Performance

Changelog

Corrections made in this version not in first posting:

- 12 April 2017: slide 31 shouldn't have had same C code twice
- 14 April 2017: slide 12: make it clearer that the inner part is another triply nested loop
- 3 May 2017: slide 43: switch Aik arithmetic so it actually make snse
- 5 May 2017: slide 57: replace "slower if" with "can be slower if"

---

performance assignments

partners or individual (your choice)

lab time for questions; we'll grade HW submission for each part

- you and partner must be able to make common lab time

two parts:

- rotate an image
- smooth (blur) an image

---

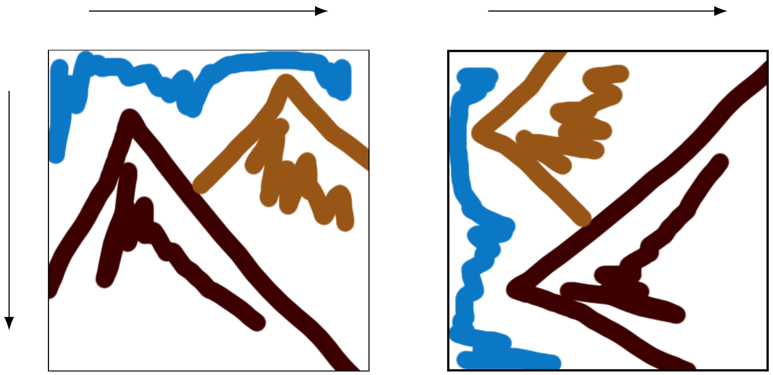
image representation

```
typedef struct { short red, green, blue; } 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

try techniques from book/lecture that seem applicable

for each assignment, one is most important

vary numbers (e.g. cache block size)

- often — too big/small is worse

some techniques combine well

---

review: cache performance

central idea: reorder accesses to avoid cache misses

example: matrix squaring

```
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];
```

access each element of B N^2 times, each element of A $2N^2$ times

naive order: a lot of these accesses are misses

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];
```

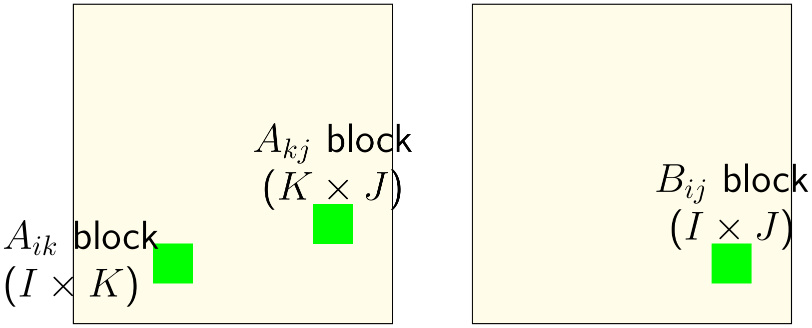
B_{ij} used K times for one miss — N^2/K misses

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

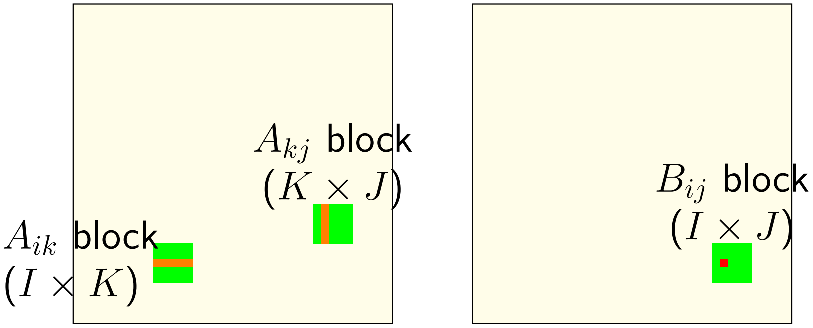
array usage: block



inner loop keeps “blocks” from A, B in cache

---

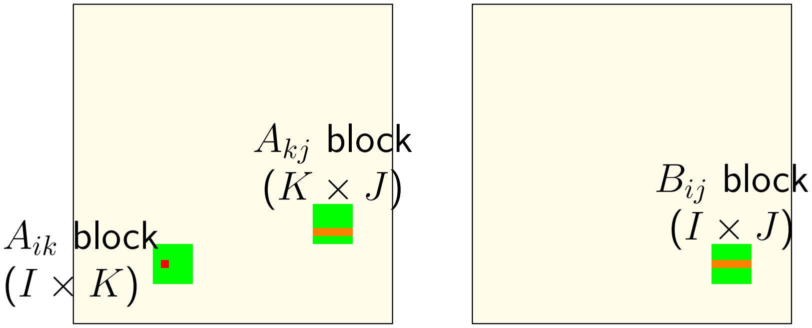
array usage: block



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

---

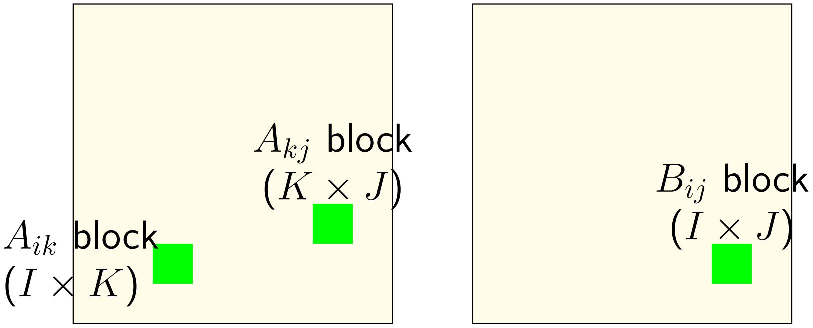
array usage: block



A_{ik} calculation uses strips from A, B
J calculations for one load (cache miss)

---

array usage: block



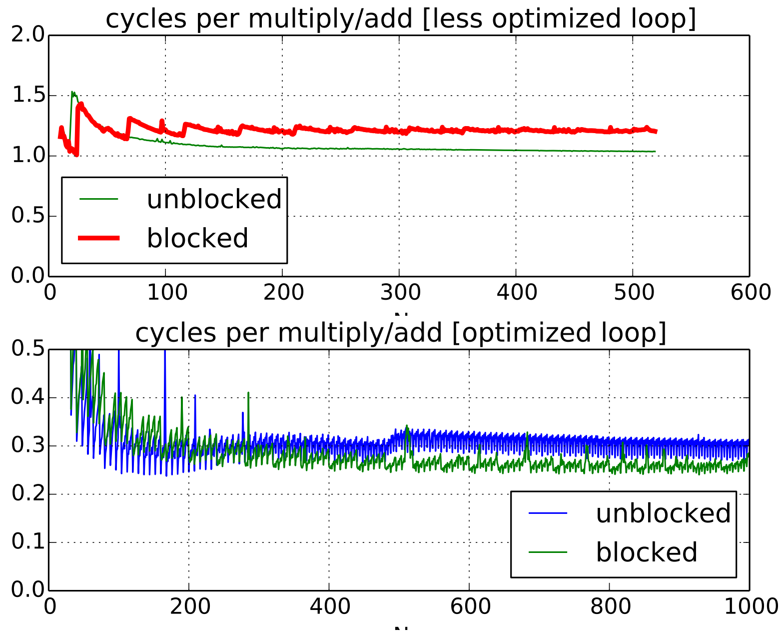
(approx.) KIJ fully cached calculations
for $KI + IJ + KJ$ loads
(assuming everything stays in cache)

