Optimization part 1

Changelog

Changes made in this version not seen in first lecture:

- 29 Feb 2018: loop unrolling performance: remove bogus instruction cache overhead remark
- 29 Feb 2018: spatial locality in Akj: correct reference to $B_{k+1,j}$ to $A_{k+1,j}$

last time

what things in C code map to same set?

- key idea: if bytes per way apart from each other

finding conflict misses in C

- how "overloaded" is each cache set

cache 'blocking' for matrix-like code

- maximize work per cache miss

some logistics

exam next week

everything up to and including this lecture

yes, I know office hours were very slow...

like to think about how to help with

- 'group' office hours?
- better tools?
- different priorities on queue?

view as an explicit cache

imagine we explicitly moved things into cache

original loop:

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

view as an explicit cache

imagine we explicitly moved things into cache

blocking in k:

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

calculation counting with explicit cache

before: load ~ 2 values for one add+multiply

after: load ~ 3 values for two add+multiply

simple blocking: temporal locality in Bij

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
```

$B_{i+0,j}$ += $A_{i+0,k+0}$ * $A_{k+0,j}$

$B_{i+0,j}$ += $A_{i+0,k+1}$ * $A_{k+1,j}$

$B_{i+1,j}$ += $A_{i+1,k+0}$ * $A_{k+0,j}$

$B_{i+1,j}$ += $A_{i+1,k+1}$ * $A_{k+1,j}$

```
    }
```

before: B_{ij}s accessed once, then not again for N^2 iters

after: B_{ij}s accessed twice, then not again for N^2 iters (next k)

simple blocking: temporal locality in Akj

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
```

$$B_{i+0,j} \mathrel{+}= A_{i+0,k+0} * A_{k+0,j}$$
$$B_{i+0,j} \mathrel{+}= A_{i+0,k+1} * A_{k+1,j}$$
$$B_{i+1,j} \mathrel{+}= A_{i+1,k+0} * A_{k+0,j}$$
$$B_{i+1,j} \mathrel{+}= A_{i+1,k+1} * A_{k+1,j}$$

```
    }
```

before blocking: A_{kj}s accessed once, then not again for N iters

after blocking: A_{kj}s accessed twice, then not again for N iters (next i)

simple blocking: temporal locality in Aik

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
```

$$B_{i+0,j} \mathrel{+}= A_{i+0,k+0} * A_{k+0,j}$$
$$B_{i+0,j} \mathrel{+}= A_{i+0,k+1} * A_{k+1,j}$$
$$B_{i+1,j} \mathrel{+}= A_{i+1,k+0} * A_{k+0,j}$$
$$B_{i+1,j} \mathrel{+}= A_{i+1,k+1} * A_{k+1,j}$$

```
    }
```

before: A_{ik}s accessed N times, then never again

after: A_{ik}s accessed N times

- but other parts of A_{ik} accessed in between
- slightly less temporal locality

simple blocking: spatial locality in Bij

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
```

$B_{i+0,j}$ += $A_{i+0,k+0}$ * $A_{k+0,j}$
$B_{i+0,j}$ += $A_{i+0,k+1}$ * $A_{k+1,j}$
$B_{i+1,j}$ += $A_{i+1,k+0}$ * $A_{k+0,j}$
$B_{i+1,j}$ += $A_{i+1,k+1}$ * $A_{k+1,j}$

```
    }
```

before blocking: perfect spatial locality ($B_{i,j}$ and $B_{i,j+1}$ adjacent)

after blocking: slightly less spatial locality

$B_{i,j}$ and $B_{i+1,j}$ far apart (N elements)
but still $B_{i,j+1}$ accessed iteration after $B_{i,j}$ (adjacent)

simple blocking: spatial locality in Akj

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
```

$$
\begin{array}{llll}
B_{i+0,j} & \texttt{+=} & A_{i+0,k+0} & \star\ A_{k+0,j} \\
B_{i+0,j} & \texttt{+=} & A_{i+0,k+1} & \star\ A_{k+1,j} \\
B_{i+1,j} & \texttt{+=} & A_{i+1,k+0} & \star\ A_{k+0,j} \\
B_{i+1,j} & \texttt{+=} & A_{i+1,k+1} & \star\ A_{k+1,j}
\end{array}
$$

```
    }
```

before: perfect spatial locality ($A_{k,j}$ and $B_{k,j+1}$ adjacent)

after: slightly less spatial locality

$A_{k,j}$ and $A_{k+1,j}$ far apart (N elements)

but still $A_{k,j+1}$ accessed iteration after $B_{k,j}$ (adjacent)

simple blocking: spatial locality in Aik

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
```

$$B_{i+0,j} \mathrel{+}= A_{i+0,k+0} * A_{k+0,j}$$
$$B_{i+0,j} \mathrel{+}= A_{i+0,k+1} * A_{k+1,j}$$
$$B_{i+1,j} \mathrel{+}= A_{i+1,k+0} * A_{k+0,j}$$
$$B_{i+1,j} \mathrel{+}= A_{i+1,k+1} * A_{k+1,j}$$

```
    }
```

before: very poor spatial locality ($A_{i,k}$ and $A_{i+1,k}$ far apart)

after: some spatial locality

- $A_{i,k}$ and $B_{i+1,k}$ still far apart (N elements)
- but still $A_{i,k}$ accessed together with $A_{i,k+1}$

