More Performance

Changelog

Changes made in this version not seen in first lecture:

- 7 November 2017: reassociation: $a \times (b \times (c \times d))) \rightarrow ((a \times b) \times c) \times d$ to be more consistent with assembly
- 7 November 2017: reassociation: correct +s to ×s.
- 7 November 2017: general advice [on perf assignment]: note not for when we give specific advice
- 7 November 2017: vector instructions: include term SIMD
- 7 November 2017: vector intrinsics: SIMD → vector

exam graded

median 80%; 25th percentile: 73%; 75th percentile: 87%

please submit regrades soon

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)

```
    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

next week — vector instructions (AKA SIMD)

both new this semester

performance HWs

partners or individual (your choice)

two parts:

- rotate an image
- smooth (blur) an image

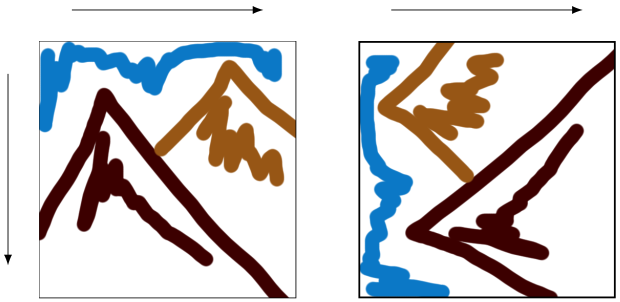
image representation

```
typedef struct { unsigned char red, green, blue, alpha; } pixe
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

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

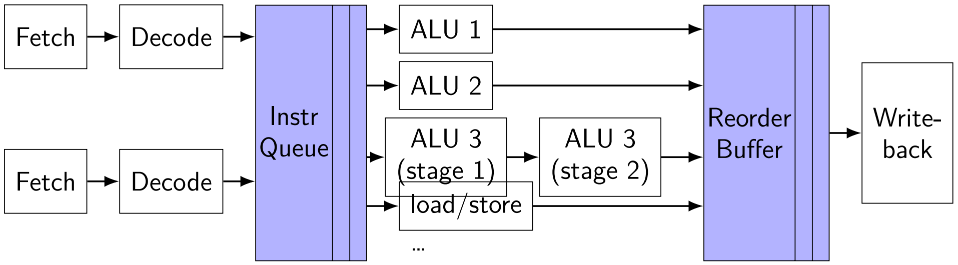
- take any instruction with available values

provide illusion that work is still done in order

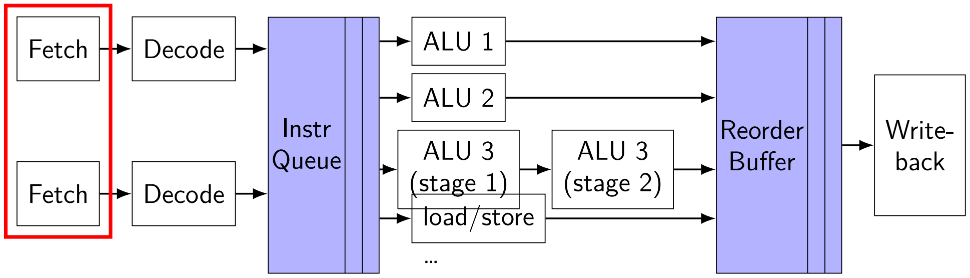
- much more complicated hazard handling logic

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)

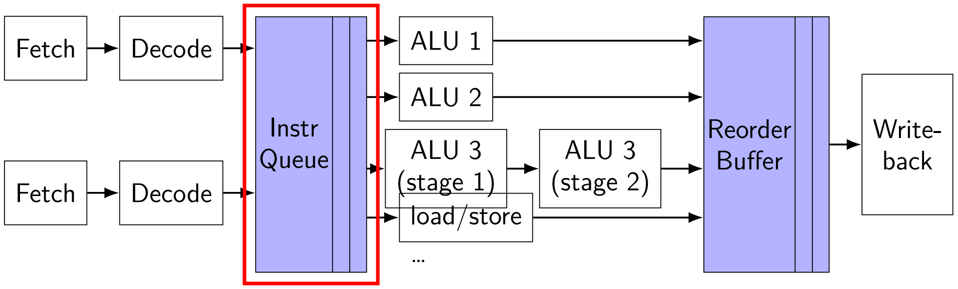


modern CPU design (instruction flow)



fetch multiple instructions/cycle

modern CPU design (instruction flow)

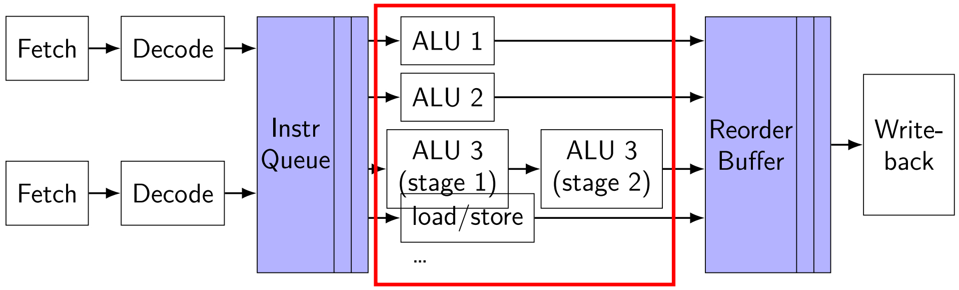


keep list of pending instructions

run instructions from list when operands available

forwarding handled here

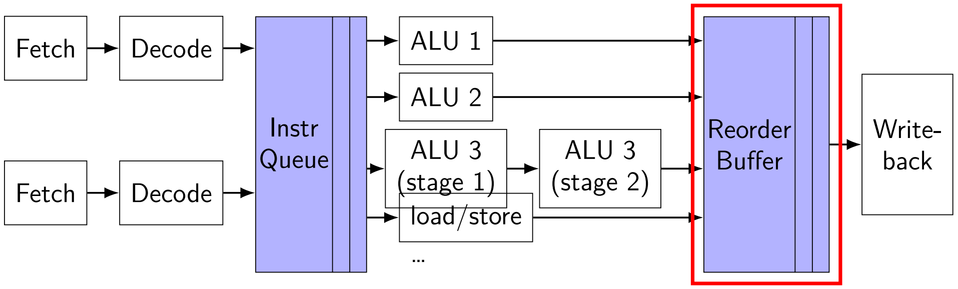
modern CPU design (instruction flow)



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	—	**…**	

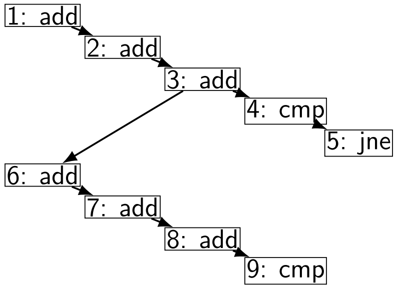
instruction queue operation

#	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
…	…	

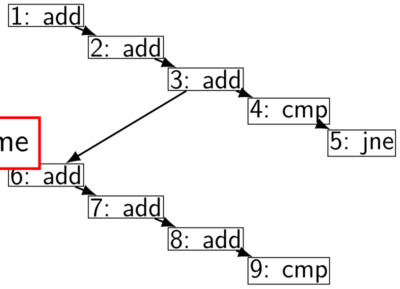

1: add
2: add
3: add
4: cmp
5: jne
6: add
7: add
8: add
9: cmp

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

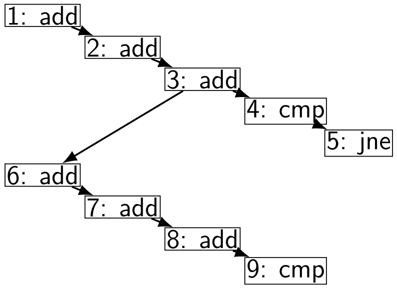

1: add
2: add
3: add
4: cmp
5: jne
6: add
7: add
8: add
9: cmp

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

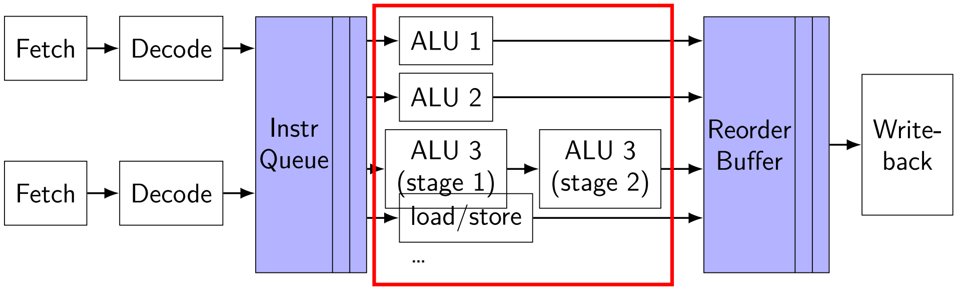
#	instruction	status
1	addq %rax, %rdx	done
2	addq %rbx, %rdx	done
3	addq %rcx, %rdx	done
4	cmpq %r8, %rdx	done
5	jne ...	done
6	ac	
7	addq %rbx, %rdx	done
8	addq %rcx, %rdx	done
9	cmpq %r8, %rdx	done
…	…	

longest path determines speed


1: add
2: add
3: add
4: cmp
5: jne
6: add
7: add
8: add
9: cmp

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

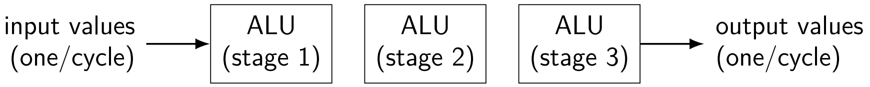
sometimes pipelined, sometimes not

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)

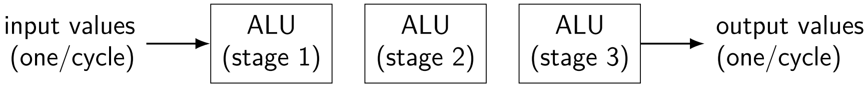


execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

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



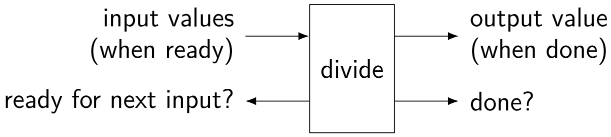
exercise: how long to compute $A \times (B \times (C \times D))$?

execution units AKA functional units (2)

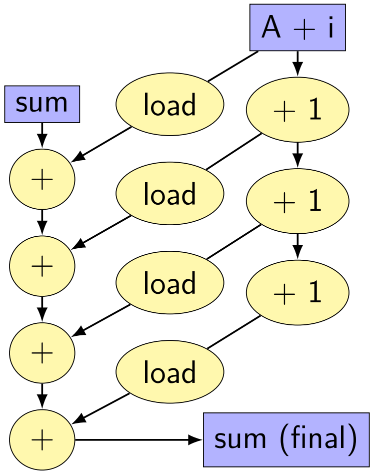
where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes unpipelined:

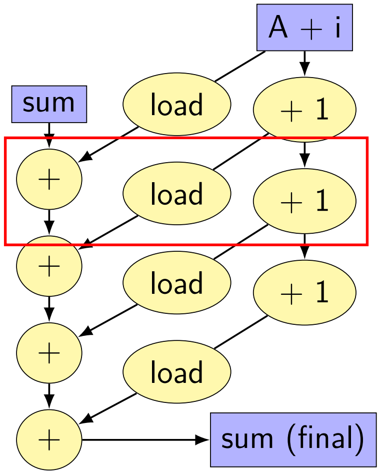


data flow model and limits



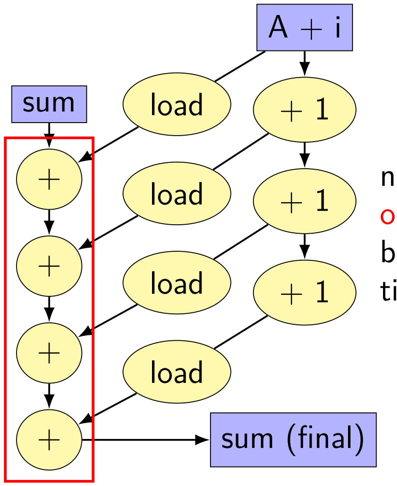
```
for (int i = 0; i < N; i += K) {
    sum += A[i];
    sum += A[i+1];
    ...
}
```

data flow model and limits



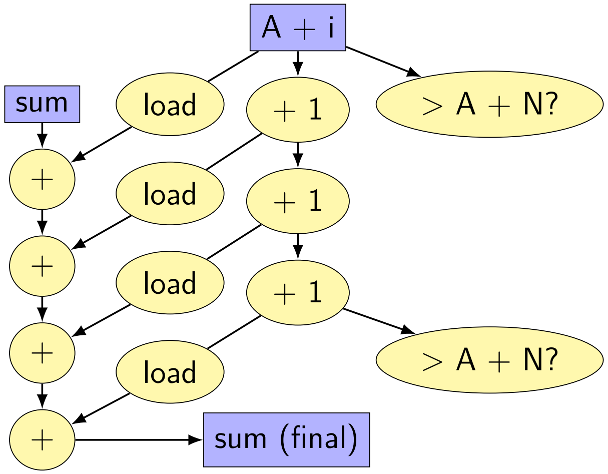
three ops/cycle (if each one cycle)

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
```

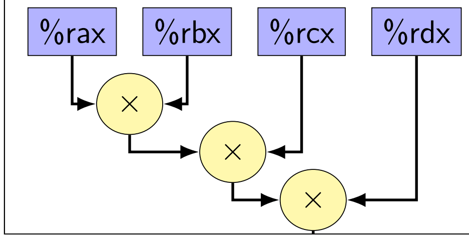
reassociation

assume a single pipelined, 5-cycle latency multiplier

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

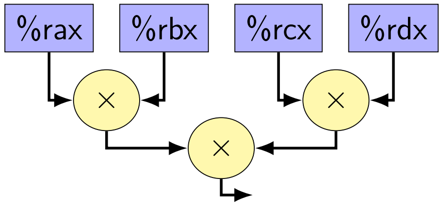
$((a \times b) \times c) \times d$

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

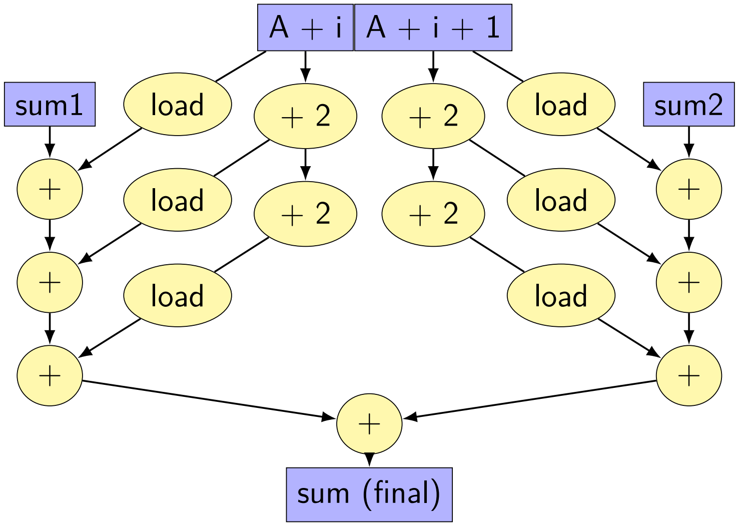


$(a \times b) \times (c \times d)$

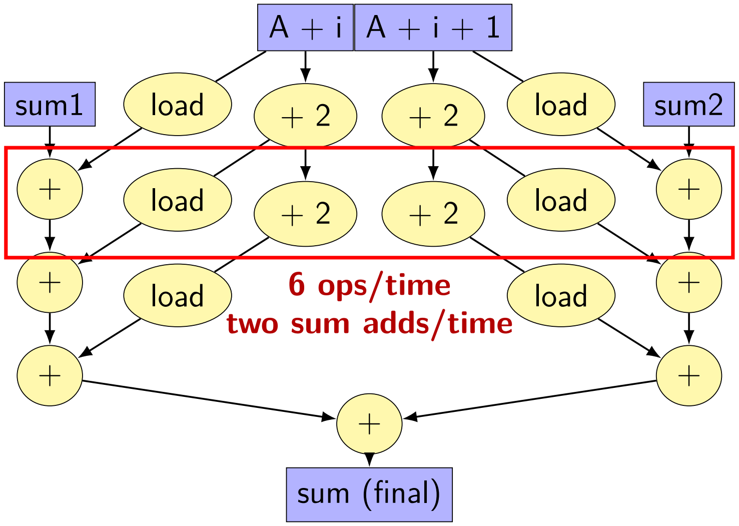
```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



