so worse temporal locality in A, spatial locaity in B, C

→ worse performance

quiz Q2

init. rbx = x0, r8 = x1, r9 = x2

```
xorq %rbx, %r8 | xorq %x0, %x1 -> %x3
andq %rbx, %r9 | xorq %x0, %x2 -> %x4
subq %r8, %rbx | xorq %x3, %x4 -> %x5
addq %rbx, %r9 | xorq %x5, %x4 -> %x6
```

quiz Q3, 5

```
movq    (%rdx,%rcx,8), %r9  /* E */
movq    (%rsi,%rax,8), %r10 /* F */
addq    %r10, %r9           /* G */
```

new 'version' of r9, r10 each time

so no dependencies (for the add) between loop iterations!

can be done in parallel

since already done in parallel, no multiple accumulators benefit

quiz Q4

```
    // (setup code)
inner_loop:
    movq    (%rdx,%rcx,8), %r9  /* E */
    movq    (%rsi,%rax,8), %r10 /* F */
    addq    %r10, %r9           /* G */
    movq    %r9, (%r8,%rcx,8)   /* H */
    incq    %rcx                /* I */
    cmpq    $1024, %rcx         /* J */
    jne     inner_loop          /* K */
    // (end of outer loop)
```

E through I duplicated

one comparision + cond jump per iteration

rotate/smooth-checkpoint due time

rotate + smooth-checkpoint originally were due next Wednesday

now due just before Thanksgiving break

strongly recommend doing early

...especially if you don't want to work on Thanksgiving week

performance HWs

assignment 1: rotate an image

assignment 2: smooth (blur) an image

- two parts
- part 1: due with rotate — optimizations we've mostly talked about
- part 2: due later — part with vector instructions

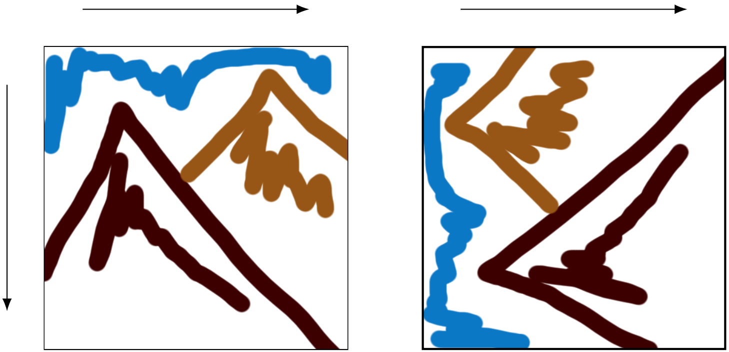
image representation

```
typedef struct {
    unsigned char red, green, blue, alpha;
} pixel;
pixel *image = malloc(dim * dim * sizeof(pixel));

image[0]            // at (x=0, y=0)
image[4 * dim + 5]  // at (x=5, y=4)
...
```

rotate assignment

```
void rotate(pixel *src, pixel *dst, int dim) {
    int i, j;
    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            dst[RIDX(dim – 1 – j, i, dim)] =
                src[RIDX(i, j, dim)];
}
```



preprocessor macros

```
#define DOUBLE(x) x*2

int y = DOUBLE(100);
// expands to:
int y = 100*2;
```

macros are text substitution (1)

```
#define BAD_DOUBLE(x) x*2

int y = BAD_DOUBLE(3 + 3);
// expands to:
int y = 3+3*2;
// y == 9, not 12
```

macros are text substitution (2)

```
#define FIXED_DOUBLE(x) (x)*2

int y = DOUBLE(3 + 3);
// expands to:
int y = (3+3)*2;
// y == 9, not 12
```

RIDX?

```
#define RIDX(x, y, n) ((x) * (n) + (y))

dst[RIDX(dim - 1 - j, 1, dim)]
// becomes *at compile-time*:
dst[((dim - 1 - j) * (dim) + (1))]
```

performance grading

you can submit multiple variants in one file
- grade: best performance
- don't delete stuff that works!

we will measure speedup on my machine
- web viewer for results (with some delay — has to run)

grade: achieving certain speedup on my machine
- thresholds based on results with certain optimizations

general advice

(for when we don't give specific advice)

try techniques from book/lecture that seem applicable

vary numbers (e.g. cache block size)

- often — too big/small is worse

some techniques combine well

- loop unrolling and cache blocking
- loop unrolling and reassociation/multiple accumulators

example assembly (unoptimized)

```
long sum(long *A, int N) {
    long result = 0;
    for (int i = 0; i < N; ++i)
        result += A[i];
    return result;
}
```

```
sum:    ...
the_loop:
        ...
        leaq    0(,%rax,8), %rdx// offset <- i * 8
        movq    —24(%rbp), %rax // get A from stack
        addq    %rdx, %rax      // add offset
        movq    (%rax), %rax    // get *(A+offset)
        addq    %rax, —8(%rbp)  // add to sum, on stack
        addl    $1, —12(%rbp)   // increment i
condition:
        movl    —12(%rbp), %eax
        cmpl    —28(%rbp), %eax
        jl      the_loop
        ...
```

example assembly (gcc 5.4 -Os)

```
long sum(long *A, int N) {
    long result = 0;
    for (int i = 0; i < N; ++i)
        result += A[i];
    return result;
}

sum:
        xorl    %edx, %edx
        xorl    %eax, %eax
the_loop:
        cmpl    %edx, %esi
        jle     done
        addq    (%rdi,%rdx,8), %rax
        incq    %rdx
        jmp     the_loop
done:
        ret
```

example assembly (gcc 5.4 -O2)

```
long sum(long *A, int N) {
    long result = 0;
    for (int i = 0; i < N; ++i)
        result += A[i];
    return result;
}
sum:
        testl   %esi, %esi
        jle     return_zero
        leal    -1(%rsi), %eax
        leaq    8(%rdi,%rax,8), %rdx // rdx=end of A
        xorl    %eax, %eax
the_loop:
        addq    (%rdi), %rax // add to sum
        addq    $8, %rdi     // advance pointer
        cmpq    %rdx, %rdi
        jne     the_loop
        rep ret
return_zero:    ...
```

example assembly (gcc 9.2 -O3)

```
sum:
        testl   %esi, %esi
        ... /* approx 10 lines omitted */
the_loop:
        movdqu  (%rax), %xmm2 /* <-- load 16 bytes from array */
        addq    $16, %rax
        paddq   %xmm2, %xmm0  /* <-- add 2 pairs of longs */
        cmpq    %rdx, %rax
        jne     the_loop
        ... /* approx 20 lines omitted */
        ret
```

example assembly (gcc 9.2 -O3 -march=skylake)

```
sum:
        testl   %esi, %esi
        ... /* approx 10 lines omitted */
the_loop:
        vpaddq  (%rax), %ymm0, %ymm0  /* <- add 4 pairs of longs */
        addq    $32, %rax
        cmpq    %rdx, %rax
        jne     the_loop
        ... /* approx 20 lines omitted */
        ret
```

