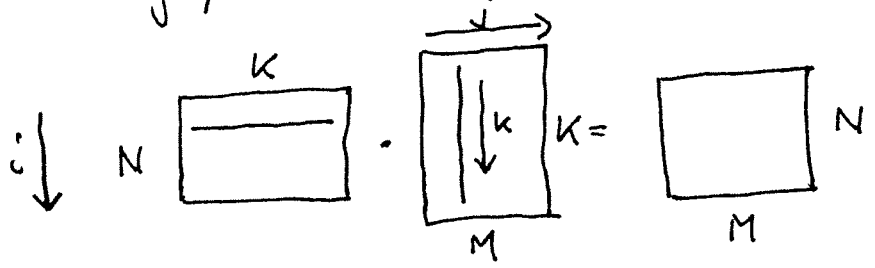
MMM generation with ATLAS

Starting point: standard triple loop



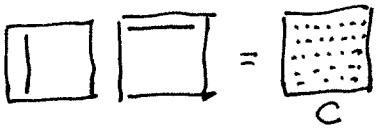

```
for i = 0:1:N-1                                   C = C + AB
  for j = 0:1:M-1
    for k = 0:1:K-1
      c_ij = c_ij + a_ik b_kj
```

Important cases (from most to least):
- ~~two~~ two out of (N, K, M) are small
- one out of (N, K, M) is small
- none out of (N, K, M) is small

Reason: based on how it is used inside LAPACK

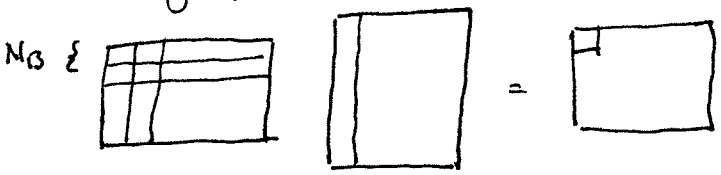
1.) loop order: i, j, k loops can be permuted into any order
- ijk: B is reused, good if $M < N$
- jik: A " , " $N < M$

ATCAS generates versions for both
- other choices are bad, e.g., kij:



poor temporal locality w.r.t. C

2.) blocking for cache



we assume for simplicity $N_B \mid N, M, K$

```
for i = 0:N_B:N-1
  for j = 0:N_B:M-1
    for k = 0:N_B:K-1
      for i' = i:1:i+N_B-1                 <- mini-MMM
        for j' = j:1:j+N_B-1
          for k' = k:1:k+N_B-1
            c_i'j' = c_i'j' + a_i'k' b_k'j'
```

This blocking is done using loop tiling + loop exchange

loop tiling (example):

```
for i = 0 : 1 : N-1
    a_i = ....
```

$\Longrightarrow$

```
for i = 0 : 4 : N-1
    for j = i : 1 : i+3
        a_j = ....
```

N_B becomes a search parameter.

Bound $N_B^2 \leq C$ (cache size)

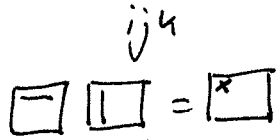
this is a loose bound — the search takes care of the rest.

For which cache should one block?

- The original ATLAS blocked for L1-cache
  - L2 was off-die
  - the important cases (see above) of MMM have one or two of N, K, M small
- For large, square matrices blocking for L2 can make sense
- Blocking for L1 & L2 can make sense
- Versions with different block sizes can make sense

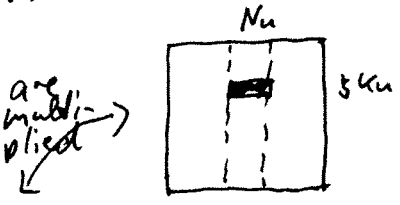
3.) Blocking Mini-MMM for registers

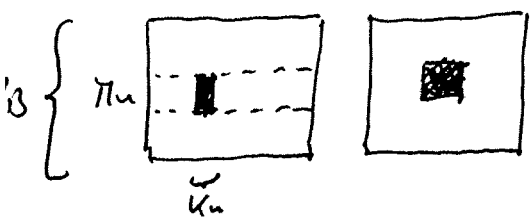
Now: k as outermost for ILP:

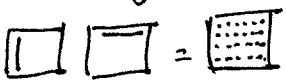


$2u$ instructions:
- u independent mults
- u dependent adds ($\geq \log_2(u)$ steps)

