Successive Approximation for Coded Matrix Multiplication

Shahrzad Kianidehkordi and Stark C. Draper

IEEE International Symposium on Information Theory

June 28, 2022

Outline

Speed Up Large-Scale Matrix Multiplication

Motivation:

- Matrices are fundamental mathematical structures for representing data and can be too large to fit in memory.
- Matrix multiplication is a core building block for numerous scientific computing and can be too time consuming.

$\Rightarrow$ Large-scale matrix multiplication needs to be parallelized across multiple working nodes and/or approximated.

Problem of stragglers: Some workers compute and/or communicate too slowly, making delay into the parallel system.

Combine coded and approximated computing: While most efforts on coded computing to date have focused on exact recovery of the target computation, in lots of applications (gradient descent, etc.) inexact will suffice.

$\Rightarrow$ We can tradeoff speed and accuracy.

Coded computing using polynomial bases

Notations: N workers and a master multiply $A \in \mathcal{R}^{N_x \times N_z}$ and $B \in \mathcal{R}^{N_z \times N_y}$.

Step 1/5: Master partitions A and B into submatrices.

- Exp) In MatDot, A vert, B horizon, AB to sum of K outer products (e.g., $K = 2$).

$$\begin{bmatrix} A_1 & A_2 \end{bmatrix} \times \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = A_1 \times B_1 + A_2 \times B_2$$

Step 2/5: Master encodes data using poly basis: $\{T_k(x)\}_{k=1}^{K}$.

- Produce encoding polys $A(x)$ and $B(x)$.
- Exp) In MatDot, $T_k(x)$ are monomial basis (e.g., $K = 2$).

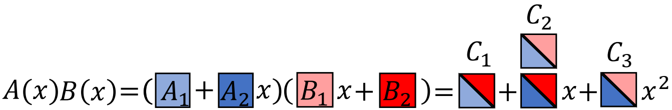
$$A(x) = A_1 + A_2 x, \quad B(x) = B_1 x + B_2$$

- Evaluate $(A(x), B(x))$ at $x \in \{x_n\}_{n=1}^{N}$; send $(A(x_n), B(x_n))$ to worker $n \in [N]$.

Coded computing using polynomial bases (con't)

Step 3/5: Worker n computes $A(x_n)B(x_n)$; sends it to the master.
Step 4/5: Master decodes $A(x_n)B(x_n)$ to recover $\{C_r\}_{r=1}^{R}$ coeffs.

- Exp) MatDot solves Vander system of equations (e.g., $R = 3$)



$$C_1 = A_1B_2,\ C_2 = A_1B_1 + A_2B_2,\ C_3 = A_2B_1$$

$$\begin{bmatrix} A(x_{i_1})B(x_{i_1}) \\ A(x_{i_2})B(x_{i_2}) \\ A(x_{i_3})B(x_{i_3}) \end{bmatrix} = \begin{bmatrix} 1 & x_{i_1} & x_{i_1}^2 \\ 1 & x_{i_2} & x_{i_2}^2 \\ 1 & x_{i_3} & x_{i_3}^2 \end{bmatrix} \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix}$$

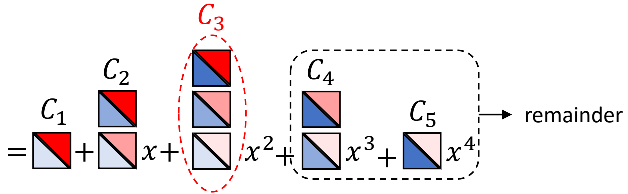
Step 5/5: In some codes, master needs post-decoding:

- Interpolate $A(x)B(x)$ at $x \in \{y_k\}_{k=1}^{K}$; calculate $AB = \sum_{k=1}^{K} \alpha_k A(y_k)B(y_k)$.

Benchmark: ϵ-Approximate MatDot (ϵAMD) [Jeong et al]

Exp: $K = 3$, use same enc polys as MatDot.

$$A(x)B(x) = (A_1 + A_2 x + A_3 x^2)(B_1 x^2 + B_2 x + B_3)$$



Idea: Remainder is small if evaluate $A(x)$, $B(x)$ at small x ($\|A_i\|_\mathrm{F}$, $\|B_i\|_\mathrm{F}$ bounded).

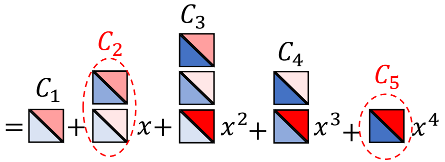
- Approximate recovery poly is $P(x) := C_1 + C_2 x + C_3 x^2 \approx A(x)B(x)$ if x is small.
- The desired AB product equals $C_3 = (A_1 B_1 + A_2 B_2 + A_3 B_3)$
- $P(x)$ is degree-2 $\Rightarrow$ Approximate recovery threshold is $R_1 = 3$.
- $A(x)B(x)$ is degree-4 $\Rightarrow$ Recovery threshold is $R = 5$.