gcc 9.2 -O3 -funroll-loops -march=skylake

```
sum:
        testl   %esi, %esi
        ... /* approx 60 lines omitted */
the_loop:  /* loop unrolled 8 times + instrs that add 4 pairs at a
        vpaddq  (%r8), %ymm0, %ymm1  /* <-- add 4 pairs of longs */
        addq    $256, %r8
        vpaddq  −224(%r8), %ymm1, %ymm2
        vpaddq  −192(%r8), %ymm2, %ymm3
        vpaddq  −160(%r8), %ymm3, %ymm4
        vpaddq  −128(%r8), %ymm4, %ymm5
        vpaddq  −96(%r8), %ymm5, %ymm6
        vpaddq  −64(%r8), %ymm6, %ymm7
        vpaddq  −32(%r8), %ymm7, %ymm0
        cmpq    %rcx, %r8
        jne     .L4
        ... /* approx 20 lines omitted */
        ret
```

example assembly (clang 9.0 -O -march=skylake)

```
sum:
        testl   %esi, %esi
        ... /* approx 35 lines omitted */
the_loop: /* loop unrolled + multiple accumulators + instrs that 4 pairs at a time */
        vpaddq  (%rdi,%rsi,8), %ymm0, %ymm0
        vpaddq  32(%rdi,%rsi,8), %ymm1, %ymm1
        vpaddq  64(%rdi,%rsi,8), %ymm2, %ymm2
        vpaddq  96(%rdi,%rsi,8), %ymm3, %ymm3
        vpaddq  128(%rdi,%rsi,8), %ymm0, %ymm0
        vpaddq  160(%rdi,%rsi,8), %ymm1, %ymm1
        vpaddq  192(%rdi,%rsi,8), %ymm2, %ymm2
        vpaddq  224(%rdi,%rsi,8), %ymm3, %ymm3
        vpaddq  256(%rdi,%rsi,8), %ymm0, %ymm0
        vpaddq  288(%rdi,%rsi,8), %ymm1, %ymm1
        vpaddq  320(%rdi,%rsi,8), %ymm2, %ymm2
        vpaddq  352(%rdi,%rsi,8), %ymm3, %ymm3
        vpaddq  384(%rdi,%rsi,8), %ymm0, %ymm0
        vpaddq  416(%rdi,%rsi,8), %ymm1, %ymm1
        vpaddq  448(%rdi,%rsi,8), %ymm2, %ymm2
        vpaddq  480(%rdi,%rsi,8), %ymm3, %ymm3
        addq    $64, %rsi
        addq    $4, %rax
        jne the_loop
```

optimizing compilers

these usually make your code fast

often not done by default

compilers and humans are good at different kinds of optimizations

compiler limitations

needs to generate code that does the same thing…

…even in corner cases that "obviously don't matter"

often doesn't 'look into' a method

needs to assume it might do anything

can't predict what inputs/values will be

e.g. lots of loop iterations or few?

can't understand code size versus speed tradeoffs

compiler limitations

needs to generate code that does the same thing…

- …even in corner cases that "obviously don't matter"

often doesn't 'look into' a method

- needs to assume it might do anything

can't predict what inputs/values will be

- e.g. lots of loop iterations or few?

can't understand code size versus speed tradeoffs

aliasing

```
void twiddle(long *px, long *py) {
    *px += *py;
    *px += *py;
}
```

the compiler **cannot** generate this:

```
twiddle: // BROKEN // %rsi = px, %rdi = py
        movq    (%rdi), %rax // rax <- *py
        addq    %rax, %rax   // rax <- 2 * *py
        addq    %rax, (%rsi) // *px <- 2 * *py
        ret
```

aliasing problem

```
void twiddle(long *px, long *py) {
    *px += *py;
    *px += *py;
    // NOT the same as *px += 2 * *py;
}
...
    long x = 1;
    twiddle(&x, &x);
    // result should be 4, not 3
```

```
twiddle: // BROKEN // %rsi = px, %rdi = py
        movq    (%rdi), %rax // rax <- *py
        addq    %rax, %rax   // rax <- 2 * *py
        addq    %rax, (%rsi) // *px <- 2 * *py
        ret
```

non-contrived aliasing

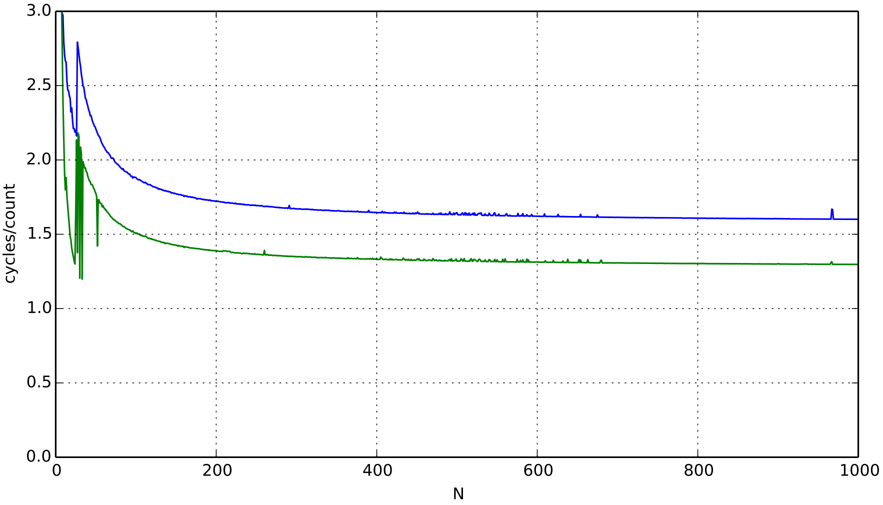
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

non-contrived aliasing

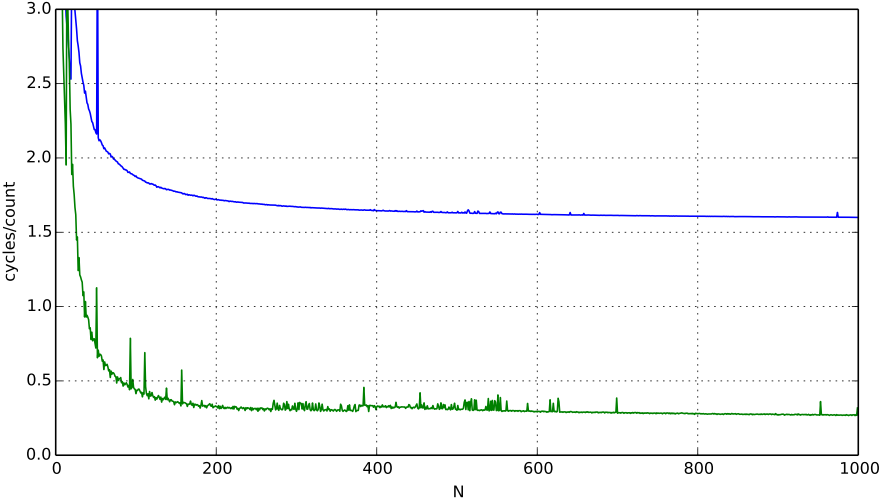
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

```
void sumRows2(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        int sum = 0;
        for (int col = 0; col < N; ++col)
            sum += matrix[row * N + col];
        result[row] = sum;
    }
}
```

aliasing and performance (1) / GCC 5.4 -O2



aliasing and performance (2) / GCC 5.4 -O3



automatic register reuse

Compiler would need to generate overlap check:

```
if (result > matrix + N * N || result < matrix) {
    for (int row = 0; row < N; ++row) {
        int sum = 0; /* kept in register */
        for (int col = 0; col < N; ++col)
            sum += matrix[row * N + col];
        result[row] = sum;
    }
} else {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
        }
    }
}
```

aliasing problems with cache blocking

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

can compiler keep `A[i*N+k]` in a register?

