How Practical is Fast Matrix Multiplication?

Grey Ballard

based on joint work with Austin Benson

December 10, 2014
Outline

1. Communication Costs
2. Strassen’s Matrix Multiplication: Theory & Practice
3. Searching for Fast Matrix Multiplication
4. Practical Performance of Fast Matrix Multiplication
Strassen’s algorithm (1969)

Strassen showed how to use 7 scalar multiplies for $2 \times 2$ matrix multiplication

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

**Strassen’s Algorithm**

\[
\begin{align*}
M_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22}) \cdot B_{11} \\
M_3 &= A_{11} \cdot (B_{12} - B_{22}) \\
M_4 &= A_{22} \cdot (B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12}) \cdot B_{22} \\
M_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22})
\end{align*}
\]

\[
\begin{align*}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{align*}
\]
Strassen’s algorithm (1969)

Strassen showed how to use 7 scalar multiplies for $2 \times 2$ matrix multiplication

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix} =
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

Classical Algorithm

\[
\begin{align*}
M_1 &= A_{11} \cdot B_{11} \\
M_2 &= A_{12} \cdot B_{21} \\
M_3 &= A_{11} \cdot B_{12} \\
M_4 &= A_{12} \cdot B_{22} \\
M_5 &= A_{21} \cdot B_{11} \\
M_6 &= A_{22} \cdot B_{21} \\
M_7 &= A_{21} \cdot B_{12} \\
M_8 &= A_{22} \cdot B_{22}
\end{align*}
\]

\[
\begin{align*}
C_{11} &= M_1 + M_2 \\
C_{12} &= M_3 + M_4 \\
C_{21} &= M_5 + M_6 \\
C_{22} &= M_7 + M_8
\end{align*}
\]

Strassen’s Algorithm

\[
\begin{align*}
M_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22}) \cdot B_{11} \\
M_3 &= A_{11} \cdot (B_{12} - B_{22}) \\
M_4 &= A_{22} \cdot (B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12}) \cdot B_{22} \\
M_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22})
\end{align*}
\]

\[
\begin{align*}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{align*}
\]
Strassen’s algorithm (1969)

Strassen showed how to use 7 scalar multiplies for $2 \times 2$ matrix multiplication

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

**Strassen’s Algorithm**

For $n \times n$ matrices, we split into quadrants and use recursion

**Flop count recurrence:**

$$
F(n) = 7 \cdot F(n/2) + O(n^2)
$$

$$
F(1) = 1
$$

$$
F(n) = O \left( n^{\log_2 7} \right)
$$

$$
\log_2 7 \approx 2.81
$$

$$
M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})
$$

$$
M_2 = (A_{21} + A_{22}) \cdot B_{11}
$$

$$
M_3 = A_{11} \cdot (B_{12} - B_{22})
$$

$$
M_4 = A_{22} \cdot (B_{21} - B_{11})
$$

$$
M_5 = (A_{11} + A_{12}) \cdot B_{22}
$$

$$
M_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})
$$

$$
M_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})
$$

$$
C_{11} = M_1 + M_4 - M_5 + M_7
$$

$$
C_{12} = M_3 + M_5
$$

$$
C_{21} = M_2 + M_4
$$

$$
C_{22} = M_1 - M_2 + M_3 + M_6
$$
Strassen’s algorithm (1969)

Strassen showed how to use 7 scalar multiplies for $2 \times 2$ matrix multiplication

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

For $n \times n$ matrices, we split into quadrants and use recursion

Word count recurrence:

$$W(n) = 7 \cdot W(n/2) + O(n^2)$$

$$W(\sqrt{M}) = O(M)$$

$$W(n) = O\left(\left(\frac{n}{\sqrt{M}}\right)^{\log_2 7} M\right)$$

($M$ is the cache size in words)
Memory models for communication costs

Algorithms have two kinds of costs: computation and *communication*
- moving data within memory hierarchy on a sequential computer
- moving data between processors on a parallel computer

For high-level analysis, we use these memory models:
Strassen’s algorithm (1969)

Strassen showed how to use 7 scalar multiplies for $2 \times 2$ matrix multiplication

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

For $n \times n$ matrices, we split into quadrants and use recursion

Word count recurrence:

$W(n) = 7 \cdot W(n/2) + O(n^2)$

$W(\sqrt{M}) = O(M)$

$W(n) = O \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} M \right)$

$(M$ is the cache size in words)
Communication costs of matrix multiplication

Classical

Seq
\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right) \]

Par
\[ O \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} M \right) \]

Strassen’s

Units = (max) words communicated

O = algorithm exists, Ω = lower bound exists, Θ = both exist

n = matrix dimension, M = fast/local memory size, P = number of processors

References:

[BDHS11], [BDH+12a], [BDH+12b], [BDHS12, BDHS14]
Communication costs of matrix multiplication

**Classical**

\[
\Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right)
\]

**Strassen’s**

\[
\Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} M \right)
\]

Units = (max) words communicated

\(O =\) algorithm exists, \(\Omega =\) lower bound exists, \(\Theta =\) both exist

\(n =\) matrix dimension, \(M =\) fast/local memory size, \(P =\) number of processors

References: [BDHS11], [BDH+12a], [BDH+12b], [BDHS12, BDHS14]
Communication costs of matrix multiplication

Classical

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right) \]

Strassen’s

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} M \right) \]

Par

\[ \Omega \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} \frac{M}{P} \right) \]

Units = (max) words communicated

\[ O = \text{algorithm exists}, \Omega = \text{lower bound exists}, \Theta = \text{both exist} \]

\[ n = \text{matrix dimension}, M = \text{fast/local memory size}, P = \text{number of processors} \]

References: [BDHS11], [BDH12a], [BDH12b], [BDHS12], [BDHS14]
Communication costs of matrix multiplication

Classical

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right) \]

Strassen’s

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} M \right) \]

Units = (max) words communicated

\[ O = \text{algorithm exists, } \Omega = \text{lower bound exists, } \Theta = \text{both exist} \]

\[ n = \text{matrix dimension, } M = \text{fast/local memory size, } P = \text{number of processors} \]

References: [BDHS11], [BDH12a], [BDH12b], [BDHS12, BDHS14]
Communication costs of matrix multiplication

Classical

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right) \]

Strassen’s

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} M \right) \]

Seq

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right) \]

Par

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} \frac{M}{P} \right) \]

Units = (max) words communicated

\( O \) = algorithm exists, \( \Omega \) = lower bound exists, \( \Theta \) = both exist

\( n \) = matrix dimension, \( M \) = fast/local memory size, \( P \) = number of processors

References: [BDHS11], [BDH+12a],
## Communication costs of matrix multiplication

<table>
<thead>
<tr>
<th></th>
<th>Classical</th>
<th>Strassen’s</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Seq</strong></td>
<td>$\Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} \right)$</td>
<td>$\Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} \right)$</td>
</tr>
<tr>
<td><strong>Par</strong></td>
<td>$\Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} \frac{M}{P} \right)$</td>
<td>$\Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} \frac{M}{P} \right)$ + $\Omega \left( \frac{n^2}{P^2/\log_2 7} \right)$</td>
</tr>
</tbody>
</table>

Units = (max) words communicated

$O = \text{algorithm exists,} \ \Omega = \text{lower bound exists,} \ \Theta = \text{both exist}$

$n = \text{matrix dimension,} \ M = \text{fast/local memory size,} \ P = \text{number of processors}$

References: [BDHS11], [BDH+12a],
Communication costs of matrix multiplication

Classical          Strassen’s

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right) \quad \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} M \right) \]

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} \frac{M}{P} \right) \quad \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} \frac{M}{P} \right) \]

\[ + \Theta \left( \frac{n^2}{P^2 / \log_2 8} \right) \quad + \Theta \left( \frac{n^2}{P^2 / \log_2 7} \right) \]

Units = (max) words communicated

\( O = \) algorithm exists, \( \Omega = \) lower bound exists, \( \Theta = \) both exist

\( n = \) matrix dimension, \( M = \) fast/local memory size, \( P = \) number of processors

References: [BDHS11], [BDH^+12a],
Communication costs of matrix multiplication

Classical

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right) \]

Strassen’s

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} M \right) \]

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} \frac{M}{P} \right) + \Theta \left( \frac{n^2}{P^2 / \log_2 8} \right) \]

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} \frac{M}{P} \right) + \Theta \left( \frac{n^2}{P^2 / \log_2 7} \right) \]

Units = (max) words communicated

\[ O = \text{algorithm exists}, \quad \Omega = \text{lower bound exists}, \quad \Theta = \text{both exist} \]

\[ n = \text{matrix dimension}, \quad M = \text{fast/local memory size}, \quad P = \text{number of processors} \]

References: [BDHS11], [BDH+12a], [BDH+12b],
Performance of optimal algorithms on large problem

Strong-scaling on a Cray XT4, $n = 94,080$

![Graph showing performance comparison]

- **Old Strassen**
- **Optimal Strassen**
- **Optimal Classical**
- **ScaLAPACK (classical)**

Effective GFLOPS per node vs. $P$ (number of processors):
- Machine peak (for classical algorithms)

Values for $P = 49$, $P = 343$, and $P = 2401$.
Execution time of optimal algorithms on small problem

Strong-scaling on a Cray XE6, $n = 4704$
Communication costs of matrix multiplication

**Classical**

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right) \]

**Strassen’s**

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} M \right) \]

**Seq**

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right) \]

**Par**

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} \frac{M}{P} \right) \]

Units = (max) words communicated

O = algorithm exists, \( \Omega \) = lower bound exists, \( \Theta \) = both exist

\( n \) = matrix dimension, \( M \) = fast/local memory size, \( P \) = number of processors

References: [BDHS11], [BDH^{+}12a], [BDH^{+}12b],
Communication costs of matrix multiplication

**Classical**

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right) \]

**Strassen’s**

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} M \right) \]

**Fast:** \( \Theta(n^{\omega_0}) \) flops

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\omega_0} M \right) \]

---

**Seq**

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} M \right) \]

**Par**

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 8} \frac{M}{P} \right) \]

\[ + \]

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} \frac{M}{P} \right) \]

\[ + \]

\[ \Theta \left( \left( \frac{n}{\sqrt{M}} \right)^{\omega_0} \frac{M}{P} \right) \]

---

Units = (max) words communicated

\( O = \) algorithm exists, \( \Omega = \) lower bound exists, \( \Theta = \) both exist

\( n = \) matrix dimension, \( M = \) fast/local memory size, \( P = \) number of processors

References: [BDHS11], [BDH+12a], [BDH+12b], [BDHS12, BDHS14]
How small can $\omega_0$ get?

People have worked on this problem for decades!

- “fast” algorithms multiply matrices using $O(n^{\omega_0})$ flops, $\omega_0 < 3$

Exponent over time

- Classical
- Strassen
- Schonhage
- Coppersmith–Winograd
- Stothers/Williams/Le Gall

Ballard
How small can $\omega_0$ get?

People have worked on this problem for decades!

- “fast” algorithms multiply matrices using $O(n^{\omega_0})$ flops, $\omega_0 < 3$

Exponent over time

$\omega_0 = 2.373 \ldots$
How small can $\omega_0$ get?

People have worked on this problem for decades!

- “fast” algorithms multiply matrices using $O(n^{\omega_0})$ flops, $\omega_0 < 3$

Most fast algorithms are only theoretical because they

- involve approximations
  - $A \cdot B = C + \lambda E$
- are not explicit
  - only proofs of existence
- have (possibly) large constants or log factors
  - most theoreticians care about only the exponent $\omega$ in $O(n^{\omega + \epsilon})$

Exponent over time

$\omega_0 = 2.373 \ldots$
Practical Fast Algorithms

- Strassen’s algorithm *is* practical

- Many algorithms are better in theory, are any better in practice?

- Can we find practical algorithms that have been overlooked?

- Can we implement and benchmark all known algorithms?
Fast algorithms are based on recursion

Strassen showed how to use 7 multiplies instead of 8 for $2 \times 2$ multiplication

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

**Classical Algorithm**

\[
\begin{align*}
M_1 &= A_{11} \cdot B_{11} \\
M_2 &= A_{12} \cdot B_{21} \\
M_3 &= A_{11} \cdot B_{12} \\
M_4 &= A_{12} \cdot B_{22} \\
M_5 &= A_{21} \cdot B_{11} \\
M_6 &= A_{22} \cdot B_{21} \\
M_7 &= A_{21} \cdot B_{12} \\
M_8 &= A_{22} \cdot B_{22}
\end{align*}
\]

**Strassen’s Algorithm**

\[
\begin{align*}
M_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22}) \cdot B_{11} \\
M_3 &= A_{11} \cdot (B_{12} - B_{22}) \\
M_4 &= A_{22} \cdot (B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12}) \cdot B_{22} \\
M_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22})
\end{align*}
\]

\[
\begin{align*}
C_{11} &= M_1 + M_2 \\
C_{12} &= M_3 + M_4 \\
C_{21} &= M_5 + M_6 \\
C_{22} &= M_7 + M_8
\end{align*}
\]

\[
\begin{align*}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{align*}
\]
Recursion allows us to focus on base case

\[ 2 \times 2 \times 2 \]

\[
\begin{bmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{bmatrix}
\begin{bmatrix}
b_{11} & b_{12} \\
b_{21} & b_{22}
\end{bmatrix}
= 
\begin{bmatrix}
c_{11} & c_{12} \\
c_{21} & c_{22}
\end{bmatrix}
\]