cache-friendliness generally

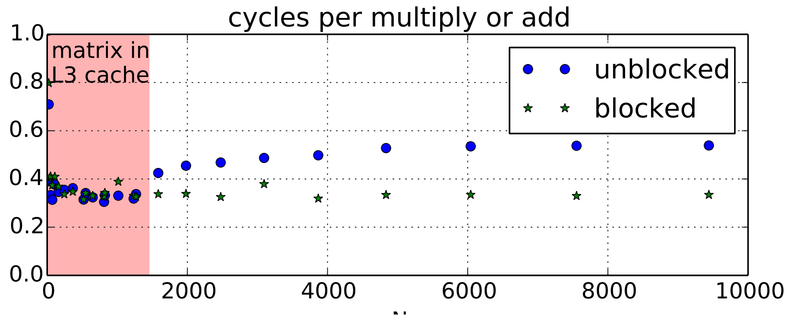
better spatial/temporal locality

best case: adapted to size of cache

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

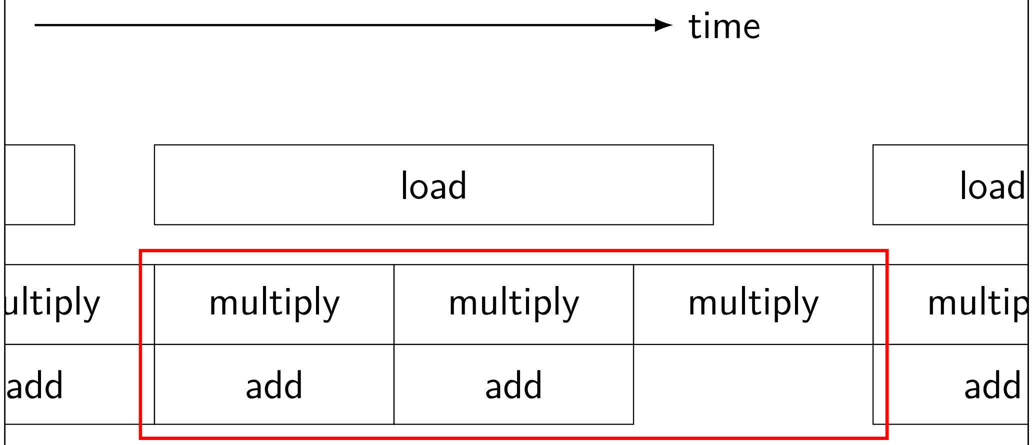
2: doing adds/multiplies/etc. with less instructions

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

---

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

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 stac
        addl    $1, -12(%rbp)   // increment i
condition:
        movl    -12(%rbp), %eax
        cmpl    -28(%rbp), %eax
```

---

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
```

---

optimizing compilers

these usually make your code fast

often not done by default

compilers and humans are good at different kinds of optimizations

compiler limitations

needs to generate code that does the same thing…

- …even in corner cases that “obviously don’t matter”

often doesn’t ‘look into’ a method

- needs to assume it might do anything

can’t predict what inputs/values will be

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

can’t understand code size versus speed tradeoffs

---

compiler limitations

needs to generate code that does the same thing…

- …even in corner cases that “obviously don’t matter”

often doesn’t ‘look into’ a method

- needs to assume it might do anything

can’t predict what inputs/values will be

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

can’t understand code size versus speed tradeoffs

---

aliasing

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

the compiler **cannot** generate this:

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

---

aliasing problem

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

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

non-contrived aliasing

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

```
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;
    }
}
```

---

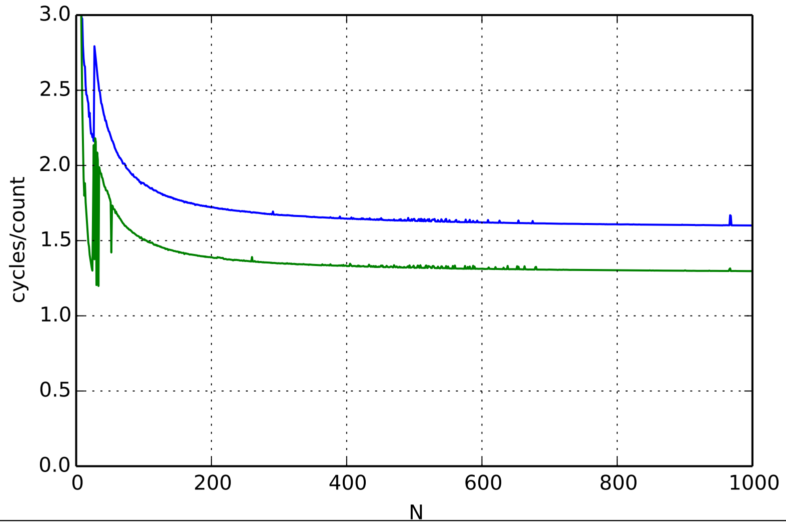
non-contrived aliasing

```
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;
    }
}
```