“register blocking”

```
for (int k = 0; k < N; ++k) {
  for (int i = 0; i < N; i += 2) {
    float Ai0k = A[(i+0)*N + k];
    float Ai1k = A[(i+1)*N + k];
    for (int j = 0; j < N; j += 2) {
      float Bkj0 = B[k*N + j+0];
      float Bkj1 = B[k*N + j+1];
      C[(i+0)*N + j+0] += Ai0k * Bkj0;
      C[(i+1)*N + j+0] += Ai1k * Bkj0;
      C[(i+0)*N + j+1] += Ai0k * Bkj1;
      C[(i+1)*N + j+1] += Ai1k * Bkj1;
    }
  }
}
```

aliasing exercise

```
void add(int *s1, int *s2, int *d) {
    for (int i = 0; i < 1000; ++i)
        d[i] = s1[i] + s2[i];
}
```

The compiler **cannot** generate code equivalent to this:

```
void add(int *s1, int *s2, int *d) {
    for (int i = 0; i < 1000; i += 2) {
        int temp1 = s1[i] + s2[i];
        int temp2 = s1[i+1] + s2[i+1];
        d[i] = temp1; d[i+1] = temp2;
    }
}
```

Which is an example of a call where the results could disagree:

A. `add(&A[0], &A[1], &B[0])` B. `add(&A[0], &A[0], &A[1])`

C. `add(&B[0], &A[10], &A[0])` D. `add(&A[1000], &A[1001], &A[0])`

(assume A, B are distinct, large arrays)

aliasing exercise

recall: s1=s2=A+0; d=A+1

```
for (int i = 0; i < 1000; ++i)
    d[i] = s1[i] + s2[i];

/* i = 0: */ A[1] = A[0] + A[0];
/* i = 1: */ A[2] = A[1] + A[1];
```

```
for (int i = 0; i < 1000; i += 2) {
    temp1 = s1[i] + s2[i];
    temp2 = s1[i] + s2[i];
    d[i] = temp1;
    d[i+1] = temp2;

/* i = 0: */ temp1 = A[0] + A[0];
             temp2 = A[1] + A[1];
             A[1] = temp1;
             A[2] = temp2;
```

aliasing and cache optimizations

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

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

`C = A`? `C = &A[10]`?

compiler can't generate same code for both

loop unrolling downsides

bigger executables → instruction cache misses

slower if small number of loop iterations

- extra code to handle leftovers, etc.

want to unroll loops that are run a lot and quick to execute

problem: compiler probably can't tell if this meets those criteria

```
for (int i = 0; i < some_variable; ++i) {
  sum += some_function();
}
```

figuring out how to unroll?

exercise: why can the compiler probably not do this transformation?

```
void foo() { int sum = 0;
  for (int i = 0; i < some_global_variable; ++i) {
    sum += some_function();
  }
}
```

```
void foo_transformed() { int sum = 0;
  int i = 0;
  if (some_global_variable % 2 == 1) {
    i += 1;
    sum += some_function();
  }
  for (; i < some_global_variable; i += 2) {
    sum += some_function();
    sum += some_function();
  }
}
```

multiple accumulators downsides

downsides of loop unrolling

- bigger executables, slower if small number of iterations

+ uses extra registers (can't use those regs for something else)

want to use multiple accumulators if latency likely bottleneck

problem: compiler probably can't tell if this meets those criteria

```
for (int i = 0; i < some_variable; ++i) {
  sum += some_function();
}
```

loop with a function call

```
int addWithLimit(int x, int y) {
    int total = x + y;
    if (total > 10000)
        return 10000;
    else
        return total;
}
...
int sum(int *array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum = addWithLimit(sum, array[i]);
    return sum;
}
```

loop with a function call

```
int addWithLimit(int x, int y) {
    int total = x + y;
    if (total > 10000)
        return 10000;
    else
        return total;
}
...
int sum(int *array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum = addWithLimit(sum, array[i]);
    return sum;
}
```

function call assembly

```
    ... loop stuff ...
    movl (%rbx), %esi // mov array[i]
    movl %eax, %edi   // mov sum
    call addWithLimit
    ... more loop stuff ...
    ...
addWithLimit:
    ... /* code here */
    ret
```

extra instructions executed: two moves, a call, and a ret

function call assembly

```
    ... loop stuff ...
    movl (%rbx), %esi // mov array[i]
    movl %eax, %edi   // mov sum
    call addWithLimit
    ... more loop stuff ...
    ...
addWithLimit:
    ... /* code here */
    ret
```

extra instructions executed: two moves, a call, and a ret

alternative: *inline* the call

```
    ... loop stuff ...
    ... /* code here (+ small changes for arguments
                        being in different places) */
    ... more loop stuff ...
```

manual inlining

```
int sum(int *array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + array[i];
        if (sum > 10000)
            sum = 10000;
    }
    return sum;
}
```

inlining pro/con

avoids call, ret, extra move instructions

allows compiler to use more registers

- no caller-saved register problems

but not always faster:

worse for instruction cache

- (more copies of function body code)

compiler inlining

compilers will inline, but...

will usually avoid making code much bigger

- heuristic: inline if function is small enough
- heuristic: inline if called exactly once

will usually not inline across .o files

some compilers allow hints to say "please inline/do not inline this function"

compiler limitations

needs to generate code that does the same thing...

...even in corner cases that "obviously don't matter"

often doesn't 'look into' a method

needs to assume it might do anything

can't predict what inputs/values will be

e.g. lots of loop iterations or few?

can't understand code size versus speed tradeoffs

remove redundant operations (1)

```
int number_of_As(const char *str) {
    int count = 0;
    for (int i = 0; i < strlen(str); ++i) {
        if (str[i] == 'a')
            count++;
    }
    return count;
}
```

remove redundant operations (1, fix)

```
int number_of_As(const char *str) {
    int count = 0;
    int length = strlen(str);
    for (int i = 0; i < length; ++i) {
        if (str[i] == 'a')
            count++;
    }
    return count;
}
```

call strlen once, not once per character!

