

UỶ BAN NHÂN DÂN
THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC SÀI GÒN



**BÁO CÁO TỔNG KẾT
ĐỀ TÀI NGHIÊN CỨU KHOA HỌC CỦA SINH VIÊN**

**TÊN ĐỀ TÀI: NGHIÊN CỨU VÀ SO SÁNH CƠ CHẾ XỬ LÝ DỮ
LIỆU PHÂN TÁN - SONG SONG VÀ MINH HOA**
Mã số đề tài: SV2024-92

Thuộc nhóm ngành khoa học: Công nghệ thông tin

Chủ nhiệm đề tài: Nguyễn Cảnh Hoàng Danh

Thành viên tham gia:

- Nguyễn Cảnh Hoàng Danh
- Lê Thị Thanh Huyền
- Trần Hà Khang
- Vạn Xuân Quang

Giảng viên hướng dẫn: TS. Đỗ Như Tài

Thành phố Hồ Chí Minh, 3/2025

UỶ BAN NHÂN DÂN
THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC SÀI GÒN

BÁO CÁO TỔNG KẾT
ĐỀ TÀI NGHIÊN CỨU KHOA HỌC CỦA SINH VIÊN

TÊN ĐỀ TÀI: NGHIÊN CỨU VÀ SO SÁNH CƠ CHẾ XỬ LÝ DỮ LIỆU PHÂN TÁN - SONG SONG VÀ MINH HOA
Mã số đề tài: SV2024-92

Xác nhận của Khoa

(ký, họ tên)

Giáo viên hướng dẫn

(ký, họ tên)

Chủ nhiệm đề tài

(ký, họ tên)

Thành phố Hồ Chí Minh, 3/2025

LỜI CẢM ƠN

Để hoàn thành dự án nghiên cứu khoa học “Nghiên cứu và so sánh cơ chế xử lý dữ liệu phân tán – song song và minh họa,” chúng em đã nhận được rất nhiều sự giúp đỡ và hỗ trợ tận tình. Chúng em xin trân trọng gửi lời cảm ơn sâu sắc đến:

- Khoa Công Nghệ Thông Tin – Trường Đại Học Sài Gòn đã tạo mọi điều kiện thuận lợi để chúng em có thể thực hiện nghiên cứu này.
- Chúng em xin gửi lời tri ân đến thầy Nguyễn Quốc Huy và thầy Đỗ Nhu Tài đã tận tình hướng dẫn, chỉ bảo trong suốt quá trình thực hiện đề tài. Sự định hướng và hỗ trợ của thầy đã giúp chúng em hoàn thành bài nghiên cứu một cách thuận lợi và hiệu quả.
- Các thành viên trong nhóm đã luôn đoàn kết, hỗ trợ lẫn nhau và nỗ lực hết mình để hoàn thành dự án với kết quả tốt nhất.

Cuối cùng, chúng em xin kính chúc các thầy cô luôn mạnh khỏe, thành công để tiếp tục dìu dắt các thế hệ học sinh, sinh viên trên con đường học tập và nghiên cứu.

MỤC LỤC

DANH MỤC HÌNH VẼ	iv
DANH MỤC BẢNG BIÊU	vii
DANH MỤC CÁC CHỮ VIẾT TẮT	viii
LỜI MỞ ĐẦU	ix
CHƯƠNG 1: PHẦN MỞ ĐẦU	1
1.1. Đặt vấn đề	1
1.2. Lý do chọn đề tài	2
1.3. Mục tiêu nghiên cứu	2
1.4. Đối tượng và phạm vi nghiên cứu	3
1.4.1. Đối tượng nghiên cứu	3
1.4.2. Phạm vi nghiên cứu	3
1.5. Phương pháp nghiên cứu	3
1.5.1. Cách tiếp cận nghiên cứu	3
1.5.2. Các nguồn dữ liệu và cách thu thập dữ liệu	3
1.5.3. Quy trình thực hiện	4
1.5.4. Dự kiến các phương pháp nghiên cứu	4
1.6. Những đóng góp mới của đề tài	5
CHƯƠNG 2: CƠ SỞ LÝ THUYẾT	6
2.1. Tổng quan cơ chế xử lý dữ liệu song song	6
2.1.1. Khái niệm	6
2.1.2. Ưu điểm của xử lý dữ liệu song song	7
2.1.3. Các kiến trúc song song phổ biến	8
2.1.4. Một số mô hình xử lý dữ liệu song song	10
2.1.5. Một số công nghệ xử lý dữ liệu song song (Công cụ và ngôn ngữ lập trình hỗ trợ)	17
2.2. Tổng quan về CUDA, Rapids, Cupy	21
2.2.1. Tổng quan về CUDA Numba	21
2.2.2. Tổng quan về RAPIDS	22
2.2.3. Tổng quan về CuPy	22
CHƯƠNG 3: CÁC BÀI TOÁN CƠ SỞ	24
3.1. Bài toán cộng vector	24

3.1.1. Mô tả bài toán	24
3.1.2. Ý tưởng	25
3.1.3. Cộng Vectơ trên CPU	25
3.1.4. Cộng Vectơ trên GPU	27
3.1.4. Kết quả thực nghiệm	34
3.2. Bài toán Histogram	37
3.2.1. Mô tả bài toán	37
3.2.2. Ý tưởng	38
3.2.3. Histogram trên CPU	38
3.2.4. Histogram trên GPU	39
3.2.5. Kết quả thực nghiệm	44
3.3. Bài toán Sum Reduce	46
3.3.1. Mô tả bài toán	46
3.3.2. Ý tưởng	46
3.3.4. Sum Reduce trên GPU	48
3.3.5. Kết quả thực nghiệm	51
3.4. Bài toán nhân ma trận	53
3.4.1. Mô tả bài toán	53
3.4.2. Ý tưởng	53
3.4.3. Nhân ma trận CPU	53
3.4.4. Nhân ma trận GPU	54
3.4.5. Kết quả thực nghiệm	66
CHƯƠNG 4: MỘT SỐ BÀI TOÁN KHÁC	69
4.1. Bài toán sắp xếp (Merge Sort)	69
4.1.1. Mô tả bài toán	69
4.1.2. Ý tưởng	69
4.1.3. Merge Sort CPU	70
4.1.4. Merge Sort GPU	71
4.1.5. Kết quả thực nghiệm	80
4.2. Bài toán Inverted Indexing	82
4.2.1. Mô tả bài toán	82
4.2.2. Ý tưởng	83

4.2.3. Inverted Indexing trên GPU	85
4.2.5. Kết quả thực nghiệm	95
CHƯƠNG 5: CƠ CHẾ XỬ LÝ DỮ LIỆU PHÂN TÁN	98
5.1. Cơ chế xử lý dữ liệu phân tán	98
5.1.1. Khái niệm xử lý dữ liệu phân tán (Distributed data processing)	98
5.1.2. Ưu điểm của xử lý dữ liệu phân tán	98
5.1.3. Một số mô hình xử lý dữ liệu phân tán	98
5.1.4. Một số công nghệ xử lý dữ liệu phân tán	99
5.2. Cơ chế xử lý dữ liệu song song phân tán	104
5.2.1. Tổng quan về cơ chế xử lý dữ liệu song song phân tán	104
5.2.2. Minh họa cơ chế xử lý dữ liệu song song phân tán	105
5.3. Tổng quan về Apache Spark	106
5.3.1. Giới thiệu	106
5.3.2. Kiến trúc của Apache Spark	107
5.3.3. Resilient Distributed Dataset (RDD)	108
5.3.4. Directed Acyclic Graph (DAG) và cơ chế lập lịch	122
5.3.5. Cơ chế quản lý bộ nhớ	123
5.3.6. Cơ chế xử lý luồng (Spark Streaming)	124
5.3.7. Tối ưu hóa hiệu suất xử lý song song	124
5.3.8. Kết luận	126
5.4. Ví dụ minh họa về cơ chế xử lý dữ liệu phân tán	126
5.4.1. Bài toán cộng vectơ	126
5.4.2. Bài toán Histogram	133
5.4.3. Bài toán Sum Reduce	137
5.4.4. Bài toán nhân ma trận	146
5.4.5. Bài toán sắp xếp (Merge Sort)	150
5.4.6. Bài toán Inverted Indexing	155
CHƯƠNG 6: KẾT LUẬN	165
6.1. Tổng kết	165
6.2. Hướng phát triển	165
TÀI LIỆU THAM KHẢO	166

DANH MỤC HÌNH VẼ

Hình 1.1. Minh họa về xử lý song song.....	1
Hình 2.1. So sánh cơ chế xử lý tuần tự và song song.....	6
Hình 2.2. Cấu trúc của CPU.....	9
Hình 2.3. Minh họa cơ chế MapReduce.....	9
Hình 2.4. So sánh cấu trúc CPU và GPU.....	10
Hình 2.5. Mô hình SIMD.....	11
Hình 2.6. Mô hình MIMD.....	12
Hình 2.7. Mô hình SIMT.....	13
Hình 2.8. Mô hình MapReduce.....	14
Hình 2.9. Kiến trúc CUDA.....	15
Hình 2.10. Mô hình nền tảng OpenCL.....	16
Hình 2.11. Mô hình Fork-Join của OpenMP.....	17
Hình 2.12. Mô hình bộ nhớ CUDA.....	19
Hình 2.13. Apache Spark sử dụng RDD để xử lý song song.....	20
Hình 2.14. Tổng quan về Rapids.....	22
Hình 3.1. Ý tưởng thực hiện cộng hai vectơ.....	25
Hình 3.2. Đoạn mã cộng hai vectơ trên mỗi lõi.....	26
Hình 3.3. Minh họa ý tưởng cộng 2 vectơ trên lập trình song song.....	28
Hình 3.4. Đoạn mã kernel cộng 2 vectơ được thực thi trên GPU.....	29
Hình 3.5. Kết quả thực nghiệm của bài toán cộng vectơ trên các phiên bản khác nhau.	35
Hình 3.6. Ý tưởng bài toán histogram.....	38
Hình 3.7. Tính toán histogram trên GPU.....	39
Hình 3.8. Minh họa cụ thể ý tưởng bài toán Histogram song song.....	40
Hình 3.9. Biểu đồ so sánh thời gian thực hiện histogram trên các bộ xử lý khác nhau.	45
Hình 3.10. Minh họa thuật toán Sum Reduce.....	46
Hình 3.11. Minh họa thuật toán Sum Reduce GPU.....	48
Hình 3.12. Biểu đồ thời gian chạy Sum Reduce trên Cython, CUDA, Cupy, Rapids..	52
Hình 3.13. Minh họa nhân ma trận trên GPU.....	55
Hình 3.14. Chia block và thread cho bài toán nhân ma trận.....	56
Hình 3.15. Cách CUDA lưu trữ ma trận.....	56

Hình 3.16. Xác định dòng và cột của ma trận trong CUDA	57
Hình 3.17. Mô hình tính toán của thread trong nhân ma trận CUDA	58
Hình 3.18. Biểu đồ biểu diễn thời gian chạy của các phiên bản nhân ma trận	67
Hình 4.1. Hình ảnh ví dụ thuật toán Merge Sort	69
Hình 4.2. Minh họa thuật toán Merge Sort trên GPU	72
Hình 4.3. Biểu đồ biểu diễn thời gian chạy của các phiên bản Merge Sort	81
Hình 4.4. Quy trình xây dựng chỉ mục đảo	83
Hình 4.5. Cấu trúc Posting List	84
Hình 4.6. Trình biên dịch của các phiên bản CUDA, Cupy, RAPIDS	93
Hình 4.7. Hiệu suất của các phương pháp khi xây dựng inverted index	96
Hình 5.1. Kiến trúc Hadoop	100
Hình 5.2. Luồng hoạt động Hadoop xử lý dữ liệu phân tán	101
Hình 5.3. Kiến trúc Apache Flink	103
Hình 5.4. Minh họa cơ chế xử lý dữ liệu song song phân tán	105
Hình 5.5. Kiến trúc tổng quan của Apache Spark	107
Hình 5.6. Minh họa quy trình đọc dữ liệu vào Apache Spark	108
Hình 5.7. Ví dụ minh họa kiến trúc xử lý dữ liệu trong Apache Spark Cluster	108
Hình 5.8. CPU Cores và Partitioning	110
Hình 5.9. Memory Considerations	111
Hình 5.10. MapReduce và Shuffling	112
Hình 5.11. Quá trình trong Spark UI	114
Hình 5.12. Minh họa DAG visualization cơ chế filter và map	114
Hình 5.13. Hình minh họa biểu đồ thời gian thực thi các tiến trình trong Spark	115
Hình 5.14. Giao diện của Spark UI hiển thị quá trình xử lý song song	117
Hình 5.15. Giao diện của Spark UI hiển thị quá trình xử lý song song	118
Hình 5.16. Hai giai đoạn của Stage	119
Hình 5.17. Quy trình thực thi của Spark	120
Hình 5.18. Hai node lưu trữ dữ liệu giống nhau	121
Hình 5.19. Giao tiếp dữ liệu giữa các node	122
Hình 5.20. Quy trình tối ưu hóa truy vấn trong Catalyst Optimizer	125
Hình 5.21. Hình ảnh minh họa bài toán cộng 2 vectơ phân tán	127
Hình 5.22. Biểu đồ thời gian chạy cộng hai vectơ của các cấu hình Spark	131

Hình 5.23. Hình ảnh minh họa thuật toán Histogram phân tán.....	133
Hình 5.24. Biểu đồ thời gian chạy Histogram của các cấu hình Spark.	136
Hình 5.25. Minh họa bài toán Sum Reduce với 2 worker.....	138
Hình 5.26. Biểu đồ thời gian chạy Sum Reduce của các cấu hình Spark.	144
Hình 5.27. Hình ảnh minh họa thuật toán nhân ma trận phân tán.	146
Hình 5.28. Biểu đồ thời gian chạy nhân ma trận của các cấu hình Spark.	149
Hình 5.29. Hình ảnh minh họa thuật toán Merge Sort phân tán.	151
Hình 5.30. Biểu đồ thời gian chạy Merge Sort của các cấu hình Spark.	154
Hình 5.31. Minh họa bài toán Inverted Indexing với 2 worker.	156
Hình 5.32. Biểu đồ thời gian chạy Inverted Indexing của các cấu hình Spark.	162

DANH MỤC BẢNG BIỂU

Bảng 3.1. Kết quả thời gian chạy của các phiên bản cộng vectơ (Mili giây)	35
Bảng 3.2. Kết quả thời gian chạy của các phiên bản Histogram (Mili giây).....	44
Bảng 3.3. Kết quả thời gian chạy của các phiên bản sum reduce (Mili giây).....	51
Bảng 3.4. Kết quả thời gian chạy của các phiên bản nhân ma trận (Mili giây).....	67
Bảng 4.1. Kết quả thời gian chạy của các phiên bản Merge Sort (Mili giây).....	81
Bảng 4.2. Kết quả thời gian chạy của các phiên bản Inverted Indexing (Mili giây) ...	95
Bảng 5.1. Kết quả thời gian chạy cộng hai vecto của các cấu hình Spark (Giây).....	131
Bảng 5.2. Kết quả thời gian chạy Histogram của các cấu hình Spark (Giây)	136
Bảng 5.3. Kết quả thời gian chạy Sum reduce của các cấu hình Spark (Giây)	143
Bảng 5.4. Kết quả thời gian chạy Merge Sort của các cấu hình Spark (Giây)	149
Bảng 5.5. Kết quả thời gian chạy Merge Sort của các cấu hình Spark (Giây)	153
Bảng 5.6. Kết quả thời gian chạy Inverted Indexing của các cấu hình Spark (Giây).	162

DANH MỤC CÁC CHỮ VIẾT TẮT

STT	Từ viết tắt	Ý nghĩa đầy đủ
1	API	Application Programming Interface
2	CPU	Central Processing Unit
3	CUDA	Compute Unified Device Architecture
4	cuDF	Compute Unified Device Architecture DataFrame
5	cuML	Compute Unified Device Architecture Machine Learning
6	cuGraph	Compute Unified Device Architecture Graph
7	CuPy	Compute Unified Device Architecture Python
8	GPGPU	General-Purpose Computing on GPU
9	GPU	Graphics Processing Unit
10	HDFS	Hadoop Distributed File System
11	IIU	Inverted Index Processing Unit
12	MIMD	Multiple Instruction, Multiple Data
13	MPI	Message Passing Interface
14	OpenCL	Open Computing Language
15	OpenMP	Open Multi-Processing
16	RAM	Random Access Memory
17	RDD	Resilient Distributed Dataset
18	SIMD	Single Instruction, Multiple Data
19	SIMT	Single Instruction, Multiple Threads
20	SM	Streaming Multiprocessor
21	SQL	Structured Query Language
22	TID	Task ID
23	vCPU	Virtual Central Processing Unit
24	YARN	Yet Another Resource Negotiator

LỜI MỞ ĐẦU

Trong thời đại dữ liệu bùng nổ, khi khối lượng thông tin ngày càng tăng trưởng và trở nên phức tạp, việc xử lý dữ liệu hiệu quả trở thành một thách thức lớn đối với các hệ thống máy tính. Nếu như trước đây, dữ liệu chủ yếu được xử lý theo cách tuần tự, thì ngày nay, với nhu cầu xử lý nhanh chóng và quy mô lớn, các phương pháp truyền thống không còn đáp ứng được yêu cầu thực tế.

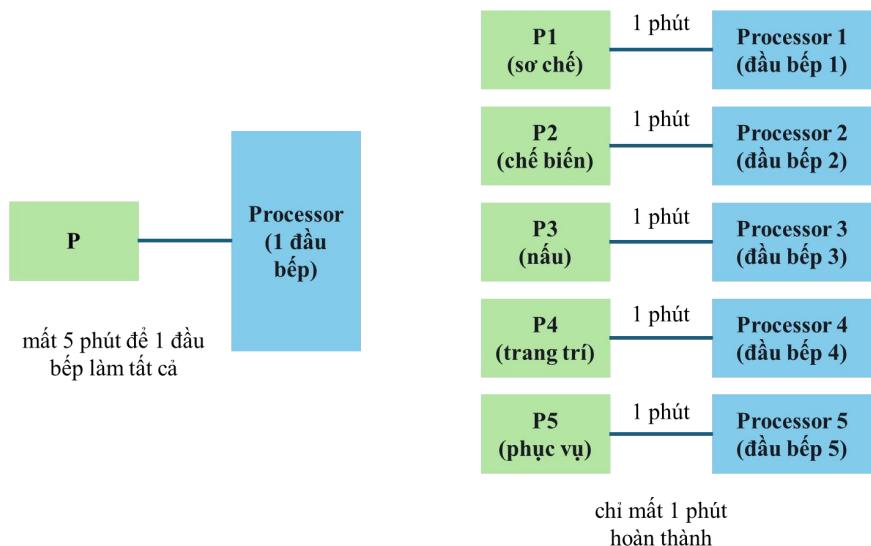
Trước thách thức đó, hai cơ chế xử lý dữ liệu chính đã ra đời: xử lý dữ liệu song song và xử lý dữ liệu phân tán. Trong khi xử lý song song tập trung khai thác sức mạnh tính toán của nhiều lõi trong cùng một hệ thống để tăng tốc độ xử lý, thì xử lý phân tán lại phân chia khối lượng công việc giữa nhiều máy tính khác nhau, giúp hệ thống có khả năng mở rộng và xử lý dữ liệu ở quy mô lớn hơn. Mỗi cơ chế đều có ưu điểm riêng và được ứng dụng trong các lĩnh vực khác nhau, từ phân tích dữ liệu lớn, trí tuệ nhân tạo đến các hệ thống lưu trữ và xử lý dữ liệu theo thời gian thực.

Vậy giữa hai cơ chế này có những khác biệt gì? Khi nào nên áp dụng xử lý song song và khi nào xử lý phân tán là lựa chọn tối ưu? Để trả lời những câu hỏi đó, đề tài này sẽ tập trung nghiên cứu, so sánh hai cơ chế xử lý dữ liệu song song và phân tán, từ đó đánh giá hiệu suất, ưu nhược điểm và khả năng ứng dụng thực tế của chúng.

CHƯƠNG 1: PHẦN MỞ ĐẦU

1.1. Đặt vấn đề

Trong thời đại bùng nổ dữ liệu như hiện nay, việc xử lý khối lượng lớn dữ liệu một cách hiệu quả là một thách thức vô cùng quan trọng và khó khăn. Hãy tưởng tượng một hệ thống xử lý giao dịch tài chính, nơi mỗi giây có hàng triệu giao dịch diễn ra. Nếu chỉ dựa vào một máy tính đơn lẻ, việc xử lý có thể trở nên chậm chạp, ảnh hưởng đến trải nghiệm người dùng và gây ra các rủi ro trong hệ thống. Chính vì vậy, các mô hình xử lý dữ liệu tiên tiến như xử lý song song và xử lý phân tán ngày càng trở nên quan trọng.



Hình 1.1. Minh họa về xử lý song song.

Theo **hình 1.1** - hãy hình dung một nhà hàng đông khách. Nếu chỉ có một đầu bếp phục vụ tất cả các món ăn, thời gian chờ sẽ rất lâu. Nhưng nếu có một đội ngũ đầu bếp làm việc cùng nhau, mỗi người đảm nhận một phần của quy trình nấu ăn, các món ăn sẽ được phục vụ nhanh hơn. Xử lý song song cũng hoạt động theo cách tương tự, tận dụng nhiều luồng tính toán trên một hệ thống để xử lý dữ liệu đồng thời.

Tuy nhiên, khi khối lượng công việc tăng lên quá lớn, lượng dữ liệu ngày càng tăng trưởng mạnh mẽ, CPU truyền thống có thể mất nhiều thời gian để xử lý dữ liệu do giới hạn về số lõi và khả năng tính toán. Trong khi đó, GPU với hàng nghìn lõi nhỏ có thể xử lý hàng loạt phép tính song song, giúp rút ngắn thời gian xử lý đáng kể. Chẳng hạn,

khi áp dụng bộ lọc trung vị trên một hình ảnh có độ phân giải cao, CPU có thể mất vài giây để hoàn thành, trong khi GPU chỉ mất một phần nhỏ của thời gian đó.

Mặt khác, nếu lượng khách hàng tăng lên đáng kể, một nhà hàng duy nhất sẽ không đủ sức phục vụ. Khi đó, giải pháp là mở rộng thành một chuỗi nhà hàng, phân bổ công việc giữa nhiều địa điểm khác nhau. Đây chính là cách xử lý phân tán hoạt động: chia nhỏ dữ liệu và xử lý trên nhiều máy tính trong hệ thống mạng.

1.2. Lý do chọn đề tài

Với sự phát triển mạnh mẽ của công nghệ dữ liệu lớn (Big Data) và trí tuệ nhân tạo (AI), nhu cầu xử lý dữ liệu hiệu quả ngày càng trở nên cấp thiết. Các hệ thống ngày nay không chỉ đòi hỏi tốc độ cao mà còn phải có khả năng mở rộng linh hoạt để đáp ứng khối lượng dữ liệu khổng lồ. Do đó, việc hiểu rõ sự khác biệt giữa hai cơ chế xử lý song song và phân tán là vô cùng quan trọng trong việc tối ưu hóa hiệu suất của các ứng dụng thực tế.

Cùng với đó là sự phát triển của các kiến trúc phần cứng hiện đại như CPU đa lõi và GPU mạnh mẽ, cũng như các nền tảng xử lý phân tán như Apache Spark, việc lựa chọn phương pháp xử lý phù hợp có thể tạo ra sự khác biệt đáng kể về hiệu năng và chi phí vận hành.

1.3. Mục tiêu nghiên cứu

Nghiên cứu nhằm phân tích và so sánh hai cơ chế xử lý dữ liệu: song song và phân tán, dựa trên các tiêu chí quan trọng như:

- Hiệu suất xử lý và thời gian thực thi.
- Khả năng mở rộng và chi phí triển khai.
- Mức độ phức tạp trong triển khai và bảo trì.

Bên cạnh đó, nghiên cứu cũng hướng đến việc đánh giá hiệu năng của từng cơ chế thông qua các bài toán tiêu biểu như cộng vector, histogram, merge sort, v.v. Những bài toán này không chỉ phổ biến trong xử lý dữ liệu mà còn giúp kiểm chứng khả năng mở rộng và tối ưu hóa của các phương pháp.

1.4. Đối tượng và phạm vi nghiên cứu

1.4.1. Đối tượng nghiên cứu

Nghiên cứu tập trung vào các cơ chế xử lý dữ liệu song song và phân tán, bao gồm:

- Các nguyên lý hoạt động và đặc điểm của từng cơ chế.
- Ứng dụng của hai cơ chế trong các bài toán thực tế.
- Hiệu suất và khả năng mở rộng khi áp dụng vào các thuật toán điển hình như cộng vector, histogram, merge sort, v.v.

Bằng cách phân tích và thực nghiệm trên những bài toán này, nghiên cứu sẽ đánh giá hiệu quả của từng phương pháp, từ đó đưa ra nhận định về ưu điểm và hạn chế của mỗi cơ chế.

1.4.2. Phạm vi nghiên cứu

Nghiên cứu này sẽ tập trung vào:

- Phân tích và thực nghiệm trên các nền tảng CPU, GPU, và Apache Spark.
- Đánh giá hiệu năng dựa trên các tiêu chí như thời gian xử lý, khả năng mở rộng và chi phí tính toán.
- So sánh hiệu suất của thuật toán temporal median filter trên từng nền tảng để rút ra kết luận về tính phù hợp của từng phương pháp xử lý.

1.5. Phương pháp nghiên cứu

1.5.1. Cách tiếp cận nghiên cứu

- Đề tài tập trung vào việc so sánh hiệu suất của các phương pháp xử lý song song trên CPU và GPU thông qua các bài toán điển hình, bao gồm: Cộng vector, tính histogram, sum reduction, nhân ma trận, merge sort, ... để hiểu rõ sự khác biệt về hiệu suất giữa các cơ chế xử lý.
- Tổng hợp kết quả thực nghiệm để đưa ra đề xuất tối ưu hóa, giúp lựa chọn phương pháp xử lý phù hợp với từng loại bài toán.

1.5.2. Các nguồn dữ liệu và cách thu thập dữ liệu

Nguồn dữ liệu:

- Tập dữ liệu đầu vào: Sử dụng các tập dữ liệu phổ biến và phù hợp với các bài toán cơ sở và thuật toán temporal median filter, chẳng hạn như các tập dữ liệu

video hoặc ảnh chuỗi cho bài toán lọc nền, và các tập dữ liệu văn bản cho bài toán wordcount và inverted indexing.

- Nguồn tài liệu nghiên cứu: Các bài báo, tài liệu kỹ thuật, và nghiên cứu liên quan đến thuật toán temporal median filter, Python Numba, CUDA Numba, và Apache Spark.

Các thu thập dữ liệu:

- Triển khai thuật toán: Phát triển và triển khai các thuật toán trên ba nền tảng xử lý (CPU, GPU, Apache Spark) theo các phương pháp tối ưu hóa.
- Ghi nhận hiệu suất: Đo thời gian xử lý, mức sử dụng tài nguyên CPU/GPU, và hiệu quả

1.5.3. Quy trình thực hiện

- Thí nghiệm với các bài toán cơ sở: Thực hiện các thí nghiệm trên các bài toán như wordcount, inverted indexing,... để so sánh hiệu suất giữa các nền tảng.
- Thí nghiệm tối ưu hóa thuật toán: Chạy các thí nghiệm với thuật toán temporal median filter trên CPU, GPU, và Apache Spark. Ghi nhận và phân tích kết quả để đánh giá hiệu suất và khả năng mở rộng.
- Khảo sát kết quả: Thu thập và phân tích phản hồi từ các thí nghiệm để đánh giá hiệu quả của các phương pháp tối ưu hóa.

1.5.4. Dự kiến các phương pháp nghiên cứu

- Phương pháp nghiên cứu thực nghiệm: Dựa vào các thí nghiệm và dữ liệu thu được để phân tích và đưa ra kết luận. Sử dụng các công cụ phân tích hiệu suất và biểu đồ để trình bày kết quả.
- Đánh giá và so sánh: Sử dụng các chỉ số hiệu suất như thời gian xử lý và tài nguyên sử dụng để so sánh và đánh giá các cơ chế xử lý.
- Đảm bảo tính chính xác và tin cậy: Lặp lại các thí nghiệm để đảm bảo tính nhất quán của kết quả. Sử dụng các phương pháp phân tích thống kê để xác nhận tính chính xác của các kết luận.

1.6. Những đóng góp mới của đòn tài

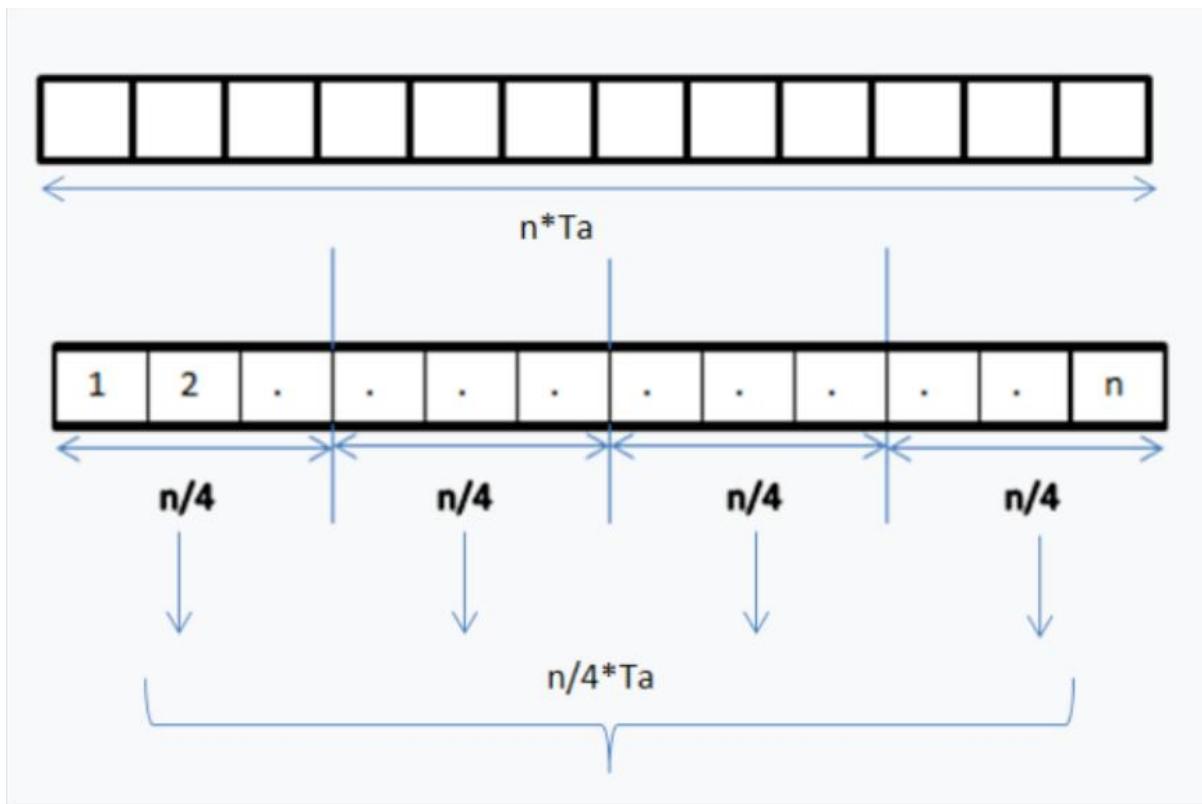
- Đánh giá toàn diện hiệu suất của các cơ chế xử lý dữ liệu trên ba nền tảng: xử lý tuần tự trên CPU (Python NumPy), xử lý song song trên GPU (CUDA, CuPy), và xử lý phân tán trên nhiều GPU (Apache Spark + RAPIDS cuDF).
- So sánh và phân tích ưu nhược điểm của từng nền tảng các bài toán cơ sở như cộng vector, histogram, Inverted Indexing, Matrix Multiplication, ... giúp xác định phương pháp phù hợp nhất cho từng bài toán.
- Đưa ra số liệu thực nghiệm về thời gian xử lý, mức sử dụng tài nguyên (CPU, GPU, bộ nhớ), và khả năng mở rộng khi tăng kích thước dữ liệu, giúp tối ưu hóa việc lựa chọn nền tảng xử lý.
- So sánh hiệu quả giữa xử lý song song trên GPU và xử lý phân tán trên nhiều GPU, làm rõ khi nào nên sử dụng CUDA, CuPy trên một GPU và khi nào nên chuyển sang Apache Spark + RAPIDS cuDF để khai thác sức mạnh của nhiều GPU.
- Đánh giá khả năng tận dụng tài nguyên phần cứng của từng phương pháp, giúp tối ưu hóa hiệu suất trên các hệ thống có cấu hình khác nhau, từ máy đơn lẻ (single-node) đến cụm máy tính (multi-node cluster).

CHƯƠNG 2: CƠ SỞ LÝ THUYẾT

2.1. Tổng quan cơ chế xử lý dữ liệu song song

2.1.1. Khái niệm

Xử lý song song là việc sử dụng nhiều bộ xử lý để thực hiện các tác vụ đồng thời, thường là bằng cách chia một vấn đề lớn thành các phần nhỏ hơn để giải quyết cùng một lúc. Điều này trái ngược với xử lý tuần tự, trong đó các lệnh được thực hiện một cách tuyến tính, từng lệnh một trên một bộ xử lý duy nhất [1]. Trong xử lý tuần tự, một tác vụ phải hoàn thành trước khi tác vụ tiếp theo bắt đầu. Ngược lại, xử lý song song cho phép nhiều phần của một tác vụ hoặc nhiều tác vụ khác nhau được thực hiện đồng thời, tận dụng nhiều tài nguyên tính toán [2].



Hình 2.1. So sánh cơ chế xử lý tuần tự và song song.

Dựa vào **hình 2.1** cho thấy một công việc song song dữ liệu trên một mảng N phần tử có thể được chia đều cho tất cả các bộ xử lý. Chúng ta hãy giả sử chúng ta muốn tính tổng tất cả các phần tử của mảng đã cho và thời gian cho một phép cộng duy nhất là đơn vị thời gian TA. Trong trường hợp thực hiện tuần tự, thời gian được thực hiện theo quy trình sẽ là thời gian $n \times TA$ vì nó phải thực hiện tính tổng tất cả các yếu tố của một mảng. Mặt khác, nếu chúng ta thực hiện công việc này dưới dạng công việc

song song dữ liệu trên 4 bộ xử lý, thời gian thực hiện sẽ giảm xuống $(n/4) \times TA +$ các đơn vị thời gian chi phí hợp nhất. Thực hiện song song dẫn đến tăng tốc 4 so với thực hiện tuần tự.

Các kiến trúc hiện nay cho tính toán song song bao gồm tính toán trên nhiều vi xử lý (cores) và việc sử dụng các card đồ họa (GPUs) [2]. Xử lý song song có thể xảy ra ở nhiều mức độ khác nhau, bao gồm mức độ bit (tăng số lượng bit dữ liệu được xử lý), mức độ lệnh (khả năng xử lý các lệnh trùng lặp dựa trên pipeline của bộ xử lý), mức độ dữ liệu (mỗi vi xử lý thực hiện cùng một tác vụ trên các phần dữ liệu khác nhau - SIMD), và mức độ nhiệm vụ (các vi xử lý khác nhau chạy các lệnh khác nhau trên các phần dữ liệu khác nhau - MIMD) [2].

2.1.2. Ưu điểm của xử lý dữ liệu song song

Có nhiều lý do và lợi ích khiến xử lý song song trở nên quan trọng và được ứng dụng rộng rãi trong các ứng dụng hiện nay:

- Tăng hiệu suất tính toán: Xử lý song song cho phép giải quyết các vấn đề phức tạp nhanh hơn nhiều so với xử lý tuần tự bằng cách chia công việc cho nhiều bộ xử lý [1]. Ví dụ, việc nhân hai ma trận có thể được thực hiện song song để giảm đáng kể thời gian tính toán [2].
- Xử lý lượng lớn dữ liệu (Big Data): Trong kỷ nguyên dữ liệu lớn, nhiều ứng dụng phải xử lý một lượng dữ liệu khổng lồ. Xử lý song song là một phương pháp hiệu quả để quản lý và phân tích dữ liệu này trong thời gian hợp lý [3]. Các kỹ thuật như MapReduce được thiết kế để xử lý song song trên các cụm máy tính lớn [4]. Các hệ thống quản lý dữ liệu chuyên sâu trong hệ thống song song cũng được nghiên cứu để xử lý hiệu quả lượng lớn dữ liệu [5].
- Cải thiện khả năng đáp ứng của ứng dụng: Trong các ứng dụng tương tác, xử lý song song có thể giúp thực hiện các tác vụ nền mà không làm chậm trải nghiệm người dùng.
- Tận dụng kiến trúc phần cứng đa lõi: Các bộ vi xử lý hiện đại thường có nhiều lõi (multi-core). Xử lý song song cho phép các ứng dụng tận dụng tối đa sức mạnh tính toán của các kiến trúc này [2]. GPU cũng là một kiến trúc song song mạnh mẽ, ban đầu được thiết kế cho đồ họa nhưng hiện được sử dụng rộng rãi cho các tác vụ tính toán song song nói chung thông qua các công nghệ như

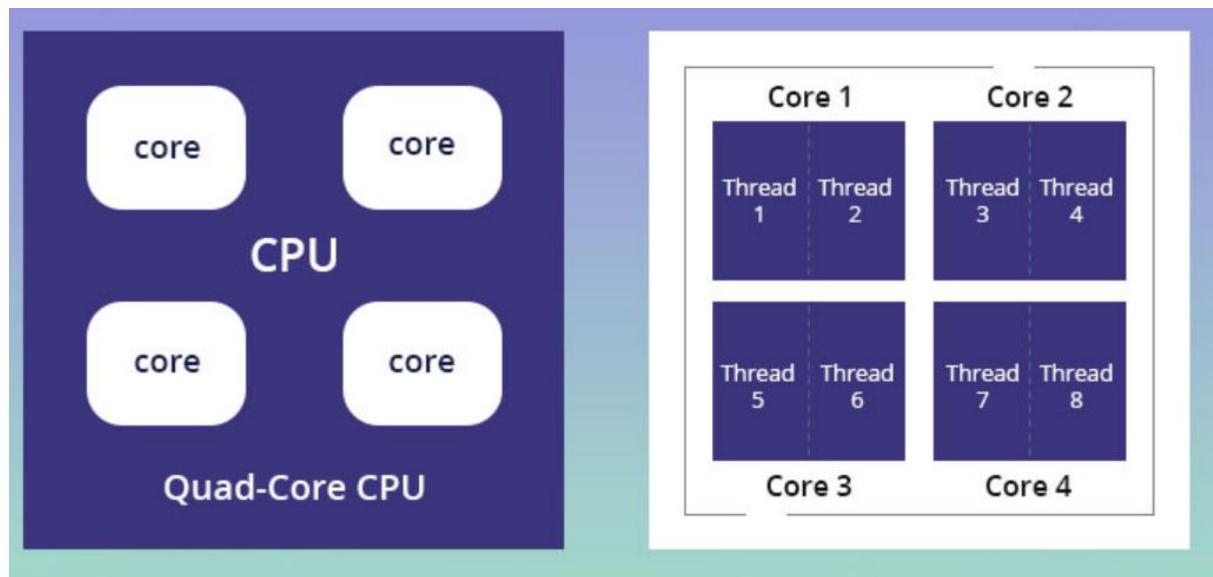
CUDA [2], GPU có nhiều băng thông bộ nhớ chính và hiệu quả về chi phí, năng lượng và kích thước so với CPU trong một số loại tác vụ [2].

- Đáp ứng yêu cầu của các ứng dụng chuyên biệt: Nhiều lĩnh vực như xử lý ảnh và video, trí tuệ nhân tạo, mô phỏng khoa học, và phân tích dữ liệu tài chính đòi hỏi khả năng tính toán song song để xử lý các tác vụ phức tạp và tốn nhiều thời gian [2] Ví dụ, trong xử lý ảnh, các thuật toán như adaptive image thresholding có thể được tăng tốc đáng kể bằng cách thực hiện song song trên GPU [6]. Trong tìm kiếm văn bản, các kiến trúc chuyên dụng như IIU (Inverted Index Processing Unit) được thiết kế để tối ưu hóa hiệu suất truy vấn song song trên các chỉ mục đảo ngược [7].
- Tính toán song song hướng dữ liệu (Data parallelism): Đây là một dạng xử lý song song phổ biến, trong đó cùng một tác vụ được thực hiện đồng thời trên các phần khác nhau của dữ liệu [1, 2]. Ví dụ, trong thuật toán MapReduce, hàm map được áp dụng song song trên các phần khác nhau của dữ liệu đầu vào [4]. Các kiến trúc như Connection Machine's CM-2 cũng là kiến trúc song song dữ liệu [5].

Tuy nhiên, cần lưu ý rằng việc lập trình và điều chỉnh cho các hệ thống xử lý song song có thể phức tạp hơn so với xử lý tuần tự, và hiệu quả của xử lý song song phụ thuộc vào bản chất của vấn đề và cách nó được song song hóa [2].

2.1.3. Các kiến trúc song song phổ biến

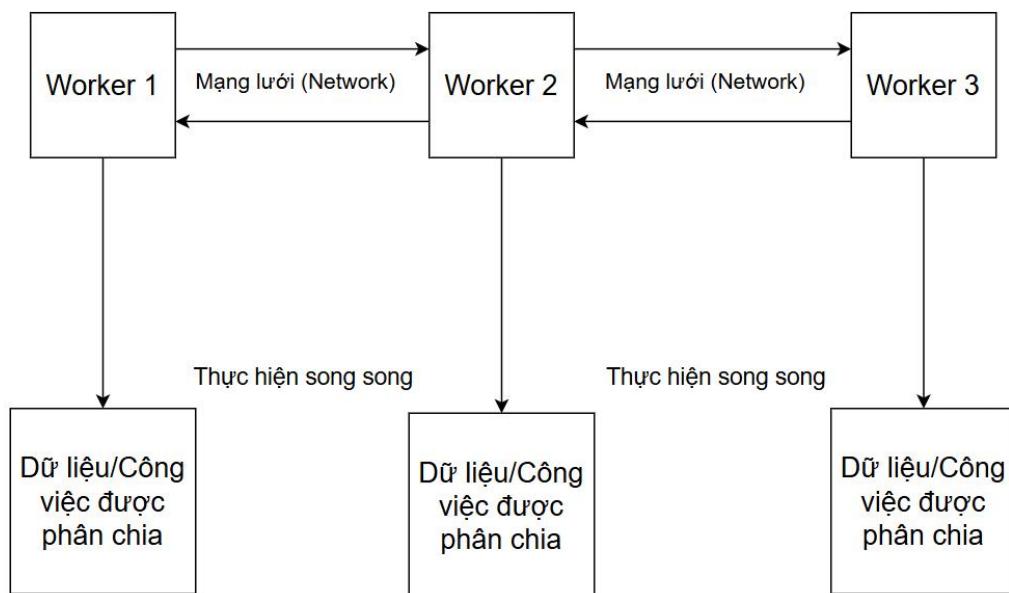
- Đa lõi (Multi-core): Các bộ vi xử lý hiện đại thường tích hợp nhiều lõi (cores) trên một chip duy nhất. Điều này cho phép thực hiện nhiều luồng (threads) hoặc quy trình (processes) đồng thời trên cùng một hệ thống, tận dụng khả năng song song hóa ở cấp độ chip [2]. **Hình 2.2** minh họa một bộ vi xử lý đa lõi thường có kiến trúc như sau:



Hình 2.2. Cấu trúc của CPU.

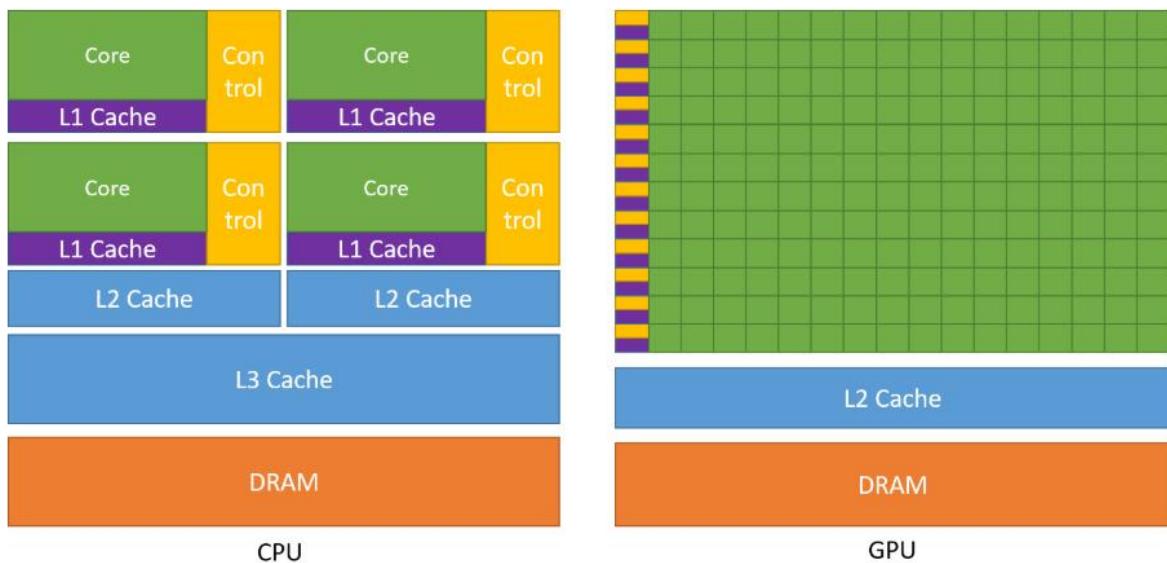
Thường thì CPU có số core và số thread giới hạn hơn GPU nhưng CPU vẫn có thể thực hiện được các quy trình xử lý song song do có nhiều hơn một core.

- Hệ thống phân tán (Distributed Systems): Các hệ thống này bao gồm nhiều nút tính toán độc lập (thường là các máy tính riêng biệt) được kết nối với nhau thông qua mạng [5]. Để tận dụng tính song song, công việc và dữ liệu cần được phân chia và phân tán trên các nút này (**Hình 2.3**). Ví dụ, hệ thống cơ sở dữ liệu song song PRISMA dựa trên kiến trúc shared-nothing, bao gồm 100 nút kết nối với nhau, trong đó một nửa số nút có đĩa riêng [5]. Các khung công việc như MapReduce cũng được thiết kế để xử lý song song lượng lớn dữ liệu trên các cụm máy tính lớn [2, 8].



Hình 2.3. Minh họa cơ chế MapReduce.

- Bộ xử lý đồ họa (Graphics Processing Units - GPUs), xem **hình 2.4**: Ban đầu được thiết kế cho xử lý đồ họa, GPU đã phát triển thành các bộ xử lý song song mạnh mẽ cho nhiều ứng dụng tính toán nói chung [2]. GPU có kiến trúc khác biệt so với CPU, với nhiều đơn vị xử lý nhỏ hơn được tối ưu hóa cho các tác vụ song song dữ liệu [6]. Các công nghệ như CUDA của NVIDIA cho phép lập trình viên khai thác sức mạnh tính toán song song của GPU cho các tác vụ không phải đồ họa [2], GPU có băng thông bộ nhớ chính lớn và hiệu quả về chi phí và năng lượng cho các ứng dụng có tính song song cao, khả năng sử dụng lại dữ liệu và tính đều đặn [2]. Ví dụ, việc thực hiện song song các thuật toán adaptive image thresholding trên GPU có thể đạt được tốc độ nhanh hơn đáng kể so với CPU [6].



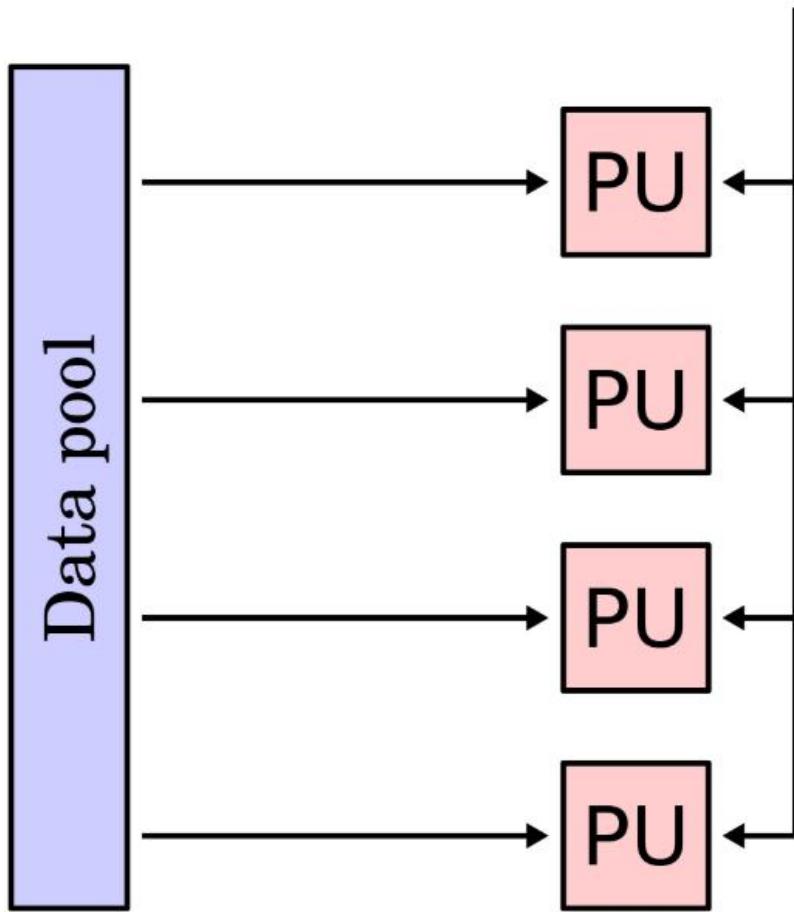
Hình 2.4. So sánh cấu trúc CPU và GPU

- Kiến trúc song song dữ liệu (Data Parallel Architectures): Các kiến trúc này được thiết kế đặc biệt để thực hiện cùng một tập lệnh trên nhiều phần dữ liệu khác nhau một cách đồng thời (SIMD - Single Instruction Multiple Data) [2]. Ví dụ, Connection Machine's CM-2 là một kiến trúc song song dữ liệu với hàng nghìn bộ xử lý 1-bit, mỗi bộ xử lý có bộ nhớ riêng.

