Discovering faster matrix multiplication algorithms with reinforcement learning

Nov 2022

Matrix multiplication is ubiquitous

- Scientific computing: NLA, NPDE
- Statistical computing: MCMC, EM
- Machine learning
- …

- LAPACK
- BLAS library:
  - BLAS is a collection of low-level matrix and vector arithmetic operations（“multiply a vector by a scalar”，“multiply two matrices and add to a third matrix”，etc …)

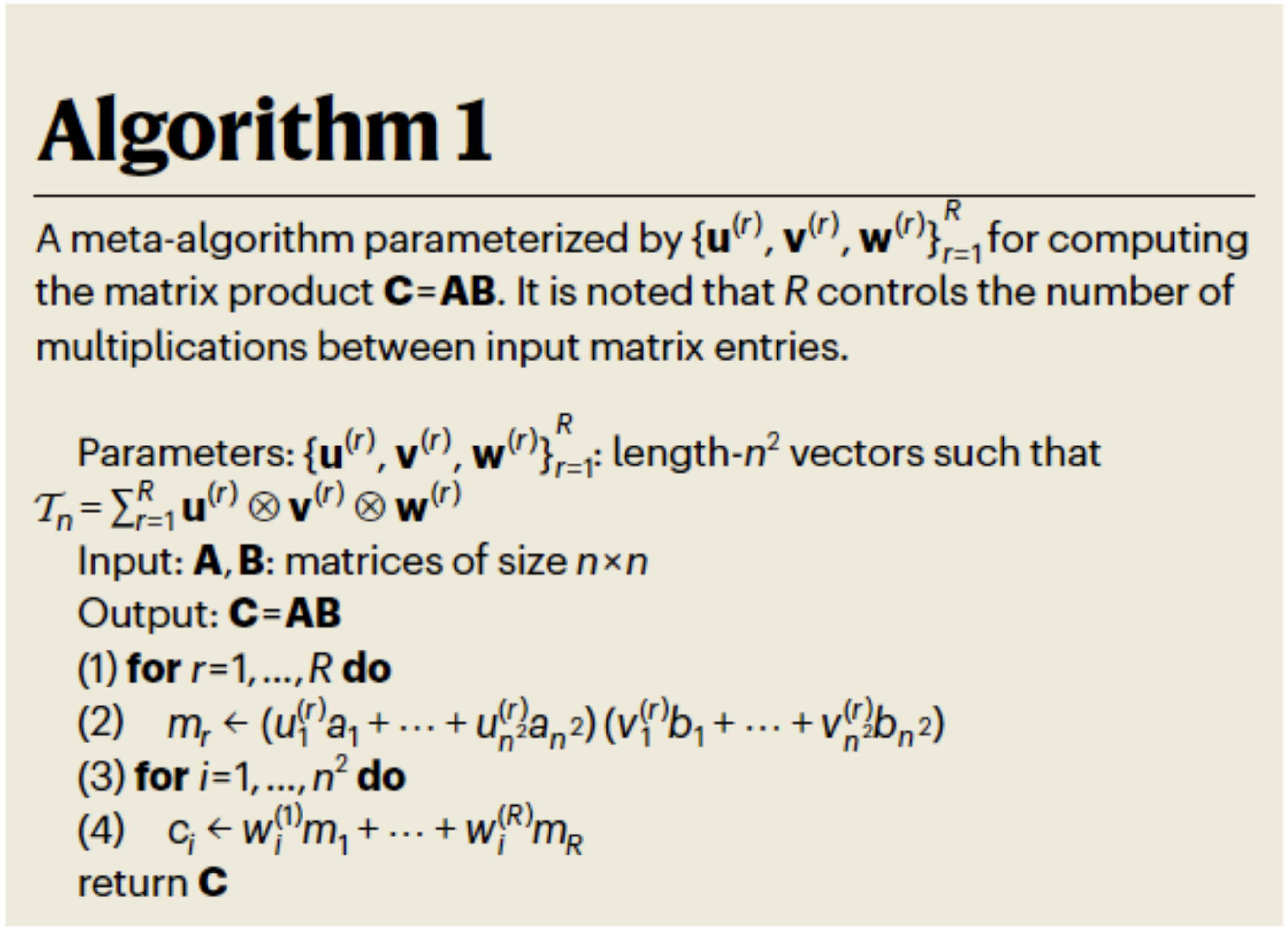
Matrix multiplication as tensor decomposition

