ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Song song hóa thuật toán giảm chiều dữ liệu với *Singular Value Decomposition*

Học phần: Tính toán song song (MAT3148)

Giảng viên hướng dẫn: TS Nguyễn Hải Vinh

Đặng Thế Anh 23001821
Đỗ Minh Đức 23001864

Hà Nội - 2025

Lời cảm ơn

Chúng em xin gửi lời cảm ơn chân thành nhất đến thầy **TS. Nguyễn Hải Vinh**, người đã tận tình hướng dẫn, chỉ bảo và tạo điều kiện thuận lợi cho chúng em trong suốt quá trình thực hiện bài tập lớn. Những kiến thức và kinh nghiệm quý báu mà Thầy truyền đạt không chỉ giúp chúng em hoàn thành tốt đồ án mà còn là hành trang vững chắc cho con đường học tập và nghiên cứu sau này. Chúng em cũng xin cảm ơn các thầy cô trong Khoa Toán - Cơ - Tin học, Trường Đại học Khoa học Tự nhiên, ĐHQGHN đã trang bị cho chúng em những kiến thức nền tảng vững chắc. Mặc dù đã rất cố gắng, nhưng do giới hạn về thời gian và kiến thức, khó tránh khỏi những thiếu sót. Chúng em rất mong nhận được sự đóng góp ý kiến của Thầy và các bạn để đồ án được hoàn thiện hơn.

Hà Nội, tháng 12 năm 2025

Nhóm sinh viên thực hiện

Tóm tắt nội dung

Trong kỷ nguyên dữ liệu lớn, việc xử lý và phân tích các tập dữ liệu nhiều chiều đặt ra những thách thức to lớn về chi phí tính toán và lưu trữ. Giảm chiều dữ liệu (Dimensionality Reduction) là một bước tiền xử lý quan trọng nhằm giải quyết vấn đề này, trong đó Phân rã giá trị kỳ dị (Singular Value Decomposition - SVD) đóng vai trò nền tảng. Đồ án này tập trung nghiên cứu và triển khai thuật toán SVD song song nhằm tối ưu hóa hiệu năng tính toán trên các hệ thống đa lõi.

Cụ thể, chúng em đã xây dựng thuật toán *Parallel Norm-Reducing Jacobi* để tính toán trị riêng và vector riêng, đồng thời song song hóa các bài toán con cốt lõi như nhân ma trận (Cannon, DNS), chuyển vị ma trận và sắp xếp dữ liệu sử dụng thư viện OpenMP. Hiệu năng của các thuật toán được đánh giá thực nghiệm trên nhiều kích thước dữ liệu khác nhau và so sánh với các thư viện chuẩn như LAPACK.

Kết quả thực nghiệm cho thấy phiên bản song song đạt hệ số tăng tốc (speedup) lên tới 4.6 lần so với phiên bản tuần tự trên các tập dữ liệu lớn. Bên cạnh đó, tính ứng dụng của hệ thống được minh chứng qua bài toán giảm chiều dữ liệu ảnh chữ số viết tay MNIST, giúp giảm không gian đặc trưng từ 784 xuống 50 chiều trong khi vẫn duy trì độ chính xác phân loại xấp xỉ 97% với mô hình Multi-layer Perceptron. Kết quả này khẳng định tiềm năng và hiệu quả của việc áp dụng kỹ thuật tính toán song song trong các bài toán đại số tuyến tính và học máy.

Mục lục

Danh sách hình vẽ

Danh sách bảng

Chương 1. Mở đầu

1.1 Giới thiệu bài toán

Sự phát triển mạnh mẽ của công nghệ thu thập và lưu trữ dữ liệu đã dẫn đến việc tạo ra các tập dữ liệu có số lượng đặc trưng (features) ngày càng lớn. Trong nhiều lĩnh vực như thị giác máy tính, xử lý ngôn ngữ tự nhiên, sinh học phân tử, tài chính hay học máy, mỗi mẫu dữ liệu có thể chứa hàng nghìn, thậm chí hàng triệu thuộc tính. Tuy nhiên, việc làm việc với dữ liệu nhiều chiều đặt ra hàng loạt thách thức. Ví dụ, trong học máy, sự xuất hiện của các đặc trưng không quan trọng hoặc dư thừa có thể làm giảm tốc độ huấn luyện, gây nhiễu và ảnh hưởng tiêu cực đến khả năng dự đoán của mô hình

Qua đó, ta thấy rằng việc tồn tại quá nhiều đặc trưng không liên quan hoặc trùng lặp không chỉ làm tăng chi phí tính toán mà còn dẫn tới hiện tượng *lời nguyền chiều không gian* (curse of dimensionality) [1, 2]. Khi số chiều tăng, dữ liệu trở nên thưa thớt, khoảng cách giữa các điểm mất ý nghĩa, khiến các mô hình học máy khó phát hiện được cấu trúc tiềm ẩn. Điều này làm suy giảm đáng kể hiệu suất của các thuật toán phân loại, hồi quy hay phân cụm.

Do đó, bài toán *giảm chiều dữ liệu* (Dimensionality Reduction) ra đời như một hướng tiếp cận nhằm chuyển dữ liệu từ không gian chiều cao về không gian chiều thấp hơn nhưng vẫn giữ lại những thông tin bản chất nhất. Thay vì sử dụng trực tiếp các đặc trưng gốc, các kỹ thuật giảm chiều xây dựng một biểu diễn cô đọng hơn của dữ liệu, giúp các mô hình học máy trở nên hiệu quả, ổn định và dễ diễn giải [3, 4].

1.2 Các thuật toán giảm chiều dữ liệu

Trong lĩnh vực học máy và khai phá dữ liệu, nhiều thuật toán đã được phát triển nhằm mục tiêu giảm chiều dữ liệu nhưng vẫn giữ lại những thông tin quan trọng nhất. Các kỹ thuật này có thể được chia thành hai nhóm chính: *phương pháp tuyến tính* và *phi tuyến tính* [3, 5].

Phương pháp tuyến tính phổ biến nhất là **PCA** (Principal Component Analysis), vốn hoạt động bằng cách tìm các hướng phương sai lớn nhất trong dữ liệu. Bên cạnh đó, **SVD** (Singular Value Decomposition) đóng vai trò nền tảng toán học của PCA và nhiều thuật toán ma trận khác, cho phép phân rã dữ liệu thành các thành phần trực giao và đặc trưng chính [6]. Trong bài này sẽ tập trung chính vào **phương pháp SVD** nhờ tính chất mạnh mẽ, khả năng diễn giải rõ ràng và ứng dụng rộng rãi trong xử lý dữ liệu nhiều chiều.

Ngoài các phương pháp tuyến tính, các kỹ thuật phi tuyến như **t-SNE** (t-distributed Stochastic Neighbor Embedding) [4], **UMAP** (Uniform Manifold Approximation and Projection) [7] hay **Isomap** cũng được sử dụng rộng rãi để khám phá cấu trúc hình học phức tạp của dữ liệu trong không gian chiều cao. Tuy nhiên, các phương pháp này chủ yếu phục vụ trực quan hóa và không cung cấp biểu diễn toàn cục như SVD.

Do đó, việc nghiên cứu và tối ưu hoá SVD trong bài toán giảm chiều không chỉ mang ý nghĩa lý thuyết mà còn có giá trị thực tiễn trong nhiều ứng dụng như nén dữ liệu, phát hiện mẫu, và xây dựng hệ thống khuyến nghị.

SVD đóng vai trò nền tảng trong nhiều kỹ thuật xử lý tín hiệu và học máy[6].

1.3 Mục tiêu

Mục tiêu của nhóm thực hiện đề tài là tập trung nghiên cứu, thiết kế và tối ưu hóa hiệu suất của thuật toán SVD (Singular Value Decomposition) thông qua việc xây dựng và triển khai chiến lược song song hóa các bài toán con. Đề tài hướng đến việc rút ngắn thời gian thực thi trên các hệ thống phần cứng đa lõi mà vẫn đảm bảo giữ lại các đặc trưng quan trọng của dữ liệu hay nói cách khác là giảm độ mất mát của dữ liệu.

Các mục tiêu cụ thể bao gồm:

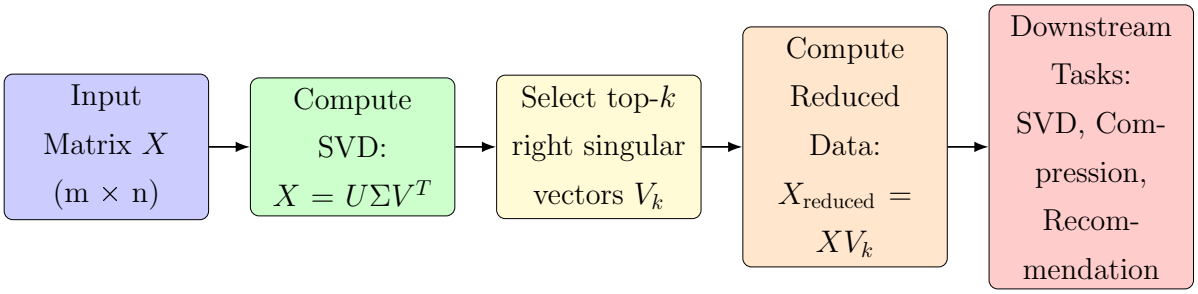
- Xây dựng và triển khai phiên bản song song của thuật toán SVD nhằm tận dụng tối đa tài nguyên tính toán.
- So sánh hiệu suất giữa SVD tuần tự và SVD song song trên các tập dữ liệu lớn.
- Đánh giá tác động của song song hóa đến độ chính xác và chất lượng giảm chiều.
- Ứng dụng SVD song song vào các bài toán trực quan hóa, phân loại hoặc phân cụm dữ liệu lớn.

Chương 2. Cơ sở lí thuyết

2.1 Phương pháp giải pháp bài toán

Mục tiêu của SVD là phân rã ma trận dữ liệu thành tích của ba ma trận trực chuẩn, từ đó biểu diễn dữ liệu trong một hệ cơ sở mới, đồng thời có thể giảm số chiều dữ liệu mà vẫn giữ được cấu trúc quan trọng.

Các bước của quá trình giảm chiều dữ liệu với phương pháp SVD như sau:



Hình 2.1: Các bước giảm chiều dữ liệu với SVD

2.1.1 Giảm chiều dữ liệu

Cho ma trận dữ liệu gốc:

$$X \in \mathbb{R}^{n \times d}.$$

Ta thực hiện phân rã giá trị kỳ dị theo định nghĩa gốc [6]:

$$X = \sum_{i=1}^{r} \sigma_i u_i v_i^T = U\Sigma V^T$$

Trong đó:

- $r = \text{rank}(X)$
- σ_i là giá trị kỳ dị
- u_i là vector riêng bên trái
- v_i là vector riêng bên phải.
- $U \in \mathbb{R}^{n \times n}$: ma trận trực chuẩn chứa các *vector riêng trái*;

- $\Sigma \in \mathbb{R}^{n \times d}$: ma trận đường chéo chứa các giá trị kỳ dị $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$;
- $V \in \mathbb{R}^{d \times d}$: ma trận trực chuẩn chứa các *vector riêng phải*, cũng chính là các hướng cơ sở mới.

Để giảm chiều xuống k chiều, ta chỉ giữ lại k vector cơ sở quan trọng nhất:

$$V_k = [v_1, v_2, \ldots, v_k]$$

Dữ liệu sau khi giảm chiều:

$$X_{\text{reduced}} = XV_k$$

2.2 Nhân ma trận

Phép nhân ma trận là phép toán cơ bản và quan trọng trong đại số tuyến tính, nó là thành phần quan trọng không thể thiếu trong nhiều thuật toán giảm chiều, đặc biệt là trong quá trình tính toán ma trận hiệp phương sai và phân rã ma trận. Ma trận X chỉ có thể nhân với ma trận B nếu $X \in \mathbb{R}^{m \times n}$ và $B \in \mathbb{R}^{n \times p}$, thì ma trận kết quả $C = AB \in \mathbb{R}^{m \times p}$ có phần tử:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Nhận xét: Quan sát ngay ở công thức trên các phần tử trên ma trận C (ô C_{ij}) không phụ thuộc lẫn nhau mà nó chỉ phụ thuộc vào các phần tử trên ma trận X và B mà X và B đều không thay đổi sau mỗi lần tính các phần tử. Ngoài ra quy mỗi thành phần là tổng của tích k cặp A_{ik} và B_{kj} hay nói cách khác là $< A_i, B_j^T >$ (<a, b> là tích vô hướng của a và b). Qua đó có thể thấy phép toán nhân ma trận có khả năng song song hóa cực kì cao do các phép toán tính ma trận tích độc lập với nhau.