2.1.4. Một số mô hình xử lý dữ liệu song song

SIMD (Single Instruction, Multiple Data) - Một Lệnh, Nhiều Dữ Liệu:

SIMD Instruction pool

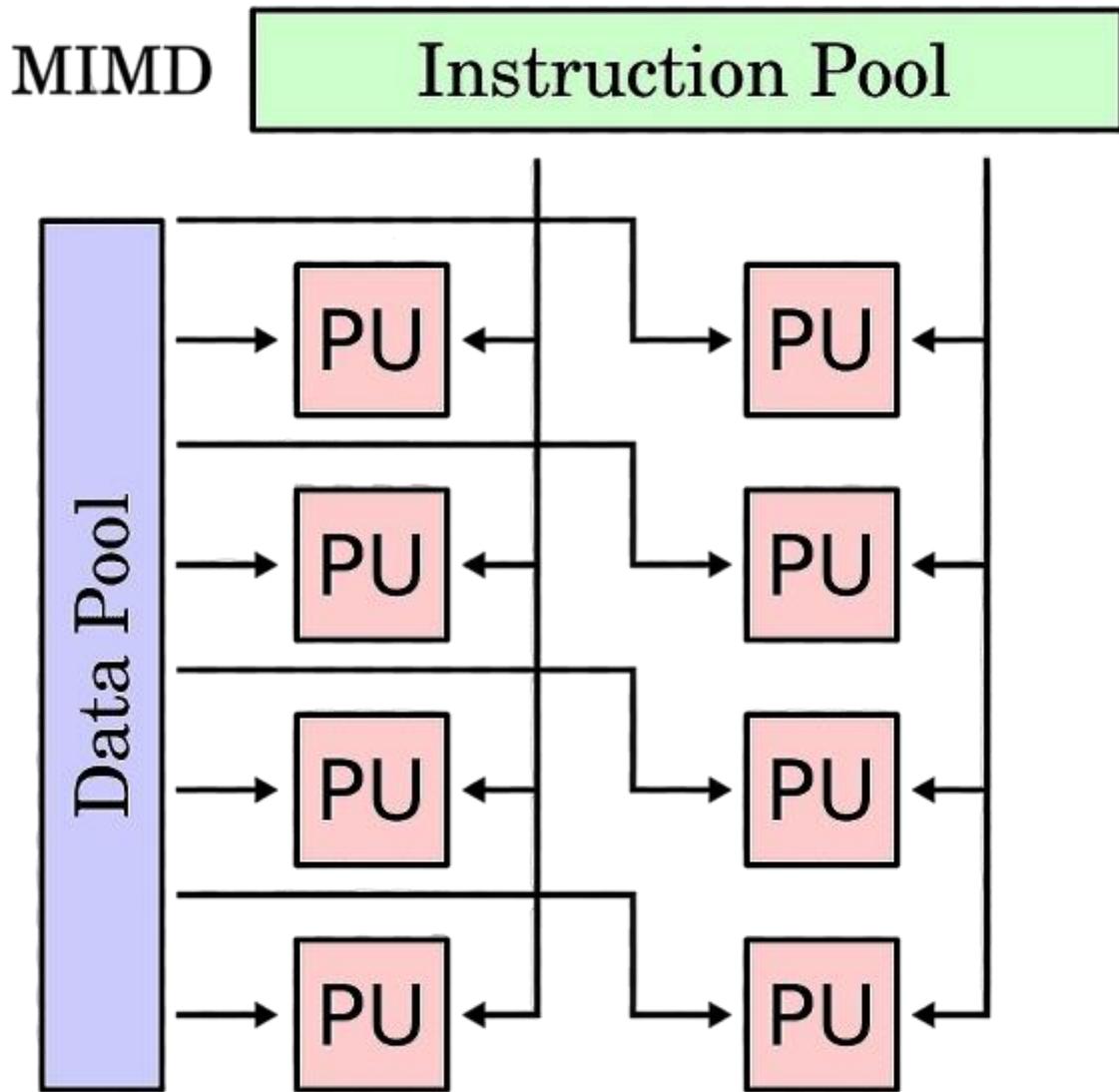


Hình 2.5. Mô hình SIMD.

Mô hình SIMD (**Hình 2.5**) thực hiện một lệnh duy nhất đồng thời trên nhiều phần tử dữ liệu khác nhau [2, 5]. Điều này có nghĩa là một bộ điều khiển sẽ gửi cùng một chỉ thị đến nhiều bộ xử lý (Instruction Pool gửi mũi tên lệnh đến tất cả các PU), và mỗi bộ xử lý sẽ thực hiện chỉ thị đó trên một phần dữ liệu riêng biệt [5] (Mỗi PU có một mũi tên kết nối đến Data Pool). Khái niệm về SIMD đã xuất hiện từ những năm 1960 với sự phát triển của máy Solomon [1]. Máy Solomon, còn được gọi là bộ xử lý vector, được phát triển để tăng tốc hiệu suất của các phép toán học bằng cách xử lý trên một mảng dữ liệu lớn, thực hiện các phép toán trên nhiều dữ liệu trong các bước thời gian liên tiếp [1]. Tính đồng thời của các phép toán dữ liệu cũng được khai thác bằng cách hoạt động trên nhiều dữ liệu cùng một lúc bằng một lệnh duy nhất. Các bộ xử lý này được gọi là 'array processors' (bộ xử lý mảng) [1]. Ngày nay, SIMD được thể hiện rõ nhất trong graphics processing units (GPU), vốn sử dụng cả hai kỹ thuật: hoạt động trên nhiều dữ liệu trong không gian và thời gian bằng một lệnh duy nhất [1, 6]. Các hệ

thống có cơ chế đồng bộ hóa ngầm định cho việc thực hiện các phép toán được gọi là hệ thống SIMD. Các hệ thống này cung cấp một bộ điều khiển duy nhất cho tất cả các bộ xử lý, và tất cả các bộ xử lý đều thực hiện cùng một lệnh nhưng trên các dữ liệu khác nhau. Ví dụ về các máy tính SIMD bao gồm CM-2, MasPar MP-1 và MP-2, và DAP610.

MIMD (Multiple Instruction, Multiple Data) - Nhiều Lệnh, Nhiều Dữ Liệu:

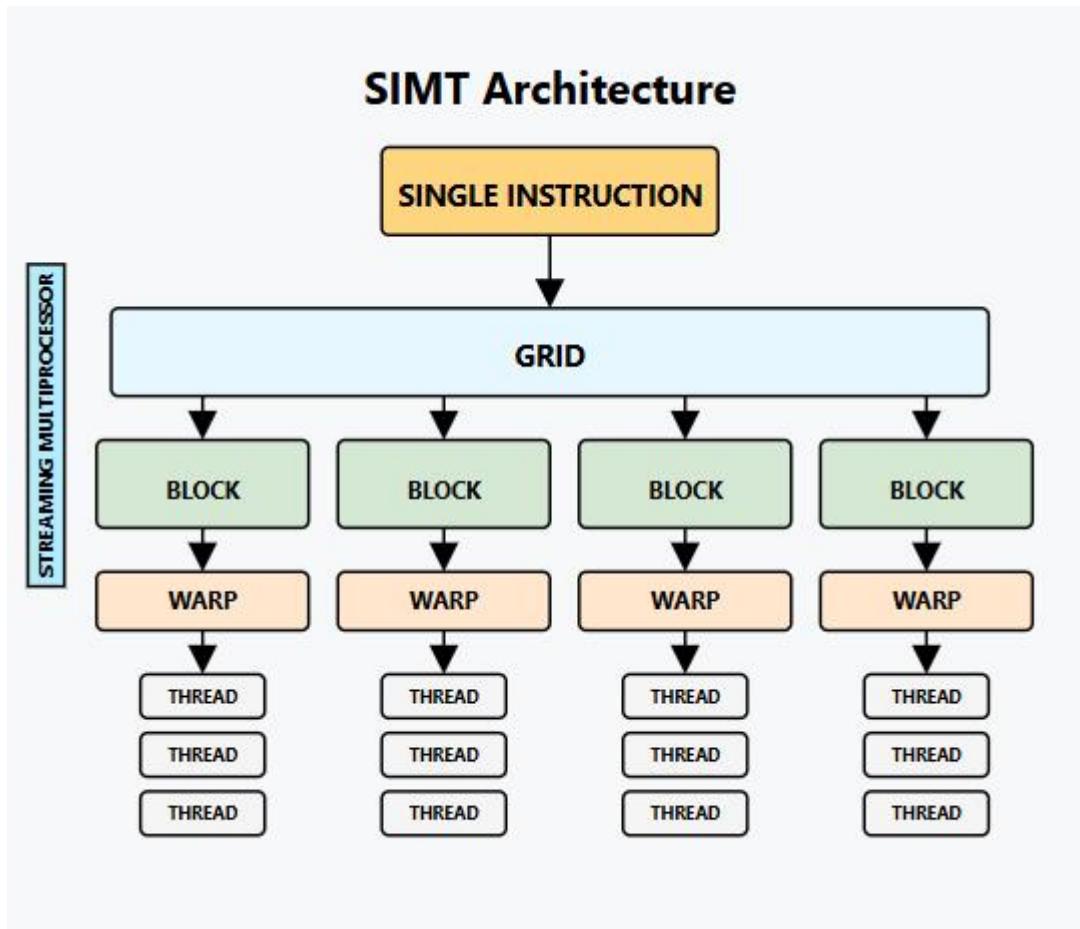


Hình 2.6. Mô hình MIMD.

Ngược lại với SIMD, mô hình MIMD (**Hình 2.6**) sử dụng nhiều bộ xử lý độc lập, trong đó mỗi bộ xử lý có thể thực hiện các lệnh khác nhau trên các tập dữ liệu khác nhau [2, 5]. Các máy tính MIMD thiếu một bộ điều khiển toàn cục; thay vào đó, mỗi cặp bộ xử lý-bộ nhớ sẽ thực hiện các lệnh một cách độc lập với nhau [5]. Việc đồng bộ hóa trong các hệ thống MIMD thường được thực hiện một cách tinh minh bởi lập

trình viễn thông qua việc sử dụng truyền thông điệp (message-passing), semaphores [5]. Ví dụ về các máy tính MIMD bao gồm Thinking Machines CM-5, Intel Paragon, KSR-1, nCube. Các hệ thống cơ sở dữ liệu song song đã được phát triển trên cả kiến trúc SIMD và MIMD [5]. Hệ thống đa lõi CPU, siêu máy tính và các hệ thống tính toán hiệu năng cao thường sử dụng mô hình MIMD.

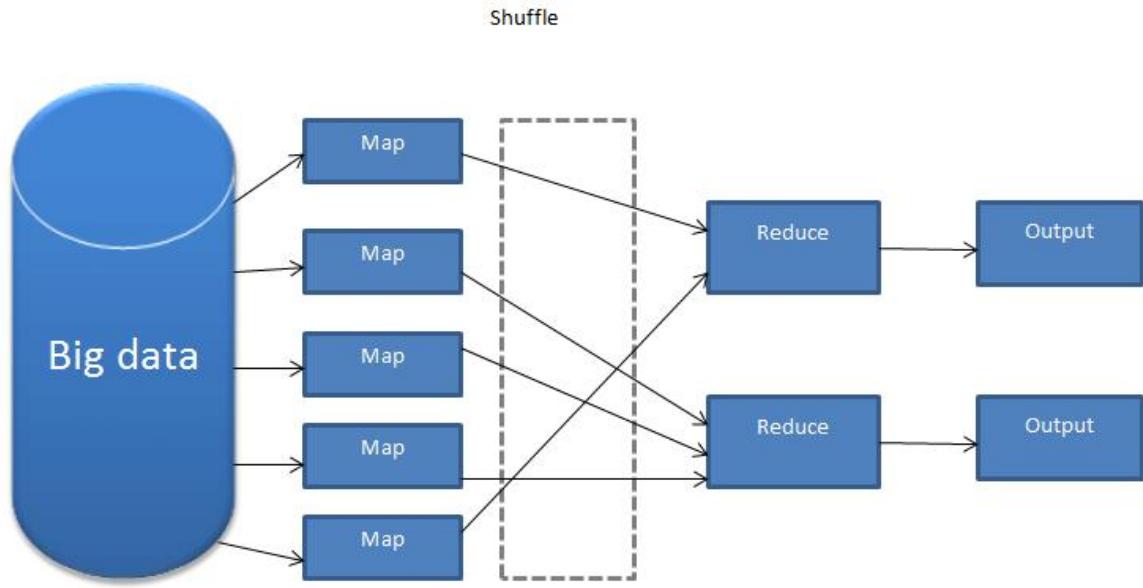
SIMT (Single Instruction, Multiple Threads) - Một Lệnh, Nhiều Luồng:



Hình 2.7. Mô hình SIMT.

Mô hình SIMT (Single Instruction, Multiple Threads), xem **hình 2.7** là kiến trúc xử lý song song được NVIDIA sử dụng trong CUDA để tối ưu hóa hiệu suất trên GPU [1]. SIMT có sự tương đồng với SIMD (Single Instruction, Multiple Data) vì nhiều luồng thực thi cùng một lệnh trên các phần dữ liệu khác nhau. Tuy nhiên, khác với SIMD, mỗi luồng trong SIMT có thể có trạng thái điều khiển riêng biệt, giúp tăng tính linh hoạt [2]. GPU được tổ chức thành các Streaming Multiprocessors (SM), trong đó mỗi SM chứa nhiều lõi xử lý hoạt động song song. GPU tổ chức luồng thành grid, block, và thread, trong đó mỗi block chứa nhiều thread, và các block có thể thực thi độc lập với nhau [3].

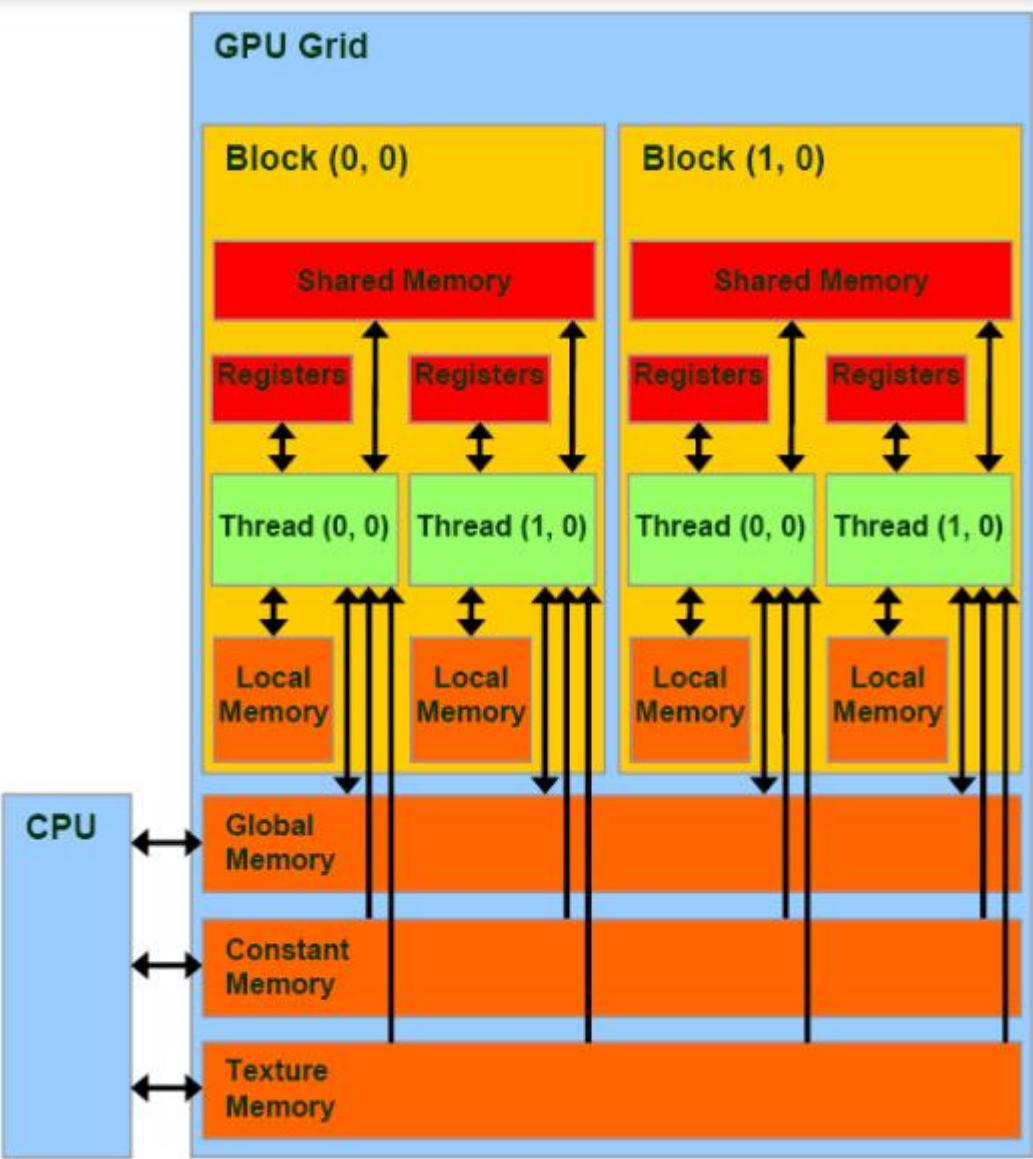
Mô hình MapReduce:



Hình 2.8. Mô hình MapReduce.

MapReduce là một mô hình lập trình được phát triển bởi Google, giúp đơn giản hóa việc xử lý song song dữ liệu lớn cho các tác vụ tổng hợp chung [4, 8]. Mặc dù thường được liên kết với xử lý dữ liệu phân tán trên các cụm máy tính, MapReduce cũng có yếu tố song song mạnh mẽ ở cấp độ thực hiện. Người dùng chỉ cần định nghĩa hai hàm chính: Map và Reduce, các tác vụ ở Map phase và Reduce phase đều được thực hiện 1 cách song song (**Hình 2.8**). Khung công việc MapReduce sẽ tự động xử lý các vấn đề phức tạp liên quan đến song song hóa như lên lịch quy trình, chịu lỗi, cân bằng tải, đồng bộ hóa, giảm thiểu giao tiếp và quản lý tính cục bộ của dữ liệu [4]. Giai đoạn Map xử lý song song các phần dữ liệu đầu vào, tạo ra các cặp khóa-giá trị trung gian. Giai đoạn Reduce sau đó cũng xử lý song song các giá trị có cùng khóa để tạo ra kết quả cuối cùng [8].

Kiến trúc CUDA (Compute Unified Device Architecture):

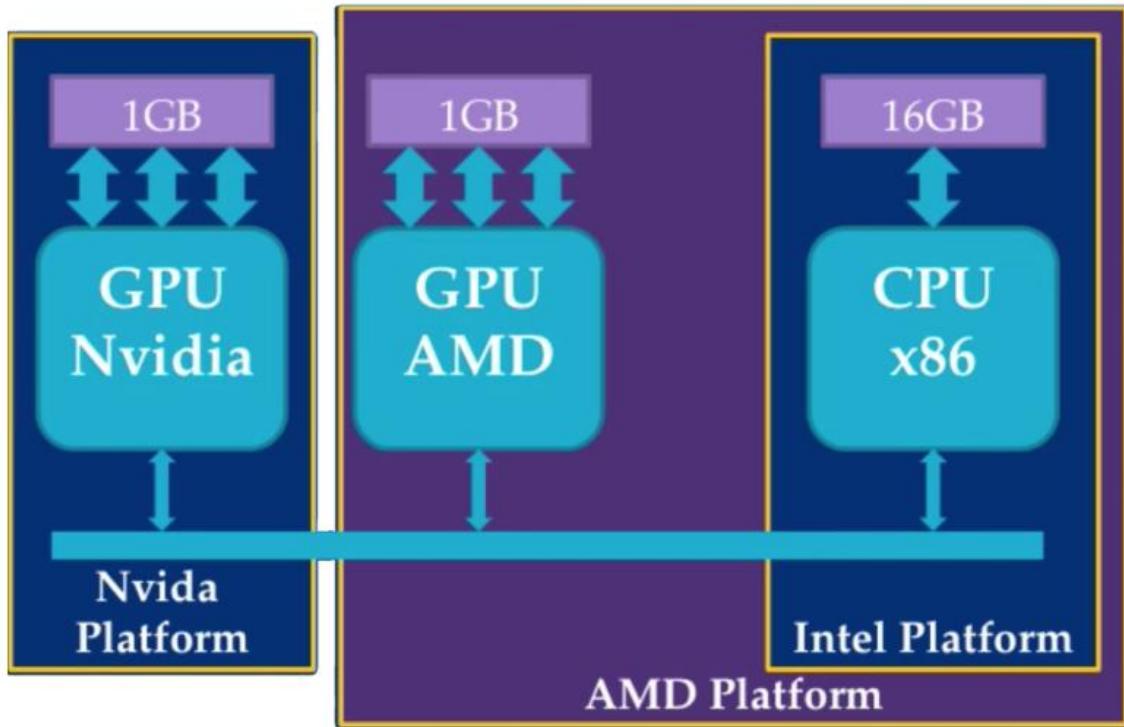


Hình 2.9. Kiến trúc CUDA.

CUDA là một nền tảng điện toán song song và mô hình lập trình được phát triển bởi NVIDIA, cho phép các kỹ sư phần mềm sử dụng các đơn vị tính toán của GPU cho mục đích xử lý đa năng (GPGPU) [1, 2, 3, 6]. CUDA là một công nghệ cho phép thực thi mã trên GPU để xử lý song song, được giới thiệu vào năm 2006 và yêu cầu card đồ họa NVIDIA GeForce 8 series trở lên [2]. CUDA hỗ trợ các ngôn ngữ lập trình như C, C++, Fortran, Matlab, Python, LabView, và cung cấp các thư viện cho các tác vụ như FFT, BLAS, tạo số ngẫu nhiên (CURAND), và xử lý ma trận thưa (CUSPARSE) [2]. Mô hình lập trình CUDA bao gồm việc chạy các kernel (chương trình chạy trên GPU) được tổ chức thành các grid chứa các block, với các thread chạy song song để xử lý tập lệnh (**Hình 2.9**). Các thread trong cùng một block có thể tương tác với nhau thông qua bộ nhớ chia sẻ (shared memory) và đồng bộ hóa [2, 3]. CUDA cung cấp API để

quản lý bộ nhớ giữa CPU (host) và GPU (device), bao gồm việc cấp phát bộ nhớ (cudaMalloc, cudaMallocHost), sao chép dữ liệu (cudaMemcpy), và giải phóng bộ nhớ (cudaFree, cudaFreeHost).

OpenCL (Open Computing Language):



Hình 2.10. Mô hình nền tảng OpenCL.

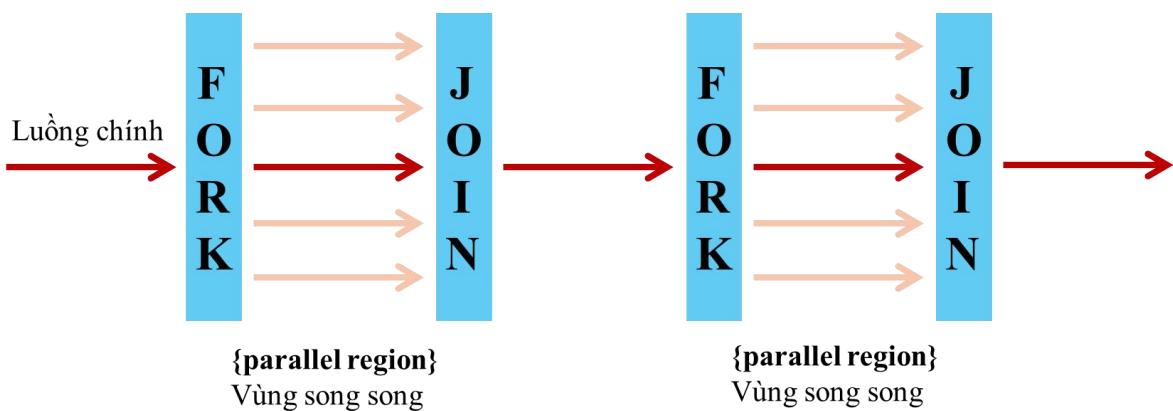
OpenCL là một tiêu chuẩn mở, đa nền tảng cho phép lập trình song song trên nhiều loại phần cứng khác nhau (**Hình 2.10**), bao gồm GPUs, CPUs, FPGAs và các bộ xử lý khác [1, 2, 3]. Tương tự như CUDA, OpenCL cung cấp một mô hình để viết các kernel có thể chạy song song trên các thiết bị tính toán khác nhau. OpenCL hướng đến tính di động cao hơn CUDA, vì nó không bị giới hạn bởi phần cứng của một nhà sản xuất cụ thể [1]. Nó cho phép các nhà phát triển tận dụng sức mạnh tính toán song song của nhiều loại thiết bị trong một hệ thống hỗn hợp [2].

Tóm lại, các mô hình xử lý dữ liệu song song này đóng vai trò nền tảng trong việc thiết kế và triển khai các hệ thống tính toán hiệu năng cao và xử lý dữ liệu lớn hiện đại. Việc lựa chọn mô hình phù hợp phụ thuộc vào đặc điểm của bài toán, kiến trúc phần cứng có sẵn và yêu cầu về hiệu suất.

2.1.5. Một số công nghệ xử lý dữ liệu song song (Công cụ và ngôn ngữ lập trình hỗ trợ)

a) OpenMP

OpenMP là một API hỗ trợ lập trình song song trên các hệ thống đa lõi (multi-core) và bộ xử lý đa luồng (multi-threading). Nó giúp lập trình viên dễ dàng triển khai tính toán song song trên kiến trúc bộ nhớ chia sẻ (shared memory). OpenMP được sử dụng rộng rãi trong các ngôn ngữ lập trình như C, C++ và Fortran để tăng tốc xử lý các tác vụ có thể thực hiện song song.



Hình 2.11. Mô hình Fork-Join của OpenMP.

Theo **hình 2.11** - OpenMP hoạt động theo mô hình Fork-Join, trong đó:

- Fork: Một luồng chính (master thread) tạo ra nhiều luồng con (worker threads) để thực hiện song song một đoạn mã.
- Join: Sau khi hoàn thành công việc, các luồng con kết thúc và hợp nhất vào luồng chính.

Quá trình này giúp tận dụng tối đa sức mạnh của CPU đa lõi để thực hiện các tác vụ tính toán nhanh hơn.

b) CUDA

CUDA (Compute Unified Device Architecture) là một nền tảng tính toán song song và mô hình lập trình do NVIDIA phát triển. Nó cho phép các lập trình viên tận dụng sức mạnh của GPU (Graphics Processing Unit) để thực hiện các tác vụ tính toán phức tạp. Thay vì chỉ xử lý đồ họa, GPU có thể được sử dụng để tăng tốc các ứng dụng trong nhiều lĩnh vực như trí tuệ nhân tạo, khoa học dữ liệu, mô phỏng vật lý và thị giác máy tính.

Khác với CPU, vốn có số lượng lõi ít, GPU có hàng ngàn lõi nhỏ hoạt động song song, giúp xử lý nhiều luồng dữ liệu cùng lúc. Điều này đặc biệt hữu ích đối với các bài toán

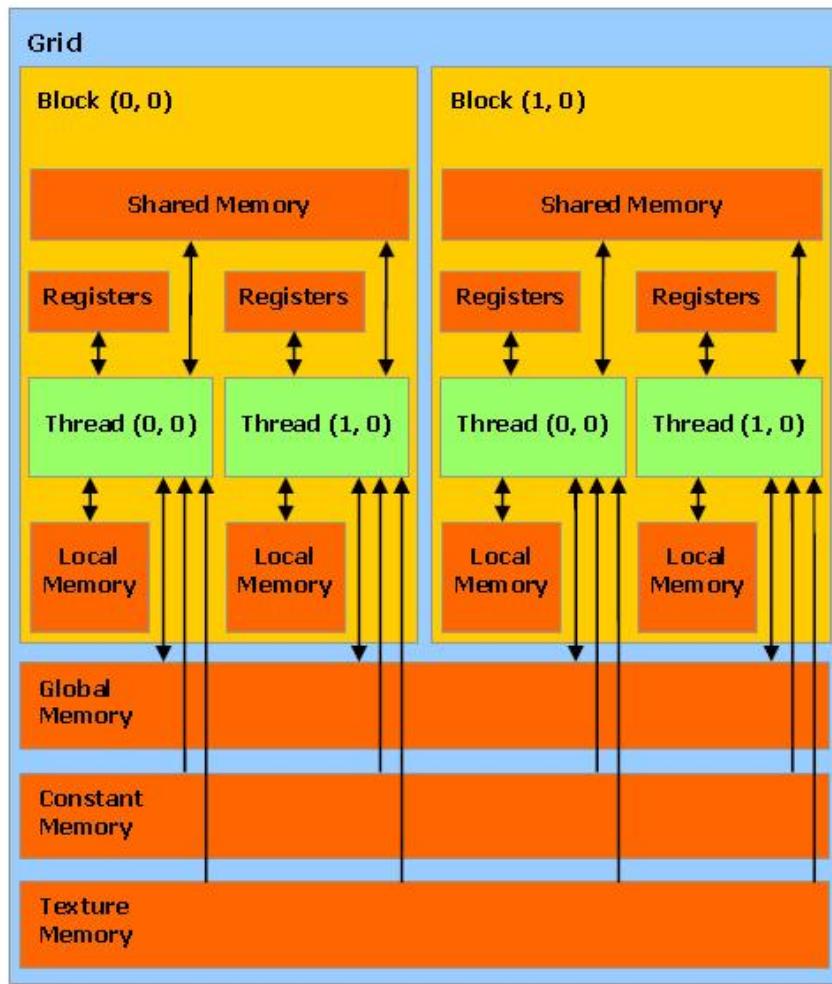
yêu cầu xử lý khối lượng dữ liệu lớn và có thể chia nhỏ thành nhiều phần để tính toán đồng thời. Chính vì thế, CUDA khai thác sức mạnh này bằng cách cho phép lập trình viên viết mã sử dụng các lõi GPU để thực hiện các tác vụ tính toán nặng, giảm tải cho CPU và tăng tốc hiệu suất xử lý.

CUDA cung cấp một môi trường lập trình mở rộng để phát triển các ứng dụng chạy trên GPU. Nó bao gồm:

- Kernel: Là một hàm chạy trên GPU, có thể thực hiện hàng ngàn luồng tính toán song song.
- Grid và Block: CUDA tổ chức các luồng tính toán thành một cấu trúc lưới (grid) chứa nhiều khối (block), trong mỗi block lại chứa nhiều luồng (thread).
- Bộ nhớ CUDA: Bao gồm bộ nhớ toàn cục (global memory), bộ nhớ chia sẻ (shared memory), bộ nhớ cục bộ (local memory) và bộ nhớ hằng số (constant memory). Việc tối ưu hóa truy cập bộ nhớ giúp tăng hiệu suất đáng kể.

Đặc điểm chính của CUDA:

- CUDA là một mở rộng của ngôn ngữ C, do đó, các lập trình viên đã quen thuộc với C/C++ có thể dễ dàng tiếp cận và sử dụng.
- Mã CUDA được chia thành hai phần chính:
 - Phần chạy trên CPU (Host) chịu trách nhiệm quản lý luồng điều khiển và giao tiếp với GPU.
 - Phần chạy trên GPU (Device), hay còn gọi là kernel, có khả năng thực thi song song hàng nghìn luồng độc lập. Mỗi luồng được gán một chỉ số định danh để xác định nhiệm vụ cụ thể của nó.
- CUDA cung cấp cơ chế linh hoạt cho phép lập trình viên tự do quyết định số lượng luồng chạy song song mà không phụ thuộc vào phần cứng cụ thể. Tuy nhiên, các luồng này phải được nhóm thành từng Block, với số lượng không vượt quá 512 luồng mỗi block. Cách tổ chức này giúp mã CUDA có thể mở rộng và chạy hiệu quả trên nhiều loại GPU khác nhau mà không cần thay đổi cấu trúc chương trình.



Hình 2.12. Mô hình bộ nhớ CUDA.

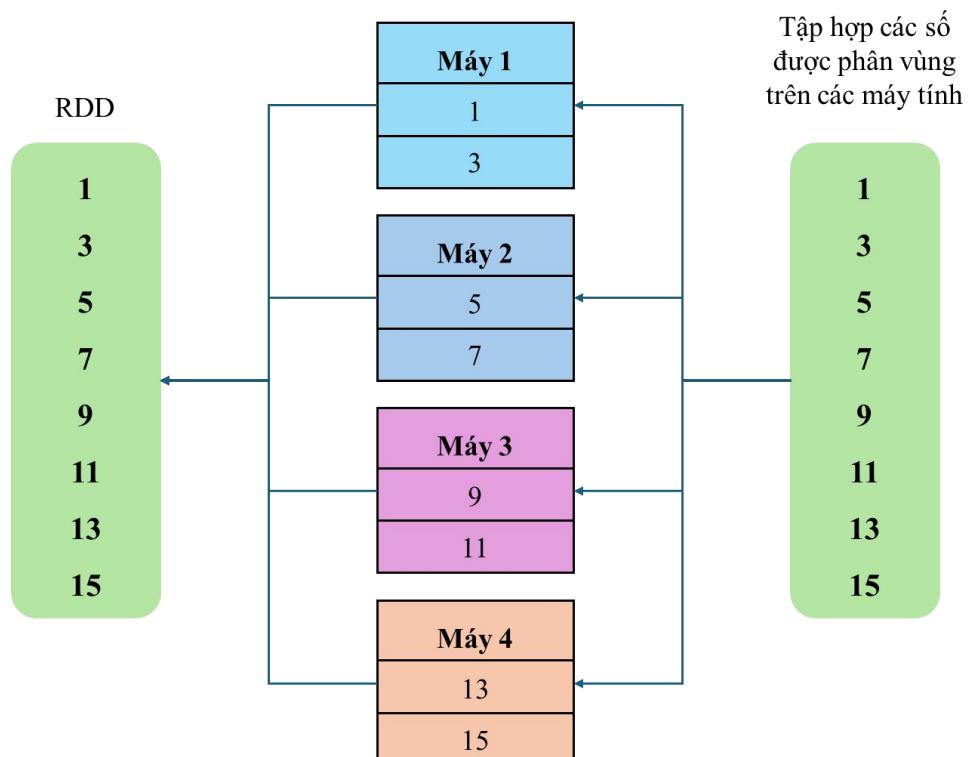
CUDA cung cấp hàm cho phép cấp phát bộ nhớ động trên từng loại bộ nhớ cụ thể, Bộ nhớ trong CUDA được chia thành nhiều cấp như **hình 2.12**, bao gồm:

- **Bộ nhớ chính (Host Memory):** Thuộc CPU, chỉ có phần mã chạy trên CPU mới có thể truy cập và thay đổi dữ liệu tại đây.
- **Bộ nhớ toàn cục GPU (Global Memory):** Là bộ nhớ chung mà tất cả các luồng GPU đều có thể truy cập. Dữ liệu từ CPU có thể được chuyển vào đây thông qua các hàm thư viện CUDA. Thông thường, bộ nhớ này dùng để lưu dữ liệu đầu vào và đầu ra của các luồng chạy song song.
- **Bộ nhớ chia sẻ (Shared Memory):** Là vùng bộ nhớ dùng chung nhưng chỉ trong phạm vi một block. Do được đặt ngay trên chip xử lý, nó có tốc độ truy cập nhanh hơn nhiều so với bộ nhớ toàn cục. Loại bộ nhớ này thường được sử dụng để lưu dữ liệu tạm thời nhằm tối ưu hiệu suất tính toán.
- **Bộ nhớ cục bộ GPU (Local Memory):** Là bộ nhớ riêng của từng luồng GPU, dùng để lưu trữ các biến cục bộ và không thể truy cập từ luồng khác.

- Bộ nhớ hằng số (Constants Memory): Là bộ nhớ dành riêng cho các giá trị không thay đổi trong quá trình thực thi của kernel. Bộ nhớ này giúp giảm thiểu băng thông truy cập so với Global Memory, nhờ vào cơ chế cache tối ưu của GPU.
- Bộ nhớ Texture (Texture Memory): là một loại bộ nhớ đặc biệt trong CUDA, ban đầu được thiết kế cho đồ họa, nhưng cũng có thể tận dụng để tối ưu một số bài toán tính toán hiệu năng cao. Nó có một cơ chế truy cập dữ liệu hiệu quả nhờ vào bộ nhớ cache đặc biệt giúp tăng tốc các truy vấn dữ liệu có tính địa phương (spatial locality).

c) Apache Spark

Mặc dù Spark thường được nhắc đến trong xử lý dữ liệu phân tán, nhưng nó cũng có cơ chế xử lý song song mạnh mẽ bằng cách sử dụng Resilient Distributed Dataset (RDD) để thực hiện tính toán trên nhiều phần tử dữ liệu cùng lúc.



Hình 2.13. Apache Spark sử dụng RDD để xử lý song song.

Như **hình 2.13** – Spark sử dụng RDD để xử lý song song như sau:

- RDD (Resilient Distributed Dataset) là tập hợp dữ liệu lớn được phân tán trên nhiều máy trong cụm (cluster).
- Dữ liệu được chia thành nhiều partition và được phân phối trên nhiều máy khác nhau.

- Các máy (Machine 1, 2, 3, 4) xử lý các partition của dữ liệu song song.
- Khi thực hiện tính toán trên RDD, Spark sẽ thực hiện xử lý trên từng partition của RDD song song, giúp tận dụng tối đa tài nguyên cụm.

Apache Spark là một công cụ phân tích nhanh, đa năng để xử lý dữ liệu quy mô lớn chạy trên YARN, Apache Mesos, Kubernetes, nằm độc lập hoặc trên dịch vụ đám mây. Với các toán tử và thư viện cấp cao dành cho SQL, xử lý luồng (stream processing), học máy và xử lý đồ thị, Spark giúp dễ dàng xây dựng các ứng dụng song song trong Scala, Python, R hoặc SQL bằng cách sử dụng interactive shell, notebook hoặc ứng dụng đóng gói (packaged applications). Spark hỗ trợ phân tích theo batch và tương tác bằng cách sử dụng mô hình lập trình chức năng và công cụ truy vấn liên quan Catalyst, chuyển đổi công việc thành kế hoạch và lịch hoạt động bên trong truy vấn trên các node trong một cụm.

Ngoài công cụ xử lý dữ liệu Spark, còn có các thư viện dành cho SQL và DataFrames, học máy, GraphX, tính toán biểu đồ và xử lý luồng. Các thư viện này có thể được sử dụng cùng nhau trên các tập dữ liệu lớn từ nhiều nguồn dữ liệu khác nhau, chẳng hạn như HDFS, Alluxio, Apache Cassandra, Apache HBase hoặc Apache Hive.

2.2. Tổng quan về CUDA, Rapids, Cupy

2.2.1. Tổng quan về CUDA Numba

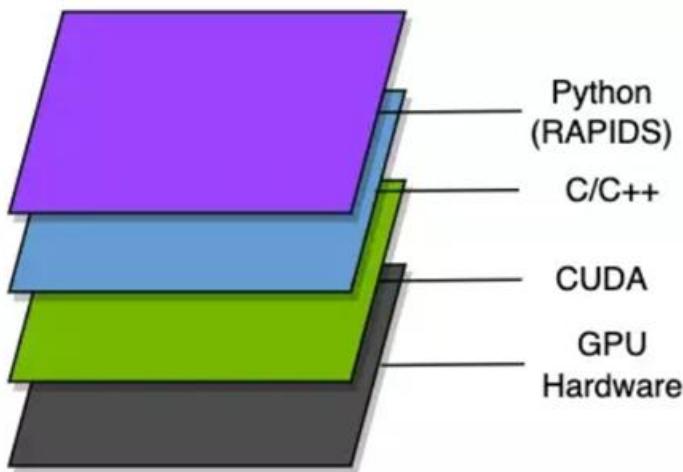
Numba hỗ trợ lập trình CUDA GPU bằng cách biên dịch trực tiếp một tập con hạn chế của mã Python thành các kernel và hàm thiết bị CUDA theo mô hình thực thi của CUDA.

Một tính năng đáng chú ý giúp việc viết các kernel GPU trở nên đơn giản hơn là Numba làm cho nó có vẻ như kernel có thể truy cập trực tiếp các mảng NumPy. Các mảng NumPy được truyền vào như các đối số của kernel sẽ tự động được chuyển giữa CPU và GPU (mặc dù điều này cũng có thể là một vấn đề trong một số trường hợp). Tuy nhiên, Numba vẫn chưa triển khai đầy đủ API của CUDA, do đó một số tính năng không có sẵn.

Những tính năng chính của numba là:

- Sinh code ngay lập tức (runtime hoặc khi import)
- Sinh code native cho CPU và GPU
- Tích hợp với các thư viện tính toán khoa học của Python như Numpy

2.2.2. Tổng quan về RAPIDS



Hình 2.14. Tổng quan về Rapids.

Rapids là một bộ thư viện mã nguồn mở được phát triển bởi NVIDIA, như **hình 2.14** nó dùng CUDA cho backends để tối ưu hóa tốc độ tính toán trên GPU. Thay vì thực hiện các thao tác dữ liệu trên CPU, RAPIDS tận dụng GPU để xử lý nhanh hơn, đặc biệt là với khối lượng dữ liệu lớn. Đây là một giải pháp hiệu quả cho các bài toán liên quan đến dữ liệu lớn và học máy, mang lại tốc độ tính toán vượt trội so với cách xử lý truyền thống trên CPU.

Rapids bao gồm nhiều thư viện với các chức năng cụ thể:

- cuDF: Thư viện hỗ trợ thao tác dữ liệu dạng bảng, tương tự như Pandas trên CPU, nhưng chạy trên GPU. Người dùng có thể áp dụng các thao tác xử lý dữ liệu như lọc, gộp nhóm, và tính toán nhanh chóng nhờ GPU.
- cuML: Thư viện học máy, tương tự như Scikit-Learn nhưng được tăng tốc trên GPU. Nó cung cấp các thuật toán học máy phổ biến như hồi quy, phân loại, cụm và giảm chiều dữ liệu, giúp xử lý nhanh hơn nhiều lần so với CPU.
- cuGraph: Thư viện hỗ trợ xử lý các bài toán đồ thị trên GPU, ví dụ như phân tích mạng, tìm đường đi ngắn nhất và các thuật toán đồ thị khác.
- cuPY: Thư viện tính toán mảng, tương tự như NumPy nhưng chạy trên GPU, hỗ trợ các thao tác tính toán ma trận và mảng.

2.2.3. Tổng quan về CuPy

CuPy là một thư viện mã nguồn mở của Python, được thiết kế để cung cấp các phép tính toán khoa học trên GPU bằng cách sử dụng CUDA (Compute Unified Device Architecture) của NVIDIA. CuPy có cú pháp và cách sử dụng tương tự như thư viện

NumPy, nhưng thay vì sử dụng CPU, CuPy thực hiện các phép tính trên GPU, giúp tăng tốc độ xử lý, đặc biệt là đối với các tác vụ tính toán nặng.

Cupy có các đặc điểm chính như:

- Cú pháp tương tự như NumPy: CuPy cung cấp các hàm và cú pháp tương tự như NumPy, do đó, các lập trình viên đã quen thuộc với NumPy có thể chuyển sang CuPy một cách dễ dàng để tận dụng sức mạnh của GPU.
- Tăng tốc tính toán: Với GPU, CuPy cho phép tăng tốc đáng kể cho các phép toán ma trận, xử lý mảng lớn, và các tính toán khoa học phức tạp, đặc biệt hữu ích trong học sâu (deep learning) và các ứng dụng cần hiệu suất cao.
- Tích hợp CUDA: CuPy được xây dựng dựa trên CUDA của NVIDIA, điều này có nghĩa là CuPy chỉ hoạt động trên các GPU của NVIDIA (không hỗ trợ GPU của AMD hay Intel).
- Hỗ trợ nhiều hàm toán học và xử lý mảng: Giống như NumPy, CuPy cung cấp nhiều hàm toán học và xử lý mảng như phép cộng, trừ, nhân, chia, tích vô hướng, phép nhân ma trận, đại số tuyến tính, xử lý FFT (Fast Fourier Transform), xử lý ngẫu nhiên, và các phép toán khác.
- Khả năng tích hợp: CuPy có thể được tích hợp với các thư viện học máy và học sâu khác như Chainer và PyTorch. Một số framework học sâu hiện đại (như Chainer) đã sử dụng CuPy để tăng tốc các phép tính trên GPU.

CHƯƠNG 3: CÁC BÀI TOÁN CƠ SỞ

3.1. Bài toán cộng vector

Trong hành trình khám phá các bài toán cơ sở, bài toán “cộng vectơ” xuất hiện như một bài tập khởi đầu nhưng lại mang nhiều ý nghĩa. Hãy tưởng tượng rằng bạn là giảng viên về khoa học dữ liệu đang phân tích điểm số của các học sinh qua các bài kiểm tra khác nhau. Mỗi học sinh có điểm số trong từng môn học được lưu dưới dạng các vectơ, và công việc đầu tiên của bạn là cộng các vectơ đó để tính tổng điểm, từ đó đưa ra những nhận định ban đầu về hiệu suất học tập. Chính từ đây, chúng em bắt đầu với bài toán cộng vectơ.

3.1.1. Mô tả bài toán

Trong không gian toán học, một vectơ là một đối tượng có hướng và độ lớn. Vectơ có thể được biểu diễn dưới dạng một bộ số có thứ tự, chẳng hạn như:

$$\vec{a} = (a_1, a_2, \dots, a_n) \in \mathbb{R}^n.$$

Phép cộng hai vectơ là một phép toán cơ bản trong đại số tuyến tính, được định nghĩa như sau:

Cho hai vectơ $\vec{a} = (a_1, a_2, \dots, a_n)$ và $\vec{b} = (b_1, b_2, \dots, b_n)$, ta có:

$$\vec{a} + \vec{b} = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$$

Một cách trực quan hơn, ta có thể hình dung phép cộng hai vectơ theo quy tắc “nối đuôi”:

- Giả sử điểm A là gốc tọa độ, vectơ \vec{a} được biểu diễn bởi đoạn thẳng có hướng từ A đến điểm B.
- Từ điểm B, vectơ \vec{b} được biểu diễn bởi đoạn thẳng có hướng từ B đến điểm C.
- Khi đó, đoạn thẳng nối từ A-> C chính là vectơ tổng $\vec{a} + \vec{b}$

Những vectơ \vec{a} và \vec{b} có thể đại diện cho các dữ liệu thực nghiệm như các tín hiệu số hay các thông số đo lường. Việc cộng từng phần tử của hai vectơ nhằm mục đích hợp nhất các thông tin đầu vào, tạo điều kiện cho các bước xử lý tiếp theo trong chuỗi phân tích dữ liệu.

3.1.2. Ý tưởng

Ý tưởng thực hiện bài toán cộng vectơ trên CPU dựa trên nguyên tắc xử lý tuần tự, như **hình 3.1**, mỗi phần tử của hai vectơ được cộng với nhau theo cách độc lập. Quá trình thực hiện có thể được chia thành các bước chính như sau:

- Khởi tạo và chuẩn bị dữ liệu:

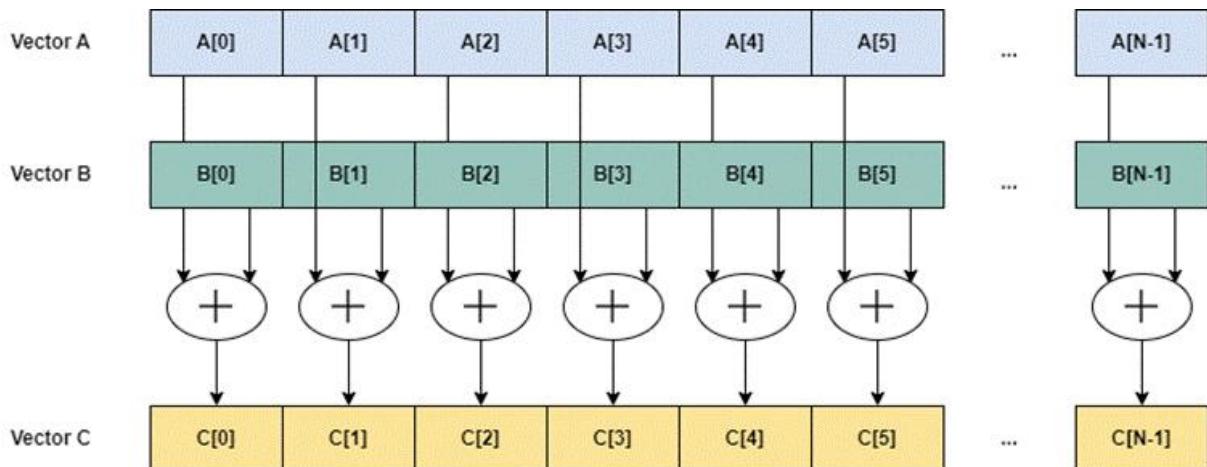
Trước tiên, ta khởi tạo hai vectơ a và b chứa các giá trị số học cần thiết. Việc khởi tạo này đảm bảo dữ liệu đầu vào được sắp xếp theo thứ tự và có kích thước đồng nhất.

