optimization 2

Changelog

Changes made in this version not seen in first lecture:

- 30 October: 8 accumulator assembly: correct assembly so sum1/sum2 have different registers
- 30 October: aliasing and cache optimization: correct condition second loop in second set of loops to use j
- 30 October: aliasing and cache optimization: adjust to use A, B, C to be more consistent with other cache blocking examples we've given
- 30 October: register blocking: temporary Bkj0, etc. variable should've retrieved value from B, not A.

slide errors last time

made a bunch of wrong register number errors on second register renaming example slide

(I definitely added that example too hastily...)

fixed in PDF, I hope

Sorry!

I hope this didn't confuse people for the quiz...

last time

one way to make OOO processors

- register renaming to detect conflicts
- dispatch from an instruction queue

disclaimer: some variation on that in real world

data flow model

- graph: vertices = operations; edges = moving value
- slowest path from beginning to end determines performance
- e.g. all adds one after the other
- “latency bound”

reassociation: $(((a \times b) \times c) \times d) \rightarrow ((a \times b) \times (c \times d))$

- shallower data flow graph — more done in parallel

multiple accumulators: reassocation applied to loops

logistics note: exam/homework

midterm 2 next week

pipehw2 due after it

this is not because we want you to wait

students often find it harder than pipehw1

- because stalling/branch prediction is tricky

topics in pipehw2 are on the midterm

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), %rax          // 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 (part of mov)

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

- over 16 because loop unrolled 16 times
- loop overhead
- compiler not as efficient as it could have been

2+3/16+1/16 = 2+1/4 instructions per element

probably 2+1/4 microinstructions, too

- cmp+jXX apparently becomes 1 microinstruction (on this Intel CPU)
- probably extra microinstruction for load in add

hardware limits on my machine

4(?) register renamings per cycle

(Intel doesn't really publish exact numbers here...)

4-6 instructions decoded/cycle

(depending on instructions)

4(?) microinstructions commited/cycle

4 (add or cmp+branch executed)/cycle

hardware limits on my machine

4(?) register renamings per cycle

(Intel doesn't really publish exact numbers here...)

4-6 instructions decoded/cycle

(depending on instructions)

4(?) microinstructions commited/cycle

4 (add or cmp+branch executed)/cycle

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

getting over this limit

the $+1/4$ was from loop overhead

solution: more loop unrolling!

common theme with optimization:

fix one bottleneck (need to do adds one after the other)

find another bottleneck

example assembly (unoptimized)

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

```
sum:    ...
the_loop:
        ...
        leaq    0(,%rax,8), %rdx// offset ← i * 8
        movq    −24(%rbp), %rax // get A from stack
        addq    %rdx, %rax      // add offset
        movq    (%rax), %rax    // get *(A+offset)
        addq    %rax, −8(%rbp)  // add to sum, on stack
        addl    $1, −12(%rbp)   // increment i
condition:
        movl    −12(%rbp), %eax
        cmpl    −28(%rbp), %eax
        jl      the_loop
        ...
```

example assembly (gcc 5.4 -Os)

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

sum:
        xorl    %edx, %edx
        xorl    %eax, %eax
the_loop:
        cmpl    %edx, %esi
        jle     done
        addq    (%rdi,%rdx,8), %rax
        incq    %rdx
        jmp     the_loop
done:
        ret
```

example assembly (gcc 5.4 -O2)

```
long sum(long *A, int N) {
    long result = 0;
    for (int i = 0; i < N; ++i)
        result += A[i];
    return result;
}
sum:
        testl   %esi, %esi
        jle     return_zero
        leal    −1(%rsi), %eax
        leaq    8(%rdi,%rax,8), %rdx // rdx=end of A
        xorl    %eax, %eax
the_loop:
        addq    (%rdi), %rax // add to sum
        addq    $8, %rdi     // advance pointer
        cmpq    %rdx, %rdi
        jne     the_loop
        rep ret
return_zero:    ...
```

example assembly (gcc 9.2 -O3)