Group-wise SAC, $D = 2$

Goal: Extend single-layer approx of ϵAMD by getting mid coeff to show up elsewhere.
Idea # 1: Carefully re-order the coeffs of enc polys.

$$\boldsymbol{\epsilon}\textbf{AMD:}\ A(x)B(x) = (A_1 + A_2 x + A_3 x^2)(B_1 x^2 + B_2 x + B_3)$$

$$\textbf{G-SAC:}\ A(x)B(x) = (A_1 + A_2 x + A_3 x^2)(B_3 x^2 + B_1 x + B_2)$$



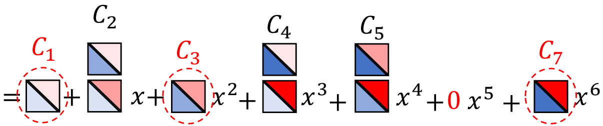
- Approximate recovery polys are $P_r(x) = \sum_{i=1}^{r} C_i x^{i-1} \approx A(x)B(x)$.
- AB equals $C_2 = A_1B_1 + A_2B_2$ plus $C_5 = A_3B_3$.
- $P_r(x)$ is degree-r $\Rightarrow$ Approximate recovery threshold is $R_r = r + 1$.
- $A(x)B(x)$ is degree-4 $\Rightarrow$ Recovery threshold is $R = 5$.

Group-wise SAC, $D = 3$

Goal: Generalize to multi-group SAC in order to recover lower resolutions earlier.
Idea # 2: Inject delays amongst coeffs to eliminate interference.

G-SAC ($D = 2$): $A(x)B(x) = (A_1 + A_2 x + A_3 x^2)(B_3 x^2 + B_1 x + B_2)$

G-SAC ($D = 3$): $A(x)B(x) = (A_1 + A_2 x + 0\, x^2 + A_3 x^3)(B_3 x^3 + 0\, x^2 + B_2 x + B_1)$



- AB equals $C_1 = A_1 B_1$ plus $C_3 = A_2 B_2$ plus $C_7 = A_3 B_3$.
- $\forall r \in [5]$, degree of $P_r(x) = \sum_{i=1}^{r} C_i x^{i-1}$ is $(r-1) \Rightarrow R_r = r$.
- $A(x)B(x)$ is degree-6 $\Rightarrow$ Recovery threshold is $R = 7$.

Prior coding schemes requiring post-decoding

In both, $A(x) = \sum_{k=1}^{K} A_k T_{k-1}(x)$, $B(x) = \sum_{k=1}^{K} B_k T_{k-1}(x)$; $T_i(x)$ is not monomial.

OrthoMatDot [Fahim et al]:

Basis: Orthonormal basis
$\int_{-1}^{1} T_i(x) T_j(x) w(x) dx = \mathbb{I}(i = j)$

Enc: Eval $A(x), B(x)$ at $x \in \{T_N(x) \text{ roots}\}$

Dec: Invert Cheby Vander to get $A(x)B(x)$

Post-dec: Gauss quad

$$AB = \int_{-1}^{1} A(x)B(x)w(x)dx$$
$$= \sum_{k=1}^{K} \frac{2}{K} A(y_k)B(y_k)$$

where $y_k \in \{K \text{ roots of } T_K(x)\}$

(+) Mitigates ill-conditioning issue.

Lagrange [Yu et al]:

Basis: Lagrange basis

$$T_i(x) = \prod_{j \neq i} \frac{(x - y_j)}{(y_i - y_j)}, \text{ for } i \in [K]$$

Enc: Eval at arbitrary
$x \in \mathcal{X}_{\text{Lag}}, \text{s.t. } |\mathcal{X}_{\text{Lag}}| = \text{N}$
$y \in \mathcal{Y}_{\text{Lag}}, \text{s.t. } |\mathcal{Y}_{\text{Lag}}| = \text{K}$

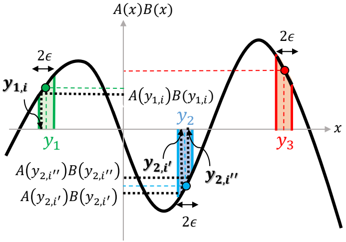
Dec: Invert Vander to get $A(x)B(x)$

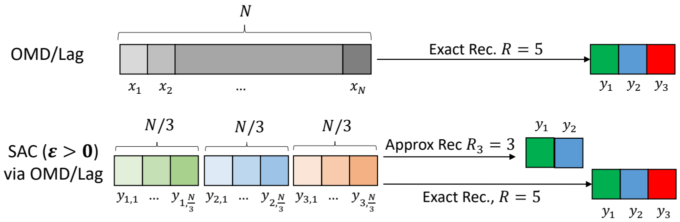
Post-dec: y_k zeros-out cross terms,
$\sum_{k=1}^{K} A(y_k)B(y_k) = AB$

(+) Extends to multi-variate polynomials and security and privacy.