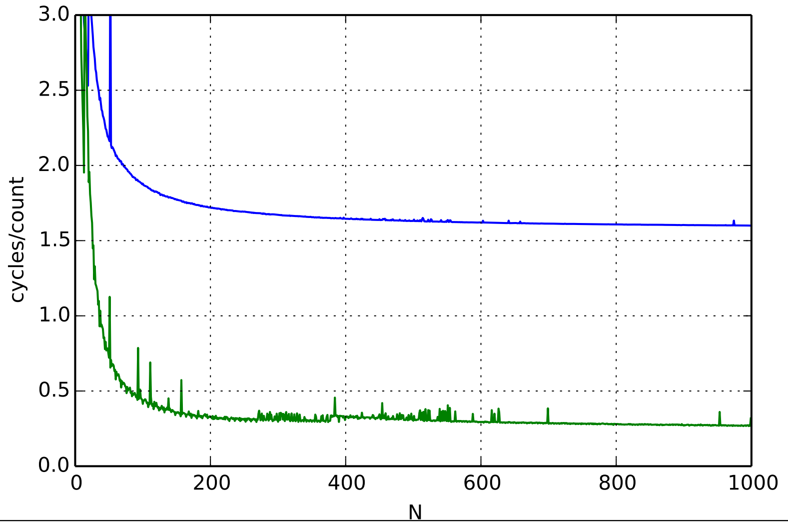
---

aliasing and performance (1) / GCC 5.4 -O2



---

aliasing and performance (2) / GCC 5.4 -O3



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

---

redundant loads

optimization: avoid redundant loads

- slower even if always hits cache

instead: use registers

compiler will do this — if it knows aliasing doesn't matter

---

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;
    }
}
```

---

redundant load?

```
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];
```

avoiding redundant load here?

remove redundant load

```
for (int k = 0; k < N; ++k) {
  for (int i = 0; i < N; ++i) {
    // make it easier for compiler
    // to keep this in a register
    float Aik = A[i * N + k];
    for (int j = 0; j < N; ++j)
      B[i*N+j] += Aik * A[k * N + j];
  }
}
```

---

exposing more redundant loads

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

exercise: what is loaded repeatedly from cache?

---

exposing more redundant loads

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

exercise: what is loaded repeatedly from cache?

---

eliminate loads of Bij

```
for (int kk = 0; k + 2 <= N; kk += 2) { // assume
  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;
    }
  }
}
```

eliminate loads of Bij

```
for (int kk = 0; k + 2 <= N; kk += 2) { // assume
  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;
    }
  }
}
```

---

eliminate loads of Aik

```
for (int kk = 0; k + 2 <= N; kk += 2) { // assume
  for (int i = 0; i < N; ++i) {
    float Aik0 = A[i * N + k];
    float Aik1 = A[i * N + k + 1];
    for (int j = 0; j < N; ++j) {
      float Bij = B[i * N + j];
      Bij += Aik0 * A[k * N + j];
      Bij += Aik1 * A[(k + 1) * N + j];
      B[i * N + j] = Bij;
    }
  }
}
```

---

eliminate loads of Aik

