Performance

last time

inlining

aliasing

removing redundant operations from loops

cache blocking

(sometimes) loop unrolling

cache blocking

```
for (i, j, k in matrices) {
    do calculation
}
```

becomes (to reuse pieces of matrices in cache more)

```
for each (B x B piece of matrices that fit in cache) {
    for each (i, j, k in pieces) {
        do calculation
    }
}
```

aliasing

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

becomes (to keep Aik in register despite maybe C, A overlapping)

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

loop unrolling (ASM)

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

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

```
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];
```

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

performance labs

this week — loop optimizations

after Thanksgiving — vector instructions (AKA SIMD)

performance HWs

INDIVIDUAL ONLY

assignment 1: rotate an image

assignment 2: smooth (blur) an image

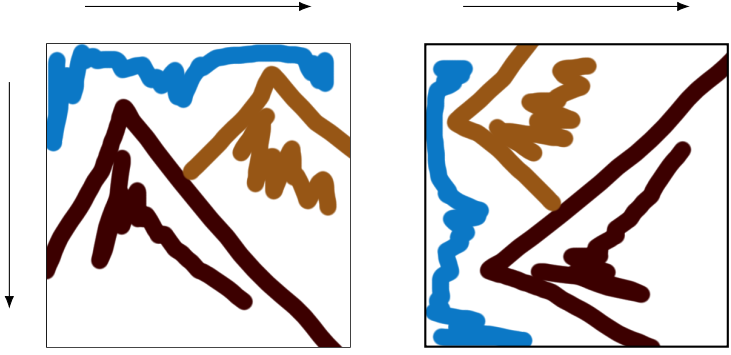
image representation

```
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

interlude: real CPUs

modern CPUs:

execute multiple instructions at once

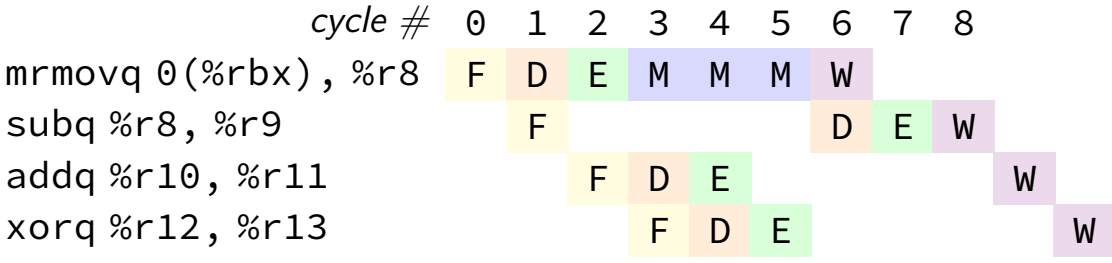