Layer-wise SAC

Goal: Apply SAC to codes with post-decoding, (e.g., OMD, Lag)
Idea: Pick $\{x_n\}_{n=1}^{N}$ used by enc to be ϵ-close (a small perturbation) of $\{y_k\}_{k=1}^{K}$ of dec.



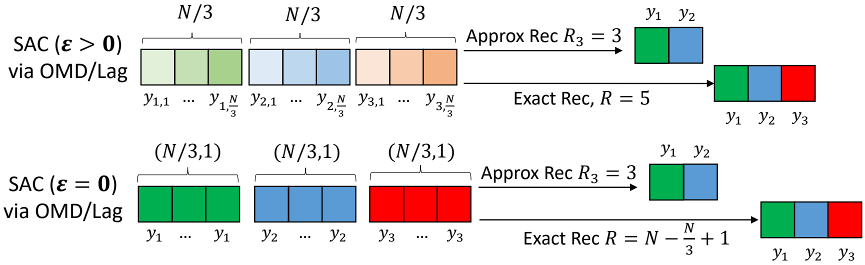


$$AB = \sum_{k=1}^{3} \alpha_k A(y_k)B(y_k) \approx \beta \left(\alpha_1 \frac{A(y_{1,i})B(y_{1,i})}{1} + \alpha_2 \frac{A(y_{2,i'})B(y_{2,i'}) + A(y_{2,i''})B(y_{2,i''})}{2} \right)$$

General: Div workers into K splits. Avg results from each split to improve estimate.

Layer-wise SAC: Hybridize repetition and coded computing

- Like Rep codes, workers in split $k \in [K]$ contribute to $A(y_k)B(y_k)$ recovery.
- Like coded computing, guarantee exact rec only when a few workers report in.
- LSAC($\epsilon = 0$) slightly better estimates, but waits longer for exact recovery.

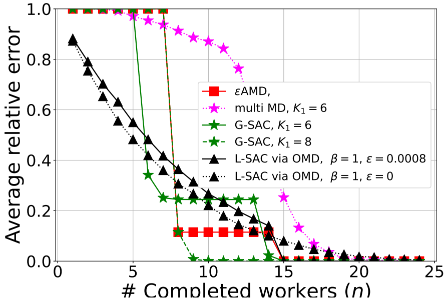


- However, compared to OMD we loose numerical benefits of Cheby Vander dec.

Relative error vs. approximation threshold

Settings:

- $N = 40,\ K = 8,\ A \in \mathbb{R}^{100\times 8000}, B \in \mathbb{R}^{8000\times 100}$, All entries $\sim \mathcal{N}(0,1)$.
- MMD: (30,6) MatDot & (10,2) MatDot.
- 2-GSAC: $K_1 \in \{6, 8\}$, $x \in \{0.15e^{\frac{i2\pi n}{N}}\}_{n=1}^{N}$.
- LSAC-OMD: $\epsilon \in \{\frac{5}{10^4}, 0\}$, $y_{k,i} \in \mu_k^{(8),\text{cheby}} \pm \epsilon$.



Takeaways:

- G-SAC,$K_1 = 8$: similar to ϵAMD upto $n = 8$, improve as $n \uparrow$.
- G-SAC,$K_1 = 6$: early estim. $n = 6$, better estim. $n \geq 14$.
- L-SAC via OMD: contin. improv. since $n = 1$.
- If $\epsilon = 0$, slight. better estim., later exact recovery.

Conclusion and future works

Conclusion:

- G-SAC and L-SAC enable approximation in coded computing, extending approximate procedure of ϵAMD to multiple layers.
- Compared to ϵAMD, SAC achieves better tradeoff between approximate threshold and relative error.

Some possible future works:

- Apply SAC to more practical applications (beyond matrix multiplication) such as training deep neural networks.
- Extend SAC to other coding schemes, such as Polynomial and Product codes.
- Study numerically stability of SAC methods and explore possible numerical stable coding schemes building on SAC.