<table>
<thead>
<tr>
<th>multiplies</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>flop count</td>
<td>( O(n^{2.58}) )</td>
<td>( O(n^{2.81}) )</td>
<td>( O(n^3) )</td>
</tr>
</tbody>
</table>
Recursion allows us to focus on base case

\[ 2 \times 2 \times 2 \]

\[
\begin{bmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
\end{bmatrix}
\begin{bmatrix}
  b_{11} & b_{12} \\
  b_{21} & b_{22}
\end{bmatrix}
= \begin{bmatrix}
  c_{11} & c_{12} \\
  c_{21} & c_{22}
\end{bmatrix}
\]

<table>
<thead>
<tr>
<th>multiplies</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>flop count</td>
<td>( O(n^{2.58}) )</td>
<td>( O(n^{2.81}) )</td>
<td>( O(n^3) )</td>
</tr>
</tbody>
</table>
Recursion allows us to focus on base case

\[ 2 \times 2 \times 2 \]

\[
\begin{bmatrix}
    a_{11} & a_{12} \\
    a_{21} & a_{22}
\end{bmatrix}
\begin{bmatrix}
    b_{11} & b_{12} \\
    b_{21} & b_{22}
\end{bmatrix}
= \begin{bmatrix}
    c_{11} & c_{12} \\
    c_{21} & c_{22}
\end{bmatrix}
\]

multiplies | 6 | 7 | 8
---|---|---|---
flop count | \(O(n^2.58)\) | \(O(n^{2.81})\) | \(O(n^3)\)

\[ 3 \times 3 \times 3 \]

\[
\begin{bmatrix}
    a_{11} & a_{12} & a_{13} \\
    a_{21} & a_{22} & a_{23} \\
    a_{31} & a_{32} & a_{33}
\end{bmatrix}
\begin{bmatrix}
    b_{11} & b_{12} & b_{13} \\
    b_{21} & b_{22} & b_{23} \\
    b_{31} & b_{32} & b_{33}
\end{bmatrix}
= \begin{bmatrix}
    c_{11} & c_{12} & c_{13} \\
    c_{21} & c_{22} & c_{23} \\
    c_{31} & c_{32} & c_{33}
\end{bmatrix}
\]

multiplies | 19 | 21 | 23 | 27
---|---|---|---|---
flop count | \(O(n^{2.68})\) | \(O(n^{2.77})\) | \(O(n^{2.85})\) | \(O(n^3)\)
Searching for a base case algorithm

Finding a better base case corresponds to computing a low-rank decomposition of a particular 3D tensor

\[ \mathcal{T} = \sum_{r=1}^{R} u_r \circ v_r \circ w_r \]

This is the main problem to solve

- various ways to attack it, but basically a search problem
- as base case gets bigger, tensor dimensions and rank get bigger
Matrix multiplication as a tensor operation

\[
A \cdot B = \begin{pmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{pmatrix} \cdot \begin{pmatrix}
b_{11} & b_{12} \\
b_{21} & b_{22}
\end{pmatrix} = \begin{pmatrix}
c_{11} & c_{12} \\
c_{21} & c_{22}
\end{pmatrix} = C
\]

is equivalent to

\[
\mathcal{T} \times_1 \begin{pmatrix} a_{11} \\ a_{12} \\ a_{21} \\ a_{22} \end{pmatrix} \times_2 \begin{pmatrix} b_{11} \\ b_{12} \\ b_{21} \\ b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} \\ c_{12} \\ c_{21} \\ c_{22} \end{pmatrix} = \mathcal{C}
\]

where \( \mathcal{T} \) is a \( 4 \times 4 \times 4 \) tensor with the following slices:

\[
\mathcal{T}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \mathcal{T}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \mathcal{T}_3 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathcal{T}_4 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}
\]

\( \Rightarrow \) Operation with low-rank decomposition
Low-rank decomposition for Strassen

\[ T = \sum_{r=1}^{7} u_r \circ v_r \circ w_r \]

Strassen’s decomposition is represented by these 3 factor matrices:

\[ U = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & -1 \end{pmatrix} \]

\[ V = \begin{pmatrix} 1 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 1 & 0 & 1 \end{pmatrix} \]

\[ W = \begin{pmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \]
Connection between factor matrices and algorithm