execute instructions out of order — whenever values available

beyond pipelining: out-of-order

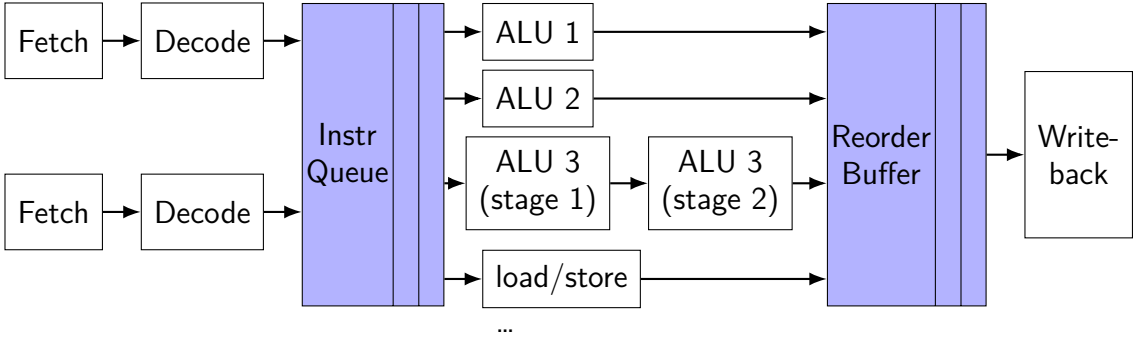
find later instructions to do instead of stalling

lists of available instructions in pipeline registers
- take any instruction with available values

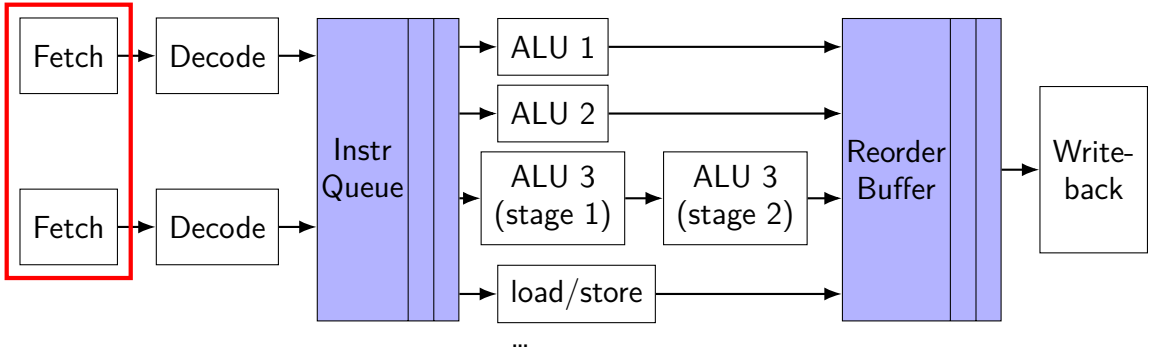
provide illusion that work is still done in order
- much more complicated hazard handling logic



modern CPU design (instruction flow)

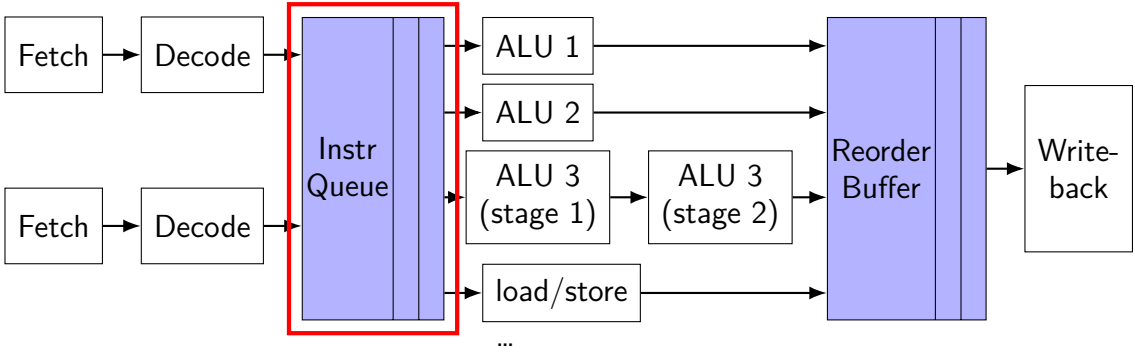


modern CPU design (instruction flow)



fetch multiple instructions/cycle

modern CPU design (instruction flow)

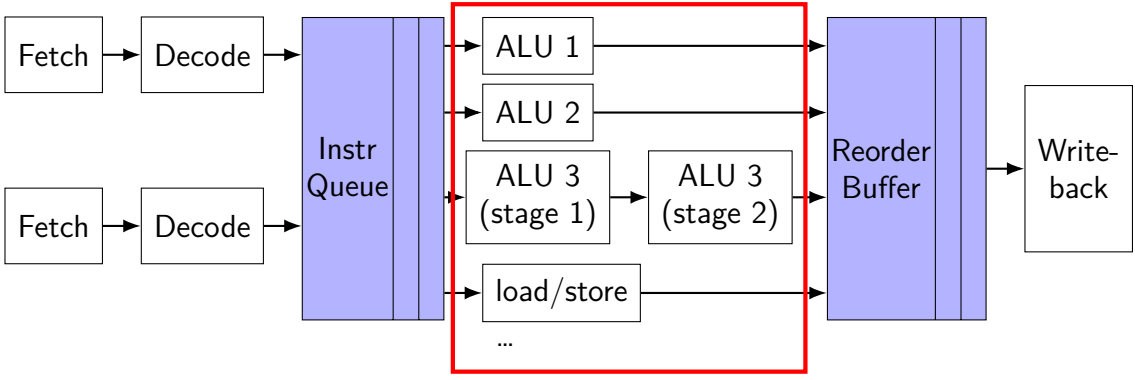


keep list of pending instructions

run instructions from list when operands available

forwarding handled here

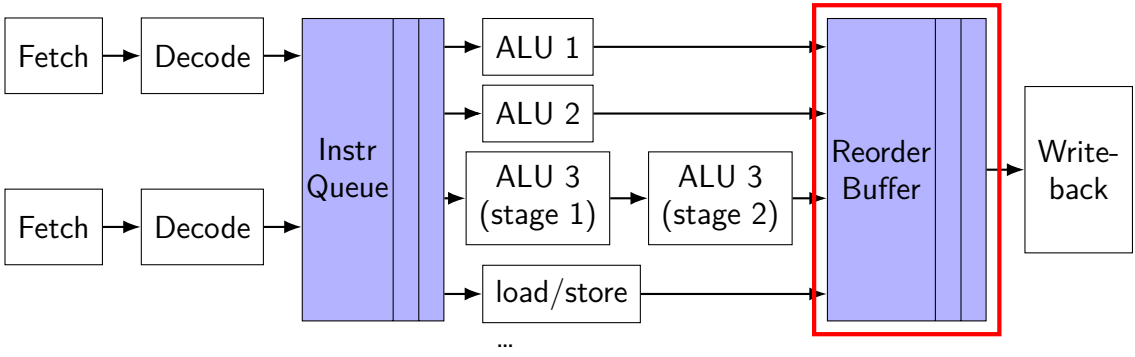
modern CPU design (instruction flow)



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 ...`	*running*
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	—	**...**	

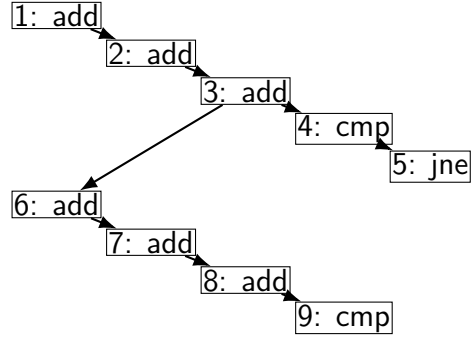
instruction queue operation

#	*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
...	...	

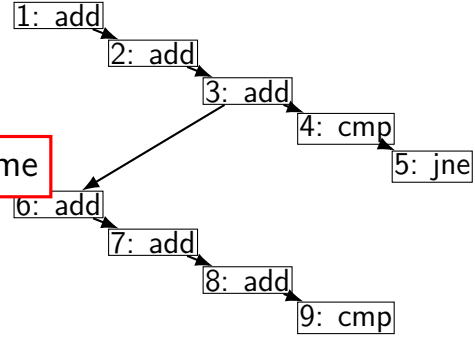


execution unit	*cycle#* 1	2	3	4	5	6	7	...
ALU 1	1	2	3	4	5	8	9	
ALU 2	—	—	—	6	7	—	...	

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