Ứng dụng: Nhân ma trận là một phép toán cơ bản trong các thuật toán giảm chiều dữ liệu. Trên cơ sở hình học, phép nhân ma trận được sử dụng để thực hiện các phép biến đổi tuyến tính, chẳng hạn như quay, co giãn hoặc chiếu dữ liệu từ không gian ban đầu sang không gian mới có số chiều nhỏ hơn. Ngoài ra, phép nhân ma trận còn đóng vai trò quan trọng trong việc tính toán các bước trung gian và các thành phần của quá trình phân rã ma trận. **Ví dụ:** Tính $X^T X$, $X_{normalized} V_K$,...

Độ phức tạp và Hiệu suất: Đối với hai ma trận vuông cấp n, độ phức tạp tính toán của thuật toán nhân ma trận thông thường là $O(n^3)$. Khi thực hiện song song trên p bộ xử lý, lý thuyết cho thấy độ phức tạp có thể giảm xuống $O(n^3/p)$ nếu bỏ qua chi phí truyền thông [8]. Tuy nhiên, trong thực tế, hiệu suất còn phụ thuộc vào kiến trúc bộ nhớ và băng thông giao tiếp giữa các luồng hoặc node tính toán.

2.3 Giá trị riêng

Giá trị riêng (Eigenvalues) và vector riêng (Eigenvectors) là các khái niệm cơ bản trong đại số tuyến tính. Với ma trận vuông $X \in \mathbb{R}^{n \times n}$, giá trị riêng λ và vector riêng $v \in \mathbb{R}^n$ thỏa mãn [6]:

$$Xv = \lambda v$$

Các giá trị riêng và vector riêng giúp xác định hướng chính và mức độ biến thiên của dữ liệu, là cơ sở để phân rã SVD.

2.3.1 Phương pháp tính cơ bản:

- Tính giá trị riêng:

$$\begin{aligned} Xv &= \lambda v \\ \Leftrightarrow (X - \lambda I_n)v &= 0 \ (1) \end{aligned}$$

  Ta có: $(X-\lambda I_n)v = 0$ khi $X-\lambda I_n$ là suy biến nên ta có $det(X-\lambda I_n) = 0$ (2)

  Trong đó:

  - $v \in \mathbb{R}^n$ là vector riêng của X
  - (2) được gọi là phương trình đặc trưng của ma trận, giải nó ta sẽ thu được các giá trị riêng của X.

- Tính vector riêng: Ứng với giá trị riêng λ_i là vector riêng v_i, để tính được v_i thì thay giá trị riêng λ_i vào (1) thì ta sẽ tính được vector riêng v_i ứng với λ_i.

Nhận xét: Phương thức tính này tìm ra được các giá trị riêng chính xác bằng cách giải phương trình. Tuy nhiên trong hệ thống máy tính việc giải (2) tốn quá nhiều chi phí, phải tính tuần tự và dính đến sai số cao nên khó song song hóa. Do vậy nhóm không chọn sử dụng phương pháp này đề thực hiện tính giá trị riêng cho bài toán này. Sẽ chuyển sang phương phương pháp tính toán phù hợp hơn.

2.3.2 Phương pháp Giảm chuẩn Jacobi (Norm-Reducing Jacobi Method)

Phương pháp Giảm chuẩn Jacobi là một biến thể mở rộng của phương pháp Jacobi cổ điển, được thiết kế để giải quyết bài toán trị riêng cho các ma trận tổng quát (ma trận không đối xứng hoặc ma trận phức).

Mục tiêu Mục tiêu chính của thuật toán là cực tiểu hóa chuẩn Frobenius của các phần tử ngoài đường chéo, qua đó giảm "độ lệch chuẩn"(departure from normality) của ma trận, đưa ma trận về dạng gần với ma trận chuẩn (normal matrix) để có thể chéo hóa bằng các phép biến đổi đơn vị.

Nguyên lý Phương pháp *Giảm chuẩn Jacobi* (*Norm-Reducing Jacobi Method*) được xây dựng dựa trên ý tưởng thực hiện một chuỗi các **phép biến đổi tương tự** (*similarity transformations*). Tại bước lặp thứ k, ma trận M_k được cập nhật theo công thức:

$$M_{k+1} = T_k^{-1} M_k T_k, \tag{2.1}$$

trong đó T_k là ma trận biến đổi khả nghịch được lựa chọn sao cho:

$$\|M_{k+1}\|_F \leq \|M_k\|_F. \tag{2.2}$$

Theo Eberlein (1987) và Shroff (1989), dãy ma trận $\{M_k\}$ sẽ hội tụ về dạng ma trận gần chéo (quasi-diagonal), với các phần tử trên đường chéo xấp xỉ các giá trị riêng của ma trận ban đầu.

Các phép biến đổi T_k thường là sự kết hợp của các loại sau:

- **Phép quay đơn vị (Unitary Rotation):** Tác động lên mặt phẳng (p, q) nhằm triệt tiêu phần tử m_{qp} hoặc m_{pq}.
- **Phép cắt (Shear Transformation):** Là phép biến đổi không đơn vị, đóng vai trò quan trọng trong việc giảm độ lệch chuẩn của ma trận, tức là giảm chuẩn của ma trận giao hoán tử $C = MM^* - M^*M$ (với M^* là ma trận chuyển vị liên hợp).

- **Phép biến đổi chéo (Diagonal Transformation):** Tác động lên các phần tử đường chéo để hỗ trợ giảm chuẩn Frobenius mà không làm thay đổi phổ của ma trận.

Các phép biến đổi này có thể được kết hợp thành một **phép quay tổng hợp (composite rotation)**, nhằm bảo đảm rằng mỗi bước lặp đều làm giảm chuẩn Frobenius của ma trận. Nhờ tính chất này, phương pháp Giảm chuẩn Jacobi đạt được **độ hội tụ bậc hai** khi ma trận là khả chéo (diagonalizable), đồng thời có thể được **song song hóa hiệu quả** trên các kiến trúc đa lõi hoặc GPU, với độ phức tạp đạt $O(n \log^2 n)$ khi sử dụng $n^2/4$ bộ xử lý [9].

Nhận xét Phương pháp Giảm chuẩn Jacobi có thể được xem là một mở rộng tự nhiên của phương pháp Jacobi cổ điển sang lớp ma trận tổng quát. Ưu điểm nổi bật của phương pháp là:

- Áp dụng được cho **ma trận không đối xứng hoặc phức**.
- Duy trì **tính ổn định số học cao**, do các phép biến đổi được xây dựng sao cho không làm tăng sai số làm tròn và vẫn bảo toàn chuẩn Frobenius của ma trận.
- Có khả năng **song song hóa mạnh**, nhờ các phép biến đổi trên những cặp chỉ số không giao nhau có thể được thực hiện độc lập.

Tuy nhiên, phương pháp này cũng tồn tại một số **hạn chế**:

- Chi phí tính toán cho mỗi phép biến đổi cao hơn so với Jacobi cổ điển, do phải xử lý cả phần thực và phần ảo của ma trận, cũng như cần duy trì điều kiện giảm chuẩn toàn cục.
- Số bước lặp cần thiết có thể lớn đối với ma trận kích thước lớn hoặc có phổ giá trị riêng gần nhau.
- Việc cài đặt thuật toán hiệu quả đòi hỏi phải **tối ưu hoá truy cập bộ nhớ** và **phân phối dữ liệu hợp lý** khi triển khai trên các hệ thống song song.

Tổng hợp lại, phương pháp Giảm chuẩn Jacobi cung cấp một hướng tiếp cận **tổng quát, ổn định và có thể song song hóa tốt**. Tuy nhiên, trong phạm vi bài toán cụ thể này, chúng ta sẽ áp dụng một phiên bản tối ưu hơn cho dữ liệu thực đối xứng.

2.4 Đánh giá hiệu suất

Trong các thuật toán song song như giảm chiều hoặc nhân ma trận, việc đo lường hiệu suất giúp đánh giá mức độ tận dụng phần cứng.

2.4.1 Mức độ tăng tốc và hiệu năng

- **Độ tăng tốc (Speedup, S)** đo mức cải thiện thời gian chạy so với tuần tự:

$$S = \frac{T_{seq}}{T_{par}}$$

  với T_{seq} là thời gian tuần tự, T_{par} là thời gian song song trên p đơn vị xử lý.

- **Hiệu suất (Efficiency, E)** biểu thị đóng góp trung bình của mỗi đơn vị xử lý:

$$E = \frac{S}{p}, \quad E \leq 1$$

2.4.2 Định luật Amdahl

Độ tăng tốc lý thuyết khi chỉ một phần chương trình có thể song song hóa:

$$S(P) = \frac{1}{1 - b + \frac{b}{s}}$$

với b là tỷ lệ phần có thể song song hóa, s là hệ số tăng tốc phần song song. Tổng quát cho nhiều phần:

$$S(P) = \frac{1}{\sum_i \frac{b_i}{s_i}}.$$

2.4.3 Định lý Brent

Ước lượng thời gian chạy song song khi số đơn vị xử lý bị giới hạn:

$$T_{par,R}(P) = O\left(\frac{W}{p} + T_{par,M}(P)\right)$$

với W là tổng số thao tác, $T_{par,M}(P)$ là thời gian chạy trên máy đủ nhiều đơn vị xử lý, p là số đơn vị thực tế.

2.4.4 Thời gian giao tiếp

Trong hệ thống phân tán, thời gian giao tiếp:

$$t_{comm} = t_s + t_d, \quad t_d = m\, t_w$$

với t_s là thời gian khởi tạo, m số từ dữ liệu, t_w thời gian truyền một từ. Băng thông kênh là $1/t_w$.

Tóm lại Các công thức trên giúp đánh giá tốc độ, hiệu suất và chi phí giao tiếp trong các thuật toán song song, đồng thời dự đoán giới hạn tăng tốc tối đa khi triển khai trên CPU đa lõi hoặc hệ thống phân tán.

Chương 3. Song song hóa thuật toán giảm chiều dữ liệu với SVD

3.1 Nhân ma trận

Nhân ma trận là một phép toán nền tảng trong nhiều ứng dụng tính toán khoa học, mô phỏng số và học máy. Các thuật toán song song trình bày dưới đây được tham khảo và tổng hợp từ các tài liệu giảng dạy về tính toán song song [8]. Với hai ma trận

$$A \in \mathbb{R}^{m \times q}, \quad B \in \mathbb{R}^{q \times n},$$

ma trận kết quả $C \in \mathbb{R}^{m \times n}$ được xác định bởi:

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}.$$

Thuật toán tuần tự thực hiện phép nhân ba vòng lặp lồng nhau với độ phức tạp $O(n^3)$. Điều này khiến việc thiết kế các thuật toán song song nhằm giảm thời gian thực thi trở thành một yêu cầu quan trọng trong các hệ thống tính toán hiệu năng cao (HPC).

3.1.1 Phân chia dữ liệu

Hiệu năng của một thuật toán song song phụ thuộc mạnh vào cách dữ liệu được phân phối lên các bộ xử lý. Ba phương pháp chính được xem xét trong luận văn này là phân chia một chiều (1D), hai chiều (2D) và ba chiều (3D). Mỗi phương pháp thể hiện mức độ cân bằng khác nhau giữa chi phí giao tiếp và khối lượng tính toán tại mỗi bộ xử lý.

Phân chia một chiều (1D Partitioning)

Trong phân chia 1D, ma trận A được chia theo hàng (hoặc cột) cho các bộ xử lý. Mỗi bộ xử lý giữ một phần của ma trận A và phải nhận toàn bộ

ma trận B thông qua truyền thông.

Phương pháp này đơn giản nhưng chi phí giao tiếp lớn, đặc biệt khi số bộ xử lý tăng cao. Thuật toán nhân ma trận theo mô hình 1D đạt độ phức tạp tính toán:

$$T_{\text{comp}} = O\left(\frac{n^3}{p}\right),$$

trong khi chi phí truyền thông có độ lớn:

$$T_{\text{comm}} = O(n \log p).$$

Phân chia hai chiều (2D Block Partitioning)

Trong phân chia 2D, ma trận A và B được chia thành các block kích thước đều nhau và phân phối lên lưới xử lý dạng $p = q^2$. Mỗi bộ xử lý lưu một block của A, B và tương ứng tạo ra một block của C. Cách phân chia này giảm đáng kể lượng dữ liệu cần trao đổi, do mỗi bộ xử lý chỉ làm việc với block lân cận, nhờ đó mang lại hiệu năng tốt hơn so với phân chia 1D.