```
sum:
        testl   %esi, %esi
        ... /* approx 10 lines omitted */
the_loop:
        movdqu  (%rax), %xmm2 /* ←- load 16 bytes from array */
        addq    $16, %rax
        paddq   %xmm2, %xmm0  /* ←- add 2 pairs of longs */
        cmpq    %rdx, %rax
        jne     the_loop
        ... /* approx 20 lines omitted */
        ret
```

example assembly (gcc 9.2 -O3 -march=skylake)

```
sum:
        testl   %esi, %esi
        ... /* approx 10 lines omitted */
the_loop:
        vpaddq  (%rax), %ymm0, %ymm0  /* ← add 4 pairs of longs */
        addq    $32, %rax
        cmpq    %rdx, %rax
        jne     the_loop
        ... /* approx 20 lines omitted */
        ret
```

gcc 9.2 -O3 -funroll-loops -march=skylake

```
sum:
        testl   %esi, %esi
        ... /* approx 60 lines omitted */
the_loop:  /* loop unrolled 8 times + instrs that add 4 pairs at a t
        vpaddq  (%r8), %ymm0, %ymm1  /* ←- add 4 pairs of longs */
        addq    $256, %r8
        vpaddq  −224(%r8), %ymm1, %ymm2
        vpaddq  −192(%r8), %ymm2, %ymm3
        vpaddq  −160(%r8), %ymm3, %ymm4
        vpaddq  −128(%r8), %ymm4, %ymm5
        vpaddq  −96(%r8), %ymm5, %ymm6
        vpaddq  −64(%r8), %ymm6, %ymm7
        vpaddq  −32(%r8), %ymm7, %ymm0
        cmpq    %rcx, %r8
        jne     .L4
        ... /* approx 20 lines omitted */
        ret
```

example assembly (clang 9.0 -O -march=skylake)

```
sum:
        testl   %esi, %esi
        ... /* approx 35 lines omitted */
the_loop: /* loop unrolled + multiple accumulators + instrs that 4 pairs at a time */
        vpaddq  (%rdi,%rsi,8), %ymm0, %ymm0
        vpaddq  32(%rdi,%rsi,8), %ymm1, %ymm1
        vpaddq  64(%rdi,%rsi,8), %ymm2, %ymm2
        vpaddq  96(%rdi,%rsi,8), %ymm3, %ymm3
        vpaddq  128(%rdi,%rsi,8), %ymm0, %ymm0
        vpaddq  160(%rdi,%rsi,8), %ymm1, %ymm1
        vpaddq  192(%rdi,%rsi,8), %ymm2, %ymm2
        vpaddq  224(%rdi,%rsi,8), %ymm3, %ymm3
        vpaddq  256(%rdi,%rsi,8), %ymm0, %ymm0
        vpaddq  288(%rdi,%rsi,8), %ymm1, %ymm1
        vpaddq  320(%rdi,%rsi,8), %ymm2, %ymm2
        vpaddq  352(%rdi,%rsi,8), %ymm3, %ymm3
        vpaddq  384(%rdi,%rsi,8), %ymm0, %ymm0
        vpaddq  416(%rdi,%rsi,8), %ymm1, %ymm1
        vpaddq  448(%rdi,%rsi,8), %ymm2, %ymm2
        vpaddq  480(%rdi,%rsi,8), %ymm3, %ymm3
        addq    $64, %rsi
        addq    $4, %rax
        jne the_loop
```

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

loop unrolling downsides

bigger executables $\rightarrow$ instruction cache misses

slower if small number of loop iterations

- extra code to handle leftovers, etc.

want to unroll loops that are run a lot and quick to execute

problem: compiler probably can't tell if this meets those criteria

```
for (int i = 0; i < some_variable; ++i) {
  sum += some_function();
}
```

figuring out how to unroll?

exercise: why can the compiler probably not do this transformation?