- Use a 3D tensor to denote the multiplication of two matrices:
  - A: n∗m matrix, B: m∗p matrix, C=AB n∗p matrix
  - $T_{A,B,C}$: $nm \times mp \times np$ tensor

- Find a low rank decomposition of this tensor:

$$\mathcal{T}_n = \sum_{r=1}^{R} \mathbf{u}^{(r)} \otimes \mathbf{v}^{(r)} \otimes \mathbf{w}^{(r)},$$

Matrix multiplication as tensor decomposition



Algorithm 1

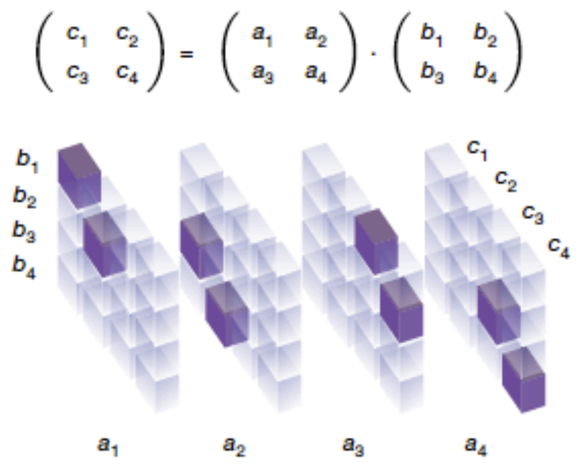
A meta-algorithm parameterized by $\{\mathbf{u}^{(r)}, \mathbf{v}^{(r)}, \mathbf{w}^{(r)}\}_{r=1}^{R}$ for computing the matrix product $\mathbf{C}=\mathbf{AB}$. It is noted that R controls the number of multiplications between input matrix entries.

```
  Parameters: $\{\mathbf{u}^{(r)}, \mathbf{v}^{(r)}, \mathbf{w}^{(r)}\}_{r=1}^{R}$: length-$n^2$ vectors such that
$\mathcal{T}_n = \sum_{r=1}^{R} \mathbf{u}^{(r)} \otimes \mathbf{v}^{(r)} \otimes \mathbf{w}^{(r)}$
  Input: $\mathbf{A}, \mathbf{B}$: matrices of size $n \times n$
  Output: $\mathbf{C}=\mathbf{AB}$
  (1) for $r=1,\dots,R$ do
  (2)     $m_r \leftarrow (u_1^{(r)}a_1 + \dots + u_{n^2}^{(r)}a_{n^2})(v_1^{(r)}b_1 + \dots + v_{n^2}^{(r)}b_{n^2})$
  (3) for $i=1,\dots,n^2$ do
  (4)     $c_i \leftarrow w_i^{(1)}m_1 + \dots + w_i^{(R)}m_R$
  return $\mathbf{C}$
```

Strassen algorithm

- Asymptotic complexity: $O(n^{2.8})$

a



$\begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \cdot \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}$
b_1
b_2
b_3
b_4
c_1
c_2
c_3
c_4
a_1
a_2
a_3
a_4

b

$$
\begin{aligned}
m_1 &= (a_1 + a_4)(b_1 + b_4) \\
m_2 &= (a_3 + a_4)\,b_1 \\
m_3 &= a_1\,(b_2 - b_4) \\
m_4 &= a_4\,(b_3 - b_1) \\
m_5 &= (a_1 + a_2)\,b_4 \\
m_6 &= (a_3 - a_1)(b_1 + b_2) \\
m_7 &= (a_2 - a_4)(b_3 + b_4) \\
c_1 &= m_1 + m_4 - m_5 + m_7 \\
c_2 &= m_3 + m_5 \\
c_3 &= m_2 + m_4 \\
c_4 &= m_1 - m_2 + m_3 + m_6
\end{aligned}
$$

c

$$
\mathbf{U} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & -1 \end{pmatrix}
$$

$$
\mathbf{V} = \begin{pmatrix} 1 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 1 & 0 & 1 \end{pmatrix}
$$

$$
\mathbf{W} = \begin{pmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}
$$