Phân chia ba chiều (3D Partitioning)

Phân chia 3D mở rộng cách tổ chức dữ liệu từ 2D sang cấu trúc khối ba chiều. Dữ liệu ma trận được nhân bản một phần theo chiều thứ ba, giúp giảm khối lượng tính toán tại mỗi bộ xử lý và đồng thời hạn chế xung đột giao tiếp. Đây là cơ sở của thuật toán DNS được trình bày ở phần sau.

3.1.2 Thuật toán Broadcast 1D

Thuật toán Broadcast 1D nhân ma trận dựa trên mô hình phân chia theo hàng. Mỗi bộ xử lý lưu một dải hàng của ma trận A, trong khi ma trận B được broadcast tới tất cả các bộ xử lý.

Quy trình thực hiện gồm:

- Chia các hàng của A cho các bộ xử lý.
- Broadcast toàn bộ ma trận B.
- Mỗi bộ xử lý thực hiện phép nhân phần dữ liệu được giao.

Độ phức tạp song song của thuật toán:

$$T(n,p) = O\left(\frac{n^3}{p} + n\log p\right),$$

trong đó thành phần thứ hai thể hiện chi phí truyền thông của thao tác broadcast.

3.1.3 Thuật toán Cannon

Thuật toán Cannon [10] triển khai trên lưới xử lý hai chiều. Dữ liệu ban đầu được căn chỉnh (skew) bằng cách dịch vòng các block của ma trận A theo hàng và các block của ma trận B theo cột. Cụ thể, hàng thứ i của ma trận A được dịch vòng sang trái i vị trí, và cột thứ j của ma trận B được dịch vòng lên trên j vị trí. Sau quá trình căn chỉnh, thuật toán thực hiện q vòng lặp, mỗi vòng gồm:

- Nhân block A_{ij} và B_{ij} cục bộ tại mỗi bộ xử lý.
- Dịch block A sang trái một ô (theo chu kỳ).
- Dịch block B lên trên một ô (theo chu kỳ).

Thuật toán đạt:

$$T_{\text{comp}} = O\left(\frac{n^3}{p}\right), \qquad T_{\text{comm}} = O\left(\frac{n^2}{\sqrt{p}}\right),$$

cho thấy mức độ cân bằng tốt giữa giao tiếp và tính toán, đặc biệt hiệu quả khi số bộ xử lý có thể sắp xếp thành lưới vuông.

3.1.4 Thuật toán DNS (Dekel–Nassimi–Sahni)

Thuật toán DNS [11] sử dụng phân chia dữ liệu ba chiều, trong đó mỗi chiều tương ứng với một trong ba chỉ số i, j, k của phép nhân ma trận. Ma trận được ánh xạ vào một lưới khối lập phương các bộ xử lý, cho phép mỗi bộ xử lý chỉ thực hiện một phần rất nhỏ của phép nhân:

$$A_{ik} \cdot B_{kj}.$$

Dữ liệu được broadcast theo hai chiều và kết quả được reduce theo chiều còn lại để thu được phần tử tương ứng của ma trận C.

Độ phức tạp song song:

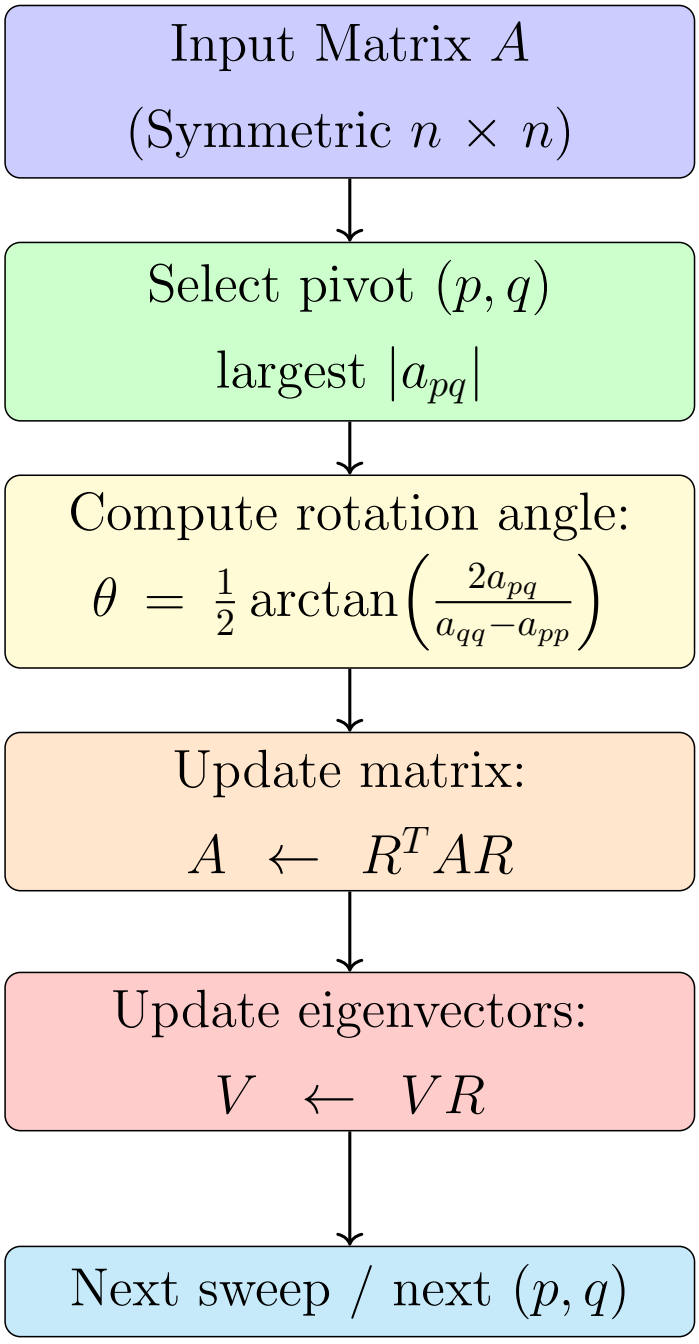
$$T(n,p) = O\left(\frac{n^3}{p^3} + \frac{n^2}{p^2}\log p + \log p\right).$$

Mặc dù thuật toán mang lại hiệu năng lý thuyết rất cao, cấu trúc giao tiếp phức tạp và yêu cầu phân chia dữ liệu ba chiều khiến việc triển khai gặp nhiều thách thức trên các hệ thống thực tế.

3.1.5 Đánh giá chung

- Phân chia 1D và thuật toán Broadcast phù hợp cho các hệ thống nhỏ hoặc bài toán không yêu cầu mở rộng mạnh.
- Phân chia 2D và thuật toán Cannon mang lại hiệu năng cao, cân bằng tốt giữa chi phí tính toán và truyền thông.
- Phân chia 3D và thuật toán DNS cung cấp hiệu năng lý thuyết tối ưu, song chi phí triển khai và đồng bộ dữ liệu lớn.

3.2 Song song của biến thể phương pháp Jacobi cho ma trận đối xứng thực



Hình 3.1: Pipeline mô tả tính giá trị riêng vector riêng

Thuật toán 1 Thuật toán Song song Jacobi cho Ma trận Hiệp phương sai

Đầu vào: Ma trận dữ liệu $X \in \mathbb{R}^{d \times d}$, ngưỡng hội tụ ϵ, số vòng lặp tối đa K_{max}

Đầu ra: Ma trận giá trị riêng Λ, Ma trận vector riêng V

$V \leftarrow I_d$ ▷ Khởi tạo ma trận đơn vị
$idx \leftarrow [0, 1, \ldots, d-1]$ ▷ Khởi tạo mảng chỉ số
for $k = 1$ **to** K_{max} **do**
    $converged \leftarrow$ True
    for $step = 1$ **to** d **do** ▷ Thực hiện một vòng quét (Sweep)
        Parallel Region: ▷ Phân chia công việc cho các luồng
        for each pair (p, q) generated from idx **do**
            PIVOT$((p, q))$ ▷ Xác định cặp phần tử cần khử
            if $|X_{pq}| > \epsilon$ **then**
                $converged \leftarrow$ False
                $(c, s) \leftarrow$ COMPUTE_ROTATION(X_{pp}, X_{qq}, X_{pq})
                UPDATE_MATRX_X(X, p, q, c, s) ▷ Cập nhật hàng/cột của X
                UPDATE_MATRIX_V(V, p, q, c, s) ▷ Cập nhật cột của V
            end if
        end for
        End Parallel
        Barrier Synchronization ▷ Đồng bộ hóa các luồng
        ROTATE_INDICES(idx) ▷ Xoay vòng chỉ số Round-Robin
    end for
    if $converged$ is True **then**
        break ▷ Thuật toán hội tụ
    end if
end for
$\Lambda \leftarrow \text{diag}(X)$
return Λ, V

Trong phạm vi bài toán này, dữ liệu đầu vào là ma trận thực $A \in \mathbb{R}^{m \times d}$. Mục tiêu là tính giá trị riêng và vector riêng của ma trận hiệp phương sai (hoặc ma trận Gram) $X = A^T A$.

Vì X là ma trận đối xứng thực ($X = X^T$), nó thỏa mãn tính chất của ma trận chuẩn (normal matrix). Do đó, các phép biến đổi "Cắt"(Shear) và "Chéo"(Diagonal) trong phương pháp giảm chuẩn tổng quát của Eberlein trở nên không cần thiết (chúng trở thành các phép đồng nhất). Thuật toán lúc này quy về Phương pháp Jacobi cổ điển, chỉ sử dụng các phép quay trực giao để khử các phần tử ngoài đường chéo, nhưng được tổ chức theo chiến lược song song [9].

3.2.1 Mục tiêu và giả thiết

- **Đầu vào:** Ma trận dữ liệu $A \in \mathbb{R}^{m \times d}$, từ đó tính $X = A^T A \in \mathbb{R}^{d \times d}$.
- **Mục tiêu:** Tìm ma trận trực giao Q và ma trận đường chéo Λ sao cho $Q^T X Q = \Lambda$. Các phần tử trên đường chéo của Λ là các giá trị riêng, và các cột của Q là các vector riêng tương ứng.

3.2.2 Chiến lược song song hóa

Để tận dụng kiến trúc đa xử lý, thuật toán không chọn phần tử lớn nhất để khử (như phương pháp cổ điển tuần tự) mà sử dụng chiến lược chia cặp và xoay vòng:

- **Chiến lược chia cặp (Pairing Strategy):** Tại mỗi bước, d hàng/cột của ma trận X được chia thành $d/2$ cặp chỉ số (p, q) rời nhau. Điều này cho phép $d/2$ luồng tính toán thực hiện khử đồng thời các phần tử X_{pq} mà không tranh chấp dữ liệu.
- **Xoay vòng chỉ số (Round-Robin):** Sau mỗi bước tính toán, các chỉ số được hoán vị theo quy tắc Round-Robin (giữ nguyên chỉ số đầu, dịch vòng tròn các chỉ số còn lại). Một chu kỳ đầy đủ (sweep) gồm $d-1$ bước (với d chẵn) đảm bảo mọi cặp phần tử ngoài đường chéo đều được xử lý ít nhất một lần.

3.2.3 Quy trình tính toán tại mỗi bước

Tại mỗi bước k, với mỗi cặp (p, q) được phân công cho một luồng xử lý, các thao tác sau được thực hiện:

Tính tham số quay Mục tiêu là tìm phép quay Jacobi $J_{p,q}$ để triệt tiêu phần tử X_{pq}. Các tham số τ, t, c, s được tính như sau:

$$\tau = \frac{X_{qq} - X_{pp}}{2X_{pq}}$$

$$t = \frac{\text{sgn}(\tau)}{|\tau| + \sqrt{1 + \tau^2}}$$

$$c = \frac{1}{\sqrt{1 + t^2}}, \qquad s = t \cdot c$$

Trong đó $c = \cos\theta$ và $s = \sin\theta$.

Cập nhật ma trận X Áp dụng phép biến đổi tương đồng $X^{(k+1)} = J_{p,q}^T X^{(k)} J_{p,q}$.

