Performance

Changelog

Changes made in this version not seen in first lecture:

26 October 2017: slide 28: remove extraneous text from code

locality exercise (1)

```
/* version 1 */
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        A[i] += B[j] * C[i * N + j]

/* version 2 */
for (int j = 0; j < N; ++j)
    for (int i = 0; i < N; ++i)
        A[i] += B[j] * C[i * N + j];
```

exercise: which has better temporal locality in A? in B? in C?

how about spatial locality?

a transformation

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

split the loop over k — should be exactly the same

(assuming even N)

a transformation

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

split the loop over k — should be exactly the same

(assuming even N)

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
  /* was here: for (int k = kk; k < kk + 2; ++k) */
    for (int i = 0; i < N; i += 2)
      for (int j = 0; j < N; ++j)
        /* load Aik, Aik+1 into cache and process: */
        for (int k = kk; k < kk + 2; ++k)
            B[i*N+j] += A[i*N+k] * A[k*N+j];
```

now reorder split loop — same calculations

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
  /* was here: for (int k = kk; k < kk + 2; ++k) */
    for (int i = 0; i < N; i += 2)
      for (int j = 0; j < N; ++j)
        /* load Aik, Aik+1 into cache and process: */
        for (int k = kk; k < kk + 2; ++k)
          B[i*N+j] += A[i*N+k] * A[k*N+j];
```

now reorder split loop — same calculations

now handle B_{ij} for $k+1$ right after B_{ij} for k

(previously: $B_{i,j+1}$ for k right after B_{ij} for k)

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
  /* was here: for (int k = kk; k < kk + 2; ++k) */
    for (int i = 0; i < N; i += 2)
      for (int j = 0; j < N; ++j)
        /* load Aik, Aik+1 into cache and process: */
        for (int k = kk; k < kk + 2; ++k)
            B[i*N+j] += A[i*N+k] * A[k*N+j];
```

now reorder split loop — same calculations

now handle B_{ij} for $k+1$ right after B_{ij} for k

(previously: $B_{i,j+1}$ for k right after B_{ij} for k)

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      B[i*N+j] += A[i*N+kk+0] * A[(kk+0)*N+j];
      B[i*N+j] += A[i*N+kk+1] * A[(kk+1)*N+j];
    }
  }
}
```

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      B[i*N+j] += A[i*N+kk+0] * A[(kk+0)*N+j];
      B[i*N+j] += A[i*N+kk+1] * A[(kk+1)*N+j];
    }
  }
}
```

Temporal locality in B_{ij}s

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      B[i*N+j] += A[i*N+kk+0] * A[(kk+0)*N+j];
      B[i*N+j] += A[i*N+kk+1] * A[(kk+1)*N+j];
    }
  }
}
```

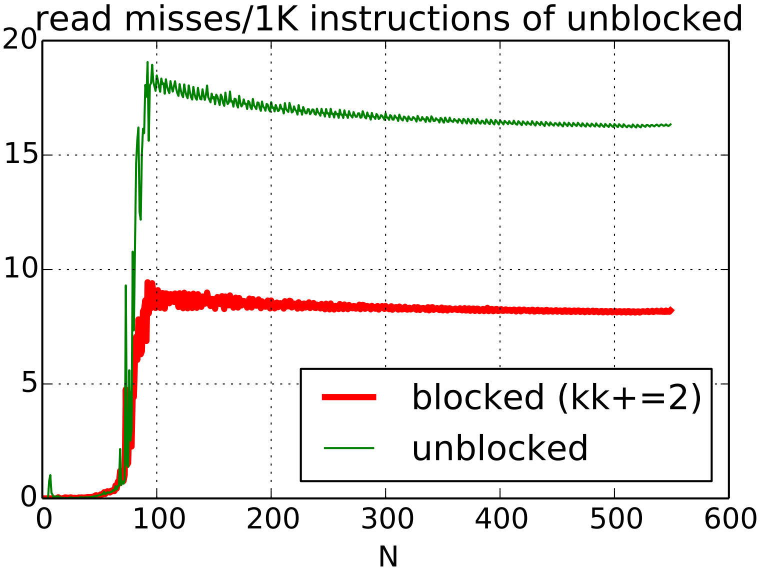
More spatial locality in A_{ik}

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      B[i*N+j] += A[i*N+kk+0] * A[(kk+0)*N+j];
      B[i*N+j] += A[i*N+kk+1] * A[(kk+1)*N+j];
    }
  }
}
```

Still have good spatial locality in A_{kj}, B_{ij}

improvement in read misses



simple blocking (2)

same thing for i in addition to k?