Strassen’s algorithm

\[ M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \]
\[ M_2 = (A_{21} + A_{22}) \cdot B_{11} \]
\[ M_3 = A_{11} \cdot (B_{12} - B_{22}) \]
\[ M_4 = A_{22} \cdot (B_{21} - B_{11}) \]
\[ M_5 = (A_{11} + A_{12}) \cdot B_{22} \]
\[ M_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \]
\[ M_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \]

\[ C_{11} = M_1 + M_4 - M_5 + M_7 \]
\[ C_{12} = M_3 + M_5 \]
\[ C_{21} = M_2 + M_4 \]
\[ C_{22} = M_1 - M_2 + M_3 + M_6 \]

Strassen’s factor matrices:

\[ U = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & -1 \end{bmatrix} \]
\[ V = \begin{bmatrix} 1 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 1 & 0 & 1 \end{bmatrix} \]
\[ W = \begin{bmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \]

\( U, V, W \) matrices encode the algorithm
Connection between factor matrices and algorithm

Strassen’s algorithm

Strassen’s factor matrices:

\[
\begin{align*}
M_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22}) \cdot B_{11} \\
M_3 &= A_{11} \cdot (B_{12} - B_{22}) \\
M_4 &= A_{22} \cdot (B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12}) \cdot B_{22} \\
M_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{align*}
\]

\[
\begin{array}{cccccccc}
&M_1&M_2&M_3&M_4&M_5&M_6&M_7 \\
\hline
\textbf{U} & A_{11} & 1 & 1 & 1 & -1 & & \\
& A_{12} & & & & & 1 & 1 \\
& A_{21} & 1 & & & & 1 & \\
& A_{22} & 1 & 1 & 1 & & -1 & \\
\hline
\textbf{V} & B_{11} & 1 & 1 & -1 & 1 & & \\
& B_{12} & & & & & 1 & 1 \\
& B_{21} & & & 1 & & 1 & \\
& B_{22} & 1 & -1 & 1 & 1 & & \\
\hline
\textbf{W} & C_{11} & 1 & 1 & -1 & 1 & & \\
& C_{12} & & & & 1 & & \\
& C_{21} & 1 & & 1 & & & \\
& C_{22} & 1 & -1 & 1 & & & 1 \\
\end{array}
\]

\textbf{U, V, W matrices encode the algorithm}
Main search problem

Given base case dimensions $M$, $P$, and $N$ (multiplying $M \times P$ and $P \times N$ matrices), the tensor $\mathcal{T} \in \{0, 1\}^{MP \times PN \times MN}$ is specified.
Main search problem

Given base case dimensions $M$, $P$, and $N$ (multiplying $M \times P$ and $P \times N$ matrices), the tensor $\mathcal{T} \in \{0, 1\}^{MP \times PN \times MN}$ is specified.

Then for some desired rank $R < MNP$, find

$$U \in \mathbb{F}^{MP \times R}, \ V \in \mathbb{F}^{PN \times R}, \ W \in \mathbb{F}^{MN \times R}$$

such that

$$t_{ijk} = \sum_{r=1}^{R} u_{ir} v_{jr} w_{kr} \quad \text{for all} \quad i, j, k$$

(these $(MNP)^2$ scalar constraints are equivalent to $\mathcal{T} = \sum u_r \circ v_r \circ w_r$)
Main search problem

Given base case dimensions $M$, $P$, and $N$ (multiplying $M \times P$ and $P \times N$ matrices), the tensor $\mathcal{J} \in \{0, 1\}^{MP \times PN \times MN}$ is specified.

Then for some desired rank $R < MNP$, find

$$U \in \mathbb{F}^{MP \times R}, V \in \mathbb{F}^{PN \times R}, W \in \mathbb{F}^{MN \times R}$$

such that

$$t_{ijk} = \sum_{r=1}^{R} u_{ir} v_{jr} w_{kr} \quad \text{for all} \quad i, j, k$$

(These $(MNP)^2$ scalar constraints are equivalent to $\mathcal{J} = \sum u_r \circ v_r \circ w_r$)

- solution corresponds to algorithm with $\omega_0 = 3 \log_{MPN} R$
How do you solve it?