```
void foo() { int sum = 0;
  for (int i = 0; i < some_global_variable; ++i) {
    sum += some_function();
  }
}
```

```
void foo_transformed() { int sum = 0;
  int i = 0;
  if (some_global_variable % 2 == 1) {
    i += 1;
    sum += some_function();
  }
  for (; i < some_global_variable; i += 2) {
    sum += some_function();
    sum += some_function();
  }
}
```

multiple accumulators downsides

downsides of loop unrolling

bigger executables, slower if small number of iterations

+ uses extra registers (can't use those regs for something else)

want to use multiple accumulators if latency likely bottleneck

problem: compiler probably can't tell if this meets those criteria

```
for (int i = 0; i < some_variable; ++i) {
  sum += some_function();
}
```

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

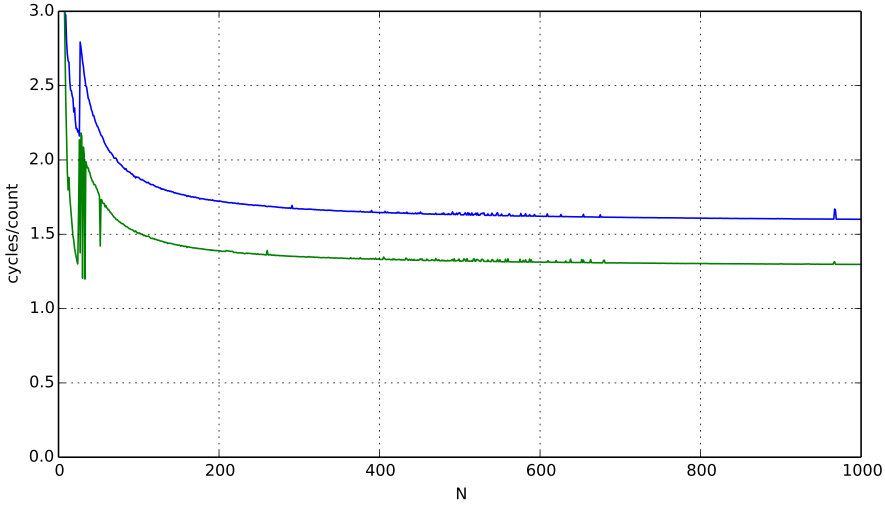
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

non-contrived aliasing

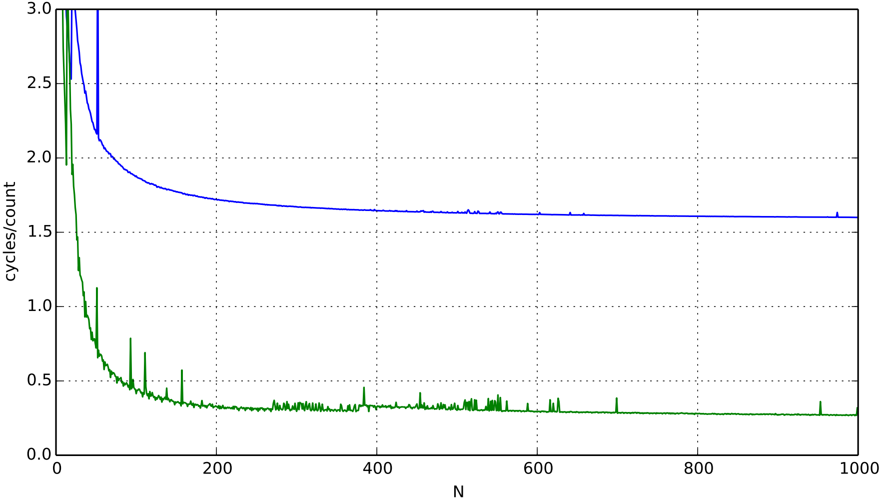
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

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

aliasing and performance (1) / GCC 5.4 -O2



aliasing and performance (2) / GCC 5.4 -O3



automatic register reuse

Compiler would need to generate overlap check:

```
if (result > matrix + N * N || result < matrix) {
    for (int row = 0; row < N; ++row) {
        int sum = 0; /* kept in register */
        for (int col = 0; col < N; ++col)
            sum += matrix[row * N + col];
        result[row] = sum;
    }
} else {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
        }
    }
}
```

aliasing and cache optimizations

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

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

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

compiler can't generate same code for both

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 = B[k*N + j+0];
      float Bkj1 = B[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;
    }
  }
}
```

caching blocking + loop unroll + no alias (1)

blocking for k, i (missing j), plus 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) {
      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 blocking + loop unroll + no alias (1)

blocking for k, i (missing j), plus 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) {
      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 blocking + loop unroll + no alias (2)