Fig. 1 | Matrix multiplication tensor and algorithms. a, Tensor $\mathcal{T}_2$ representing the multiplication of two 2 × 2 matrices. Tensor entries equal to 1 are depicted in purple, and 0 entries are semi-transparent. The tensor specifies which entries from the input matrices to read, and where to write the result. For example, as $c_1 = a_1b_1 + a_2b_3$, tensor entries located at (a_1, b_1, c_1) and (a_2, b_3, c_1) are set to 1. **b**, Strassen's algorithm[2] for multiplying 2 × 2 matrices using 7 multiplications. **c**, Strassen's algorithm in tensor factor representation. The stacked factors **U**, **V** and **W** (green, purple and yellow, respectively) provide a rank-7 decomposition of $\mathcal{T}_2$ (equation (1)). The correspondence between arithmetic operations (**b**) and factors (**c**) is shown by using the aforementioned colours.

Practical issue

- Is Strassen algorithm really used in practice?
  - Not really, reasons include:

  - Large constant factor
  - Sparse structured matrix
  - Parallelism
  - Caching and architecture specific quirks

Tensor Game

- Modeled as a single player game:
  - State variable: S_t, $S_0 = T_n$
  - Action variable: $u^t \otimes v^t \otimes w^t$, $S_{t+1} = S_t - u^t \otimes v^t \otimes w^t$ with entries in F = {-2, -1, 0, 1, 2}
  - Reward
  - Limit time step
- Use DNN to guide Monte Carlo tree search for action:
  - Network takes input state variables and history output distribution over action and reward (rank guess of the tensor)