```
for (int kk = 0; k + 2 <= N; kk += 2) { // assume
  for (int i = 0; i < N; ++i) {
    float Aik0 = A[i * N + k];
    float Aik1 = A[i * N + k + 1];
    for (int j = 0; j < N; ++j) {
      float Bij = B[i * N + j];
      Bij += Aik0 * A[k * N + j];
      Bij += Aik1 * A[(k + 1) * N + j];
      B[i * N + j] = Bij;
    }
  }
}
```

---

register blocking

```
for (int k = 0; k + 2 <= N; k += 2) { // assume N even
  for (int i = 0; i + 2 <= N; i += 2) {
    float A_i_0_k_0 = A[(i + 0) * N + (k + 0)];
    float A_i_0_k_1 = A[(i + 0) * N + (k + 1)];
    float A_i_1_k_0 = A[(i + 1) * N + (k + 0)];
    float A_i_1_k_1 = A[(i + 1) * N + (k + 1)];
    for (int j = 0; j + 1 <= N; j += 1) {
        float B_i_0_j_0 = B[(i + 0) * N + (j + 0)];
        float B_i_1_j_0 = B[(i + 1) * N + (j + 0)];
        float A_k_0_j_0 = A[(k + 0) * N + (j + 0)];
        float A_k_1_j_0 = A[(k + 1) * N + (j + 0)];
        B_i_0_j_0 += A_i_0_k_0 * A_k_0_j_0 + A_i_0_k_1 * A_k_1_j_0;
        B_i_1_j_0 += A_i_1_k_0 * A_k_0_j_0 + A_i_1_k_1 * A_k_1_j_0;
        B[(i + 0) * N + (j + 0)] = B_i_0_j_0;
        B[(i + 1) * N + (j + 0)] = B_i_1_j_0;
    }
  }
}
```

idea: compiler uses about 8 registers for values

avoid reloading `A_i_0_k_0`, etc. from cache

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) {
    ...
}
```

---

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

---

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 sumWithLimit(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 = sumWithLimit(sum, array[i]);
    return sum;
}
```

loop with a function call