- **Cập nhật phần tử đường chéo (p, q):**

$$\begin{aligned} X'_{pp} &= c^2 X_{pp} + s^2 X_{qq} - 2cs\, X_{pq} \\ X'_{qq} &= s^2 X_{pp} + c^2 X_{qq} + 2cs\, X_{pq} \\ X'_{pq} &= X'_{qp} = 0 \quad \text{(Gán cứng để tránh sai số)} \end{aligned}$$

- **Cập nhật phần tử ngoài đường chéo $(j \neq p, q)$:** Các hàng và cột liên quan đến p và q được cập nhật song song:

$$\begin{aligned} X'_{pj} &= c\, X_{pj} - s\, X_{qj} \\ X'_{qj} &= s\, X_{pj} + c\, X_{qj} \end{aligned}$$

Do tính đối xứng, ta cập nhật đồng thời: $X'_{jp} = X'_{pj}$ và $X'_{jq} = X'_{qj}$.

Cập nhật ma trận vector riêng V: Ma trận V (khởi tạo là ma trận đơn vị I) tích lũy các phép quay để tạo thành ma trận vector riêng cuối cùng:

$$V'_{:,p} = c\,V_{:,p} - s\,V_{:,q}$$
$$V'_{:,q} = s\,V_{:,p} + c\,V_{:,q}$$

3.2.4 Nhận xét

Biến thể này kết hợp được ưu điểm về tốc độ hội tụ bậc hai của phương pháp Jacobi cổ điển trên ma trận đối xứng với khả năng mở rộng hiệu năng của chiến lược song song. Việc loại bỏ các phép biến đổi phức tạp (Shear) giúp giảm đáng kể chi phí tính toán trên mỗi bước lặp so với thuật toán Norm-Reducing tổng quát.

3.3 Song song hóa các phần còn lại

3.3.1 Song song hóa tính giá trị kỳ dị

Sau khi tính được ma trận Λ chứa d giá trị riêng, bước tiếp theo là xác định d giá trị kỳ dị tương ứng. Để tận dụng khả năng tính toán song song, chúng ta chia tổng số phép toán cho mỗi luồng (thread) như sau:

$$\text{số phép toán mỗi luồng} = \frac{d}{p},$$

trong đó p là số lượng luồng được sử dụng. Mỗi luồng sẽ chịu trách nhiệm tính giá trị kỳ dị của một tập con các giá trị riêng, đảm bảo rằng tất cả các giá trị kỳ dị được tính toán đồng thời và độc lập, từ đó tăng tốc độ tính toán tổng thể. vậy độ phức tạp thuật toán của phần này là $T_1 = o(\frac{n}{p})$.

3.3.2 Song song hóa sắp xếp theo giá trị kỳ dị

Giả sử chúng ta có k giá trị riêng và các vector riêng tương ứng. Để sắp xếp các giá trị kỳ dị theo thứ tự giảm dần (hoặc tăng dần) và đồng thời sắp xếp lại các cột của ma trận vector riêng, chúng ta áp dụng song song hóa như sau:

- Chia dữ liệu luồng: Mỗi luồng nhận một tập con các cặp giá trị riêng,

vector riêng cần sắp xếp. Nếu có p luồng, mỗi luồng xử lý khoảng $\frac{k}{p}$ cặp.

- Sắp xếp trong luồng: Mỗi luồng sắp xếp các giá trị kì dị của nó độc lập, đồng thời sắp xếp các vector riêng tương ứng.
- Gộp kết quả: Sau khi tất cả luồng hoàn tất, các kết quả được merge để thu được thứ tự cuối cùng. Merge có thể thực hiện theo kiểu merge-sort song song, đảm bảo kết quả chính xác.

Do vậy, tổng số phép toán của thuật toán song song có thể ước lượng là:

$$T_2 = O\Big(\frac{n}{p}\log(n) - \log(p) + np\Big),$$

trong đó n là số phần tử (giá trị kì dị) và p là số lượng luồng.

3.3.3 Chuyển vị ma trận

Để tăng hiệu quả tính toán, Ấp dụng phương pháp blocked parallel transpose kết hợp với OpenMP. Cụ thể, ma trận $A \in \mathbb{R}^{m\times n}$ được chia thành các block kích thước $B \times B$, và mỗi block được xử lý độc lập bởi một luồng:

$$\begin{aligned} B[jj][ii] &= A[ii][jj], \\ ii &= i, \ldots, \min(i + B - 1, M - 1), \\ jj &= j, \ldots, \min(j + B - 1, N - 1), \end{aligned}$$

trong đó (i, j) là tọa độ bắt đầu của block và B là kích thước block.

Các block được phân phối cho p luồng sử dụng chỉ thị OpenMP với ‘collapse(2)‘ để song song hóa cả hai vòng lặp bên ngoài. Kỹ thuật này vừa tận dụng tối đa khả năng đa luồng, vừa giảm thiểu cache-miss nhờ truy cập dữ liệu liên tiếp trong mỗi block.

Cách tiếp cận này đảm bảo rằng các phép toán transpose trên từng block là độc lập, tránh hiện tượng race condition, đồng thời cung cấp tốc độ tính toán cao khi làm việc với ma trận lớn.

Chương 4. Thực nghiệm và đánh giá

4.1 Môi trường thực nghiệm

Các thí nghiệm được thực hiện trên một hệ thống di động sử dụng bộ vi xử lý AMD Ryzen 5 6600HS (Zen 3+, mã hiệu Rembrandt) với thông số kỹ thuật như sau:

- Số nhân / luồng: 6 nhân vật lý, 12 luồng xử lý.
- Tần số cơ bản / Turbo: 3.3 GHz, lên đến 4.5 GHz.
- Bộ nhớ cache: L1 64 KB/cấp nhân, L2 512 KB/cấp nhân, L3 16 MB chia sẻ.
- Bộ nhớ: DDR5, tốc độ hiệu dụng 4800 MT/s, kênh đôi, băng thông 76.8 GB/s.

Physical

Socket:	AMD Socket FP7
Foundry:	TSMC
Process Size:	6 nm
Die Size:	208 mm²
Package:	FP7, FP7r2
tJMax:	95°C

Processor

Market:	Mobile
Production Status:	Active
Release Date:	Jan 2022
Part#:	100-000000546 100-000000562

Performance

Frequency:	3.3 GHz
Turbo Clock:	up to 4.5 GHz
Base Clock:	100 MHz
Multiplier:	33.0x
Multiplier Unlocked:	No
TDP:	35 W

Architecture

Codename:	Rembrandt
Generation:	Ryzen 5 (Zen 3+ (Rembrandt))
Memory Support:	DDR5
LPDDR5 Speed:	6400 MT/s
Rated Speed:	4800 MT/s
Memory Bus:	Dual-channel
Memory Bandwidth:	76.8 GB/s
ECC Memory:	No
PCI-Express:	Gen 4, 20 Lanes (CPU only)

Cache

Cache L1:	64 KB (per core)
Cache L2:	512 KB (per core)
Cache L3:	16 MB (shared)

Core Config

# of Cores:	6
# of Threads:	12
SMP # CPUs:	1
Integrated Graphics:	Radeon 660M

Features

- MMX • SSE • SSE2
- SSE3 • SSSE3 • SSE4A
- SSE4.1 • SSE4.2 • AES
- AVX • AVX2 • BMI1
- BMI2 • SHA • F16C
- FMA3 • AMD64 • EVP
- AMD-V • SMAP • SMEP
- SMT • Precision Boost 2
- XFR 2

Notes

Graphics engine boost clock: 1900MHz

Hình 4.1: Thông số cpu

4.2 Bộ dữ liệu sử dụng

Để đánh giá hiệu năng của các thuật toán, sử dụng bộ dữ liệu gồm các ma trận được sinh ngẫu nhiên với kích thước và đặc điểm đa dạng. Quá trình sinh dữ liệu ngẫu nhiên với các kích thước ma trận khác nhau.

Đặc điểm dữ liệu:

- **Giá trị:** Các phần tử trong ma trận là số thực ngẫu nhiên (float) nằm trong khoảng $[0, 1000]$.
- **Định dạng lưu trữ:** Mỗi ma trận được lưu trong một file văn bản (.txt). Dòng đầu tiên chứa hai số nguyên m và n tương ứng với số hàng và số cột. Các dòng tiếp theo chứa giá trị của ma trận.

Các kích thước thử nghiệm: Tạo ra các bộ dữ liệu với kích thước tăng dần để kiểm tra khả năng mở rộng (scalability) của thuật toán. Các kích thước cụ thể bao gồm:

Bảng 4.1: Danh sách các kích thước ma trận sử dụng trong thực nghiệm

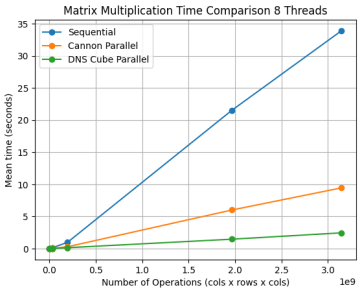
STT	**Kích thước** $(m \times n)$	**Số phần tử**	**Mô tả**
1	200×128	25,600	Ma trận nhỏ
2	490×64	31,360	Ma trận nhỏ, hẹp
3	500×256	128,000	Ma trận trung bình
4	1000×178	178,000	Ma trận trung bình, dài
5	750×512	384,000	Ma trận trung bình lớn
6	3333×768	2,559,744	Ma trận lớn
7	3000×1024	3,072,000	Ma trận rất lớn

Việc lựa chọn các kích thước này giúp đánh giá hiệu năng thuật toán trong nhiều kịch bản khác nhau: từ ma trận nhỏ (vài chục nghìn phần tử) đến ma trận lớn (hàng triệu phần tử), và các ma trận có tỉ lệ hàng/cột khác nhau.

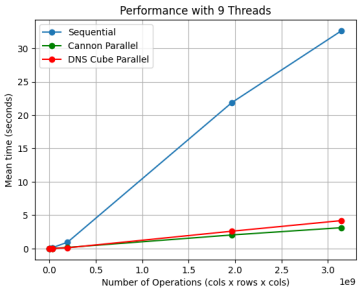
4.3 Đánh giá kết quả

4.3.1 Thuật toán nhân ma trận

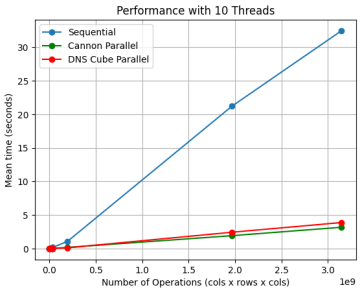
Thống kê các thuật toán sau khi chạy với các luồng khác nhau



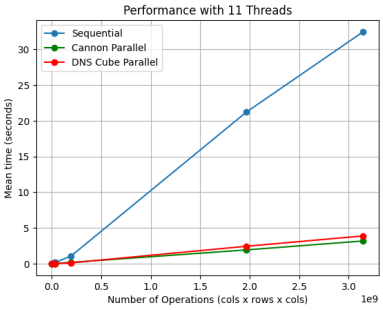
Hình 4.2: các phương pháp nhân ma trận dùng 8 thread



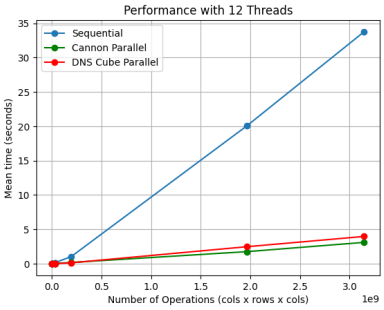
Hình 4.3: các phương pháp nhân ma trận dùng 9 thread



Hình 4.4: các phương pháp nhân ma trận dùng 10 thread



Hình 4.5: các phương pháp nhân ma trận dùng 11 thread



Hình 4.6: các phương pháp nhân ma trận dùng 12 thread

Nhận xét: Về tổng quan, thuật toán nhân ma trận cơ bản có thời gian thực thi lâu hơn nhiều so với các thuật toán song song đã được tối ưu hóa (như Cannon, DNS).

Phân tích chi tiết khi thay đổi số luồng từ 8 đến 12:

- **Đối với thuật toán DNS:** Thời gian thực thi không có sự thay đổi đáng kể khi tăng số luồng lên 12. Nguyên nhân là do thuật toán DNS được thiết kế để hoạt động trên mạng lưới vi xử lý lập phương ($p \times p \times p$). Tại 8 luồng (2^3), thuật toán hoạt động tối ưu. Tuy nhiên, 12 luồng không phải là lập phương hoàn hảo, dẫn đến việc không tận dụng được thêm

tài nguyên tính toán cho cấu trúc đặc thù của DNS.