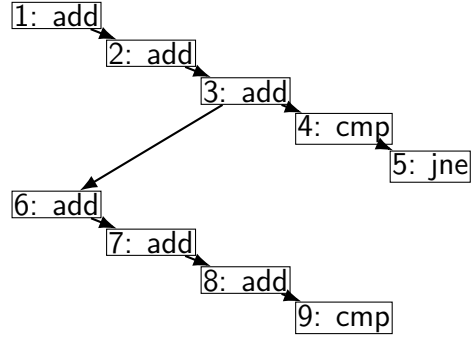


execution unit	*cycle#* 1	2	3	4	5	6	7	…
ALU 1	1	2	3	4	5	8	9	
ALU 2		—	—	—	6	7	—	…

data flow

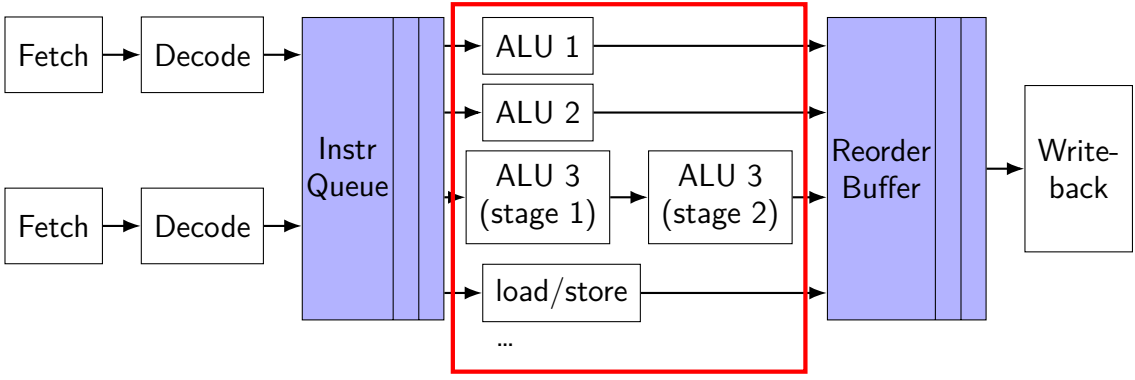
#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`ad`	
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	done
9	`cmpq %r8, %rdx`	done
...	...	

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

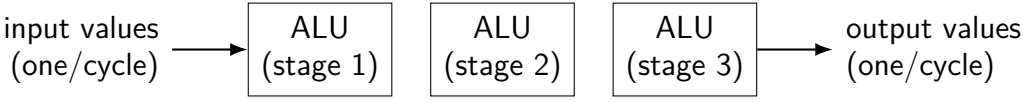
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

(here: 1 op/cycle; 3 cycle latency)



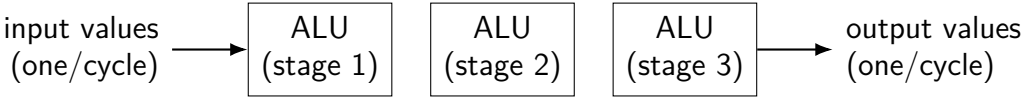
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

(here: 1 op/cycle; 3 cycle latency)



exercise: how long to compute $A \times (B \times (C \times D))$?

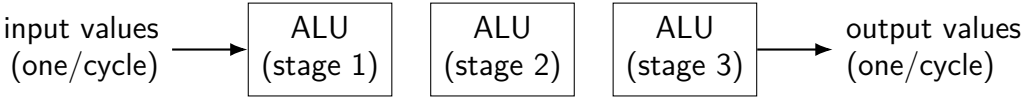
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

(here: 1 op/cycle; 3 cycle latency)



exercise: how long to compute $A \times (B \times (C \times D))$?

3×3 cycles + any time to forward values

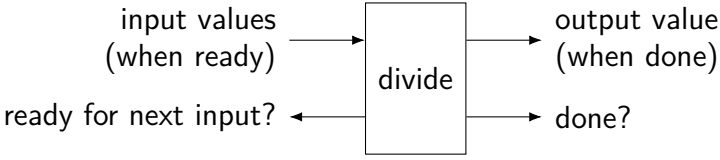
no parallelism!

execution units AKA functional units (2)

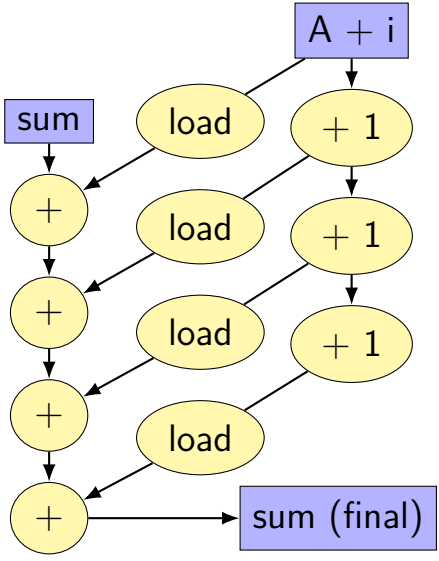
where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes unpipelined:

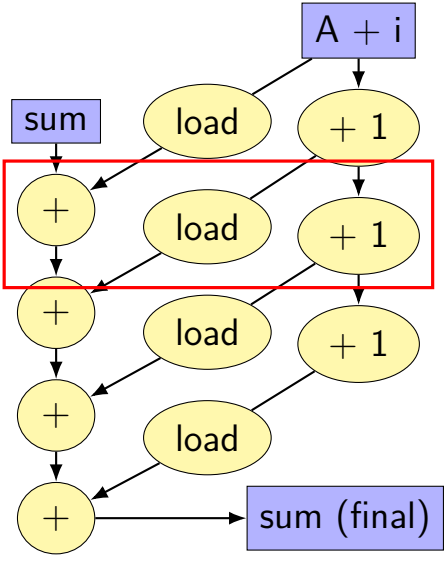


data flow model and limits



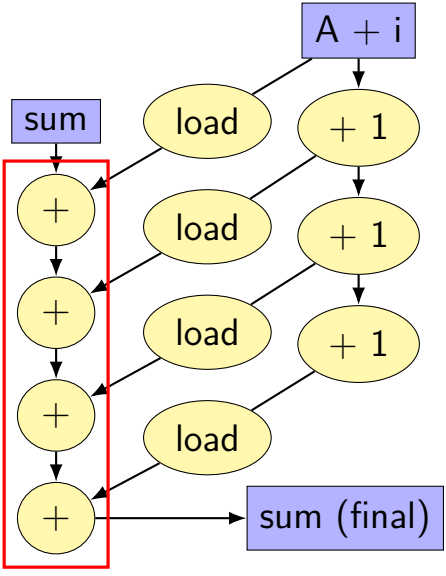