```
for (int kk = 0; kk < N; kk += 2) {
  for (int ii = 0; ii < N; ii += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
      for (int k = kk; k < kk + 2; ++k)
        for (int i = 0; i < ii + 2; ++i)
            B[i*N+j] += A[i*N+k] * A[k*N+j];
    }
  }
}
```

simple blocking — expanded

```
for (int k = 0; k < N; k += 2) {
  for (int i = 0; i < N; i += 2) {
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
      B_{i+0,j} += A_{i+0,k+0} * A_{k+0,j}
      B_{i+0,j} += A_{i+0,k+1} * A_{k+1,j}
      B_{i+1,j} += A_{i+1,k+0} * A_{k+0,j}
      B_{i+1,j} += A_{i+1,k+1} * A_{k+1,j}
    }
  }
}
```

simple blocking — expanded

```
for (int k = 0; k < N; k += 2) {
  for (int i = 0; i < N; i += 2) {
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
      B_{i+0,j} += A_{i+0,k+0} * A_{k+0,j}
      B_{i+0,j} += A_{i+0,k+1} * A_{k+1,j}
      B_{i+1,j} += A_{i+1,k+0} * A_{k+0,j}
      B_{i+1,j} += A_{i+1,k+1} * A_{k+1,j}
    }
  }
}
```

Now A_{kj} reused in inner loop — more calculations per load!

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                           * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                          * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                          * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                         * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

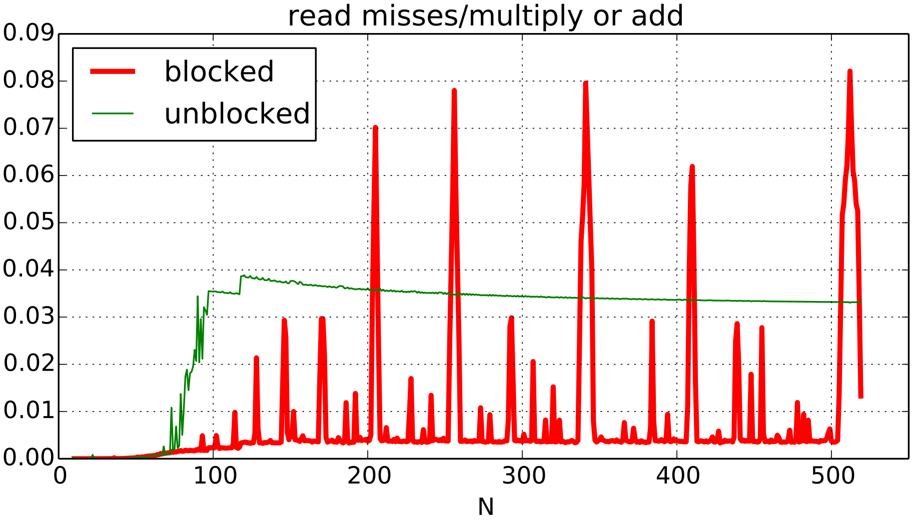
A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

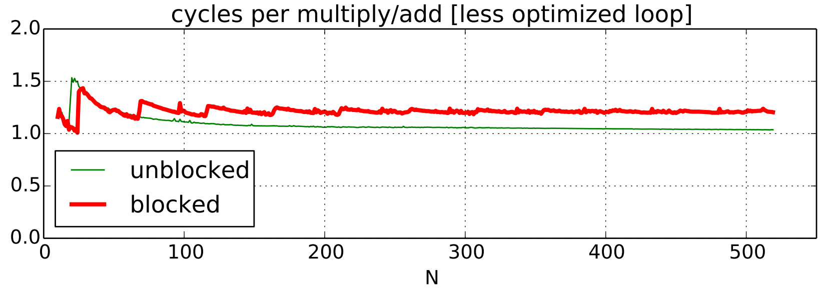
view 2: divide and conquer

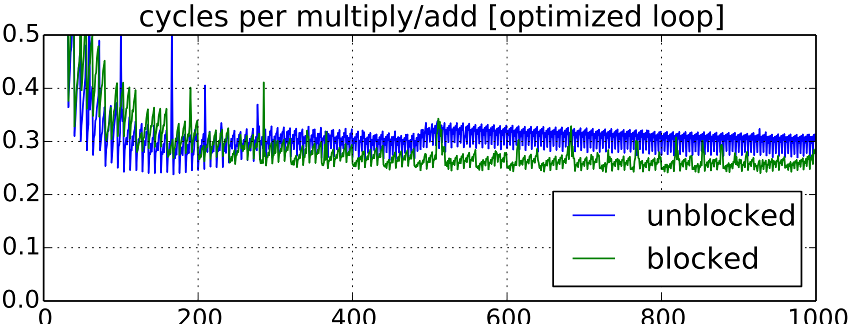
```
partial_square(float *A, float *B,
               int startI, int endI, ...) {
  for (int i = startI; i < endI; ++i) {
    for (int j = startJ; j < endJ; ++j) {
      ...
}
square(float *A, float *B, int N) {
  for (int ii = 0; ii < N; ii += BLOCK)
    ...
      /* segment of A, B in use fits in cache! */
      partial_square(
            A, B,
            ii, ii + BLOCK,
            jj, jj + BLOCK, ...);
}
```

cache blocking and miss rate

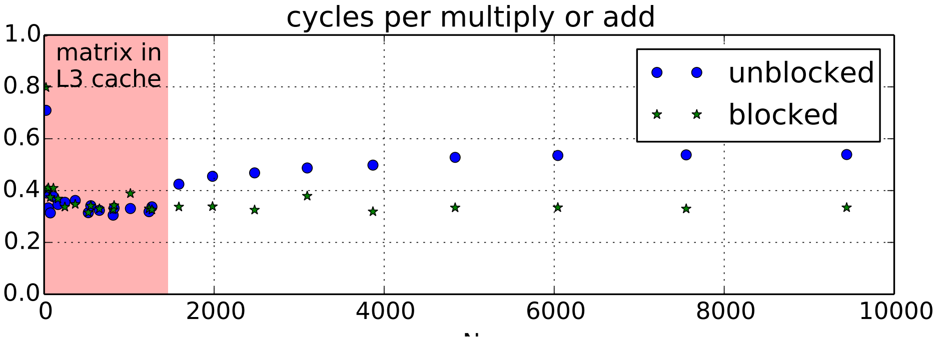


what about performance?





performance for big sizes



optimized loop???

performance difference wasn't visible at small sizes

until I optimized arithmetic in the loop

(mostly by supplying better options to GCC)

1: reducing number of loads

2: doing adds/multiplies/etc. with less instructions

3: simplifying address computations

optimized loop???

performance difference wasn't visible at small sizes

until I optimized arithmetic in the loop

(mostly by supplying better options to GCC)

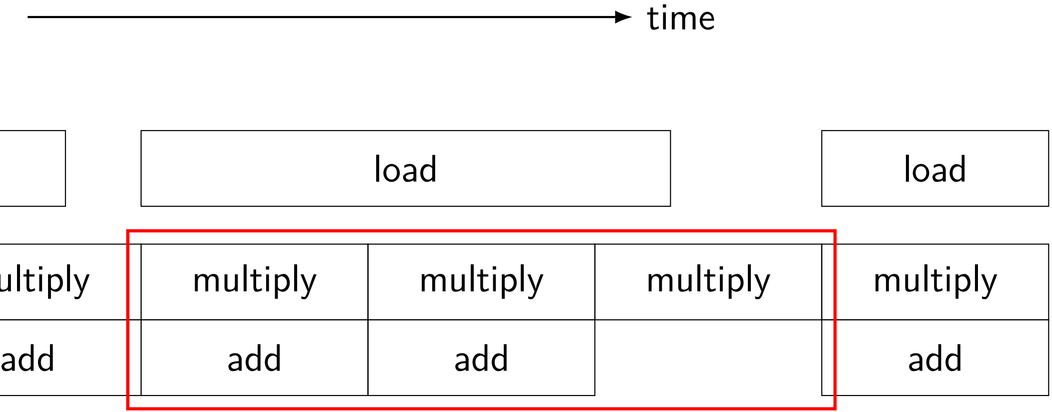
1: reducing number of loads

2: doing adds/multiplies/etc. with less instructions

3: simplifying address computations

but… how can that make cache blocking better???

overlapping loads and arithmetic



speed of load might not matter if these are slower

optimization and bottlenecks

arithmetic/loop efficiency was the bottleneck

after fixing this, cache performance was the bottleneck

common theme when optimizing:

X may not matter until Y is optimized

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:    ...
```