- Thực hiện phép cộng:

Trên CPU, phép cộng được thực hiện thông qua một vòng lặp duyệt qua các chỉ số từ 0 đến $n - 1$. Với mỗi chỉ số i , chương trình thực hiện phép tính $c[i] = a[i] + b[i]$. Vì mỗi phép tính cộng là một phép toán cơ bản, nên chúng ta sử dụng cấu trúc lặp đơn giản để thực hiện toàn bộ quá trình này.

- Thu thập kết quả:

Sau khi duyệt qua hết các phần tử, vectơ c chứa kết quả của phép cộng được thu thập và sử dụng cho các bước xử lý dữ liệu tiếp theo.



Hình 3.1. Ý tưởng thực hiện cộng hai vectơ.

Quá trình cộng vectơ như **hình 3.1**, mặc dù đơn giản nhưng lại đóng vai trò quan trọng trong việc hiểu rõ nguyên lý hoạt động của các thuật toán xử lý mảng trên CPU, từ đó tạo cơ sở để chuyển sang những phương pháp tối ưu hơn như xử lý song song trên GPU.

3.1.3. Cộng Vectơ trên CPU

Trong kiến trúc máy tính truyền thống, CPU được thiết kế với số lượng lõi xử lý hạn chế và hoạt động chủ yếu theo mô hình xử lý tuần tự. Do đó khi thực hiện các phép toán cơ sở như cộng hai vectơ, CPU sẽ duyệt qua từng phần tử một cách tuần tự, gây

ra sự chậm trễ rõ rệt khi kích thước của vectơ (n) tăng lên. Dù cách tiếp cận này đơn giản và dễ hiểu, nhưng nó không còn khả năng đáp ứng hiệu quả đối với các ứng dụng đòi hỏi xử lý dữ liệu lớn hoặc tính toán thời gian thực.

Cụ thể, bài toán cộng vectơ trên CPU được thực hiện theo các bước sau:

- Khởi tạo dữ liệu:

Hai vectơ A và B gồm n phần tử được khởi tạo và lưu trữ trong bộ nhớ chính. Các giá trị của A và B có thể đại diện cho các thông số đo lường hoặc tín hiệu số thu thập được từ các hệ thống thực tế.

- Thực hiện phép cộng tuần tự:

Với mỗi chỉ số i từ 0 đến $n - 1$, CPU thực hiện phép tính cộng:

$$C[i] = A[i] + B[i]$$

Việc này được thực hiện trong mỗi vòng lặp, nơi mỗi bước thực hiện một phép cộng cho một phần tử duy nhất. Do CPU xử lý các thao tác này theo chuỗi, nên thời gian tính toán tổng hợp tăng đáng kể khi n lớn.

- Thu thập kết quả:

Sau khi hoàn thành vòng lặp, kết quả tổng được lưu vào vectơ C và sẵn sàng cho các bước xử lý tiếp theo hoặc phân tích dữ liệu.

Để minh họa cho các bước trên, mã giả dưới đây mô tả cách thức thực hiện trên CPU:

```
Thuật toán cộng 2 vectơ trên CPU
Input: Hai vectơ A, B với kích cỡ N
Output: Vectơ C với C[i] = A[i] + B[i]
1.Khởi tạo vectơ C với kích cỡ N
2.i <- 1
3.while i <= n do
4.    C[i] <- A[i] + B[i]
5.    i <- i + 1
6.Trả về C
```

```
void add( int *a, int *b, int *c ) {
    int tid = 0;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 4;
    }
}
```

CPU
Core 1

```
void add( int *a, int *b, int *c ) {
    int tid = 1;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 4;
    }
}
```

CPU
Core 2

Hình 3.2. Đoạn mã cộng hai vectơ trên mỗi lõi.

Giả sử như **hình 3.2**, một CPU có hai lõi, trong đó mỗi lõi đảm nhiệm xử lý một phần của mảng dữ liệu từ chỉ mục 0 đến $n-1$. Khi thực hiện phép cộng hai vector song song trên hai lõi, mã nguồn được chia thành hai phần.

Nhược điểm của phương pháp xử lý trên CPU này là việc các phép tính cộng đang thực hiện theo một cách riêng lẻ, dẫn đến thời gian xử lý tăng tuyến tính theo số phần tử.

Bên cạnh đó còn là về giới hạn về số lỗi, số lượng lỗi xử lý của CPU thường ít hơn rất nhiều so với GPU, khiến cho khả năng xử lý song song bị hạn chế. Đồng thời khi xử lý các vectơ có kích thước lớn, tốc độ truy cập và truyền tải dữ liệu giữa bộ nhớ và CPU cũng là yếu tố làm giảm hiệu suất tổng thể.

Từ những hạn chế này đã thúc đẩy việc nghiên cứu và áp dụng các phương pháp xử lý song song trên GPU, chúng em có đề xuất sử dụng các nền tảng như CUDA, CuPy và Rapids, nhằm tối ưu hóa tốc độ xử lý và khả năng mở rộng cho bài toán với quy mô lớn.

3.1.4. Cộng Vectơ trên GPU

a) Cộng Vectơ trên nền tảng CUDA

Trong bài toán này, chúng ta sử dụng GPU trên nền tảng CUDA để thực hiện phép cộng hai vector bằng cách phân chia công việc thành nhiều thread song song. Phép toán được triển khai thông qua CUDA kernel, trong đó mỗi thread xử lý một phần tử của vector và nhiều thread được tổ chức trong các block.

Giả sử chúng ta có hai vectơ A và B, mỗi vectơ có 36 phần tử và chúng ta cần tính tổng của chúng để tạo ra vectơ kết quả C, sao cho:

$$C[i] = A[i] + B[i], \text{ với mọi } i \in [0, 35]$$

Trên CPU, phép toán này sẽ được thực hiện theo kiểu tuần tự bằng một vòng lặp, nhưng trên GPU, ta có thể tận dụng tính toán song song bằng cách sử dụng mô hình Grid - Block - Thread của CUDA.

CUDA sử dụng mô hình lập trình song song trong đó Grid chịu trách nhiệm thực thi một kernel duy nhất. Grid là một mảng mà mỗi phần tử là một Block. Block là tập hợp các Thread và thread là đơn vị xử lý nhỏ nhất. Và khi áp dụng vào bài toán cộng 2 vectơ này ta có:

- Số lượng phần tử: $N = 36$
- Số lượng Grid là 1 vì số lượng phần tử khá nhỏ nên chỉ cần 1 Grid là đủ.
- Số lượng Blocks là 4.
- Số lượng Threads trong mỗi Block là 9.

Mỗi thread chịu trách nhiệm tính tổng một phần tử của vectơ bằng cách lấy chỉ số idx theo công thức:

$$\text{idx} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

Với:

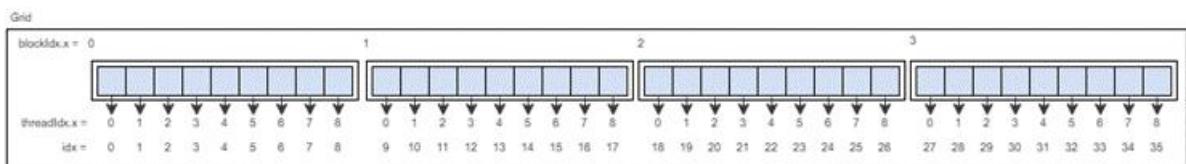
- `blockIdx.x`: chỉ số block.
- `blockDim.x`: số thread trong mỗi block.
- `threadIdx.x`: chỉ số thread trong một block.

Ví dụ:

- Block 0, Thread 0 -> $\text{idx} = 0 * 9 + 0 = 0$
- Block 1, Thread 2 -> $\text{idx} = 1 * 9 + 2 = 11$
- Block 3, Thread 8 -> $\text{idx} = 3 * 9 + 8 = 35$

Như vậy mỗi thread sẽ thực hiện phép toán:

$$C[\text{idx}] = A[\text{idx}] + B[\text{idx}]$$



Hình 3.3. Minh họa ý tưởng cộng 2 vectơ trên lập trình song song.

Theo ý tưởng trên **hình 3.3** thì mỗi block sẽ xử lý một nhóm 9 phần tử của vectơ. Mỗi thread trong block xử lý một phần tử cụ thể. Cuối cùng, chỉ số thread và block quyết định vị trí dữ liệu và thread đó xử lý.

Block 0 xử lý phần tử từ 0 -> 8, Block 1 xử lý phần tử từ 9 -> 17, Block 2 xử lý phần tử từ 18 -> 26 và block 3 xử lý phần tử từ 27 -> 35.

Khi chạy chương trình, GPU sẽ khởi tạo tất cả các thread song song,, thực hiện phép cộng cho từng phần tử của vectơ và ghi kết quả vào bộ nhớ toàn cục.

Mã giả cộng 2 vectơ trên GPU với nền tảng CUDA:

Thuật toán cộng 2 vectơ trên GPU với nền tảng CUDA Input: Hai vectơ A, B với kích cỡ N Output: Vectơ C với $C[i] = A[i] + B[i]$ 1. Cấp phát vùng nhớ cho <code>device_A</code> , <code>device_B</code> , and <code>device_C</code> trên bộ nhớ GPU, mỗi vùng có kích thước đủ để chứa N phần tử. 2. Sao chép toàn bộ dữ liệu của vectơ A,B từ bộ nhớ CPU tương ứng sang cho <code>device_A</code> , <code>device_B</code> . 3. Xác định kích thước grid và block, sau đó gọi hàm <code>kernel<<<grid_size, block_size>>></code> trên GPU với tham số là <code>device_A, device_B, device_C, N</code> . 4. For mỗi thread trên GPU có chỉ số tid.
--

```

5. If tid < N
6.     device_C[tid] = device_A[tid] + device_B[tid]
7. Kết thúc kernel.
8. Sao chép kết quả từ device_C (trên GPU) về lại bộ nhớ CPU. Sau đó
lưu vào vecto C.
9. Giải phóng bộ nhớ GPU: device_A, device_B, device_C
10. Trả về vecto C.

```

Dựa vào mã giả trên, ta áp dụng vào ví dụ của mình để thực hiện tính toán song song như sau:

- Cấp phát bộ nhớ trên GPU: Sao chép dữ liệu đầu vào từ CPU sang GPU.
- Gọi kernel CUDA với blockDim.x = 4 (tức là 4 block) và block
- Dim.x = 9 (tức là 9 thread mỗi block).
- Mỗi thread thực hiện phép cộng:
 - Lấy chỉ số tid của phần tử mình cần xử lý.
 - Truy xuất hai phần tử tương ứng từ vector A và B.
 - Thực hiện phép cộng $C[tid] = A[tid] + B[tid]$.
 - Ghi kết quả vào bộ nhớ toàn cục C.
- Chép kết quả từ GPU về CPU và xuất ra màn hình.
- Giải phóng bộ nhớ GPU sau khi tính toán xong.

Block 1

```

__global__ void vectorAddCuda
(int *a, int *b, int *c) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

```

Block 2

```

__global__ void vectorAddCuda
(int *a, int *b, int *c) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

```

Block 3

```

__global__ void vectorAddCuda
(int *a, int *b, int *c) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

```

Block 4

```

__global__ void vectorAddCuda
(int *a, int *b, int *c) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

```

Hình 3.4. Đoạn mã kernel cộng 2 vecto được thực thi trên GPU.

Ta thấy Hình 3.4 có 4 khung (Block 1, Block 2, Block 3, Block 4) và mỗi khung đều thể hiện cùng một hàm kernel `vectorAddCuda(int*a, int*b, int*c)`. Hình 3.4 đã đơn giản hóa khái niệm “threadID”, giúp chúng ta dễ hình dung giá trị `tid` (thread ID) được đặt tĩnh khác nhau (0, 1, 2, 3) trong mỗi khung. Đây thực chất chỉ là cách biểu diễn mang tính minh họa để cho thấy rằng mỗi thread trong GPU sẽ tính toán một phần khác nhau của vecto A và B, chứ không phải tất cả cùng thực hiện trên cùng một chỉ số. Trên thực tế, khi ta gọi:

```
vectorAddCuda<<<gridSize, blockSize>>>(a, b, c)
```

Thì hàng trăm hoặc hàng nghìn thread sẽ chạy đồng thời. Mỗi thread có một `tid` duy nhất và thực hiện việc cộng `a[tid]` và `b[tid]`.

Cài đặt nhân kernel cho Vector Add CUDA:

```
__global__ void vector_add(
float * in1, float * in2, float * out, int len
)
{
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < len) {
out[idx] = in1[idx] + in2[idx];
}
}
```

Trong đoạn mã kernel trên, hàm `vector_add` được khai báo từ khóa `__global__`, để cho biết rằng hàm này được thi thi trên GPU và có thể được gọi từ host CPU.

Biến `idx` được tính theo công thức:

$$\text{idx} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

Giúp xác định vị trí của từng thread trong toàn bộ grid. Điều này đảm bảo mỗi thread xử lý một phần tử cụ thể của các vecto.

Mỗi thread thực hiện phép tính:

$$\text{out}[idx] = \text{in1}[idx] + \text{in2}[idx]$$

b) Cộng vecto sử dụng thư viện CuPy

Một trong những ưu điểm nổi bật của CuPy là khả năng chuyển đổi dữ liệu một cách trực quan từ CPU sang GPU, đồng thời cho phép tái sử dụng kernel CUDA tùy chỉnh thông qua hàm `cp.RawKernel`. Trong bài toán cộng hai vecto, ngoài việc sử dụng các phép toán tích hợp sẵn, chúng ta còn có thể tối ưu hóa hiệu suất xử lý bằng cách viết kernel CUDA chuyên dụng để thực hiện phép cộng từng phần tử một cách song song.

Chúng ta sẽ thực hiện chuyển đổi dữ liệu từ CPU sang GPU bằng cách sử dụng `cp.asarray` để chuyển các vectơ A và B từ định dạng NumPy (hoặc dữ liệu ban đầu trên CPU) sang mảng của CuPy, cho phép lưu trữ dữ liệu trực tiếp trong bộ nhớ GPU.

Dựa trên kernel CUDA đã được phát triển ở phần trước, chúng ta sẽ tái sử dụng bằng cách biên dịch nó thông qua `cp.RawKernel`. Cuối cùng sau khi kernel hoàn thành, GPU được đồng bộ hóa để đảm bảo mọi tính toán đã hoàn thành, sau đó kết quả từ vectơ C có thể được chuyển về CPU bằng hàm `cp.asarray`.

Cài đặt kernel cho Vector Add Cupy:

```
extern "C" __global__ void vector_add(
    const long long* a, const long long* b,
    long long* c, int n)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < n) {
        c[tid] = a[tid] + b[tid];
    }
}
```

Sau khi hoàn thành kernel, ta sẽ tiến hành viết một hàm Python để thực hiện Vector

```
Add trên GPU tích hợp kernel trên.
def vector_add_gpu(a, b, gpu_id):
    with cp.cuda.Device(gpu_id):
        #Copy data to GPU
        start_copy_to = time.time()
        d_a = cp.asarray(a, dtype=cp.int64)
        d_b = cp.asarray(b, dtype=cp.int64)
        cp.cuda.Device(gpu_id).synchronize()
        copy_to_time = (time.time() - start_copy_to) * 1000
        print(f"GPU {gpu_id} - Time to copy data from CPU to GPU:
{copy_to_time:.4f} ms")

    # Kernel computation
    Start_kernel = time.time()
    n = len(d_a)
    block_size = 256
    grid_size = (n + block_size - 1)
    d_result = cp.zeros(n, dtype=cp.int64)

    # Compile kernel
    vector_add_kernel = cp.RawKernel(kernel_code, 'vector_add')
    # Launch kernel
    vector_add_kernel((grid_size,), (block_size,), (d_a, d_b, d_result,
n))
    cp.cuda.Device(gpu_id).synchronize()

    kernel_time = (time.time() - start_kernel) * 1000
```

```

print(f"GPU {gpu_id} - Time for kernel computation on GPU:
{kernel_time:.4f} ms")

# Copy result back to CPU
start_copy_from = time.time()
result = d_result.get()
cp.cuda.Device(gpu_id).synchronize()
copy_from_time = (time.time() - start_copy_from) * 1000
print(f"GPU {gpu_id} - Time to copy data from GPU to CPU:
{copy_from_time:.4f} ms")

return result

```

Hàm `vector_add_gpu` được thiết kế để thực hiện phép cộng vector trên GPU, tích hợp với việc khởi chạy kernel đã được biên dịch sẵn. Cấu trúc của hàm gồm các bước chính như sau:

Chọn thiết bị GPU và sao chép dữ liệu từ CPU sang GPU:

- Đoạn lệnh `with cp.cuda.Device(gpu_id)`: đảm bảo rằng toàn bộ các thao tác bên trong khối lệnh này sẽ được thực hiện trên GPU có định danh `gpu_id`.
- Hai vector đầu vào `a` và `b` được chuyển sang bộ nhớ của GPU thông qua hàm `cp.asarray`. Đây là bước cần thiết để dữ liệu sẵn sàng cho các thao tác tính toán song song trên GPU.
- Sau khi chuyển dữ liệu, sử dụng `cp.cuda.Device(gpu_id).synchronize()` để đảm bảo rằng việc chuyển dữ liệu đã hoàn tất, đồng thời đo thời gian sao chép từ CPU sang GPU.

Thiết lập và khởi chạy kernel:

- Biến `n` lưu trữ số phần tử của vector.
- Thiết lập các tham số về kích thước block và grid để đảm bảo toàn bộ các phần tử của vector được bao phủ.
- Kernel được biên dịch thông qua:

```
cp.RawKernel(kernel_code, 'vector_add')
```

- Sau đó kernel được khởi chạy với tham số cấu hình `grid_size`, `block_size` và truyền vào các tham số cần thiết bao gồm vector `d_a`, `d_b`, `d_result` và kích thước `n`.
- Sau khi khởi chạy, gọi hàm `cp.cuda.Device(gpu_id).synchronize()` để đảm bảo rằng toàn bộ quá trình tính toán của kernel đã hoàn tất, đồng thời đo thời gian thực hiện kernel.

Sao chép kết quả từ GPU về CPU:

- Sau khi tính toán xong, kết quả được chuyển trả lại bộ nhớ CPU thông qua phương thức `d_result.get()`
- Một lần đồng bộ hóa nữa được thực hiện để đảm bảo rằng toàn bộ quá trình chuyển kết quả đã hoàn tất, và thời gian sao chép từ GPU về CPU được tính toán tương tự như bước chuyển dữ liệu ban đầu.

Trả về kết quả:

- Cuối cùng, hàm trả về vector kết quả được tính toán trên GPU, sau khi trải qua các bước chuyển dữ liệu, tính toán kernel và chuyển kết quả về CPU.

c) Cộng 2 vectơ với nền tảng Rapids kết hợp Raw Kernel Cupy

Trong phần này, chúng ta khai thác sức mạnh của GPU thông qua sự kết hợp của Rapids và tận dụng lại Cupy để tối ưu hóa bài toán cộng hai vectơ.

Cài đặt nhân kernel cho Vector Add Rapids

```
extern "C" __global__ void vector_add(
const long long* a, const long long* b,
long long* c, int n)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < n) {
        c[tid] = a[tid] + b[tid];
    }
}
```

Với Rapids, ta sử dụng cuDF giúp xử lý dữ liệu dạng DataFrame, sau đó chuyển các cột dữ liệu sang mảng CuPy.

```
df1 = cudf.DataFrame({'values': cp.arange(n, dtype=cp.int64)})
df2 = cudf.DataFrame({'values': cp.arange(n, dtype=cp.int64)})

vector_a = df1['values'].to_cupy()
vector_b = df2['values'].to_cupy()
```

Tiếp theo, kernel CUDA tùy chỉnh được tái sử dụng thông qua hàm `cp.RawKernel` để thực hiện cộng từng phần tử một cách song song. Quá trình này cho phép tối ưu hiệu suất tính toán trong khi vẫn giữ được tính trực quan và dễ dàng của Python.

```
def vector_add_gpu(a, b, gpu_id):
    with cp.cuda.Device(gpu_id):
        # Copy data to GPU
        start_copy_to = time.time()
        d_a = cp.asarray(a, dtype=cp.int64)
        d_b = cp.asarray(b, dtype=cp.int64)
        cp.cuda.Device(gpu_id).synchronize()
        copy_to_time = (time.time() - start_copy_to) * 1000
```

```

        print(f"GPU {gpu_id} - Time to copy data from CPU to GPU:
{copy_to_time:.4f} ms")

        # Kernel computation
        start_kernel = time.time()
        n = len(d_a)
        block_size = 256
        grid_size = (n + block_size - 1) // block_size
        d_result = cp.zeros(n, dtype=cp.int64)

        # Compile kernel
        vector_add_kernel = cp.RawKernel(kernel_code, 'vector_add')

        # Launch kernel
        vector_add_kernel((grid_size,), (block_size,), (d_a, d_b,
d_result, n))
        cp.cuda.Device(gpu_id).synchronize()

        kernel_time = (time.time() - start_kernel) * 1000
        print(f"GPU {gpu_id} - Time for kernel computation on GPU:
{kernel_time:.4f} ms")

        # Copy result back to CPU
        start_copy_from = time.time()
        result = d_result.get()
        cp.cuda.Device(gpu_id).synchronize()
        copy_from_time = (time.time() - start_copy_from) * 1000
        print(f"GPU {gpu_id} - Time to copy data from GPU to CPU:
{copy_from_time:.4f} ms")

    return result

```

3.1.4. Kết quả thực nghiệm

a) Môi trường thực nghiệm

Thí nghiệm được thực hiện trên môi trường Kaggle với GPU T4x2. Mục tiêu của thí nghiệm này là đánh giá hiệu năng của bài toán cộng vectơ khi triển khai trên GPU so với cách tiếp cận truyền thống trên CPU.

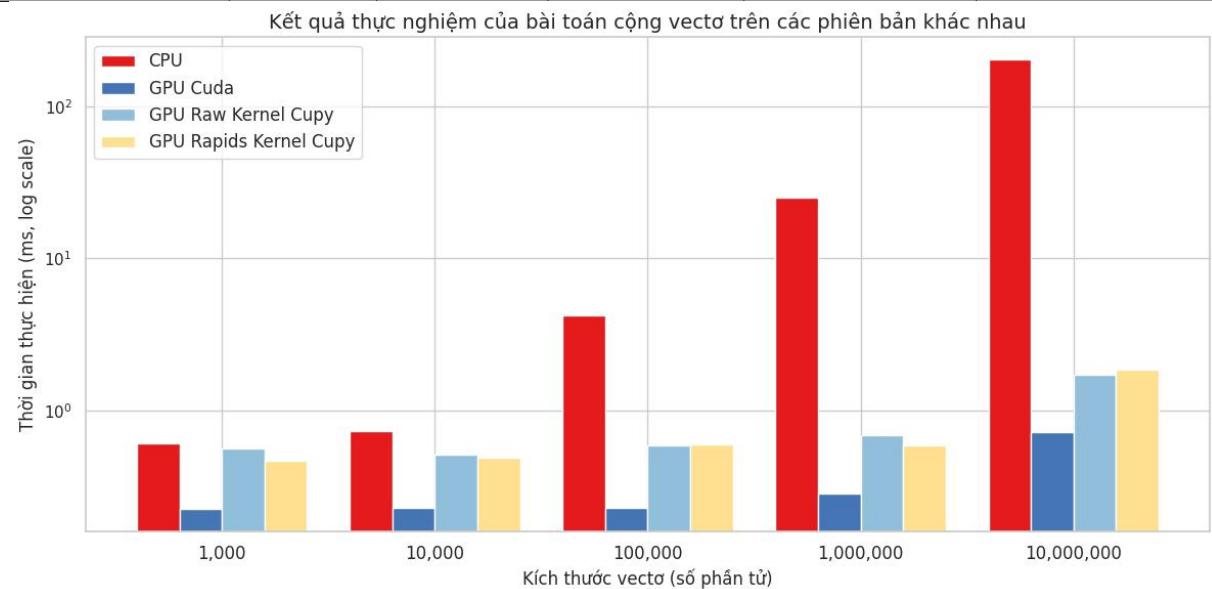
b) Dữ liệu

Để đánh giá hiệu năng của các phương pháp cộng hai vectơ, chúng em đã tiến hành thực nghiệm trên nhiều kích thước của các vectơ khác nhau {1000; 10000; 100000; 1000000; 10000000} và thu được kết quả như trong bảng IV.1 . Mỗi cột thể hiện thông thời gian thực hiện phép cộng (đơn vị: ms), bao gồm quá trình khởi tạo dữ liệu, thực thi thuật toán, và thu thập kết quả.

c) Kết quả

Bảng 3.1. Kết quả thời gian chạy của các phiên bản cộng vectơ (Mili giây).

Bộ xử lý	Tổng thời gian thực hiện tính toán cộng vectơ cho các kích thước vectơ khác nhau (đơn vị ms)				
	1.000	10.000	100.000	1.000.000	10.000.000
CPU	0.60	0.73	4.23	25.27	202.86
GPU Cuda	0.22	0.23	0.23	0.28	0.72
GPU Raw Kernel Cupy	0.56	0.51	0.58	0.68	1.71
GPU Rapids Kernel Cupy	0.46	0.48	0.59	0.58	1.86



Hình 3.5. Kết quả thực nghiệm của bài toán cộng vectơ trên các phiên bản khác nhau.

Từ **bảng 3.1**, ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như **hình 3.5**.

Qua kết quả trình bày trong **bảng 3.1** và **hình 3.5**, có thể rút ra các nhận định và phân tích như sau:

So sánh giữa CPU và GPU:

- Ở phiên bản CPU, thời gian thực hiện cộng vectơ trên CPU tăng gần như tuyến tính theo kích thước của dữ liệu. Cụ thể, với 10 triệu phần tử, CPU mất khoảng

202.86ms để xử lý, cho thấy hạn chế rõ rệt của phương pháp xử lý tuần tự khi đối mặt với khối lượng dữ liệu lớn.

- Còn ở phiên bản GPU thì ngược lại, các phiên bản GPU cho thấy thời gian thực hiện rất thấp bất kể quy mô dữ liệu. Ví dụ, với 10 triệu phần tử, GPU Cuda mất **0.72ms** nhanh hơn CPU đến khoảng 282 lần, GPU Raw Kernel Cupy cũng chỉ mất **1.71ms** và GPU Rapids mất 1.86ms, nhanh hơn CPU khoảng 118 và 109 lần tương ứng. Những con số này khẳng định sức mạnh của xử lý song song trên GPU.

Hiệu năng giữa các phiên bản GPU:

- Cuda là phiên bản cho hiệu năng cao nhất, với thời gian xử lý thấp nhất (**0.72ms** với 10 triệu phần tử). Đây là kết quả tối ưu khi tối ưu hóa trực tiếp bằng Cuda, tận dụng triệt để khả năng tính toán song song của GPU.

Ở phiên bản Cupy và Rapids :

- Hiệu suất thấp hơn so với Cuda: Các phiên bản Cupy có thời gian xử lý cao hơn một chút so với phiên bản Cuda (**1.71ms** và **1.86ms** với 10 triệu phần tử). Sự chênh lệch này chủ yếu do overhead của thư viện Cupy, vốn cung cấp lớp trừu tượng giúp đơn giản hóa quá trình viết code và quản lý tài nguyên GPU.
- Lợi thế về mặt phát triển: Mặc dù hiệu suất của Cupy thấp hơn so với Cuda thuận tiện, nhưng Cupy là một lựa chọn tốt khi ưu tiên tính đơn giản và dễ bảo trì của code. Việc sử dụng Cupy giúp giảm bớt yêu cầu tối ưu hóa chuyên sâu, đồng thời vẫn đảm bảo hiệu năng cao so với các phương pháp xử lý trên CPU.
- Còn với phiên bản Rapids thì thời gian xử lý khoảng **1.86ms** cho 10 triệu phần tử, phiên bản này được thiết kế nhằm tối ưu hóa việc xử lý lớn, đặc biệt là các dạng dataframe. Nhờ vậy, Rapids phù hợp hơn cho các ứng dụng phân tích dữ liệu phức tạp, nơi không chỉ cần xử lý các phép tính số học đơn giản mà còn cần thao tác, xử lý và phân tích dữ liệu dạng bảng.

Từ đó, có thể kết luận rằng việc lựa chọn các công nghệ xử lý GPU phụ thuộc vào yêu cầu cụ thể của ứng dụng:

- Cuda cho những trường hợp cần tối ưu hiệu suất cực đại.
- Cupy cho trường hợp ưu tiên đơn giản hóa code và bảo trì, với hiệu năng vẫn cao hơn nhiều so với CPU.

- Rapids khi làm việc với các tập dữ liệu dạng dataframe và yêu cầu xử lý dữ liệu lớn một cách tổng hợp.

3.2. Bài toán Histogram

Sau khi tính tổng điểm của từng học sinh bằng phép cộng vectơ, ta có thể hình dung một bức tranh tổng quan hơn về hiệu suất học tập của cả lớp. Tuy nhiên, chỉ xét tổng điểm riêng lẻ của từng học sinh chưa đủ để hiểu rõ xu hướng chung. Chẳng hạn, chúng ta có thể muốn trả lời các câu hỏi như:

- Số lượng học sinh đạt điểm cao có nhiều không?
- Có bao nhiêu học sinh có điểm trung bình thấp hơn một ngưỡng nhất định?
- Phân bố điểm số trong lớp có đồng đều hay không?

Để trả lời những câu hỏi này, ta cần một cách tiếp cận khác, thay vì chỉ cộng tổng điểm. Lúc này, histogram trở thành một công cụ hữu ích.

Histogram giúp chúng ta trực quan hóa sự phân bố của tổng điểm, bằng cách chia tổng điểm thành các khoảng (bins) và đếm số lượng học sinh rơi vào mỗi khoảng. Điều này cho phép ta dễ dàng nhận diện các xu hướng trong dữ liệu, chẳng hạn như lớp có nhiều học sinh đạt điểm cao hay có một nhóm lớn học sinh có điểm thấp cần được hỗ trợ thêm.

3.2.1. Mô tả bài toán

Histogram là một phương pháp giúp đếm số lần xuất hiện của từng giá trị trong một tập dữ liệu. Việc tính toán histogram có thể thực hiện trên GPU, giúp tăng tốc đáng kể so với CPU. Nhiều thuật toán yêu cầu tính toán histogram từ nhiều tập dữ liệu khác nhau. Về cơ bản, khi có một tập hợp các phần tử, histogram sẽ ghi lại tần suất xuất hiện của từng phần tử đó. Nhờ GPU có khả năng xử lý song song, việc tính toán histogram có thể nhanh hơn rất nhiều so với cách làm trên CPU.

Tính toán histogram được sử dụng trong khoa học máy tính để xử lý hình ảnh, thống kê dữ liệu, thị giác máy tính, học máy, mã hóa âm thanh và nhiều tính năng khác.

Ví dụ: chúng ta có một biểu đồ histogram của các chữ cái trong câu “TINH TOAN HISTOGRAM”

T	I	N	H	O	A	S	G	R	M
3	2	2	2	2	2	1	1	1	1

3.2.2. Ý tưởng

Dữ liệu:

T	I	N	H	T	O	A	N	H	I	S	T	O	G	R	A	M
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Chia thành các bins:

T	I	N	H	O	A	S	G	R	M
---	---	---	---	---	---	---	---	---	---

Đếm số lần xuất hiện:

T	I	N	H	O	A	S	G	R	M
3	2	2	2	2	2	1	1	1	1

Hình 3.6. Ý tưởng bài toán histogram.

Quá trình xây dựng Histogram có thể được thực hiện theo các bước sau như **hình 3.6**:

Bước 1: Chia khoảng giá trị thành các bins (nhóm giá trị)

- Ví dụ: Với ảnh grayscale, các pixel có giá trị từ 0-255, ta chia thành 256 bins, mỗi bin tương ứng với một giá trị pixel. Còn nếu dữ liệu là số thực, ta cần chia thành các khoảng (bins) với độ rộng phù hợp.

Bước 2: Đếm số lượng phần tử trong từng bin

- Tiến hành duyệt qua từng phần tử trong dữ liệu để xác định bin mà phần tử thuộc vào rồi tăng biến đếm cho bin đó lên 1.

Bước 3: Tính mật độ tần suất

- Nếu cần chuẩn hóa, ta chia số lượng phần tử trong mỗi bin cho độ rộng bin (interval width).

3.2.3. Histogram trên CPU

Khi tính toán Histogram trên CPU, dữ liệu đầu vào sẽ được duyệt tuần tự để đếm số lần xuất hiện của từng giá trị. Khi dữ liệu lớn, CPU phải quét qua toàn bộ tập dữ liệu nhiều lần, làm tăng thời gian xử lý. Đối với cách tiếp cận truyền thống là:

- Xác định số lượng bins (khoảng giá trị): Nếu dữ liệu có giá trị từ 0 đến max_value, ta cần mảng bins có kích thước max_value + 1 để chứa số lần xuất hiện của từng giá trị.
- Khởi tạo mảng histogram (bin[]): Tạo một mảng có kích thước bằng số lượng bins, tất cả phần tử ban đầu là 0. Duyệt qua từng phần tử trong data và tăng giá trị ở vị trí tương ứng.

```
TinhtoanHistogram_CPU(data)
    Giatrilonnhat = MAX(data)
    bin = ARRAY[Giatrilonnhat + 1]

    for i in 0 -> Giatrilonnhat:
        bin[i] = 0

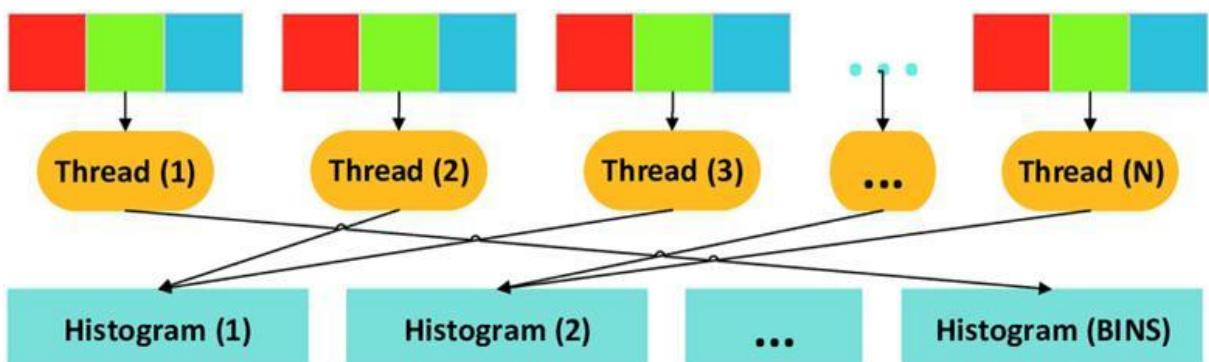
    for mỗi giá trị in data:
        bin[value] = bin[value] + 1

    return bin[]
```

Thay vì duyệt từng phần tử và cập nhật thủ công, NumPy cung cấp hàm np.histogram() để có thể tính toán histogram một cách hiệu quả hơn. Nó giúp tự động chia dữ liệu vào các bins, và lưu số lần xuất hiện của giá trị vào trong data

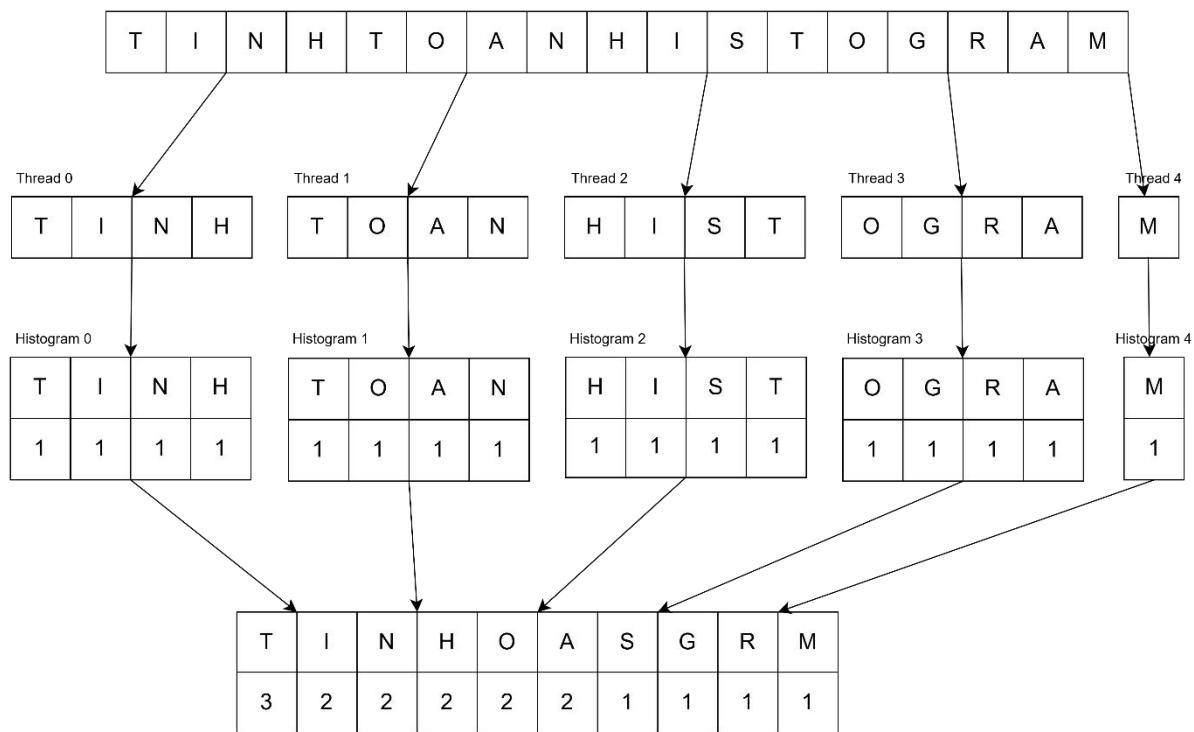
```
def compute_histogram(data):
    histo, _ = np.histogram(data, bins=np.arange(data.max() + 2))
    return histo
```

3.2.4. Histogram trên GPU



Hình 3.7. Tính toán histogram trên GPU.

Thay vì xây dựng histogram tuần tự như trên CPU, ta sẽ tận dụng khả năng tính toán song song của GPU như **hình 3.7** để đếm tần suất xuất hiện của các giá trị. Mỗi luồng (thread) sẽ đảm nhận việc đếm một phần của dữ liệu đầu vào, từ đó phân chia công việc xây dựng histogram thành nhiều phần nhỏ và thực hiện song song.



Hình 3.8. Minh họa cụ thể ý tưởng bài toán Histogram song song.

Ý tưởng thực hiện tính toán Histogram được minh họa cụ thể trong **hình 3.8**

Bước 1: Phân chia dữ liệu đầu vào

- Dữ liệu đầu vào được chia thành các phần nhỏ, mỗi phần sẽ được xử lý bởi một luồng (thread) riêng biệt.
- Các phần này sẽ được tổ chức thành các khối (blocks) và grid. Mỗi thread trong mỗi block sẽ xử lý một đoạn nhỏ của dữ liệu.

Bước 2: Tạo histogram cục bộ

- Mỗi luồng (thread) sẽ tạo một histogram cục bộ (local histogram) cho phần dữ liệu mà nó xử lý.
- Các luồng này hoạt động song song và đếm tần suất xuất hiện của các giá trị trong phần dữ liệu mà chúng được giao.

Bước 3: Gộp các histogram cục bộ

- Sau khi mỗi luồng hoàn thành việc tạo histogram cục bộ, các histogram này cần phải được gộp lại để tạo ra kết quả cuối cùng.
- Để đảm bảo việc gộp không bị xung đột (race condition), GPU sử dụng các phép toán nguyên tử (atomic operations), như atomicAdd(), giúp nhiều luồng có thể cập nhật cùng một vị trí trong histogram mà không làm hỏng kết quả.

Bước 4: Tổng hợp kết quả

- Histogram cuối cùng được tổng hợp từ tất cả các histogram cục bộ.
- Kết quả này là histogram hoàn chỉnh cho toàn bộ dữ liệu đầu vào, sau khi các phần tử đã được đếm và gộp lại một cách song song.

Tại bước gộp các histogram cục bộ, nếu tất cả các thread đều ghi trực tiếp vào một histogram chung, sẽ xảy ra hiện tượng xung đột ghi (race condition), dẫn đến kết quả sai hoặc giảm hiệu suất. Để giải quyết vấn đề này, GPU có thể sử dụng một trong hai cách tiếp cận sau:

- Sử dụng histogram cục bộ: Mỗi thread, hoặc mỗi nhóm thread (block), sẽ có một histogram riêng, có thể cập nhật song song mà không bị xung đột. Sau khi hoàn thành việc xử lý, các histogram cục bộ này sẽ được hợp nhất lại để tạo ra histogram cuối cùng.
- Sử dụng các phép toán nguyên tử (atomic operations): Mỗi thread sẽ ghi trực tiếp vào histogram chung, nhưng sử dụng các phép toán nguyên tử như atomicAdd(). Điều này đảm bảo rằng nhiều thread có thể cập nhật cùng một bin mà không gặp phải xung đột dữ liệu, vì phép toán nguyên tử sẽ giúp bảo vệ các thao tác ghi vào bộ nhớ chung.

a) Histogram trên nền tảng CUDA

Thực hiện tính histogram trên GPU bằng cách chia dữ liệu cho các thread, mỗi thread cập nhật trực tiếp vào histogram toàn cục bằng atomicAdd, sau đó sao chép kết quả về CPU để ghi ra file.

Bước 1: Phân chia dữ liệu đầu vào

Dữ liệu đầu vào là một mảng các số nguyên hostInput có độ dài inputLength. Để xử lý song song trên GPU, mảng được chia thành các phần nhỏ, mỗi phần sẽ được xử lý bởi một thread.

- Mảng hostInput được đọc từ file và lưu trữ trên host (CPU).
 - Dữ liệu sau đó được sao chép sang GPU (deviceInput) bằng cudaMemcpy(deviceInput, hostInput, byteSize, cudaMemcpyHostToDevice);
 - Mỗi thread sẽ xử lý một phần tử trong mảng input dựa trên chỉ số idx
- ```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

#### Bước 2: Tạo histogram cục bộ

Mỗi thread sẽ đếm tần suất xuất hiện của một giá trị trong phần dữ liệu mà nó được giao. Tuy nhiên, trong phần này, không có bước tạo histogram cục bộ riêng lẻ cho

từng thread (như trong hình vẽ của bạn). Thay vào đó, mỗi thread trực tiếp cập nhật vào histogram toàn cục (bins) bằng phép toán nguyên tử (atomicAdd)

```
if (idx < n) {
 atomicAdd(&bins[input[idx]], 1ULL); }
```

### Bước 3: Gộp các histogram cục bộ

Thông thường, nếu có histogram cục bộ, các histogram này sẽ được gộp lại thành một histogram toàn cục. Tuy nhiên, nếu không tạo histogram cục bộ thì mỗi thread cập nhật trực tiếp vào histogram toàn cục (deviceOutput) thông qua atomicAdd, bước này được thực hiện ngay trong kernel.

```
1: Cấp phát bộ nhớ cho GPU: dev_input, dev_histogram
2: Sao chép dữ liệu đầu vào vào dev_input
3: Khởi tạo dev_histogram = 0
4: Khởi chạy CUDA kernel với N luồng, trong đó N = (inputLength +
blockSize - 1) / blockSize
5: for mỗi luồng with index tid do
6: if tid < inputLength then
7: atomicAdd(&dev_histogram[input[tid]], 1ULL)
8: end if
9: end for
10: Sao chép dev_histogram về histogram
11: for i = 0 to inputLength - 1 do
12: print histogram[i]
13: end for
14: Giải phóng bộ nhớ GPU: dev_input, dev_histogram
15: return histogram
```

### b) Histogram sử dụng thư viện CuPy

Sử dụng CuPy cho bài toán tính toán histogram trên GPU có nhiều lợi ích, đặc biệt là khi làm việc với các bộ dữ liệu lớn và cần hiệu suất cao

Ý tưởng thực hiện histogram trên CuPy

- Chuyển dữ liệu từ CPU sang GPU: Dữ liệu đầu vào (mảng số nguyên) sẽ được chuyển từ NumPy sang CuPy để lưu trữ trực tiếp trong bộ nhớ GPU.
- Tái sử dụng kernel CUDA: Kernel CUDA để tính toán histogram sẽ được viết và biên dịch lại bằng cp.RawKernel, giúp thực hiện phép toán song song cho từng phần tử của mảng đầu vào.
- Sao chép kết quả từ GPU về CPU: Sau khi tính toán xong trên GPU, kết quả histogram sẽ được sao chép về CPU để kiểm tra và lưu kết quả.

```
1: Hàm histogram_gpu(input_data, n_bins, gpu_id)
2: Thiết lập GPU device thành gpu_id
3: Sao chép input_data vào GPU (d_input)
```

```

4: Khởi tạo mảng (d_output) với giá trị 0, kích thước n_bins
5: Gán n là độ dài của d_input
6: Gán block_size = 256
7: Tính grid_size = (n + block_size - 1) / block_size
8: Khởi chạy kernel histogram với tham số (grid_size,) và
(block_size,)
9: Chờ GPU hoàn tất tính toán
10: Sao chép kết quả từ d_output về CPU
11: Return kết quả

```

### c) Histogram trên nền tảng RAPIDS kết hợp Raw Kernel Cupy

Sử dụng Raw Kernel trong CuPy cho phép tối ưu hóa tính toán trên GPU thông qua mã CUDA, mang lại hiệu suất rất cao khi làm việc với các bộ dữ liệu lớn. Việc sử dụng RAPIDS và cuDF cho phép các thao tác dữ liệu như chuyển đổi và xử lý dữ liệu được thực hiện nhanh chóng trên GPU. Khi sử dụng RAPIDS với Raw Kernel trong CuPy để tính histogram trên GPU, ta thực hiện các bước sau:

- Tải dữ liệu vào GPU: Dữ liệu từ CPU sẽ được chuyển sang GPU bằng CuPy (sử dụng `cp.asarray()`), giúp tận dụng khả năng tính toán song song của GPU.
- Tạo Kernel: Viết một kernel tùy chỉnh bằng CUDA (như trong mã giả trước) để tính toán histogram. Kernel này sẽ đếm số lần xuất hiện của mỗi giá trị trong dữ liệu, sử dụng hàm `atomicAdd` để thực hiện các phép cộng an toàn khi nhiều thread cùng truy cập vào một bin.
- Tính toán Histogram:
  - Xác định số lượng block và grid cần thiết dựa trên kích thước dữ liệu (sử dụng `block_size` và `grid_size`).
  - Gọi kernel CUDA với CuPy Raw Kernel để thực thi tính toán histogram trên GPU.
- Kết quả và đồng bộ hóa: Sau khi tính toán trên GPU, kết quả (là histogram) sẽ được sao chép từ GPU về CPU (dùng `.get()` của CuPy). Sau đó, kiểm tra thời gian thực hiện và in ra kết quả.

```

1: Hàm histogram_cudf(input_data, n_bins, gpu_id)
2: Thiết lập GPU device thành gpu_id
3: Nạp dữ liệu đầu vào vào cuDF DataFrame
4: Tạo cuDF DataFrame từ input_data
5: In ra 10 phần tử đầu tiên của cuDF DataFrame
6: Tính toán histogram bằng value_counts() của cuDF
7: Sắp xếp kết quả của value_counts() theo index
8: Chuyển đổi index và giá trị histogram của cuDF thành mảng NumPy
9: Điene giá trị thiếu trong histogram bằng = 0

```

|                                                         |
|---------------------------------------------------------|
| 10: <b>Return</b> histogram đầy đủ dưới dạng mảng NumPy |
|---------------------------------------------------------|

### 3.2.5. Kết quả thực nghiệm

#### a) Môi trường thực nghiệm

Môi trường thực nghiệm được thiết lập trên nền tảng Kaggle, sử dụng GPU NVIDIA T4 với 16GB VRAM:

- Bộ xử lý (CPU): Intel Xeon
- Bộ nhớ RAM: 16GB
- Bộ xử lý đồ họa (GPU): NVIDIA T4 (16GB VRAM)
- Hệ điều hành: Linux (Ubuntu-based trên Kaggle)
- Phiên bản Python: 3.8+

#### b) Dữ liệu

Tổng số lượng phần tử trong histogram khi xử lý dữ liệu có kích thước khác nhau

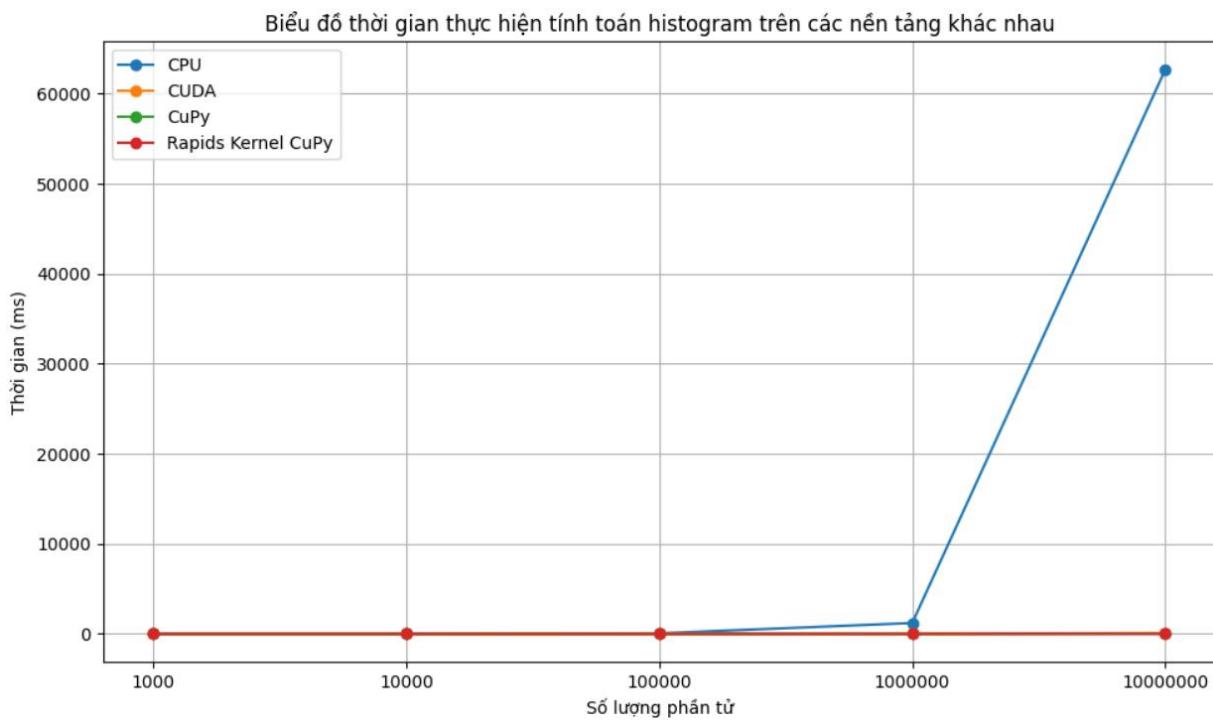
- Với tập dữ liệu có 1.000 phần tử thì tổng giá trị trong histogram là 2.000
- Với tập dữ liệu có 10.000 phần tử thì tổng giá trị trong histogram là 20.000
- Với tập dữ liệu có 100.000 phần tử thì tổng giá trị trong histogram là 200.000
- Với tập dữ liệu có 1.000.000 phần tử thì tổng giá trị trong histogram là 2.000.000
- Với tập dữ liệu có 10.000.000 phần tử thì tổng giá trị trong histogram là 20.000.000

#### c) Kết quả:

*Bảng 3.2. Kết quả thời gian chạy của các phiên bản Histogram (Mili giây).*

| <b>Bộ xử lý</b>        | <b>Tổng thời gian thực hiện tính toán histogram cho các kích thước bộ nhớ khác nhau (đơn vị ms)</b> |        |         |           |              |
|------------------------|-----------------------------------------------------------------------------------------------------|--------|---------|-----------|--------------|
|                        | 1.000                                                                                               | 10.000 | 100.000 | 1.000.000 | 10.000.000   |
| CPU                    | 1.47                                                                                                | 7.45   | 62      | 1207      | <b>62635</b> |
| GPU Cuda               | 0.35                                                                                                | 0.33   | 0.77    | 9.3       | 86.65        |
| GPU Raw<br>Kernel Cupy | 3.31                                                                                                | 2.56   | 2.98    | 4.7       | 30           |

|     |                    |      |      |      |      |             |
|-----|--------------------|------|------|------|------|-------------|
| GPU | Rapids Kernel Cupy | 0.76 | 0.81 | 0.74 | 2.31 | <b>12.9</b> |
|-----|--------------------|------|------|------|------|-------------|



**Hình 3.9.** Biểu đồ so sánh thời gian thực hiện histogram trên các bộ xử lý khác nhau.

Từ **bảng 3.2**, ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như **hình 3.9**. Sau khi thực hiện tính toán histogram trên các bộ xử lý khác nhau đã cho ra các mẫu dữ liệu khác nhau với các tập dữ liệu từ 1.000 cho đến 10.000.000 phần tử. CPU có xu hướng tăng thời gian xử lý nhanh chóng khi kích thước dữ liệu tăng, trong khi GPU có tốc độ xử lý nhanh hơn đáng kể.

Trên CPU, tính toán histogram được xây dựng trong 1.47ms đối với tập dữ liệu có 1.000 phần tử. Nhưng khi tập dữ liệu trở nên lớn hơn, lên đến 10.000.000 phần tử, tính toán histogram được xây dựng trong 62635ms (khoảng hơn 1 phút). Chứng tỏ CPU không xử lý tốt với dữ liệu lớn do bị giới hạn về khả năng song song hóa.

Trên GPU, đối với các tập dữ liệu nhỏ khoảng 100.000 phần tử thì GPU Cuda xử lý nhanh vượt trội so với 2 bộ xử lý còn lại. Nhưng đối với các tập dữ liệu lớn từ 1.000.000 phần tử thì GPU Rapids Kernel Cupy lại có thời gian xử lý nhanh nhất.

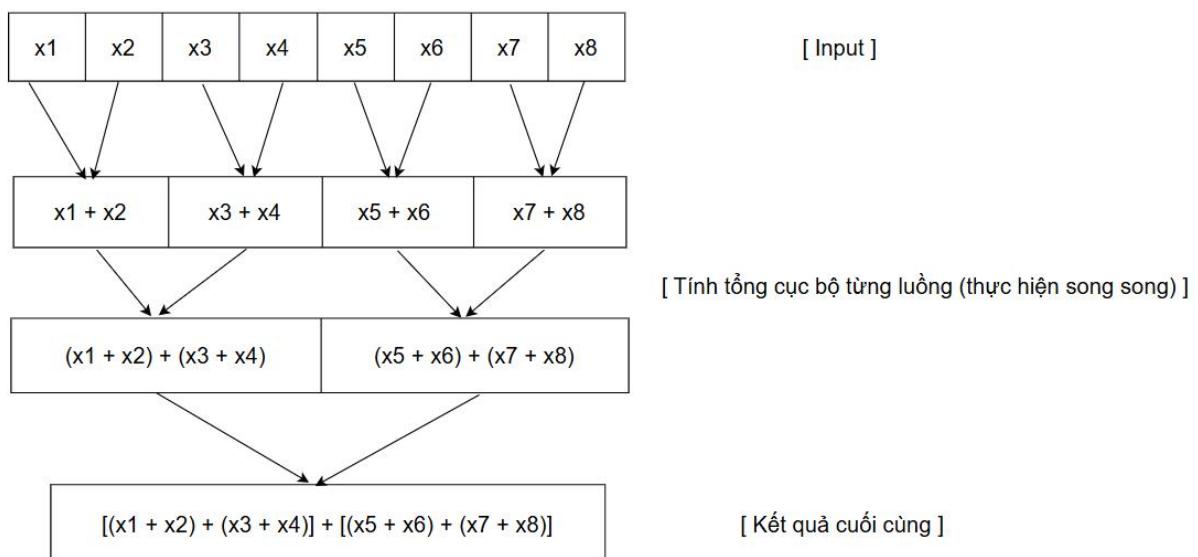
### 3.3. Bài toán Sum Reduce

#### 3.3.1. Mô tả bài toán

Sum Reduce là một thuật toán song song được sử dụng để tính tổng của một tập hợp các số. Nó hoạt động bằng cách chia tập hợp các số thành các cặp, sau đó tính tổng của mỗi cặp. Kết quả của các tổng này lại được chia thành các cặp và quá trình này được lặp lại cho đến khi chỉ còn lại một số, đó là tổng của tất cả các số ban đầu.

Thuật toán Sum Reduce là một khái niệm cơ bản trong điện toán song song và thường được sử dụng trong nhiều bối cảnh khác nhau, chẳng hạn như trong các khung lập trình song song như CUDA, OpenMP hoặc thậm chí trong các hệ thống điện toán phân tán.

#### 3.3.2. Ý tưởng



**Hình 3.10.** Minh họa thuật toán Sum Reduce.

Sum Reduce (**Hình 3.10**) là một thuật toán tính tổng một tập dữ liệu lớn bằng cách chia nhỏ dữ liệu và giảm dần (reduce) tổng qua nhiều bước.

- Chia nhỏ dữ liệu: Tập dữ liệu được chia thành nhiều phần nhỏ hơn. Mỗi phần sẽ được xử lý song song.
- Tính tổng cục bộ (local sum): Các tổng cục bộ được tính toán trong từng luồng (thread) hoặc lõi (core).
- Giảm dần (Reduction): Sau khi tính tổng cục bộ, kết quả được hợp nhất dần dần để tính tổng toàn bộ.

Dựa vào **hình 3.10** ta có thể thấy rõ từng bước:

### Bước 1: Phân chia dữ liệu

- Dữ liệu ban đầu  $[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$  được chia thành các khối nhỏ hơn:

$$[x_1, x_2], [x_3, x_4], [x_5, x_6], [x_7, x_8].$$

### Bước 2: Tính tổng cục bộ

- Các luồng song song tính tổng từng khối:
  - Tổng khối 1:  $x_1 + x_2$
  - Tổng khối 2:  $x_3 + x_4$
  - Tổng khối 3:  $x_5, x_6$
  - Tổng khối 4:  $x_7, x_8$

### Bước 3: Hợp nhất tổng (Reduction)

- Tổng của hai khối được hợp nhất dần dần:
  - $(x_1 + x_2) + (x_3 + x_4)$
  - $(x_5 + x_6) + (x_7 + x_8)$

Cuối cùng, kết hợp kết quả của 2 khối trên cho ra kết quả cuối:  $[(x_1 + x_2) + (x_3 + x_4)] + [(x_5 + x_6) + (x_7 + x_8)]$

Mảng được chia thành các cặp phần tử liền kề nhau, mỗi đoạn được xử lý bởi một luồng khác nhau. Mỗi luồng tính tổng cục bộ cho đoạn của mình (các luồng này hoạt động song song), sau đó các kết quả cục bộ được kết hợp lại thành một nửa số lượng ban đầu. Quá trình này lặp lại để cho ra tổng cuối cùng. Kỹ thuật này thường sử dụng các thư viện hỗ trợ như OpenMP [1] (cho CPU) để tối ưu hóa.

### Cơ chế xử lý trên CPU:

- Cấp phát bộ nhớ cho tổng từng luồng: Mỗi luồng tính một phần tổng riêng, lưu vào mảng trung gian có kích thước là số luồng của CPU (Sử dụng mảng trung gian để copy kết quả của mảng local trên các thread nhằm hạn chế dùng chung biến mà cho ra kết quả cuối để tránh xung đột).
- Mỗi luồng tính một phần tổng riêng: Mỗi luồng sẽ tính tổng trên một phần dữ liệu dựa vào biến start, end là số phần tử đầu và cuối trong 1 mảng đã được giao cho 1 thread.
- Tính tổng từng phần trong mỗi luồng: Mỗi luồng duyệt qua phần dữ liệu của nó và tính tổng. Sau đó lưu vào mảng trung gian.

- Kết hợp tổng bằng Binary Tree Reduction: Sau khi mỗi luồng hoàn thành, ta cộng tất cả giá trị trong mảng trung gian để lấy tổng cuối cùng.

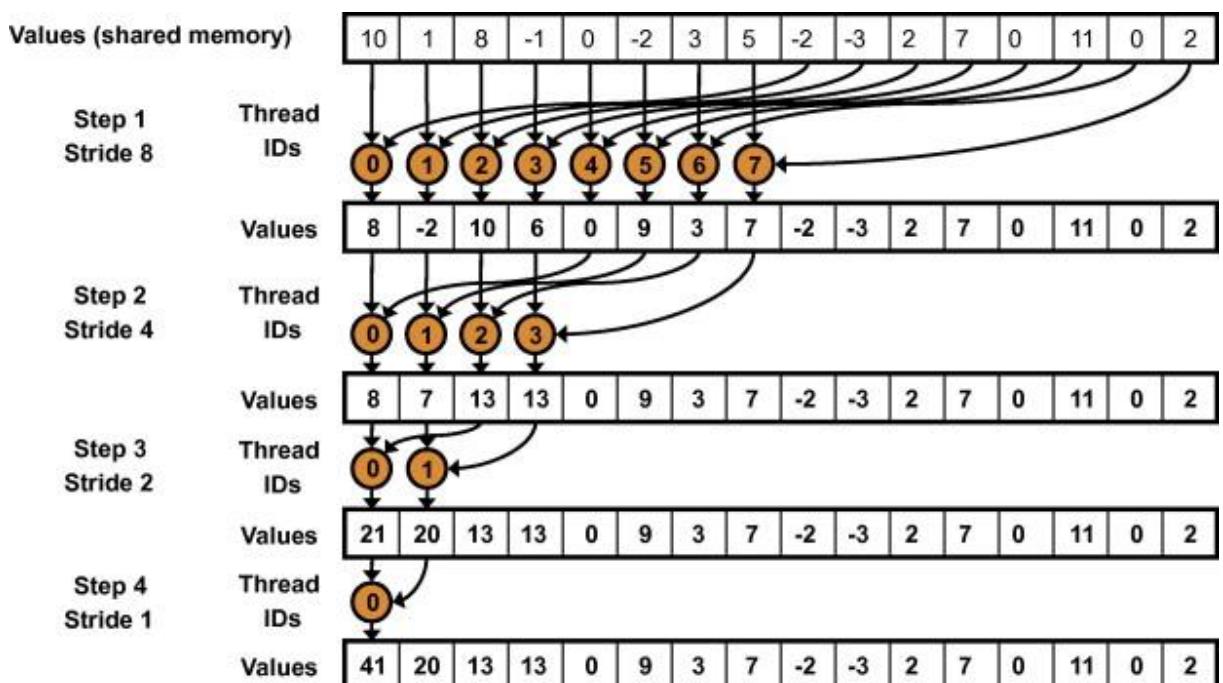
Mã giả Sum Reduce trên CPU:

```

1. Size ← n
2. num_threads ← openmp.omp_get_max_threads() // sử dụng hàm này để
lấy số thread của CPU
3. PARALLEL FOR thread_id ← 0 TO num_threads-1 DO
4. partial_sums[i] ← 0
5. start ← (size * thread_id) // num_threads
6. end ← (size * (thread_id + 1)) // num_threads
7. partial_sums[thread_id] ← sum(data[start to end-1])
8. sum += sum of partial_sums[thread_id]
9. RETURN sum

```

### 3.3.4. Sum Reduce trên GPU



Hình 3.11. Minh họa thuật toán Sum Reduce GPU.

Dựa trên **hình 3.11**, các đặc điểm của Sum Reduce song song trên GPU bao gồm:

- Sử dụng bộ nhớ chia sẻ (Shared Memory): sử dụng biến shared làm bộ nhớ chia sẻ giữa các thread trong cùng một block để giảm truy cập Global Memory giúp tăng tốc xử lý vì shared memory có băng thông cao hơn nhiều so với Global memory
- Mỗi thread xử lý nhiều phần tử: mỗi thread sẽ xử lý 4 phần tử giúp giảm số lần truy cập Global Memory → tăng hiệu suất, sau đó ghi vào biến shared của shared memory để tận dụng bộ nhớ nhanh trên SM (Streaming Multiprocessor).

- Giảm kích thước dữ liệu một cách song song (Parallel Reduction): Mỗi bước, số phần tử cần cộng giảm một nửa bằng cách cộng dồn cặp giá trị gần nhau. Ví dụ:
  - Bước 1:  $(T_1+T_2), (T_3+T_4), (T_5+T_6), (T_7+T_8)$  ← 50% phần tử bị loại
  - Bước 2:  $(T_1+T_2) + (T_3+T_4), (T_5+T_6) + (T_7+T_8)$  ← 75% phần tử bị loại
  - Bước 3:  $[(T_1+T_2) + (T_3+T_4)] + [(T_5+T_6) + (T_7+T_8)]$  ← Chỉ còn 1 giá trị
- Tận dụng song song hóa, giúp giảm số bước từ  $O(n)$  (trên CPU) xuống  $O(\log n)$  (GPU).
- Lưu kết quả cuối cùng vào Global Memory: Kết quả cuối cùng được lưu ở thread id 0 (thread đầu tiên) là kết quả cuối cùng của 1 block được ghi vào mảng kết quả. Mảng kết quả này cần được xử lý tiếp tục nếu có nhiều block.

Mã giả Sum Reduce trên GPU:

```

1. sum ← 0
2. PARALLEL FOR each block DO
3. shared_sum ← 0
4. FOR EACH thread IN block DO
5. shared_sum ← sum(data[thread_index + 0], ..., data[thread_index +
3 * block_size])
6. REDUCE shared_sum IN block
7. IF thread IS first IN block THEN
8. sum ← sum + shared_sum
9. RETURN sum

```

### a) Sum Reduce trên nền tảng CUDA

#### Cơ Chế Xử Lý:

- Mỗi thread chịu trách nhiệm xử lý 4 phần tử của mảng. Cách xử lý này giúp tối ưu bằng thông truy cập bộ nhớ toàn cục (global memory).
- Sử dụng shared memory để lưu trữ các tổng trung gian trong mỗi block, giảm thời gian truy cập so với việc liên tục truy xuất bộ nhớ toàn cục.
- Vòng lặp giảm dần được sử dụng (với lệnh `__syncthreads()`) để hợp nhất các kết quả từ các thread của cùng một block.

#### Quản lý bộ nhớ và đồng bộ hóa:

- Cấp phát bộ nhớ GPU bằng `cudaMalloc` và giải phóng bằng `cudaFree`.
- Sao chép dữ liệu giữa host và device thông qua `cudaMemcpy`.
- Đo thời gian thực thi thông qua `cudaEventRecord`, `cudaEventSynchronize` và `cudaEventElapsedTime`.

**Đánh giá:** Hiệu suất cao do sử dụng C++ với CUDA API – cấp thấp và kiểm soát chi tiết bộ nhớ và đồng bộ hóa giúp tối ưu hóa hiệu suất trên GPU nhưng code phức tạp, khó bảo trì và debug đòi hỏi hiệu năng tối đa và kiểm soát chi tiết.

### b) Sum Reduce sử dụng thư viện CuPy

**Cơ Chế Xử Lý:**

- Sử dụng thư viện Cupy viết bằng Python
- Kernel được biên dịch thông qua cp.RawKernel với cùng cơ chế giảm dần như trong phiên bản CUDA C++. Logic giống như kernel CUDA.

**Quản lý bộ nhớ:**

- Quản lý bộ nhớ GPU tự động qua CuPy Memory Pool giúp tái sử dụng bộ nhớ thay vì gọi cudaMalloc nhiều lần.
- Sử dụng cp.zeros() hoặc cp.array() để cấp phát mảng trên GPU.
- Không cần quản lý bộ nhớ thủ công. Đồng bộ qua cp.cuda.Device(). synchronize(), trừu tượng hóa.

**Đánh giá:** Viết bằng Python nên code dễ đọc, dễ bảo trì và tích hợp với các thư viện khác như NumPy, Pandas, CuPy quản lý bộ nhớ thông qua memory pool và tích hợp với garbage collection của Python, giảm bớt gánh nặng quản lý thủ công nhưng giao diện Python có thể gây Overhead (độ trễ) và là ngôn ngữ lập trình bậc cao nên ít khả năng điều chỉnh tối ưu ở mức thấp như C++.

### c) Sum Reduce sử dụng thư viện RAPIDS cudf

**Cơ chế xử lý:**

- Thay vì tạo mảng CuPy trực tiếp, dữ liệu được sinh ra và lưu trữ trong một cuDF DataFrame giúp tận dụng các chức năng xử lý dữ liệu của cuDF và dễ dàng thao tác với dữ liệu dạng bảng.
- Quá trình thực thi kernel và logic tương tự Cupy và CUDA, sử dụng cp.RawKernel

**Quản lý bộ nhớ:** Sử dụng các hàm đo thời gian và giải phóng bộ nhớ tương tự Cupy, nhưng tích hợp sẵn trong hệ sinh thái RAPIDS (cuDF + CuPy) giúp tăng hiệu quả khi xử lý dữ liệu lớn.

**Đánh giá:** Code dễ hơn Cupy nhờ các abstraction cấp cao, tích hợp liền mạch với các pipeline khoa học dữ liệu hiện có nhưng không linh hoạt bằng CUDA C++ trong việc

tối ưu hóa chuyên sâu và chỉ có thể tốt với các bài toán dữ liệu dạng bảng hoặc một số bài nhất định nên hiệu suất có thể kém hơn nếu thuật toán không phù hợp với các API có sẵn.

### **3.3.5. Kết quả thực nghiệm**

#### **a) Môi trường thực nghiệm:**

được thiết lập trên nền tảng Kaggle, sử dụng GPU NVIDIA T4 với 16GB VRAM:

- Bộ xử lý (CPU): Intel Xeon
- Bộ nhớ RAM: 16GB
- Bộ xử lý đồ họa (GPU): NVIDIA T4 (16GB VRAM)
- Hệ điều hành: Linux (Ubuntu-based trên Kaggle)
- Phiên bản Python: 3.8+

#### **b) Dữ liệu**

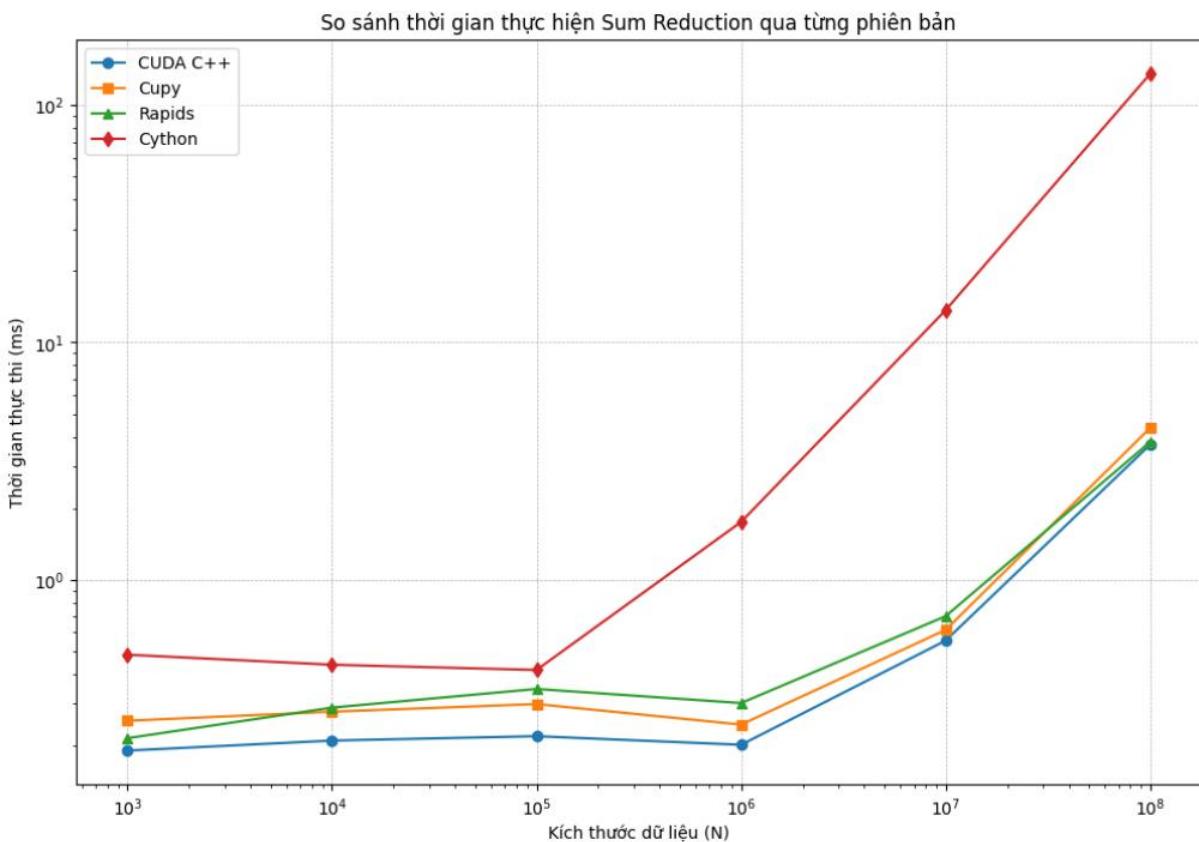
Các mảng một chiều số nguyên với các kích thước sau:

- 1000 phần tử
- 10000 phần tử
- 100000 phần tử
- 1000000 phần tử
- 10000000 phần tử
- 20000000 phần tử

#### **c) Kết quả:**

*Bảng 3.3. Kết quả thời gian chạy của các phiên bản sum reduce (Mili giây).*

| Kích thước<br>mảng | Kết<br>quả | CPU time<br>(Cython) | GPU time<br>(CUDA) | GPU time<br>(CuPy) | GPU time<br>(RAPIDS cudf) |
|--------------------|------------|----------------------|--------------------|--------------------|---------------------------|
| 1,000              | Correct    | 0.4830               | <b>0.1905</b>      | 0.2544             | 0.2148                    |
| 10,000             | Correct    | 0.4382               | <b>0.2099</b>      | 0.2778             | 0.2890                    |
| 100,000            | Correct    | 0.4165               | <b>0.2190</b>      | 0.2992             | 0.3464                    |
| 1,000,000          | Correct    | 1.7521               | <b>0.2015</b>      | 0.2451             | 0.3021                    |
| 10,000,000         | Correct    | 13.6826              | <b>0.5539</b>      | 0.6146             | 0.7019                    |
| 20,000,000         | Correct    | 135.7248             | <b>3.7104</b>      | 4.3635             | 3.8078                    |



**Hình 3.12.** Biểu đồ thời gian chạy Sum Reduce trên Cython, CUDA, Cupy, Rapids.

Từ **bảng 3.3** ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như **hình 3.12** và rút ra nhận xét như sau:

Biểu đồ của hình 3 thể hiện thời gian thực thi (ms) của các phương pháp CUDA C++, CuPy, RAPIDS, và Cython khi thực hiện sum reduction trên GPU và CPU với kích thước đầu vào khác nhau.

- Nhận xét:
  - CUDA C++ cho hiệu suất tốt nhất, đặc biệt với dữ liệu lớn. Với mảng 1,000 phần tử, thời gian thực thi trên CPU là 0.4830 ms, trong khi GPU chỉ mất **0.1905 ms** (CUDA), 0.2544 ms (CuPy), và 0.2148 ms (RAPIDS).
  - CuPy và RAPIDS có hiệu suất tương đương, nhưng chậm hơn một chút so với CUDA C++. Khi xử lý dữ liệu lớn Rapids có thể nhanh hơn một chút so với Cupy.
  - Cython (chạy trên CPU) chậm hơn đáng kể khi kích thước dữ liệu tăng. Khi kích thước mảng đạt 10 triệu phần tử, CPU mất 13.6826 ms, cao gấp ~25 lần CUDA, ~22 lần CuPy, và ~19.5 lần RAPIDS. Với 20 triệu phần tử, CPU mất 135.7248 ms, cao hơn ~36 lần CUDA, ~31 lần CuPy, và ~35 lần RAPIDS.

## 3.4. Bài toán nhân ma trận

### 3.4.1. Mô tả bài toán

Phép nhân ma trận là một trong những phép toán đại số tuyến tính cơ bản và quan trọng nhất. Phép nhân ma trận được sử dụng cho nhiều phép tính khoa học trong nhiều lĩnh vực khác nhau và bất kỳ sự giảm thời gian tính toán nào cũng sẽ cực kỳ có lợi. [10]

### 3.4.2. Ý tưởng

Phép nhân hai ma trận chỉ thực hiện được khi số cột của ma trận bên trái bằng số dòng của ma trận bên phải. Nếu ma trận A có kích thước mxn và ma trận B có kích thước nxp, thì ma trận tích AB có kích thước mxp có phần tử đứng ở hàng thứ i, cột thứ j xác định bởi:

$$C_{i,j} = A_{i,1} \cdot B_{1,j} + A_{i,2} \cdot B_{2,j} + \dots + A_{i,n} \cdot B_{n,j}$$

với mọi cặp  $i = 1..m; j = 1..p$ .

Ví dụ:

$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix} \times \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} (1 \times 3 + 0 \times 2 + 2 \times 1) & (1 \times 1 + 0 \times 1 + 2 \times 0) \\ (-1 \times 3 + 3 \times 2 + 1 \times 1) & (-1 \times 1 + 3 \times 1 + 1 \times 0) \end{bmatrix}$$

$$= \begin{bmatrix} 5 & 1 \\ 4 & 2 \end{bmatrix}$$

### 3.4.3. Nhân ma trận CPU

Nhân ma trận là một trong những bài toán cơ bản trong tính toán khoa học và xử lý dữ liệu. Khi thực hiện nhân hai ma trận trên CPU, thuật toán sẽ duyệt qua từng phần tử của hai ma trận đầu vào, thực hiện phép nhân và cộng dồn kết quả để tạo ra ma trận kết quả. Vì CPU xử lý các phép toán một cách tuần tự, nên khi kích thước ma trận tăng lên, thời gian thực hiện cũng tăng theo do số lượng phép nhân và cộng tăng theo cấp số nhân.

Bước 1: Xác định điều kiện nhân ma trận

- Hai ma trận có thể nhân được với nhau nếu số cột của ma trận thứ nhất bằng số hàng của ma trận thứ hai.
- Nếu ma trận A có kích thước là  $M \times N$  và B là ma trận có kích thước  $N \times P$ , thì ma trận kết quả C sẽ có kích thước là  $M \times P$ .

Bước 2: Thực hiện phép nhân theo công thức

- Mỗi phần tử tại vị trí  $C[i][j]$  trong ma trận kết quả được tính bằng cách nhân từng phần tử của hàng thứ  $i$  của ma trận A với từng phần tử của cột thứ  $j$  của ma trận B. Sau đó cộng dồn các giá trị này.

Bước 3: Duyệt tuần tự để tính toán từng phần tử

- Lặp qua từng hàng của ma trận A.
- Với mỗi hàng của A, A, lặp qua từng cột của ma trận B.
- Với mỗi cặp (hàng, cột), tính tổng tích của các phần tử tương ứng.

Mã giả:

```

1. Matrix_Multiplication(A, B, C, M, K, N)
2. for i ← 0 to M - 1:
3. for j ← 0 to N - 1:
4. C[i][j] ← 0
5. for k ← 0 to K - 1:
6. C[i][j] ← C[i][j] + A[i][k] * B[k][j]

```

Độ phức tạp thuật toán:

Thuật toán nhân hai ma trận A kích thước  $M \times K$  và B kích thước  $K \times N$  có độ phức tạp  $O(M * K * N)$  do:

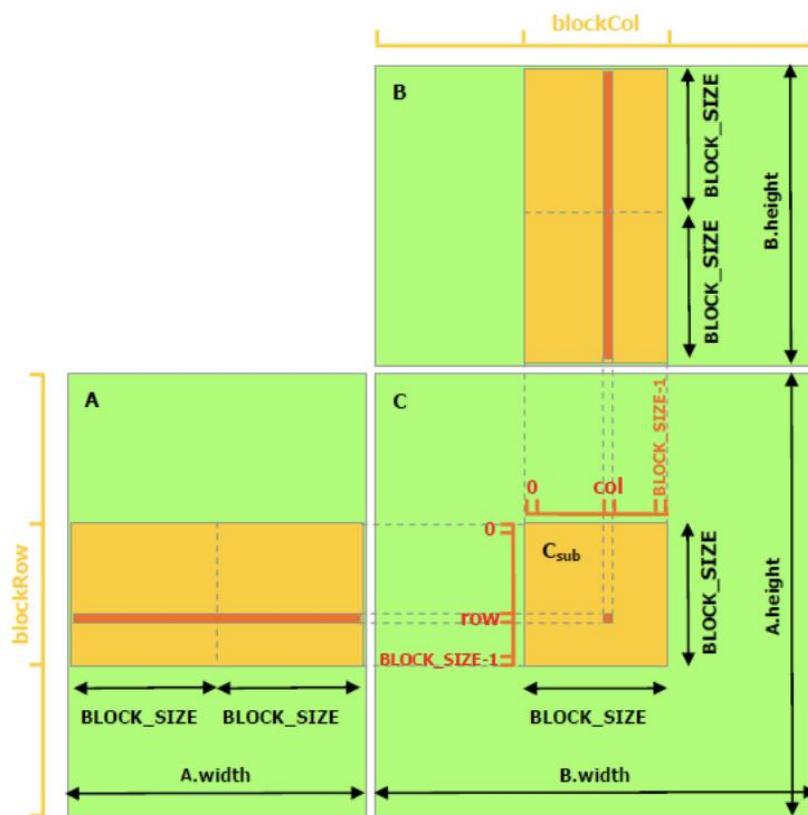
- Duyệt qua từng phần tử của ma trận kết quả C: Có  $M * N$  phần tử cần tính.
- Tính mỗi phần tử  $C[i][j]$ : Cần thực hiện một vòng lặp qua  $K$  phần tử để tính tổng tích của hàng  $i$  của A và cột  $j$  của B, dẫn đến  $O(K)$  cho mỗi phần tử.
- Tổng quát:  $O(M * K * N)$ .

Hạn chế:

- Tốn thời gian cho ma trận lớn: Khi  $M, K, N$  lớn, số phép nhân và cộng tăng nhanh, khiến thuật toán chậm.
- Không tối ưu cache: Khi duyệt ma trận, việc truy cập vào các phần tử không liên tục có thể gây ra cache miss, làm giảm hiệu suất.

#### **3.4.4. Nhân ma trận GPU**

Đối với thuật toán nhân ma trận trên GPU, ta vẫn giữ nguyên ý tưởng của thuật toán nhân ma trận truyền thống. Tuy nhiên, thay vì duyệt từng phần tử theo cách tuần tự trên CPU, ta tận dụng khả năng tính toán song song của GPU để xử lý nhiều phần tử cùng lúc. Mỗi luồng (thread) sẽ đảm nhiệm việc tính toán một phần tử trong ma trận kết quả.



Hình 3.13. Minh họa nhân ma trận trên GPU.

Như **hình 3.13**, điều này giúp tăng tốc đáng kể so với CPU khi xử lý ma trận lớn. Các bước cụ thể để thực hiện nhân ma trận trên GPU:

Bước 1: Ánh xạ công việc đến các luồng trên GPU:

- Mỗi luồng (thread) trên GPU sẽ chịu trách nhiệm tính toán một phần tử của ma trận kết quả.
- Sử dụng mô hình grid - block - thread để phân chia công việc:
  - Grid: Chứa nhiều blocks, mỗi block xử lý một phần của ma trận.
  - Block: Chứa nhiều threads, mỗi thread tính một phần tử trong block.
  - Thread: Xác định vị trí hàng và cột của phần tử cần tính.

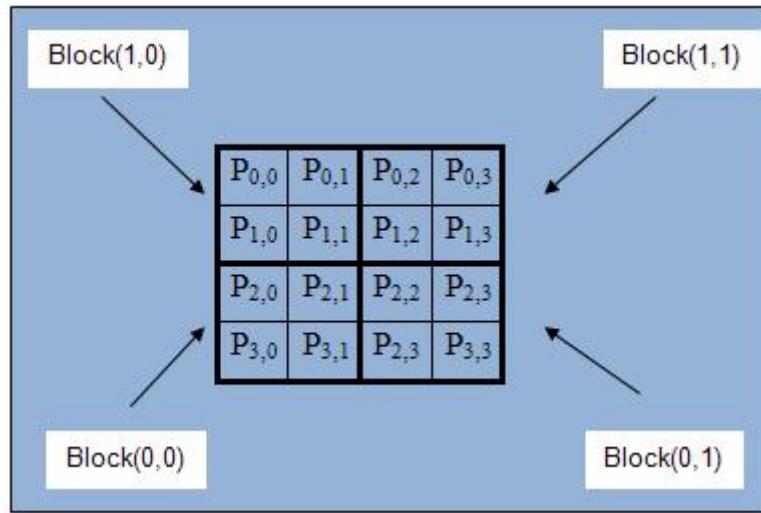
Bước 2: Tính giá trị của từng phần tử trong ma trận kết quả:

- Mỗi thread xác định chỉ số hàng  $i$  và cột  $j$  mà nó sẽ tính toán.
- Thực hiện phép nhân từng cặp phần tử từ hàng thứ  $i$  của ma trận A và cột thứ  $j$  của ma trận B, sau đó cộng dồn lại để tính  $C[i][j]$ .

Bước 3: Ghi kết quả vào bộ nhớ toàn cục của GPU:

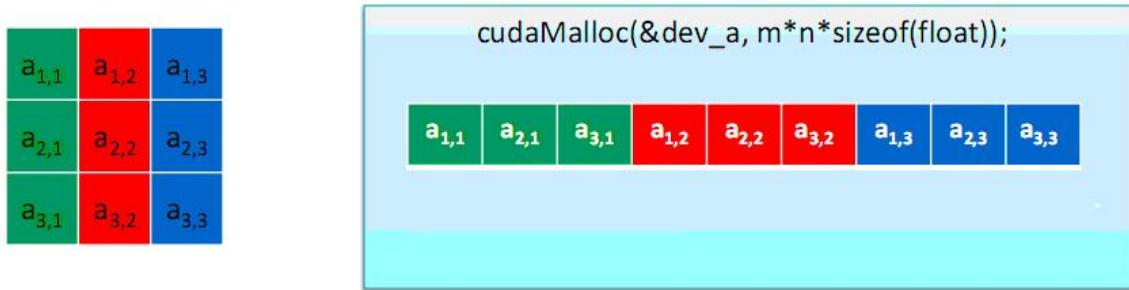
- Sau khi tính xong  $C[i][j]$ , mỗi thread ghi kết quả vào ma trận C nằm trong bộ nhớ toàn cục (global memory) của GPU.

Giả sử ta có một ma trận  $4 \times 4$  như **hình 3.14** có thể chia ma trận thành 4 Block, mỗi Block 4 threads. Mỗi Thread sẽ tính một phần tử của ma trận kết quả.



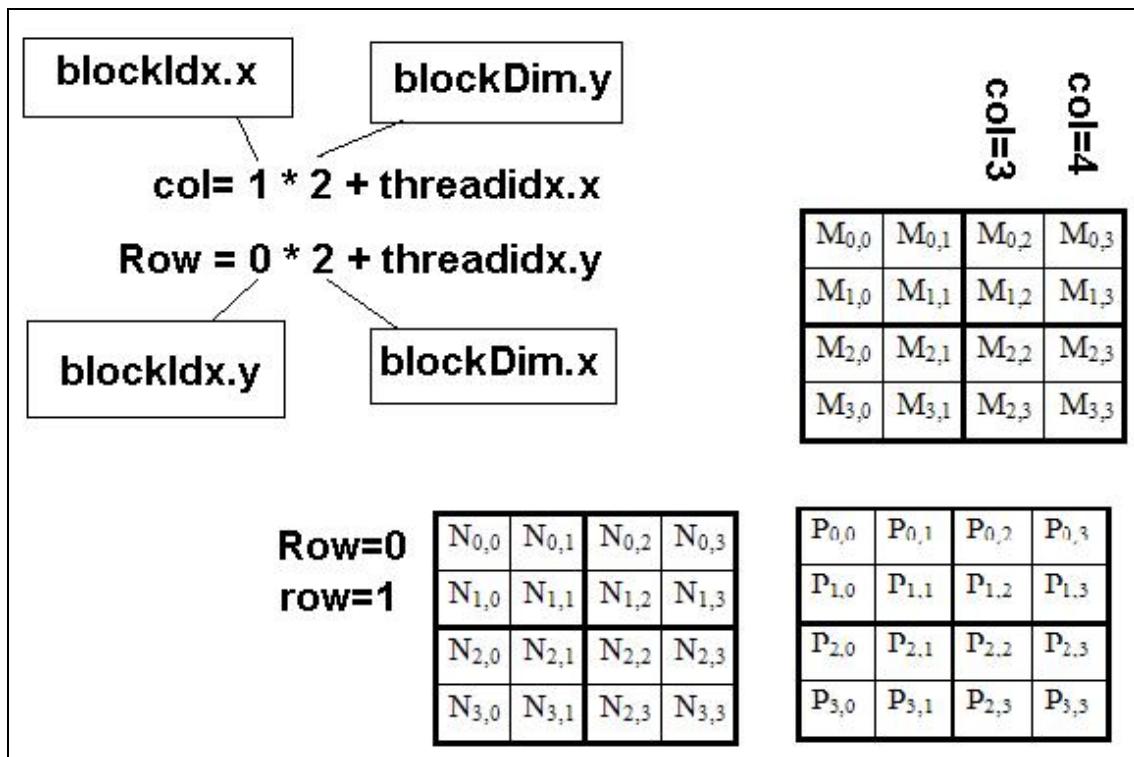
**Hình 3.14.** Chia block và thread cho bài toán nhân ma trận.

Khi cấp phát bộ nhớ cho ma trận thì một ma trận sẽ trở thành một mảng chứa các phần tử của ma trận.



**Hình 3.15.** Cách CUDA lưu trữ ma trận.

Theo **hình 3.15** ta có thể thấy từ một ma trận ban đầu gồm các hàng và các cột CUDA sẽ lưu trữ thành một mảng một chiều theo thứ tự cột. Các phần tử trong cùng một cột sẽ được lưu liên tiếp nhau trong bộ nhớ.



Hình 3.16. Xác định dòng và cột của ma trận trong CUDA.

Theo **hình 3.16** mỗi phần tử của ma trận được xử lý bởi một thread, và vị trí của phần tử trong ma trận được xác định dựa trên blockIdx, blockDim, và threadIdx như sau:

- Dòng (Row) của phần tử trong ma trận:

$$\text{row} = \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y}$$

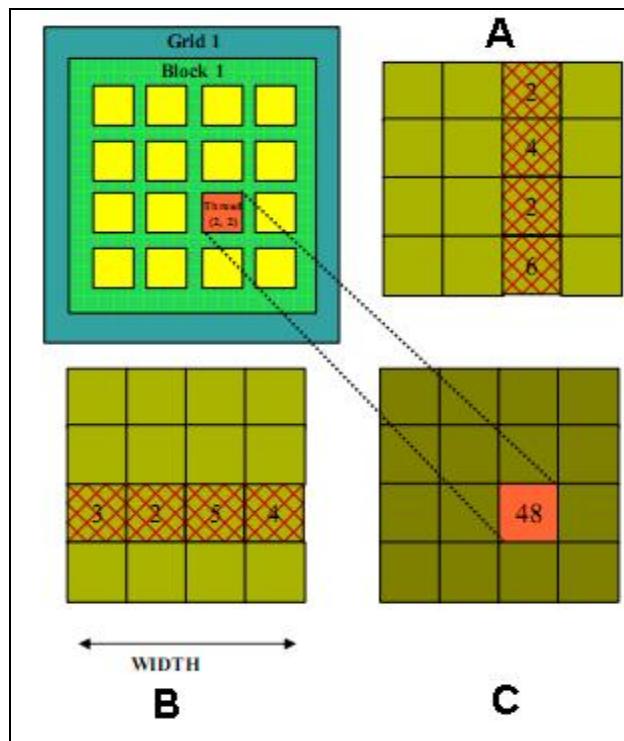
Điều này có nghĩa là chỉ số dòng được tính bằng chỉ số khối theo chiều dọc (blockIdx.y) nhân với số lượng thread trong một block theo chiều dọc (blockDim.y), cộng với chỉ số thread trong block đó theo chiều dọc (threadIdx.y).

- Cột (Col) của phần tử trong ma trận:

$$\text{col} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

Chỉ số cột được tính tương tự, dựa trên blockIdx.x, blockDim.x và threadIdx.x.

Mỗi thread sẽ tính toán một giá trị của ma trận C.



**Hình 3.17.** Mô hình tính toán của thread trong nhân ma trận CUDA.

Theo **hình 3.17** Mỗi thread sẽ đảm nhận việc tính toán một phần tử của ma trận kết quả C như sau:

Bước 1: Xác định vị trí thread:

- Mỗi thread trong CUDA được gán một vị trí duy nhất trong lưới (grid) và khối (block). Trong hình, thread đang xét có vị trí (2, 2) trong Block 1 của Grid 1.

Bước 2: Tính toán giá trị kết quả:

- Thread này chịu trách nhiệm tính toán một phần tử duy nhất của ma trận kết quả, được biểu thị bằng ô màu đỏ có giá trị 48 trong ma trận C.
- Để tính giá trị này, thread sẽ truy cập vào các phần tử tương ứng của hai ma trận đầu vào (A và B).
- Cụ thể, thread sẽ nhân các phần tử trong hàng thứ 2 của ma trận A với các phần tử trong cột thứ 2 của ma trận B, sau đó cộng dồn các kết quả lại.
- Trong hình, các phần tử được sử dụng để tính toán được tô màu cam sọc chéo.

Bước 3: Lưu trữ kết quả:

- Sau khi tính toán xong, thread sẽ lưu trữ kết quả (48) vào vị trí tương ứng trong ma trận kết quả C.

### a) Nhân ma trận trên nền tảng CUDA

Trong phần này, chúng ta sử dụng GPU trên nền tảng CUDA để thực hiện thuật toán nhân ma trận bằng cách phân chia công việc thành nhiều thread song song. Quá trình này được triển khai thông qua CUDA kernel, trong đó mỗi thread chịu trách nhiệm tính toán một phần tử của ma trận kết quả.

Mã giả nhân ma trận trên GPU với nền tảng CUDA:

```
Nhân ma trận trên GPU CUDA
Input: Ma trận A (kích thước mxk), Ma trận B (kích thước kxn)
Output: Ma trận C (kích thước mxn)
1. Cấp phát bộ nhớ trên GPU cho dev_A, dev_B, và dev_C
2. Sao chép ma trận A vào dev_A
3. Sao chép ma trận B vào dev_B
4. Thiết lập blockDim dưới dạng khối hai chiều của các luồng (threads)
5. Thiết lập gridDim dựa trên kích thước của ma trận C
6. Khởi chạy CUDA kernel để thực hiện phép nhân ma trận
7. Tính chỉ số hàng: row = blockIdx.y * blockDim.y + threadIdx.y
8. Tính chỉ số cột: col = blockIdx.x * blockDim.x + threadIdx.x
9. If row < m and col < n, then:
10. Khởi tạo value = 0
11. For i = 0 to k - 1, do:
12. value += dev_A[row, i] * dev_B[i, col]
13. Gán value vào dev_C[row, col]
14. End if
15. Sao chép dev_C từ GPU về bộ nhớ máy chủ thành ma trận C
16. Giải phóng bộ nhớ trên GPU (dev_A, dev_B, dev_C)
17. Return matrix C
```

Cấp phát bộ nhớ trên GPU:

- Cấp phát vùng nhớ trên GPU để lưu trữ ma trận A, B và C (dev\_A, dev\_B, dev\_C).

Gọi kernel CUDA để thực hiện Merge Sort song song:

- Chuyển dữ liệu của ma trận A từ CPU lên dev\_A trên GPU.
- Chuyển dữ liệu của ma trận B từ CPU lên dev\_B trên GPU.

Cấu hình lưới và khối tính toán

- Thiết lập blockDim: Kích thước của mỗi khối (block) gồm nhiều thread.
- Thiết lập gridDim: Kích thước của lưới (grid) gồm nhiều block, đảm bảo bao phủ toàn bộ ma trận C.

Gọi kernel CUDA để thực hiện nhân ma trận

- Xác định vị trí của mỗi thread:

- Chỉ số hàng của ma trận kết quả:  $\text{row} = \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y}$
- Chỉ số cột của ma trận kết quả:  $\text{col} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$
- Tính toán giá trị của phần tử  $C[\text{row}, \text{col}]$  nếu nó nằm trong phạm vi hợp lệ:
  - Khởi tạo  $\text{value} = 0$ .
  - Duyệt qua từng phần tử trong hàng  $\text{row}$  của  $A$  và cột  $\text{col}$  của  $B$ :  $\text{value} += A[\text{row}, i] * B[i, \text{col}]$ , với  $i$  chạy từ 0 đến  $k - 1$
  - Gán  $\text{value}$  vào phần tử tương ứng của ma trận kết quả  $C$ .

Sao chép kết quả từ GPU về CPU

- Chuyển dữ liệu từ  $\text{dev\_C}$  về ma trận  $C$  trên CPU.

Giải phóng bộ nhớ trên GPU

- Giải phóng  $\text{dev\_A}$ ,  $\text{dev\_B}$ ,  $\text{dev\_C}$  để tránh lãng phí tài nguyên.

Cài đặt nhân kernel cho nhân ma trận CUDA:

```
// Matrix multiplication kernel
__global__ void matrixMultiply(float* A, float* B, float* C,
 int numARows, int numAColumns,
 int numBRows, int numBColumns,
 int numCRows, int numCColumns) {
 int row = blockIdx.y * blockDim.y + threadIdx.y;
 int col = blockIdx.x * blockDim.x + threadIdx.x;

 if ((row < numCRows) && (col < numCColumns)) {
 float value = 0;
 #pragma unroll
 for (int k = 0; k < numAColumns; ++k) {
 value += A[row * numAColumns + k] * B[k * numBColumns +
col];
 }
 C[row * numCColumns + col] = value;
 }
}
```

Ngoài ra ta vẫn còn có thể cải tiến nhân ma trận trên GPU bằng cách sử dụng shared memory, sau đây là cài đặt nhân ma trận trên GPU sử dụng shared memory:

```
// Matrix multiplication kernel using shared memory
__global__ void matrixMultiplyShared(float* A, float* B, float* C,
 int numARows, int numAColumns,
 int numBRows, int numBColumns,
 int numCRows, int numCColumns) {
 __shared__ float sharedA[TILE_WIDTH][TILE_WIDTH];
 __shared__ float sharedB[TILE_WIDTH][TILE_WIDTH];

 int bx = blockIdx.x;
```

```

int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;

// Global row and column indices
int row = by * TILE_WIDTH + ty;
int col = bx * TILE_WIDTH + tx;

float value = 0;

// Loop over tiles
for (int t = 0; t < (numAColumns-1)/TILE_WIDTH + 1; ++t) {
 // Load data into shared memory
 if (row < numARows && (t*TILE_WIDTH + tx) < numAColumns)
 sharedA[ty][tx] = A[row * numAColumns + t * TILE_WIDTH +
tx];
 else
 sharedA[ty][tx] = 0.0f;

 if ((t*TILE_WIDTH + ty) < numBRows && col < numBColumns)
 sharedB[ty][tx] = B[(t * TILE_WIDTH + ty) * numBColumns +
col];
 else
 sharedB[ty][tx] = 0.0f;

 __syncthreads();

 // Compute partial dot product within tile
 #pragma unroll
 for (int k = 0; k < TILE_WIDTH; ++k) {
 value += sharedA[ty][k] * sharedB[k][tx];
 }
 __syncthreads();
}

// Write result
if (row < numCRows && col < numCColumns) {
 C[row * numCColumns + col] = value;
}
}
}

```

Bằng việc sử dụng shared memory ta có thể cải thiện về hiệu suất của nhân ma trận trên GPU. Cách thức hoạt động của nhân ma trận sử dụng shared memory như sau:

- Ma trận được chia thành các khối con (tile) nhỏ kích thước  $TILE\_WIDTH \times TILE\_WIDTH$ .
- Mỗi block xử lý một ô (tile) của ma trận kết quả.
- Dữ liệu trong tile được tải vào shared memory trước khi tính toán, sau đó được chia sẻ giữa các thread trong block.

- Các thread tính toán giá trị của C[row, col] bằng cách truy xuất từ shared memory, giúp giảm số lần truy cập Global Memory.

### b) Nhân ma trận sử dụng thư viện CuPy (Python)

CuPy là một thư viện mạnh mẽ giúp lập trình GPU dễ dàng hơn trong Python, có API tương tự NumPy nhưng hoạt động trên GPU. Việc sử dụng CuPy thay vì viết CUDA thủ công mang lại nhiều lợi ích:

#### Tích hợp dễ dàng với Python:

- Python là một ngôn ngữ phổ biến, dễ đọc, dễ triển khai.
- CuPy cung cấp API gần giống NumPy, giúp tận dụng GPU mà không cần thay đổi nhiều code.
- Cho phép tái sử dụng kernel CUDA tùy chỉnh thông qua hàm cp.RawKernel.

#### Tận dụng GPU mà không cần quản lý bộ nhớ thủ công:

- Không cần viết code CUDA C++ phức tạp.
- Quản lý bộ nhớ GPU tự động, giảm lỗi liên quan đến cấp phát và giải phóng bộ nhớ.

#### Hiệu năng cao nhưng đơn giản:

- CuPy tối ưu các phép toán trên GPU bằng CUDA mà không cần lập trình phức tạp.
- Thích hợp cho các bài toán xử lý dữ liệu lớn như Merge Sort.

Cài đặt kernel cho nhân ma trận CuPy:

```
extern "C" __global__ void matrixMultiplyShared(
 const float* A, const float* B, float* C,
 int numARows, int numAColumns,
 int numBRows, int numBColumns,
 int numCRows, int numCColumns) {

 __shared__ float sharedA[32][32];
 __shared__ float sharedB[32][32];

 int bx = blockIdx.x;
 int by = blockIdx.y;
 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int row = by * 32 + ty;
 int col = bx * 32 + tx;

 float value = 0;
```

```

for (int t = 0; t < (numAColumns-1)/32 + 1; ++t) {
 if (row < numARows && (t*32 + tx) < numAColumns)
 sharedA[ty][tx] = A[row * numAColumns + t * 32 + tx];
 else
 sharedA[ty][tx] = 0.0f;

 if ((t*32 + ty) < numBRows && col < numBColumns)
 sharedB[ty][tx] = B[(t * 32 + ty) * numBColumns + col];
 else
 sharedB[ty][tx] = 0.0f;

 __syncthreads();

#pragma unroll
 for (int k = 0; k < 32; ++k) {
 value += sharedA[ty][k] * sharedB[k][tx];
 }
 __syncthreads();
}

if (row < numCRows && col < numCColumns) {
 C[row * numCColumns + col] = value;
}
}

```

### c) Nhận ma trận sử dụng RAPIDS (cudf)

cuDF là một thư viện xử lý dữ liệu hiệu năng cao của RAPIDS, được thiết kế để chạy trên GPU bằng CUDA. Sử dụng cuDF cho Merge Sort thay vì CuPy hoặc CUDA thuận tiện mang lại nhiều lợi ích:

#### Tối ưu hóa cho xử lý dữ liệu lớn:

- cuDF được thiết kế để làm việc với tập dữ liệu lớn tương tự Pandas nhưng chạy trên GPU.
- Merge Sort có thể tận dụng GPU hiệu quả mà không cần lập trình CUDA phức tạp.

#### Để dàng sử dụng với Python:

- API của cuDF rất giống Pandas, giúp việc triển khai thuật toán trên GPU trở nên đơn giản hơn.
- Không cần quản lý bộ nhớ thủ công như trong CUDA thuận tiện.

#### Tích hợp với hệ sinh thái RAPIDS:

- Có thể kết hợp với cuML (học máy trên GPU) hoặc Dask-cuDF để xử lý dữ liệu phân tán.

- Phù hợp với các bài toán xử lý dữ liệu lớn trên GPU.

Cài đặt kernel nhân ma trận với cudf của RAPIDS:

```
extern "C" __global__ void matrixMultiplyShared(
 const float* A, const float* B, float* C,
 int numARows, int numAColumns,
 int numBRows, int numBColumns,
 int numCRows, int numCColumns) {

 __shared__ float sharedA[32][32];
 __shared__ float sharedB[32][32];

 int bx = blockIdx.x;
 int by = blockIdx.y;
 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int row = by * 32 + ty;
 int col = bx * 32 + tx;

 float value = 0;

 for (int t = 0; t < (numAColumns-1)/32 + 1; ++t) {
 if (row < numARows && (t*32 + tx) < numAColumns)
 sharedA[ty][tx] = A[row * numAColumns + t * 32 + tx];
 else
 sharedA[ty][tx] = 0.0f;

 if ((t*32 + ty) < numBRows && col < numBColumns)
 sharedB[ty][tx] = B[(t * 32 + ty) * numBColumns + col];
 else
 sharedB[ty][tx] = 0.0f;

 __syncthreads();

 #pragma unroll
 for (int k = 0; k < 32; ++k) {
 value += sharedA[ty][k] * sharedB[k][tx];
 }
 __syncthreads();
 }

 if (row < numCRows && col < numCColumns) {
 C[row * numCColumns + col] = value;
 }
}
```

Sau khi hoàn thành kernel, ta sẽ tiến hành viết một hàm Python để thực hiện Merge Sort trên GPU và tích hợp hàm đó vào cudf.

```
Custom Matrix Multiplication Function
```

```

def matmul(self, other):
 if not isinstance(other, cudf.DataFrame):
 raise ValueError("Matrix multiplication requires another
cuDF DataFrame")

 # Start total execution time measurement
 total_start = time.time()

 # 1. Time for Memory Allocation (GPU)
 start_alloc = time.time()
 A_gpu = cp.empty(self.shape, dtype=cp.float32)
 B_gpu = cp.empty(other.shape, dtype=cp.float32)
 C_gpu = cp.empty((self.shape[0], other.shape[1]),
 dtype=cp.float32)
 end_alloc = time.time()

 # 2. Time for CPU → GPU Copy
 start_copy_h2d = time.time()
 A_gpu[:] = cp.array(self.values, dtype=cp.float32)
 B_gpu[:] = cp.array(other.values, dtype=cp.float32)
 end_copy_h2d = time.time()

 # Matrix dimensions
 N, M = A_gpu.shape
 M_B, P = B_gpu.shape
 if M != M_B:
 raise ValueError("Matrix dimensions do not match for
multiplication")

 # 3. Time for Kernel Execution
 threads_per_block = (16, 16)
 blocks_per_grid = ((P + 15) // 16, (N + 15) // 16)
 start_kernel = time.time()
 matrix_mult_kernel(blocks_per_grid, threads_per_block, (A_gpu,
B_gpu, C_gpu, N, M, P))
 cp.cuda.Device(0).synchronize() # Ensure GPU computation
finishes
 end_kernel = time.time()

 # 4. Time for GPU → CPU Copy
 start_copy_d2h = time.time()
 result = C_gpu.get()
 end_copy_d2h = time.time()

 # End total execution time measurement
 total_end = time.time()

 # Convert back to cuDF DataFrame
 result_df = cudf.DataFrame(result)

```

```

Convert all times to milliseconds
alloc_time = (end_alloc - start_alloc) * 1000
copy_h2d_time = (end_copy_h2d - start_copy_h2d) * 1000
kernel_time = (end_kernel - start_kernel) * 1000
copy_d2h_time = (end_copy_d2h - start_copy_d2h) * 1000
total_time = (total_end - total_start) * 1000

Print Timing Information
print(f"Memory Allocation Time: {alloc_time:.3f} ms")
print(f"CPU → GPU Copy Time: {copy_h2d_time:.3f} ms")
print(f"Kernel Execution Time: {kernel_time:.3f} ms")
print(f"GPU → CPU Copy Time: {copy_d2h_time:.3f} ms")
print(f"Total Execution Time: {total_time:.3f} ms")

return result_df

Monkey-patch cuDF DataFrame to add the custom method
cudf.DataFrame.matmul = matmul

```

Sau khi đã tích hợp ta có thể khởi tạo cudf dataframe và áp dụng merge sort trên cudf của RAPIDS.

### **3.4.5. Kết quả thực nghiệm**

#### **a) Môi trường thực nghiệm:**

Môi trường thực nghiệm được thiết lập trên nền tảng Kaggle, sử dụng GPU NVIDIA T4 với 16GB VRAM:

- Bộ xử lý (CPU): Intel Xeon
- Bộ nhớ RAM: 16GB
- Bộ xử lý đồ họa (GPU): NVIDIA T4 (16GB VRAM)
- Hệ điều hành: Linux (Ubuntu-based trên Kaggle)
- Phiên bản Python: 3.8+

#### **b) Dữ liệu:**

Ta sẽ khởi tạo ngẫu nhiên các ma trận số nguyên dương với các kích thước sau và lưu vào các file tương ứng:

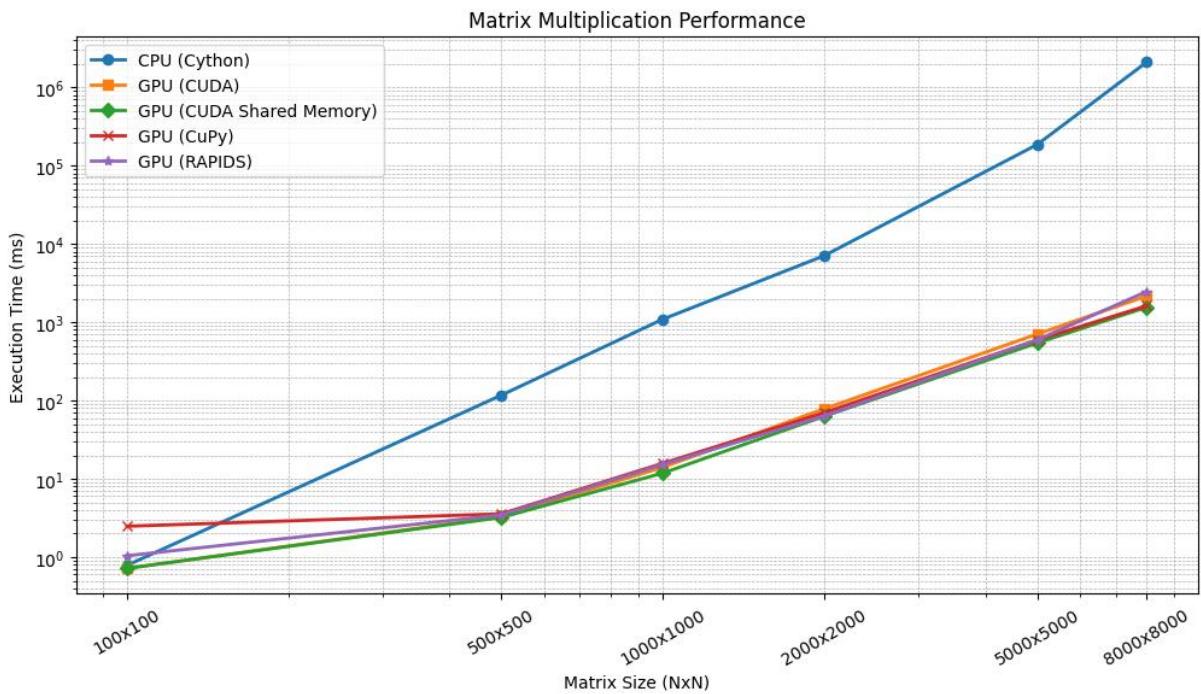
- 100x100 - Matrix\_100x100.INP
- 500x500 - Matrix\_500x500.INP
- 1000x1000 - Matrix\_1000x1000.INP
- 2000x2000 - Matrix\_2000x2000.INP
- 5000x5000 - Matrix\_5000x5000.INP
- 8000x8000 - Matrix\_8000x8000.INP

### c) Kết quả:

*Bảng 3.4. Kết quả thời gian chạy của các phiên bản nhân ma trận (Mili giây).*

| Kích thước ma trận | Kết quả | CPU time (Cython) | GPU time (CUDA) | GPU time (CUDA shared memory) | GPU time (CuPy) | GPU time (RAPIDS cudf) |
|--------------------|---------|-------------------|-----------------|-------------------------------|-----------------|------------------------|
| 100x100            | Correct | 0.785             | <b>0.718</b>    | 0.720                         | 2.481           | 1.042                  |
| 500x500            | Correct | 117.389           | 3.303           | <b>3.217</b>                  | 3.574           | 3.467                  |
| 1000x1000          | Correct | 1094.072          | 14.039          | <b>11.843</b>                 | 15.800          | 15.285                 |
| 2000x2000          | Correct | 7078.108          | 78.436          | <b>62.904</b>                 | 69.872          | 63.401                 |
| 5000x5000          | Correct | 186787.659        | 707.448         | <b>544.035</b>                | 595.061         | 597.800                |
| 8000x8000          | Correct | 2106937.052       | 2164.860        | <b>1557.420</b>               | 1625.026        | 2472.573               |

**Biểu đồ:**



*Hình 3.18. Biểu đồ biểu diễn thời gian chạy của các phiên bản nhân ma trận.*

Từ **bảng 3.4**, ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như **hình 3.18** và rút ra nhận xét như sau:

Hiệu suất trên CPU (Cython) vs. GPU

- Hiệu suất của CPU (Cython) giảm mạnh khi kích thước ma trận tăng. Ở kích thước  $1000 \times 1000$ , CPU mất 1094 mili giây, trong khi GPU chỉ mất khoảng 11-16 mili giây, cho thấy lợi thế rõ ràng của GPU.
- Ở kích thước  $8000 \times 8000$ , GPU nhanh hơn CPU hơn 1000 lần, khẳng định rằng CPU không phải là lựa chọn phù hợp cho bài toán nhân ma trận kích thước lớn.

### So sánh các phiên bản GPU

- CUDA Global Memory vs. Shared Memory:
  - Phiên bản sử dụng Shared Memory luôn nhanh hơn Global Memory, đặc biệt rõ rệt khi kích thước ma trận lớn (ví dụ:  $8000 \times 8000$ , Shared Memory nhanh hơn Global Memory khoảng 28%).
  - Điều này khẳng định rằng việc tối ưu bộ nhớ trên GPU có tác động lớn đến hiệu suất.
- CuPy vs. CUDA thuần túy:
  - CuPy có hiệu suất thấp hơn một chút so với CUDA trực tiếp (~10-15%), do overhead của thư viện.
  - Tuy nhiên, CuPy vẫn là một lựa chọn tốt nếu ưu tiên đơn giản hóa code mà không cần tối ưu hóa quá sâu.
- RAPIDS cuDF:
  - RAPIDS có hiệu suất tương đương CUDA Shared Memory khi ma trận  $\leq 2000 \times 2000$  nhưng chậm hơn khi  $8000 \times 8000$ , có thể do overhead xử lý dữ liệu theo kiểu dataframe.
  - Điều này cho thấy RAPIDS phù hợp hơn cho xử lý dữ liệu lớn dạng dataframe hơn là bài toán thuần toán học như nhân ma trận.

### Xu hướng tổng quát

- GPU có lợi thế rõ ràng so với CPU, đặc biệt với ma trận lớn.
- Shared Memory cải thiện hiệu suất đáng kể so với Global Memory.
- CuPy và RAPIDS tiện lợi nhưng không nhanh bằng CUDA tối ưu trực tiếp.
- Ở kích thước cực lớn ( $8000 \times 8000$ ), tối ưu bộ nhớ trên GPU là yếu tố quyết định hiệu suất.

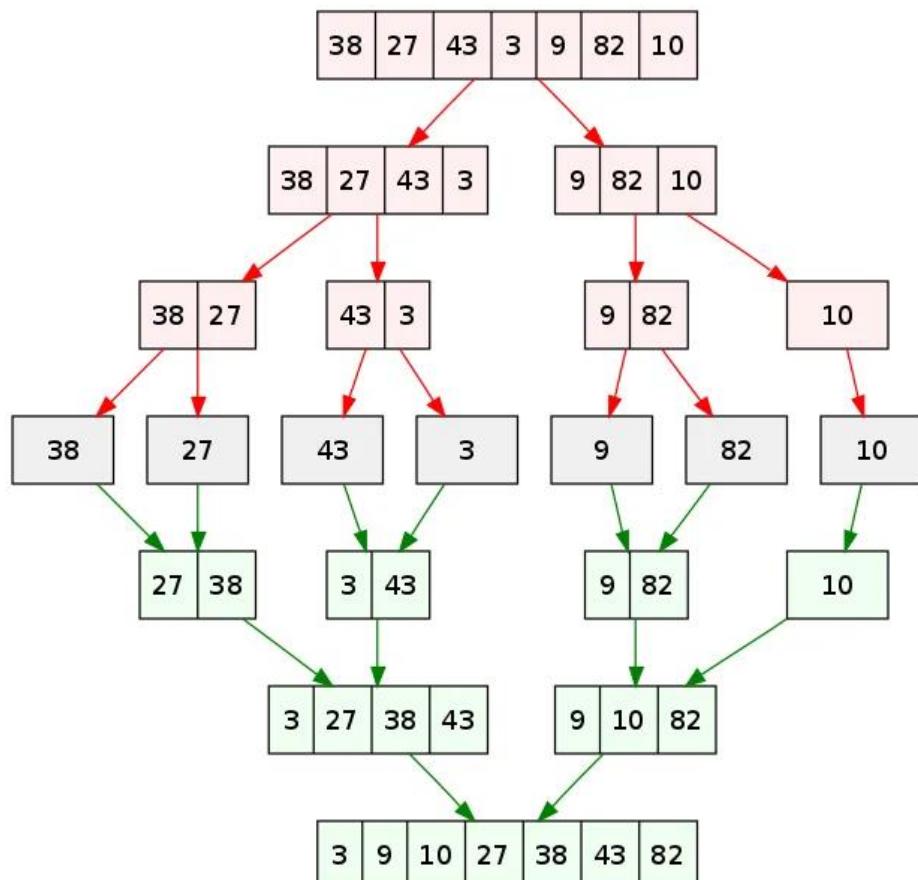
## CHƯƠNG 4: MỘT SỐ BÀI TOÁN KHÁC

### 4.1. Bài toán sắp xếp (Merge Sort)

#### 4.1.1. Mô tả bài toán

Merge sort hay sắp xếp trộn là một thuật toán sắp xếp dữ liệu theo một trật tự nhất định nào đó. Merge sort là một thuật toán hiệu quả và dễ hiểu so với các thuật toán chia để trị khác. Do đó đây được xem là một đại diện tiêu biểu cho các thuật toán sắp xếp và thường được sử dụng để giới thiệu cách tiếp cận kỹ thuật chia để trị [11].

#### 4.1.2. Ý tưởng



Hình 4.1. Hình ảnh ví dụ thuật toán Merge Sort.

Dựa trên ý tưởng của **hình 4.1**, Merge sort hoạt động bằng cách chia mảng cần sắp xếp thành 2 mảng con có số lượng phần tử bằng nhau. Tiếp tục lặp lại điều này trên các mảng con cho đến khi mỗi mảng con chỉ còn một phần tử. Sau cùng gộp các mảng con lại thành một mảng đã được sắp xếp.

Ví dụ: Cho mảng 1 chiều các số nguyên gồm 7 phần tử: arr = [38, 27, 43, 3, 9, 82, 10]

### 4.1.3. Merge Sort CPU

Khi thực hiện sắp xếp một dãy số chiều theo thứ tự tăng dần hoặc giảm dần, CPU sẽ chia mảng ban đầu thành các mảng con sau đó duyệt qua từng mảng con để sắp xếp và gộp chúng lại. Do CPU phải duyệt qua toàn bộ tất cả các mảng con nên việc sắp xếp sẽ càng trở nên chậm chạp hơn khi dữ liệu lớn lên. Các bước cụ thể để thực hiện merge sort trên CPU:

Bước 1: Chia dãy số ra làm 2 phần bằng nhau:

- Dãy thứ nhất bắt đầu từ vị trí của số đầu tiên (left) đến vị trí của số ở giữa dãy số (mid).
- Dãy thứ hai bắt đầu từ sau vị trí của số ở giữa dãy số (mid) đến vị trí của số cuối cùng (right).

Bước 2: Sắp xếp 2 dãy con:

- Khi mỗi dãy con chỉ còn duy nhất một phần tử, chúng được coi là đã sắp xếp.
- Các dãy con này sẽ được gửi lên cấp độ cao hơn để hợp nhất.

Bước 3: Gộp (Merge):

- So sánh từng phần tử của hai dãy con.
- Chèn phần tử nhỏ hơn vào mảng kết quả.
- Nếu một trong hai mảng con còn phần tử, thêm toàn bộ phần còn lại vào mảng kết quả.

Mã giả:

```

1. Merge-sort(A, l, r)
2. if l < r:
3. m ← (l + r) / 2
4. Merge-sort(A, l, m)
5. Merge-sort(A, m + 1, r)
6. Merge(A, l, m, r)
7. Merge(A, l, m, r)
8. n1 ← m - l + 1
9. n2 ← r - m
10. create arrays L[0..n1] and R[0..n2]
11. for i ← 1 to n1:
12. L[i] ← A[l + i - 1]
13. for j ← i to n2:
14. R[j] ← A[m + j]
15. i = 0
16. j = 0
17. k = l
18. while i < n1 and j < n2:
19. if L[i] ≤ R[j]:
20. A[k] = L[i]
21. i = i + 1

```

```

22. else:
23. A[k] = R[j]
24. j = j + 1
25. k = k + 1
26. while i < n1:
27. A[k] = L[i]
28. i = i + 1
29. k = k + 1
30. while j < n2:
31. A[k] = R[j]
32. j = j + 1
33. k = k + 1

```

Độ phức tạp thuật toán:

Merge Sort có độ phức tạp  $O(n \log n)$  do:

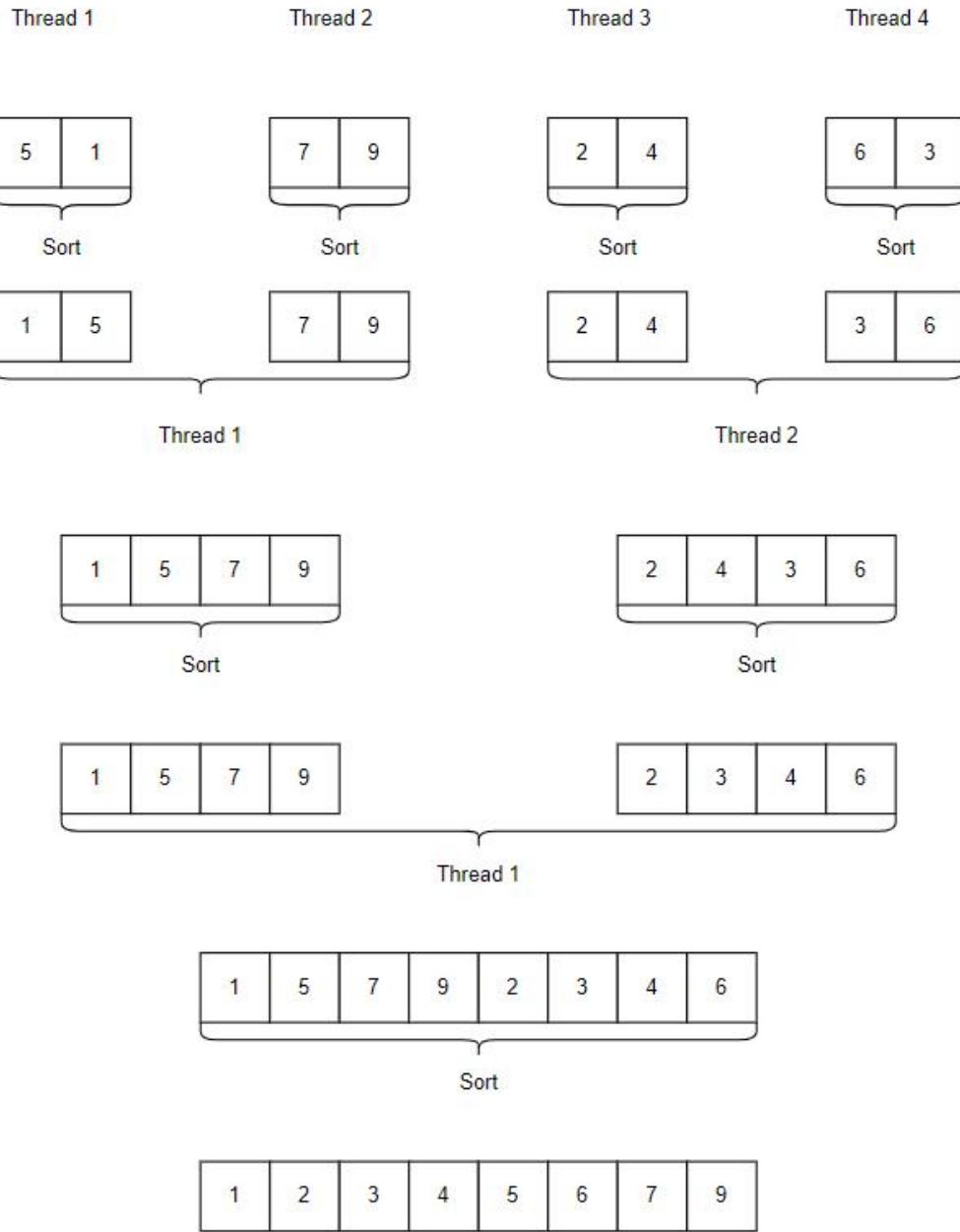
- Chia mảng: Mỗi lần chia đôi số phần tử giảm đi một nửa  $\rightarrow \log n$  lần chia.
- Hợp nhất mảng: Mỗi phần tử được so sánh và di chuyển một lần trong mỗi cấp độ  $\rightarrow O(n)$  thời gian cho mỗi cấp độ.
- Tổng quát:  $O(n \log n)$ .

Hạn chế:

- Tốn bộ nhớ: Merge Sort cần thêm bộ nhớ  $O(n)$  để lưu trữ các mảng con trong quá trình hợp nhất.
- Không tối ưu cache: Do truy cập vào các vùng nhớ không liên tục khi thực hiện quá trình chia và hợp nhất.
- Chậm hơn QuickSort trong thực tế: Mặc dù có cùng độ phức tạp trung bình, QuickSort thường nhanh hơn trên bộ nhớ đệm vì có cách truy cập dữ liệu tốt hơn.

#### 4.1.4. Merge Sort GPU

Đối với thuật toán Merge Sort khi thực hiện trên GPU, ta vẫn sẽ giữ nguyên ý tưởng ban đầu của thuật toán. Nhưng thay vì duyệt qua từng dãy con như trên CPU, ta sẽ tận dụng khả năng tính toán song song của GPU để thực hiện sắp xếp và gộp các mảng con. Khi này mỗi luồng (thread) sẽ đảm nhiệm sắp xếp và gộp một đoạn nhỏ của dãy số. Như thế công việc sắp xếp dãy số sẽ được chia nhỏ cho các luồng và thực hiện một cách song song.



**Hình 4.2.** Minh họa thuật toán Merge Sort trên GPU.

Dựa trên **hình 4.2**, các bước cụ thể để thực hiện Merge Sort trên GPU:

Bước 1: Chia dãy số thành các phần nhỏ:

- Mỗi luồng (thread) sẽ xử lý một đoạn dữ liệu nhỏ trên GPU.
- Các đoạn dữ liệu nhỏ ban đầu được coi như là đã sắp xếp (Do chỉ có một phần tử)

Bước 2: Sắp xếp song song các đoạn nhỏ

- Ta thực hiện song song nhiều cặp dữ liệu nhỏ trên GPU.
- Các luồng (threads) xử lý việc hợp nhất các đoạn nhỏ này.

Bước 3: Hợp nhất các đoạn nhỏ đã sắp xếp:

- Dữ liệu được hợp nhất theo kích thước tăng dần (2, 4, 8, 16, ...).

- GPU song song hóa quá trình này bằng cách thực hiện gộp trên mỗi luồng

Bước 4: Lặp lại quá trình hợp nhất cho đến khi toàn bộ dãy số được sắp xếp hoàn chỉnh.

Độ phức tạp thuật toán:

Phân chia song song:  $O(\log n)$

Sắp xếp các đoạn nhỏ:  $O(n)$

Hợp nhất song song:  $O(n \log n)$

Mặc dù độ phức tạp thuật toán vẫn là  $O(n \log n)$  như trên CPU, nhưng khi triển khai song song ta có thể thấy được sự cải thiện so với việc thực hiện Merge Sort tuần tự trên CPU. Tiếp theo ta sẽ tiến hành cài đặt Merge Sort trên GPU thông qua các nền tảng và thư viện như: CUDA, CuPy, RAPIDS...

### a) Merge Sort trên nền tảng CUDA

Trong phần này, chúng ta sử dụng GPU trên nền tảng CUDA để thực hiện thuật toán Merge Sort bằng cách phân chia công việc thành nhiều thread song song. Quá trình này được triển khai thông qua CUDA kernel, trong đó mỗi thread chịu trách nhiệm hợp nhất một phần của mảng, và nhiều thread được tổ chức trong các block để xử lý đồng thời.

Giả sử chúng ta có một mảng A gồm 1024 phần tử và cần sắp xếp mảng này theo thứ tự tăng dần bằng thuật toán Merge Sort.

Thuật toán Merge Sort bao gồm hai bước chính:

- Chia nhỏ mảng: Phân tách mảng thành các đoạn nhỏ hơn có kích thước 1, 2, 4, 8,...
- Hợp nhất (Merge): Ghép các đoạn nhỏ đã sắp xếp lại với nhau theo từng cặp để tạo thành một mảng đã được sắp xếp hoàn chỉnh.

Trên CPU, Merge Sort thường được thực hiện theo kiểu đệ quy (recursion), nhưng trên GPU, chúng ta sử dụng phương pháp Bottom-Up Merge Sort để khai thác tính toán song song.

Trên GPU, ta có thể tận dụng mô hình Grid - Block - Thread của CUDA để thực hiện Merge Sort một cách song song:

- Số lượng phần tử cần sắp xếp:  $N = 1024$
- Số lượng Grid: 1 (vì dữ liệu không quá lớn)
- Số lượng Blocks: 16

- Số lượng Threads trong mỗi Block: 64
- Mỗi thread sẽ đảm nhận việc hợp nhất một phần của mảng dựa trên công thức tính chỉ số:

$$\text{idx} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

Trong đó:

- **blockIdx.x**: chỉ số của Block.
- **blockDim.x**: số thread trong mỗi Block.
- **threadIdx.x**: chỉ số của thread trong một Block.

Ví dụ:

- Block 0, Thread 0 → idx = 0 \* 64 + 0 = 0
- Block 1, Thread 2 → idx = 1 \* 64 + 2 = 66
- Block 15, Thread 63 → idx = 15 \* 64 + 63 = 1023

Mỗi thread sẽ đảm nhận việc hợp nhất hai đoạn nhỏ trong mảng:

- Bước 1: Chia mảng thành các đoạn có kích thước nhỏ (ban đầu là 1 phần tử).
- Bước 2: Hợp nhát các đoạn kích thước 2, 4, 8, ...
- Bước 3: Tiếp tục hợp nhát cho đến khi mảng được sắp xếp hoàn toàn.

Mỗi kernel trong CUDA sẽ thực hiện hợp nhát hai đoạn nhỏ bằng cách sử dụng một mảng trung gian. Khi một kernel hoàn thành việc hợp nhát, kết quả được ghi lại vào mảng gốc để sử dụng ở vòng hợp nhát tiếp theo.

Mã giả Merge Sort trên GPU với nền tảng CUDA:

```

Merge Sort trên GPU CUDA
Input: Mảng A với N phần tử
Output: Mảng A đã được sắp xếp
1. Cấp phát bộ nhớ trên GPU: dev_A, dev_tempA
2. Sao chép A lên GPU: dev_A
3. Khởi tạo kích thước đoạn ban đầu: seg_size = 2
4. While seg_size < N do
5. Khởi chạy CUDA kernel với N/seg_size luồng
6. For mỗi luồng có chỉ mục tid do
7. Tính left = tid * seg_size
8. Tính mid = left + seg_size / 2
9. Tính right = min(left + seg_size - 1, N - 1)
10. If mid < right then
11. Gộp dev_A[left:mid] and dev_A[mid:right] into
dev_tempA[left:right]
12. End if
13. End for
14. Hoán đổi dev_A và dev_tempA
15. Nhân đôi seg_size

```

```

16. End while
17. Sao chép dev_A về mảng A trên CPU
18. Giải phóng bộ nhớ GPU: dev_A, dev_tempA
19. Return A

```

Cấp phát bộ nhớ trên GPU:

- Sao chép dữ liệu đầu vào từ CPU sang bộ nhớ toàn cục của GPU.
- Cấp phát bộ nhớ cho mảng đầu vào A và mảng tạm dùng để lưu trữ kết quả sau mỗi bước merge.

Gọi kernel CUDA để thực hiện Merge Sort song song:

- Thuật toán chia nhỏ dãy thành các đoạn con và tiến hành hợp nhất (merge) dần dần.
- Khởi chạy kernel với số lượng thread song song dựa trên kích thước đoạn (seg\_size).
- Ở mỗi vòng lặp, seg\_size bắt đầu từ 2 và tăng gấp đôi sau mỗi lần merge.

Mỗi thread thực hiện quá trình Merge như sau:

- Lấy chỉ số tid (thread index) của đoạn mình chịu trách nhiệm hợp nhất.
- Xác định hai đoạn con cần merge, sử dụng seg\_size để chia nhỏ dãy.
- Thực hiện thuật toán Merge:
  - So sánh phần tử giữa hai đoạn con.
  - Sắp xếp và lưu kết quả vào mảng tạm.
  - Mỗi thread chỉ xử lý một phần của đoạn con được gán.

Ghi kết quả vào bộ nhớ toàn cục của GPU:

- Kết quả sau mỗi lần hợp nhất sẽ được ghi vào mảng tạm trên GPU.
- Sau khi hoàn thành toàn bộ quá trình merge, mảng được cập nhật lại.

Chép kết quả từ GPU về CPU và xuất ra màn hình:

- Sao chép dữ liệu từ GPU về CPU sau khi quá trình Merge Sort hoàn tất.
- Xuất kết quả mảng đã sắp xếp ra màn hình.

Giải phóng bộ nhớ trên GPU:

- Giải phóng tất cả bộ nhớ đã cấp phát trên GPU (dev\_A, dev\_temp).
- Đảm bảo không rò rỉ bộ nhớ sau khi thuật toán hoàn thành.

Cài đặt nhân kernel cho Merge Sort CUDA:

```

// Device function to merge two sorted subarrays
__device__ void merge(float* data, float* temp, int left, int mid,
int right) {
 int i = left, j = mid, k = left;

```

```

 while (i < mid && j < right) {
 if (data[i] < data[j])
 temp[k++] = data[i++];
 else
 temp[k++] = data[j++];
 }

 while (i < mid)
 temp[k++] = data[i++];
 while (j < right)
 temp[k++] = data[j++];

 // Copy sorted elements back to the original array
 for (int i = left; i < right; i++)
 data[i] = temp[i];
 }

// Kernel function for parallel merge sort
__global__ void mergeSortKernel(float* data, float* temp, int size,
int width) {
 int idx = blockIdx.x * blockDim.x + threadIdx.x;
 int left = idx * width * 2;

 if (left >= size) return; // Out of bounds

 int mid = min(left + width, size);
 int right = min(left + 2 * width, size);

 if (mid < right)
 merge(data, temp, left, mid, right);
}

```

### b) Merge Sort sử dụng thư viện CuPy (Python)

CuPy là một thư viện mạnh mẽ giúp lập trình GPU dễ dàng hơn trong Python, có API tương tự NumPy nhưng hoạt động trên GPU. Việc sử dụng CuPy thay vì viết CUDA thủ công mang lại nhiều lợi ích:

#### Tích hợp dễ dàng với Python:

- Python là một ngôn ngữ phổ biến, dễ đọc, dễ triển khai.
- CuPy cung cấp API gần giống NumPy, giúp tận dụng GPU mà không cần thay đổi nhiều code.
- Cho phép tái sử dụng kernel CUDA tùy chỉnh thông qua hàm cp.RawKernel.

#### Tận dụng GPU mà không cần quản lý bộ nhớ thủ công:

- Không cần viết code CUDA C++ phức tạp.

- Quản lý bộ nhớ GPU tự động, giảm lỗi liên quan đến cấp phát và giải phóng bộ nhớ.

### **Hiệu năng cao nhưng đơn giản:**

- CuPy tối ưu các phép toán trên GPU bằng CUDA mà không cần lập trình phức tạp.
- Thích hợp cho các bài toán xử lý dữ liệu lớn như Merge Sort.

Cài đặt kernel cho Merge Sort CuPy:

```
extern "C" __global__ void merge(
 float* data, float* temp, int size, int width) {

 int idx = blockIdx.x * blockDim.x + threadIdx.x;
 int left = idx * width * 2;

 if (left >= size) return; // Nếu ngoài phạm vi thì bỏ qua

 int mid = min(left + width, size);
 int right = min(left + 2 * width, size);

 int i = left, j = mid, k = left;

 while (i < mid && j < right) {
 if (data[i] < data[j])
 temp[k++] = data[i++];
 else
 temp[k++] = data[j++];
 }

 while (i < mid)
 temp[k++] = data[i++];
 while (j < right)
 temp[k++] = data[j++];

 for (int i = left; i < right; i++)
 data[i] = temp[i];
}
```

### **c) Merge Sort sử dụng RAPIDS (cudf)**

cuDF là một thư viện xử lý dữ liệu hiệu năng cao của RAPIDS, được thiết kế để chạy trên GPU bằng CUDA. Sử dụng cuDF cho Merge Sort thay vì CuPy hoặc CUDA thuận tiện mang lại nhiều lợi ích:

### **Tối ưu hóa cho xử lý dữ liệu lớn:**

- cuDF được thiết kế để làm việc với tập dữ liệu lớn tương tự Pandas nhưng chạy trên GPU.

- Merge Sort có thể tận dụng GPU hiệu quả mà không cần lập trình CUDA phức tạp.

### Dễ dàng sử dụng với Python:

- API của cuDF rất giống Pandas, giúp việc triển khai thuật toán trên GPU trở nên đơn giản hơn.
- Không cần quản lý bộ nhớ thủ công như trong CUDA thuận tiện.

### Tích hợp với hệ sinh thái RAPIDS:

- Có thể kết hợp với cuML (học máy trên GPU) hoặc Dask-cuDF để xử lý dữ liệu phân tán.
- Phù hợp với các bài toán xử lý dữ liệu lớn trên GPU.

Cài đặt kernel Merge Sort với cudf của RAPIDS:

```
extern "C" __global__ void merge(
 float* data, float* temp, int size, int width) {

 int idx = blockIdx.x * blockDim.x + threadIdx.x;
 int left = idx * width * 2;

 if (left >= size) return;

 int mid = min(left + width, size);
 int right = min(left + 2 * width, size);

 int i = left, j = mid, k = left;

 while (i < mid && j < right) {
 if (data[i] < data[j])
 temp[k++] = data[i++];
 else
 temp[k++] = data[j++];
 }

 while (i < mid)
 temp[k++] = data[i++];
 while (j < right)
 temp[k++] = data[j++];

 for (int i = left; i < right; i++)
 data[i] = temp[i];
}
```

Sau khi hoàn thành kernel, ta sẽ tiến hành viết một hàm Python để thực hiện Merge Sort trên GPU và tích hợp hàm đó vào cudf.

```
Custom GPU Merge Sort Function
def gpu_merge_sort(self):
 if self.shape[1] != 1:
```

```

 raise ValueError("Merge sort only supports single-column
cuDF DataFrames")

 total_start = time.time()

 # 1. Memory Allocation (GPU)
 start_alloc = time.time()
 arr_gpu = cp.empty(self.shape, dtype=cp.int32)
 temp_gpu = cp.empty(self.shape, dtype=cp.int32)
 end_alloc = time.time()

 # 2. Copy Data from CPU to GPU
 start_copy_h2d = time.time()
 arr_gpu[:] = cp.array(self.values, dtype=cp.int32).flatten()
 end_copy_h2d = time.time()

 # 3. Kernel Execution (Iterative Merge Sort)
 start_kernel = time.time()
 n = arr_gpu.size
 threads_per_block = 256
 blocks_per_grid = (n + threads_per_block - 1) // threads_per_block

 width = 1
 while width < n:
 merge_sort_kernel((blocks_per_grid,), (threads_per_block,),
 (arr_gpu, temp_gpu, n, width))
 cp.cuda.Device(0).synchronize() # Ensure GPU computation
 finishes
 width *= 2
 end_kernel = time.time()

 # 4. Copy Data from GPU to CPU
 start_copy_d2h = time.time()
 sorted_result = arr_gpu.get()
 end_copy_d2h = time.time()

 # Total execution time
 total_end = time.time()

 # Convert back to cuDF DataFrame
 sorted_df = cudf.DataFrame(sorted_result, columns=self.columns)

 # Convert all times to milliseconds
 alloc_time = (end_alloc - start_alloc) * 1000
 copy_h2d_time = (end_copy_h2d - start_copy_h2d) * 1000
 kernel_time = (end_kernel - start_kernel) * 1000
 copy_d2h_time = (end_copy_d2h - start_copy_d2h) * 1000
 total_time = (total_end - total_start) * 1000

```

