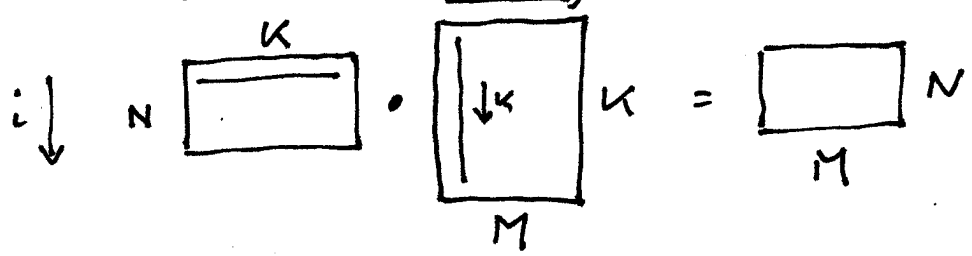
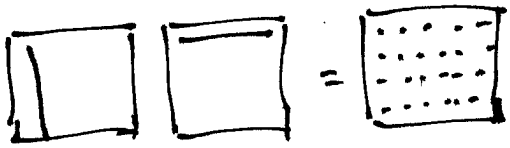
MMM Generation with ATLAS

Starting point: standard triple loop




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

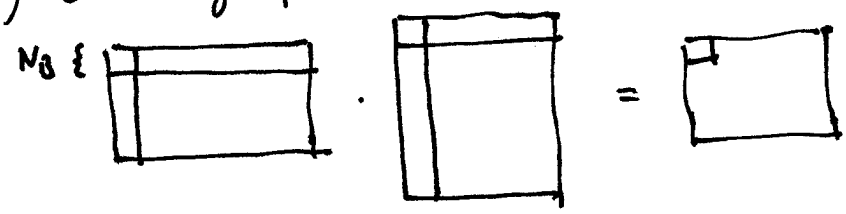
1.) loop order
- i,j,k can be permuted into any order
- i j k : B is reused, good if B is smaller than A : M < N
- j i k : A is reused B : N < M
- ATLAS searches versions for both
- other choices are bad, e.g., k i j



col. row
⇒ no output locality

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

2.) Blocking for cache (assume one cache)



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
        for j' = j:1:j+N_B-1
          for k' = k:1:k+N_B-1
```

} mini-MMM

- formally done using loop tiling and loop exchange

loop tiling: for $i = 0 : 1 : N-1$

$a_i \dots$

$\rightarrow$ for $i = 0 : 4 : N-1$

for $i' = i : 1 : i+3$

$a_{i'} \dots$

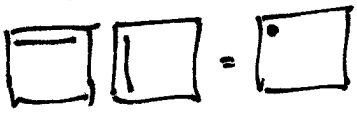
- (N_B) becomes a search parameter in ATLAS

trivial bound: $N_B^2 \leq C$ (cache size)

3.) blocking for registers

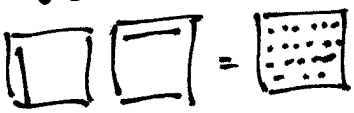
Now: k as outermost loop for instruction parallelism

ijk:



$2n$ instructions:
- n parallel mults
- n dependent adds (at least $\log_2(n)$ steps)

kij:

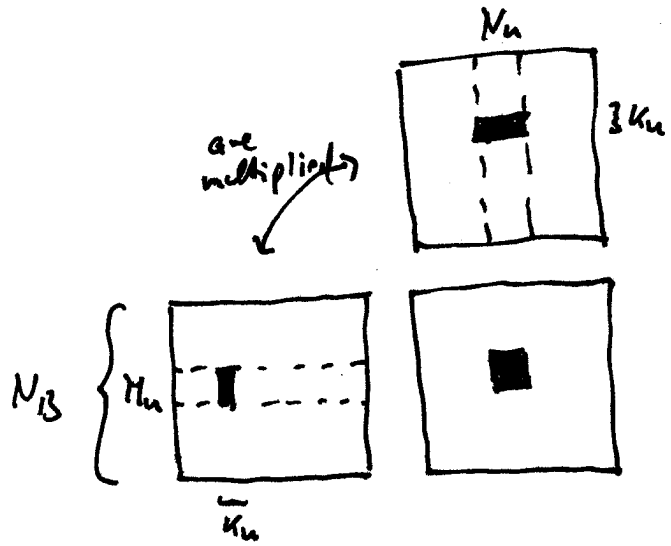


$2n^2$ instructions:
- n^2 parallel mults
- n^2 parallel adds

bad: $2n+1$ live variables

n^2+2n live variables

"register pressure" since larger working set



Code:

```
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
        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'']
```

possibly suffix a tile of C (see 5. below)

- (m_u, n_u, k_u) become search parameters
  bound $\underbrace{m_u n_u + m_u + n_u}_{\text{live variables}} \leq N_R$ (number of registers)

4.) Basic block optimizations

step 1: unroll ⊛ and do scalar replacement

unroll: many of those a, b, c are arrays

$$\begin{cases} c_{ij} = c_{ij} + a_{ik} b_{kj} \\ \vdots \\ \vdots \end{cases}$$

scalar replacement: replace array elements with scalar variables

```
t0 = c_ij
t1 = a_ik
t2 = b_kj
 :                    } loads
t0 = t0 + t1 * t2     } computation
 :
 :
c_ij = ....           } stores
 :
 :
```

- enables register allocation by compiler and other optimizations (for other numerical problems)

step 2: reorder load, adds, mults, stores to take dependencies and latencies into account

example: $c_{ij} = c_{ij} + a_{ik} b_{kj}$

$\Leftrightarrow$ $t = a_{ik} b_{kj}$
$c_{ij} = c_{ij} + t$
} dependent, problem if mult latency > add latency

solution: "skewing" with L_S

```
mul1
add1
mul2
add2
 :
```
standard: $L_S = 1$

```
mul1
mul2
add1
mul3
add2
mul4
add3
 :
```
$L_S = 2$

(L_S) is search parameter

other reordering: interleave loads and adds/mults

step 3: unroll "k" loop, k_u is limited by size of I-cache

- micro-MMM now becomes:
  | load comp store | load comp store | ... | ... | ($k_u = 4$)
- software pipelining: move loads from one iteration of k' loop to previous iteration

5.) Buffering:
Idea: copy working set that is reused in a way that may cause thrashing into contiguous memory

Example: