Changelog

Corrections made in this version not in first posting:

- 27 Mar 2017: slide 18: mark suspect numbers for 1 accumulator
- 5 May 2017: slide 7: “slower if” to “can be slower if”

notes on rotate

I probably set threshold too low

- it's possible to avoid strategies we want you to do
- our reference solutions were biased toward old size/platform — made it look harder

I think too late to reasonably change

to learn what you should learn…

aim for at least 1.65x or 1.70x, not 1.60x

smooth is more work, probably

loop optimizations

back to simpler example

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

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?
```

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?
```

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];
```

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?

can be slower if small number of iterations

larger code — could exceed instruction cache space

loop unrolling performance

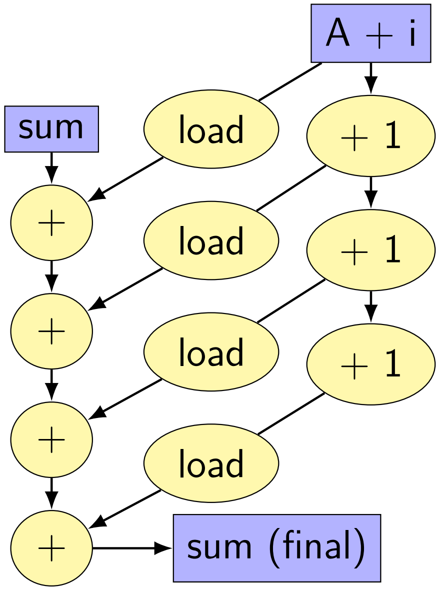
on my laptop with 992 elements (fits in L1 cache)

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

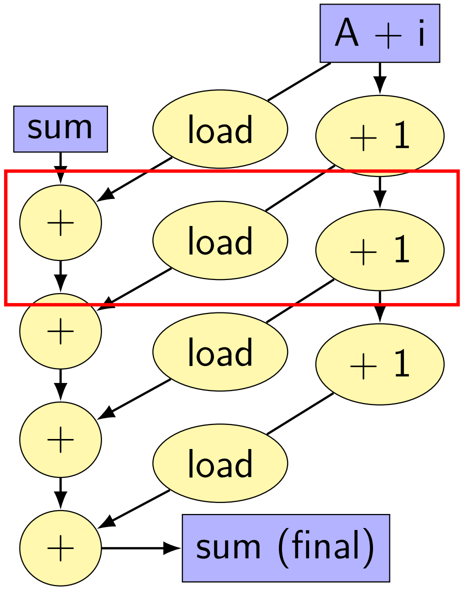
instruction cache/etc. overhead

1.01 cycles/element — latency bound

data flow model and limits

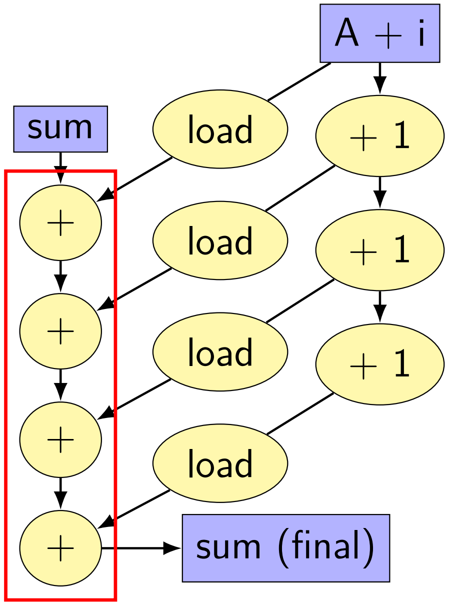


data flow model and limits



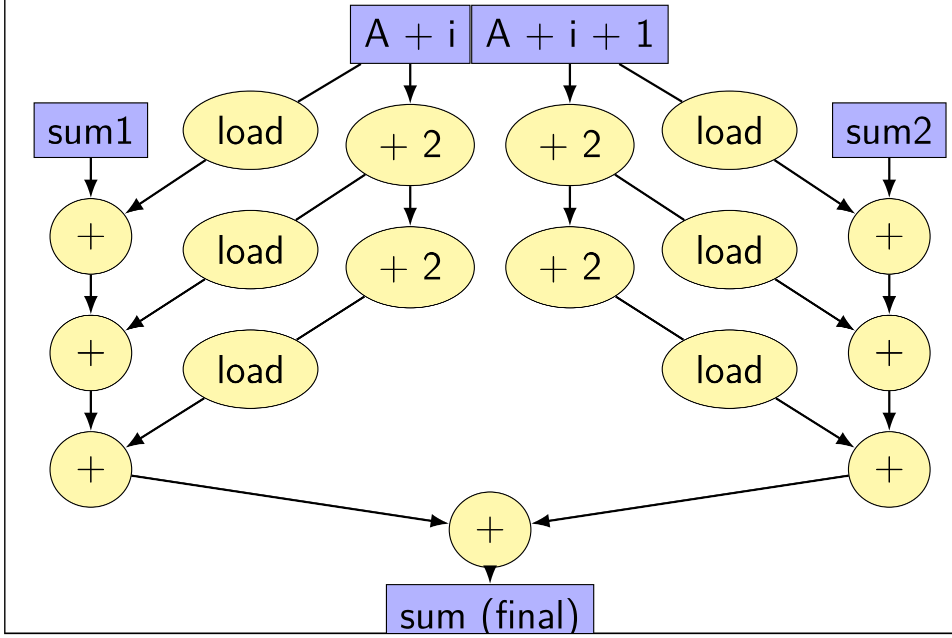