```

Print Timing Information
print(f"Memory Allocation Time: {alloc_time:.3f} ms")
print(f"CPU → GPU Copy Time: {copy_h2d_time:.3f} ms")
print(f"Kernel Execution Time: {kernel_time:.3f} ms")
print(f"GPU → CPU Copy Time: {copy_d2h_time:.3f} ms")
print(f"Total Execution Time: {total_time:.3f} ms")

return sorted_df

Monkey-patch cuDF DataFrame to add the custom method
cudf.DataFrame.gpu_merge_sort = gpu_merge_sort

```

Sau khi đã tích hợp ta có thể khởi tạo cudf dataframe và áp dụng merge sort trên cudf của RAPIDS.

#### **4.1.5. Kết quả thực nghiệm**

##### **a) Môi trường thực nghiệm**

Môi trường thực nghiệm được thiết lập trên nền tảng Kaggle, sử dụng GPU NVIDIA T4 với 16GB VRAM:

- Bộ xử lý (CPU): Intel Xeon
- Bộ nhớ RAM: 16GB
- Bộ xử lý đồ họa (GPU): NVIDIA T4 (16GB VRAM)
- Hệ điều hành: Linux (Ubuntu-based trên Kaggle)
- Phiên bản Python: 3.8+

##### **b) Dữ liệu**

Ta sẽ khởi tạo ngẫu nhiên các mảng số nguyên một chiều với các kích thước sau và lưu vào các file tương ứng:

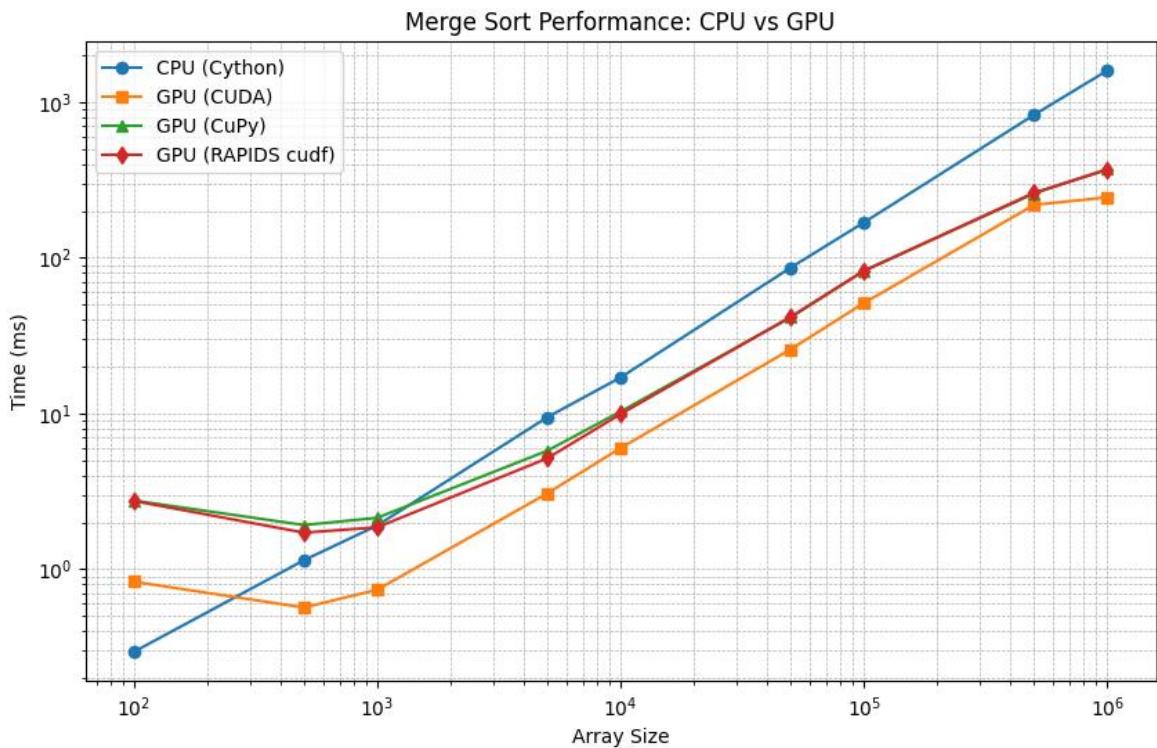
- 100 phần tử - MergeSort\_100.INP
- 500 phần tử - MergeSort\_500.INP
- 1000 phần tử - MergeSort\_1000.INP
- 5000 phần tử - MergeSort\_5000.INP
- 10000 phần tử - MergeSort\_10000.INP
- 50000 phần tử - MergeSort\_50000.INP
- 100000 phần tử - MergeSort\_100000.INP
- 500000 phần tử - MergeSort\_500000.INP
- 1000000 phần tử - MergeSort\_1000000.INP

### c) Kết quả

Bảng 4.1. Kết quả thời gian chạy của các phiên bản Merge Sort (Mili giây).

| Array size | Result  | CPU time (Cython) | GPU time (CUDA) | GPU time (CuPy) | GPU time (RAPIDS cudf) |
|------------|---------|-------------------|-----------------|-----------------|------------------------|
| 100        | Correct | <b>0.295</b>      | 0.832           | 2.763           | 2.747                  |
| 500        | Correct | 1.144             | <b>0.568</b>    | 1.923           | 1.718                  |
| 1000       | Correct | 1.911             | <b>0.739</b>    | 2.141           | 1.860                  |
| 5000       | Correct | 9.478             | <b>3.074</b>    | 5.773           | 5.164                  |
| 10000      | Correct | 17.058            | <b>6.006</b>    | 10.311          | 9.930                  |
| 50000      | Correct | 86.552            | <b>25.874</b>   | 41.462          | 41.804                 |
| 100000     | Correct | 168.884           | <b>51.304</b>   | 82.585          | 82.539                 |
| 500000     | Correct | 828.747           | <b>218.977</b>  | 258.903         | 261.700                |
| 1000000    | Correct | 1596.694          | <b>244.668</b>  | 371.162         | 368.920                |

Biểu đồ:



Hình 4.3. Biểu đồ biểu diễn thời gian chạy của các phiên bản Merge Sort.

Từ bảng 4.1, ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như hình 4.3, ta có thể rút ra được nhận xét như sau:

Hiệu suất trên CPU (Cython) vs. GPU

- Trên CPU (Cython), thời gian thực thi tăng tuyến tính theo kích thước mảng. Khi mảng đạt 1,000,000 phần tử, CPU mất 1596 mili giây, trong khi GPU nhanh hơn đáng kể (244-368 mili giây, tùy phiên bản).
- Tuy nhiên, với mảng nhỏ (< 1000 phần tử), CPU có hiệu suất tốt hơn GPU, do overhead truyền dữ liệu lên GPU làm giảm lợi thế tính toán song song.

So sánh các phiên bản GPU

- CUDA vs. CuPy vs. RAPIDS cuDF:
- CUDA có hiệu suất tốt nhất trong tất cả các kích thước mảng, đặc biệt với mảng lớn (1,000,000 phần tử), CUDA nhanh hơn CuPy và RAPIDS khoảng 1.5 lần.
- CuPy và RAPIDS có hiệu suất tương đương nhau, nhưng luôn chậm hơn CUDA do overhead từ các thư viện hỗ trợ.
- Với mảng nhỏ (100 phần tử), CUDA thậm chí chậm hơn CPU, cho thấy rằng GPU chỉ có lợi thế khi kích thước dữ liệu đủ lớn.

Xu hướng tổng quát

- Merge Sort trên CPU hoạt động hiệu quả với mảng nhỏ, nhưng bị GPU vượt xa khi kích thước mảng lớn.
- CUDA có hiệu suất tốt nhất nhờ tối ưu hóa bộ nhớ và xử lý song song hiệu quả.
- CuPy và RAPIDS tiện lợi nhưng không nhanh bằng CUDA thuận túy, phù hợp hơn khi cần dễ dàng tích hợp vào các pipeline dữ liệu lớn.
- Hiệu suất mở rộng của GPU không hoàn toàn tuyến tính do overhead truyền dữ liệu, nhưng lợi ích vẫn rất rõ ràng với dữ liệu lớn.

## 4.2. Bài toán Inverted Indexing

### 4.2.1. Mô tả bài toán

Inverted index là một cấu trúc dữ liệu cốt lõi trong các công cụ tìm kiếm quy mô lớn, mạng xã hội và các hệ thống lưu trữ khác. Mục đích chính của nó là tăng tốc độ truy vấn bằng cách ánh xạ các term (từ khóa) tới danh sách các document (tài liệu) mà chúng xuất hiện [12].

Về cơ bản, inverted index bao gồm hai thành phần chính:

- Từ điển (Dictionary): Chứa danh sách tất cả các từ khóa duy nhất xuất hiện trong tập hợp tài liệu [12].

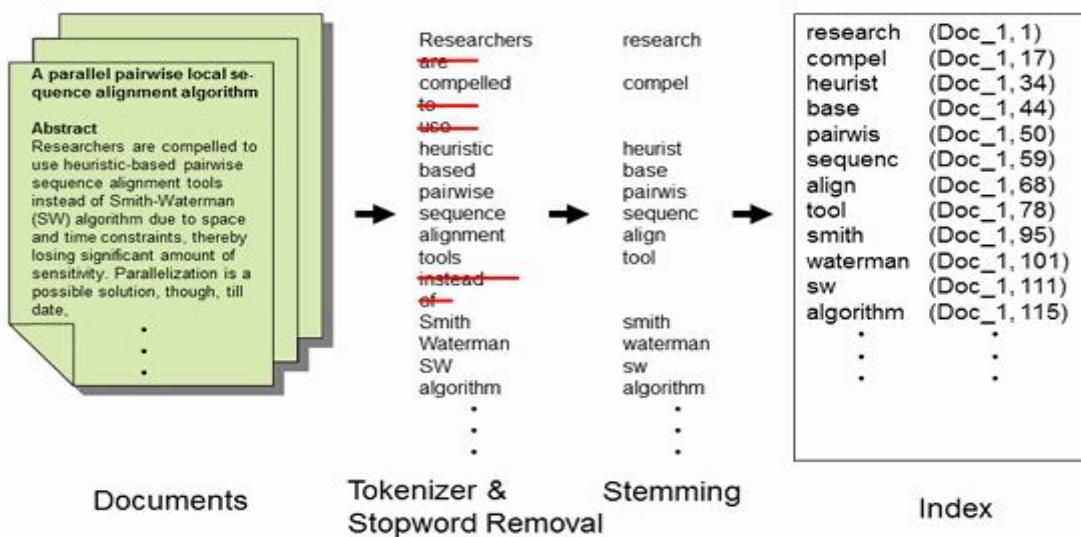
- Danh sách Posting (Posting Lists): Với mỗi từ khóa trong từ điển, có một danh sách posting tương ứng, chứa các docID (identifiers của documents) mà từ khóa đó xuất hiện. Các docID này thường được sắp xếp theo thứ tự tăng dần để tối ưu hóa việc tìm kiếm [12].

#### 4.2.2. Ý tưởng

Tiền xử lý dữ liệu trên CPU: Việc xử lý trên CPU trước khi đưa vào tận dụng đúng các điểm mạnh của cả CPU đảm bảo GPU nhận dữ liệu sạch để tập trung vào xây dựng chỉ mục. Giúp giảm lượng dữ liệu cần chuyển vào GPU và tận dụng các thư viện và công cụ sẵn có trên CPU. Việc tiền xử lý dữ liệu gồm các giai đoạn [7]:

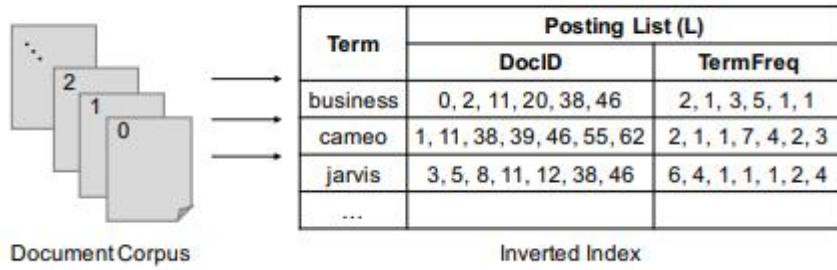
- Tokenization: Duyệt qua từng ký tự của văn bản và gom các ký tự hợp lệ (chữ, số, dấu nháy đơn) thành từng token.
- Loại bỏ ký tự thừa: Loại bỏ dấu nháy đơn ở đầu và cuối của token nếu có.
- Loại bỏ stopwords: Kiểm tra xem token có nằm trong danh sách stopwords không.
- Stemming: Áp dụng thuật toán stemming để đưa token về dạng gốc.

Xây dựng inverted index: Sau khi xử lý xong, với mỗi token hợp lệ, xây dựng một bảng ánh xạ (dictionary) lưu trữ thông tin: từ khóa → danh sách các document ID (với số lần xuất hiện của từ đó trong tài liệu). Việc này cho phép truy vấn nhanh chóng, khi người dùng nhập một từ, hệ thống sẽ trả về danh sách các tài liệu chứa từ đó kèm theo tần suất hoặc thông tin liên quan. **Hình 4.4**



**Hình 4.4.** Quy trình xây dựng chỉ mục đảo.

Sử dụng cấu trúc Posting list (**Hình 4.5**) để lưu trữ danh sách các tài liệu mà một từ khóa (term) xuất hiện. Mỗi mục trong danh sách thường chứa ID của tài liệu (docID) nơi từ khóa xuất hiện giúp hệ thống truy xuất thông tin nhanh chóng biết được từ khóa đó nằm trong tài liệu nào, posting list có thể lưu thêm thông tin như tần suất xuất hiện (frequency) của từ trong tài liệu hoặc vị trí xuất hiện (positions) để hỗ trợ các truy vấn phức tạp hơn.



**Hình 4.5.** Cấu trúc Posting List.

### Cơ chế xử lý trên CPU

- Đọc dữ liệu từ file CSV: Mở file CSV và đọc dữ liệu từ cột chứa nội dung chính sau đó lưu vào mảng danh sách chứa các nội dung của từng tài liệu
- Tokenize dữ liệu trước khi xử lý: Duyệt từng tài liệu và tách thành danh sách các token, mỗi token đã được tiền xử lý và được gán id của tài liệu gốc để đánh dấu. Việc tokenize trước giúp giảm số lần gọi hàm tokenize() để tối ưu tốc độ, danh sách các token đó được lưu trong RAM
- Xây dựng Inverted Index từ danh sách tokenized:
  - Duyệt qua từng tài liệu đã tokenize sau đó tính tổng số token của tài liệu hiện tại để theo dõi kích thước dữ liệu (1 vòng lặp xử lý 1 docid 1 cách tuần tự, có thể sử dụng OpenMP để song song hóa nhưng mức độ song song hóa vẫn thấp do số lượng lõi hạn chế)
  - Thực hiện đếm tần suất xuất hiện của từng token trong tài liệu bằng cách duyệt qua từng token 1 cách tuần tự.
  - Sau khi đã đếm tần suất thì thực hiện duyệt tuần tự qua mảng frequency[] để cập nhật word\_table (là mảng lưu trữ các từ khóa xuất hiện trong tài liệu thường là 1 dạng dictionary, kết hợp với mỗi từ khóa là định danh tài liệu mà từ khóa đó xuất hiện). Với mỗi token, CPU cập nhật word\_table với docid là tài liệu hiện tại đang xử lý và frequency của token đã đếm trước đó.

Inverted Index (word\_table) có dạng:

```
{
'word1': { doc_id1: freq1, doc_id2: freq2, ... },
'word2': { doc_id1: freq1, doc_id3: freq3, ... },
...
}
```

Mã giả thuật toán Inverted Index trên CPU:

```
1. FOR row IN csv file // Đọc file CSV
2. IF column IN row THEN append content to documents[]
3. FOR doc_id FROM 1 TO LENGTH(documents[]) // Tiền xử lý
4. tokenized_docs[doc_id] ← tokenize(documents[doc_id])
5. // xây dựng inverted index
6. FOR doc_id FROM 1 TO LENGTH(tokenized_docs[])
7. frequency ← {} // HashMap để đếm số lần xuất hiện của mỗi từ
8. FOR token IN tokenized_docs[doc_id]:
9. frequency[token] += 1 // Đếm số lần xuất hiện của token
10. FOR token, freq IN frequency
11. IF token NOT IN word_table
12. word_table[token] ← {}
13. word_table[token][doc_id] ← freq // Cập nhật Inverted Index
```

#### **4.2.3. Inverted Indexing trên GPU**

1. Đọc dữ liệu từ file CSV: Đọc file CSV từ đường dẫn truyền vào khi chạy chương trình và xác định vị trí cột chứa nội dung cần xử lý sau đó lặp qua từng dòng của file csv, tăng biến đếm docid để tạo docid, lưu nội dung của dòng đó vào mảng document[docid] (ánh xạ docId -> nội dung). Tạo ra một ánh xạ documents, trong đó mỗi docId sẽ trỏ đến một đoạn văn bản. Ví dụ mảng documents sẽ chứa:

```
{
1: "This is a GPU accelerated search.",
2: "CUDA speeds up computations."
}
```

2. Tiền xử lý văn bản: khởi tạo các biến:

- allDocsText để lưu toàn bộ từ trong tất cả văn bản thành một mảng char\* nối tiếp.
- docOffsets để lưu chỉ số bắt đầu của từng từ trong allDocsText.
- docIds để lưu ID của tài liệu chứa từ đó.

Sau đó duyệt qua từng mảng documents để gọi hàm tokenizeAndNormalize() để tiền xử lý văn bản (chuyển chữ thường, loại bỏ dấu câu, stop words, stemmer,...). Với mỗi từ cập nhật dung lượng để để biết cần bao nhiêu bộ nhớ cần cấp phát cho GPU, lưu vị trí bắt đầu của 1 từ vào docOffsets và docid tương ứng vào docIds sau đó thêm từ vào allDocsText kèm theo ký hiệu '\0' để kết thúc từ (Ký tự '\0' đóng vai trò là dấu phân

tách (delimiter) giữa các từ, giúp xác định điểm kết thúc của mỗi từ giúp giảm độ phức tạp ví dụ nếu không dùng ‘\0’ thì phải lưu và tính độ dài của từng từ vào 1 mảng).

Hình minh họa dữ liệu được dùng để đưa lên GPU:

```
allDocsText =
"this\0is\0a\0gpu\@accelerated\0search\@cuda\0speeds\0up\0computations\0"
docOffsets = [0, 5, 8, 10, 14, 25, 32, 37, 44, 47]
docIds = [1, 1, 1, 1, 1, 1, 2, 2, 2, 2]
```

Việc này rất hữu ích vì docOffsets sẽ cung cấp vị trí bắt đầu từ thứ idx (index của 1 thread trong GPU) trong allDocText, mỗi chỉ số idx ánh xạ chính xác một từ với tài liệu của nó sự đồng bộ giữa hai mảng này đảm bảo rằng vị trí của từ trong allDocsText (qua docOffsets) và ID tài liệu (qua docIds) luôn tương ứng với nhau, giúp xác định được docid của từ đó.

3. Cấp phát bộ nhớ trên GPU và sao chép dữ liệu từ CPU sang GPU: Cấp phát bộ nhớ vừa đủ cho GPU dựa vào dữ liệu của mảng documents đã được tiền xử lý trên host đảm bảo rằng bộ nhớ cấp phát là vừa đủ gồm các bộ nhớ của allDocsText (vừa đủ để chứa toàn bộ từ), docOffsets (vừa đủ để chứa tất cả offsets), docIDs (vừa đủ để chứa tất cả ID tài liệu) và biến đếm giúp cập nhật danh sách postings (luôn là sizeof(int) không cần tính). Thuật toán cũng cung cấp 20 triệu slot cho bảng băm và 800 triệu mục posting.

Do chưa xác định các từ duy nhất (unique) và số lần xuất hiện nên 2 hằng số trên là cố định.

Đánh giá hằng số cho bảng băm như:

- Vừa đủ: Để tránh xung đột băm, bảng nên lớn hơn số từ duy nhất (unique words). Với 20 triệu slot, nó có thể chứa 20 triệu từ duy nhất nếu không có xung đột. Trong thực tế, với hệ số tải (load factor) khoảng 0.7, nó phù hợp cho khoảng 14 triệu từ duy nhất.
- Thiếu: Nếu bộ sưu tập có hơn 14-20 triệu từ duy nhất (rất hiếm trong các bộ dữ liệu thông thường), sẽ có xung đột nghiêm trọng, làm hỏng chỉ mục.
- Dư: Với bộ dữ liệu nhỏ (ví dụ: 5 từ duy nhất hoặc thậm chí vài triệu từ), 20 triệu slot là quá lớn, gây lãng phí bộ nhớ.

Đánh giá hằng số cho các mục posting:

- Vừa đủ: Mỗi mục posting lưu docId và frequency cho một lần xuất hiện của từ trong tài liệu. Với 800 triệu mục, nó đủ cho 800 triệu lần xuất hiện của từ. Với PostingEntry là 8 byte (2 int), nên cần 6.4 GB bộ nhớ.
- Thiếu: Nếu tổng số lần xuất hiện vượt quá 800 triệu (ví dụ: bộ sưu tập rất lớn với từ lặp lại nhiều), sẽ tràn bộ nhớ (overflow).
- Dư: Với bộ dữ liệu nhỏ (ví dụ: 5 từ, 5 posting), 800 triệu là quá dư, bộ nhớ GPU (thường chỉ có vài chục GB).

Sau đó sao chép dữ liệu từ RAM (CPU) sang VRAM của GPU. Sau bước này dữ liệu đã sẵn sàng trên GPU.

#### 4. Thực hiện xây dựng inverted index trên GPU

Kernel xây dựng Inverted Index trên GPU bằng cách:

- Lưu trữ danh sách từ (word) và các vị trí tài liệu (docId) chứa từ đó.
- Dùng bảng băm (hash table) để ánh xạ từ khóa vào danh sách tài liệu.
- Sử dụng atomic operations để xử lý đồng thời mà không gây lỗi ghi đè.

Sau đây là cơ chế xử lý từng bước:

a. Xác định từ và tài liệu cần xử lý: Mỗi thread sẽ xử lý một từ duy nhất từ dữ liệu đầu vào dựa trên docOffsets[idx] (với idx là id của 1 thread xử lý từ đó) và xác định docid của từ đó. Nếu idx vượt quá số từ (totalWord) thì kết thúc nhằm đảm bảo thread không hợp lệ không truy cập ngoài phạm vi của docOffsets và docIds làm hỏng chỉ mục đảo ngược do dữ liệu rác được xử lý và tránh lãng phí tài nguyên. Vì CUDA yêu cầu blockDim.x và gridDim.x là các số nguyên nên phải làm tròn lên số thread nên sẽ có các thread dư thừa.

b. Sao chép từ cần xử lý ra bộ nhớ tạm: Sao chép từ ra biến currentWord, đảm bảo không truy cập sai bộ nhớ. Vì thuật toán dùng hàm băm MurmurHash xử lý theo khối 4 byte nếu con trỏ chuỗi không được căn chỉnh đúng (địa chỉ không chia hết cho 4), việc đọc 4 byte sẽ gây ra lỗi. Vì currentWord được cấp phát trên stack nên nó được căn chỉnh đúng theo mặc định của trình biên dịch (Đảm bảo alignment do CUDA tự động căn chỉnh vùng nhớ stack.). Việc copy word vào currentWord đảm bảo rằng dữ liệu đầu vào cho hashWord luôn được căn chỉnh hợp lệ.

c. Sử dụng bộ nhớ local (currentWord) để giảm truy cập vào global memory: Do bài toán Inverted index cần được cập nhật từ nhiều block khác nhau (sử dụng nhiều atomic operations và truy cập dữ liệu ngẫu nhiên vào bảng băm, điều này phù hợp hơn với

việc sử dụng global memory, nơi các cơ chế đồng bộ hóa và atomic được hỗ trợ tốt hơn cho truy cập từ nhiều block.). Shared memory chỉ khả dụng cho các thread trong cùng một block, nên không thích hợp cho dữ liệu mà tất cả các block đều cần truy cập và cập nhật đồng bộ và do Shared memory trên GPU có dung lượng rất hạn chế (thường vài chục KB cho mỗi block). Các cấu trúc dữ liệu như bảng băm wordTable và MemoryPool cần xử lý tập dữ liệu lớn, vượt quá khả năng chứa của shared memory.

d. Băm từ và xác định vị trí trong bảng băm: Băm từ để tạo giá trị hash duy nhất. Mỗi thread sẽ lấy một từ từ mảng docs và tính giá trị băm của từ đó và phân phối các từ một cách đồng đều qua bảng băm, tính chỉ số (slot) ban đầu của từ để sau này dùng để kiểm tra quay vòng. Tạo biến slot để lưu vị trí băm ban đầu lúc nãy để có thể điều chỉnh khi có xung đột (collision) và biến attempts được khởi tạo bằng 0, dùng để đếm số lần dò tìm slot mới trong trường hợp có xung đột, nhằm áp dụng cơ chế quadratic probing (gấp đôi số lần thử) sau này.

e. Xử lý xung đột (va chạm) băm và chèn/cập nhật mục từ (collision resolution):

Kernel bước vào vòng lặp while(true) để tìm một slot thích hợp nhằm chèn hoặc cập nhật dữ liệu của từ.

Mục tiêu:

- Mỗi thread cần chiếm được một slot trong bảng băm để chèn hoặc cập nhật thông tin cho từ hiện tại.
- Các thao tác phải được thực hiện an toàn trong môi trường song song, do đó sử dụng các atomic operations để đảm bảo tính nhất quán của dữ liệu.

Tạo biến currentId để cho ra các nhánh xử lý, currentId sẽ chứa giá trị ban đầu của wordTable[slot].wordId (là địa chỉ của trường “wordId” trong struct WordEntry tại vị trí “slot” trong bảng băm wordTable. Đây là biến mà thread đang cố gắng truy cập và sửa đổi) tại thời điểm atomicCAS được thực thi, bất kể phép thay thế có thành công hay không. Các nhánh của currentId:

- Nếu wordTable[slot].wordId (mã băm của từ) bằng expected (0) thì currentId nhận giá trị ban đầu, tức là expected (currentId = 0) là slot trống nó sẽ thay thế giá trị đó bằng lockState (0xFFFFFFFF) và wordTable[slot].wordId sẽ trở thành lockState (0xFFFFFFFF), giúp “khóa” slot để không thread nào khác ghi vào trong slot này lúc thread hiện tại đang xử lý.

- Nếu mã băm của từ bằng lockState thì không thay đổi currentId cũng như wordTable[slot].wordId (cả 2 vẫn lưu là lockState) và nếu slot (mã băm của từ) đã có từ khác thì currentId và wordTable[slot].wordId cũng không thay đổi (cả 2 vẫn lưu là mã băm của từ khác).

Xử lý từng trường hợp:

### 1. Xử Lý Khi Slot Trống (currentId == 0)

Đây là trường hợp đơn giản nhất dung để khởi tạo entry đầu tiên cho từ đó (Nếu currentId == 0, nghĩa là chưa có tài liệu nào chứa từ này → cần tạo mới.). Khi giá trị wordTable[slot].wordId bằng 0, nghĩa là slot này chưa được sử dụng (chưa chứa từ đang cần xử lý). Các bước thực hiện như sau:

- Sao chép currentWord vào trường word của slot, đảm bảo rằng slot này có chứa thông tin định danh của từ cần chèn
- Hệ thống cấp phát một phần bộ nhớ ban đầu để lưu danh sách các tài liệu chứa từ đó (danh sách posting ban đầu).
  - Nếu sau khi cấp phát, base + INITIAL\_POSTING\_SIZE vượt quá giới hạn MAX\_POSTINGS, tiến hành gọi atomicExch(&wordTable[slot].wordId, 0) để reset slot và in thông báo lỗi (trong chế độ debug), sau đó thoát khỏi vòng lặp.
  - Nếu cấp phát thành công, thực hiện các bước khởi tạo:
    - Cập nhật con trỏ postings của slot bằng pool.memory + base. Nghĩa là ta tính toán địa chỉ bắt đầu của vùng nhớ vừa cấp phát. Điều này cho slot biết nơi bắt đầu lưu danh sách posting.
    - Gán postingCount bằng 1, vì chỉ có một posting đầu tiên với thông tin {docId, 1}.
    - Thêm posting đầu tiên (chứa docId hiện tại và tần suất ban đầu là 1) vào vị trí đầu của danh sách.
- Sau khi khởi tạo xong, dùng atomicExch để thay thế giá trị wordId (hiện đang ở trạng thái khóa) bằng giá trị hash của từ, biểu thị rằng slot đã được khởi tạo với từ cụ thể này (giải phóng khóa).
- Sau khi đã chèn thành công từ mới, thread thoát ra khỏi vòng lặp.

### 2. Xử Lý Khi Slot Đang Bị Khóa (currentId == lockState)

Khi `wordTable[slot].wordId` có giá trị khóa (là 0xFFFFFFFF), tức slot đã chứa từ cần xử lý nhưng đang được một thread khác cập nhật (đang trong trạng thái `lockState`), các bước xử lý như sau:

Mục tiêu chính là chờ đợi và xử lý an toàn khi slot đang được thao tác

- Chờ mở khóa: Dùng vòng lặp `while` với `atomicAdd(&wordTable[slot].wordId, 0)` để đọc liên tục giá trị của `wordId` cho đến khi nó không còn là `lockState` nữa.
- Kiểm tra lại slot sau khi mở khóa:
  - Lấy lại giá trị của `wordId` và so sánh với `hash`.
  - Sử dụng `safe_strncmp` để so sánh chuỗi trong slot với `currentWord` (vì `hash` có thể bị trùng nên phải dùng `safe_strncmp` để xác minh).
  - Nếu trùng khớp, nghĩa là từ đang cần xử lý đã được khởi tạo bởi thread khác và ta chỉ cần cập nhật `postings`:
    - Duyệt qua danh sách `postings` để kiểm tra xem tài liệu `docId` đã có chưa.
    - Nếu có, tăng `frequency` bằng `atomicAdd`.
    - Nếu chưa có, dùng `atomicAdd` để tăng `postingCount` và chèn entry mới.
    - Nếu `postingCount` vượt quá `postingCapacity`, cần mở rộng bằng cách tính `newCapacity` bằng cách nhân đôi khả năng lưu trữ và cấp phát một vùng nhớ mới, sao chép các `postings` cũ sang vùng mới. Sau đó `atomicAdd` để tăng `postingCount`.
    - Sau khi cập nhật, thoát khỏi vòng lặp với `break`.
  - Nếu không khớp: Nghĩa là slot đang chứa một từ khác (từ này có trùng mã băm và không phải từ mà slot này cần cập nhật). Ta tăng biến đếm va chạm (`debug_collisions`) và dùng quadratic probing để chuyển sang slot khác. Nếu slot quay lại giá trị ban đầu, kết thúc vòng lặp vì không còn slot trống.

### 3. Xử Lý Khi Slot Đã Có Dữ Liệu (currentId khác 0 và khác `lockState`)

Trường hợp này xảy ra khi slot đã chứa một từ đang cần xử lý và không bị khóa (giá trị `wordId` là một giá trị hash hợp lệ) nghĩa là slot đó đang không có thread nào đang thay đổi nó. Trong trường hợp này, thread không cần đợi vì slot không bị khóa, thread có thể kiểm tra và xử lý ngay lập tức.

- Kiểm tra từ trùng khớp:
  - So sánh `currentId` với `hash` và sử dụng `safe_strncmp` để xác minh rằng từ trong slot trùng khớp với `currentWord` (vì `hash`, `băm` có thể bị trùng do quá

nhiều từ nên có thể gây trùng băm nên phải dùng hàm safe\_strcmp để xác minh).

- Nếu trùng khớp:
  - Duyệt qua danh sách postings:
    - Nếu tài liệu docId đã có, tăng frequency bằng atomicAdd.
    - Nếu chưa có, dùng atomicAdd để nhận chỉ số mới và chèn posting mới.
    - Nếu số lượng postings vượt quá khả năng chứa, thực hiện mở rộng bộ nhớ cho postings sau đó atomicAdd để tăng postingCount.
    - Sau khi cập nhật, thoát khỏi vòng lặp với break.
  - Nếu từ không khớp: trường hợp này là do bị trùng hash (trùng giá trị băm) nhưng không phải từ mà slot này cần xử lý. Để giải quyết xung đột băm cần tăng biến đếm collision và dùng quadratic probing để chuyển sang slot khác. Nếu slot quay lại giá trị ban đầu, kết thúc vòng lặp vì không còn slot trống.

Mặc dù trường hợp 2 (slot bị khóa) và trường hợp 3 (slot đã có từ khác) có cùng logic xử lý nhưng việc có cả hai trường hợp xử lý riêng biệt là cần thiết để đảm bảo tính nhất quán trong môi trường song song:

- Tránh race condition: Nếu nhiều thread cùng cố gắng thao tác thực hiện thay đổi trong 1 slot sẽ gây ra tượng race condition, cơ chế khóa đảm bảo chỉ một thread được thực hiện việc này giúp đảm bảo tính nhất quán cho dữ liệu.
- Cải thiện hiệu suất: Bằng cách phân biệt rõ ràng các trường hợp, code tránh được việc chờ đợi không cần thiết khi slot đã ở trạng thái ổn định.
- Sử dụng Quadratic Probing cho cả 2 trường hợp để xử lý va chạm (collision): việc tăng attempts và nhân đôi nó đảm bảo rằng ta bỏ qua các slot đã kiểm tra, nhằm giảm khả năng va chạm lặp lại. Nếu sau nhiều lần thử mà slot quay lại giá trị ban đầu, ta cho rằng bảng đã đầy hoặc không thể chèn thêm, và do đó thoát vòng lặp.

Mã giả:

```

1. function buildInvertedIndexKernel() {
2. Xác định từng thread index cho từng từ
3. IF (current_thread_index >= total_words) RETURN
4. word ← extract_word_from_documents()
5. docId ← get_document_id() // Lấy ID tài liệu chứa từ đó

```

```

6. slot ← find_initial_hash_slot(word) // Tìm vị trí slot trong
bảng băm dựa trên hàm băm
7. WHILE true
8. // Điều kiện khởi tạo
9. IF slot is empty
10. create_word_entry(word, docId) // Khởi tạo mục từ
mới với posting ban đầu cho docId
11. break
12. ELSE IF slot is IN lockState
13. WHILE (unlocked) {} // Đợi đến khi slot được mở
14. IF word already has docId entry
15. increment_word_frequency(docId) // Tăng tần
suất xuất hiện của từ trong docId
16. break
17. ELSE IF word not has docId entry
18. IF have enough capacity
19. add_new_posting_for_word(docId) // Thêm
tài liệu vào danh sách postings
20. ELSE
21. allocate more capacity // Cấp phát thêm
bộ nhớ nếu cần
22. add_new_posting_for_word(docId)
23. break
24. ELSE
25. move_to_next_slot_with_quadratic_probing(attempts++)
26. // Dùng quadratic probing để chuyển sang slot tiếp
theo tránh xung đột
27. IF (slot == originalSlot) break
28. ELSE // Slot đã có dữ liệu và không bị khóa
29. Xử lý giống như trường hợp slot đang bị khóa, nhưng
bỏ vòng lặp đợi mở khóa.
30. }