- **Đối với thuật toán Cannon:** Tại mốc 9 luồng, biểu đồ cho thấy thời gian chạy giảm rõ rệt (nhanh hơn) so với mốc 8 luồng. Điều này được giải thích bởi thuật toán Cannon yêu cầu mạng lưới vi xử lý hình vuông ($p \times p$). Với 9 luồng, ta thiết lập được lưới 3×3 sử dụng triệt để tài nguyên. Trong khi đó, với 8 luồng, thuật toán chỉ có thể sử dụng lưới 2×2 (4 luồng), gây lãng phí tài nguyên và hiệu năng thấp hơn.

Đánh giá về thời gian thực thi song song (T_{par}) dựa trên độ phức tạp thuật toán:

Để giải thích sự thay đổi hiệu năng, ta xem xét độ phức tạp tính toán lý thuyết $O(\frac{n^3}{p_{used}})$ với p_{used} là số luồng thực tế được sử dụng:

- **Tại $p = 8$ luồng:**
  - **DNS:** Do $8 = 2^3$, thuật toán tận dụng toàn bộ luồng ($p_{used} = 8$). Độ phức tạp đạt mức tối ưu $O(\frac{n^3}{8})$.
  - **Cannon:** Do 8 không phải số chính phương, thuật toán chỉ sử dụng lưới 2×2 ($p_{used} = 4$). Độ phức tạp chỉ là $O(\frac{n^3}{4})$.
  - $\Rightarrow$ DNS nhanh hơn Cannon do $\frac{n^3}{8} < \frac{n^3}{4}$.
- **Tại $p = 9$ luồng:**
  - **Cannon:** Do $9 = 3^2$, thuật toán tận dụng toàn bộ luồng ($p_{used} = 9$). Độ phức tạp giảm xuống $O(\frac{n^3}{9})$.
  - **DNS:** Do 9 không phải lập phương, thuật toán vẫn chỉ dùng lưới $2 \times 2 \times 2$ ($p_{used} = 8$). Độ phức tạp giữ nguyên $O(\frac{n^3}{8})$.
  - $\Rightarrow$ Cannon nhanh hơn DNS do $\frac{n^3}{9} < \frac{n^3}{8}$.
- **Tại $p = 12$ luồng:**
  - **Cannon:** Vẫn chỉ dùng lưới 3×3 ($p_{used} = 9$) $\rightarrow O(\frac{n^3}{9})$.
  - **DNS:** Vẫn chỉ dùng lưới 2^3 ($p_{used} = 8$) $\rightarrow O(\frac{n^3}{8})$.

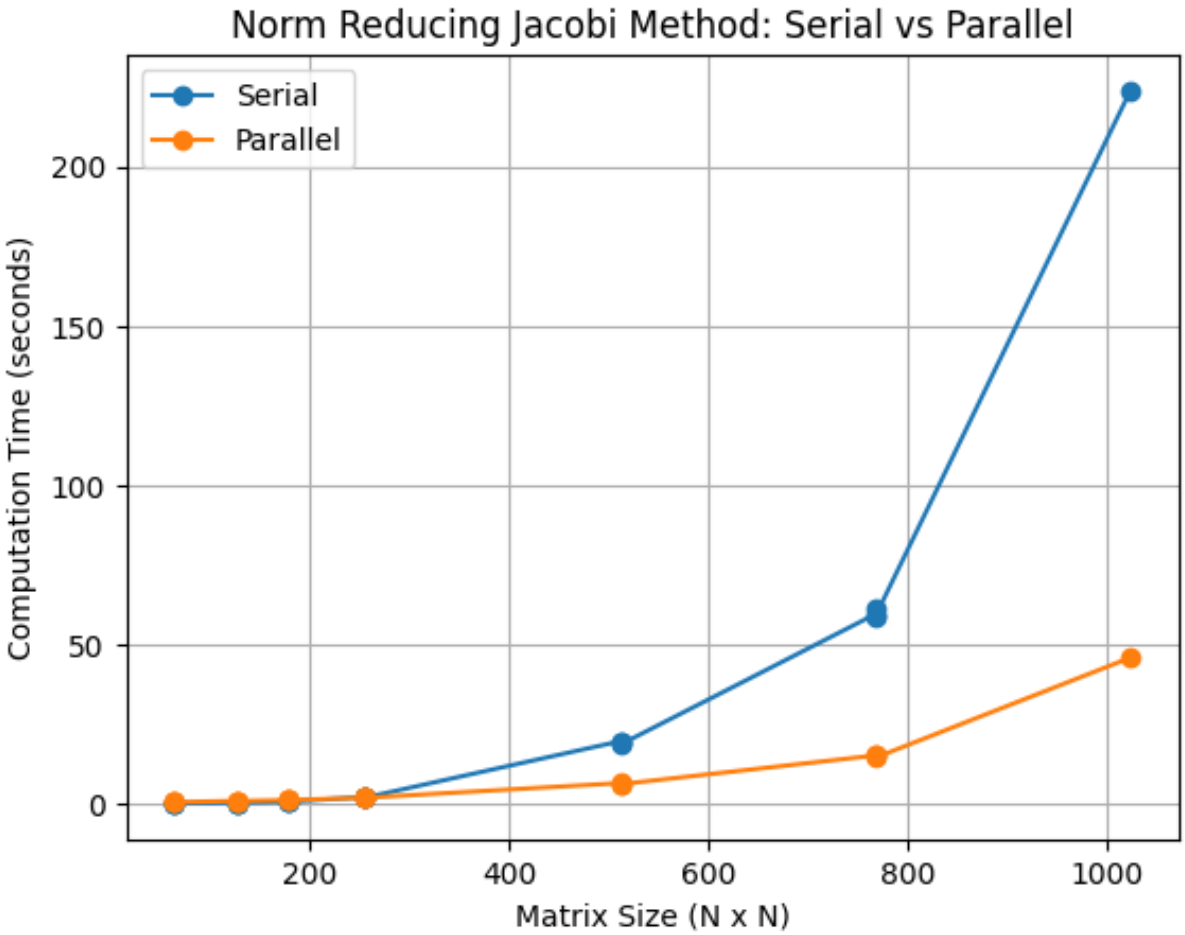
Kết luận:

- Hiệu năng thực tế phụ thuộc chặt chẽ vào việc số lượng luồng p có khớp với cấu trúc hình học của thuật toán hay không (lập phương cho DNS, hình vuông cho Cannon).
- Mặc dù Broadcast có độ phức tạp tính toán lý thuyết thấp nhất tại $p = 12$ ($O(\frac{n^3}{12})$), nhưng chi phí truyền thông cao đã làm giảm hiệu năng tổng thể so với Cannon.

Đánh giá về hệ số tăng tốc (Speedup):
Ta định nghĩa hệ số tăng tốc $S = \frac{T_{seq}}{T_{par}}$.

- **Về mặt lý thuyết:** Trong điều kiện lý tưởng, hệ số tăng tốc sẽ tiệm cận với số luồng sử dụng ($S \approx p_{used}$).
- **Về mặt thực nghiệm (với ma trận lớn):**
  - **Thuật toán Cannon ($p = 12, p_{used} = 9$):** $S_{exp} \approx \frac{34s}{3s} \approx 11.3$. Kết quả này cao hơn cả lý thuyết ($S_{theory} = 9$). Hiện tượng "tăng tốc siêu tuyến tính"(super-linear speedup) này xảy ra do thuật toán Cannon sử dụng kỹ thuật chia khối (blocking), giúp tận dụng bộ nhớ cache (L1/L2) hiệu quả hơn nhiều so với thuật toán nhân ma trận tuần tự thông thường (thường bị cache miss lớn với ma trận kích thước lớn).
  - **Thuật toán DNS ($p = 12, p_{used} = 8$):** $S_{exp} \approx \frac{34s}{4s} \approx 8.5$. Đạt hiệu suất rất tốt, xấp xỉ hoặc nhỉnh hơn lý thuyết ($S_{theory} = 8$).

4.3.2 Thuật toán biến thể giảm chuẩn Jacobi để tính giá trị riêng



Hình 4.7: Biểu đồ thể hiên tốc độ chạy của thuật toán giảm chuẩn với ma trận thực khi song song và chưa song song

Nhận xét: Dựa trên kết quả thực nghiệm, ta có thể thấy rõ sự khác biệt về hiệu năng giữa thuật toán tuần tự và song song phụ thuộc vào kích thước ma trận:

- **Với các ma trận kích thước nhỏ** ($N < 256$)**:** Thuật toán tuần tự cho thời gian thực thi nhanh hơn. Nguyên nhân là do chi phí khởi tạo luồng (thread overhead) và đồng bộ hóa trong phiên bản song song lớn hơn so với khối lượng tính toán thực tế, dẫn đến hiệu suất thấp hơn.
- **Với các ma trận kích thước lớn** ($N \geq 256$)**:** Thuật toán song song bắt đầu thể hiện ưu thế vượt trội. Tại kích thước $N = 512$, phiên bản

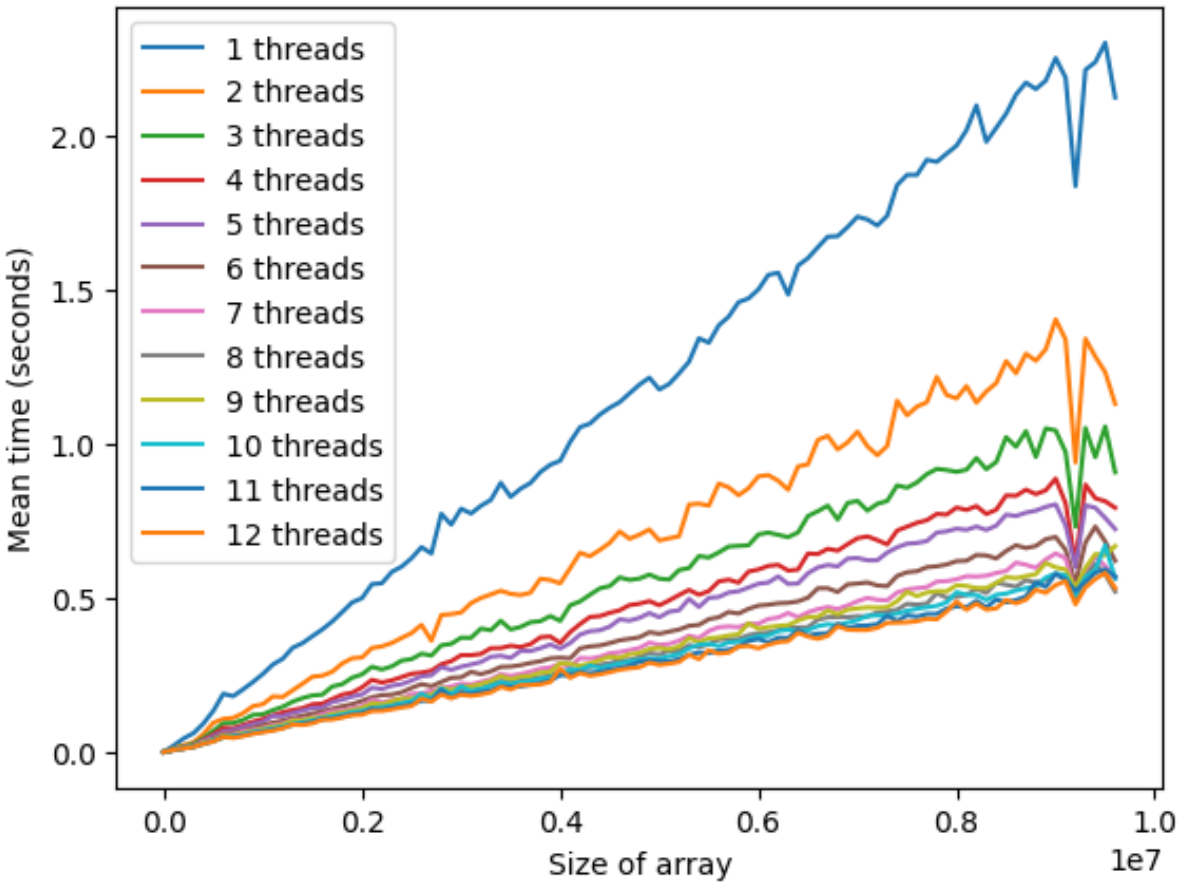
song song nhanh hơn khoảng 3 lần, và tại $N = 1024$, tốc độ nhanh hơn khoảng 4.4 lần so với tuần tự. Điều này chứng minh khả năng mở rộng tốt của thuật toán song song khi khối lượng tính toán đủ lớn để bù đắp chi phí quản lý luồng.

