A FRAMEWORK FOR PRACTICAL FAST MATRIX MULTIPLICATION

arXiv: 1409.2908

Parallel performance of Strassen on $<N,N,N>$

Austin Benson (arbenson@stanford.edu), ICME, Stanford
Grey Ballard, Sandia National Laboratories
BLIS Retreat, September 26, 2014
There are a number of Strassen-like algorithms for matrix multiplication that have only been “discovered” recently. [Smirnov13], [Benson&Ballard14]

We show that they can achieve higher performance with respect to MKL (sequential and sometimes in parallel).

We use code generation to do extensive prototyping. There are several practical issues, and there is plenty of room for improvement (lots of expertise at UT to help here!)
Strassen’s algorithm

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

\[
S_1 = A_{11} + A_{22} \\
S_2 = A_{21} + A_{22} \\
S_3 = A_{11} \\
S_4 = A_{22} \\
S_5 = A_{11} + A_{12} \\
S_6 = A_{21} - A_{11} \\
S_7 = A_{12} - A_{22}
\]

\[
T_1 = B_{11} + B_{22} \\
T_2 = B_{11} \\
T_3 = B_{12} - B_{22} \\
T_4 = B_{21} - B_{11} \\
T_5 = B_{22} \\
T_6 = B_{11} + B_{12} \\
T_7 = B_{21} + B_{22}
\]

\[
M_r = S_r \cdot T_r \\
1 \leq r \leq 7
\]

\[
C_{11} = M_1 + M_4 - M_5 + M_7 \\
C_{12} = M_3 + M_5 \\
C_{21} = M_2 + M_4 \\
C_{22} = M_1 - M_2 + M_3 + M_6
\]
Key ingredients of Strassen’s algorithm

• 1. Block partitioning of matrices (<2, 2, 2>)
• 2. **Seven** linear combinations of sub-blocks of A
• 3. **Seven** linear combinations of sub-blocks of B
• 4. **Seven** matrix multiplies to form $M_r$ (recursive)
• 5. Linear combinations of $M_r$ to form $C_{ij}$
Key ingredients of fast matmul algorithms

1. Block partitioning of matrices ($<M, K, N>$)
2. $R$ linear combinations of sub-blocks of $A$
3. $R$ linear combinations of sub-blocks of $B$
4. $R$ matrix multiplies to form $M_r$ (recursive)
   $R < MKN \rightarrow$ faster than classical
5. Linear combinations of $M_r$ to form $C_{ij}$
“Outer product” fast algorithm

\[
\begin{bmatrix}
C_{11} & C_{12} & C_{13} & C_{14}
\end{bmatrix}
\begin{bmatrix}
C_{21} & C_{22} & C_{23} & C_{24}
\end{bmatrix}
\begin{bmatrix}
C_{31} & C_{32} & C_{33} & C_{34}
\end{bmatrix}
\begin{bmatrix}
C_{41} & C_{42} & C_{43} & C_{44}
\end{bmatrix}
= 
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22} \\
A_{31} & A_{32} \\
A_{41} & A_{42}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} & B_{13} & B_{14} \\
B_{21} & B_{22} & B_{23} & B_{24}
\end{bmatrix}
\]

- \text{<4, 2, 4> partitioning}
- \text{R = 26 multiplies (}< 4 * 2 * 4 = 32\text{)}
  \rightarrow 23\% \text{ speedup per recursive step (if everything else free)}
- Linear combinations of \(A_{ij}\) to form \(S_r\): 68 terms
- Linear combinations of \(B_{ij}\) to form \(T_r\): 52 terms
- Linear combinations of \(M_r\) to form \(C_{ij}\): 69 terms
Discovering fast algorithms is a numerical challenge