Big-Oh improvement!

remove redundant operations (1, fix)

```
int number_of_As(const char *str) {
    int count = 0;
    int length = strlen(str);
    for (int i = 0; i < length; ++i) {
        if (str[i] == 'a')
            count++;
    }
    return count;
}
```

call strlen once, not once per character!

Big-Oh improvement!

remove redundant operations (2)

```
int shiftArray(int *source, int *dest, int N, int amount) {
    for (int i = 0; i < N; ++i) {
        if (i + amount < N)
            dest[i] = source[i + amount];
        else
            dest[i] = source[N - 1];
    }
}
```

compare i + amount to N many times

remove redundant operations (2, fix)

```
int shiftArray(int *source, int *dest, int N, int amount) {
    int i;
    for (i = 0; i + amount < N; ++i) {
        dest[i] = source[i + amount];
    }
    for (; i < N; ++i) {
        dest[i] = source[N - 1];
    }
}
```

eliminate comparisons

compiler limitations

needs to generate code that does the same thing…

…even in corner cases that "obviously don't matter"

often doesn't 'look into' a method

needs to assume it might do anything

can't predict what inputs/values will be

e.g. lots of loop iterations or few?

can't understand code size versus speed tradeoffs

exercise: when optimizations backfire...

Which of these optimizations are likely to **increase** machine code size? (**Select all that apply.**)

Which of these optimizations are likely to **increase** number of instructions executed? (**Select all that apply.**)

A. cache blocking B. function inlining C. loop unrolling
D. moving a calculation outside a loop
E. multiple accumulators (after loop unrolling)

backup slides

loop unrolling performance

on my laptop with 992 elements (fits in L1 cache)

work/loop iteration	cycles/element	instructions/element
1	1.33	4.02
2	1.03	2.52
4	1.02	1.77
8	1.01	1.39
16	1.01	1.21
32	1.01	1.15

1.01 cycles/element — latency bound

multiple accumulators

```
int i;
long sum1 = 0, sum2 = 0;
for (i = 0; i + 1 < N; i += 2) {
    sum1 += A[i];
    sum2 += A[i+1];
}
// handle leftover, if needed
if (i < N)
    sum1 += A[i];
sum = sum1 + sum2;
```

multiple accumulators performance

on my laptop with 992 elements (fits in L1 cache)

16x unrolling, variable number of accumulators

accumulators	cycles/element	instructions/element
1	1.01	1.21
2	0.57	1.21
4	0.57	1.23
8	0.59	1.24
16	0.76	1.57

starts hurting after too many accumulators

why?

multiple accumulators performance

on my laptop with 992 elements (fits in L1 cache)

16x unrolling, variable number of accumulators

accumulators	cycles/element	instructions/element
1	1.01	1.21
2	0.57	1.21
4	0.57	1.23
8	0.59	1.24
16	0.76	1.57

starts hurting after too many accumulators

why?

8 accumulator assembly

```
sum1 += A[i + 0];
sum2 += A[i + 1];
...
...
```

```
addq    (%rdx), %rax        // sum1 +=
addq    8(%rdx), %rcx       // sum2 +=
subq    $-128, %rdx         // i +=
addq    -112(%rdx), %rbx    // sum3 +=
addq    -104(%rdx), %r11    // sum4 +=
...
....
cmpq  %r14, %rdx
```

register for each of the `sum1`, `sum2`, …variables:

16 accumulator assembly

compiler runs out of registers

starts to use the stack instead:

```
movq    32(%rdx), %rax  // get A[i+13]
addq    %rax, -48(%rsp) // add to sum13 on stack
```

code does extra cache accesses

also — already using all the adders available all the time

so performance increase not possible

multiple accumulators performance

on my laptop with 992 elements (fits in L1 cache)

16x unrolling, variable number of accumulators

accumulators	cycles/element	instructions/element
1	1.01	1.21
2	0.57	1.21
4	0.57	1.23
8	0.59	1.24
16	0.76	1.57

starts hurting after too many accumulators

why?

maximum performance

2 additions per element:

- one to add to sum
- one to compute address (part of mov)

3/16 add/sub/cmp + 1/16 branch per element:

- over 16 because loop unrolled 16 times
- loop overhead
- compiler not as efficient as it could have been

2+3/16+1/16 = 2+1/4 instructions per element

hardware limits on my machine

4(?) register renamings per cycle

(Intel doesn't really publish exact numbers here...)

4-6 instructions decoded/cycle

(depending on instructions)

4(?) microinstructions commited/cycle

4 (add or cmp+branch executed)/cycle

hardware limits on my machine

4(?) register renamings per cycle

(Intel doesn't really publish exact numbers here...)

4-6 instructions decoded/cycle

(depending on instructions)

4(?) microinstructions commited/cycle

4 (add or cmp+branch executed)/cycle

$(2 + 1/4) \div 4 \approx 0.57$ cycles/element

getting over this limit

the $+1/4$ was from loop overhead

solution: more loop unrolling!

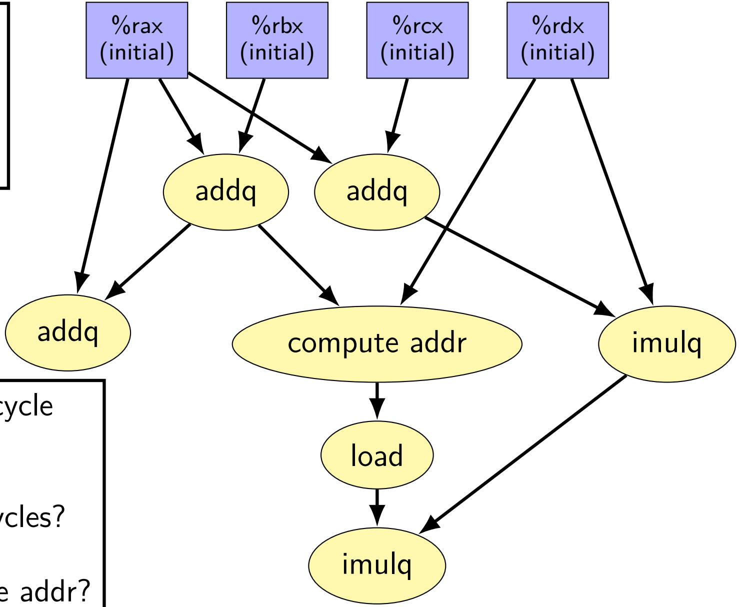
common theme with optimization:

fix one bottleneck (need to do adds one after the other)