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

```c
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];
    }
}
```

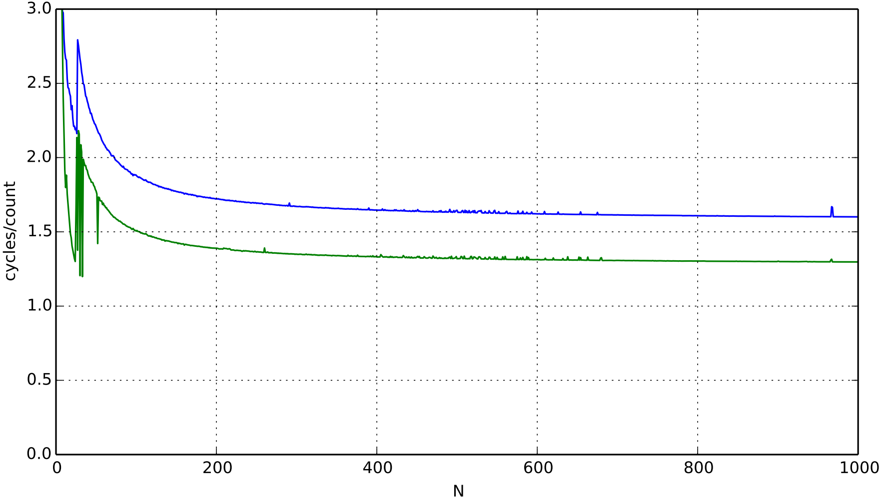
```c
void sumRows2(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        int sum = 0;
        for (int col = 0; col < N; ++col)
            sum += matrix[row * N + col];
        result[row] = sum;
    }
}
```

non-contrived aliasing

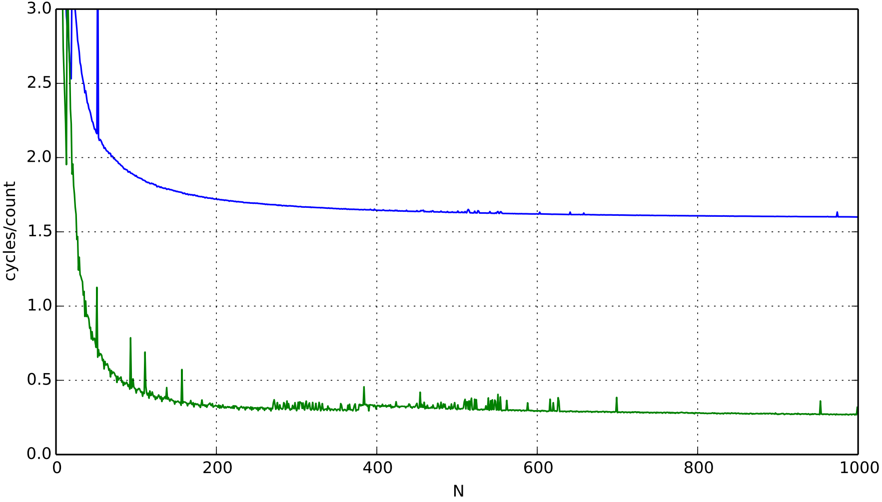
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

```
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 and cache optimizations

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

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

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

compiler can't generate same code for both

“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 Akj0 = A[k*N + j+0];
      float Akj1 = A[k*N + j+1];
      B[(i+0)*N + j+0] += Ai0k * Akj0;
      B[(i+1)*N + j+0] += Ai1k * Akj0;
      B[(i+0)*N + j+1] += Ai0k * Akj1;
      B[(i+1)*N + j+1] += Ai1k * Akj1;
    }
  }
}
```