Tensor Game

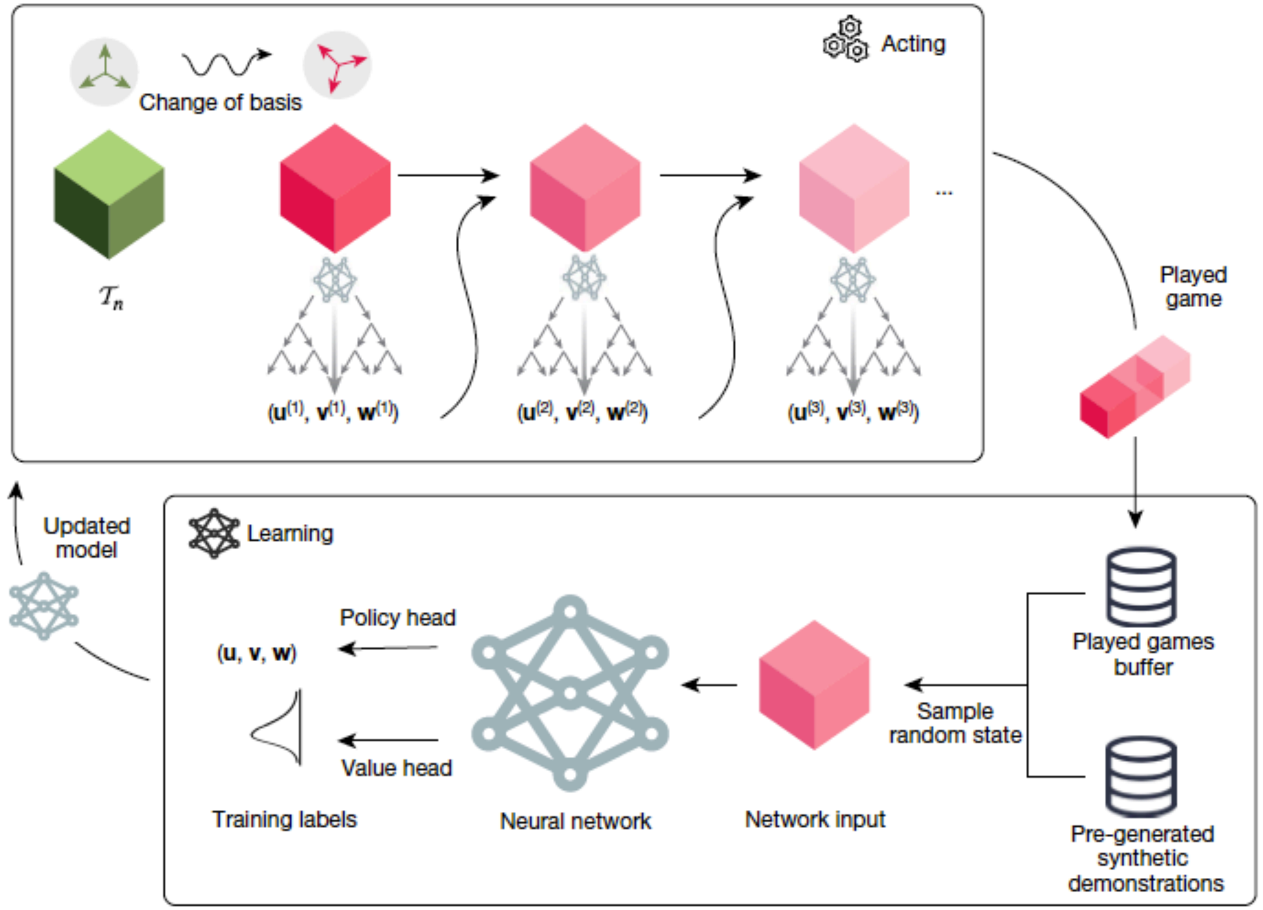
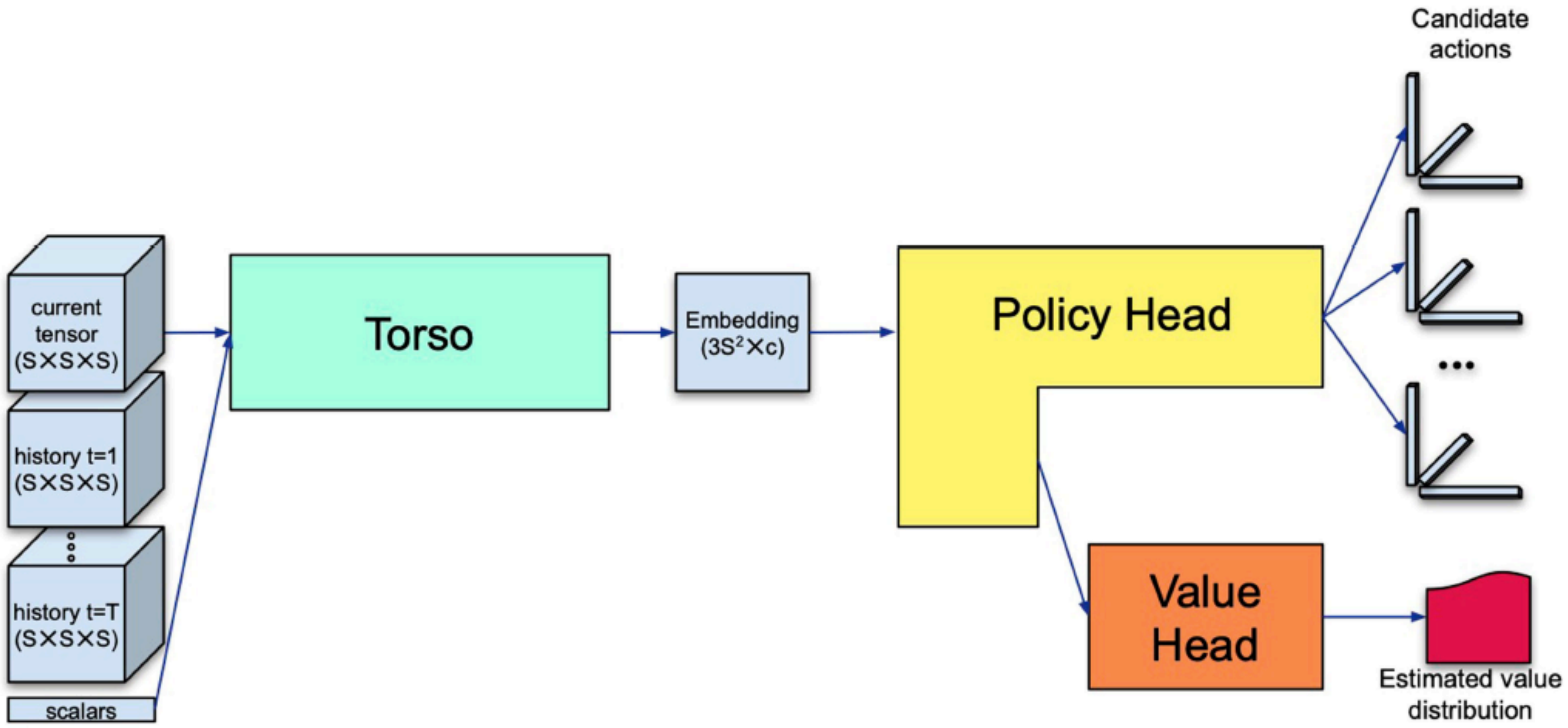