find another bottleneck

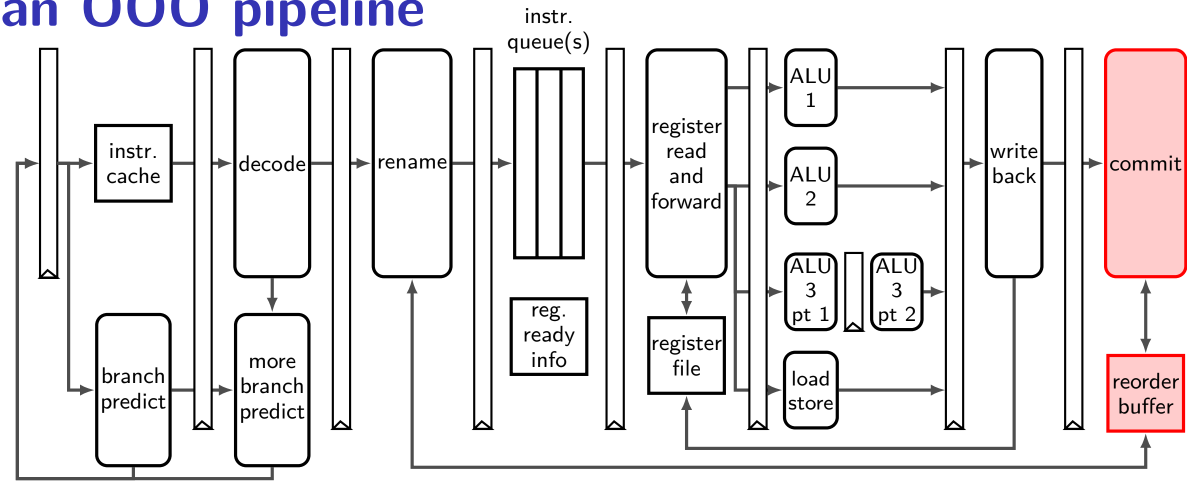
a data flow example

```
addq %rax, %rbx
addq %rax, %rcx
imulq %rdx, %rcx
movq (%rbx, %rdx), %r8
imulq %r8, %rcx
addq %rax, %rbx
```



addq, compute addr: 1 cycle
imulq: 3 cycle latency
load: 3 cycle latency
Q1: latency bound on cycles?
Q2: what can be done
at same time as compute addr?

an OOO pipeline



reorder buffer: on rename

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
...	...

free list

%x19
%x23
...
...

reorder buffer: on rename

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
…	…

free list

%x19
%x23
…
…

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		

reorder buffer contains instructions started,
but not fully finished new entries created on rename
(not enough space? stall rename stage)

reorder buffer: on rename

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
…	…

free list

%x19
%x23
…
…

remove here when committed →

add here on rename →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		

place newly started instruction at end of buffer
remember at least its destination register
(both architectural and physical versions)

reorder buffer: on rename

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x23
…
…

remove here when committed →

add here on rename →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		
32	0x1230	%rdx / %x19		

next renamed instruction goes in next slot, etc.

reorder buffer: on rename

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x23
…
…

remove here when committed →

add here on rename →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		
32	0x1230	%rdx / %x19		

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
…

remove here when committed →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
…

remove here when committed →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓

instructions marked done in reorder buffer
when result is computed
but not removed from reorder buffer ('committed') yet

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
…

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	%x23
%rdx	%x21
…	…

remove here
when committed →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓

commit stage tracks architectural to physical register map
for committed instructions

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
%x23

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	~~%x23~~ %x24
%rdx	%x21
…	…

remove here
when committed →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24	✓	
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓
32	0x1230	%rdx / %x19		

when next-to-commit instruction is done
update this register map and free register list
and remove instr. from reorder buffer

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
%x23

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	~~%x23~~ %x24
%rdx	%x21
…	…

remove here
when committed →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	~~✓~~	
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓
32	0x1230	%rdx / %x19		

when next-to-commit instruction is done
update this register map and free register list
and remove instr. from reorder buffer

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x19
...	...

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
...	...

free list

~~%x19~~
%x13
...
...

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	~~✓~~	
~~15~~	~~0x1239~~	~~%rax / %x30~~	~~✓~~	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	~~✓~~	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	~~✓~~	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	~~✓~~	
~~19~~	~~0x1249~~	~~%rax / %x38~~	~~✓~~	
→ 20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
...	...	...	...	...
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x19
…	…

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
…	…

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	~~✓~~	
~~15~~	~~0x1239~~	~~%rax / %x30~~	~~✓~~	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	~~✓~~	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	~~✓~~	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	~~✓~~	
~~19~~	~~0x1249~~	~~%rax / %x38~~	~~✓~~	
→ 20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

when committing a mispredicted instruction…
this is where we undo mispredicted instructions

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x38
%rcx	%x32
%rbx	%x24
%rdx	%x34
…	…

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
…	…

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	~~✓~~	
~~15~~	~~0x1239~~	~~%rax / %x30~~	~~✓~~	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	~~✓~~	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	~~✓~~	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	~~✓~~	
~~19~~	~~0x1249~~	~~%rax / %x38~~	~~✓~~	
20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

copy commit register map into rename register map
so we can start fetching from the correct PC

reorder buffer: commit mispredict (one way)

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x38
%rcx	%x32
%rbx	%x24
%rdx	%x34
…	…

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
…	…

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

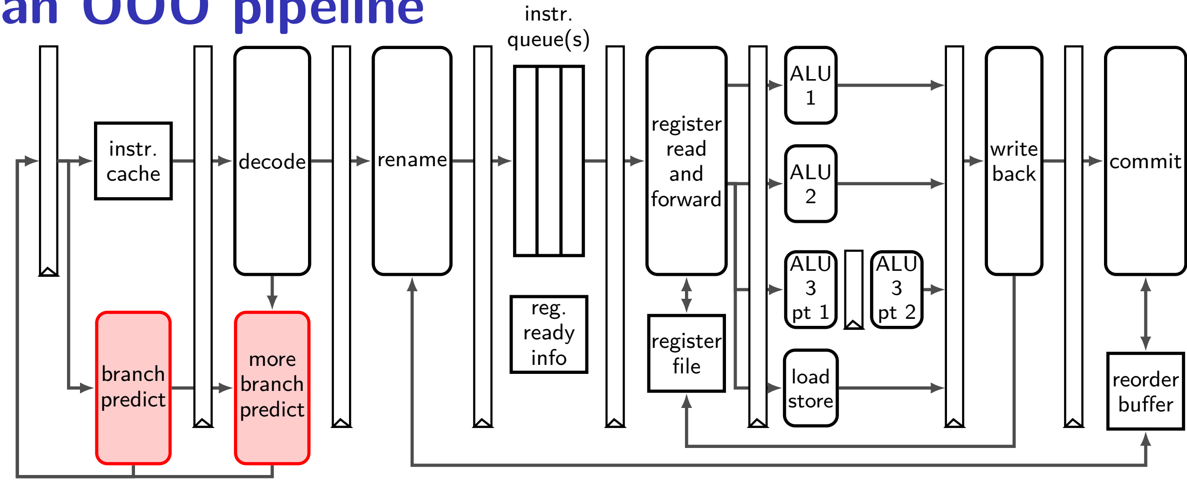
instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	~~✓~~	
~~15~~	~~0x1239~~	~~%rax / %x30~~	~~✓~~	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	~~✓~~	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	~~✓~~	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	~~✓~~	
~~19~~	~~0x1249~~	~~%rax / %x38~~	~~✓~~	
20	0x1254	PC	✓	✓
~~21~~	~~0x1260~~	~~%rcx / %x17~~		
~~…~~	~~…~~	~~…~~	~~…~~	~~…~~
~~31~~	~~0x129f~~	~~%rax / %x12~~	~~✓~~	
~~32~~	~~0x1230~~	~~%rdx / %x19~~		