avoiding redundant loads summary

move repeated load outside of loop

create variable — tell compiler "not aliased"

aside: the restrict hint

C has a keyword '`restrict`' for pointers

"I promise this pointer doesn't alias another"

(if it does — undefined behavior)

maybe will help compiler do optimization itself?

```
void square(float * restrict B, float * restrict A) {
    ...
}
```

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

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

```
movl (%rbx), %esi // mov array[i]
movl %eax, %edi   // mov sum
call addWithLimit
```

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

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”

remove redundant operations (1)

```
char 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

loop unrolling (ASM)

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        incq    %rdx
        jmp
endOfLoop:
```

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        addq    8(%rdi,%rdx,8), %rax
        addq    $2, %rdx
        jmp     loop
        // plus handle leftover?
endOfLoop:
```

loop unrolling (ASM)

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        incq    %rdx
        jmp
endOfLoop:
```

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        addq    8(%rdi,%rdx,8), %rax
        addq    $2, %rdx
        jmp     loop
        // plus handle leftover?
endOfLoop:
```

loop unrolling (C)

```c
for (int i = 0; i < N; ++i)
    sum += A[i];
```

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

more loop unrolling (C)

```
int i;
for (i = 0; i + 4 <= N; i += 4) {
    sum += A[i];
    sum += A[i+1];
    sum += A[i+2];
    sum += A[i+3];
}
// handle leftover, if needed
for (; i < N; i += 1)
    sum += A[i];
```

automatic loop unrolling

loop unrolling is easy for compilers

...but often not done or done very much

why not?

automatic loop unrolling

loop unrolling is easy for compilers

...but often not done or done very much

why not?

slower if small number of iterations

larger code — could exceed instruction cache space

loop unrolling performance

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

times unrolled	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

instruction cache/etc. overhead

1.01 cycles/element — latency bound

interlude: real CPUs

modern CPUs:

execute multiple instructions at once

execute instructions out of order — whenever values available

beyond pipelining: out-of-order

find later instructions to do instead of stalling

lists of available instructions in pipeline registers

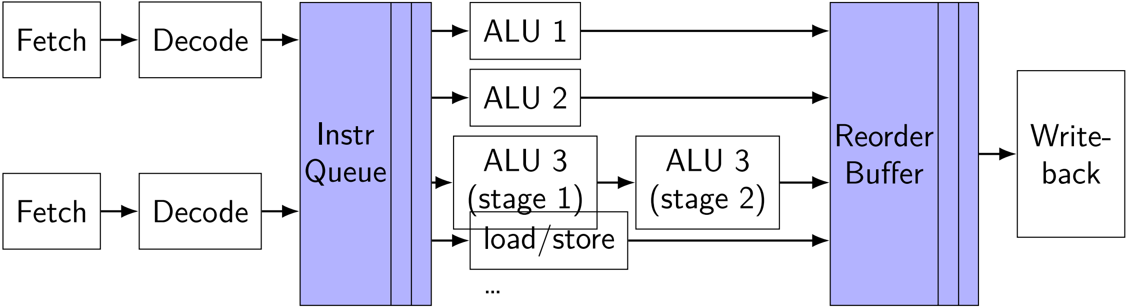
- take any instruction with available values