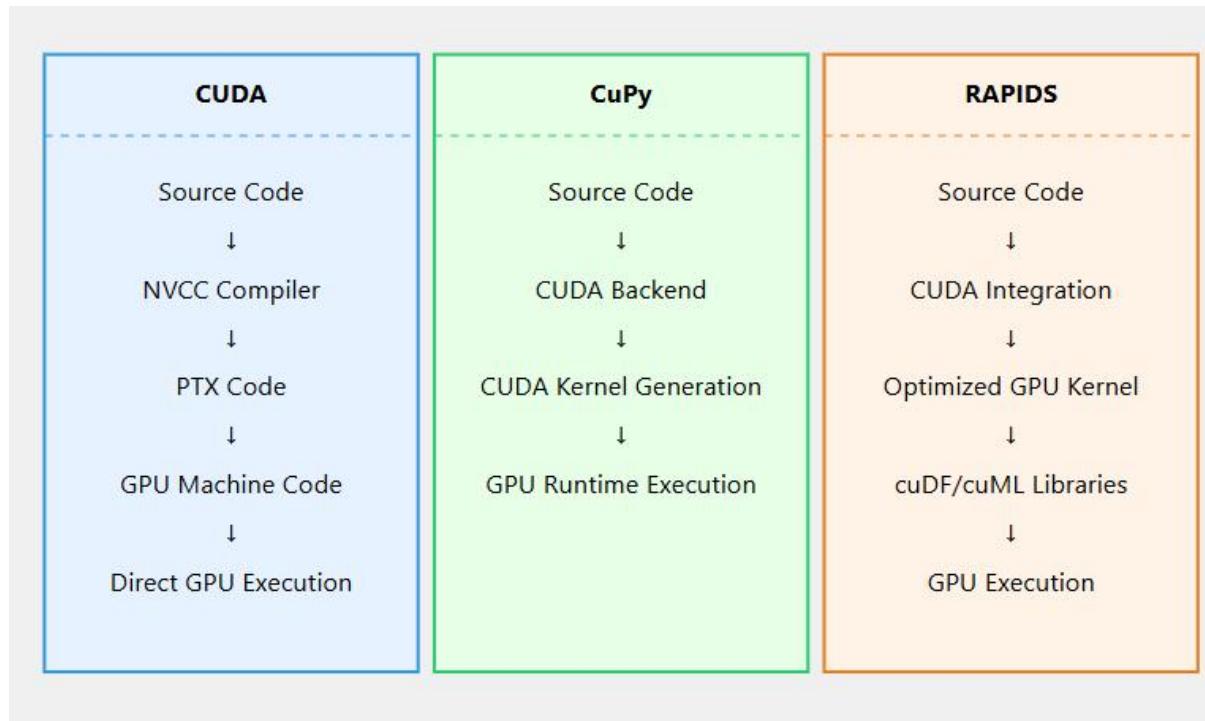
```

Phân tích ba phương pháp triển khai:

Cơ chế xử lý (logic kernel) về mặt thuật toán — từ việc trích xuất, băm, xử lý xung đột và cập nhật posting — là nhất quán giữa ba phiên bản vì cùng sử dụng kernel code (được biên dịch bằng NVCC).

Sự khác biệt chủ yếu nằm ở cách host code tích hợp kernel, chuyển dữ liệu lên GPU, và quản lý bộ nhớ.

Luồng biên dịch của 3 phiên bản:



**Hình 4.6.** Trình biên dịch của các phiên bản CUDA, CuPy, RAPIDS.

Giải thích cho **hình 4.6**:

#### a) Inverted Indexing trên nền tảng CUDA

Source Code

- Viết kernel trực tiếp bằng CUDA C++
- Quản lý bộ nhớ bằng cudaMalloc, cudaMemcpy, cudaFree
- Dùng con trỏ (PostingEntry\*) giúp truy cập dữ liệu nhanh hơn

NVCC Compiler

- Biên dịch mã nguồn thành mã PTX

PTX Code → GPU Machine Code

- Chuyển đổi thành mã máy chạy trên GPU

Direct GPU Execution

- **Ưu điểm:** Hiệu suất cao nhất do kiểm soát tài nguyên GPU tốt
- **Nhược điểm:**
  - Code phức tạp, yêu cầu kiến thức sâu về CUDA
  - Phải tự quản lý bộ nhớ, dễ gặp lỗi tràn bộ nhớ

#### b) Inverted Indexing sử dụng thư viện CuPy

Source Code

- Kernel CUDA được viết trong file .cu, load và biên dịch bằng cp.RawKernel

## CUDA Backend

- Cung cấp API cấp phát bộ nhớ: cp.zeros(), cp.array(), cp.ascontiguousarray()
- Dữ liệu được ánh xạ sang numpy.dtype thay vì dùng con trỏ

## CUDA Kernel Generation

- Thực hiện kernel thông qua CuPy, không cần tự quản lý con trỏ như CUDA C++

## GPU Runtime Execution

- Ưu điểm:
  - Dễ triển khai hơn CUDA C++, tận dụng sức mạnh CUDA mà không cần viết code quản lý bộ nhớ trực tiếp
- Nhược điểm:
  - Có overhead do Python
  - Không thể dùng con trỏ trực tiếp, cần ánh xạ dữ liệu cẩn thận

## c) Inverted Indexing sử dụng RAPIDS cudf

### Source Code

- Kernel CUDA được load bằng cp.RawKernel

### CUDA Integration

- Quản lý bộ nhớ bằng rmm.DeviceBuffer để tối ưu hóa phát

### Optimized GPU Kernel

- Dữ liệu xử lý bằng cuDF, hỗ trợ Arrow memory format để tối ưu batch processing

### cuDF/cuML Libraries

- Ưu điểm:
  - Tiết kiệm thời gian xử lý dữ liệu trên GPU nhanh hơn so với CuPy
  - Hiệu suất cao bằng hoặc nhanh hơn CuPy khi xử lý dữ liệu lớn
- Nhược điểm:
  - Không thể thay thế hoàn toàn kernel CUDA do thiếu hỗ trợ cấu trúc dữ liệu động như linked lists
  - Cần hiểu rõ cuDF, Arrow, và RMM

#### **4.2.5. Kết quả thực nghiệm**

##### **a) Môi trường thực nghiệm**

Môi trường thực nghiệm được thiết lập trên nền tảng Kaggle, sử dụng GPU NVIDIA T4 với 16GB VRAM:

- Bộ xử lý (CPU): Intel Xeon
- Bộ nhớ RAM: 16GB
- Bộ xử lý đồ họa (GPU): NVIDIA T4 (16GB VRAM)
- Hệ điều hành: Linux (Ubuntu-based trên Kaggle)
- Phiên bản Python: 3.8+

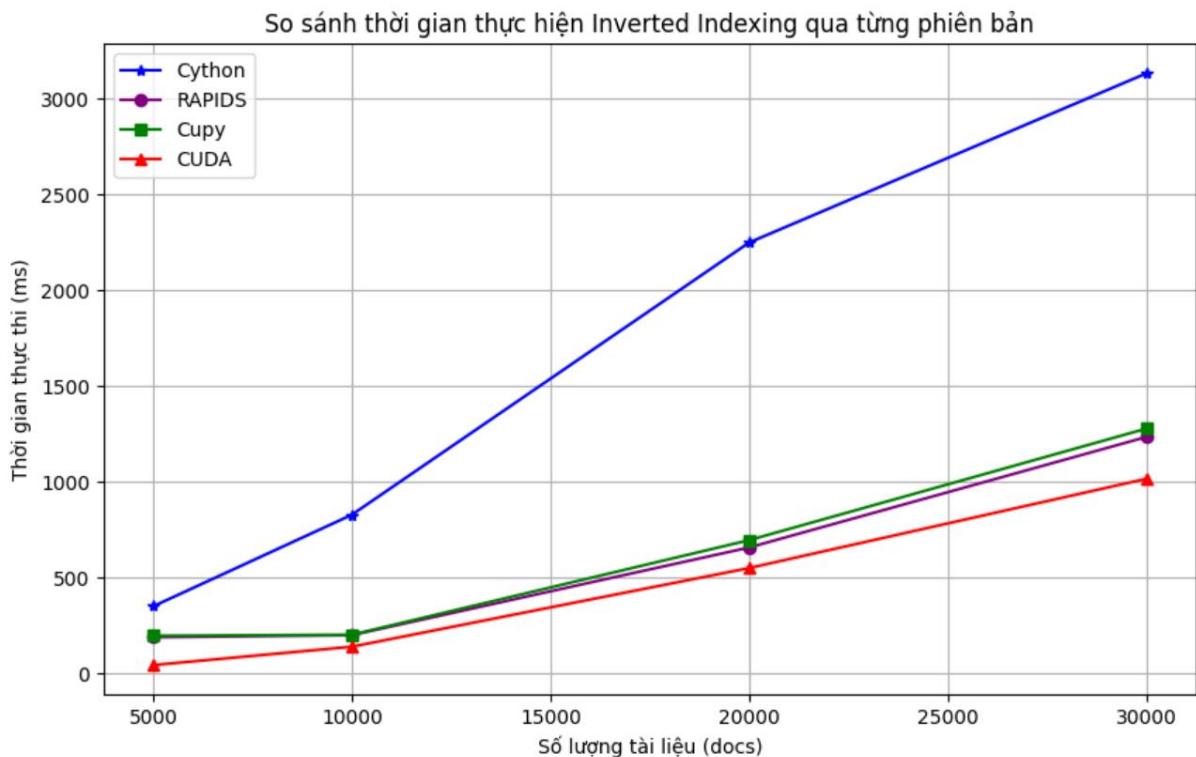
##### **b) Dữ liệu**

Sử dụng tập dữ liệu Wikipedia Movie Plots từ Kaggle, tài liệu sẽ có dạng là với mỗi hàng sẽ là 1 document và chỉ lấy cột Plot là nội dung của document đó.

##### **c) Kết quả**

*Bảng 4.2. Kết quả thời gian chạy của các phiên bản Inverted Indexing (Mili giây).*

| <b>Số lượng tập tài liệu</b> | <b>CPU time<br/>(Cython)</b> | <b>GPU time<br/>(CUDA)</b> | <b>GPU time<br/>(CuPy)</b> | <b>GPU time<br/>(RAPIDS)</b> |
|------------------------------|------------------------------|----------------------------|----------------------------|------------------------------|
| 5,000                        | 348.71                       | <b>41.81</b>               | 194.69                     | 186.99                       |
| 10,000                       | 826.43                       | <b>137.63</b>              | 200.03                     | 198.14                       |
| 20,000                       | 2,246.93                     | <b>548.08</b>              | 693.75                     | 655.43                       |
| 30,000                       | 3,129.94                     | <b>1,013.93</b>            | 1,275.84                   | 1,232.14                     |



**Hình 4.7.** Hiệu suất của các phương pháp khi xây dựng inverted index.

Từ **bảng 4.2**, ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như **hình 4.7**, ta có thể rút ra được nhận xét như sau:

- CUDA thể hiện ưu thế vượt trội nhất - điều này không ngạc nhiên vì CUDA là framework gốc phát triển bởi NVIDIA cho phép truy cập trực tiếp vào bộ xử lý đồ họa (GPU), giảm thiểu chi phí trung gian.
  - Với 5,000 tài liệu, CUDA chỉ mất **41.81 ms**, trong khi CuPy và RAPIDS lần lượt mất 194.69 ms và 186.99 ms.
  - Khi kích thước tập tài liệu tăng lên 30,000, CUDA mất **1,013.93 ms**, nhanh hơn gần 1.3 lần RAPIDS (1,232.14 ms) và 1.25 lần CuPy (1,275.84 ms).
- RAPIDS và CuPy có hiệu suất khá tương đồng nhưng kém CUDA, phản ánh fact rằng cả hai đều là các wrapper Python bậc cao hơn cho CUDA, gây thêm một lớp trung gian trong quá trình xử lý.
  - Ở 10,000 tài liệu, RAPIDS mất **198.14 ms**, gần bằng CuPy (200.03 ms).
  - Ở 20,000 tài liệu, RAPIDS nhanh hơn CuPy một chút (**655.43 ms** vs. 693.75 ms).

- Cython cho thấy hiệu năng kém nhất với mức độ tăng phi tuyến rõ rệt khi kích thước dữ liệu tăng, điều này cho thấy các giới hạn khi thực hiện song song hóa trên CPU so với GPU.
  - Với 5,000 tài liệu, CPU mất 348.71 ms, tức là gấp 8.3 lần CUDA.
  - Khi tăng lên 30,000 tài liệu, CPU mất 3,129.94 ms, tức là gấp hơn 3 lần CuPy và RAPIDS và hơn 3 lần CUDA.

# CHƯƠNG 5: CƠ CHẾ XỬ LÝ DỮ LIỆU PHÂN TÁN

## 5.1. Cơ chế xử lý dữ liệu phân tán

### 5.1.1. Khái niệm xử lý dữ liệu phân tán (*Distributed data processing*)

Xử lý dữ liệu phân tán là một phương pháp xử lý lượng lớn dữ liệu bằng cách phân phối khối lượng công việc trên nhiều máy, máy chủ hoặc nút. Thay vì có một máy chủ duy nhất xử lý tất cả dữ liệu, một hệ thống phân tán phân phối khối lượng công việc xử lý dữ liệu giữa nhiều máy chủ. Hệ thống xử lý dữ liệu phân tán cho phép xử lý dữ liệu diễn ra đồng thời, giúp tăng tốc thời gian xử lý và cải thiện hiệu suất hệ thống.

### 5.1.2. Ưu điểm của xử lý dữ liệu phân tán

- Khả năng mở rộng: Hệ thống xử lý dữ liệu phân tán có khả năng mở rộng cao.
- Khả năng phục hồi: Các hệ thống phân tán có khả năng phục hồi tốt hơn trước các lỗi hệ thống. Nếu một máy chủ bị lỗi, các máy chủ còn lại vẫn có thể hoạt động bình thường.
- Cải thiện hiệu suất: Với xử lý dữ liệu phân tán, khối lượng công việc được chia nhỏ và phân bổ trên nhiều máy chủ, dẫn đến thời gian xử lý nhanh hơn.

### 5.1.3. Một số mô hình xử lý dữ liệu phân tán

- Mô hình chia sẻ bộ nhớ (Shared Memory Model): Các bộ xử lý trong hệ thống cùng truy cập vào một bộ nhớ chung. Điều này giúp việc trao đổi dữ liệu giữa các bộ xử lý nhanh hơn, vì không cần truyền dữ liệu qua mạng. Tuy nhiên, mô hình này gặp hạn chế về khả năng mở rộng, vì số lượng bộ xử lý truy cập cùng lúc vào bộ nhớ có thể gây nghẽn.
- Mô hình chia sẻ đĩa (Shared Disk Model): Trong mô hình này, mỗi nút có bộ nhớ và bộ xử lý riêng nhưng sử dụng chung một hệ thống lưu trữ. Điều này giúp giảm trùng lặp dữ liệu giữa các nút, nhưng cũng có thể dẫn đến tình trạng nghẽn cổ chai nếu nhiều nút truy cập vào ổ đĩa cùng lúc. Mô hình này thường được sử dụng trong cơ sở dữ liệu phân tán và hệ thống lưu trữ dữ liệu lớn.
- Mô hình tập trung (Centralized Model): Một máy chủ chính (central node) chịu trách nhiệm điều phối và xử lý dữ liệu, trong khi các nút khác đóng vai trò như các trạm đầu cuối (clients) gửi yêu cầu đến máy chủ. Mô hình này phù hợp với

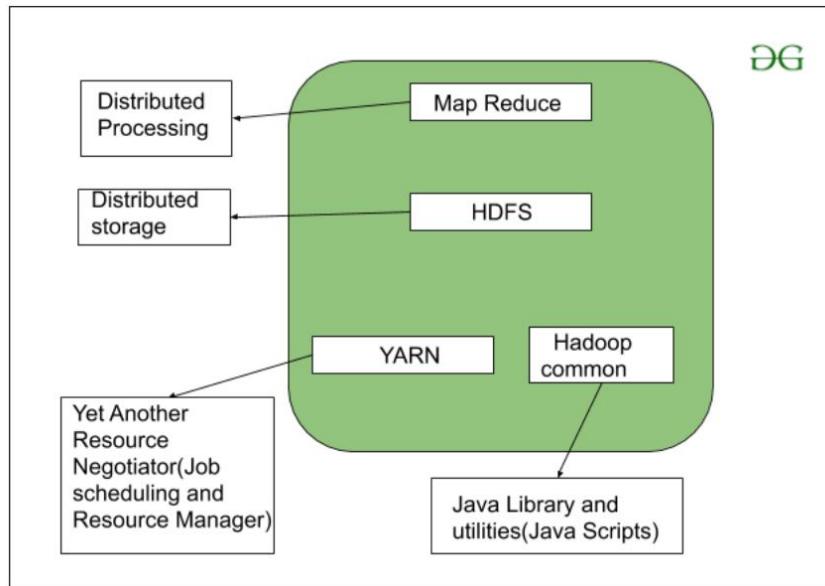
hệ thống nhỏ, nhưng khi dữ liệu và số lượng yêu cầu tăng lên, máy chủ có thể trở thành điểm nghẽn. Mô hình ngang hàng (Peer-to-Peer Model): Không có máy chủ trung tâm, các nút trong hệ thống đều ngang hàng và có thể trao đổi dữ liệu trực tiếp với nhau. Mô hình này giúp hệ thống phân tán có tính linh hoạt cao, không bị phụ thuộc vào một điểm điều phối duy nhất, và có thể mở rộng dễ dàng. Một ví dụ điển hình của mô hình này là mạng chia sẻ tệp tin (P2P) như BitTorrent.

- Mô hình pipeline song song (Pipeline Parallelism Model): Dữ liệu được xử lý theo từng giai đoạn liên tiếp, mỗi giai đoạn chạy trên một nút khác nhau. Điều này giống như một dây chuyền sản xuất, trong đó mỗi công đoạn xử lý một phần của dữ liệu trước khi chuyển sang giai đoạn tiếp theo. Mô hình này được áp dụng trong xử lý luồng dữ liệu và hệ thống ETL (Extract, Transform, Load).
- Mô hình song song dữ liệu (Data Parallelism Model): Dữ liệu được chia nhỏ thành nhiều phần và xử lý đồng thời trên nhiều nút khác nhau. Mô hình này giúp tăng tốc độ xử lý đáng kể, đặc biệt là khi làm việc với tập dữ liệu lớn. Nó được ứng dụng rộng rãi trong các hệ thống tính toán hiệu năng cao (HPC), big data và machine learning.
- Mô hình song song tác vụ (Task Parallelism Model): Các tác vụ độc lập được phân chia và thực thi đồng thời trên nhiều nút khác nhau. Mô hình này phù hợp khi mỗi tác vụ thực hiện các công việc khác nhau, thay vì xử lý chung một tập dữ liệu như mô hình song song dữ liệu. Nó thường được sử dụng trong hệ thống phân tích dữ liệu phức tạp, nơi các quy trình có thể chạy độc lập và không cần chia sẻ dữ liệu nhiều.

#### **5.1.4. Một số công nghệ xử lý dữ liệu phân tán**

##### **a) Hadoop**

Hadoop là một framework mã nguồn mở được thiết kế lưu trữ và xử lý một lượng lớn dữ liệu. Cốt lõi của khung Hadoop là Hệ thống tệp phân tán Hadoop (HDFS), là một hệ thống tệp phân tán được thiết kế để lưu trữ và quản lý lượng lớn dữ liệu trên nhiều nút trong một cụm. HDFS dựa trên Hệ thống tệp Google (GFS) và được tối ưu hóa để xử lý các tập dữ liệu lớn. Hadoop cũng bao gồm mô hình lập trình MapReduce, được sử dụng để xử lý dữ liệu trên nhiều nút trong một cụm.

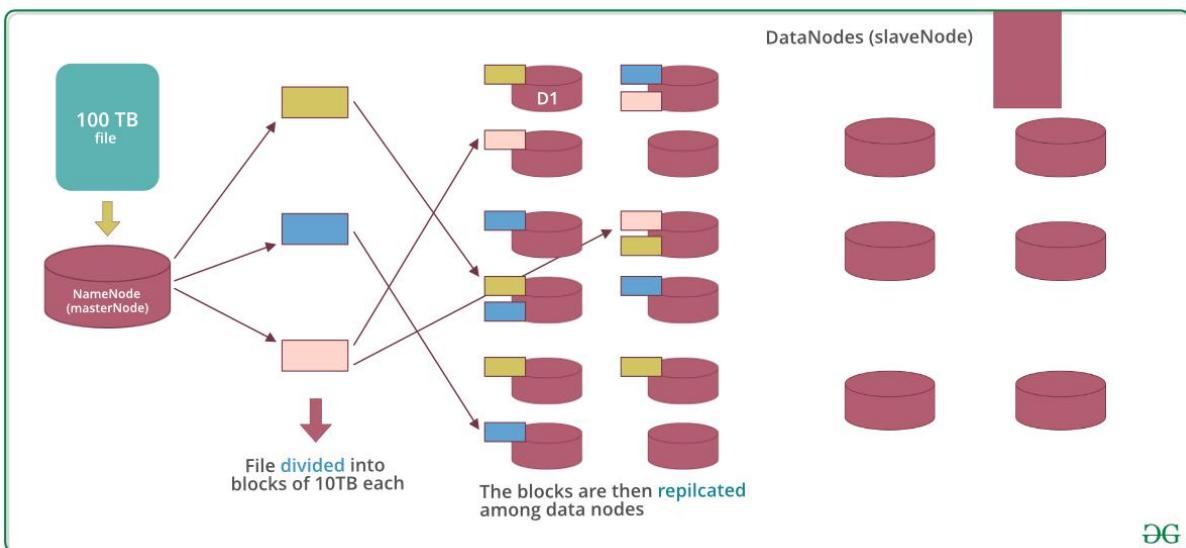


**Hình 5.1.** Kiến trúc Hadoop.

Dựa trên **hình 5.1**, Hadoop gồm các thành phần sau:

- Hệ thống tệp phân tán Hadoop (HDFS): Đây là lớp lưu trữ của Hadoop, được thiết kế để lưu trữ lượng lớn dữ liệu trên nhiều nút trong một cụm. HDFS có khả năng chịu lỗi, có nghĩa là nó có thể xử lý lỗi của các nút riêng lẻ mà không làm mất dữ liệu,
- MapReduce: MapReduce là lớp xử lý của Hadoop, được sử dụng để xử lý dữ liệu trên nhiều nút trong một cụm. MapReduce hoạt động bằng cách chia dữ liệu thành các phần nhỏ hơn, xử lý song song các phần đó và sau đó kết hợp kết quả.
- YARN: Yet Another Resource Negotiation là lớp quản lý tài nguyên của Hadoop, được sử dụng để quản lý tài nguyên trong cụm Hadoop. YARN chịu trách nhiệm phân bổ tài nguyên cho các ứng dụng chạy trên cụm và giám sát các ứng dụng đó.
- Hadoop Common: Đây là tập hợp các tiện ích và thư viện được sử dụng bởi tất cả các thành phần của Hadoop.

Luồng hoạt động của Hadoop:



ĐG

**Hình 5.2.** Luồng hoạt động Hadoop xử lý dữ liệu phân tán.

**Hình 5.2** mô tả luồng hoạt động của Hadoop trong xử lý dữ liệu phân tán. Khi một tệp lớn 100 TB được nhập vào hệ thống Hadoop, HDFS sẽ chia tệp này thành các khối nhỏ (blocks), mỗi khối có kích thước 10 TB. Việc chia nhỏ này giúp tăng hiệu suất xử lý song song và giảm tải trên mỗi máy. NameNode (Master Node) chịu trách nhiệm quản lý metadata, bao gồm vị trí và trạng thái của các khối dữ liệu, nhưng không lưu trữ dữ liệu thực tế. Trong khi đó, các DataNodes (Slave Nodes) đảm nhận việc lưu trữ từng khối dữ liệu, mỗi DataNode chứa một phần của tệp ban đầu. Để đảm bảo tính sẵn sàng và an toàn dữ liệu, HDFS tự động sao chép các khối dữ liệu giữa nhiều DataNodes, thường theo mặc định là ba bản sao. Cơ chế nhân bản này giúp hệ thống có khả năng chịu lỗi cao, ngăn ngừa mất dữ liệu ngay cả khi một DataNode gặp sự cố. Hadoop được sử dụng rộng rãi trong xử lý dữ liệu lớn và phổ biến vì khả năng mở rộng, khả năng chịu lỗi và hiệu quả chi phí. Hadoop có thể được chạy trên phần cứng hàng hóa, điều này làm cho nó tiết kiệm chi phí hơn so với các hệ thống xử lý dữ liệu truyền thống.

### b) Spark

Apache Spark là một hệ thống điện toán phân tán mã nguồn mở được thiết kế để xử lý dữ liệu lớn. Spark được phát triển để đáp ứng những hạn chế của mô hình lập trình MapReduce được sử dụng trong Hadoop. Mặc dù MapReduce có hiệu quả đối với xử lý hàng loạt, nhưng nó không phù hợp để xử lý thời gian thực hoặc xử lý lặp đi lặp lại. Mặt khác, Spark được thiết kế để xử lý các loại khối lượng công việc này.

Cốt lõi của Apache Spark là trùu tượng hóa Bộ dữ liệu phân tán có khả năng phục hồi (RDD), được sử dụng để lưu trữ và xử lý dữ liệu trên nhiều nút trong một cụm. RDD là bất biến, có nghĩa là chúng không thể thay đổi sau khi chúng được tạo. Điều này cho phép Spark thực hiện song song các thao tác trên RDD mà không lo xung đột giữa nhiều luồng.

Spark cũng bao gồm một số thư viện cho máy học (MLlib), xử lý đồ thị (GraphX) và xử lý luồng (Spark Streaming). Các thư viện này cung cấp các API cấp cao cho các tác vụ xử lý dữ liệu phổ biến, giúp các nhà phát triển viết các ứng dụng xử lý dữ liệu phân tán dễ dàng hơn.

Spark có một số lợi thế so với Hadoop, bao gồm:

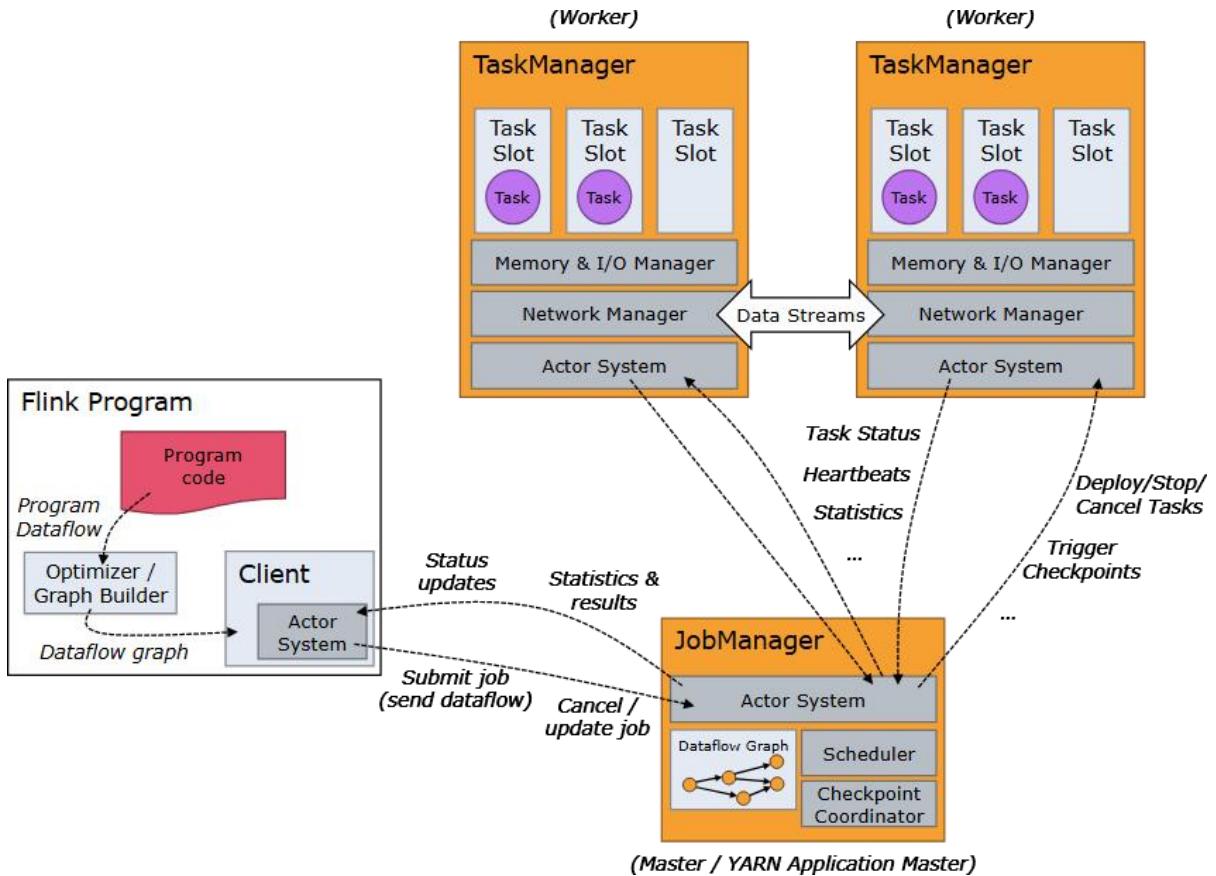
- Tốc độ: Spark có thể xử lý dữ liệu nhanh hơn tới 100 lần so với Hadoop, nhờ khả năng xử lý trong bộ nhớ của nó.
- Xử lý thời gian thực: Spark được thiết kế để xử lý khối lượng công việc xử lý theo thời gian thực, điều này rất phù hợp với các ứng dụng yêu cầu xử lý độ trễ thấp.
- Xử lý lặp lại: Spark rất phù hợp với khối lượng công việc machine learning, yêu cầu xử lý lặp đi lặp lại.
- Dễ sử dụng: Spark cung cấp các API cấp cao cho các tác vụ xử lý dữ liệu phổ biến, giúp các nhà phát triển viết các ứng dụng xử lý dữ liệu phân tán dễ dàng hơn.

### c) Flink

Apache Flink là một hệ thống điện toán phân tán mã nguồn mở được thiết kế để xử lý dữ liệu trực tuyến và xử lý hàng loạt. Flink ban đầu được phát triển bởi Đại học Kỹ thuật Berlin vào năm 2009 và sau đó được mã nguồn mở vào năm 2014. Flink được thiết kế để xử lý khối lượng công việc xử lý dữ liệu theo thời gian thực, yêu cầu xử lý độ trễ thấp.

Cốt lõi của Apache Flink là API DataStream, được sử dụng để xử lý dữ liệu phát trực tuyến trong thời gian thực. Flink cũng bao gồm API DataSet, được sử dụng để xử lý dữ liệu hàng loạt. Cả hai API đều dựa trên một API thống nhất được gọi là Flink API, cho phép các nhà phát triển viết mã có thể được sử dụng cho cả phát trực tuyến và xử lý hàng loạt.

Flink sử dụng công cụ luồng dữ liệu phân tán để thực hiện các tác vụ xử lý dữ liệu song song trên nhiều nút trong một cụm. Flink cũng tự động xử lý chuyển đổi dự phòng và khôi phục, điều này làm cho nó rất phù hợp với các ứng dụng quan trọng. Flink cũng bao gồm một số thư viện để học máy (FlinkML), xử lý đồ thị (Gelly) và SQL (Flink SQL). Các thư viện này cung cấp các API cấp cao cho các tác vụ xử lý dữ liệu phổ biến, giúp các nhà phát triển viết các ứng dụng xử lý dữ liệu phân tán dễ dàng hơn.



Hình 5.3. Kiến trúc Apache Flink.

Dựa trên **hình 5.3**, kiến trúc của chương trình Apache Flink bao gồm:

- Flink Program: Chứa mã chương trình của người dùng, được chuyển thành đồ thị luồng dữ liệu (Dataflow graph).
- Client: Đóng vai trò xây dựng và gửi đồ thị luồng dữ liệu đến JobManager.
- JobManager: Chịu trách nhiệm lên lịch (scheduler), điều phối checkpoint, và quản lý thực thi các tác vụ.
- TaskManager: Xử lý các nhiệm vụ được giao, sử dụng các task slot để thực thi luồng dữ liệu.
- Actor System: Dùng để giao tiếp giữa các thành phần.
- Data Streams: Đại diện cho luồng dữ liệu truyền giữa các TaskManager.

Flink có một số lợi thế so với các khung xử lý dữ liệu phân tán khác, bao gồm:

- Xử lý độ trễ thấp: Flink được thiết kế để xử lý khói lượng công việc xử lý dữ liệu theo thời gian thực, yêu cầu xử lý độ trễ thấp.
- Khả năng chịu lỗi: Flink có thể tự động xử lý chuyển đổi dự phòng và khôi phục, điều này làm cho nó rất phù hợp cho các ứng dụng quan trọng.
- API linh hoạt: Flink cung cấp các API linh hoạt cho cả xử lý dữ liệu hàng loạt và phát trực tuyến, giúp các nhà phát triển viết các ứng dụng xử lý dữ liệu phân tán dễ dàng hơn.
- Các tính năng nâng cao: Flink bao gồm một số tính năng nâng cao, chẳng hạn như hỗ trợ xử lý sự kiện phức tạp, xử lý gia tăng và xử lý trạng thái.

Trong những năm gần đây, Flink đã trở nên phổ biến như một khung xử lý dữ liệu phân tán nhanh và linh hoạt và hiện được sử dụng rộng rãi trong môi trường sản xuất.

## 5.2. Cơ chế xử lý dữ liệu song song phân tán

### 5.2.1. Tổng quan về cơ chế xử lý dữ liệu song song phân tán

Trong bối cảnh xử lý dữ liệu lớn, các hệ thống hiện đại ngày nay không còn dựa vào một mô hình đơn lẻ mà thường kết hợp các cơ chế xử lý dữ liệu song song và phân tán để đạt được hiệu suất tối ưu. Mặt khác, mặc dù các chương trước đã đi sâu vào cơ chế xử lý dữ liệu song song và phân tán riêng lẻ, thì ở phần này, chúng ta sẽ tập trung vào việc dẫn dắt, khai thác và tích hợp hai cơ chế này, nhằm tạo ra một hệ thống xử lý toàn diện và linh hoạt.

Các cơ chế này có thể được coi là những “mảnh ghép” bổ trợ cho nhau. Trong một hệ thống tích hợp, xử lý song song không chỉ đảm nhiệm nhiệm vụ khai thác tối đa sức mạnh của phần cứng nội bộ – như khai thác đa lõi, đa tiến trình hay đa luồng – mà còn đóng vai trò quan trọng khi dữ liệu đã được phân chia và phân phối qua các nút của một hệ thống phân tán. Điều này giúp các hệ thống không chỉ rút ngắn thời gian xử lý mà còn đảm bảo khả năng mở rộng và độ tin cậy cao khi đổi mới với khói lượng dữ liệu khổng lồ và các trường hợp lỗi ngoài ý muốn [14].

Chúng ta có thể hình dung quá trình xử lý như một chuỗi liên tục trong đó dữ liệu được chia nhỏ trên các nút (xử lý phân tán), sau đó, mỗi nút lại thực hiện xử lý nội bộ một cách song song để giải quyết các tác vụ nhỏ hơn. Quá trình này không đơn thuần

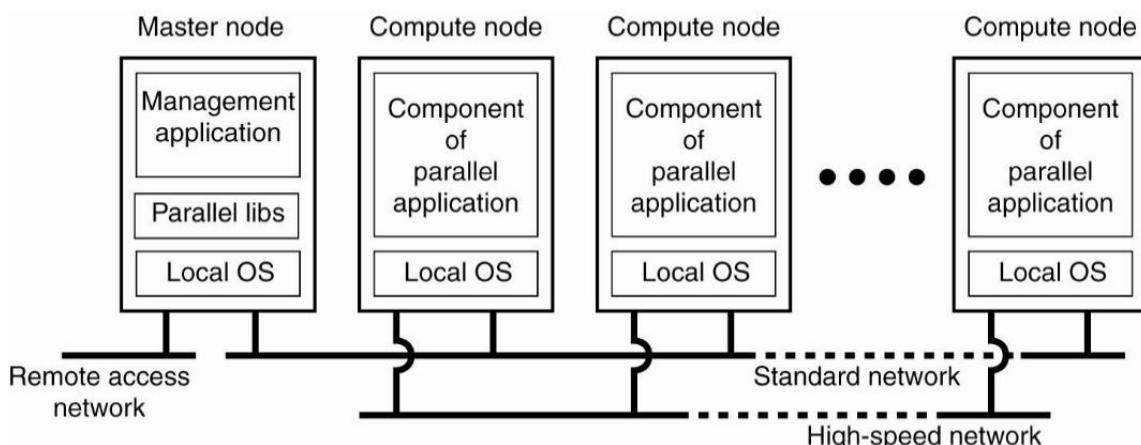
là hai bước riêng biệt mà là một luồng thống nhất, nơi mà việc xử lý song song tại từng nút góp phần giảm thiểu độ trễ của toàn hệ thống, đồng thời, khả năng phân tán giúp cân bằng tải và khắc phục sự cố.

Sự tích hợp này không chỉ giúp giảm thiểu thời gian tính toán mà còn tăng cường khả năng mở rộng của hệ thống. Khi nhu cầu xử lý tăng cao, việc mở rộng quy mô phân tán dữ liệu trên nhiều nút đồng nghĩa với việc mở rộng cả khả năng xử lý song song tại từng nút. Điều này tạo ra một vòng phản hồi tích cực, nơi mà các giải pháp về mặt phần mềm và phần cứng hỗ trợ lẫn nhau, góp phần hiện thực hóa mục tiêu tối ưu hóa hiệu suất và khả năng chịu lỗi của hệ thống.

### **5.2.2. Minh họa cơ chế xử lý dữ liệu song song phân tán**

Hai cơ chế song song và phân tán không tồn tại để cạnh tranh, mà là hai mặt của một chiến lược tổng thể:

- Phân tán → Giải quyết vấn đề không gian (scalability, fault tolerance).
- Song song → Giải quyết vấn đề thời gian (throughput, latency).



**Hình 5.4. Minh họa cơ chế xử lý dữ liệu song song phân tán.**

**Hình 5.4** minh họa kiến trúc cụm máy tính phân tán (cluster) cho xử lý song song và phân tán, thường được sử dụng trong hệ thống tính toán hiệu năng cao và xử lý dữ liệu lớn.

**Các thành phần chính trong kiến trúc:**

#### **a) Master Node (Nút chủ)**

- Chứa phần mềm quản lý (Management application) điều phối toàn bộ cụm
- Tích hợp các thư viện hỗ trợ song song (Parallel libs) để phối hợp các tác vụ
- Chạy trên hệ điều hành cục bộ (Local OS)
- Đóng vai trò điều khiển, phân phối tác vụ và giám sát các nút tính toán

### b) Compute Nodes (Các nút tính toán)

- Nhiều nút tính toán (có thể mở rộng) chạy các thành phần của ứng dụng song song
- Mỗi nút có cấu trúc giống nhau với các thành phần ứng dụng song song và hệ điều hành cục bộ
- Thực hiện các tác vụ được phân công từ nút chủ

### c) Hạ tầng mạng đa tầng

- Remote access network: Mạng truy cập từ xa, cho phép người dùng và quản trị viên kết nối đến cụm
- Standard network: Mạng tiêu chuẩn cho giao tiếp thông thường giữa các nút
- High-speed network: Mạng tốc độ cao dành riêng cho truyền dữ liệu khối lượng lớn và giao tiếp giữa các tiến trình song song

**Kiến trúc này thể hiện rõ cả hai cơ chế xử lý:**

- **Xử lý phân tán:** Thông qua việc phân tán các thành phần ứng dụng trên nhiều nút tính toán vật lý khác nhau, kết nối qua mạng
- **Xử lý song song:** Các thành phần của cùng một ứng dụng song song chạy đồng thời trên các nút tính toán

## 5.3. Tổng quan về Apache Spark

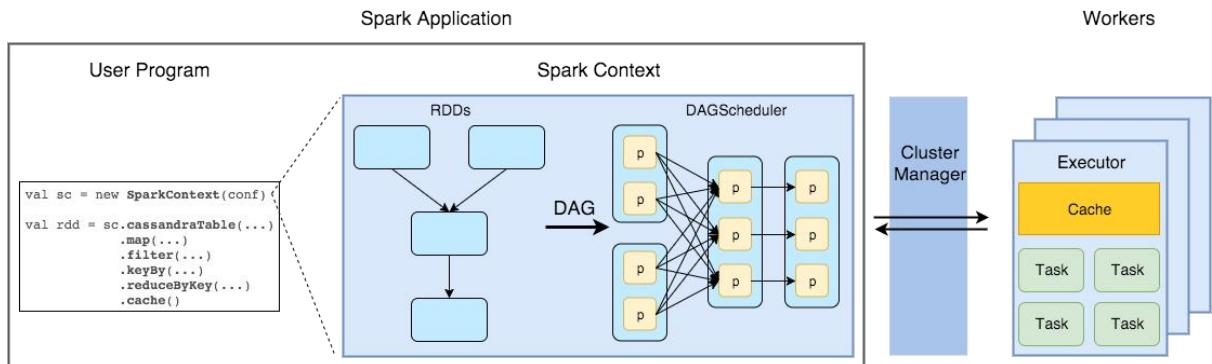
### 5.3.1. Giới thiệu

Apache Spark là một framework xử lý dữ liệu phân tán mã nguồn mở được phát triển vào năm 2009 tại AMPLab thuộc Đại học California, Berkeley, và sau đó được đóng góp cho Apache Software Foundation vào năm 2013 [15]. Spark được thiết kế nhằm khắc phục những hạn chế của mô hình MapReduce truyền thống, đặc biệt là trong việc xử lý các thuật toán lặp và phân tích dữ liệu tương tác. Hiện nay, Spark đã trở thành một trong những công nghệ xử lý dữ liệu lớn (big data) phổ biến nhất trong cộng đồng nghiên cứu và công nghiệp.

Nghiên cứu này tập trung phân tích cơ chế xử lý song song và phân tán của Apache Spark, một yếu tố quan trọng giúp nó đạt được hiệu suất vượt trội so với các framework trước đây. Cụ thể, chúng tôi sẽ nghiên cứu kiến trúc cốt lõi, mô hình lập trình, cơ chế quản lý bộ nhớ, và các chiến lược tối ưu hóa của Spark.

### 5.3.2. Kiến trúc của Apache Spark

#### a) Tổng quan kiến trúc



Hình 5.5. Kiến trúc tổng quan của Apache Spark.

Dựa trên **hình 5.5**, Apache Spark được xây dựng trên mô hình master-worker, bao gồm các thành phần chính sau [16]:

- Driver Program: Chứa hàm main() của ứng dụng và định nghĩa SparkContext, điều phối quá trình thực thi của ứng dụng Spark.
- Cluster Manager: Quản lý tài nguyên trong cụm máy tính (ví dụ: YARN, Mesos, Kubernetes, hoặc Spark Standalone).
- Worker Node: Máy tính trong cụm có khả năng thực thi các tác vụ.
- Executor: Tiến trình được khởi chạy trên mỗi worker node, chịu trách nhiệm thực thi các tác vụ và lưu trữ dữ liệu.

Khi một ứng dụng Spark thực thi, Driver Program tạo ra một DAG đại diện cho các phép biến đổi dữ liệu. DAG này được phân tích và chuyển đổi thành kế hoạch thực thi vật lý, sau đó được chia thành các giai đoạn (stages) và nhiệm vụ (tasks). Các task được phân phối đến các executor trên các worker node để xử lý song song.

Kiến trúc này cho phép Spark phân tán việc xử lý dữ liệu trên nhiều máy tính, tận dụng sức mạnh tính toán song song để xử lý khối lượng dữ liệu lớn một cách hiệu quả.

#### b) Mô hình xử lý song song và phân tán

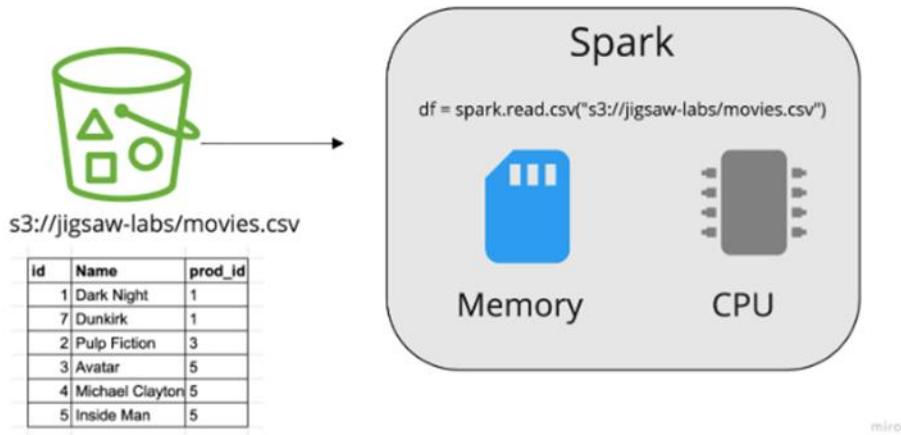
Cơ chế xử lý song song và phân tán của Spark dựa trên hai khái niệm cốt lõi:

- Resilient Distributed Dataset (RDD): Cấu trúc dữ liệu phân tán cơ bản, bất biến, có khả năng phục hồi và được phân vùng.
- Directed Acyclic Graph (DAG): Đồ thị có hướng không chu trình biểu diễn luồng xử lý dữ liệu.

### 5.3.3. Resilient Distributed Dataset (RDD)

#### a) Spark in memory

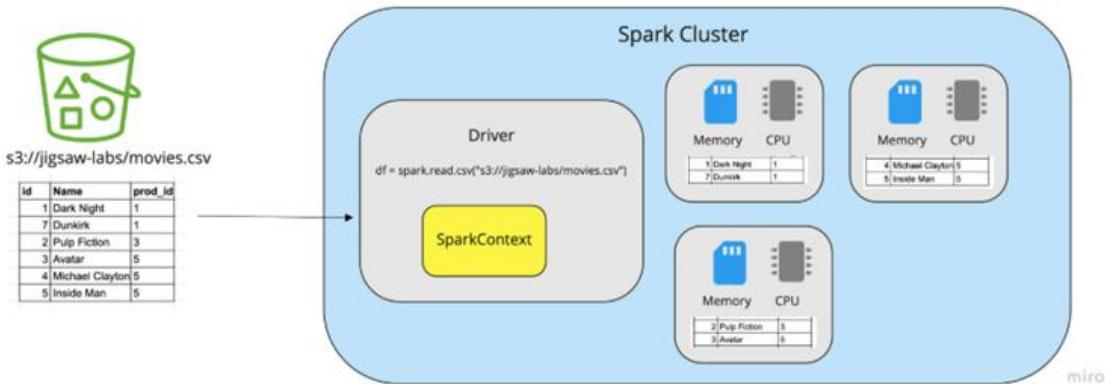
Setup cấu hình cho spark driver và spark executor



**Hình 5.6.** Minh họa quy trình đọc dữ liệu vào Apache Spark.

Dựa trên **hình 5.6**, Spark đọc tệp CSV từ S3, tạo DataFrame và lưu trữ nó trong bộ nhớ để tăng tốc độ xử lý trên CPU. Khi làm việc với Spark, quy trình xử lý dữ liệu thường diễn ra theo các bước sau::

- Bước 1: Kết nối đến nguồn dữ liệu bên ngoài (như cơ sở dữ liệu, Amazon S3, HDFS, hoặc file CSV).
- Bước 2: Tạo bảng tạm trong Spark để lưu trữ dữ liệu.
- Bước 3: Đọc và load dữ liệu vào Spark dưới dạng DataFrame hoặc RDD.
- Bước 4: Thực hiện các truy vấn và tính toán trực tiếp trên dữ liệu trong bộ nhớ.



**Hình 5.7.** Ví dụ minh họa kiến trúc xử lý dữ liệu trong Apache Spark Cluster.

**Hình 5.7** là quá trình đọc dữ liệu từ S3 vào Spark, phân chia dữ liệu thành các partition (3 partitions), phân phối task đến các Worker Nodes để xử lý song song và

tập hợp kết quả về Driver. Cơ chế này cho phép Spark xử lý dữ liệu lớn một cách hiệu quả và nhanh chóng.

- Spark lưu trữ dữ liệu chủ yếu trong bộ nhớ (RAM)
  - Cơ sở dữ liệu truyền thống (MySQL, PostgreSQL, etc.): Lưu trữ dữ liệu chủ yếu trên ổ đĩa và chỉ tải dữ liệu lên bộ nhớ khi cần truy vấn.
  - Apache Spark: Tải dữ liệu lên bộ nhớ RAM ngay từ đầu để tăng tốc xử lý.

Ví dụ:

- Khi bạn đọc dữ liệu từ S3 (Amazon Simple Storage Service), Spark sẽ lấy dữ liệu đó và lưu tạm trong bộ nhớ RAM thay vì ổ cứng.
- Khi bạn shutdown cluster, dữ liệu này sẽ bị xóa khỏi bộ nhớ, nhưng vẫn còn lưu trong S3 để có thể đọc lại sau này.
- Cách Spark hoạt động theo mô hình Cluster
  - Spark không chạy trên một máy duy nhất, mà hoạt động trên một cụm máy chủ (Cluster).
  - Spark ưu tiên lưu trữ dữ liệu trong RAM thay vì ổ cứng để truy vấn nhanh hơn.
  - Khi tắt Spark Cluster, dữ liệu trong RAM sẽ bị xóa, nhưng dữ liệu gốc vẫn còn lưu trên S3 hoặc nguồn dữ liệu ban đầu.
  - Spark chạy trên nhiều máy chủ để xử lý dữ liệu song song, giúp xử lý dữ liệu lớn hiệu quả.

## b) Parallel Processing with Executors

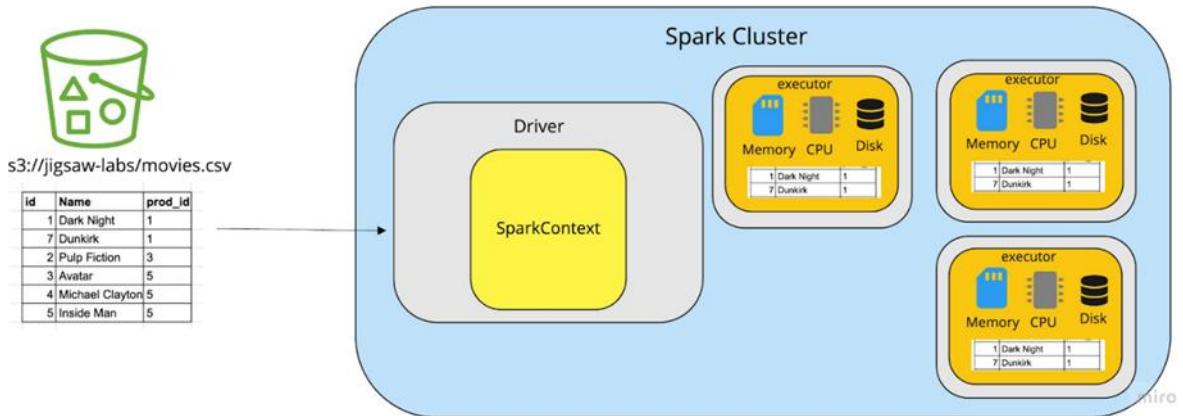
Apache Spark cho phép lưu trữ và xử lý dữ liệu trong bộ nhớ trên một cụm phân tán. Cụm này được tổ chức gồm:

- Driver node: Điểm truy cập chính vào cụm Spark.
- Worker nodes: Các nút làm việc nơi các nhiệm vụ được thực thi.
- Executors: Phần mềm chạy trên các worker nodes để thực hiện xử lý dữ liệu song song.

Trong tài liệu này, chúng ta sẽ đi sâu vào cách thức hoạt động của executors và cách chúng giúp xử lý dữ liệu song song.

**Hiểu về Executors và Parallelization:**

**CPU Cores và Partitioning**



**Hình 5.8. CPU Cores và Partitioning.**

Dựa vào **hình 5.8**, ta có thể thấy được dữ liệu được phân chia và xử lý song song trong Spark Cluster theo nguyên tắc:

- Mỗi worker node chạy một executor.
- Số lượng **cores** trên executor quyết định số partitions dữ liệu có thể xử lý đồng thời.
- Khi đọc dữ liệu vào Spark, nó được chia thành các **partitions** trên các executor.
- Số **partitions** tối đa được xác định bởi số **cores** của executors.
- Nhiều cores giúp tăng khả năng xử lý song song.

Ví dụ:

```
movies = ['Shazam!', 'Captain Marvel', 'Escape Room', 'How to Train
Your Dragon: The Hidden World']
movies_rdd = sc.parallelize(movies)
print(movies_rdd.getNumPartitions())
Kết quả: 2 (tương ứng với số cores đã chỉ định trong local[2])
```

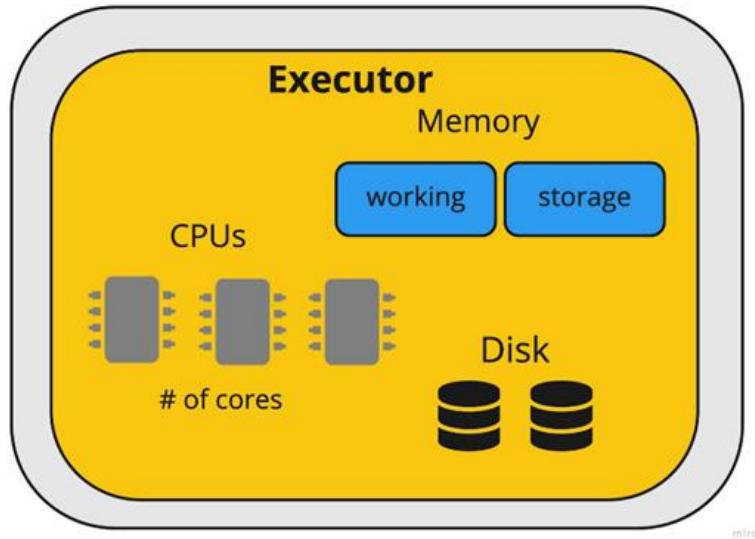
### Searching in Parallel

Khi thực hiện tìm kiếm dữ liệu, Spark sẽ phân chia công việc giữa các partitions để tìm kiếm đồng thời:

```
movies_rdd.filter(lambda movie: movie == 'Captain Marvel').collect()
```

Kết quả: ['Captain Marvel']

### Memory Considerations



*Hình 5.9. Memory Considerations.*

Theo **hình 5.9**, Memory trên executor được chia thành 2 phần chính:

- Working Memory: Dùng để xử lý dữ liệu.
- Storage Memory: Dùng để lưu trữ dữ liệu trong bộ nhớ.
- Nếu Storage Memory chiếm quá nhiều, sẽ không còn đủ Working Memory để xử lý dữ liệu.

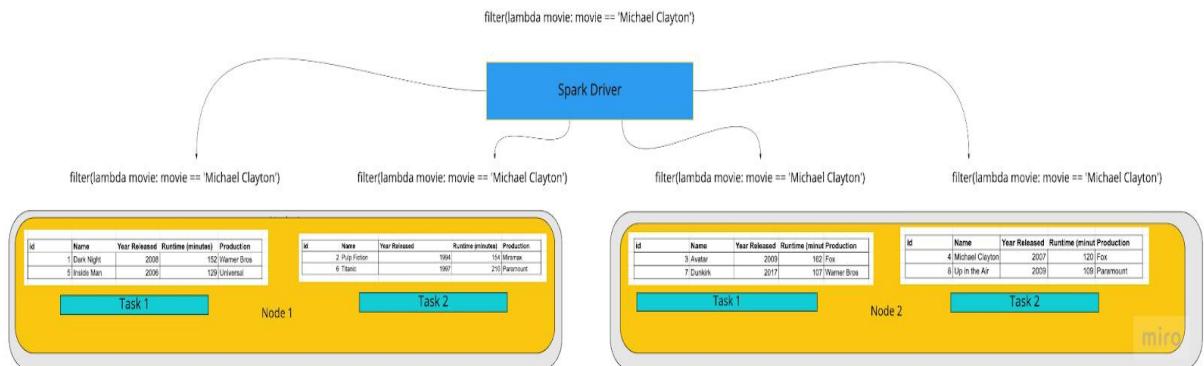
Điều này là do mặc dù chúng tôi chỉ có thể có một CPU thực hiện công việc trong một người thực thi, nếu có nhiều lõi, thì nhiều quy trình có thể được thực hiện đồng thời. Và vì dữ liệu của chúng tôi nằm trong bộ nhớ, chúng tôi có thể phân vùng dữ liệu của mình, để mỗi lõi có thể xử lý một phân chia dữ liệu. Vì vậy, khi chúng tôi nhìn vào một người thực thi, một điều chính cần xem xét là số lượng lõi có sẵn cho chúng tôi để phân vùng dữ liệu của chúng tôi.

Vì vậy, chúng tôi chỉ thấy rằng một phần cứng có thể xác định việc xử lý dữ liệu của chúng tôi là số lượng lõi trong mỗi người thực hiện. Một cách khác, phần cứng của người thực thi có thể hạn chế cách chúng ta xử lý dữ liệu với bộ nhớ có sẵn. Lý do tại sao bộ nhớ có thể tác động đến việc xử lý dữ liệu của chúng tôi là vì bộ nhớ của chúng tôi trong một người thực thi được chia thành bộ nhớ làm việc và lưu trữ. Và theo mặc định, cả làm việc và lưu trữ đều được phân bổ 50% tổng số bộ nhớ trên một nút. Và điều này rất quan trọng để xem xét bởi vì nếu chúng ta quyết định tồn tại dữ liệu của mình trong bộ nhớ, chúng ta có thể tiêu thụ một số bộ nhớ làm việc cần thiết để nói bộ lọc thông qua dữ liệu của chúng ta.

### c) Map Reduce và Shuffling

#### Map Reduce trong PySpark

#### Tạo và phân vùng RDD



**Hình 5.10.** MapReduce và Shuffling.

Dựa trên **hình 5.10**, giả sử ta có danh sách các phim:

```
movies = ['Shazam!', 'Minari', 'Captain Marvel', 'Pulp Fiction',
 'Casablanca', 'Michael Clayton', 'Sicario']
```

Ta chuyển danh sách này thành một RDD với 4 partitions:

```
rdd = sc.parallelize(movies, 4)
```

→ Khi sử dụng parallelize, ta tạo ra một Resilient Distributed Dataset (RDD) – một tập dữ liệu được phân tán trên các cores của executor.

#### Task có phải là Partition không?

Không hoàn toàn đúng! Nhưng mỗi partition thường sẽ được xử lý bởi một task.

Trong hình minh họa:

- RDD được chia thành 4 partitions khi sử dụng sc.parallelize(movies, 4).
- Spark phân chia dữ liệu trên 2 nodes. ( do ban đầu setup )
- Mỗi node có 2 partitions
- Mỗi partition được xử lý bởi một task.

Vậy, số task thường bằng số partition, nhưng không bắt buộc. Trong một số trường hợp:

- Nếu một node có nhiều CPU core, nó có thể xử lý nhiều task song song.
- Nếu cluster bị quá tải, Spark có thể điều chỉnh số task để phù hợp với tài nguyên.

#### Các phép biến đổi Map và Filter

- **Filter:** Ví dụ, ta tìm phim 'Michael Clayton' trong RDD:

```
rdd.filter(lambda movie: movie == 'Michael Clayton').collect()
```

Query được gửi từ driver đến các executors, mỗi partition thực hiện thao tác filter đồng thời và trả kết quả về driver.

- **Map:** Ví dụ, chuyển tất cả tên phim sang chữ in hoa:

```
upper_rdd = rdd.map(lambda movie: movie.upper()).collect()
print(upper_rdd)
Kết quả: ['SHAZAM!', 'MINARI', 'CAPTAIN MARVEL', 'PULP FICTION',
'CASABLANCA', 'MICHAEL CLAYTON', 'SICARIO']
```

Mỗi partition xử lý độc lập và sau đó các kết quả được thu gọn về driver. Quá trình này được gọi là map reduce – thực hiện map trên các phần tử rồi gộp kết quả lại.

### Shuffling

Shuffling là quá trình Spark cần chuyển đổi và phân phối lại dữ liệu giữa các partitions, thường xảy ra khi dữ liệu cần được gom nhóm lại theo một tiêu chí (ví dụ: group by, join...).

Shuffling đòi hỏi truyền dữ liệu qua mạng giữa các worker nodes, do đó thường tốn thời gian và tài nguyên, đặc biệt khi dữ liệu lớn hoặc băng thông hạn chế.

### Shuffling với GroupBy

Giả sử ta muốn nhóm các phim theo chữ cái đầu tiên:

```
grouped_rdd = rdd.groupBy(lambda movie: movie[0]).map(lambda x:
(x[0], list(x[1])))
print(grouped_rdd.collect())
```

Kết quả có thể là:

```
[('M', ['Minari', 'Michael Clayton']),
 ('S', ['Shazam!', 'Sicario']),
 ('C', ['Captain Marvel', 'Casablanca']),
 ('P', ['Pulp Fiction'])]
```

- **groupBy:** Phân nhóm các phần tử theo chữ cái đầu.
- Vì các phần tử cần được nhóm lại từ các partition khác nhau, quá trình này yêu cầu shuffling để chuyển dữ liệu giữa các executors nhằm đảm bảo rằng các phim cùng nhóm được đưa về cùng một partition.

### Chi tiết quá trình trong Spark UI

**Spark Jobs (?)**

User: root  
Total Uptime: 31 min  
Scheduling Mode: FIFO  
Completed Jobs: 3

Event Timeline

Completed Jobs (3)

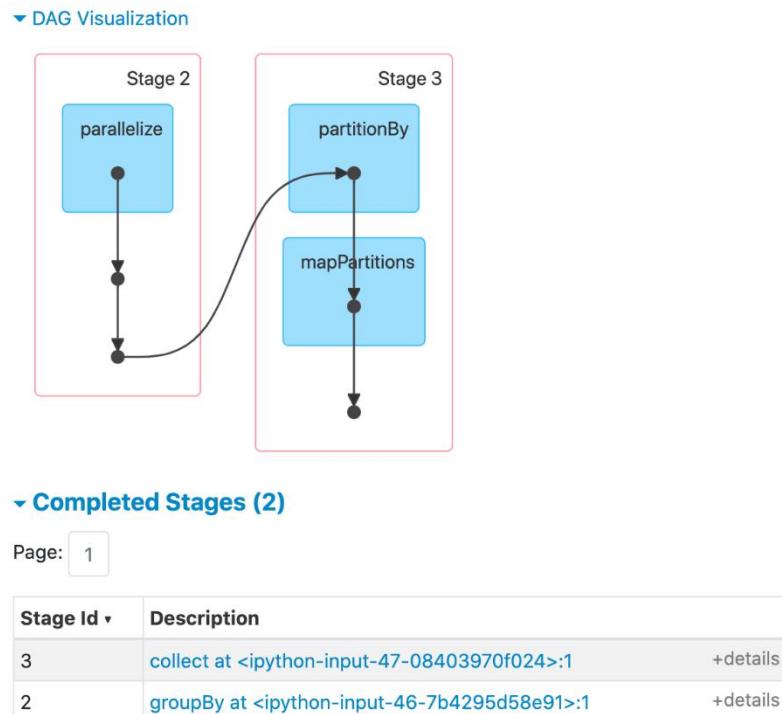
| Job Id | Description                                                                                  | Submitted           | Duration | Stages: Succeeded/Total | Tasks (for all stages): Succeeded/Total |
|--------|----------------------------------------------------------------------------------------------|---------------------|----------|-------------------------|-----------------------------------------|
| 2      | collect at <ipython-input-13-08403970f024>:1<br>collect at <ipython-input-13-08403970f024>:1 | 2025/03/24 06:49:45 | 1 s      | 2/2                     | 8/8                                     |
| 1      | collect at <ipython-input-11-09e9a7b7322c>:1<br>collect at <ipython-input-11-09e9a7b7322c>:1 | 2025/03/24 06:49:37 | 0.6 s    | 1/1                     | 4/4                                     |
| 0      | collect at <ipython-input-10-bf78ae522f90>:1<br>collect at <ipython-input-10-bf78ae522f90>:1 | 2025/03/24 06:47:51 | 2 s      | 1/1                     | 4/4                                     |

**Hình 5.11.** Quá trình trong Spark UI.

**Hình 5.11** hiển thị danh sách các Spark Jobs đã hoàn thành trong 1 ứng dụng Spark đang chạy, trong đó:

- Job 0: Filter: Lọc dữ liệu theo điều kiện.
- Job 1: Map: Biến đổi dữ liệu.
- Job 2: Group: Gom nhóm dữ liệu theo khóa.

### Filter + Map



**Hình 5.12.** Minh họa DAG visualization cơ chế filter và map.

**Hình 5.12** hiển thị DAG visualization từ giao diện người dùng của Apache Spark. Spark khởi tạo RDD từ tập dữ liệu ban đầu. filter được áp dụng để giữ lại chỉ những

phần tử phù hợp với điều kiện. map sau đó sẽ biến đổi từng phần tử của tập dữ liệu đầu vào.

- Filter lọc bỏ các phần tử không cần thiết.
- Map biến đổi từng phần tử một cách độc lập.

Parallelize: Đọc dữ liệu từ file hoặc tạo từ một tập dữ liệu có sẵn → FilterRDD: Loại bỏ những phần tử không phù hợp → MapRDD: Áp dụng phép biến đổi lên từng phần tử của RDD. Kết quả: Dữ liệu đã lọc và chuyển đổi, sẵn sàng để được nhóm lại.

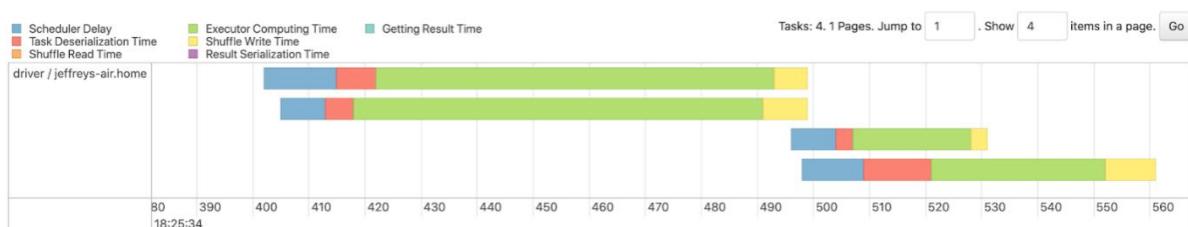
### groupBy

Gom nhóm dữ liệu theo một khóa cụ thể.

Sau khi dữ liệu đã được lọc và biến đổi, Spark nhóm dữ liệu theo khóa sử dụng groupBy. Quá trình này cần shuffle, vì dữ liệu có cùng khóa có thể đang nằm trên các executor khác nhau.

1. ShuffledRDD: Shuffle dữ liệu để đảm bảo cùng một khóa nằm trên cùng một executor.
2. PairwiseRDD: Gom nhóm dữ liệu theo khóa.

Kết quả: Dữ liệu đã được nhóm lại theo từng khóa cụ thể.



**Hình 5.13.** Hình minh họa biểu đồ thời gian thực thi các tiến trình trong Spark.

**Hình 5.13**, biểu đồ giúp người dùng hiểu rõ hơn về thời gian thực hiện của từng tác vụ và các giai đoạn trong quá trình thực hiện.

### Chú thích:

- Màu xanh lá nhạt (Executor Computing Time): Đây là thời gian Spark dành để thực hiện xử lý trên dữ liệu, chẳng hạn như thực hiện toán tử map hoặc filter.
- Màu vàng nhạt (Shuffle Write Time): Đây là thời gian dành cho việc ghi dữ liệu shuffle trước khi nó được chuyển đến các executor khác để nhóm lại (groupBy).
- Màu xanh dương (Scheduler Delay): Thời gian Spark chờ trước khi bắt đầu thực thi tác vụ.
- Màu đỏ (Task Deserialization Time): Thời gian cần để deserialization dữ liệu trước khi thực hiện tính toán

- Màu cam (Shuffle Read Time): Thời gian đọc dữ liệu sau khi shuffle.

#### Nhận xét:

- Thời gian tính toán (Executor Computing Time) chiếm phần lớn, cho thấy bước map và filter mất nhiều thời gian.
- Shuffle Time nhỏ vì tất cả dữ liệu đều nằm trên cùng một máy tính, dẫn đến chi phí trao đổi dữ liệu thấp.
- groupBy xảy ra trong Stage 2, do đó việc tổ chức lại dữ liệu (shuffling) xảy ra vào thời điểm này.

#### d) Lazy RDDs

Lazy RDDs (Resilient Distributed Datasets lười thực thi) là một trong những đặc điểm quan trọng giúp Apache Spark tối ưu hóa hiệu suất tính toán. Khác với các hệ thống truyền thống thực thi ngay khi có lệnh gọi, Spark sử dụng cơ chế Lazy Evaluation (tính toán lười) để trì hoãn việc thực thi cho đến khi cần thiết. Điều này giúp giảm chi phí tính toán không cần thiết và tận dụng tối đa tài nguyên hệ thống.

- Transformations: các phương thức không thực hiện xử lý dữ liệu ngay lập tức: map, filter, chỉ xây dựng kế hoạch xử lý
- Actions: các phương thức xử lý dữ liệu và trả về kết quả cho người dùng như: collect, take, count

#### Ưu điểm khi thực hiện Lazy RDDs

- Tối ưu hóa hiệu suất: giúp Spark tự động tối ưu hóa chuỗi các thao tác, giảm số lần truy xuất dữ liệu, Spark có thể nhóm nhiều transformation lại với nhau và chạy tối ưu nhất.
- Giảm chi phí tính toán: Nếu không cần kết quả, Spark sẽ không thực thi. (không có action nào được gọi thì Spark không thực hiện)
- Tránh việc tính toán dư thừa: Nếu thực hiện nhiều thao tác, Spark có thể bỏ qua những thao tác không cần thiết.

#### Cơ chế hoạt động

Khi thực hiện một thao tác trên RDD (như map, filter), Spark không thực hiện ngay lập tức. Thay vào đó, Spark chỉ ghi nhớ chuỗi các thao tác này trong một biểu đồ tính toán (DAG - Directed Acyclic Graph) và chỉ thực thi khi có một hành động (action) yêu cầu kết quả, như collect() hoặc count() được gọi.

Ví dụ:

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd_filtered = rdd.filter(lambda x: x % 2 == 0)
```

Đã gọi filter(), nhưng Spark chưa thực thi mà chỉ lưu thao tác này. Chỉ khi có hành động collect() thì Spark mới thực hiện việc lọc dữ liệu và trả về kết quả

```
print(rdd_filtered.collect())
```

Khi gọi các transformation như map() hoặc filter(), Spark chỉ tạo một kế hoạch xử lý (DAG), nhưng không thực thi ngay lập tức. Chỉ khi có một action như collect() hoặc take(), Spark mới thực sự chạy computation trên RDD.