**Problem:** Find $\mathbf{U}, \mathbf{V}, \mathbf{W}$ such that $\mathcal{T} = \sum \mathbf{u}_r \circ \mathbf{v}_r \circ \mathbf{w}_r$

- the problem is NP-complete (for general $\mathcal{T}$)
- many combinatorial formulations of the problem
- efficient numerical methods can compute low-rank *approximations*
  - typical approach is “alternating least squares” (ALS)
  - pitfall: getting stuck at local minima $> 0$
  - pitfall: facing ill-conditioned linear least squares problems
  - pitfall: numerical solution is good only to machine precision

- we seek exact, discrete, and sparse solutions
Alternating least squares with regularization

Most successful scheme due to Smirnov [Smi13]

Repeat

1. \[ U = \arg\min_U \| T(U) - U(W \odot V)^T \|_F^2 + \lambda \| U - \tilde{U} \|_F^2 \]

2. \[ V = \arg\min_V \| T(V) - V(W \odot U)^T \|_F^2 + \lambda \| V - \tilde{V} \|_F^2 \]

3. \[ W = \arg\min_W \| T(W) - W(V \odot U)^T \|_F^2 + \lambda \| W - \tilde{W} \|_F^2 \]

Until convergence

Art of optimization scheme in tinkering with \( \lambda, \tilde{U}, \tilde{V}, \tilde{W} \) (each iteration)
## Discovered algorithms

<table>
<thead>
<tr>
<th>Algorithm base case</th>
<th>Multiplies (fast)</th>
<th>Multiplies (classical)</th>
<th>Speedup per recursive step</th>
<th>$\omega_0$</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⟩ [Str69]</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>27</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⟩ [Smi13]</td>
<td>40</td>
<td>54</td>
<td>35%</td>
<td>2.77</td>
</tr>
<tr>
<td>⟨2, 2, 3⟩* [BCRL79]</td>
<td>10</td>
<td>12</td>
<td>20%</td>
<td>2.78</td>
</tr>
<tr>
<td>⟨3, 3, 3⟩* [Sch81]</td>
<td>21</td>
<td>27</td>
<td>29%</td>
<td>2.77</td>
</tr>
</tbody>
</table>
Example algorithm: $\langle 4, 2, 4 \rangle$

Partition matrices like this:

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

1. Take 26 linear combos of $A_{ij}$’s according to $U$ (68 adds)
2. Take 26 linear combos of $B_{ij}$’s according to $V$ (52 adds)
3. Perform 26 multiplies (recursively)
4. Take linear combos of outputs to form $C_{ij}$’s acc. to $W$ (69 adds)

Classical algorithm performs 32 multiplies yielding a possible speedup of 23% per step
### Discovered algorithms

<table>
<thead>
<tr>
<th>Algorithm base case</th>
<th>Multiplies (fast)</th>
<th>Multiplies (classical)</th>
<th>Speedup per recursive step</th>
<th>$\omega_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle 2, 2, 3 \rangle$</td>
<td>11</td>
<td>12</td>
<td>9%</td>
<td>2.89</td>
</tr>
<tr>
<td>$\langle 2, 2, 5 \rangle$</td>
<td>18</td>
<td>20</td>
<td>11%</td>
<td>2.89</td>
</tr>
<tr>
<td>$\langle 2, 2, 2 \rangle$</td>
<td>7</td>
<td>8</td>
<td>14%</td>
<td>2.81</td>
</tr>
<tr>
<td>$\langle 3, 3, 3 \rangle$</td>
<td>23</td>
<td>27</td>
<td>17%</td>
<td>2.85</td>
</tr>
<tr>
<td>$\langle 2, 3, 3 \rangle$</td>
<td>15</td>
<td>18</td>
<td>20%</td>
<td>2.81</td>
</tr>
<tr>
<td>$\langle 2, 3, 4 \rangle$</td>
<td>20</td>
<td>24</td>
<td>20%</td>
<td>2.83</td>
</tr>
<tr>
<td>$\langle 2, 4, 4 \rangle$</td>
<td>26</td>
<td>32</td>
<td>23%</td>
<td>2.82</td>
</tr>
<tr>
<td>$\langle 3, 3, 4 \rangle$</td>
<td>29</td>
<td>36</td>
<td>24%</td>
<td>2.82</td>
</tr>
<tr>
<td>$\langle 3, 4, 4 \rangle$</td>
<td>38</td>
<td>48</td>
<td>26%</td>
<td>2.82</td>
</tr>
<tr>
<td>$\langle 3, 3, 6 \rangle$</td>
<td>40</td>
<td>54</td>
<td>35%</td>
<td>2.77</td>
</tr>
<tr>
<td>$\langle 2, 2, 3 \rangle^*$</td>
<td>10</td>
<td>12</td>
<td>20%</td>
<td>2.78</td>
</tr>
<tr>
<td>$\langle 3, 3, 3 \rangle^*$</td>
<td>21</td>
<td>27</td>
<td>29%</td>
<td>2.77</td>
</tr>
</tbody>
</table>
How do these algorithms perform in practice?