plus explicitly unroll loop over k values

```
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];
      float Bk0j = B[(k+0)*N+j];
      float Bk1j = B[(k+1)*N+j];
      Ci0j += A[(i+0)*N+k] * Bk0j;
      Ci0j += A[(i+0)*N+k+1] * Bk1j;
      Ci1j += A[(i+1)*N+k] * Bk0j;
      Ci1j += A[(i+1)*N+k+1] * Bk1j;
      C[(i+0)*N+j] += Ci0j;
      C[(i+1)*N+j] += Ci1j;
    }
```

caching blocking + loop unroll + no alias (2)

plus explicitly unroll loop over k values

```
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];
      float Bk0j = B[(k+0)*N+j];
      float Bk1j = B[(k+1)*N+j];
      Ci0j += A[(i+0)*N+k] * Bk0j;
      Ci0j += A[(i+0)*N+k+1] * Bk1j;
      Ci1j += A[(i+1)*N+k] * Bk0j;
      Ci1j += A[(i+1)*N+k+1] * Bk1j;
      C[(i+0)*N+j] += Ci0j;
      C[(i+1)*N+j] += Ci1j;
    }
```

compiler limitations

needs to generate code that does the same thing...

...even in corner cases that "obviously don't matter"

often doesn't 'look into' a method

needs to assume it might do anything

can't predict what inputs/values will be

e.g. lots of loop iterations or few?

can't understand code size versus speed tradeoffs

loop with a function call

```
int addWithLimit(int x, int y) {
    int total = x + y;
    if (total > 10000)
        return 10000;
    else
        return total;
}
...
int sum(int *array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum = addWithLimit(sum, array[i]);
    return sum;
}
```

loop with a function call

```
int addWithLimit(int x, int y) {
    int total = x + y;
    if (total > 10000)
        return 10000;
    else
        return total;
}
...
int sum(int *array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum = addWithLimit(sum, array[i]);
    return sum;
}
```

function call assembly

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

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

manual inlining

```
int sum(int *array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + array[i];
        if (sum > 10000)
            sum = 10000;
    }
    return sum;
}
```

inlining pro/con

avoids call, ret, extra move instructions

allows compiler to use more registers

- no caller-saved register problems

but not always faster:

worse for instruction cache

- (more copies of function body code)

compiler inlining

compilers will inline, but…

will usually avoid making code much bigger

- heuristic: inline if function is small enough
- heuristic: inline if called exactly once

will usually not inline across .o files

some compilers allow hints to say “please inline/do not inline this function”

remove redundant operations (1)

```
int number_of_As(const char *str) {
    int count = 0;
    for (int i = 0; i < strlen(str); ++i) {
        if (str[i] == 'a')
            count++;
    }
    return count;
}
```

remove redundant operations (1, fix)

```
int number_of_As(const char *str) {
    int count = 0;
    int length = strlen(str);
    for (int i = 0; i < length; ++i) {
        if (str[i] == 'a')
            count++;
    }
    return count;
}
```

call strlen once, not once per character!

Big-Oh improvement!

remove redundant operations (1, fix)

```
int number_of_As(const char *str) {
    int count = 0;
    int length = strlen(str);
    for (int i = 0; i < length; ++i) {
        if (str[i] == 'a')
            count++;
    }
    return count;
}
```

call strlen once, not once per character!

Big-Oh improvement!

remove redundant operations (2)

```
int shiftArray(int *source, int *dest, int N, int amount) {
    for (int i = 0; i < N; ++i) {
        if (i + amount < N)
            dest[i] = source[i + amount];
        else
            dest[i] = source[N - 1];
    }
}
```

compare i + amount to N many times

remove redundant operations (2, fix)

```
int shiftArray(int *source, int *dest, int N, int amount) {
    int i;
    for (i = 0; i + amount < N; ++i) {
        dest[i] = source[i + amount];
    }
    for (; i < N; ++i) {
        dest[i] = source[N - 1];
    }
}
```

eliminate comparisons

performance labs

week after exam — loop optimizations

after that — vector instructions (AKA SIMD)

performance HWs

INDIVIDUAL ONLY

assignment 1: rotate an image

assignment 2: smooth (blur) an image

- two parts
- part 1: due with rotate — optimizations we've mostly talked about
- part 2: due later — part with vector instructions