…and discard all the mispredicted instructions (without committing them)

better? alternatives

- can take snapshots of register map on each branch
  - don't need to reconstruct the table
  - (but how to efficiently store them)
- can reconstruct register map before we commit the branch instruction
  - need to let reorder buffer be accessed even more?
- can track more/different information in reorder buffer

an OOO pipeline



branch target buffer

can take several cycles to fetch+decode jumps, calls, returns

still want 1-cycle prediction of next thing to fetch

BTB: cache for branches

idx	valid	tag	ofst	type	target	(more info?)
0x00	1	0x400	5	Jxx	0x3FFFF3	…
0x01	1	0x401	C	JMP	0x401035	---
0x02	0	---	---	---	---	---
0x03	1	0x400	9	RET	---	…
…	…	…	…	…	…	…
0xFF	1	0x3FF	8	CALL	0x404033	…

valid	…
1	…
0	…
0	…
0	…
…	…
0	…

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
…          …
0x400031:  ret
…          …
```

BTB: cache for branches

idx	valid	tag	ofst	type	target	(more info?)
0x00	1	0x400	5	Jxx	0x3FFFF3	…
0x01	1	0x401	C	JMP	0x401035	---
0x02	0	---	---	---	---	---
0x03	1	0x400	9	RET	---	…
…	…	…	…	…	…	…
0xFF	1	0x3FF	8	CALL	0x404033	…

valid	…
1	…
0	…
0	…
0	…
…	…
0	…

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
…          …
0x400031:  ret
…          …
```

BTB: cache for branches

idx	valid	tag	ofst	type	target	(more info?)
0x00	1	0x400	5	Jxx	0x3FFFF3	…
0x01	1	0x401	C	JMP	0x401035	---
0x02	0	---	---	---	---	---
0x03	1	0x400	9	RET	---	…
…	…	…	…	…	…	…
0xFF	1	0x3FF	8	CALL	0x404033	…