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

cache blocking overview

reorder calculations

typically work in square-ish chunks of input

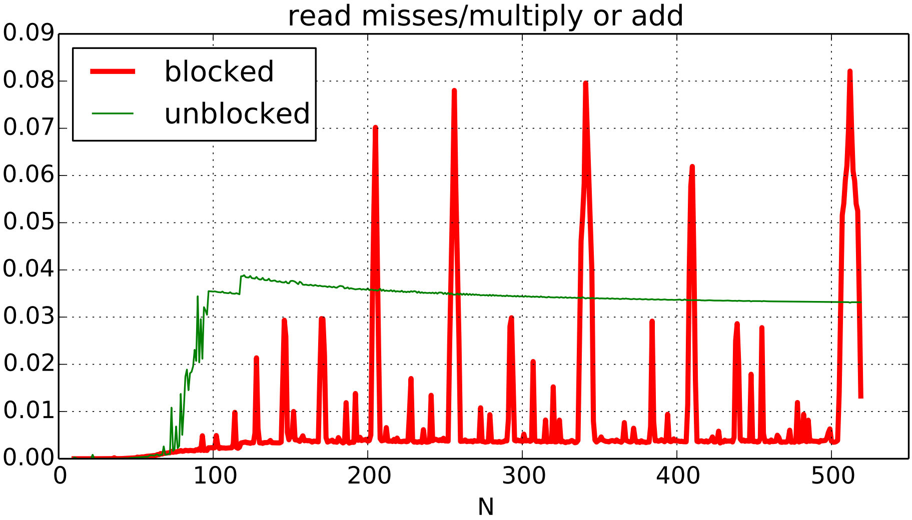
goal: maximum calculations per load into cache

- typically: use *every* value several times after loading it

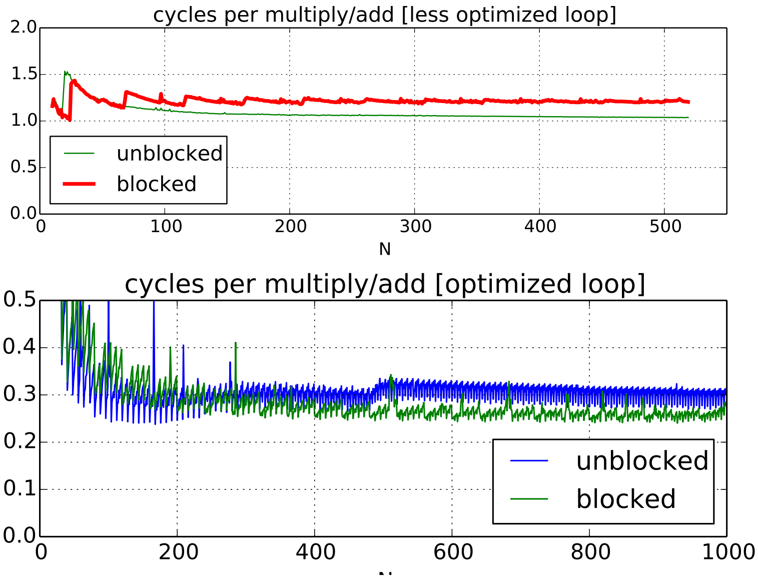
versus naive loop code:

- some values loaded, then used *once*
- some values loaded, then used *all possible times*

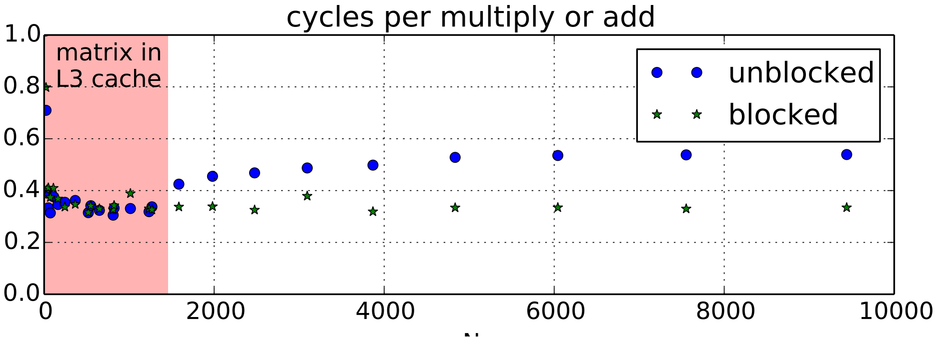
cache blocking and miss rate



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)