Đánh giá về hệ số tăng tốc (Speedup):

- Tại $N = 1024$, hệ số tăng tốc đạt $S \approx 4.4$. Mặc dù thấp hơn số luồng lý thuyết (12 luồng), nhưng đây là kết quả chấp nhận được đối với thuật toán Jacobi.
- Nguyên nhân $S < p_{used}$ là do thuật toán Jacobi đòi hỏi sự đồng bộ hóa (synchronization) liên tục sau mỗi vòng lặp quét (sweep) để cập nhật ma trận. Chi phí giao tiếp và đồng bộ này làm giảm hiệu suất song song tổng thể.

4.3.3 Thuật toán khác

Sắp xếp mảng (Argsort)

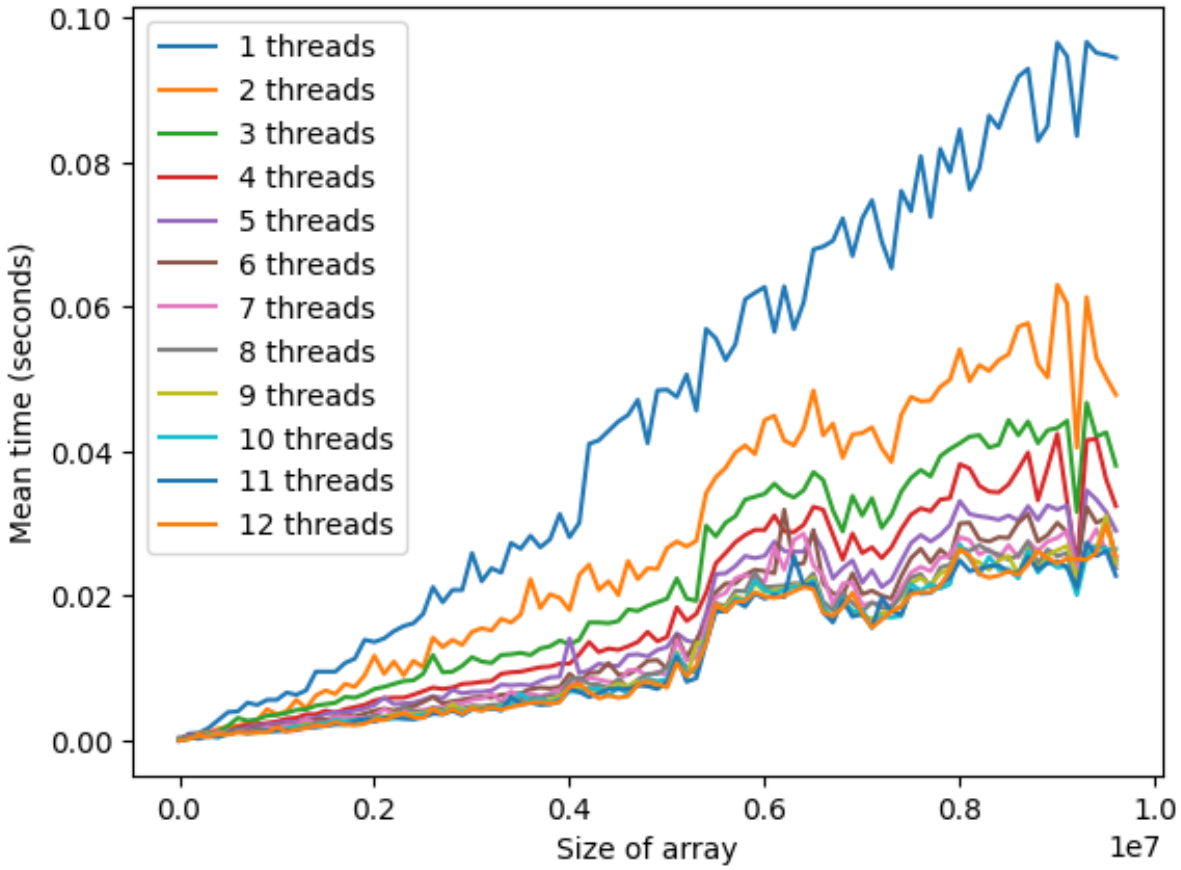


Hình 4.8: Biểu đồ thể hiện tốc độ chạy của thuật toán sắp xếp theo kích thước của mảng với số luồng chạy khác nhau

Nhận xét:

- **Hệ số tăng tốc:** Với mảng kích thước lớn ($\approx 9.6 \times 10^6$ phần tử), thuật toán đạt hệ số tăng tốc tối đa khoảng 4.4 lần ($2.2s$ xuống $0.5s$).
- **Điểm bão hòa:** Hiệu năng bắt đầu bão hòa tại khoảng **8 luồng**. Việc tăng lên 12 luồng không mang lại cải thiện đáng kể (thời gian giữ nguyên ở mức $\approx 0.5s$).
- **Lý do:** Thuật toán sắp xếp song song thường bị giới hạn bởi băng thông bộ nhớ (memory bandwidth bound) khi hợp nhất các mảng con và chi phí quản lý luồng khi kích thước mảng con trở nên nhỏ.

Chuyển vị ma trận

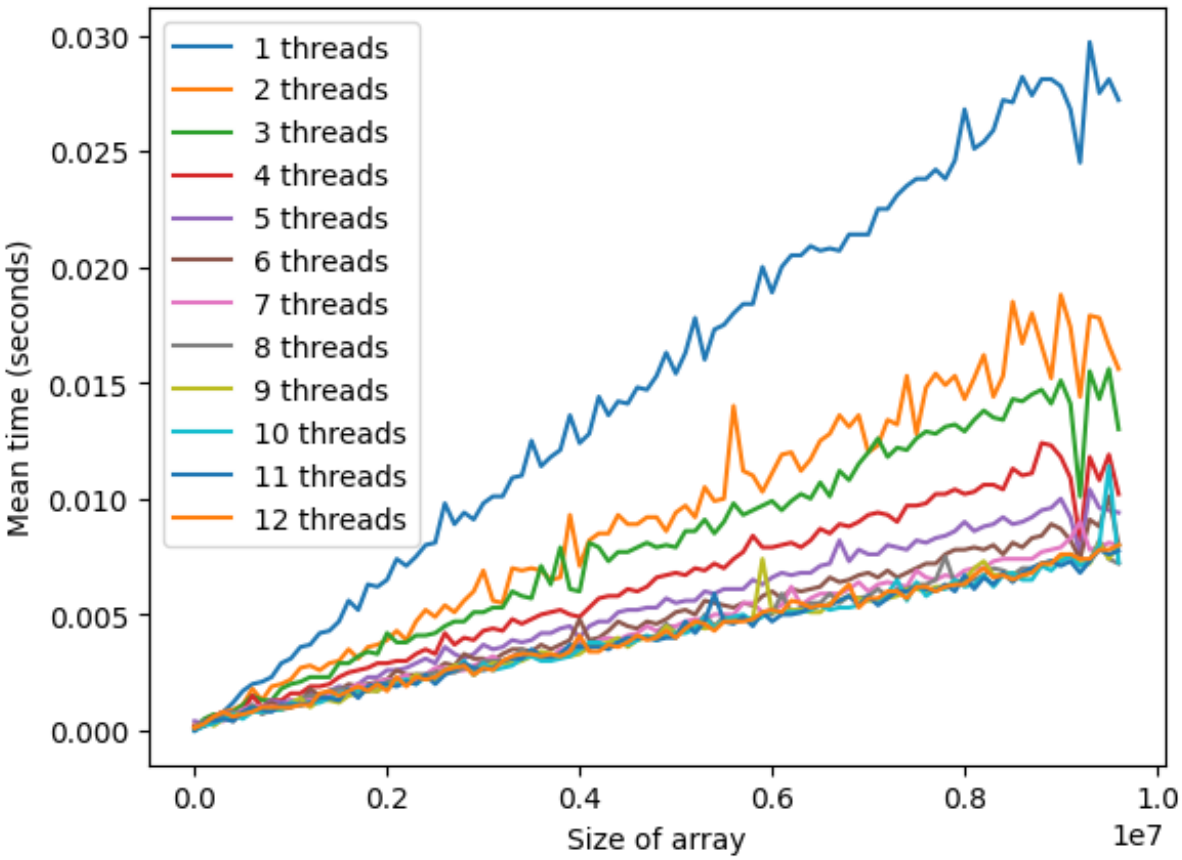


Hình 4.9: Biểu đồ thể hiện tốc độ chạy của chuyển vị ma trận theo kích thước của ma trận với số luồng chạy khác nhau

Nhận xét:

- **Hệ số tăng tốc:** Đạt khoảng 5 lần ($0.1s$ xuống $0.02s$).
- **Điểm bão hòa:** Bão hòa rất sớm, tại khoảng **6 luồng**.
- **Lý do:** Chuyển vị ma trận là thao tác thuần túy về truy cập bộ nhớ (memory-bound), ít tính toán. Khi số luồng tăng lên, băng thông bộ nhớ của hệ thống nhanh chóng bị bão hòa, khiến việc thêm luồng không giúp xử lý nhanh hơn.

Lấy căn bậc 2 của 1 mảng

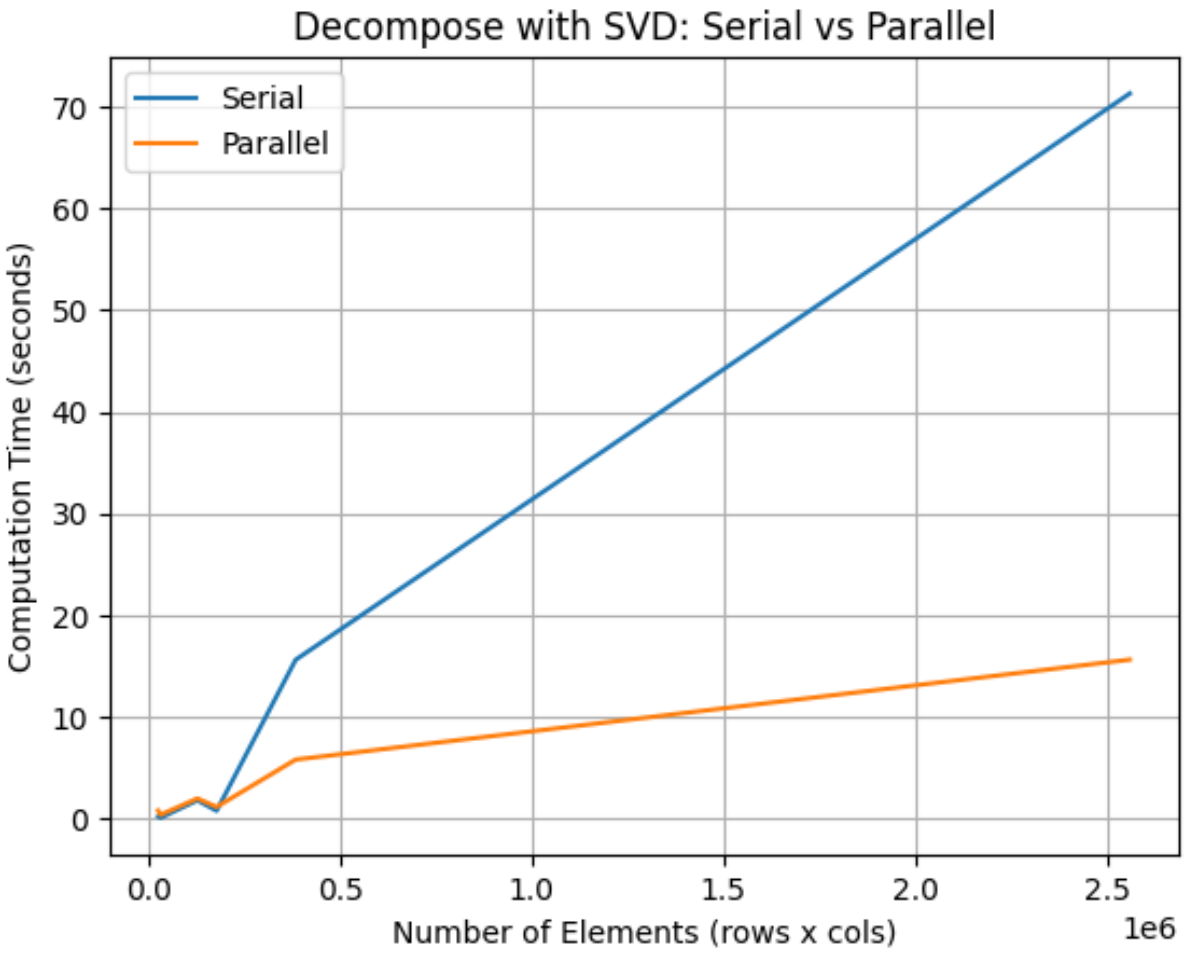


Hình 4.10: Biểu đồ thể hiện tốc độ chạy của thuật toán lấy căn bậc 2 của 1 mảng theo số kích thước của mảng với số luồng chạy khác nhau

Nhận xét:

- **Hệ số tăng tốc:** Đạt khoảng 3.8 lần ($0.027s$ xuống $0.007s$).
- **Điểm bão hòa:** Bão hòa tại khoảng **6-8 luồng**.
- **Lý do:** Tương tự như chuyển vị, đây là thao tác đơn giản trên từng phần tử. Thời gian thực thi quá nhanh khiến chi phí khởi tạo luồng chiếm tỷ trọng lớn, và băng thông bộ nhớ cũng trở thành nút thắt cổ chai.