three ops/cycle
(if each one cycle)

data flow model and limits

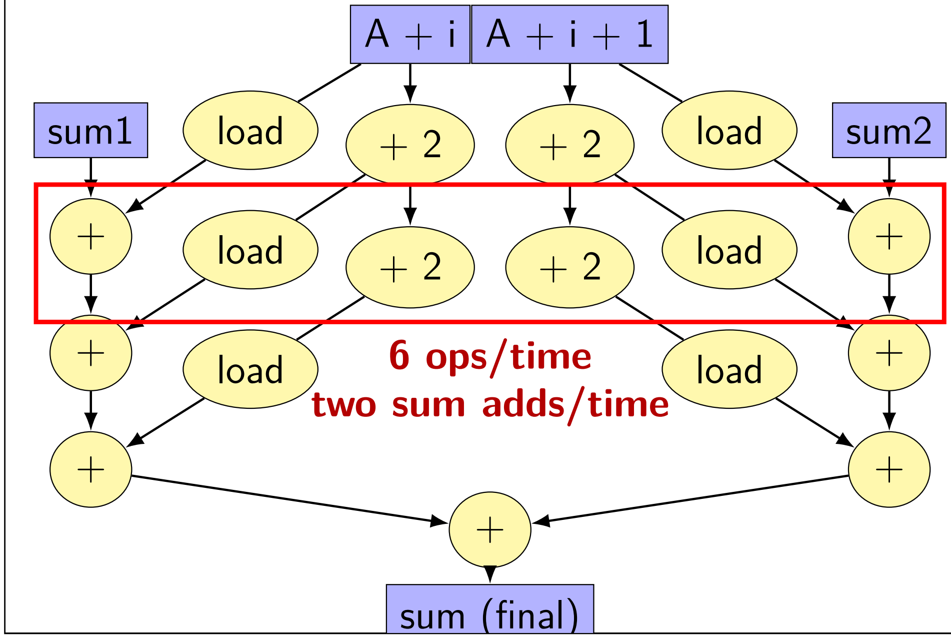


need to do additions
one-at-a-time
book's name: critical path

better data-flow

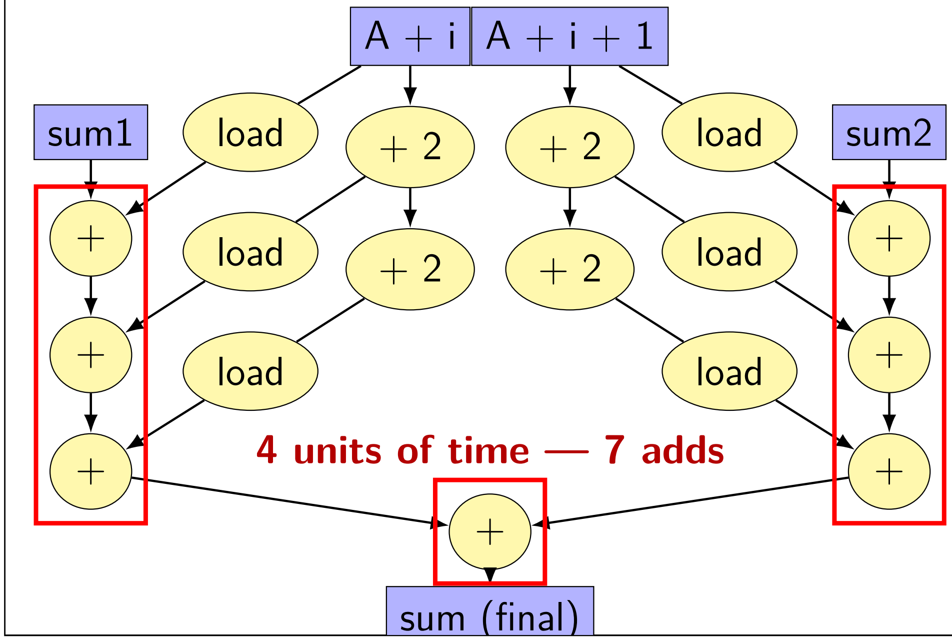


better data-flow



6 ops/time
two sum adds/time

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

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

multiple accumulators — multiply

same as before — but with multiply not add

accumulators	cycles/element	instructions/element
1	2.93(??)	1.21
2	1.51	1.21
4	1.02	1.23
8	1.03	1.24
16	1.05	1.64

throughput: 1 cycle/multiply (max of my hardware)

each takes ~3 cycles (according to Intel manual)

- max throughput: at least 3 active at any time

other loop unrolling notes

full loop unrolling can be really good

no loop overhead at all

may help compiler make other optimizations

- easier to reason about code without loop

compilers manage register usage

usually do a good job

keep things in registers if possible

but won't tell you if they start using the stack instead

common reason for "optimization" to hurt performance

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
    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
    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

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
```

