HMM Generation with ATLAS

Starting point: standard triple loop

\[
\begin{array}{c}
\downarrow \quad N \quad \downarrow \quad K \quad \downarrow \quad M
\end{array}
\]

\[K = \begin{array}{c}
\downarrow \quad N \quad \downarrow \quad M
\end{array}\]

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
- \(ijk\) can be permuted into any order
- \(ijk\): \(A\) is reused, good if \(B\) is smaller \(N < A\)
- \(ijk\): \(A\) is reused
- ATLAS generates versions for code
- other choices are dead, e.g., \(kij\)

\[
\begin{array}{c}
\downarrow \quad \downarrow \\
\end{array}
\]
\[
\begin{array}{c}
\downarrow \quad \downarrow
\end{array}
\]

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)

\[
\begin{array}{c}
\downarrow \quad \downarrow \\
\end{array}
\]
\[
\begin{array}{c}
\downarrow \quad \downarrow
\end{array}
\]

for \(i = 0:1:N_a - 1\)
for \(j = 0:1:M_a - 1\)
for \(k = 0:1:K_a - 1\)
for \(i' = i + 1: i + N_a - 1\)
for \(j' = j + 1: j + M_a - 1\)
for \(k' = k + 1: k + N_a - 1\)

\[
\min_{i,j} \left\{ \sum_{k=1}^{K_a} c_{ij} + a_{ik} b_{kj} \right\}
\]
- formally done using loop tiling and loop exchange

  loop tiling: \( \text{for } i = 0 : 1 : N-1 \)
  \( q_i \ldots \)

  \(-\) \( \text{for } i = 0 : 4 : N-1 \)

  \( q_i \ldots \)

  \( q_i \)

- \( N_3 \) becomes a search parameter in ATLAS

  trivial bound: \( N_3^2 \leq C \) (each side)

\[ \text{3.2) blocking for registers} \]

Now: \( k \) as outermost loop for instruction parallelism

- \( ijk \):
  \[\begin{array}{ccc}
  & & \\
  \hline
  & & \\
  \hline
  & & \\
  \end{array}\]

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

  End: \( 2n+1 \) line variables

- \( kij \):
  \[\begin{array}{ccc}
  & & \\
  \hline
  & & \\
  \hline
  & & \\
  \end{array}\]

  2n^2 instructions:
  - \( n^2 \) parallel adds
  - \( n^2 \) parallel adds

  \( n^2 + 2n \) line variables

  "register pressure" since larger working set

\[\begin{array}{c}
N_3 \\
\hline
N_3 \\
\hline
\end{array}\]

\[\begin{array}{c}
N_k \\
\hline
1N_k \\
\hline
\end{array}\]
Code:
for i = 0 : N_l : N_l - 1
  for j = 0 : N_k : N_k - 1
    for k = 0 : K : K - 1
      for i' = i : N_l : i' + N_l - 1
        for j' = j : N_k : j' + N_k - 1
          for k' = k : K : k' + K - 1
            for i'' = i' : i' + N_l - 1
              for j'' = j' : j' + N_k - 1
                  \( C_{i,j} = C_{i'',j''} + a_{i''} u_{k'} b_{k''} \)

- \((N_l, N_k, K)\) become search parameter
- bound \( N_l N_k + N_k + N_l \leq N_k \) (number of registers)

4.1) Basic block optimization
   step 1: unroll \( \otimes \) and do scalar replacement

   unroll:
   \( C_{i,j} = C_{i,j} + a_{i''} b_{k''} \)
   many of those
   \( a, b, c \) are arrays

   scalar replacement: replace array elements with scalar variables

\( t_0 = C_{i,j} \)
\( t_1 = a_{i''} \)
\( t_2 = b_{k''} \)
\( t_0 = t_0 + t_1 t_2 \)
\( \vdots \)
\( C_{i,j} = \quad \)
- enables register allocation, IA32 compiler
  and other optimizations (for other numerical problems)

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

example: \[ C_{ij} = C_{ij} + \text{add}_{byj} \]

\[ \therefore \quad v = \text{add}_{byj} \quad \} \quad \text{dependent, problem if} \]
\[ \text{mul} \quad \text{latency} > \text{fixed latency} \]

solution: "skewing" with \( L_s \)
{\begin{align*}
\text{mul}_1, \\
\text{add}_1, \\
\text{mul}_2, \\
\text{add}_2, \\
\vdots \\
\end{align*}}

\text{standard: } L_s = 1 \\
\text{\( L_s = 2 \)}

\( L_s \) is skew parameter
other reordering: interleave loads and adds/multiplies

step 3: unroll \( k \) loop, \( k \) is limited by size of \( I_{cude} \)

- micro-OPM now becomes:

\[
\text{load core store | load core store | \ldots | \ldots | (k=4)}
\]

- software pipelining: more 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:

\[
\begin{bmatrix}
N_0 \\
\vdots \\
\end{bmatrix}
\quad =
\begin{bmatrix}
\vdots \\
\end{bmatrix}
\]

\text{possibly}

\text{buffer}