Blocked MMM
and other Optimizations
Andrew Lenharth

MMM – The problem

- Move Right?
  - No reuse in B
- Move down?
  - No reuse in A
- Too big?
  - No reuse

Each C requires N/B MMM
Each one is B^3
There are N/B * N/B Cs
N^2/B^2 * N/B * B^3 = N^3
As in blocked MVM, we actually need to stripmine only two loops

\[
\text{For } \mathbf{k} := \mathbf{i} \text{ to } \mathbf{N} \text{ by } \mathbf{B} \text{ do }
\]
\[
\text{for } \mathbf{j} := \mathbf{i} \text{ to } \mathbf{N} \text{ by } \mathbf{B} \text{ do }
\]
\[
\text{for } \mathbf{i} := \mathbf{I} \text{ to } \mathbf{N} \text{ do }
\]
\[
\text{for } \mathbf{k} := \mathbf{k} \text{ to } \min(\mathbf{k} + \mathbf{B} - 1, \mathbf{N}) \text{ do }
\]
\[
r = A[\mathbf{i}, \mathbf{k}] \text{ /* register allocated */ }
\]
\[
\text{for } \mathbf{j} := \mathbf{j} \text{ to } \min(\mathbf{j} + \mathbf{B} - 1, \mathbf{N}) \text{ do }
\]
\[
C[\mathbf{i}, \mathbf{j}] += r \times B[\mathbf{k}];
\]
Strassen’s Matrix Multiply

- The traditional algorithm (with or without tiling) has $O(n^3)$ flops
- Strassen discovered an algorithm with asymptotically lower flops
  $= O(n^{2.81})$

Consider a 2x2 matrix multiply, normally takes 8 multiplies, 7 adds
- Strassen does it with 7 multiplies and 18 adds

$p_1 = (a_{11} + a_{12}) \times (b_{11} + b_{22})$
$p_2 = (a_{11} + a_{22}) \times (b_{11} + b_{22})$
$p_3 = a_{11} \times b_{12} + a_{12} \times b_{22}$
$p_4 = a_{21} \times b_{11} + a_{22} \times b_{21}$
$p_5 = a_{11} \times b_{22} - (p_1 + p_2)$
$p_6 = a_{22} \times b_{11} - (p_1 + p_2)$

$p_7 = (a_{11} - a_{12}) \times (b_{21} - b_{22})$
$p_8 = (a_{21} - a_{22}) \times (b_{11} - b_{12})$
$p_9 = a_{11} \times b_{11} + a_{12} \times b_{21}$

If $n \leq 2$, return $C = A \times B$

Recursive Matrix Multiplication

(RMM) (1/2)

- For simplicity: square matrices with $n = 2^m$

\[
C = RMM(A_{11}, B_{11}, n/2) + RMM(A_{12}, B_{12}, n/2) + RMM(A_{21}, B_{12}, n/2) + RMM(A_{22}, B_{22}, n/2)
\]

True when each $A_i$ etc 1x1 or $n/2 \times n/2$

Recursive: Cache Oblivious Algorithms

- The tiled algorithm requires finding a good block size
  - Machine dependent
  - What if there are multiple levels of cache? Need to “block” b x b
    matrix multiply in inner most loop
    - 1 level of memory = 3 nested loops (naive algorithm)
    - 2 levels of memory = 6 nested loops
    - 3 levels of memory = 9 nested loops ...

- Cache Oblivious Algorithms offer an alternative
  - Treat nn matrix multiply as a set of smaller problems
  - Eventually, these will fit in cache
  - Will minimize # words moved between every level of memory
    hierarchy (between L1 and L2 cache, L2 and L3, L3 and main
    memory etc.) – at least asymptotically

Recursive Matrix
Multiplication (2/2)

func $C = RMM(A, B, n)$

if $n=1$, $C = A \times B$, else

\[
C_{11} = RMM(A_{11}, B_{11}, n/2) + RMM(A_{12}, B_{21}, n/2)
\]
\[
C_{12} = RMM(A_{11}, B_{12}, n/2) + RMM(A_{12}, B_{22}, n/2)
\]
\[
C_{21} = RMM(A_{21}, B_{11}, n/2) + RMM(A_{22}, B_{21}, n/2)
\]
\[
C_{22} = RMM(A_{21}, B_{12}, n/2) + RMM(A_{22}, B_{22}, n/2)
\]

return

$A(n) = \#$ arithmetic operations in RMM(\ldots, n)
  $= 8 \cdot A(n/2) + 4(n/2)^2 \text{ if } n > 1$, else 1
  $= 2n^2 \ldots$ same operations as usual, in different order

$M(n) = \#$ words moved between fast, slow memory by RMM(\ldots, n)
  $= 8 \cdot M(n/2) + 4(n/2)^2 \text{ if } n > 32 > M_{\text{fast}}, \text{ else } 3n^2$
  $= O(n^2 / (M_{\text{fast}})^{1/2} + n^2)$ \ldots same as blocked matmul

Strassen (continued)

$T(n) = \text{Cost of multiplying nxn matrices}$

$= 7T(n/2) + 18(n/2)^2$

$= O(n \log_7 7)$

$= O(n 2.81)$

- Asymptotically faster
  - Several times faster for large n in practice
  - Cross-over depends on machine
  - Available in several libraries

- Caveats
  - Needs more memory than standard algorithm
  - Can be less accurate because of roundoff error
  - Current world’s record is $O(n 2.307..)$
  - Why does Hong/Kung theorem not apply?

Recursive: Cache Oblivious
Algorithms

- Recursion for general A (nxm) * B (mxp)

\[
\begin{bmatrix}
A_1 \\
A_2
\end{bmatrix} = \begin{bmatrix}
A_1 & A_2
\end{bmatrix} \begin{bmatrix}
B_1 \\
B_2
\end{bmatrix} = (A_1 B_1 + A_2 B_2)
\]

Case 1

\[
A R_1 + B R_2 = A R_1 + B R_2
\]

Case 2

\[
\begin{bmatrix}
A_1, A_2
\end{bmatrix} \begin{bmatrix}
B_1 \\
B_2
\end{bmatrix} = (A_1 B_1 + A_2 B_2)
\]

Case 3

\[
A (R_1, R_2) = \begin{bmatrix}
A R_1, A R_2
\end{bmatrix}
\]

- Attains lower bound in $O()$ sense

\[
\begin{bmatrix}
A_1, A_2
\end{bmatrix} \begin{bmatrix}
B_1 \\
B_2
\end{bmatrix} = (A_1 B_1 + A_2 B_2)
\]

Case 2
Experience with Cache-Oblivious Algorithms

• In practice, need to cut off recursion well before 1x1 blocks
  – Call “Micro-kernel” for small blocks, eg 16 x 16
  – Implementing a high-performance Cache-Oblivious code is not easy

• Using fully recursive approach with highly optimized recursive micro-kernel, Pingali et al report that they never got more than 2/3 of peak.

• Issues with Cache Oblivious (recursive) approach
  – Recursive Micro-Kernels yield less performance than iterative ones using same scheduling techniques
  – Pre-fetching is needed to compete with best code: not well-understood in the context of Cache Oblivious codes

Unpublished work, presented at LACSI 2006