- Low-rank tensor decompositions lead to fast algorithms
- Tensors are small, but we need exact decompositions \( \rightarrow \) NP-hard
- Use alternating least squares with regularization and rounding tricks [Smirnov13], [Benson&Ballard14]
- We have around 10 fast algorithms for \(<M, K, N>\) decompositions. Also have permutations, e.g., \(<K, M, N>\).
<table>
<thead>
<tr>
<th>Algorithm base case</th>
<th>Multiples (fast)</th>
<th>Multiples (classical)</th>
<th>speedup per recursive step</th>
<th>exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨2, 2, 3⟩</td>
<td>11</td>
<td>12</td>
<td>9%</td>
<td>2.89</td>
</tr>
<tr>
<td>⟨2, 2, 5⟩</td>
<td>18</td>
<td>20</td>
<td>11%</td>
<td>2.89</td>
</tr>
<tr>
<td>⟨2, 2, 2⟩[Strassen69]</td>
<td>7</td>
<td>8</td>
<td>14%</td>
<td>2.81</td>
</tr>
<tr>
<td>⟨2, 2, 4⟩</td>
<td>14</td>
<td>16</td>
<td>14%</td>
<td>2.85</td>
</tr>
<tr>
<td>⟨3, 3, 3⟩</td>
<td>23</td>
<td>26</td>
<td>17%</td>
<td>2.85</td>
</tr>
<tr>
<td>⟨2, 3, 3⟩</td>
<td>15</td>
<td>18</td>
<td>20%</td>
<td>2.81</td>
</tr>
<tr>
<td>⟨2, 3, 4⟩</td>
<td>20</td>
<td>24</td>
<td>20%</td>
<td>2.83</td>
</tr>
<tr>
<td>⟨2, 4, 4⟩</td>
<td>26</td>
<td>32</td>
<td>23%</td>
<td>2.82</td>
</tr>
<tr>
<td>⟨3, 3, 4⟩</td>
<td>29</td>
<td>36</td>
<td>24%</td>
<td>2.82</td>
</tr>
<tr>
<td>⟨3, 4, 4⟩</td>
<td>38</td>
<td>48</td>
<td>26%</td>
<td>2.82</td>
</tr>
<tr>
<td>⟨3, 3, 6⟩[Smirnov13]</td>
<td>40</td>
<td>54</td>
<td>35%</td>
<td>2.77</td>
</tr>
</tbody>
</table>
Code generation lets us prototype algorithms quickly

- We have compact representation of many fast algorithms:
  1. dimensions of block partitioning ($<M, K, N>$)
  2. linear combinations of sub-blocks ($S_r, T_r$)
  3. linear combinations of $M_r$ to form $C_{ij}$

- We use code generation to rapidly prototype fast algorithms

- Our approach: test all algorithms on a bunch of different problem sizes and look for patterns
Practical issues

• Best way to do matrix additions? (in paper)
• Can we eliminate redundant linear combinations? (in paper)
• Different problem shapes other than square (this talk)
• When to stop recursion? (this talk)
• How to parallelize? (this talk)
Recursion cutoff: look at gemm curve

Sequential dgemm performance

Parallel dgemm performance (24 cores)

Basic idea: take another recursive step if the sub-problems will still operate at high performance

\(<M, K, N> = <4, 2, 3>\)
Sequential performance

Effective GFLOPS for $M \times K \times N$ multiplies

$= 1e^{-9} \times 2 \times MKN / \text{time in seconds}$
Sequential performance

- All algorithms beat MKL on large problems
- Strassen’s algorithm is hard to beat
Sequential performance

Sequential performance on N x 1600 x N

- Almost all algorithms beat MKL
- <4, 2, 4> and <3, 2, 3> tend to perform the best
Sequential performance

- Almost all algorithms beat MKL
- <4, 3, 3> and <4, 2, 3> tend to perform the best
Parallelization

\[ S_7 = A_{12} - A_{22} \]
\[ T_7 = B_{21} + B_{22} \]
\[ M_7 = S_7 \cdot T_7 \]
DFS Parallelization

\[
S_7 = A_{12} - A_{22} \\
T_7 = B_{21} + B_{22} \\
M_7 = S_7 \cdot T_7
\]

All threads
Use parallel MKL

+ Easy to implement
+ Load balanced
+ Same memory footprint as sequential
- Need large base cases for high performance
BFS Parallelization

\[
S_7 = A_{12} - A_{22} \\
T_7 = B_{21} + B_{22} \\
M_7 = S_7 \cdot T_7
\]

- High performance for smaller base cases
- Sometimes harder to load balance: 24 threads, 49 subproblems
- More memory
HYBRID parallelization

\[ S_7 = A_{12} - A_{22} \]
\[ T_7 = B_{21} + B_{22} \]
\[ M_7 = S_7 \cdot T_7 \]