better data-flow

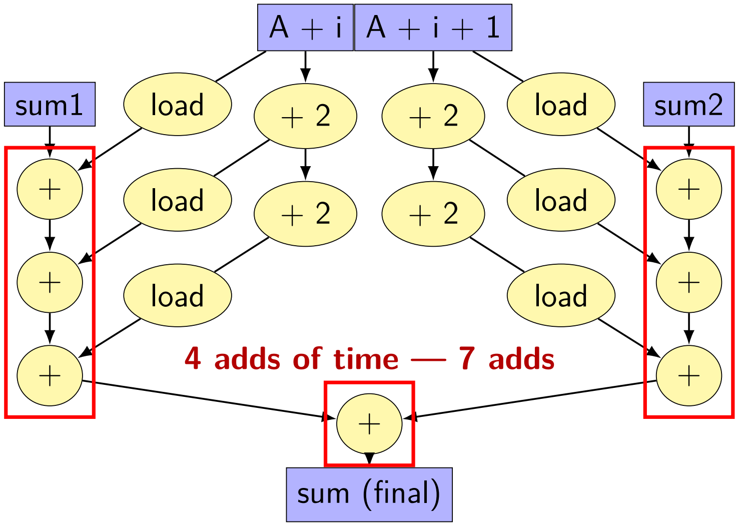


better data-flow



6 ops/time
two sum adds/time

better data-flow



4 adds of time — 7 adds

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

vector instructions

modern processors have registers that hold “vector” of values

example: X86-64 has 128-bit registers

- 4 ints or 4 floats or 2 doubles or …

128-bit registers named `%xmm0` through `%xmm15`

instructions that act on all values in register

- vector instructions or SIMD (single instruction, multiple data) instructions

extra copies of ALUs only accessed by vector instructions

example vector instruction

`paddd %xmm0, %xmm1` (packed add dword (32-bit))

Suppose registers contain (interpreted as 4 ints)

- %xmm0: [1, 2, 3, 4]
- %xmm1: [5, 6, 7, 8]

Result will be:

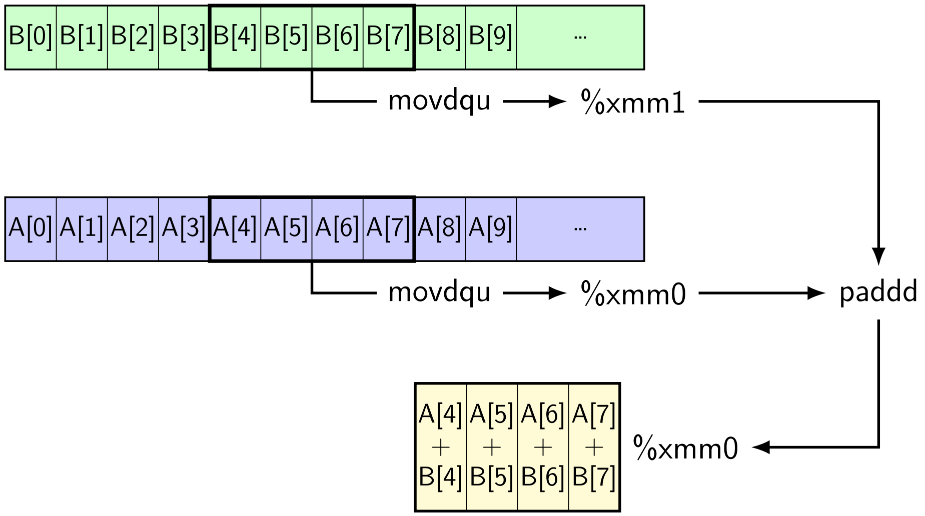
- %xmm1: [6, 8, 10, 12]