```
int sumWithLimit(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 = sumWithLimit(sum, array[i]);
    return sum;
}
```

function call assembly

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

extra instructions: 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, etc.

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 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"

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 in assembly

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

most instructions are loop maintainence

loop in assembly

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

most instructions are loop maintainence

---

loop in assembly

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

most instructions are loop maintainence

---

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?

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)

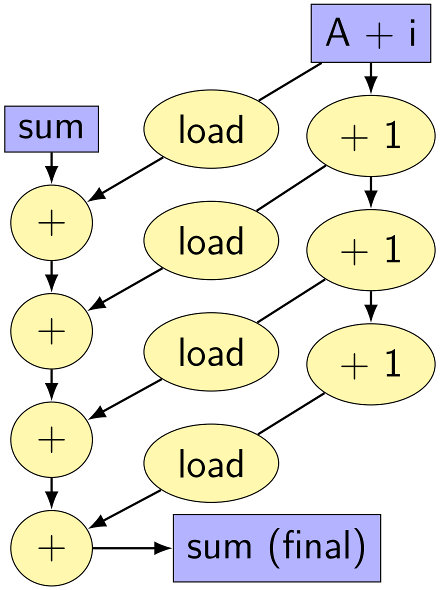
times unrolled	cycles/element	instructions/element
1	1.33	4.02
2	1.03	2.52
4	1.02	1.77
8	1.01	1.39