Acting
Change of basis
$\mathcal{T}_n$
$(\mathbf{u}^{(1)}, \mathbf{v}^{(1)}, \mathbf{w}^{(1)})$
$(\mathbf{u}^{(2)}, \mathbf{v}^{(2)}, \mathbf{w}^{(2)})$
$(\mathbf{u}^{(3)}, \mathbf{v}^{(3)}, \mathbf{w}^{(3)})$
...
Played game
Updated model
Learning
Policy head
(u, v, w)
Value head
Training labels
Neural network
Network input
Sample random state
Played games buffer
Pre-generated synthetic demonstrations

Fig. 2 | Overview of AlphaTensor. The neural network (bottom box) takes as input a tensor $\mathcal{S}_t$, and outputs samples $(\mathbf{u}, \mathbf{v}, \mathbf{w})$ from a distribution over potential next actions to play, and an estimate of the future returns (for example, of $-\text{Rank}(\mathcal{S}_t)$). The network is trained on two data sources: previously played games and synthetic demonstrations. The updated network is sent to the actors (top box), where it is used by the MCTS planner to generate new games.

Network architecture



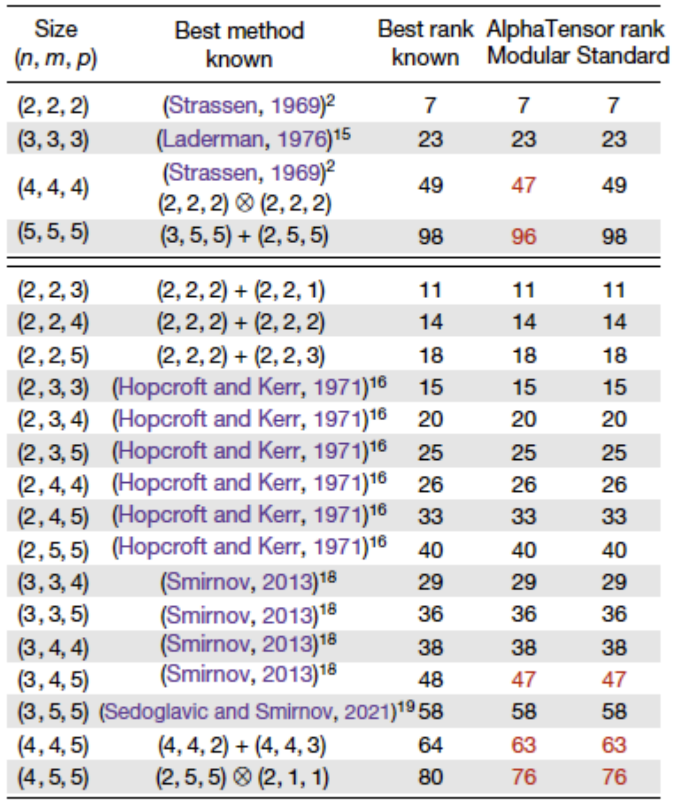
Extended Data Fig. 3 | AlphaTensor's network architecture. The network takes as input the list of tensors containing the current state and previous history of actions, and a list of scalars, such as the time index of the current action. It produces two kinds of outputs: one representing the value, and the other inducing a distribution over the action space from which we can sample from. The architecture of the network is accordingly designed to have a common torso, and two heads, the value and the policy heads. *c* is set to 512 in all experiments.

Data generation

- Target tensor
- Synthetic data
- Data augmentation: basis change:

$$\mathcal{T}_{ijk}^{(\mathbf{A},\mathbf{B},\mathbf{C})} = \sum_{a=1}^{S} \sum_{b=1}^{S} \sum_{c=1}^{S} \mathbf{A}_{ia} \mathbf{B}_{jb} \mathbf{C}_{kc} \mathcal{T}_{abc}.$$

Algorithm discovery



Size (n, m, p)	Best method known	Best rank known	AlphaTensor rank Modular	AlphaTensor rank Standard
(2, 2, 2)	(Strassen, 1969)[2]	7	7	7
(3, 3, 3)	(Laderman, 1976)[15]	23	23	23
(4, 4, 4)	(Strassen, 1969)[2] (2, 2, 2) ⊗ (2, 2, 2)	49	47	49
(5, 5, 5)	(3, 5, 5) + (2, 5, 5)	98	96	98
(2, 2, 3)	(2, 2, 2) + (2, 2, 1)	11	11	11
(2, 2, 4)	(2, 2, 2) + (2, 2, 2)	14	14	14
(2, 2, 5)	(2, 2, 2) + (2, 2, 3)	18	18	18
(2, 3, 3)	(Hopcroft and Kerr, 1971)[16]	15	15	15
(2, 3, 4)	(Hopcroft and Kerr, 1971)[16]	20	20	20
(2, 3, 5)	(Hopcroft and Kerr, 1971)[16]	25	25	25
(2, 4, 4)	(Hopcroft and Kerr, 1971)[16]	26	26	26
(2, 4, 5)	(Hopcroft and Kerr, 1971)[16]	33	33	33
(2, 5, 5)	(Hopcroft and Kerr, 1971)[16]	40	40	40
(3, 3, 4)	(Smirnov, 2013)[18]	29	29	29
(3, 3, 5)	(Smirnov, 2013)[18]	36	36	36
(3, 4, 4)	(Smirnov, 2013)[18]	38	38	38
(3, 4, 5)	(Smirnov, 2013)[18]	48	47	47
(3, 5, 5)	(Sedoglavic and Smirnov, 2021)[19]	58	58	58
(4, 4, 5)	(4, 4, 2) + (4, 4, 3)	64	63	63
(4, 5, 5)	(2, 5, 5) ⊗ (2, 1, 1)	80	76	76

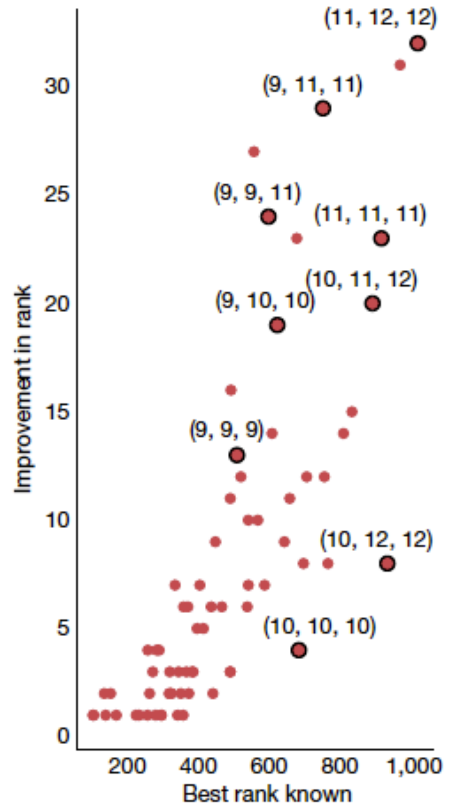


Fig. 3 | Comparison between the complexity of previously known matrix multiplication algorithms and the ones discovered by AlphaTensor. Left: column (n, m, p) refers to the problem of multiplying $n \times m$ with $m \times p$ matrices. The complexity is measured by the number of scalar multiplications (or equivalently, the number of terms in the decomposition of the tensor). 'Best rank known' refers to the best known upper bound on the tensor rank (before this paper), whereas 'AlphaTensor rank' reports the rank upper bounds obtained with our method, in modular arithmetic ($\mathbb{Z}_2$) and standard arithmetic. In all cases, AlphaTensor discovers algorithms that match or improve over known state of the art (improvements are shown in red). See Extended Data Figs. 1 and 2 for examples of algorithms found with AlphaTensor. Right: results (for arithmetic in $\mathbb{R}$) of applying AlphaTensor-discovered algorithms on larger tensors. Each red dot represents a tensor size, with a subset of them labelled. See Extended Data Table 1 for the results in table form. State-of-the-art results are obtained from the list in ref. [64].

Hardware Performance

- The test case is a matrix multiplication of size 8192, considering as a 4-by-4 blockwise with each block of size 2048. Finding the algorithm which provides the best combination of these blocks to provide the best acceleration
- Two hardwares are considered: Nvidia V100 GPU and Google TPU v2
- The baseline is the cuBLAS for GPU
- The acceleration also scales to larger dimension matrices

Hardware Performance

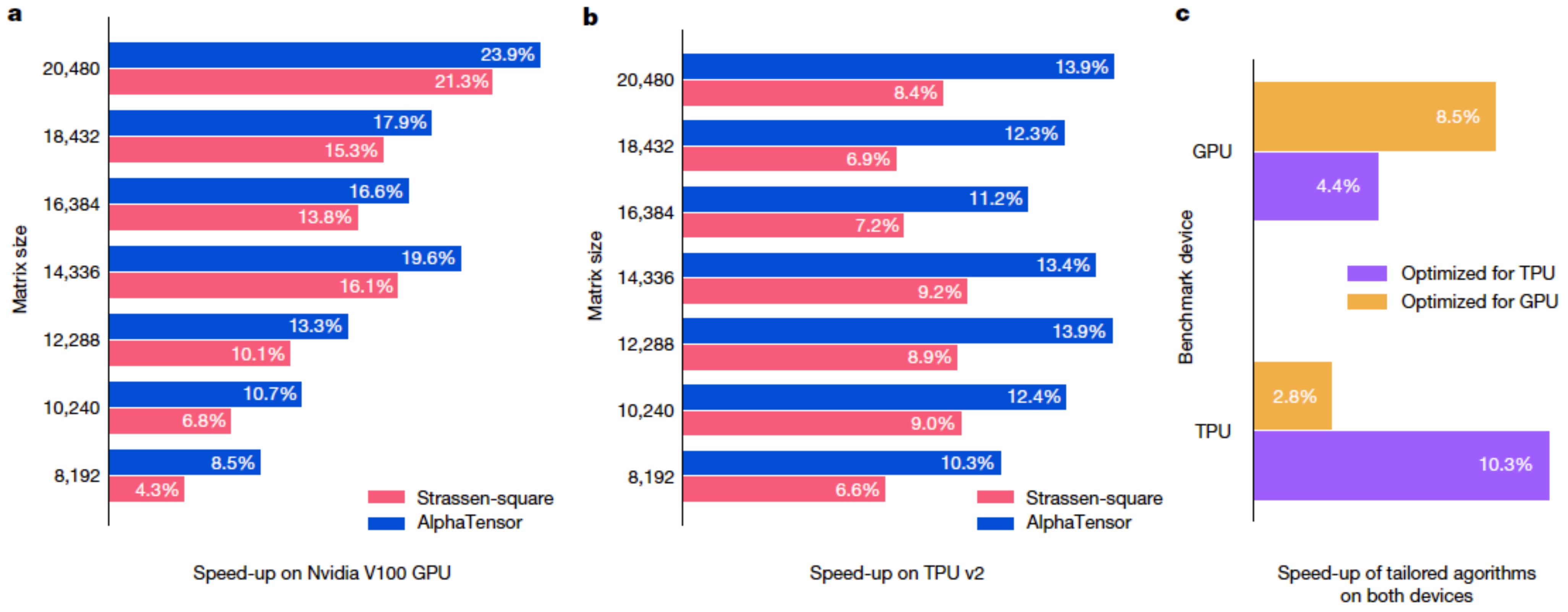


Fig. 5 | Speed-ups of the AlphaTensor-discovered algorithm. a,b, Speed-ups (%) of the AlphaTensor-discovered algorithms tailored for a GPU (**a**) and a TPU (**b**), optimized for a matrix multiplication of size 8,192 × 8,192. Speed-ups are measured relative to standard (for example, cuBLAS for the GPU) matrix multiplication on the same hardware. Speed-ups are reported for various matrix sizes (despite optimizing the algorithm only on one matrix size). We also report the speed-up of the Strassen-square algorithm. The median speed-up is reported over 200 runs. The standard deviation over runs is <0.4 percentage points (see Supplementary Information for more details). **c**, Speed-up of both algorithms (tailored to a GPU and a TPU) benchmarked on both devices.

Related work

- Continuous based approach: not exact multiplication
- Symmetry of matric multiplication