- All these algorithms have the same structure:
  - perform additions according to $U, V, W$, and make recursive calls

- Code generator can translate $U, V, W$ into an implementation

- Sequential performance is based on:
  - classical multiplication implementation performance (vendor library)
  - efficiency of additions
  - crossover point of fast to classical

- Parallel performance depends also on parallelization approach
  - and hardware parameters
Classical performance

Intel’s Math Kernel Library (MKL) $\texttt{dgemm}$
Square Matrix Multiplication (Sequential)

The shape of the $\texttt{dgemm}$ curve gives a rule of thumb for the crossover point.

- $\texttt{dgemm}$ curve shape gives a rule of thumb for the crossover point.
Performing (and optimizing) additions

Additions are completely memory bandwidth bound
  - time is proportional to communication (flops are free)

We micro-benchmarked three approaches:
  1. Pairwise: most straightforward
  2. Streaming: minimizes communication (in theory)
  3. Write-once: best performance

We also considered common subexpression elimination
  1. can help pairwise and streaming approaches
  2. often hurts write-once approach
Write-once approach to additions

\[ S_1 = A_{11} - A_{12} + A_{22} \]
\[ S_2 = A_{21} - A_{22} \]
\[ \vdots \]
Sequential performance of fast algorithms

Square Matrix Multiplication

<table>
<thead>
<tr>
<th>Dimension (n)</th>
<th>Effective GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16</td>
</tr>
<tr>
<td>2000</td>
<td>18</td>
</tr>
<tr>
<td>3000</td>
<td>20</td>
</tr>
<tr>
<td>4000</td>
<td>22</td>
</tr>
<tr>
<td>5000</td>
<td>24</td>
</tr>
<tr>
<td>6000</td>
<td>26</td>
</tr>
<tr>
<td>7000</td>
<td>28</td>
</tr>
<tr>
<td>8000</td>
<td>30</td>
</tr>
</tbody>
</table>
Sequential performance of fast algorithms

Square Matrix Multiplication

<table>
<thead>
<tr>
<th>Dimension (n)</th>
<th>Effective GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;3,2,3&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;3,3,3&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;4,2,4&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;3,3,6&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Ballard
Sequential performance of fast algorithms

Rectangular Matrix Multiplication

- best algorithms match “shape” of problem
Parallelization schemes: recursion tree traversal

We consider 3 methods for shared-memory parallelization, based on traversing recursion tree
Parallelization schemes: recursion tree traversal

**DFS**: depth first search is simplest scheme
- all parallelism in calls to dgemm, always load balanced
- requires large subproblems for high performance
Parallelization schemes: recursion tree traversal

**BFS**: breadth first search relies on sequential *dgemm*
- maintains high performance for small subproblems
- load balancing of multiplies is no longer guaranteed

2 steps of Strassen creates 49 subproblems; we have 24 cores
Parallelization schemes: recursion tree traversal

HYBRID

- use BFS as much as possible
- use DFS to load balance leftovers

2 steps of Strassen creates 49 subproblems; we have 24 cores
Parallel performance of Strassen on <n,n,n>

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

At 24 threads, not only are the additions bandwidth bound, but they don’t scale as well as the multiplies (bandwidth scaling is < 6 ×).
Parallel performance of fast algorithms (square case)

- at 24 threads, not only are the additions bandwidth bound, but they don’t scale as well as the multiplies (bandwidth scaling is < 6×).
Parallel performance of fast algorithms (rect case)

Parallel performance of \( <4,2,4> \) on \( <n,2800,n> \)

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

- In theory, fast algorithms reduce both computation and communication.
- In practice, fast algorithms like Strassen’s can outperform `dgemm`.
- For square matrices, Strassen’s algorithm is hard to beat.
- For rectangular matrices, algorithm should match the shape.
- Shared-memory parallelization faces bandwidth bottleneck.
Open questions

1. What are the numerical properties of all these algorithms?

2. How will they perform on distributed-memory parallel architectures?

3. Are there applications that can benefit from approximate algorithms?

4. Have we exhausted the possibilities of practical fast algorithms?

5. Can we use fast algorithms in the context of linear algebra and other applications?
How Practical is Fast Matrix Multiplication?