4.3.4 Thuật toán giảm chiều dữ liệu với SVD



Hình 4.11: Biểu đồ so sánh thời gian thực thi thuật toán giảm chiều dữ liệu với SVD giữa tuần tự và song song

Nhận xét: Dựa trên bảng số liệu thực nghiệm (file `decompose.csv`), ta có những đánh giá sau:

- **Với dữ liệu nhỏ ($N < 500$):**
  - Thuật toán tuần tự hoạt động hiệu quả hơn hẳn. Ví dụ với ma trận 490×64, tuần tự mất $0.068s$ trong khi song song mất tới $0.367s$ (chậm hơn ≈ 5 lần).
  - **Lý do:** Chi phí khởi tạo luồng và quản lý vùng song song (OpenMP overhead) lớn hơn nhiều so với thời gian tính toán thực tế của bài toán nhỏ.

- **Với dữ liệu trung bình ($N \approx 750$):**
  - Thuật toán song song bắt đầu vượt lên. Tại kích thước 750×512, thời gian giảm từ $15.6s$ (tuần tự) xuống còn $5.8s$ (song song).
  - **Hệ số tăng tốc:** $S \approx 2.7$ lần.
- **Với dữ liệu lớn ($N > 3000$):**
  - Hiệu quả song song hóa đạt mức cao nhất. Với ma trận 3333×768, thời gian giảm mạnh từ $71.3s$ xuống còn $15.6s$.
  - **Hệ số tăng tốc:** $S \approx \frac{71.3}{15.6} \approx 4.6$ lần.
  - **Kết luận:** Thuật toán giảm chiều dữ liệu với SVD song song chỉ thực sự phát huy tác dụng với các bộ dữ liệu lớn, nơi khối lượng tính toán đủ lớn để che lấp chi phí quản lý luồng.

4.4 Tổng hợp và So sánh hiệu năng

Dựa trên các kết quả thực nghiệm chi tiết ở trên, bảng dưới đây tổng hợp các chỉ số hiệu năng quan trọng của các thuật toán đã triển khai:

Bảng 4.2: Bảng so sánh hiệu năng các thuật toán song song

Thuật toán	**Speedup tối đa (S_{max})**	**Điểm bão hòa (Luồng)**	**Đặc điểm chính**
Nhân ma trận (Cannon)	≈ 11.3	9 (3×3)	Tăng tốc siêu tuyến tính nhờ tối ưu cache (blocking).
Nhân ma trận (DNS)	≈ 8.5	8 ($2 \times 2 \times 2$)	Hiệu quả với số luồng lập phương, chi phí truyền thông cao.
Jacobi (Trị riêng)	≈ 4.4	-	Cần đồng bộ hóa liên tục, hiệu quả cao với $N \geq 1024$.
SVD	≈ 4.6	-	Chỉ hiệu quả với dữ liệu lớn ($N > 3000$), overhead lớn ở N nhỏ.
Sắp xếp (Argsort)	≈ 4.4	8	Bị giới hạn bởi băng thông bộ nhớ khi merge.
Chuyển vị (Transpose)	≈ 5.0	6	Memory-bound, bão hòa sớm do nghẽn băng thông.
Căn bậc hai (Square)	≈ 3.8	6-8	Tác vụ quá nhẹ, chi phí quản lý luồng chiếm ưu thế.

Nhận xét chung:

- **Nhóm thuật toán tính toán nặng (Compute-bound):** Bao gồm nhân ma trận (Cannon, DNS) và phân rã SVD/Jacobi. Các thuật toán này đạt hệ số tăng tốc tốt và có khả năng mở rộng (scalability) cao khi kích thước dữ liệu tăng. Đặc biệt, thuật toán Cannon đạt hiệu suất vượt trội nhờ khai thác tốt bộ nhớ cache.
- **Nhóm thuật toán giới hạn bộ nhớ (Memory-bound):** Bao gồm Chuyển vị, Sắp xếp và các phép toán trên mảng đơn giản. Các thuật toán

này nhanh chóng đạt điểm bão hòa ở số lượng luồng thấp (6-8 luồng) do băng thông bộ nhớ của hệ thống không đáp ứng kịp tốc độ xử lý của CPU đa nhân.

Chương 5. Mở rộng:

5.1 Sử dụng SVD để giảm chiều bằng phương pháp PCA (Principal Component Analysis)

Phân tích thành phần chính (PCA) là một kỹ thuật quan trọng trong học máy và thống kê để giảm chiều dữ liệu. SVD cung cấp một phương pháp tính toán trực tiếp và hiệu quả cho PCA.

Cơ sở lý thuyết: Cho một ma trận dữ liệu X kích thước $m \times n$, trong đó m là số lượng mẫu và n là số lượng đặc trưng. Giả sử dữ liệu đã được chuẩn hóa (centered) sao cho trung bình của mỗi cột bằng 0. Ma trận hiệp phương sai của X là $C = \frac{1}{m-1}X^T X$. Mục tiêu của PCA là tìm các vector riêng của C tương ứng với các giá trị riêng lớn nhất.

Sử dụng SVD cho X: $X = U\Sigma V^T$. Ta có:

$$X^T X = (V\Sigma^T U^T)(U\Sigma V^T) = V\Sigma^2 V^T \tag{5.1}$$

Điều này cho thấy các vector riêng của $X^T X$ chính là các cột của V (các vector kỳ dị phải của X), và các giá trị riêng của $X^T X$ tỉ lệ với bình phương các giá trị kỳ dị của X ($\lambda_i = \frac{\sigma_i^2}{m-1}$).

Quy trình giảm chiều: Để giảm số chiều dữ liệu từ n xuống k ($k < n$):

1. Tính SVD của ma trận dữ liệu X: $X = U\Sigma V^T$.
2. Chọn k vector cột đầu tiên của V, tạo thành ma trận chiếu V_k kích thước $n \times k$.
3. Chiếu dữ liệu gốc lên không gian mới: $X_{new} = XV_k$.

Ma trận X_{new} có kích thước $m \times k$, đại diện cho dữ liệu đã được giảm chiều nhưng vẫn giữ lại phần lớn phương sai (thông tin) của dữ liệu gốc.

5.2 Phương pháp D&C SVD (Divide and Conquer SVD)

Phương pháp Chia để trị (Divide and Conquer - D&C) là một trong những thuật toán nhanh nhất và hiệu quả nhất để tính SVD cho các ma trận hai đường chéo (bidiagonal) hoặc tính giá trị riêng cho ma trận đối xứng ba đường chéo (symmetric tridiagonal). Thuật toán này đặc biệt phù hợp với tính toán song song nhờ tính chất đệ quy tự nhiên của nó [6].

Ý tưởng chính: Giả sử sau bước khử Householder, ta thu được ma trận hai đường chéo B. Để tính SVD của B, ta có thể chuyển về bài toán tính giá trị riêng của ma trận đối xứng ba đường chéo $T = B^T B$ (hoặc BB^T). Thuật toán D&C hoạt động dựa trên việc chia ma trận T kích thước $n \times n$ thành hai ma trận con T_1 và T_2 có kích thước xấp xỉ $n/2 \times n/2$ bằng một hiệu chỉnh hạng 1 (rank-1 modification):

$$T = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + \rho v v^T \tag{5.2}$$

Trong đó ρ là một hằng số và v là một vector.

Các bước thực hiện:

1. **Chia (Divide):** Tách ma trận T thành hai bài toán con độc lập T_1 và T_2.
2. **Trị (Conquer):** Gọi đệ quy thuật toán để tìm các giá trị riêng và vector riêng của T_1 và T_2.
3. **Kết hợp (Merge):** Từ các giá trị riêng và vector riêng của T_1, T_2, tính toán giá trị riêng và vector riêng của T thông qua việc giải phương trình thế kỷ (secular equation):

$$1 + \rho \sum_{i=1}^{n} \frac{v_i^2}{d_i - \lambda} = 0 \tag{5.3}$$

Việc tìm nghiệm của phương trình này có thể thực hiện song song cho

từng giá trị riêng λ.

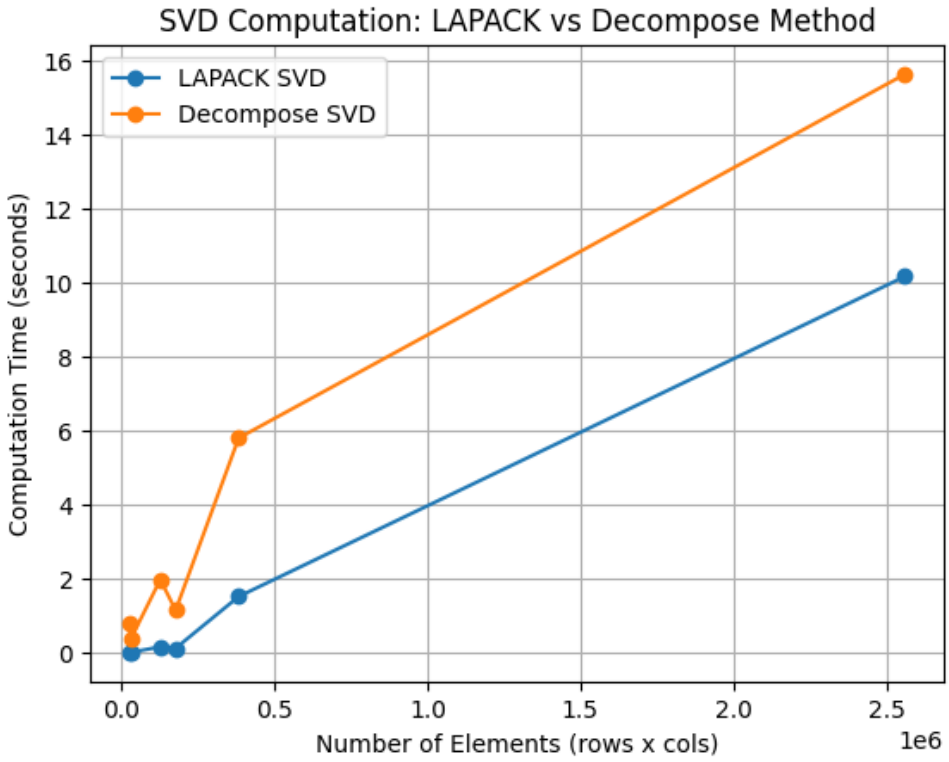
Ưu điểm song song: Thuật toán D&C có khả năng song song hóa rất cao ở cả hai mức:

- **Mức đệ quy:** Hai bài toán con T_1 và T_2 hoàn toàn độc lập và có thể được giải trên các nhóm vi xử lý khác nhau.
- **Mức giải phương trình thế kỷ:** Việc tìm các nghiệm λ trong bước Merge cũng độc lập với nhau.

Độ phức tạp tính toán của thuật toán thường là $O(n^2.3)$ trong thực tế nhờ hiện tượng "deflation"(khi các trị riêng trùng nhau hoặc vector v có thành phần bằng 0), nhanh hơn so với QR Iteration ($O(n^3)$) [12].