valid	…
1	…
0	…
0	…
0	…
…	…
0	…

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
…          …
0x400031:  ret
…          …
```

aside on branch pred. and performance

modern branch predictors are very good

- we might explore how later in semester (if time)

…usually can assume most branches will be predicted

but could be a problem if really no pattern

- e.g. branch based on random number?

generally: measure and see

if branch prediction is bad...

avoiding branches — conditional move, etc.

replace multiple branches with single lookup?

- one misprediction better than K?

recall: shifts

we mentioned that compilers compile $x/4$ into a shift instruction

they are really good at these types of of transformation…

- "strength reduction": replacing complicated op with simpler one

but can't do without seeing special case (e.g. divide by constant)

Intel Skylake OOO design

2015 Intel design — codename ‘Skylake’

94-entry instruction queue-equivalent

168 physical integer registers

168 physical floating point registers

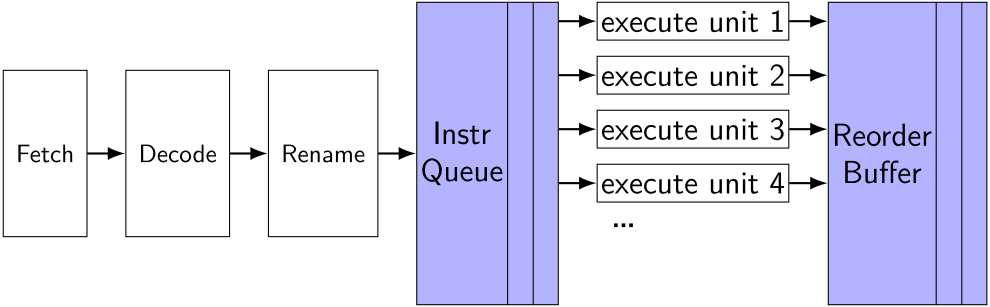
4 ALU functional units
- but some can handle more/different types of operations than others

2 load functional units
- but pipelined: supports multiple pending cache misses in parallel

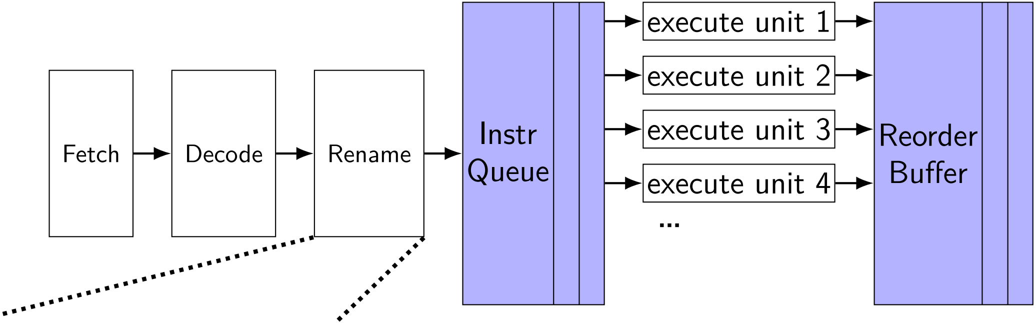
1 store functional unit

224-entry reorder buffer
- determines how far ahead branch mispredictions, etc. can happen

exceptions and OOO (one strategy)



exceptions and OOO (one strategy)



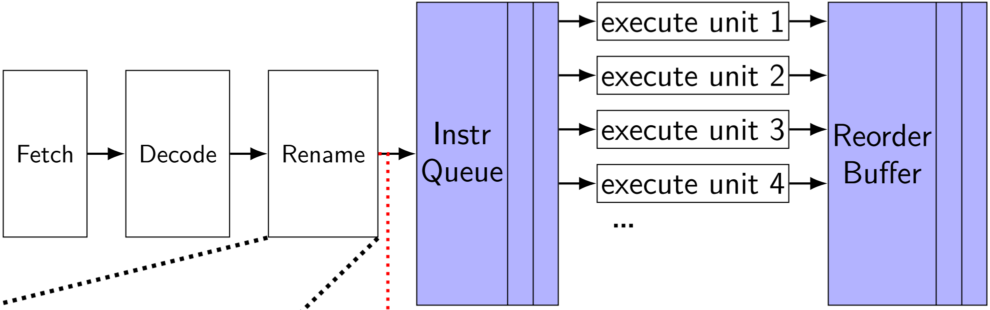
free regs

X19
X23
...

for new instrs

arch. reg	**phys. reg**
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

exceptions and OOO (one strategy)



free regs

X19
X23
…

for new instrs

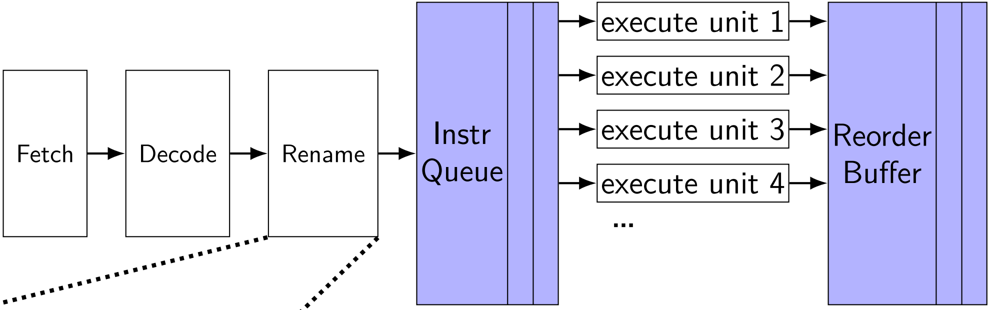
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

done instrs committed in order

new instrs added

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
17	0x1244	RCX / X32		
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



free regs

X19
X23
…

for new instrs

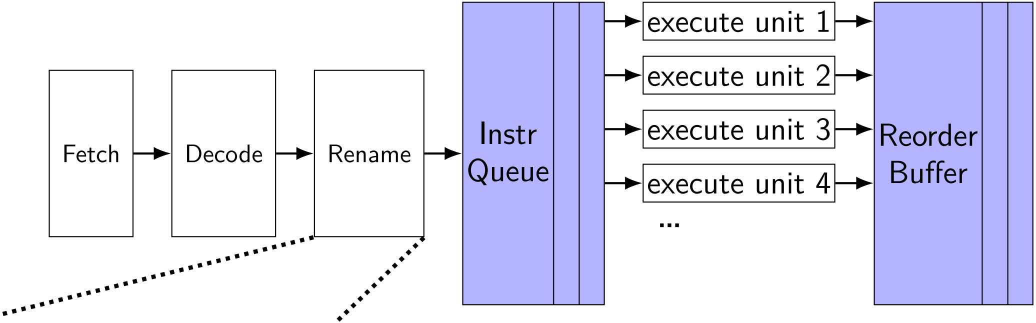
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

arch. reg	phys. reg
RAX	X21
RCX	~~X2~~ X32
RBX	X48
RDX	X37
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

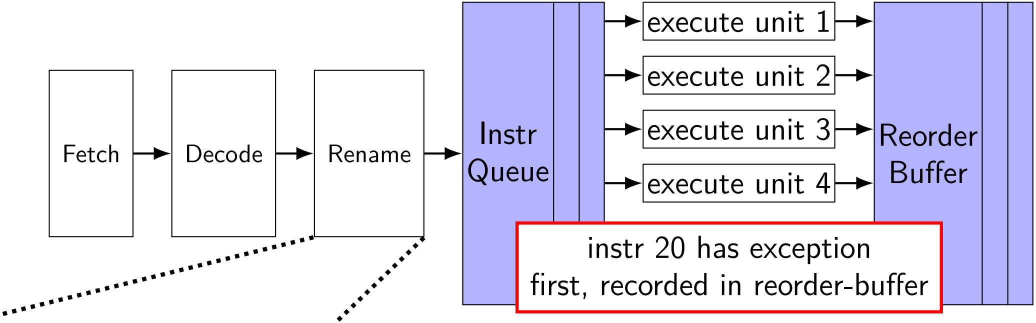
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	X21
RCX	~~X2~~ X32
RBX	X48
RDX	X37
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



free regs

X19
X23
…

for new instrs

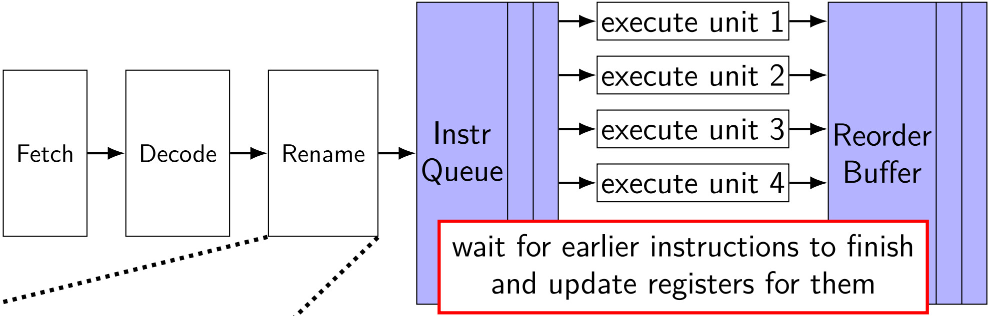
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

arch. reg	phys. reg
RAX	X21
RCX	~~X2~~ X32
RBX	X48
RDX	X37
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

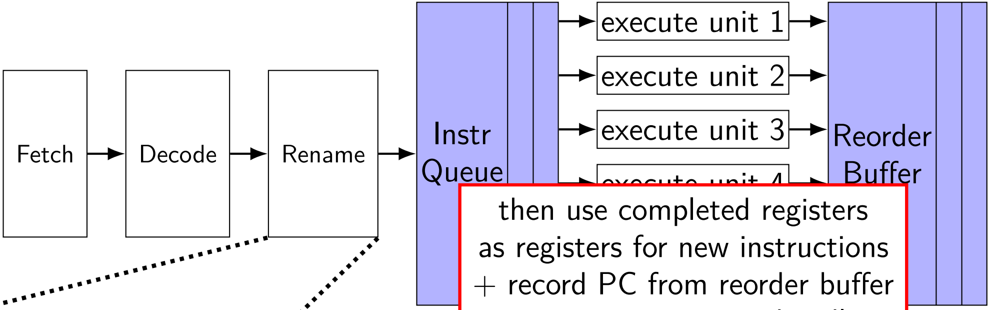
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	✓	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



then use completed registers
as registers for new instructions
+ record PC from reorder buffer
+ jump to exception handler

free regs

X19
X23
…

for new instrs

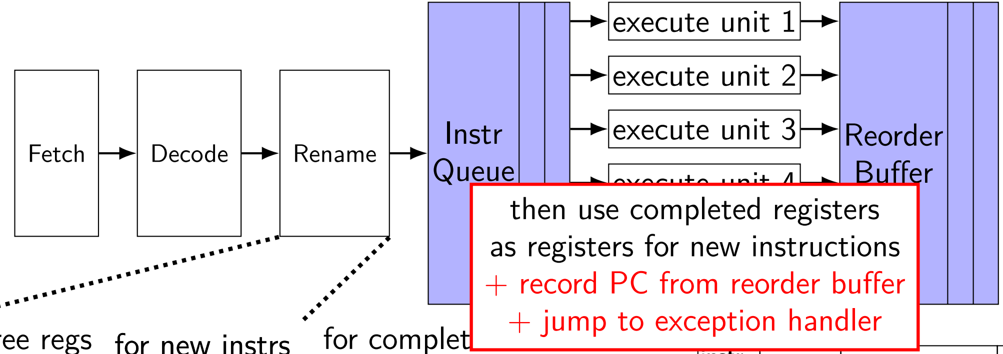
arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
…	…

for complet…

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	✓	
~~18~~	~~0x1248~~	~~RDX / X34~~	✓	
~~19~~	~~0x1249~~	~~RAX / X38~~	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



free regs

X19
X23
…

for new instrs

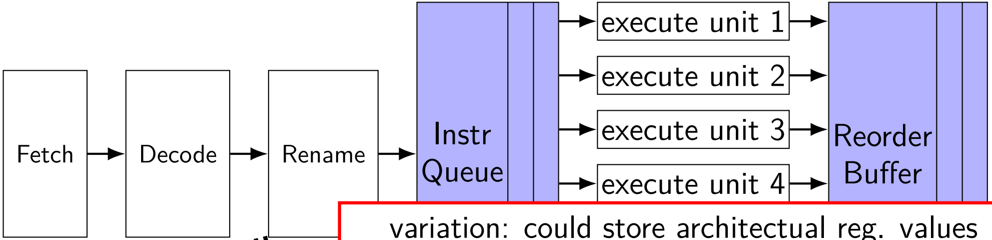
arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
…	…

for complet

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
~~18~~	~~0x1248~~	~~RDX / X34~~	~~✓~~	
~~19~~	~~0x1249~~	~~RAX / X38~~	~~✓~~	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



variation: could store architectual reg. values instead of mapping for completed instrs.
(and copy values instead of mapping on exception)

free regs

X19
X23
…

for new instrs

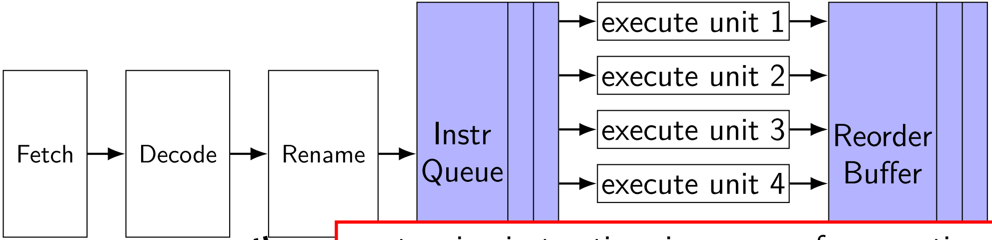
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

arch. reg	value
RAX	0x12343
RCX	0x234543
RBX	0x56782
RDX	0xF83A4
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



stopping instructions in progress for exception
similar to how 'squashing' mispredicted instructions

free regs

X19
X23
...

for new instrs

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

addressing efficiency

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float Cij = C[i * N + j];
      for (int k = kk; k < kk + 2; ++k) {
        Cij += A[i * N + k] * B[k * N + j];
      }
      C[i * N + j] = Cij;
    }
  }
}
```