Grey Ballard

For more details, see tech report:
http://arxiv.org/abs/1409.2908

gmballa@sandia.gov
www.sandia.gov/~gmballa
1. Communication models
2. Matmul-as-tensor-operation using low-rank decomposition
3. Classical algorithm’s factor matrices
4. Bini’s factor matrices
5. Code generator performance comparison
Runtime Model

Measure computation in terms of \# flops performed

Time per flop: $\gamma$

Measure communication in terms of \# words communicated

Time per word: $\beta$

Total running time of an algorithm (ignoring overlap):

$$\gamma \cdot (\# \text{ flops}) + \beta \cdot (\# \text{ words})$$

$\beta \gg \gamma$ as measured in time and energy, and the relative cost of communication is increasing
Here’s the matrix multiplication as tensor operation again:

$$\mathcal{T} \times_1 \mathbf{a} \times_2 \mathbf{b} = \mathbf{c}$$

Here’s our low-rank decomposition:

$$\mathcal{T} = \sum_{r=1}^{R} \mathbf{u}_r \circ \mathbf{v}_r \circ \mathbf{w}_r$$

Here’s an encoding of our new matrix multiplication algorithm:

$$\mathcal{T} \times_1 \mathbf{a} \times_2 \mathbf{b} = \sum_{r=1}^{R} (\mathbf{u}_r \circ \mathbf{v}_r \circ \mathbf{w}_r) \times_1 \mathbf{a} \times_2 \mathbf{b} = \sum_{r=1}^{R} (\mathbf{a}^T \mathbf{u}_r) \cdot (\mathbf{b}^T \mathbf{v}_r) \cdot \mathbf{w}_r$$
Connection between factor matrices and algorithm

Classical algorithm:

\[
M_1 = A_{11} \cdot B_{11} \\
M_2 = A_{12} \cdot B_{21} \\
M_3 = A_{11} \cdot B_{12} \\
M_4 = A_{12} \cdot B_{22} \\
M_5 = A_{21} \cdot B_{11} \\
M_6 = A_{22} \cdot B_{21} \\
M_7 = A_{21} \cdot B_{12} \\
M_8 = A_{22} \cdot B_{22}
\]

\[
C_{11} = M_1 + M_2 \\
C_{12} = M_3 + M_4 \\
C_{21} = M_5 + M_6 \\
C_{22} = M_7 + M_8
\]

Classical factor matrices:

\[
U = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}
\]

\[
V = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}
\]

\[
W = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}
\]
Factor matrices for an approximate algorithm (Bini’s)

\[ U = \begin{bmatrix}
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \lambda & \lambda & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda & \lambda & \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{bmatrix} \]

\[ V = \begin{bmatrix}
\lambda & 0 & 0 & -\lambda & 0 & 1 & 1 & -1 & 1 & 0 \\
0 & 0 & 0 & 0 & \lambda & 0 & 0 & -1 & 0 & 1 \\
0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & \lambda & 0 \\
1 & -1 & 1 & 0 & 1 & \lambda & 0 & 0 & 0 & -\lambda \\
\end{bmatrix} \]

\[ W = \begin{bmatrix}
\frac{1}{\lambda} & \frac{1}{\lambda} & -\frac{1}{\lambda} & \frac{1}{\lambda} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -\frac{1}{\lambda} & 0 & \frac{1}{\lambda} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & -1 & 0 \\
1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{\lambda} & 0 & \frac{1}{\lambda} & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{1}{\lambda} & -\frac{1}{\lambda} & \frac{1}{\lambda} & 0 & \frac{1}{\lambda} \\
\end{bmatrix} \]
Code generated vs tuned performance

Sequential performance on N x N x N

MKL DGEMM
Code generated Strassen
D’Alberto Strassen–Winograd

Effective GFLOPS vs dimension (N)

Tuned code: [DBN11]

$O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication.


Brief announcement: strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds.


Communication-optimal parallel algorithm for Strassen’s matrix multiplication.


Graph expansion and communication costs of fast matrix multiplication.


Graph expansion and communication costs of fast matrix multiplication.


Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz.

Communication costs of strassen’s matrix multiplication.

Paolo D’Alberto, Marco Bodrato, and Alexandru Nicolau.
Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: Matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation.

A. Schönhage.
Partial and total matrix multiplication.

A.V. Smirnov.
The bilinear complexity and practical algorithms for matrix multiplication.

V. Strassen.
Gaussian elimination is not optimal.
10.1007/BF02165411.