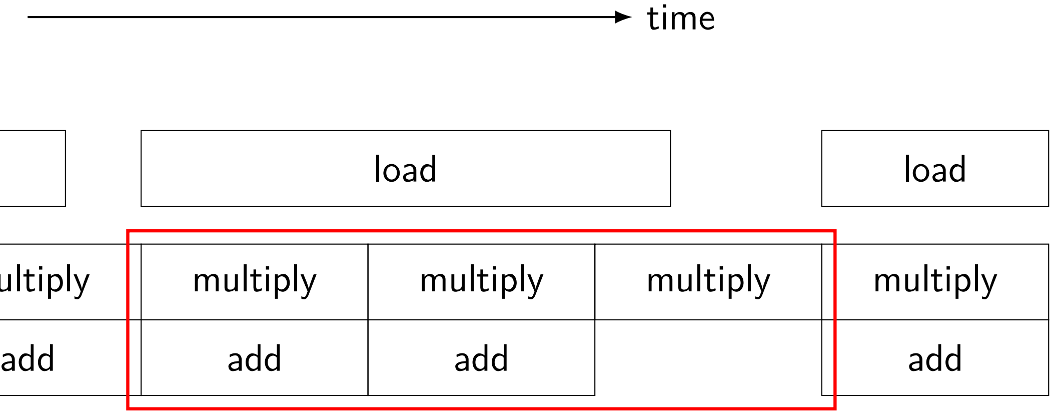
1: reducing number of loads

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

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:    ...
```

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

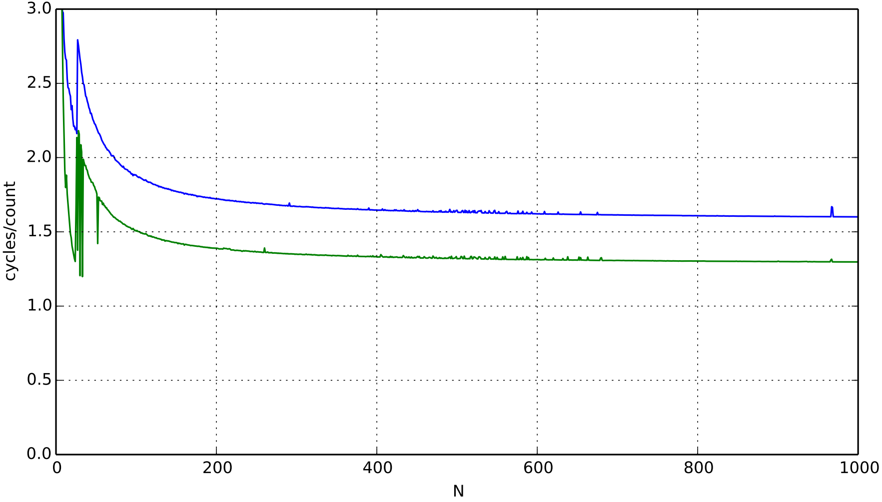
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

non-contrived aliasing

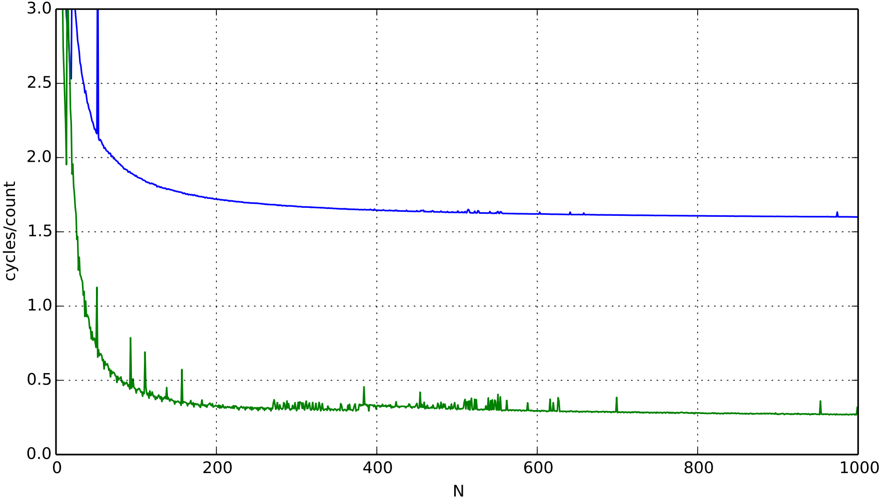
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

```
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)
      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

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) {
      B[(i+0)*N + j+0] += A[i*N+k] * A[k*N+j];
      B[(i+1)*N + j+0] += A[(i+1)*N+k] * A[k*N+j];
      B[(i+0)*N + j+1] += A[i*N+k] * A[k*N+j+1];
      B[(i+1)*N + j+1] += A[(i+1)*N+k] * A[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 Akj0 = A[k*N + j+0];
      float Akj1 = A[k*N + j+1];
      B[(i+0)*N + j+0] += Ai0k * Akj0;
      B[(i+1)*N + j+0] += Ai1k * Akj0;
      B[(i+0)*N + j+1] += Ai0k * Akj1;
      B[(i+1)*N + j+1] += Ai1k * Akj1;
    }
  }
}
```

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) {
    ...
}
```

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)

```
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 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

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 (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)

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

```c
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

1.01 cycles/element — latency bound