+ Better load balancing
- Explicit synchronization or else we can over-subscribe threads
Parallel performance of <4,2,4> on <N,2800,N>

- MKL, 6 cores
- MKL, 24 cores
- DFS, 6 cores
- DFS, 24 cores
- BFS, 6 cores
- BFS, 24 cores
- HYBRID, 6 cores
- HYBRID, 24 cores
Bandwidth problems

- We rely on the cost of matrix multiplications to be much more expensive than the cost of matrix additions
- Parallel dgemm on 24 cores: easily get 50-75% of peak
- STREAM benchmark: < 6x speedup in read/write performance on 24 cores

\[
\begin{align*}
S_7 &= A_{12} - A_{22} \\
T_7 &= B_{21} + B_{22} \\
M_7 &= S_7 \cdot T_7
\end{align*}
\]
Parallel performance

- 6 cores: similar performance to sequential
- 24 cores: can sometimes beat MKL, but barely
Parallel performance

- 6 cores: similar performance to sequential
- 24 cores: MKL best for large problems
Parallel performance

- 6 cores: similar performance to sequential
- 24 cores: MKL usually the best
High-level conclusions

• For square matrix multiplication, Strassen’s algorithm is hard to beat
• For rectangular matrix multiplication, use a fast algorithm that “matches the shape”
• Bandwidth limits the performance of shared memory parallel fast matrix multiplication → should be less of an issue in distributed memory

Future work:
• Numerical stability
• Using fast matmul as a kernel for other algorithms in numerical linear algebra
A FRAMEWORK FOR PRACTICAL FAST MATRIX MULTIPLICATION

Parallel performance of Strassen on $<N,N,N>$

Effective GFLOPS / core

- MKL, 6 cores
- MKL, 24 cores
- DFS, 6 cores
- BFS, 6 cores
- HYBRID, 6 cores
- DFS, 24 cores
- BFS, 24 cores
- HYBRID, 24 cores

Austin Benson (arbenson@stanford.edu), ICME, Stanford
Grey Ballard, Sandia National Laboratories
BLIS Retreat, September 26, 2014
Matrix additions (linear combinations)

\[ S_1 = A_{11} + A_{22} \]
\[ S_2 = A_{21} + A_{22} \]
\[ S_3 = A_{11} \]
\[ S_4 = A_{22} \]
\[ S_5 = A_{11} + A_{12} \]
\[ S_6 = A_{21} - A_{11} \]
\[ S_7 = A_{12} - A_{22} \]

“Pairwise”
Matrix additions (linear combinations)

\[ S_1 = A_{11} + A_{22} \]
\[ S_2 = A_{21} + A_{22} \]
\[ S_3 = A_{11} \]
\[ S_4 = A_{22} \]
\[ S_5 = A_{11} + A_{12} \]
\[ S_6 = A_{21} - A_{11} \]
\[ S_7 = A_{12} - A_{22} \]
Matrix additions (linear combinations)

\[
\begin{align*}
S_1 &= A_{11} + A_{22} \\
S_2 &= A_{21} + A_{22} \\
S_3 &= A_{11} \\
S_4 &= A_{22} \\
S_5 &= A_{11} + A_{12} \\
S_6 &= A_{21} - A_{11} \\
S_7 &= A_{12} - A_{22}
\end{align*}
\]

“Streaming”

Entry-wise updates
Common subexpression elimination (CSE)

- Example in <4, 2, 4> algorithm (R = 26 multiples):

\[ T_{11} = B_{24} - (B_{12} + B_{22}) \]
\[ T_{25} = B_{23} + B_{12} + B_{22} \]

Four additions, six reads, two writes
Common subexpression elimination (CSE)

- Example in <4, 2, 4> algorithm (R = 26 multiples):

\[
Y = B_{12} + B_{22} \\
T_{11} = B_{24} - Y \\
T_{25} = B_{23} + Y
\]

Three additions, six reads, three writes
→ Net increase in communication!
CSE does not really help

Effective GFLOPS for M x K x N multiplies

\[ = 1e-9 * 2 * MKN / \text{time in seconds} \]