vector instructions

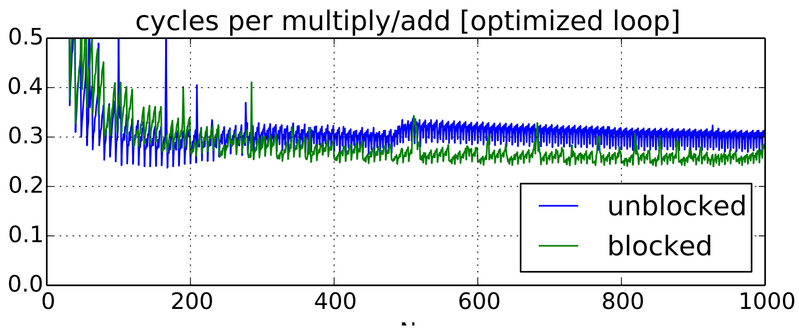
```
void add(int * restrict a, int * restrict b) {
    for (int i = 0; i < 128; ++i)
        a[i] += b[i];
}
```

```
add:
        xorl    %eax, %eax          // init. loop counter
the_loop:
        movdqu  (%rdi,%rax), %xmm0  // load 4 from A
        movdqu  (%rsi,%rax), %xmm1  // load 4 from B
        paddd   %xmm1, %xmm0        // add 4 elements!
        movups  %xmm0, (%rdi,%rax)  // store 4 in A
        addq    $16, %rax           // +4 ints = +16
        cmpq    $512, %rax          // 512 = 4 * 128
        jne     the_loop
        rep ret
```

vector add picture


B[0] B[1] B[2] B[3] B[4] B[5] B[6] B[7] B[8] B[9] …
movdqu
%xmm1
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9] …
movdqu
%xmm0
paddd
A[4] + B[4]
A[5] + B[5]
A[6] + B[6]
A[7] + B[7]
%xmm0

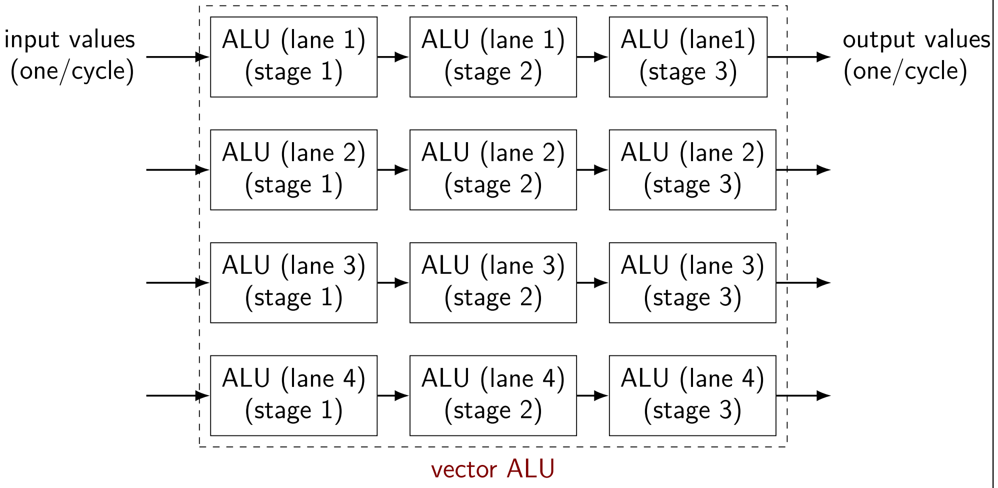
wiggles on prior graphs



variance from this optimization

8 elements in vector, so multiples of 8 easier

one view of vector functional units


input values (one/cycle)

ALU (lane 1) (stage 1) → ALU (lane 1) (stage 2) → ALU (lane1) (stage 3)

ALU (lane 2) (stage 1) → ALU (lane 2) (stage 2) → ALU (lane 2) (stage 3)

ALU (lane 3) (stage 1) → ALU (lane 3) (stage 2) → ALU (lane 3) (stage 3)

ALU (lane 4) (stage 1) → ALU (lane 4) (stage 2) → ALU (lane 4) (stage 3)

output values (one/cycle)

vector ALU

why vector instructions?

lots of logic not dedicated to computation

- instruction queue
- reorder buffer
- instruction fetch
- branch prediction
- …

adding vector instructions — little extra control logic

…but a lot more computational capacity

vector instructions and compilers

compilers can sometimes figure out how to use vector instructions

(and have gotten much, much better at it over the past decade)

but easily messsed up:

- by aliasing
- by conditionals
- by some operation with no vector instruction
- …

fickle compiler vectorization (1)