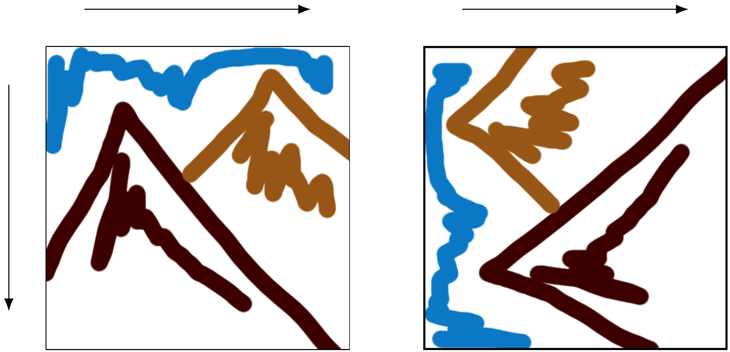
image representation

```
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

- sometime after midterm: 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

addressing efficiency

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

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 Cij = C[i * N + j];
      float *Bkj_pointer = &B[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] * Bkj_pointer;
        Bkj_pointer += N;
      }
      C[i * N + j] = Bij;
    }
  }
```

transforms loop to iterate with pointer

compiler will often do this

increment/decrement by N ($\times$ sizeof(float))

addressing transformation

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

transforms loop to iterate with pointer

compiler will often do this

increment/decrement by N ($\times$ sizeof(float))

addressing efficiency

compiler will usually eliminate slow multiplies

doing transformation yourself often slower if so

`i * N; ++i` into `i_times_N; i_times_N += N`

way to check: see if assembly uses lots multiplies in loop

if it doesn't — do it yourself

another addressing transformation

```
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
int offset = 0;
float *Ai0_base = &A[k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += Ai0_base[offset] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[offset] * B[k * n + j];
    // ...
    offset += n;
```

compiler will sometimes do this, too

another addressing transformation

```
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
int offset = 0;
float *Ai0_base = &A[k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += Ai0_base[offset] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[offset] * B[k * n + j];
    // ...
    offset += n;
```

compiler will sometimes do this, too

another addressing transformation

```
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
int offset = 0;
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
    offset += n;
```

storing 20 `AiX_base`? — need the stack

maybe faster (quicker address computation)

maybe slower (can't do enough loads)

another addressing transformation

```
    for (int i = 0; i < n; i += 20) {
        C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
        C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
        // ...
```

```
    int offset = 0;
    float *Ai0_base = &A[0*n+k];
    float *Ai1_base = Ai0_base + n;
    float *Ai2_base = Ai1_base + n;
    // ...
    for (int i = 0; i < n; i += 20) {
        C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
        C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
        // ...
        offset += n;
```

storing 20 `AiX_base`? — need the stack

maybe faster (quicker address computation)

maybe slower (can't do enough loads)

alternative addressing transformation

instead of:

```
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

could do:

```
float *Ai0_base = &A[k];
for (int i = 0; i < n; i += 20) {
    float *A_ptr = &Ai0_base[i*n];
    C[(i+0) * n + j] += *A_ptr * A[k * n + j];
    A_ptr += n;
    C[(i+1) * n + j] += *A_ptr * B[k * n + j];
    // ...
```

avoids spilling on the stack, but more dependencies

alternative addressing transformation

instead of:

```
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

---

could do:

```
float *Ai0_base = &A[k];
for (int i = 0; i < n; i += 20) {
    float *A_ptr = &Ai0_base[i*n];
    C[(i+0) * n + j] += *A_ptr * A[k * n + j];
    A_ptr += n;
    C[(i+1) * n + j] += *A_ptr * B[k * n + j];
    // ...
```

avoids spilling on the stack, but more dependencies

addressing efficiency generally

mostly: compiler does very good job itself

- eliminates multiplications, use pointer arithmetic
- often will do better job than if how typically programming would do it manually

sometimes compiler won't take the best option

- if spilling to the stack: can cause weird performance anomalies
- if indexing gets too complicated — might not remove multiply

if compiler doesn't, you can always make addressing simple yourself

- convert to pointer arith. without multiplies