```
for (int i = 0; i < N; i += K) {
    sum += A[i];
    sum += A[i+1];
    ...
}
```

data flow model and limits



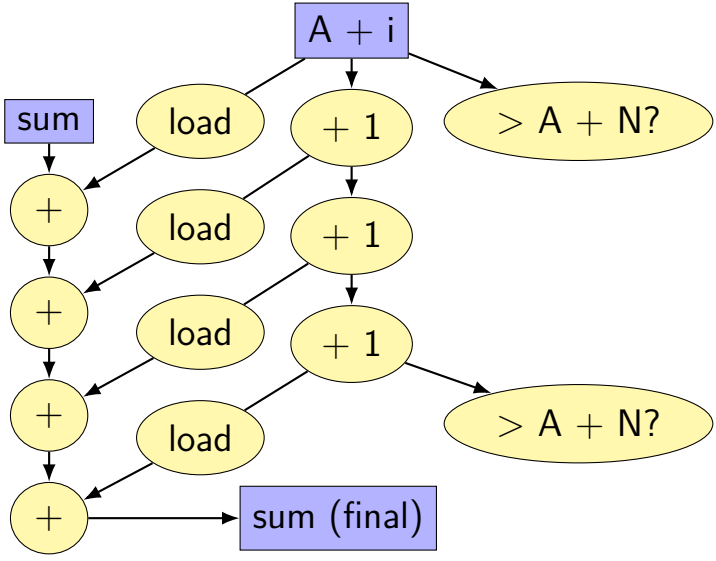
three ops/cycle (if each one cycle)

data flow model and limits



need to do additions
one-at-a-time
book's name: critical path
time needed: sum of latencies

data flow model and limits



reassociation

assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```

$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```

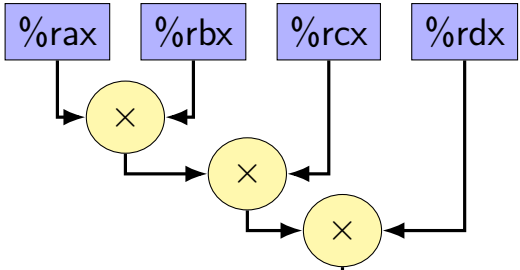
reassociation

assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

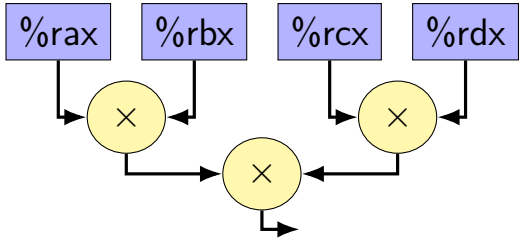
$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```



$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



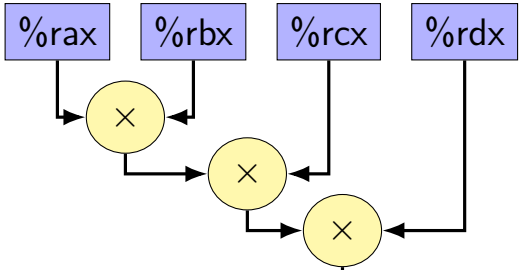
reassociation

assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

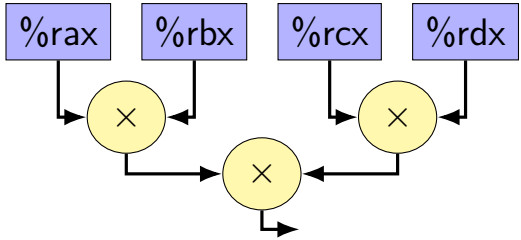
$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```



$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



reassociation

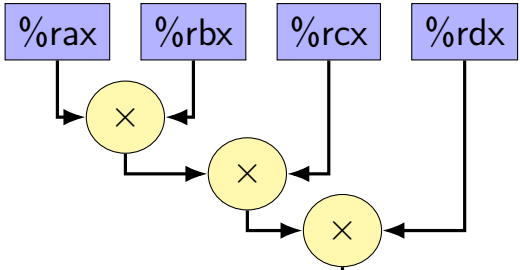
assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```

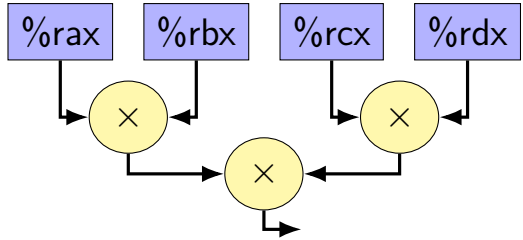
15 cycles



$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```

11 cycles



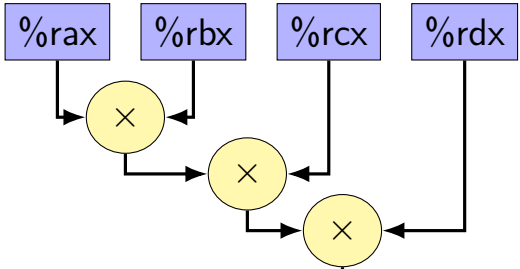
reassociation

assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

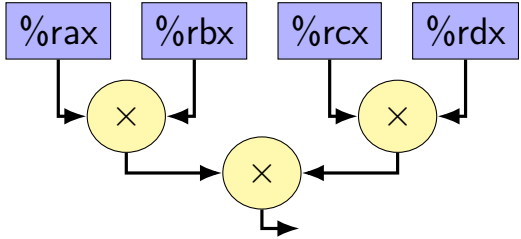
$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```

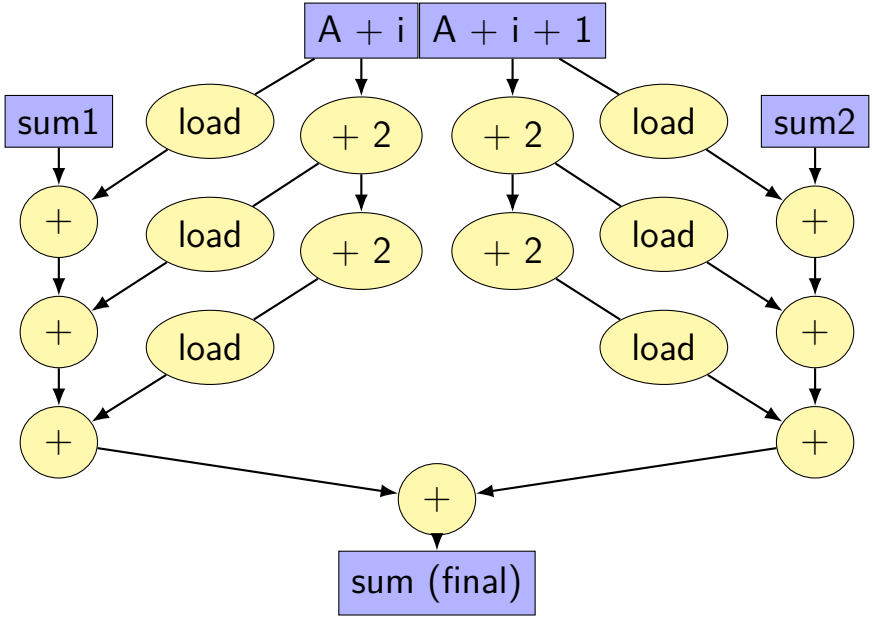


$(a \times b) \times (c \times d)$

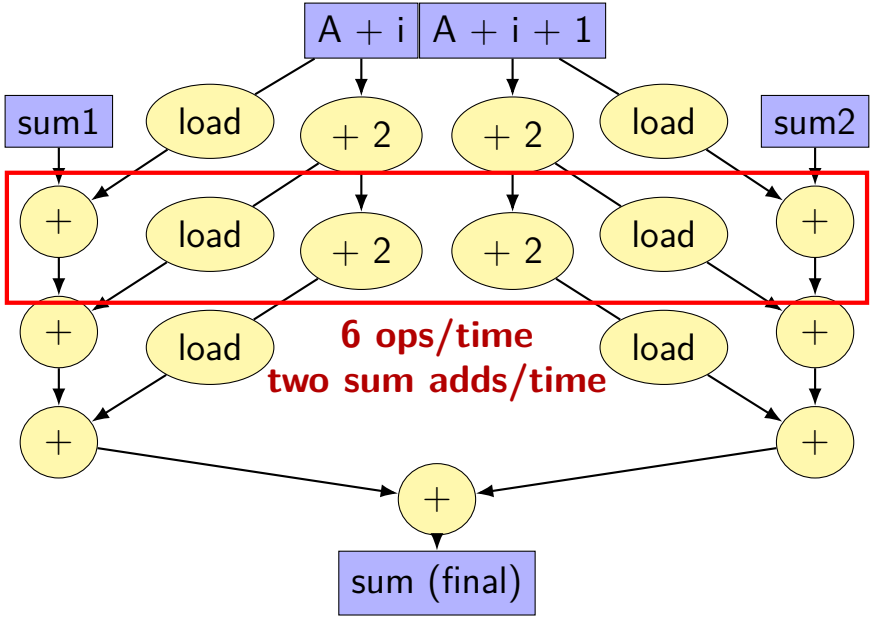
```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