GCC 7.2 and Clang 5.0 generate vector instructions for this:

```
#define N 1024
void foo(unsigned int *A, unsigned int *B) {
    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];
}
```

but not:

```
#define N 1024
void foo(unsigned int *A, unsigned int *B) {
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
                B[i * N + j] += A[i * N + k] * A[j * N + k];
}
```

fickle compiler vectorization (2)

Clang 5.0.0 generates vector instructions for this:

```
void foo(int N, unsigned int *A, unsigned int *B) {
    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];
}
```

but not: (probably bug?)

```
void foo(long N, unsigned int *A, unsigned int *B) {
    for (long k = 0; k < N; ++k)
        for (long i = 0; i < N; ++i)
            for (long j = 0; j < N; ++j)
                B[i * N + j] += A[i * N + k] * A[k * N + j];
}
```

vector intrinsics

if compiler doesn't work...

could write vector instruction assembly by hand

second option: "intrinsic functions"

C functions that compile to particular instructions

vector intrinsics: add example

```
void vectorized_add(int *a, int *b) {
  for (int i = 0; i < 128; i += 4) {
     // "si128" --> 128 bit integer
     // a_values = {a[i], a[i+1], a[i+2], a[i+3]}
     __m128i a_values = _mm_loadu_si128((__m128i*) &a[i]);
     // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
     __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);

     // add four 32-bit integers
     // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
     __m128i sums = _mm_add_epi32(a_values, b_values);

     // {a[i], a[i+1], a[i+2], a[i+3]} = sums
     _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

vector intrinsics: add example

special type `__m128i` — "128 bits of integers"
other types: `__m128` (floats), `__m128d` (doubles)

```
void vec
  for (int i = 0; i < 128; i += 4) {
    // "si128" --> 128 bit integer
    // a_values = {a[i], a[i+1], a[i+2], a[i+3]}
    __m128i a_values = _mm_loadu_si128((__m128i*) &a[i]);
    // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
    __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);

    // add four 32-bit integers
    // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
    __m128i sums = _mm_add_epi32(a_values, b_values);

    // {a[i], a[i+1], a[i+2], a[i+3]} = sums
    _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

vector intrinsics: add example

functions to store/load
`si128` means "128-bit integer value"
u for "unaligned" (otherwise, pointer address must be multiple of 16)

```
v
    // "si128" --> 128 bit integer
    // a_values = {a[i], a[i+1], a[i+2], a[i+3]}
    __m128i a_values = _mm_loadu_si128((__m128i*) &a[i]);
    // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
    __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);

    // add four 32-bit integers
    // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
    __m128i sums = _mm_add_epi32(a_values, b_values);

    // {a[i], a[i+1], a[i+2], a[i+3]} = sums
    _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

vector intrinsics: add example

function to add
`epi32` means "4 32-bit integers"

```
void vectorized_add(int *a, int *b) {
  for (int i = 0; i < 128; i += 4) {
    // "si128"
    // a_values                                 +3]}
    __m128i a_v                                 28i*) &a[i]);
    // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
    __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);

    // add four 32-bit integers
    // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
    __m128i sums = _mm_add_epi32(a_values, b_values);

    // {a[i], a[i+1], a[i+2], a[i+3]} = sums
    _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

vector intrinsics: different size

```
void vectorized_add_64bit(long *a, long *b) {
  for (int i = 0; i < 128; i += 2) {
   // a_values = {a[i], a[i+1]} (2 x 64 bits)
    __m128i a_values = _mm_loadu_si128((__m128i*) &a[i]);
   // b_values = {b[i], b[i+1]} (2 x 64 bits)
    __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);
    // add two 64-bit integers: paddq %xmm0, %xmm1
    // sums = {a[i] + b[i], a[i+1] + b[i+1]}
    __m128i sums = _mm_add_epi64(a_values, b_values);
    // {a[i], a[i+1]} = sums
    _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

vector intrinsics: different size