provide illusion that work is still done in order

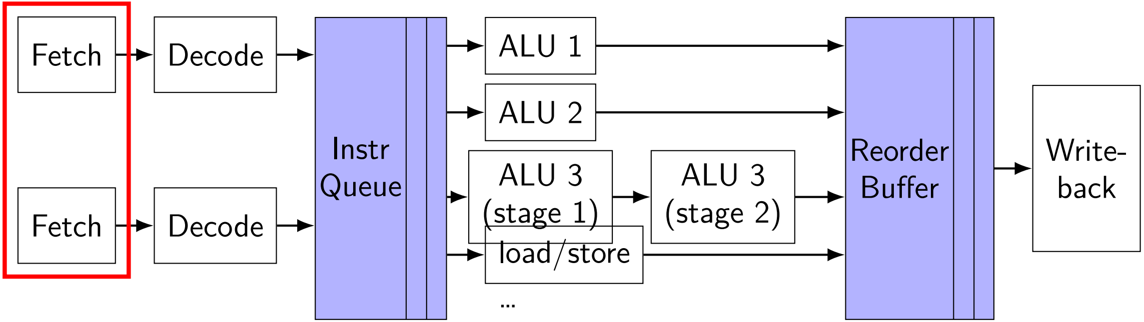
- much more complicated hazard handling logic

cycle #	0	1	2	3	4	5	6	7	8		
`mrmovq 0(%rbx), %r8`	F	D	E	M	M	M	W				
`subq %r8, %r9`		F					D	E	W		
`addq %r10, %r11`			F	D	E					W	
`xorq %r12, %r13`				F	D	E					W
…											

modern CPU design (instruction flow)

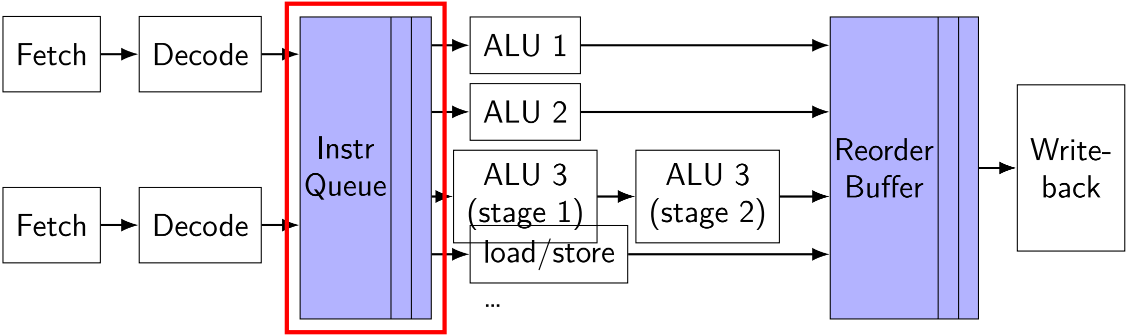


modern CPU design (instruction flow)



fetch multiple instructions/cycle

modern CPU design (instruction flow)

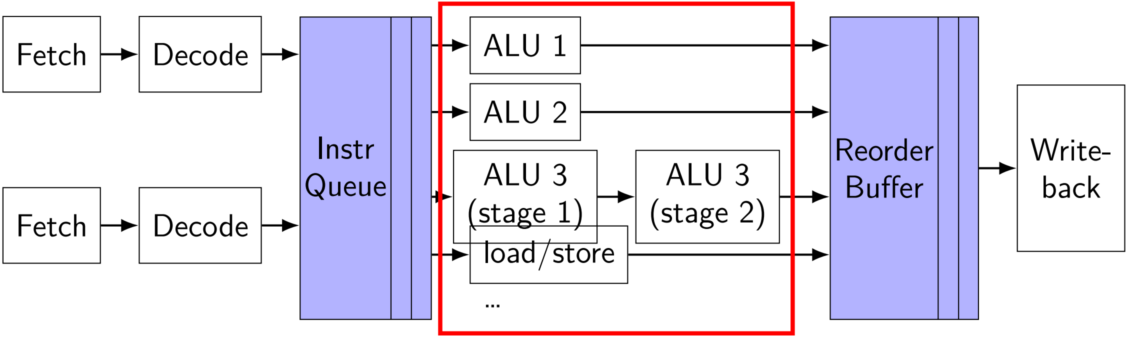


keep list of pending instructions

run instructions from list when operands available

forwarding handled here

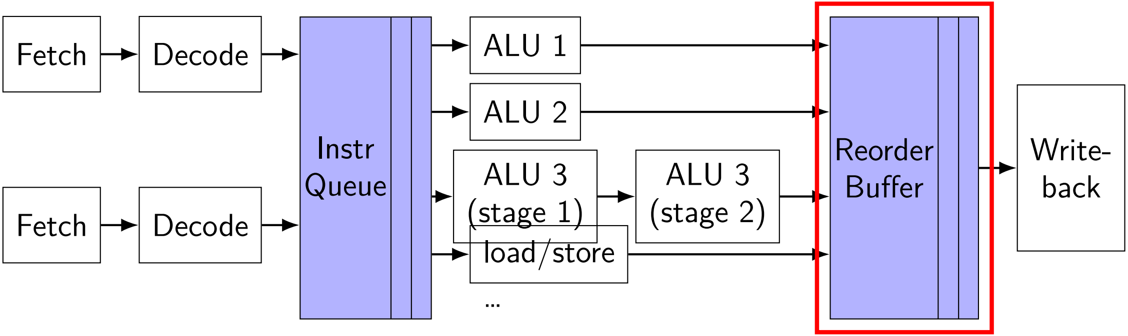
modern CPU design (instruction flow)



multiple "execution units" to run instructions
e.g. possibly many ALUs

sometimes pipelined, sometimes not

modern CPU design (instruction flow)



collect results of finished instructions

helps with forwarding, squashing

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	ready
2	`addq %rbx, %rdx`	waiting for 1
3	`addq %rcx, %rdx`	waiting for 2
4	`cmpq %r8, %rdx`	waiting for 3
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	waiting for 3
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

execution unit	...
ALU 1	
ALU 2	

instruction queue operation

#	instruction	status
1	`addq %rax, %rdx`	*running*
2	`addq %rbx, %rdx`	waiting for 1
3	`addq %rcx, %rdx`	waiting for 2
4	`cmpq %r8, %rdx`	waiting for 3
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	waiting for 3
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

execution unit	*cycle# 1*	...
ALU 1	**1**	
ALU 2	—	

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	ready
3	`addq %rcx, %rdx`	waiting for 2
4	`cmpq %r8, %rdx`	waiting for 3
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	waiting for 3
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
…	…	

execution unit	*cycle# 1*	…
ALU 1	1	
ALU 2	—	

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	*running*
3	`addq %rcx, %rdx`	waiting for 2
4	`cmpq %r8, %rdx`	waiting for 3
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	waiting for 3
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
…	…	

execution unit	*cycle# 1*	*2*	…
ALU 1	1	**2**	
ALU 2	—	—	

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	*running*
4	`cmpq %r8, %rdx`	waiting for 3
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	waiting for 3
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

execution unit	*cycle# 1*	*2*	*3*	...
ALU 1	1	2	**3**	
ALU 2	—	—	—	

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	ready
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	ready
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