16	1.01	1.21
32	1.01	1.15

instruction cache/etc. overhead

1.01 cycles/element — latency bound

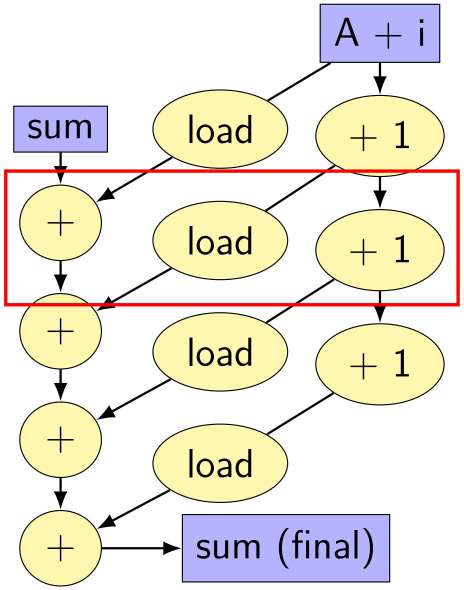
---

data flow model and limits



---

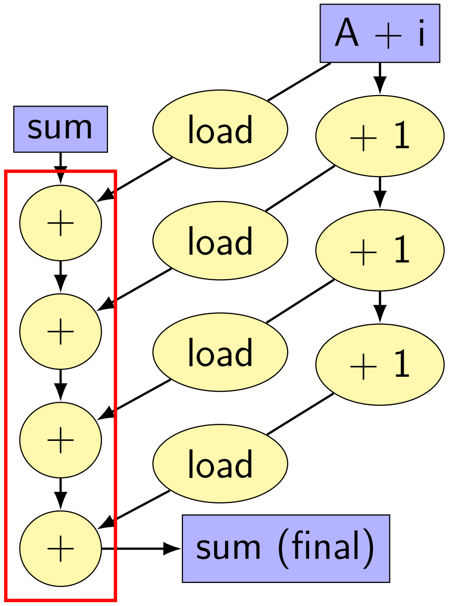
data flow model and limits



three ops/cycle (if each one cyc

---

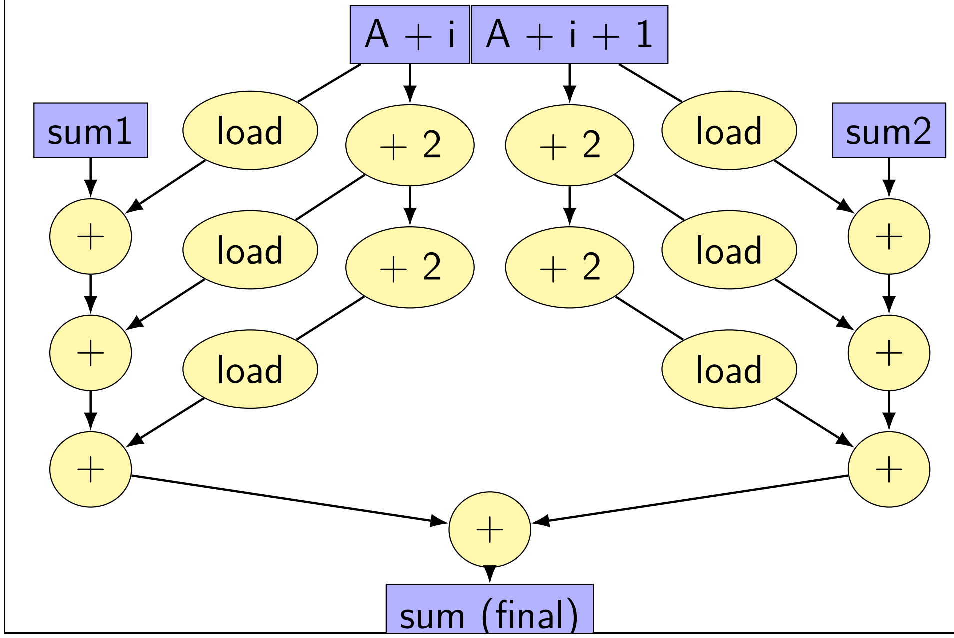
data flow model and limits



need to do additions one-at-a-time

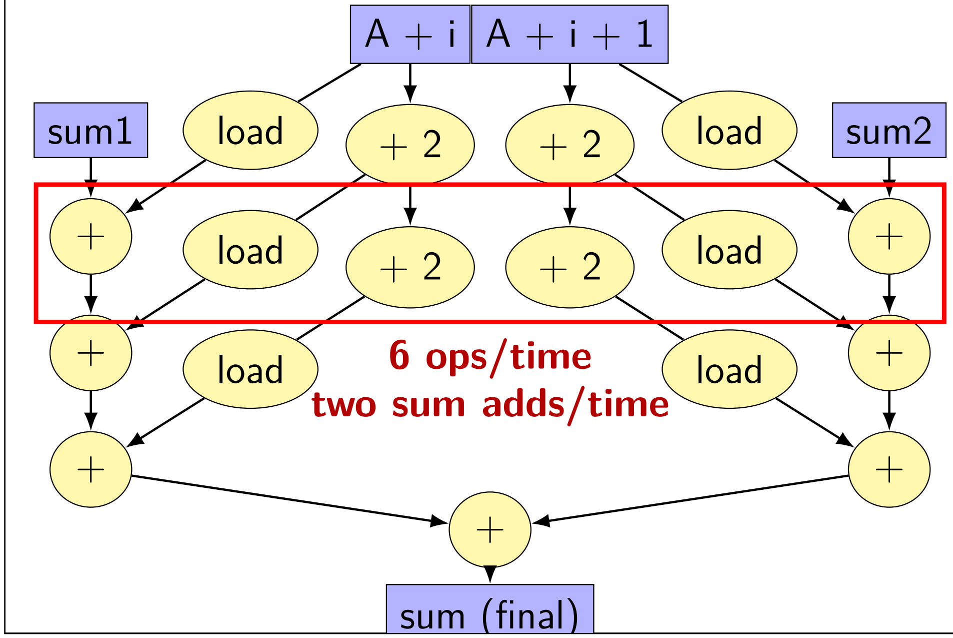
book's name: critical path

better data-flow



---

better data-flow

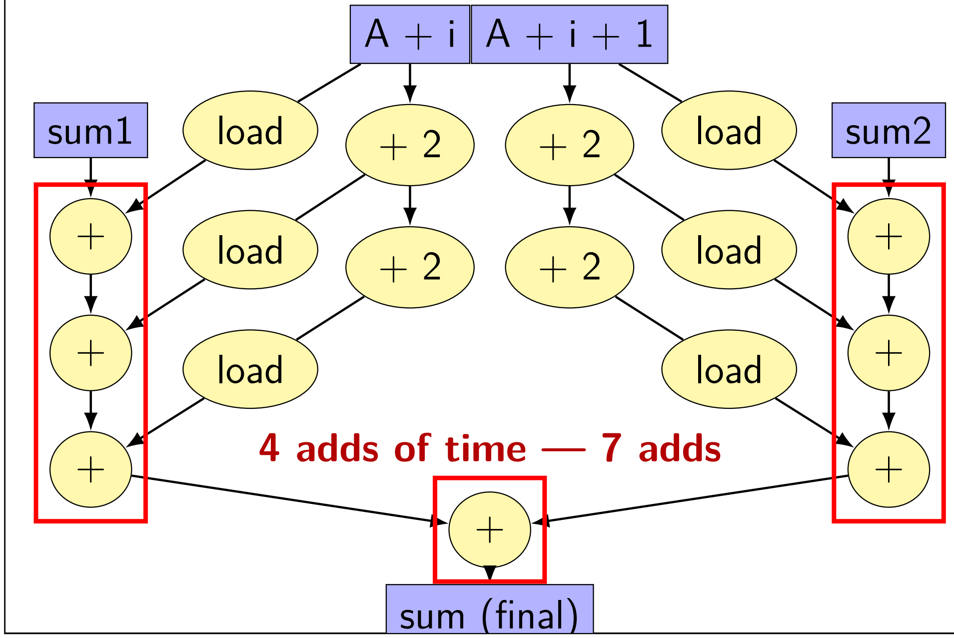


6 ops/time

two sum adds/time

---

better data-flow



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;
```

<

1-2>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

<

---

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

---

other loop unrolling notes

full loop unrolling can be really good

no loop overhead at all

helps 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

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

---

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