```
void vectorized_add_64bit(long *a, long *b) {
  for (int i = 0; i < 128; i += 2) {
    // a_values = {a[i], a[i+1]} (2 x 64 bits)
     __m128i a_values = _mm_loadu_si128((__m128i*) &a[i]);
    // b_values = {b[i], b[i+1]} (2 x 64 bits)
     __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);
     // add two 64-bit integers: paddq %xmm0, %xmm1
     // sums = {a[i] + b[i], a[i+1] + b[i+1]}
     __m128i sums = _mm_add_epi64(a_values, b_values);
     // {a[i], a[i+1]} = sums
     _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

recall: square

```
void square(unsigned int *A, unsigned int *B) {
    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 *
}
```

square unrolled

```
void square(unsigned int *A, unsigned int *B) {
  for (int k = 0; k < N; ++k) {
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; j += 4) {
        /* goal: vectorize this */
        B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
        B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
        B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
        B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
      }
}
```

handy intrinsic functions for square

`_mm_set1_epi32` — load four copies of a 32-bit value into a 128-bit value

instructions generated vary; one example: `movq` + `pshufd`

`_mm_mullo_epi32` — multiply four pairs of 32-bit values, give lowest 32-bits of results

generates `pmulld`

vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

```
// load four elements from B
Bij = _mm_loadu_si128(&B[i * N + j + 0]);
... // manipulate vector here
// store four elements into B
_mm_storeu_si128((__m128i*) &B[i * N + j + 0], Bij);
```

vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

```
// load four elements from A
Akj = _mm_loadu_si128(&A[k * N + j + 0]);
... // multiply each by A[i * N + k] here
```

vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

```
// load four elements starting with A[k * n + j]
Akj = _mm_loadu_si128(&A[k * N + j + 0]);
// load four copies of A[i * N + k]
Aik = _mm_set1_epi32(A[i * N + k]);
// multiply each pair
multiply_results = _mm_mullo_epi32(Aik, Akj);
```

vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

```
Bij = _mm_add_epi32(Bij, multiply_results);
// store back results
_mm_storeu_si128(..., Bij);
```

square vectorized

```
__m128i Bij, Akj, Aik, Aik_times_Akj;

// Bij = {$B_{i,j}$, $B_{i,j+1}$, $B_{i,j+2}$, $B_{i,j+3}$}
Bij = _mm_loadu_si128((__m128i*) &B[i * N + j]);
// Akj = {$A_{k,j}$, $A_{k,j+1}$, $A_{k,j+2}$, $A_{k,j+3}$}
Akj = _mm_loadu_si128((__m128i*) &A[k * N + j]);

// Aik = {$A_{i,k}$, $A_{i,k}$, $A_{i,k}$, $A_{i,k}$}
Aik = _mm_set1_epi32(A[i * N + k]);

// Aik_times_Akj = {$A_{i,k} \times A_{k,j}$, $A_{i,k} \times A_{k,j+1}$, $A_{i,k} \times A_{k,j+2}$, $A_{i,k} \times A_{k,j+3}$}
Aik_times_Akj = _mm_mullo_epi32(Aij, Akj);

// Bij= {$B_{i,j} + A_{i,k} \times A_{k,j}$, $B_{i,j+1} + A_{i,k} \times A_{k,j+1}$, ...}
Bij = _mm_add_epi32(Bij, Aik_times_Akj);

// store Bij into B
_mm_storeu_si128((__m128i*) &B[i * N + j], Bij);
```

other vector instructions

multiple extensions to the X86 instruction set for vector instructions

this class: SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2

- supported on lab machines
- 128-bit vectors

latest X86 processors: AVX, AVX2, AVX-512

- 256-bit and 512-bit vectors

other vector instructions features

AVX2/AVX/SSE pretty limiting

other vector instruction sets often more featureful:

- (and require more sophisticated HW support)

better conditional handling

better variable-length vectors

ability to load/store non-contiguous values

optimizing real programs

spend effort where it matters

e.g. 90% of program time spent reading files, but optimize computation?

e.g. 90% of program time spent in routine A, but optimize B?

profilers

first step — tool to determine where you spend time

tools exist to do this for programs

example on Linux: `perf`

perf usage

sampling profiler

- stops periodically, takes a look at what's running

`perf record OPTIONS program`

- example OPTIONS:
- `-F 200` — record 200/second
- `--call-graph=dwarf` — record stack traces

`perf report` or `perf annotate`

children/self

"children" — samples in function or things it called

"self" — samples in function alone

demo

other profiling techniques

count number of times each function is called

not sampling — exact counts, but higher overhead

- might give less insight into amount of time

tuning optimizations

biggest factor: how fast is it actually

setup a benchmark

- make sure it's realistic (right size? uses answer? etc.)

compare the alternatives

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 (× 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 (× 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