execution unit	*cycle# 1*	*2*	*3*	...
ALU 1	1	2	3	
ALU 2	—	—	—	

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	*running*
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	*running*
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8

... ...

execution unit	*cycle# 1*	*2*	*3*	*4*	...
ALU 1	1	2	3	**4**	
ALU 2	—	—	—	**6**	

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	ready
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	ready
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
…	…	

execution unit	*cycle# 1*	*2*	*3*	*4*	…
ALU 1	1	2	3	4	
ALU 2	—	—	—	6	

instruction queue operation

#	instruction	status
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	*running*
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8

... ...

execution unit	*cycle# 1*	*2*	*3*	*4*	*5*	...
ALU 1	1	2	3	4	**5**	
ALU 2	—	—	—	6	**7**	

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	*running*
9	`cmpq %r8, %rdx`	waiting for 8

... ...

execution unit	*cycle# 1*	*2*	*3*	*4*	*5*	*6*	...
ALU 1	1	2	3	4	5	**8**	
ALU 2	—	—	—	6	7	—	

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	done
9	`cmpq %r8, %rdx`	*running*
…	…	

execution unit	*cycle# 1*	*2*	*3*	*4*	*5*	*6*	*7*	…
ALU 1	1	2	3	4	5	8	**9**	
ALU 2	—	—	—	6	7	—	**…**	

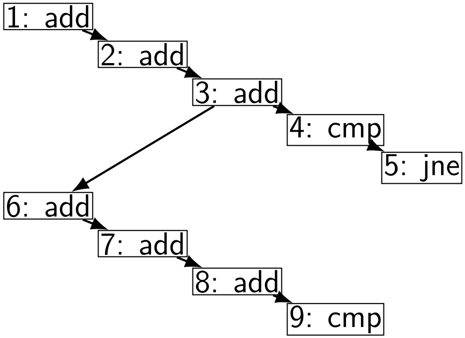
instruction queue operation

#	instruction	status
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	done
9	`cmpq %r8, %rdx`	done
…	…	

execution unit	*cycle# 1*	*2*	*3*	*4*	*5*	*6*	*7*	…
ALU 1	1	2	3	4	5	8	9	
ALU 2	—	—	—	6	7	—	…	

data flow

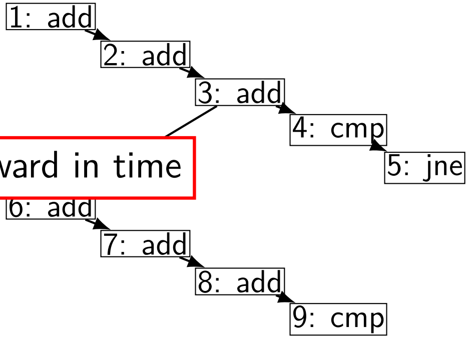
#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	done
9	`cmpq %r8, %rdx`	done
…	…	



execution unit	*cycle# 1*	*2*	*3*	*4*	*5*	*6*	*7*	…
ALU 1	1	2	3	4	5	8	9	
ALU 2	—	—	—	6	7	—	…	

data flow

#	instruction	status
1	addq %rax, %rdx	done
2	addq %rbx, %rdx	done
3	addq %rcx, %rdx	done
4	cmpq %r8, %rdx	done
5	jne ...	
6	addq %rax, %rdx	done
7	addq %rbx, %rdx	done
8	addq %rcx, %rdx	done
9	cmpq %r8, %rdx	done
…	…	



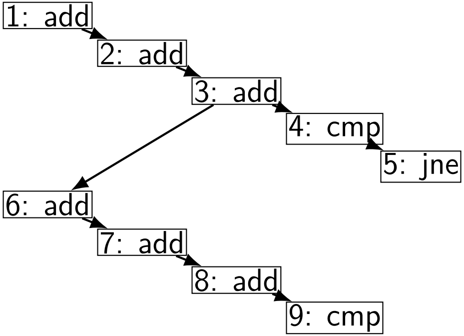
rule: arrows must go forward in time

execution unit	cycle# 1	2	3	4	5	6	7	…
ALU 1	1	2	3	4	5	8	9	
ALU 2	—	—	—	6	7	—	…	

data flow

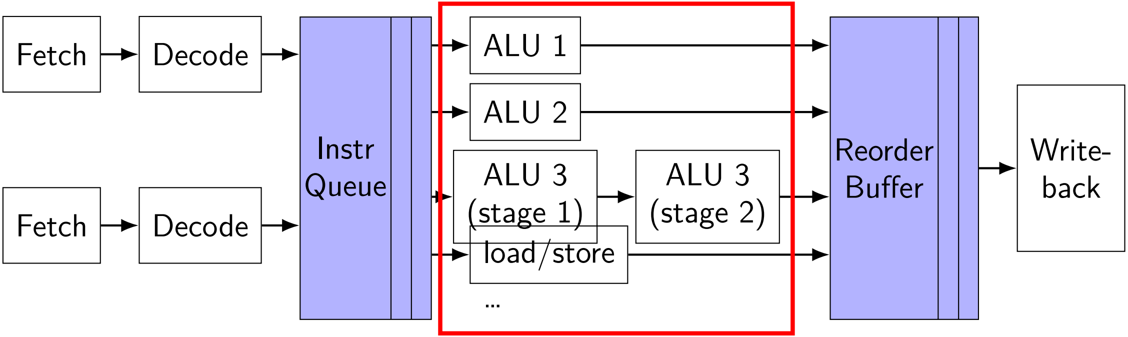
#	instruction	status
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`ad`	
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	done
9	`cmpq %r8, %rdx`	done
...	...	

longest path determines speed



execution unit	cycle# 1	2	3	4	5	6	7	...
ALU 1	1	2	3	4	5	8	9	
ALU 2	—	—	—	6	7	—	...	

modern CPU design (instruction flow)



multiple "execution units" to run instructions
e.g. possibly many ALUs

sometimes pipelined, sometimes not

constant multiplies/divides (1)