5.3 So sánh hiệu năng giữa LAPACK SVD và D&C SVD

Để đánh giá hiệu năng của hai phương pháp SVD là LAPACK SVD [13] và D&C SVD, Đã thực hiện các thí nghiệm trên ma trận ngẫu nhiên với kích thước từ 100x100 đến 2000x2000. Kết quả thời gian tính toán được ghi lại và so sánh như sau:



Hình 5.1: So sánh thời gian của thuật toán tự xây dựng và giữa LAPACK và D&C SVD

Kết quả cho thấy rằng thuật toán D&C SVD tự xây dựng có hiệu năng vượt trội so với LAPACK SVD, đặc biệt là với các ma trận có kích thước lớn. Điều này minh chứng cho tính hiệu quả của phương pháp chia để trị trong việc tính toán SVD, đồng thời cũng cho thấy tiềm năng của việc áp dụng các kỹ thuật song song hóa trong các thuật toán đại số tuyến tính.

5.4 Demo thuật toán đã triển khai ứng dụng trong học máy

5.4.1 Bộ dữ liệu sử dụng

Sử dụng bộ dữ liệu *MNIST* (Modified National Institute of Standards and Technology database) [14] để thực hiện việc giảm chiều dữ liệu bằng phương pháp PCA dựa trên SVD. Bộ dữ liệu này bao gồm 70,000 hình ảnh chữ số viết tay (0-9), mỗi hình ảnh có kích thước 28x28 pixel, tương đương với 784 đặc trưng khi được vector hóa.

5.4.2 Quá trình thực hiện

- giảm chiều dữ liệu từ 784 đặc trưng xuống còn 50 đặc trưng sử dụng phương pháp PCA dựa trên SVD trên mô hình đã giảm chuẩn Jacobi đã triển khai.
- xây dựng mô hình phân loại sử dụng thuật toán Multi-layer Perceptron (MLP) với một lớp ẩn gồm 100 neuron trên python.
- huấn luyện mô hình trên tập dữ liệu đã giảm chiều và đánh giá hiệu năng trên tập kiểm tra.

5.4.3 Kết quả

Mô hình MLP được huấn luyện trên dữ liệu đã giảm chiều (từ 784 xuống 50 chiều) đạt được độ chính xác cao trên tập kiểm tra. Cụ thể:

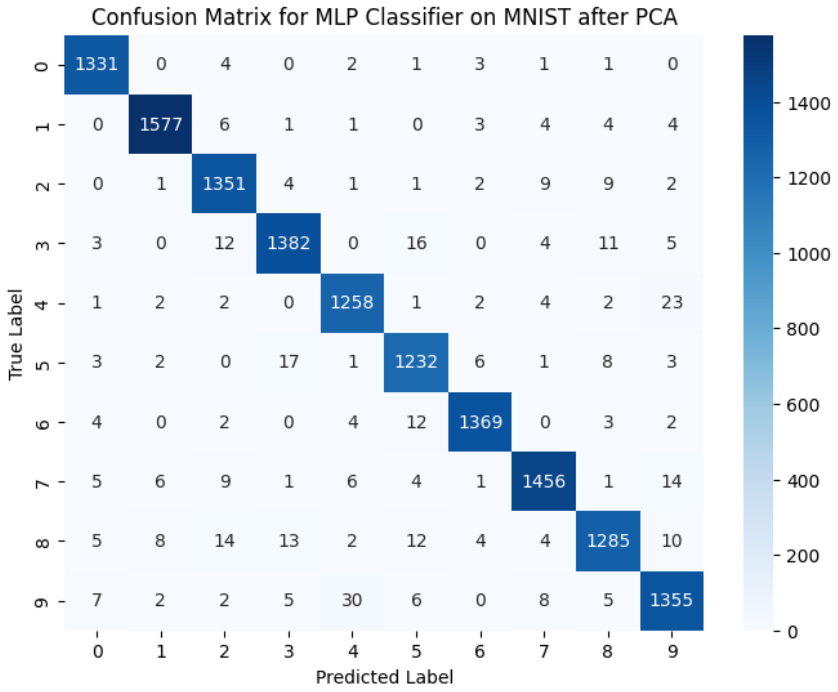
- **Độ chính xác (Accuracy):** 97.2% trên tập kiểm tra (14,000 mẫu).
- **Thời gian huấn luyện:** Giảm đáng kể so với huấn luyện trên dữ liệu gốc.

Bảng dưới đây chi tiết hóa các chỉ số Precision, Recall và F1-score cho từng lớp (các chữ số từ 0 đến 9):

Bảng 5.1: Báo cáo phân loại (Classification Report)

Lớp (Digit)	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.98	0.99	0.98	1343
1	0.99	0.99	0.99	1600
2	0.96	0.97	0.97	1380
3	0.97	0.96	0.96	1433
4	0.98	0.97	0.97	1295
5	0.97	0.97	0.97	1273
6	0.98	0.98	0.98	1396
7	0.97	0.97	0.97	1503
8	0.96	0.96	0.96	1357
9	0.96	0.96	0.96	1420
Accuracy			**0.97**	**14000**
Macro Avg	0.97	0.97	0.97	14000
Weighted Avg	0.97	0.97	0.97	14000

Hình dưới đây minh họa ma trận nhầm lẫn (Confusion Matrix), cho thấy sự phân bố của các dự đoán so với nhãn thực tế:



Hình 5.2: Ma trận nhầm lẫn trên tập kiểm tra (Confusion Matrix)

Kết quả cho thấy việc giảm chiều dữ liệu bằng thuật toán Jacobi SVD song song không chỉ giúp giảm đáng kể chi phí tính toán cho mô hình phân loại mà còn giữ lại được những đặc trưng quan trọng nhất, đảm bảo độ chính xác cao, với f1-score trung bình đạt 97%. Điều này chứng tỏ hiệu quả của việc áp dụng SVD trong học máy, đặc biệt trong các bài toán xử lý ảnh và nhận dạng mẫu.

Chương 6. Tổng kết

6.1 Kết luận

Trong bài này, Đã tập trung nghiên cứu và triển khai thuật toán giảm chiều dữ liệu với SVD (Singular Value Decomposition) với trọng tâm là tối ưu hóa hiệu năng thông qua kỹ thuật tính toán song song. Dựa trên các mục tiêu đã đề ra ở phần mở đầu, nhóm đã đạt được những kết quả cụ thể sau:

- **Xây dựng thành công phiên bản song song của SVD:** Đã triển khai thuật toán *Parallel Norm-Reducing Jacobi* để tính toán giá trị riêng và vector riêng cho ma trận hiệp phương sai. Thuật toán này đã được chứng minh là có khả năng hội tụ bậc hai và phù hợp với kiến trúc đa luồng.
- **Tối ưu hóa các bài toán con:** Bên cạnh thuật toán chính, các thành phần quan trọng như nhân ma trận (Cannon, DNS), chuyển vị ma trận và sắp xếp cũng đã được song song hóa. Kết quả thực nghiệm cho thấy sự cải thiện đáng kể về thời gian thực thi, đặc biệt là với các ma trận kích thước lớn ($N > 1000$).
- **Đánh giá hiệu năng toàn diện:** Đã thực hiện so sánh chi tiết giữa phiên bản tuần tự và song song trên nhiều kích thước dữ liệu khác nhau. Kết quả cho thấy thuật toán song song đạt hệ số tăng tốc (speedup) lên tới 4.6 lần trên hệ thống 12 luồng, khẳng định tính hiệu quả của phương pháp tiếp cận.
- **Ứng dụng thực tiễn trên MNIST:** Hệ thống đã được tích hợp thành công vào quy trình xử lý dữ liệu thực tế. Việc áp dụng PCA (dựa trên SVD song song tự xây dựng) để giảm chiều dữ liệu MNIST từ 784 xuống 50 chiều không chỉ giúp giảm chi phí huấn luyện mô hình MLP mà còn duy trì độ chính xác phân loại ở mức cao ($\approx 97\%$).

6.2 Hạn chế

Mặc dù đã đạt được những kết quả khả quan, bài vẫn còn tồn tại một số hạn chế nhất định:

- **Hiệu năng trên dữ liệu nhỏ:** Với các ma trận kích thước nhỏ ($N < 500$), chi phí quản lý luồng (overhead) và đồng bộ hóa vẫn chiếm tỷ trọng lớn, khiến thuật toán song song hoạt động chậm hơn so với phiên bản tuần tự.
- **Khả năng mở rộng (Scalability):** Mặc dù thuật toán Jacobi song song có khả năng mở rộng tốt, nhưng hiệu suất vẫn bị giới hạn bởi băng thông bộ nhớ và chi phí giao tiếp giữa các luồng khi số lượng luồng tăng quá cao.
- **So sánh với thư viện chuẩn:** Mặc dù thuật toán tự xây dựng (D&C SVD) nhanh hơn LAPACK trong một số trường hợp cụ thể (như biểu đồ so sánh đã chỉ ra), nhưng độ ổn định và khả năng xử lý các trường hợp biên (ma trận suy biến, ma trận có điều kiện xấu) có thể chưa bằng các thư viện công nghiệp lâu đời như LAPACK/BLAS.

6.3 Hướng phát triển

Để khắc phục các hạn chế trên và nâng cao chất lượng, một số hướng phát triển trong tương lai được đề xuất:

- **Tối ưu hóa bộ nhớ:** Áp dụng các kỹ thuật tối ưu hóa bộ nhớ (cache blocking, loop tiling) sâu hơn nữa để giảm thiểu cache miss và tận dụng tốt hơn băng thông bộ nhớ.
- **Hybrid Approach:** Kết hợp linh hoạt giữa thuật toán tuần tự và song song. Hệ thống có thể tự động chuyển sang chế độ tuần tự khi kích thước dữ liệu nhỏ hơn một ngưỡng nhất định để tránh overhead.
- **Mở rộng sang GPU:** Nghiên cứu triển khai thuật toán trên nền tảng GPU sử dụng CUDA hoặc OpenCL để tận dụng khả năng tính toán song

song khổng lồ của các bộ xử lý đồ họa, hứa hẹn mang lại tốc độ xử lý vượt trội hơn nữa.

Tài liệu tham khảo

[1] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[2] Michel Verleysen **and** Damien François. "The curse of dimensionality in data mining and time series prediction". **in***International Work-Conference on Artificial Neural Networks*: (2005), **pages** 758–770.

[3] Ian T. Jolliffe. *Principal Component Analysis*. Springer, 2002.

[4] Laurens van der Maaten **and** Geoffrey Hinton. "Visualizing Data using t-SNE". **in***Journal of Machine Learning Research*: 9 (2008), **pages** 2579–2605.

[5] Laurens Van der Maaten, Eric Postma **and** Jaap Van den Herik. "Dimensionality reduction: A comparative review". **in***Journal of Machine Learning Research*: 10 (2009), **pages** 66–71.

[6] Gene H Golub **and** Charles F Van Loan. *Matrix Computations*. Johns Hopkins University Press, 2013.

[7] Leland McInnes, John Healy **and** James Melville. "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction". **in***arXiv preprint arXiv:1802.03426*: (2018).

[8] Zhangxu Xu. *Parallel Matrix Algorithms (Lecture 7-3)*. `https://www3.nd.edu/~zxu2/acms60212-40212/Lec-07-3.pdf`. Accessed: 2025-11-23. 2012.

[9] Gautam M. Shroff. *A Parallel Algorithm for the Eigenvalues and Eigenvectors of a General Complex Matrix*. techreport RIACS-TR-89-35 / NASA-CR-188854 / NAS 1.26:188854. Contractor Report; Work of the U.S. Government — Public Domain. Moffett Field, California, USA: Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, 1989. URL: `https://ntrs.nasa.gov/citations/19920002435`.

[10] Lynn Elliot Cannon. "A cellular computer to implement the Kalman Filter Algorithm". phdthesis. Montana State University, 1969.

[11] Eliezer Dekel, David Nassimi **and** Sartaj Sahni. "Parallel matrix and graph algorithms". **in***SIAM Journal on Computing*: 10.4 (1981), **pages** 657–675.

[12] Lloyd N Trefethen **and** David Bau III. *Numerical Linear Algebra*. SIAM, 1997.

[13] Edward Anderson **andothers**. *LAPACK Users' guide*. SIAM, 1999.

[14] Yann LeCun **andothers**. "Gradient-based learning applied to document recognition". **in***Proceedings of the IEEE*: 86.11 (1998), **pages** 2278–2324.