But: $2u+1$ "live" variables







$2u^2$ instructions:
- u^2 indep. mults
- u^2 " adds

$u^2 + 2u$ "live" variables

More "register pressure" since larger working set (always when more ILP)

```
for i = 0 : N_B : N-1
  for j = 0 : N_B : M-1
    for k = 0 : N_B : K-1
      for i' = i : M_u : i+N_B-1                          } mini-MMM
        for j' = j : N_u : j+N_B-1
          for k' = k : K_u : k+N_B-1
            for k'' = k' : 1 : k'+K_u-1                   } micro-MMM
              for i'' = i' : 1 : i'+M_u-1                 } ⊗
                for j'' = j' : 1 : j'+N_u-1
                  c_i''j'' = c_i''j'' + a_i''k'' b_k''j''
```

M_u, N_u, K_u become search parameters, we assume all $/ N_B$

Bound: $\underbrace{M_u + N_u + M_u N_u}_{\text{live variables in } \otimes} \leq N_R$ (no. of registers)

K_u: later

4.) Basic block optimizations

step 1: unroll ⊗

$c_{...} = c_{..} + a_{..} b_{...}$
$c_{...} = c_{..} + a_{..} b_{...}$
⋮
} $M_u N_u$ many

a..., b... reused
c... not

step 2: scalar replacement
Typically we would only do it on a..., b... (since reused) but since we later unroll the k''-loop, c.. will be reused as well.

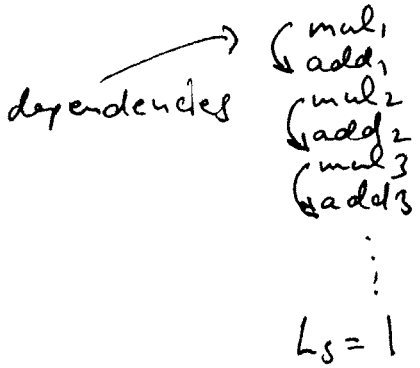
$t_0 = c_{...}$
$t_1 = a_{...}$
$t_2 = b_{...}$
⋮
} loads $(M_u + N_u + M_u N_u)$ many → all fit into register

$t_3 = t_0 + t_1 t_2$
⋮
} computation: $M_u N_u$ adds, $M_u N_u$ mults

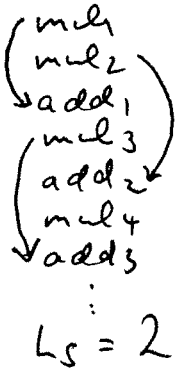
$c_{..} = t_3$
⋮
} stores $M_u N_u$ many

step 3: optimize computation part by "skewing"

computation as is:



skewed: ($L_S = 2$)



skewed by L_S:
dependent mul → add separated by $2L_S - 1$ instructions (except for beginning)

L_S is search parameter

Bound: $M_u + N_u + M_u N_u + L_S \leq N_R$

Also: Not all loads are done in the beginning. Only block of I_F loads and later blocks of N_F loads as needed.

step 4: unroll K^{th} loop

K_u is limited by I-cache size

Bound: $K_u \leq \frac{N_I}{2}$

Example: $K_u = 3$

load
comp
store
load
comp
store
load
comp
store

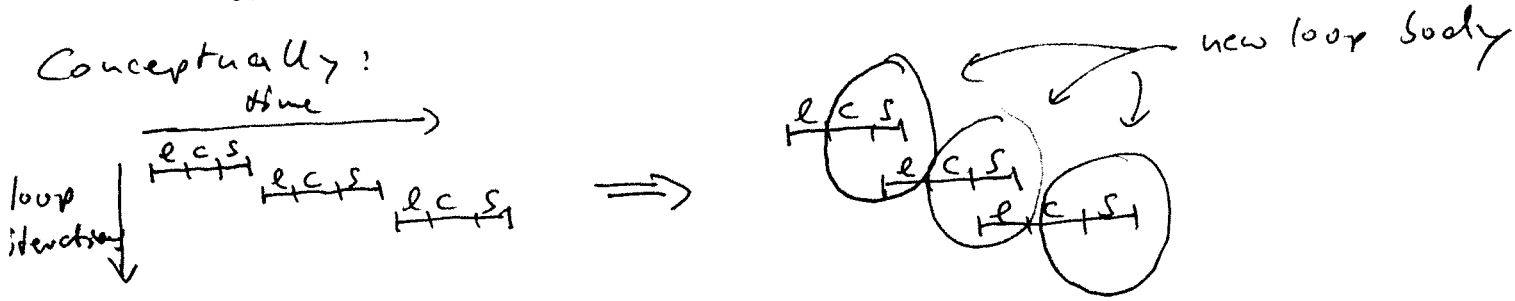
all those write to the same location (c...)
So now we have reuse on c...
⇒ get rid of first 2 stores

Result:

load
comp
load
comp
load
comp
store

step 5: (Software pipelining)
Reorganize k'-loop so loads of one iteration are moved to previous iteration

Conceptually:



l = load
c = compute
s = store

5.) Buffering to avoid TCB misses

Comes a little later