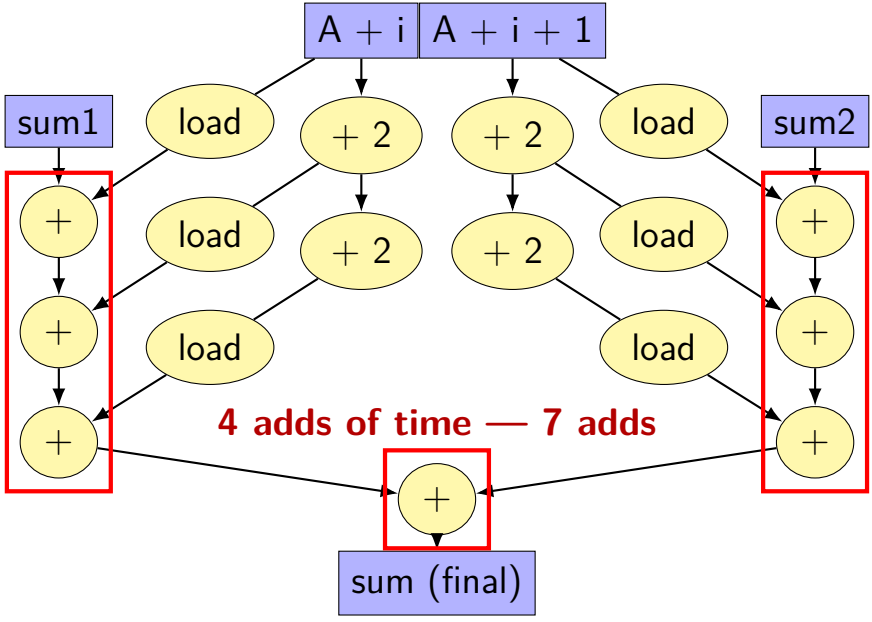
better data-flow



better data-flow



better data-flow



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), %rcx          // 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

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

- loop overhead
- compiler not as efficient as it could have been

my machine: 4 add/etc. or branches/cycle

- 4 copies of ALU (effectively)

$(2 + 2/16 + 1/16 + 1/16) \div 4 \approx 0.57$ cycles/element

loop unrolling v cache blocking

cache blocking in k *and* then loop unrolling

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

---

pretty useless loop unrolling

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

loop unrolling v cache blocking (0)

cache blocking for k only: (1 by 1 by 2 matrix blocks)

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

---

cache blocking for k and loop unrolling:

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

loop unrolling v cache blocking (0)

cache blocking for k only: (1 by 1 by 2 matrix blocks)

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

---

cache blocking for k and loop unrolling:

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

loop unrolling v cache blocking (1)

cache blocking for k,i only: (1 by 2 by 2 matrix blocks)

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

---

cache blocking for k,i and loop unrolling for i:

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

caching block + loop unroll + no aliasing

+ preventing aliasing problems:

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      float Ci0j = C[(i+0)*N+j];
      float Ci1j = C[(i+1)*N+j];
      for(int kk = k; kk < k + 2; ++kk) {
        float Bkj = B[kk*N+j];
        Ci0j += A[(i+0)*N+kk] * Bkj;
        Ci1j += A[(i+1)*N+kk] * Bkj;
      }
      C[(i+0)*N+j] += Ci0j;
      C[(i+1)*N+j] += Ci1j;
    }
```

caching block + loop unroll + no aliasing

+ preventing aliasing problems:

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      float Ci0j = C[(i+0)*N+j];
      float Ci1j = C[(i+1)*N+j];
      for(int kk = k; kk < k + 2; ++kk) {
        float Bkj = B[kk*N+j];
        Ci0j += A[(i+0)*N+kk] * Bkj;
        Ci1j += A[(i+1)*N+kk] * Bkj;
      }
      C[(i+0)*N+j] += Ci0j;
      C[(i+1)*N+j] += Ci1j;
    }
```

loop unrolling v cache blocking

loop unrolling in j:

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

aliasing problems with cache blocking

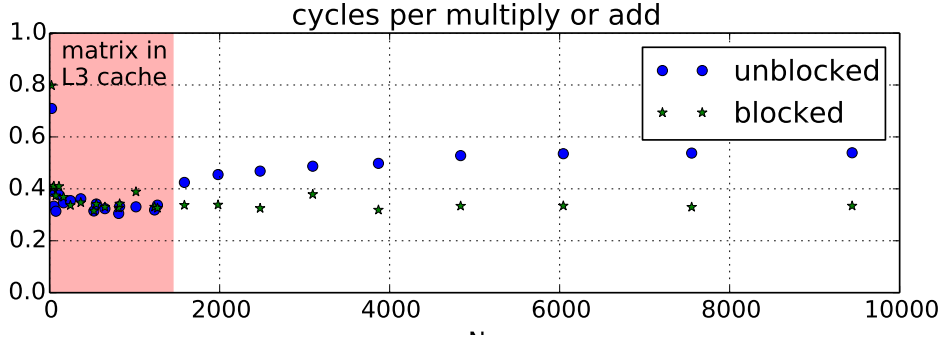
```
for (int k = 0; k < N; k++) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; j += 2) {
      C[(i+0)*N + j+0] += A[i*N+k] * B[k*N+j];
      C[(i+1)*N + j+0] += A[(i+1)*N+k] * B[k*N+j];
      C[(i+0)*N + j+1] += A[i*N+k] * B[k*N+j+1];
      C[(i+1)*N + j+1] += A[(i+1)*N+k] * B[k*N+j+1];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

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

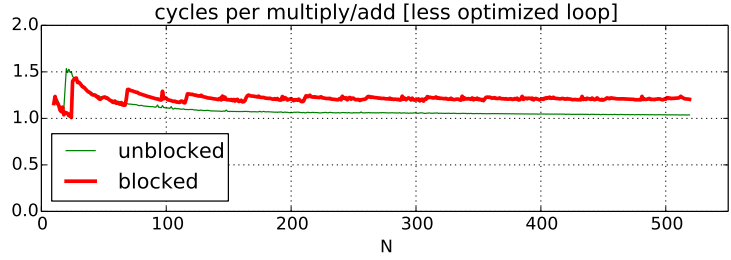
“register blocking”

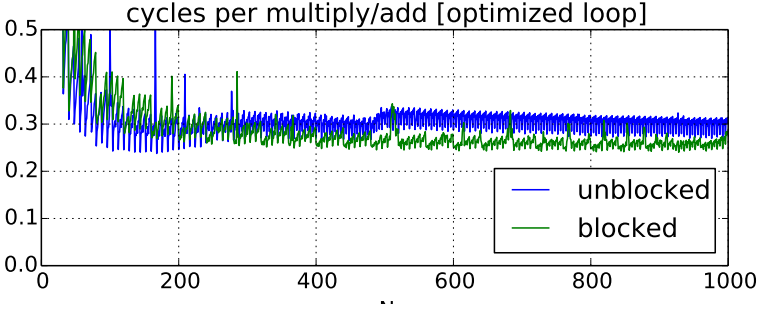
```
for (int k = 0; k < N; ++k) {
  for (int i = 0; i < N; i += 2) {
    float Ai0k = A[(i+0)*N + k];
    float Ai1k = A[(i+1)*N + k];
    for (int j = 0; j < N; j += 2) {
      float Bkj0 = A[k*N + j+0];
      float Bkj1 = A[k*N + j+1];
      C[(i+0)*N + j+0] += Ai0k * Bkj0;
      C[(i+1)*N + j+0] +=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

cache blocking performance (big sizes)

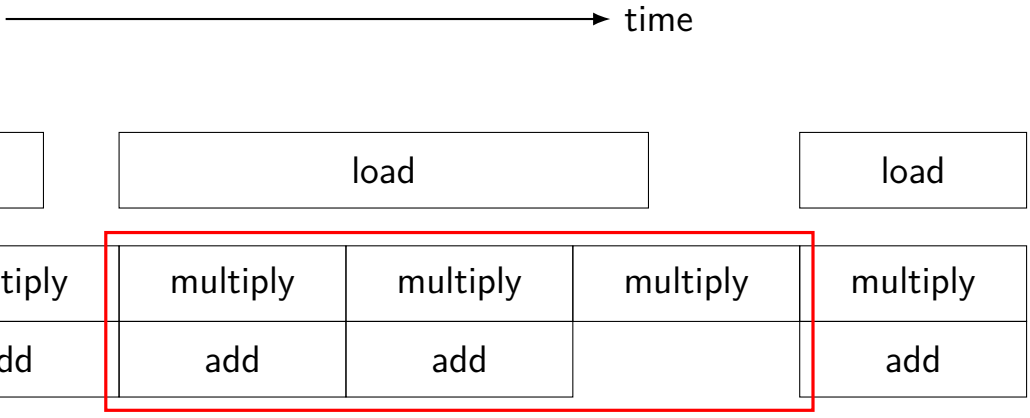


cache blocking performance (small sizes)