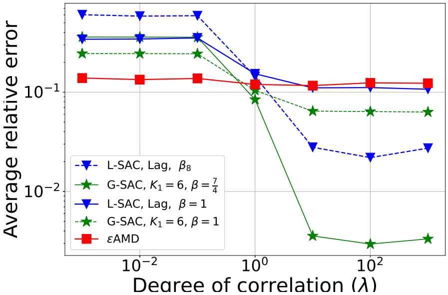
References

- **Full version:** S. Kiani and S. C. Draper, "Successive Approximation for Coded Matrix Multiplication," arXiv:2201.03486.
- N. Ferdinand and S. C. Draper, "Anytime coding for distributed computation," in IEEE Annual Allerton Conf. on Commun., Control, and Comput., 2016.
- S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," IEEE Trans. on Inf. Theory, 2019.
- H. Jeong, A. Devulapalli, V. R. Cadambe, and F. P. Calmon, "ϵ approximate coded matrix multiplication is nearly twice as efficient as exact multiplication," IEEE J. Sel. Areas Inf. Theory, 2021.
- M. Fahim and V. R. Cadambe, "Numerically stable polynomially coded computing," IEEE Trans. on Inf. Theory, 2021.
- Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, and S. A. Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security, and privacy," in Int. Conf. on Artificial Intelligence and Statistics (AISTATS), 2019.

Effect of correlation (Additional)

Settings:

- $n = 8$.
- LSAC-Lag: $\epsilon = 3.33 \times 10^{-2}$, $y_{k,i}$ ϵ-close to k.
- $A_k = \lambda A^{(0)} + A_k^{(1)}$, $B_k = \lambda B^{(0)} + B_k^{(1)}$, $A^{(0)}, \ldots, B_k^{(1)} \sim \mathcal{N}(0,1)$.



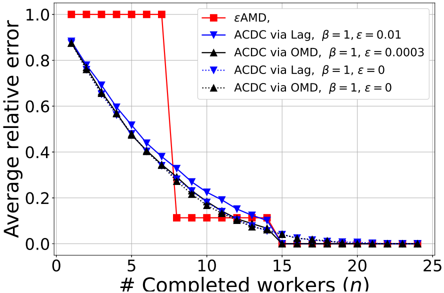
Takeaways:

- 2-GSAC and LSAC-Lag better estim. than ϵAMD if highly correlated (λ large) and parameters set optimally.

Relative Err. vs. Approx. Threshold (Additional)

Settings:

- $N = 24,\ K = 8,\ A, B \sim \mathcal{N}(0,1)$, $A, B : (6 \times 10^2, 9 \times 10^3, 6 \times 10^2)$.
- LSAC-OMD: $\epsilon \in \{3 \times 10^{-4}, 0\}$, $y_{k,i}$ ϵ-close to $\mu_k^{(8),\text{cheby}}$.
- LSAC-Lag: $\epsilon = 10^{-2}$, ϵ-close to k.
- ϵAMD: $x \in \{0.15 e^{\frac{i2\pi n}{N}}\}_{n=1}^{N}$.

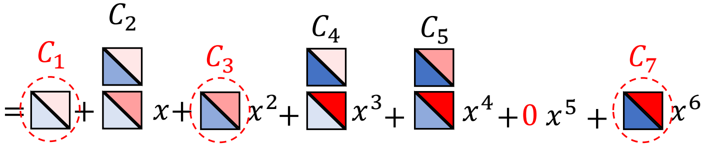


Takeaways:

- LSAC-Lag $\epsilon > 0$: contin improv from $n = 1$. Better than ϵAMD $n < 8$.
- LSAC-OMD $\epsilon > 0$: contin improv from $n = 1$. Better than ϵAMD $n < 8$ or $n > 11$.
- If $\epsilon = 0$, slight. better estim., later exact recovery.

Recalling GSAC & further developments (see papers)

G-SAC ($D = 3$): $A(x)B(x) = (A_1 + A_2 x + 0\,x^2 + A_3 x^3)(B_3 x^3 + 0\,x^2 + B_2 x + B_1)$



- **Reducing interference:** For $n = 2, 4, 5$ can approx higher-order polynomial, reducing "interference" from even higher-order terms (analogus to SINR)
- **Total error & evaluation points:** Total error $=$ (approx. error) $+$ (numerical precision). Selecting evaluation points complex equal-magnitude increases computation but reduces numerical errors (Ramamoorthy & Tang ISIT'21)
- **Avoid worst-case:** Randomly jointly permute the $\{A_k\}_{k=1}^K$ and the $\{B_k\}_{k=1}^K$ to avoid worst-case of largest-norm $A_i B_i$ being recovered last.
- **# groups:** Extension in paper to more groups (> 3)