```
unsigned int fiveEights(unsigned int x) {
    return x * 5 / 8;
}
```

```
fiveEights:
        leal    (%rdi,%rdi,4), %eax
        shrl    $3, %eax
        ret
```

constant multiplies/divides (2)

```
int oneHundredth(int x) { return x / 100; }
```

```
oneHundredth:
        movl    %edi, %eax
        movl    $1374389535, %edx
        sarl    $31, %edi
        imull   %edx
        sarl    $5, %edx
        movl    %edx, %eax
        subl    %edi, %eax
        ret
```

$$\frac{1374389535}{2^{37}} \approx \frac{1}{100}$$

constant multiplies/divides

compiler is very good at handling

…but need to actually use constants

addressing efficiency

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

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 Bij = B[i * N + j];
      float *Akj_pointer = &A[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] * Akj_pointer;
        Akj_pointer += N;
      }
      B[i * N + j] = Bij;
    }
  }
```

transforms loop to iterate with pointer

compiler will usually 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 Bij = B[i * N + j];
      float *Akj_pointer = &A[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] * Akj_pointer;
        Akj_pointer += N;
      }
      B[i * N + j] = Bij;
    }
  }
```

transforms loop to iterate with pointer

compiler will usually 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

cache blocking ugliness — fringe

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    for (int jj = 0; jj < N; jj += J) {
      for (int k = kk; k < min(kk + K, N) ; ++k) {
        // ...
      }
    }
  }
}
```

cache blocking ugliness — fringe

```
for (kk = 0; kk + K <= N; kk += K) {
  for (ii = 0; ii + I <= N; ii += I) {
    for (jj = 0; jj + J <= N; ii += J) {
      // ...
    }
    for (; jj < N; ++jj) {
      // handle remainder
    }
  }
  for (; ii < N; ++ii) {
    // handle remainder
  }
}
for (; kk < N; ++kk) {
  // handle remainder
}
```

avoiding conflict misses

problem — array is scattered throughout memory

observation: 32KB cache can store 32KB contiguous array

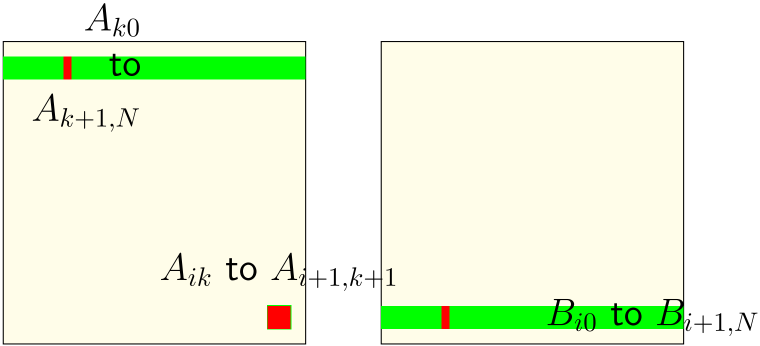
- contiguous array is split evenly among sets

solution: copy block into contiguous array

avoiding conflict misses (code)

```
process_block(ii, jj, kk) {
  float B_copy[I * J];
  /* pseudocode for loop to save space */
  for i = ii to ii + I, j = jj to jj + J:
    B_copy[i * J + j] = B[i * N + j];
  for i = ii to ii + I, j = jj to jj + J, k:
    B_copy[i * J + j] += A[k * N + j] * A[i * N + k];
  for all i, j:
    B[i * N + j] = B_copy[i * J + j];
}
```

array usage (better)



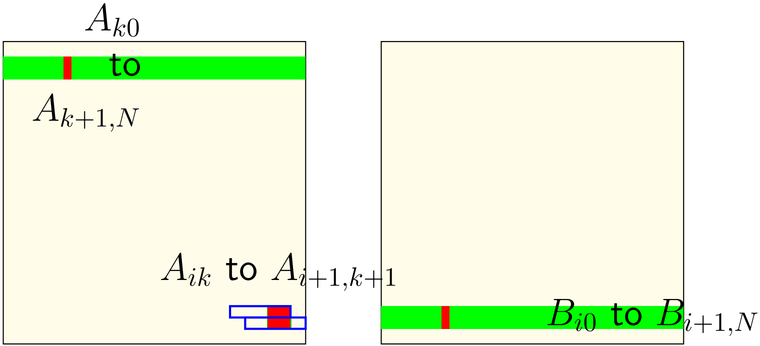
more temporal locality:

N calculations for each A_{ik}

2 calculations for each B_{ij} (for k, $k+1$)

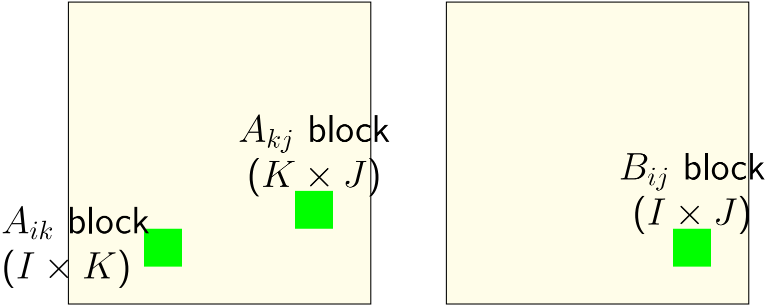
2 calculations for each A_{kj} (for k, $k+1$)

array usage (better)



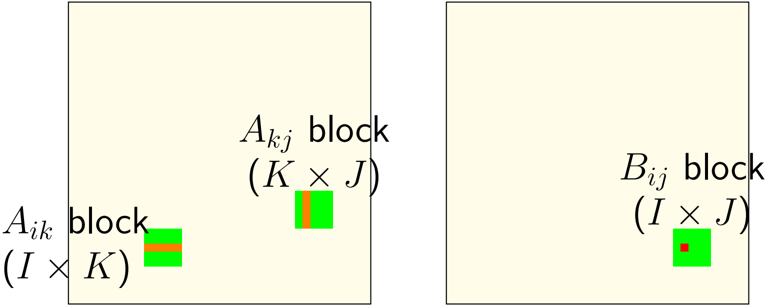
more spatial locality:
calculate on each $A_{i,k}$ and $A_{i,k+1}$ together
both in same cache block — same amount of cache loads

array usage: block



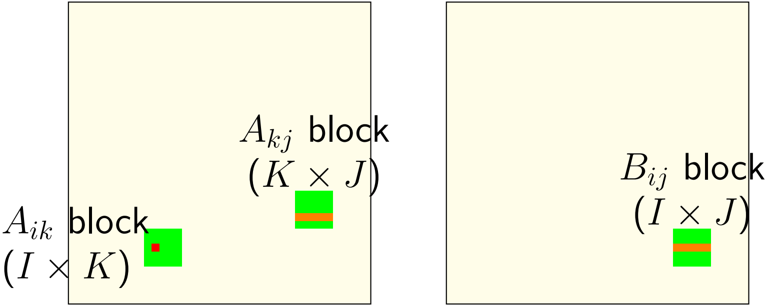
inner loop keeps "blocks" from A, B in cache

array usage: block



B_{ij} calculation uses strips from A
K calculations for one load (cache miss)

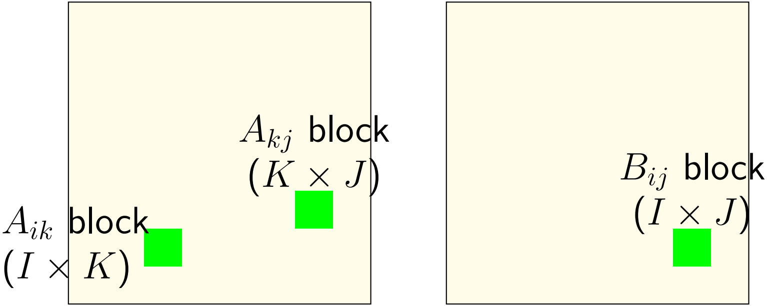
array usage: block



A_{ik} calculation uses strips from A, B

J calculations for one load (cache miss)

array usage: block



(approx.) KIJ fully cached calculations
for $KI + IJ + KJ$ loads
(assuming everything stays in cache)

cache blocking efficiency

load $I \times K$ elements of A_{ik}:

  do $> J$ multiplies with each

load $K \times J$ elements of A_{kj}:

  do I multiplies with each

load $I \times J$ elements of B_{ij}:

  do K adds with each

bigger blocks — more work per load!

catch: $IK + KJ + IJ$ elements must fit in cache

cache blocking rule of thumb

fill the most of the cache with useful data

and do as much work as possible from that

example: my desktop 32KB L1 cache

$I = J = K = 48$ uses $48^2 \times 3$ elements, or 27KB.

assumption: conflict misses aren't important

register reuse

```
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      B[i*N+j] += A[i*N+k] * A[k*N+j];
// optimize into:
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i) {
    float Aik = A[i*N+k]; // hopefully keep in register!
                          // faster than even cache hit!
    for (int j = 0; j < N; ++j)
      B[i*N+j] += Aik * A[k*N+j];
  }
}
```

can compiler do this for us?

can compiler do register reuse?

Not easily — What if `A=B`? What if `A=&B[10]`

```
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i) {
    // want to preload A[i*N+k] here!
    for (int j = 0; j < N; ++j) {
      // but if A = B, modifying here!
      B[i*N+j] += A[i*N+k] * A[k*N+j];
    }
  }
}
```