tons of multiplies by N??

isn't that slow?

addressing transformation

```
for (int kk = 0; k < N; kk += 2)
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float Cij = C[i * N + j];
      float *Bkj_pointer = &B[kk * N + j];
      for (int k = kk; k < kk + 2; ++k) {
        // Bij += A[i * N + k] * A[k * N + j~];
        Bij += A[i * N + k] * Bkj_pointer;
        Bkj_pointer += N;
      }
      C[i * N + j] = Bij;
    }
  }
```

transforms loop to iterate with pointer

compiler will often do this

increment/decrement by N ($\times$ sizeof(float))

addressing transformation

```
for (int kk = 0; k < N; kk += 2)
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float Cij = C[i * N + j];
      float *Bkj_pointer = &B[kk * N + j];
      for (int k = kk; k < kk + 2; ++k) {
        // Bij += A[i * N + k] * A[k * N + j~];
        Bij += A[i * N + k] * Bkj_pointer;
        Bkj_pointer += N;
      }
      C[i * N + j] = Bij;
    }
  }
```

transforms loop to iterate with pointer

compiler will often do this

increment/decrement by N ($\times$ sizeof(float))

addressing efficiency

compiler will usually eliminate slow multiplies

doing transformation yourself often slower if so

`i * N; ++i` into `i_times_N; i_times_N += N`

way to check: see if assembly uses lots multiplies in loop

if it doesn't — do it yourself

another addressing transformation

```
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
int offset = 0;
float *Ai0_base = &A[k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += Ai0_base[offset] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[offset] * B[k * n + j];
    // ...
    offset += n;
```

compiler will sometimes do this, too

another addressing transformation

```
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
int offset = 0;
float *Ai0_base = &A[k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += Ai0_base[offset] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[offset] * B[k * n + j];
    // ...
    offset += n;
```

compiler will sometimes do this, too

another addressing transformation

```
    for (int i = 0; i < n; i += 20) {
        C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
        C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
        // ...
```

```
    int offset = 0;
    float *Ai0_base = &A[0*n+k];
    float *Ai1_base = Ai0_base + n;
    float *Ai2_base = Ai1_base + n;
    // ...
    for (int i = 0; i < n; i += 20) {
        C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
        C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
        // ...
        offset += n;
```

storing 20 `AiX_base`? — need the stack

maybe faster (quicker address computation)

maybe slower (can't do enough loads)

another addressing transformation

```
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
int offset = 0;
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
    offset += n;
```

storing 20 `AiX_base`? — need the stack

maybe faster (quicker address computation)

maybe slower (can't do enough loads)

alternative addressing transformation

instead of:

```
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

---

could do:

```
float *Ai0_base = &A[k];
for (int i = 0; i < n; i += 20) {
    float *A_ptr = &Ai0_base[i*n];
    C[(i+0) * n + j] += *A_ptr * A[k * n + j];
    A_ptr += n;
    C[(i+1) * n + j] += *A_ptr * B[k * n + j];
    // ...
```

avoids spilling on the stack, but more dependencies

alternative addressing transformation

instead of:

```
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

---

could do:

```
float *Ai0_base = &A[k];
for (int i = 0; i < n; i += 20) {
    float *A_ptr = &Ai0_base[i*n];
    C[(i+0) * n + j] += *A_ptr * A[k * n + j];
    A_ptr += n;
    C[(i+1) * n + j] += *A_ptr * B[k * n + j];
    // ...
```

avoids spilling on the stack, but more dependencies

addressing efficiency generally

mostly: compiler does very good job itself

- eliminates multiplications, use pointer arithmetic
- often will do better job than if how typically programming would do it manually

sometimes compiler won't take the best option

- if spilling to the stack: can cause weird performance anomalies
- if indexing gets too complicated — might not remove multiply

if compiler doesn't, you can always make addressing simple yourself

- convert to pointer arith. without multiplies