constant multiplies/divides

compiler is very good at handling

…but need to actually use constants

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 1500` — record 1500/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

cache feature: prefetching

processors can bring values into cache before requested

called prefetching

method one: CPU looks for periodic access patterns

- mostly just makes code faster

method two: explicit hints from programmer ("prefetch instruction")

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

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:

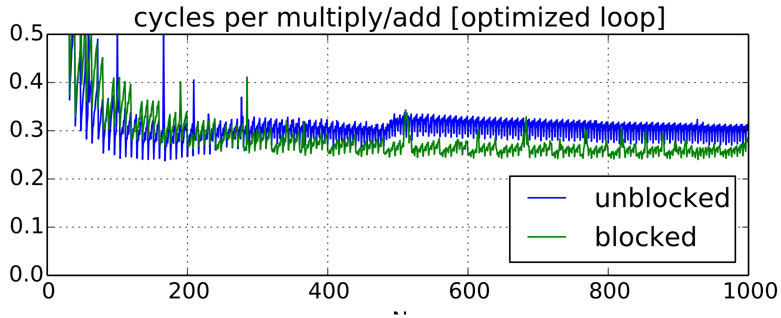
%xmm1: [6, 8, 10, 12]

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
```

wiggles on prior graphs



variance from this optimization

multiples of 8 were easier with naive implementation

vector instructions efficiency

do a lot more work per instruction

easy to implement: more copies of ALU

hard for compilers to use

- need to compress 4 loop iterations into one
- what if some operation doesn't have obvious instruction?
- what if there might be aliasing?

but modern compilers sometimes manage to do this

prefetching

processors try to fetch blocks into cache before requested

main method: look for periodic patterns

usually this is just automatic

if not — special instructions to explicitly trigger

...or make your pattern more periodic

branch prediction

unpredictable branches are really slow on modern CPUs

30+ mispredicted instructions squashed

what to do?

- conditional moves?
- less branches?

but — modern branch predictors usually right