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

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
```

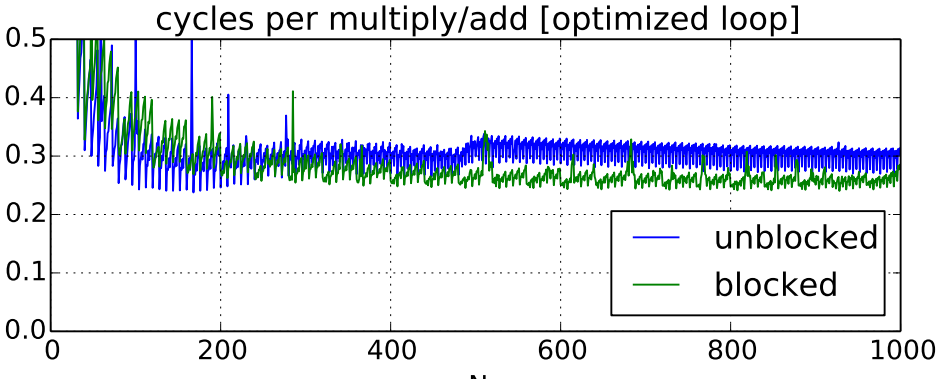
$$\frac{1374389535}{2^{37}} \approx \frac{1}{100}$$

constant multiplies/divides

compiler is very good at handling

...but need to actually use constants

wiggles on prior graphs



variance from this optimization

8 elements in vector, so multiples of 8 easier

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

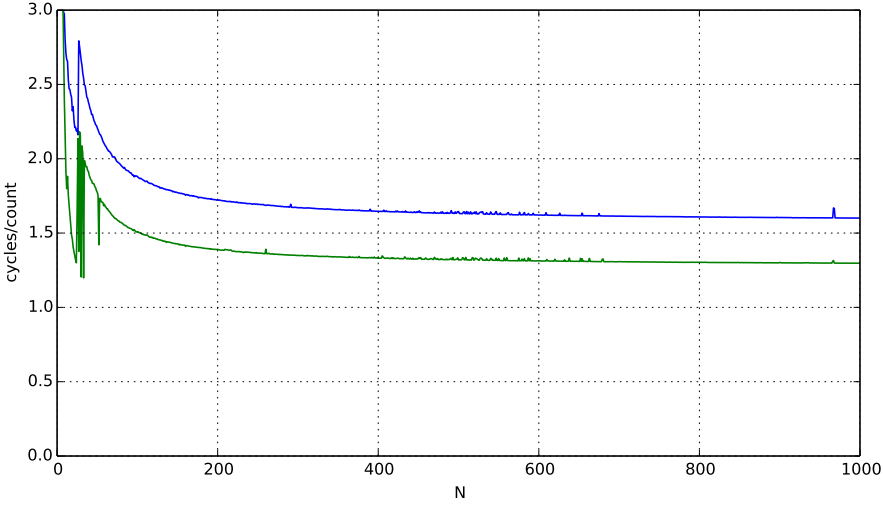
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

non-contrived aliasing

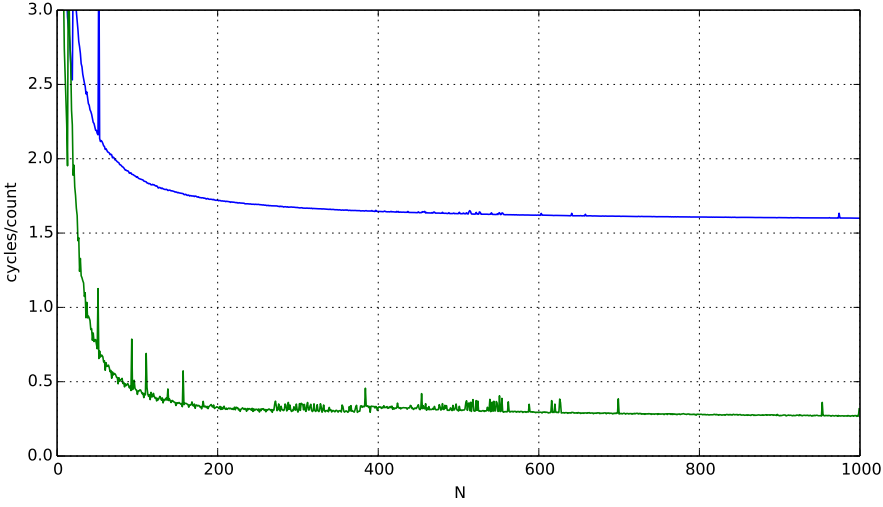
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

```
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



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