- movies\_rdd.map(lambda movie: movie.title())

Chỉ gọi transformation map() → không có kết quả trả về, hiển thị 1 RDD Object

- movies\_rdd.map(lambda movie: movie.title()).filter(lambda movie: movie[0] == 'd')

Gọi cả 2 transformation map() và filter() → không có kết quả trả về vì chưa có action

- movies\_rdd.filter(lambda movie: movie[0] == 'd').map(lambda movie: movie.title()).collect()

Gọi action collect() → Spark thực thi DAG, lọc và chuyển đổi dữ liệu, sau đó trả về kết quả cho người dùng

## Đánh giá

Thực hiện so sánh trong cách Spark xử lý dữ liệu khi sử dụng 2 action khác nhau là collect() và take(1)

### Đối với collect():

*movies\_rdd.map(lambda movie: movie.title()).collect()*



**Hình 5.14.** Giao diện của Spark UI hiển thị quá trình xử lý song song.

Dựa trên **hình 5.14**, giao diện của Spark UI trong quá trình xử lý song song

- Các thanh màu khác nhau đại diện cho các giai đoạn xử lý khác nhau
- Khi dùng collect(), Spark phải xử lý qua tất cả các partition (phân vùng) của dữ liệu
- Bạn có thể thấy các thanh màu trải dài qua toàn bộ biểu đồ, thể hiện thời gian xử lý trên tất cả các partition

## Đối với take(1):

`movies_rdd.map(lambda movie: movie.title()).take(1)`



**Hình 5.15.** Giao diện của Spark UI hiển thị quá trình xử lý song song.

Dựa trên **hình 5.15**, giao diện của Spark UI trong quá trình xử lý song song

- Thay vì xử lý tất cả các partition, Spark chỉ cần xử lý một partition (phân vùng dữ liệu) để lấy ra kết quả đầu tiên
- Các thanh màu trong biểu đồ ngắn hơn và chỉ xuất hiện ở một phần của biểu đồ
- Điều này minh họa rằng Spark đã tối ưu hóa quá trình xử lý bằng cách chỉ làm việc trên phần dữ liệu cần thiết

## Nhận xét:

- Spark chỉ thực hiện đúng lượng công việc cần thiết để đáp ứng yêu cầu. Ví dụ, khi sử dụng `take(1)`, Spark chỉ xử lý đủ dữ liệu để trả về một kết quả, thay vì thực thi toàn bộ pipeline trên toàn bộ tập dữ liệu.
- Lazy Evaluation giúp Spark tối ưu hóa hiệu suất, đặc biệt khi làm việc với các tập dữ liệu lớn, bằng cách hoãn thực thi transformations cho đến khi một action được gọi.
- Hệ thống lập lịch và tối ưu hóa của Spark sẽ nhóm các transformations lại để giảm số lần quét dữ liệu, giảm thời gian tính toán và tiết kiệm tài nguyên.

## e) Spark Workflow

Spark Workflow là Quy trình làm việc của Spark, mô tả cách Spark thực thi các phép toán (operations) trên dữ liệu. Nó bao gồm các khái niệm chính như Jobs, Stages và Tasks, tất cả đều được thiết kế để xử lý dữ liệu phân tán một cách hiệu quả.

Spark Workflow gồm:

- Action → Gây ra Job
- Job → Gồm nhiều Stages
- Stage → Chứa nhiều Tasks
- Task → Xử lý dữ liệu trên một partition

## Jobs (Công việc)

- Trong Spark, transformations (map, filter, groupBy, etc.) chỉ tạo ra một chuỗi các thao tác (DAG – Directed Acyclic Graph), nhưng không thực thi ngay.
- Action (take, collect, count, saveAsTextFile, etc.) mới kích hoạt Spark Job. Một Job trong Spark tương ứng với một hành động (action) được thực thi.
- Mỗi lần gọi action, Spark sẽ bắt đầu một job.

```
movies_rdd = sc.textFile("movies.csv") # Transformation (không chạy ngay)
```

```
first_movie = movies_rdd.take(1) # Action (gây ra Job)
```

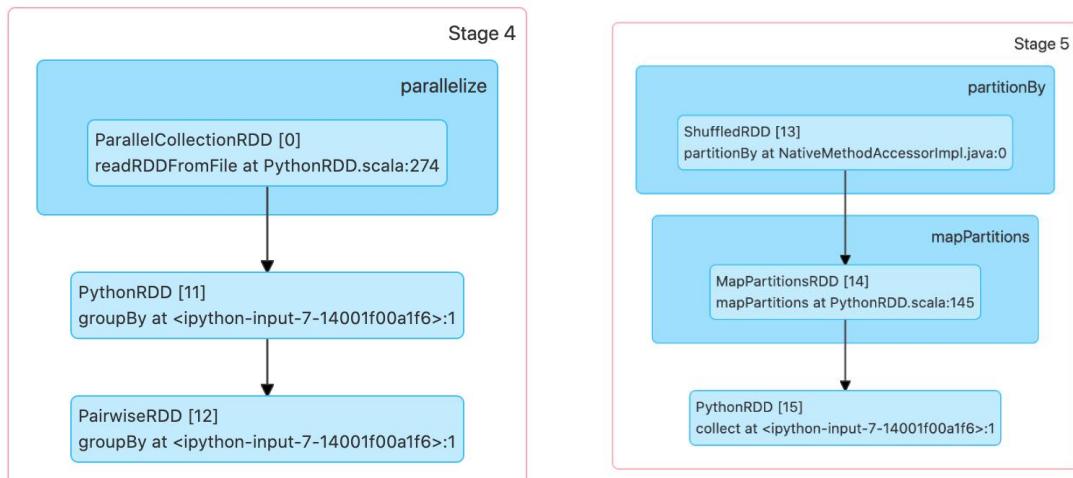
→ Action take(1) kích hoạt 1 job để lấy dữ liệu

### Stage (Giai đoạn)

- Một Job có thể được chia thành nhiều Stages.
- Stages đại diện cho các nhóm logic của các phép toán có thể được thực hiện cùng một lúc.
- Stages thường được phân chia dựa trên các phép toán shuffle. Shuffle là quá trình xáo trộn và phân phối lại dữ liệu giữa các phân vùng (partitions).

```
movies_rdd.map(lambda word: word.title()). \
groupByKey(lambda title: title[0]). \
map(lambda group: (group[0], len(group[1]))).collect()
```

- Các Stages thường được chia thành hai loại:
  - ShuffleMapStage: Giai đoạn này bao gồm các phép toán xảy ra trước shuffle, như đọc dữ liệu và nhóm dữ liệu.
  - ResultStage: Giai đoạn này bao gồm các phép toán xảy ra sau shuffle, như tính toán kết quả cuối cùng.



**Hình 5.16.** Hai giai đoạn của Stage.

**Hình 5.16** cho thấy phép toán groupBy() có thể chia thành 2 stages, 1 stage trước shuffle và 1 stage sau shuffle

- Stage 4 (trước shuffle): groupBy() thực hiện nhóm dữ liệu cục bộ trên mỗi phân vùng, chưa có sự trao đổi dữ liệu giữa các executors.
- Stage 5 (sau shuffle): partitionBy thực hiện shuffle để sắp xếp lại dữ liệu và đảm bảo tất cả các bản ghi có cùng key nằm trên cùng một executor. Sau đó, mapPartitions được thực hiện trên dữ liệu đã được shuffle.

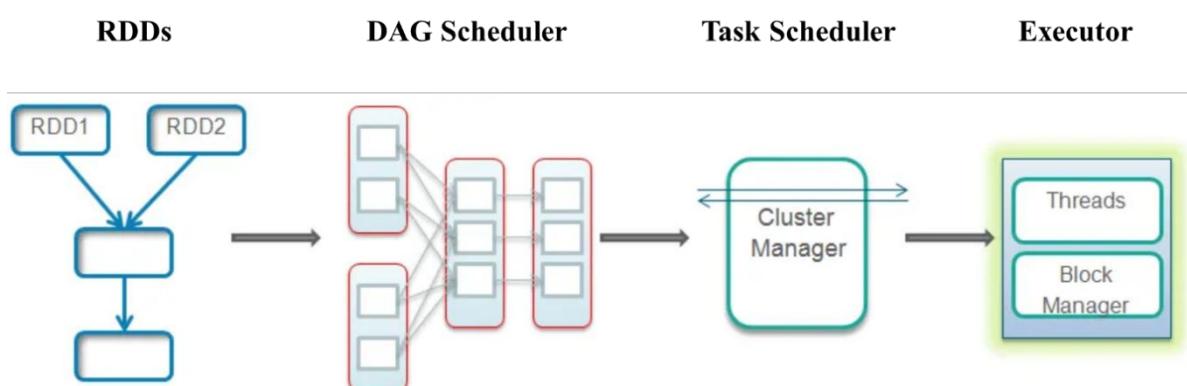
Điều này giúp Spark tối ưu hóa hiệu suất bằng cách thực hiện nhóm dữ liệu cục bộ trước khi thực hiện shuffle tốn kém.

### Task (Tác vụ)

- Mỗi Stage được thực hiện trên nhiều phân vùng dữ liệu song song.
- Một Task là đơn vị thực thi nhỏ nhất trong Spark, đại diện cho việc xử lý một phân vùng dữ liệu cụ thể.
- Mỗi phân vùng dữ liệu sẽ được xử lý bởi một Task.
- Số lượng Task trong một Stage thường bằng số lượng phân vùng dữ liệu.

Ví dụ: Nếu một stage phải xử lý 10 phân vùng dữ liệu, stage đó sẽ có 10 tasks.

### Quy trình làm việc



**Hình 5.17.** Quy trình thực thi của Spark.

Quy trình thực thi của Spark dựa trên **hình 5.17** được thực hiện theo các bước sau:

- Bước 1: Xây dựng RDD/DataFrame/Dataset: Bạn bắt đầu bằng cách tạo một RDD (Resilient Distributed Dataset), DataFrame hoặc Dataset từ dữ liệu nguồn.
- Bước 2: Áp dụng Transformations: Bạn áp dụng các phép toán biến đổi (transformations) lên dữ liệu. Các phép toán này tạo ra các RDD/DataFrame/Dataset mới mà không thực sự tính toán ngay lập tức.

- Bước 3: Thực thi Actions: Khi bạn gọi một hành động (action), Spark sẽ tạo ra một Job.
- Bước 4: Chia thành Stages: Job được chia thành một hoặc nhiều Stages, dựa trên các phép toán shuffle.
- Bước 5: Phân chia thành Tasks: Mỗi Stage được chia thành nhiều Tasks, mỗi Task xử lý một phân vùng dữ liệu.
- Bước 6: Thực thi song song: Spark thực thi các Tasks song song trên các worker nodes trong cluster.
- Bước 7: Thu thập kết quả: Kết quả từ các Tasks được thu thập và trả về cho driver program.

### f) Resilient RDDs

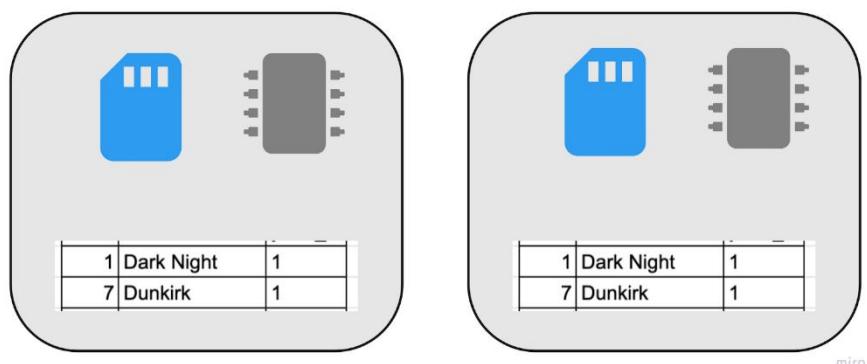
- RDD là một tập dữ liệu được lưu trữ trên nhiều node trong một cluster Spark.
- Các executor (tiến trình thực thi) trong cluster sẽ truy vấn và thao tác trên dữ liệu này song song (parallel processing) để tăng tốc độ xử lý.
- Tuy nhiên, vì dữ liệu chủ yếu được lưu trong bộ nhớ (in-memory storage), Spark cần có một cơ chế để đảm bảo không bị mất dữ liệu nếu một node bị lỗi.

#### Tại sao cần Resilient (tính chịu lỗi)

##### Vấn đề:

Khi lưu dữ liệu trong bộ nhớ RAM (không lưu vào ổ cứng), nếu một node bị lỗi, dữ liệu trên đó sẽ bị mất.

##### Giải pháp truyền thống trong hệ thống phân tán:



**Hình 5.18.** Hai node lưu trữ dữ liệu giống nhau.

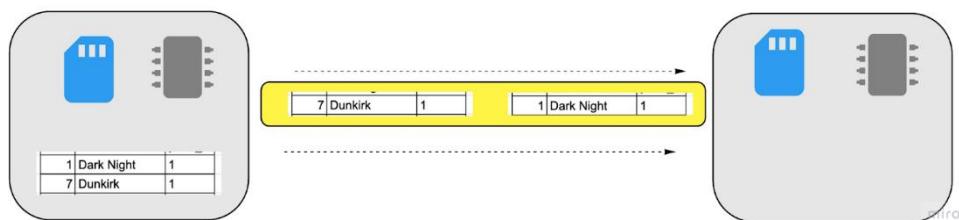
- Các hệ thống dữ liệu phân tán thường sao chép dữ liệu sang nhiều node khác để có bản dự phòng (replication) như **hình 5.18**

- Nếu một node bị lỗi, dữ liệu vẫn có thể được khôi phục từ một bản sao trên node khác.

### Nhược điểm của việc Replication (sao chép dữ liệu)

- Tốn nhiều dung lượng lưu trữ do phải sao chép dữ liệu trên nhiều node.
- Tăng độ trễ (latency) vì dữ liệu phải được truyền qua mạng khi có thay đổi, gây chậm nếu băng thông thấp.

### Cách tiếp cận Spark



*Hình 5.19. Giao tiếp dữ liệu giữa các node.*

- Không sao chép dữ liệu giữa các node như cách truyền thống.
- Thay vào đó, Spark ghi nhớ tất cả các bước để tạo ra RDD ban đầu (gọi là lineage - lịch sử tái tạo dữ liệu) như **hình 5.19**
- Khi một node bị lỗi, Spark có thể tái tạo lại dữ liệu bị mất bằng cách áp dụng lại các phép biến đổi (transformations) trên RDD gốc.

### Ưu điểm:

- Tiết kiệm bộ nhớ, không cần lưu nhiều bản sao của dữ liệu.
- Nhanh hơn vì không cần sao chép dữ liệu liên tục giữa các node.
- Dữ liệu có thể phục hồi một cách tự động dựa trên lineage.

#### 5.3.4. Directed Acyclic Graph (DAG) và cơ chế lập lịch

##### a) DAG Scheduler

DAG Scheduler là thành phần trung tâm trong cơ chế xử lý song song của Spark. Khi một hành động được gọi, DAG Scheduler thực hiện các bước sau [1]:

1. Xây dựng đồ thị thực thi dựa trên lineage của RDD.
2. Chia đồ thị thành các giai đoạn (stages) dựa trên các phép shuffle.
3. Tạo ra các tập nhiệm vụ (TaskSet) cho mỗi giai đoạn.
4. Gửi các TaskSet đến Task Scheduler để thực thi.

Việc chia thành các giai đoạn dựa trên phép shuffle là một khía cạnh quan trọng trong tối ưu hóa xử lý song song. Các phép biến đổi trong cùng một giai đoạn có thể được "pipeline" để giảm thiểu chi phí I/O và cải thiện hiệu suất [17].

### b) DAG Scheduler

Task Scheduler nhận các TaskSet từ DAG Scheduler và chịu trách nhiệm phân phối các nhiệm vụ đến các executor. Spark sử dụng nhiều chiến lược lập lịch khác nhau, bao gồm:

1. **FIFO Scheduling**: Các nhiệm vụ được thực thi theo thứ tự đến trước.
2. **Fair Scheduling**: Tài nguyên được phân phối công bằng giữa các ứng dụng.
3. **Delay Scheduling**: Cải thiện tính cục bộ của dữ liệu bằng cách trì hoãn các nhiệm vụ không có dữ liệu cục bộ.
4. **Speculation**: Phát hiện và khởi chạy lại các nhiệm vụ chậm (straggler tasks).

Các chiến lược lập lịch này giúp tối ưu hóa việc sử dụng tài nguyên và cải thiện hiệu suất xử lý song song [18].

### 5.3.5. Cơ chế quản lý bộ nhớ

#### a) Memory Manager

Spark sử dụng một hệ thống quản lý bộ nhớ tinh vi để tối ưu hóa hiệu suất xử lý dữ liệu. Bộ nhớ trong Spark được chia thành ba phần chính:

1. **Execution Memory**: Dùng cho việc thực thi các phép biến đổi (ví dụ: join, sort, aggregation).
2. **Storage Memory**: Dùng cho việc cache dữ liệu và broadcast variables.
3. **User Memory**: Dành cho cấu trúc dữ liệu do người dùng định nghĩa.

Memory Manager của Spark sử dụng thuật toán LRU (Least Recently Used) để quản lý bộ nhớ cache và có khả năng tự động điều chỉnh phân bổ bộ nhớ giữa execution và storage dựa trên nhu cầu [19].

#### b) Tungsten

Tungsten là dự án tối ưu hóa bộ nhớ của Spark, nhằm mục đích cải thiện hiệu suất bằng cách:

1. **Binary Processing**: Xử lý dữ liệu ở định dạng nhị phân thay vì đối tượng Java.
2. **Cache-aware Computation**: Tối ưu hóa việc sử dụng bộ nhớ đệm của CPU.
3. **Code Generation**: Tạo mã bytecode tại thời điểm chạy để tối ưu hóa thực thi.

Các nghiên cứu đã chỉ ra rằng Tungsten có thể cải thiện hiệu suất xử lý Spark lên đến 10 lần trong một số trường hợp [20].

### **5.3.6. Cơ chế xử lý luồng (Spark Streaming)**

#### **a) Discretized Streams (DStreams)**

Spark Streaming mở rộng mô hình RDD để hỗ trợ xử lý dữ liệu luồng theo thời gian thực. Nó sử dụng khái niệm Discretized Streams (DStreams), đại diện cho một chuỗi các RDD được tạo ra trong các khoảng thời gian cố định.

DStreams cho phép Spark áp dụng cùng một mô hình lập trình và cơ chế xử lý song song của batch processing cho streaming processing, mang lại nhiều lợi ích [8]:

1. **Unified Programming Model:** Cùng API cho cả batch và streaming.
2. **Exactly-once Semantics:** Đảm bảo mỗi bản ghi được xử lý đúng một lần.
3. **Fault Tolerance:** Khả năng phục hồi sau sự cố mà không mất dữ liệu.
4. **Integration:** Tích hợp liền mạch với các thành phần khác của hệ sinh thái Spark.

#### **b) Structured Streaming**

Structured Streaming là mô hình xử lý luồng mới của Spark, xem dữ liệu luồng như một bảng vô hạn liên tục được cập nhật. Mô hình này cung cấp:

1. **Event-time Processing:** Xử lý dựa trên thời gian sự kiện thay vì thời gian đến.
2. **Watermarking:** Cơ chế để xử lý dữ liệu đến muộn.
3. **State Management:** Quản lý trạng thái tự động cho các phép aggregate.
4. **Continuous Processing:** Giảm độ trễ xử lý xuống mức mili-giây.

Armbrust et al. [21] đã chứng minh Structured Streaming có thể đạt được throughput cao hơn và độ trễ thấp hơn so với các hệ thống xử lý luồng truyền thống.

### **5.3.7. Tối ưu hóa hiệu suất xử lý song song**

#### **a) Catalyst Optimizer**

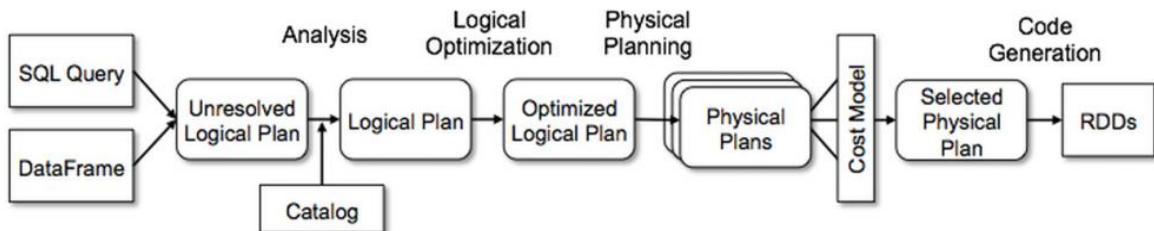
Catalyst Optimizer trong Spark SQL là một framework tối ưu hóa mạnh mẽ giúp xử lý dữ liệu song song một cách hiệu quả. Nó giúp cải thiện hiệu suất truy vấn bằng cách tự động tối ưu hóa logic và kế hoạch thực thi SQL hoặc DataFrame API mà ta viết.

#### **Ưu điểm:**

- Xử lý dữ liệu lớn: Spark được thiết kế để xử lý lượng dữ liệu rất lớn, nên cần một bộ tối ưu hóa giúp chọn cách thực thi tốt nhất.

- Tự động tối ưu hóa: Catalyst tự động áp dụng các chiến lược tối ưu mà không cần lập trình viên phải can thiệp thủ công.
- Hỗ trợ cả SQL và DataFrame: Catalyst không chỉ tối ưu các câu lệnh SQL mà còn tối ưu cả các API DataFrame.

### Quy trình tối ưu hóa trong Catalyst Optimizer:



**Hình 5.20.** Quy trình tối ưu hóa truy vấn trong Catalyst Optimizer.

Quy trình tối ưu hóa truy vấn trong Catalyst Optimizer như **hình 5.20**, có 4 giai đoạn chính, bao gồm:

- Analysis (Phân tích): Kiểm tra và xác định các quan hệ, thuộc tính chưa được giải quyết trong truy vấn.
- Logical Optimization (Tối ưu hóa logic): Áp dụng các quy tắc tối ưu như Constant Folding, Predicate Pushdown, Projection Pruning.
- Physical Planning (Lập kế hoạch vật lý): Tạo một hoặc nhiều kế hoạch thực thi và chọn kế hoạch có chi phí thấp nhất (Cost-based Optimization).
- Code Generation (Tạo mã tối ưu): Dịch truy vấn thành mã Java bytecode để tăng tốc độ thực thi.

Armbrust et al. [3] đã báo cáo rằng Catalyst có thể cải thiện hiệu suất truy vấn lên đến 10 lần so với RDD API truyền thống.

#### b) Các chiến lược tối ưu hóa khác

Ngoài Catalyst, Spark còn sử dụng nhiều chiến lược tối ưu hóa khác để cải thiện hiệu suất xử lý song song, bao gồm:

1. **Dynamic Partition Pruning:** Loại bỏ các phân vùng không cần thiết dựa trên các điều kiện filter.
2. **Adaptive Query Execution:** Điều chỉnh kế hoạch thực thi dựa trên thông kê runtime.
3. **Broadcast Join:** Sử dụng broadcast variables để tối ưu hóa join giữa bảng lớn và bảng nhỏ.

4. **Bloom Filter Push Down:** Sử dụng bộ lọc Bloom để giảm lượng dữ liệu cần đọc và truyền qua mạng.

Các nghiên cứu đã chỉ ra rằng những chiến lược này có thể giảm đáng kể thời gian thực thi và sử dụng tài nguyên trong các tác vụ phân tích dữ liệu phức tạp [22].

### **5.3.8. Kết luận**

Apache Spark đã cách mạng hóa lĩnh vực xử lý dữ liệu lớn với cơ chế xử lý song song và phân tán tiên tiến. Trong chương này, chúng tôi đã phân tích sâu về kiến trúc, RDD, DAG, cơ chế quản lý bộ nhớ, và các chiến lược tối ưu hóa của Spark. Những đặc điểm này không chỉ giúp Spark đạt được hiệu suất vượt trội so với các framework trước đó, mà còn cung cấp một mô hình lập trình thống nhất và linh hoạt cho nhiều loại ứng dụng phân tích dữ liệu.

Tuy nhiên, Spark vẫn đối mặt với một số thách thức, đặc biệt là trong việc xử lý luồng thời gian thực với độ trễ cực thấp và tối ưu hóa sử dụng tài nguyên trong môi trường đa người dùng. Các nghiên cứu trong tương lai nên tập trung vào việc giải quyết những thách thức này và mở rộng khả năng của Spark để đáp ứng nhu cầu ngày càng tăng của các ứng dụng phân tích dữ liệu hiện đại.

## **5.4. Ví dụ minh họa về cơ chế xử lý dữ liệu phân tán**

### **5.4.1. Bài toán cộng vector**

#### **a) Cơ chế xử lý**

Bài toán cộng hai vector có kích thước lớn đặt ra thách thức về hiệu suất khi thực hiện trên một hệ thống đơn lẻ. Để giải quyết vấn đề này, việc tận dụng các framework phân tán như Apache Spark là một giải pháp hiệu quả. Cơ chế xử lý phép cộng hai vector trong môi trường Spark được thực hiện theo các giai đoạn chính sau:

- Phân tán dữ liệu:

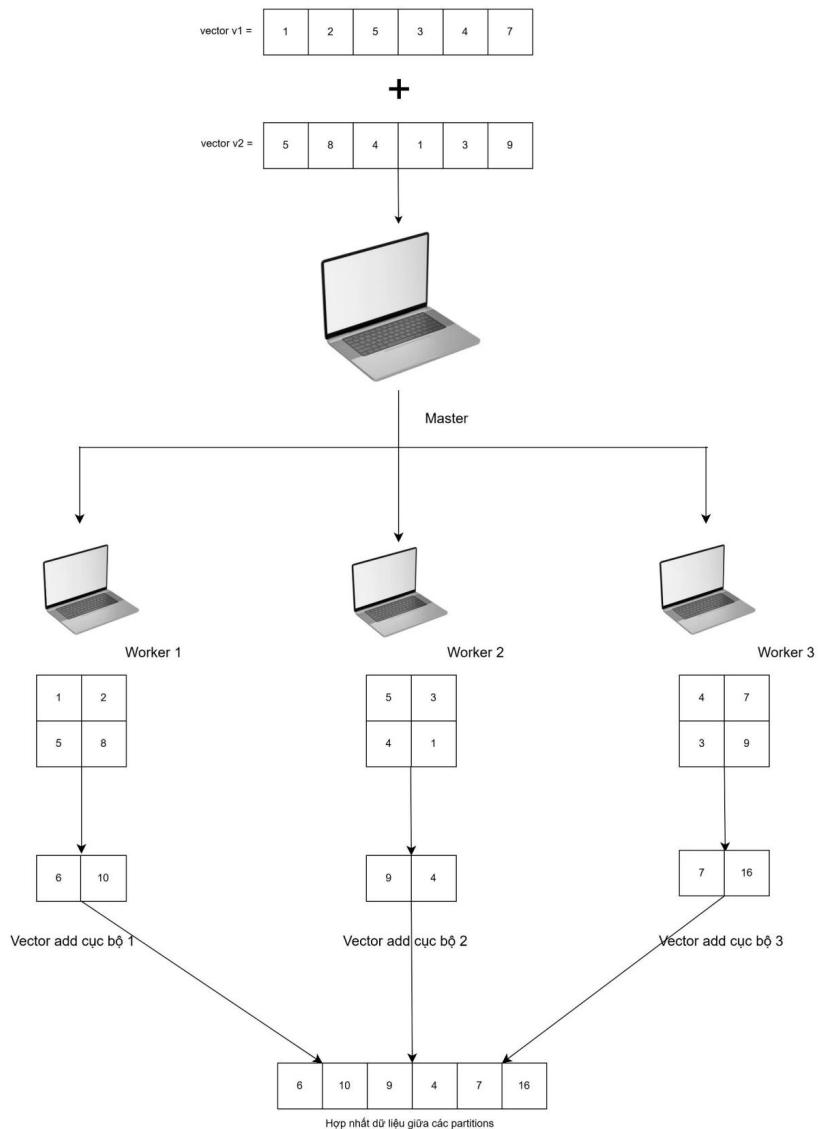
Hai vector đầu vào, ký hiệu là v1 và v2, được phân chia thành một số lượng P các phân vùng (partitions). Mỗi phân vùng này được biểu diễn dưới dạng một Resilient Distributed Dataset (RDD), là đơn vị lưu trữ và xử lý dữ liệu song song trong Spark. Quá trình phân tán đảm bảo rằng mỗi nút worker trong cluster sẽ chịu trách nhiệm xử lý một phần dữ liệu.

- Xử lý song song tại các node worker:

Sau khi dữ liệu được phân tán, mỗi nút worker sẽ thực hiện phép cộng tương ứng trên các phần tử thuộc các phân vùng được giao. Cụ thể, với mỗi phân vùng  $i$  ( $1 \leq i \leq P$ ), các phần tử từ phân vùng thứ  $I$  của  $v1$  sẽ được kết hợp với các phần tử từ phân vùng thứ  $i$  của  $v2$  thông qua phép toán zip. Tiếp theo, chúng ta sẽ áp dụng một hàm ánh xạ để thực hiện phép cộng từng phần tử tương ứng trong cặp đã được kết hợp. Quá trình này được thực hiện song song trên tất cả các nút worker, tận dụng khả năng tính toán song song của Spark.

Kết quả của phép cộng tại mỗi phân vùng sẽ được tạo thành một phần của vectơ ết quả cuối cùng. Trong bài toán cộng vectơ, không có bước hợp nhất dữ liệu như trong các thuật toán sắp xếp phức tạp. Thay vào đó, vectơ kết quả được hình thành từ các kết quả được tính toán song song tại các nút worker.

Ví dụ:



**Hình 5.21.** Hình ảnh minh họa bài toán cộng 2 vectơ phân tán.

Dựa trên **hình 5.21**, chúng ta tiến hành xét hai vectơ  $v1 = [1, 2, 5, 3, 4, 7]$  và  $v2 = [5, 8, 4, 1, 3, 9]$ . Với cấu hình cluster gồm 1 master và 3 worker, giả sử chúng ta chia mỗi vector thành 3 partitions.

- Partition 1 ( $v1$ ): [1,2]
- Partition 1 ( $v2$ ): [5,8]
- Partition 2 ( $v1$ ): [5,3]
- Partition 2 ( $v2$ ): [4,1]
- Partition 3 ( $v1$ ): [4,7]
- Partition 3 ( $v2$ ): [3,9]

Các worker thực hiện phép cộng trên các partitions được gán:

- Worker 1:  $[1 + 5, 2 + 8] \rightarrow [6, 10]$
- Worker 2:  $[5 + 4, 3 + 1] \rightarrow [9, 4]$
- Worker 3:  $[4 + 3, 7 + 9] \rightarrow [7, 16]$

Vectơ kết quả thu được tổng hợp từ các worker: [6,10,9,4,7,16]

### b) Cài đặt bài toán cộng vectơ trên Spark

- Cấu hình một Spark cluster gồm 1 master và nhiều worker sử dụng Docker:

```
version: '3.7'
services:
 spark-master:
 image: bitnami/spark:3.5.1
 command: bin/spark-class org.apache.spark.deploy.master.Master
 ports:
 - 8080:8080
 - 7077:7077
 deploy:
 resources:
 limits:
 cpus: "2.0"
 memory: "2GB"
 reservations:
 cpus: "1.0"
 memory: "1GB"
 volumes:
 - ./conf/spark-defaults.conf:/opt/bitnami/spark/conf/spark-
default.conf
 - ./app:/app # persistent storage
 spark-worker:
 image: bitnami/spark:3.5.1
```

```

command: bin/spark-class org.apache.spark.deploy.worker.Worker
spark://spark-master:7077
depends_on:
 - spark-master
environment:
 SPARK_MODE: worker
 SPARK_WORKER_CORES: 2
 SPARK_WORKER_MEMORY: 2g
 SPARK_MASTER_URL: spark://spark-master:7077
deploy:
 resources:
 limits:
 cpus: "2.0"
 memory: "2GB"
 reservations:
 cpus: "1.0"
 memory: "1GB"
 volumes:
 - ./conf/spark-defaults.conf:/opt/bitnami/spark/conf/spark-
defaults.conf
 - ./app:/app # persistent storage

jupyter-local:
 depends_on:
 - spark-master
 build: .
 command: python -m jupyterlab --ip "0.0.0.0" --no-browser --
NotebookApp.token=''
 ports:
 - 8888:8888
 volumes:
 - ./app:/app # persistent storage
 environment:
 - JUPYTER_ENABLE_LAB=yes

volumes:
 spark-logs:

```

- Viết mã PySpark để thực hiện cộng 2 vectơ trên Spark RDD

Mã giả:

```

Input: Vectơ A (kích thước n), Vectơ B (kích thước n)
Output: Vectơ kết quả C (kích thước n)

```

- 1.Khởi tạo SparkSession và SparkContext.
- 2.Phân chia vectơ A thành P phân vùng dưới dạng RDD.
- 3.Phân chia vectơ B thành P phân vùng dưới dạng RDD.
- 4.Ghép cặp các phần tử vectơ A và vectơ B bằng zip
- 5.Áp dụng mapPartitions() để cộng từng cặp phần tử.
- 6.Thu thập toàn bộ kết quả vào vectơ kết quả C bằng collect()
- 7.Kết thúc SparkSession

Ban đầu ta sẽ khởi tạo môi trường tính toán phân tán Spark, cho phép tương tác với cluster và thực hiện các phép tính song song.

Ta sẽ chuyển dữ liệu từ vectơ A, B sang dạng RDD và phân chia thành P phân vùng tương ứng để tối ưu việc xử lý phân tán.

Sau đó ta sẽ sử dụng hàm `zip` để kết hợp từng phần tử tương ứng của hai RDD và tạo thành các cặp (a, b) cho các phép tính sau. Dùng phương thức `mapPartitions` để xử lý theo từng phân vùng, trong đó thực hiện phép cộng từng cặp số (a,b) một cách song song, giúp giảm overhead chuyển đổi giữa các task.

Cuối cùng ta dùng hàm `collect()` để thu thập toàn bộ dữ liệu kết quả từ các phân vùng về driver và tạo thành vectơ kết quả C hoàn chỉnh và đóng phiên làm việc của Spark để giải phóng tài nguyên và kết thúc quá trình tính toán.

### c) Kết quả thực nghiệm

#### Môi trường thực nghiệm:

Cấu hình một Spark Cluster gồm 1 master và nhiều worker trên môi trường Docker với cấu hình cụ thể như sau:

- Docker Image: bitnami/spark:3.5.1
- Cấu hình Spark Master: 2 vCPU, 2GB RAM
- Cấu hình Spark Worker: 2 vCPU, 2GB RAM

Tiến hành thực nghiệm trên các trường hợp sau:

- 1 master 0 worker
- 1 master 1 worker
- 1 master 2 worker
- 1 master 3 worker

Dữ liệu:

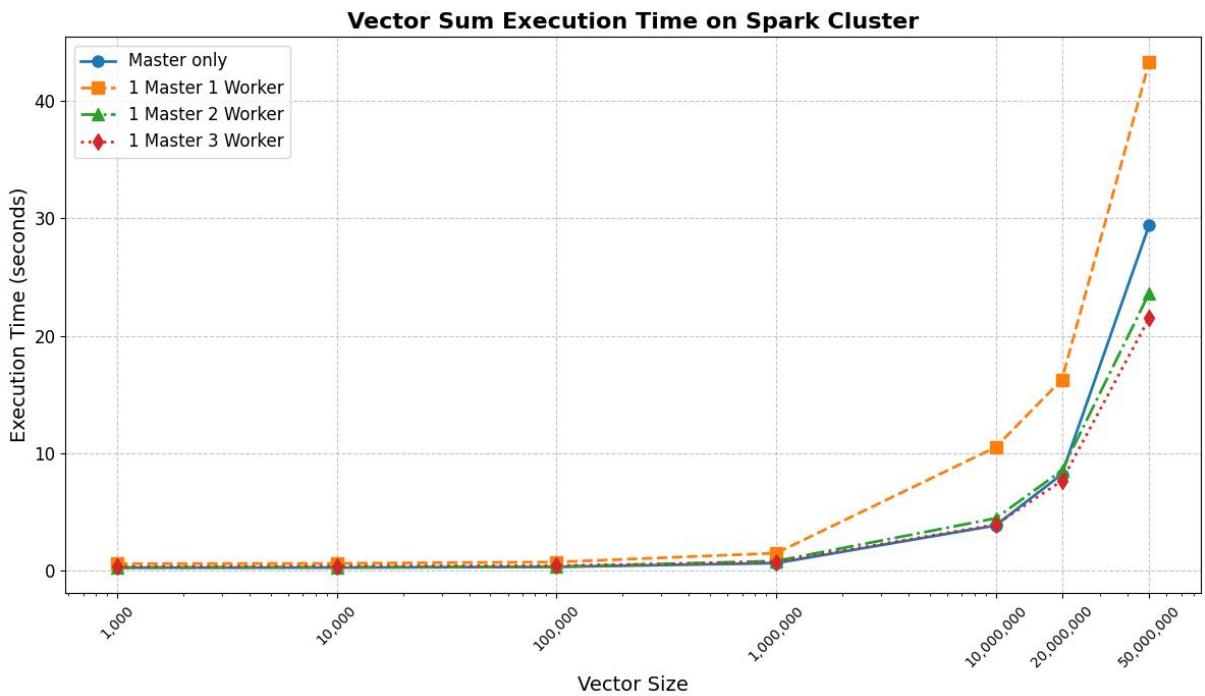
Gồm các mảng số nguyên dương với các kích thước sau:

- 1000 phần tử
- 10000 phần tử
- 100000 phần tử
- 1000000 phần tử
- 10000000 phần tử
- 20000000 phần tử
- 50000000 phần tử

## Kết quả:

**Bảng 5.1.** Kết quả thời gian chạy cộng hai vector của các cấu hình Spark (Giây).

| VectorSize | Result  | Master only | 1 Master 1 Worker | 1 Master 2 Worker | 1 Master 3 Worker |
|------------|---------|-------------|-------------------|-------------------|-------------------|
| 1000       | Correct | <b>0.24</b> | <b>0.58</b>       | 0.26              | 0.33              |
| 10000      | Correct | 0.25        | 0.61              | 0.28              | 0.38              |
| 100000     | Correct | 0.31        | 0.72              | 0.32              | 0.41              |
| 1000000    | Correct | 0.63        | 1.48              | 0.81              | 0.72              |
| 10000000   | Correct | 3.84        | 10.51             | 4.45              | 3.91              |
| 20000000   | Correct | 8.20        | 16.26             | 8.52              | <b>7.64</b>       |
| 50000000   | Correct | 29.45       | <b>43.31</b>      | 23.64             | <b>21.52</b>      |



**Hình 5.22.** Biểu đồ thời gian chạy cộng hai vector của các cấu hình Spark.

Từ **bảng 5.1**, ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như **hình 5.22**, ta có nhận xét như sau:

Với kích thước dữ liệu từ  $10^3$  đến  $10^5$  phần tử, cấu hình "Master only" cho hiệu suất tối ưu. Hiện tượng này có thể được giải thích bởi chi phí overhead của việc phân phối tác vụ vượt trội so với lợi ích của việc xử lý song song. Điều này phù hợp với nghiên cứu của Li et al. (2019) về overhead trong hệ thống phân tán [14].

Cấu hình "1 Master 1 Worker" thể hiện hiệu suất kém nhất trong mọi kích thước dữ liệu, với thời gian xử lý cao hơn 142% so với "Master only" ở dữ liệu kích thước  $10^3$ . Kết quả này minh họa rõ chi phí truyền dữ liệu, khởi tạo tác vụ, và tổng hợp kết quả trong môi trường phân tán.

Kết quả thực nghiệm chỉ ra một điểm uốn hiệu suất quan trọng tại kích thước dữ liệu  $10^6$  phần tử, khi mà cấu hình "1 Master 3 Worker" bắt đầu cạnh tranh với "Master only". Điểm uốn này đánh dấu ngưỡng mà tại đó lợi ích của việc xử lý song song bắt đầu vượt qua chi phí overhead của việc phân tán.

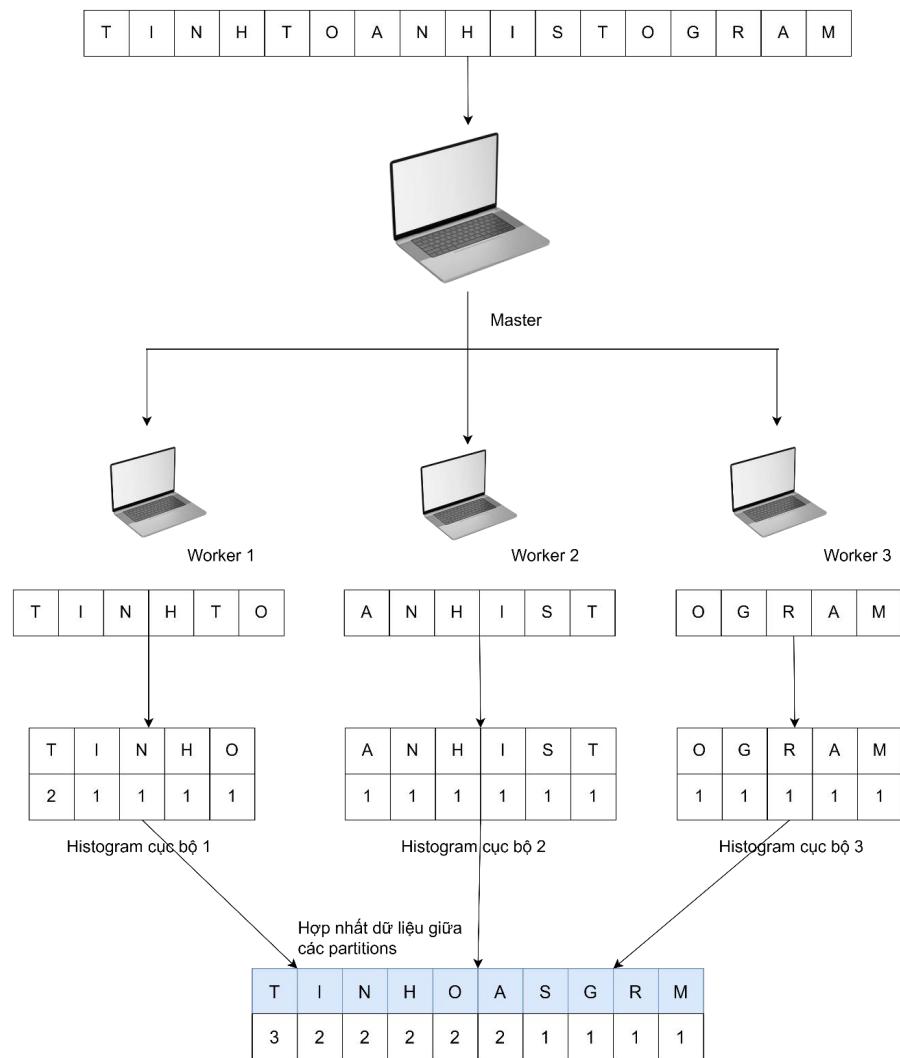
Với kích thước dữ liệu từ  $10^7$  phần tử trở lên, lợi thế của xử lý phân tán trở nên rõ rệt. Cấu hình "1 Master 3 Worker" cải thiện hiệu suất 26.9% so với "Master only" ở kích thước  $5 \times 10^7$  phần tử. Kết quả này phản ánh định luật Gustafson-Barsis [8] về khả năng mở rộng song song, theo đó hiệu quả của việc tăng số lượng đơn vị xử lý tỉ lệ thuận với kích thước của vấn đề.

Kết luận:

- Với dữ liệu nhỏ ( $< 10^6$  phần tử), xử lý tập trung hiệu quả hơn xử lý phân tán.
- Tồn tại một điểm uốn hiệu suất tại kích thước dữ liệu khoảng  $10^6$  phần tử.
- Số lượng worker tối ưu phụ thuộc vào kích thước dữ liệu và bản chất của tác vụ.
- Hiệu quả mở rộng không tuyến tính với số lượng worker, thể hiện quy luật hiệu suất cận biên giảm dần.

### 5.4.2. Bài toán Histogram

#### a) Cơ chế xử lý



**Hình 5.23.** Hình ảnh minh họa thuật toán Histogram phân tán.

Dựa trên **hình 5.23**, Histogram trên Spark sẽ được thực hiện theo các bước sau:

- Phân tán dữ liệu:
  - Dữ liệu đầu vào (ví dụ: 1 danh sách các giá trị số) được chia thành nhiều phân vùng (partitions) và lưu trữ dưới dạng RDD (Resilient Distributed Dataset).
  - Mỗi partition sẽ chứa một phần của dữ liệu cần sắp xếp.
- Sắp xếp cục bộ trong từng partition
  - Mỗi partition sẽ thực hiện tính toán histogram cục bộ bằng cách đếm số lần xuất hiện của từng giá trị trong phần dữ liệu của nó.

- Điều này có thể được thực hiện bằng cách sử dụng mapPartitions hoặc map để ánh xạ mỗi phần dữ liệu thành một từ điển chứa các giá trị và tần suất xuất hiện.
- Hợp nhất dữ liệu giữa các partitions
  - Sau khi tính toán histogram cục bộ, Spark tiến hành hợp nhất kết quả từ tất cả các partitions bằng cách sử dụng reduceByKey() hoặc aggregate().
  - Quá trình này đảm bảo rằng các giá trị từ tất cả partitions được cộng dồn lại để tạo ra histogram cuối cùng.

Ví dụ: Ta có một mảng các kí tự như sau:

Array = [T, I, N, H, T, O, A, N, H, I, S, T, O, G, R, A, M].

Hệ thống Spark bao gồm: 1 máy Master và 3 máy Worker.

Ban đầu dữ liệu được chia thành 3 partitions:

- Partition 1: [T, I, N, H, T, O]
- Partition 2: [A, N, H, I, S, T]
- Partition 3: [O, G, R, A, M]

Mỗi partition sẽ được xử lý bởi 1 worker khác nhau

- Các worker thực hiện tính toán histogram trên partition của mình:
  - Worker 1 tạo histogram cục bộ {‘T’: 2, ‘I’: 1, ‘N’: 1, ‘H’: 1, ‘O’: 1}
  - Worker 2 tạo histogram cục bộ {‘A’: 1, ‘N’: 1, ‘H’: 1, ‘I’: 1, ‘S’: 1, ‘T’: 1}
  - Worker 3 tạo histogram cục bộ {‘O’: 1, ‘G’: 1, ‘R’: 1, ‘A’: 1, ‘M’: 1}
- Master thực hiện hợp nhất dữ liệu từ tất cả các partition {‘T’: 3, ‘I’: 2, ‘N’: 2, ‘H’: 2, ‘O’: 2, ‘A’: 2, ‘S’: 1, ‘G’: 1, ‘R’: 1, ‘M’: 1, }

### b) Cài đặt bài toán Histogram trong Spark

- Cấu hình một Spark cluster gồm 1 master và nhiều worker sử dụng Docker:

```
version: '3'
services:
 spark-master:
 image: bitnami/spark:3.5.1
 container_name: spark-master
 environment:
 - SPARK_MODE=master
 ports:
 - "7077:7077"
 - "8080:8080"
```

```

spark-worker:
 image: bitnami/spark:3.5.1
 environment:
 - SPARK_MODE=worker
 - SPARK_MASTER_URL=spark://spark-master:7077
 depends_on:
 - spark-master

```

- Viết mã PySpark để thực hiện Histogram trên Spark RDD

Mã giả:

```

Đầu vào: Mảng A chưa được sắp xếp có kích thước n
Đầu ra: Histogram của mảng A

```

1. Khởi tạo Spark
2. Chuyển đổi mảng A thành RDD với P partitions
3. Thực hiện map trên từng partition để tính histogram cục bộ
4. Giảm các histogram cục bộ để có histogram toàn cục
5. Thu thập và **return** histogram\_A
6. **Stop Spark**

### c) Kết quả thực nghiệm

#### Môi trường thực nghiệm:

Cấu hình một Spark Cluster gồm 1 master và nhiều worker trên môi trường Docker với cấu hình cụ thể như sau:

- Docker Image: bitnami/spark:3.5.1
- Cấu hình Spark Master: 2 vCPU, 2GB RAM
- Cấu hình Spark Worker: 2 vCPU, 2GB RAM

Tiến hành thực nghiệm trên các trường hợp sau:

- 1 master 0 worker
- 1 master 1 worker
- 1 master 2 worker
- 1 master 3 worker

#### Dữ liệu:

Gồm các mảng số nguyên dương với các kích thước sau:

- 1000 phần tử
- 10000 phần tử
- 100000 phần tử
- 1000000 phần tử
- 10000000 phần tử

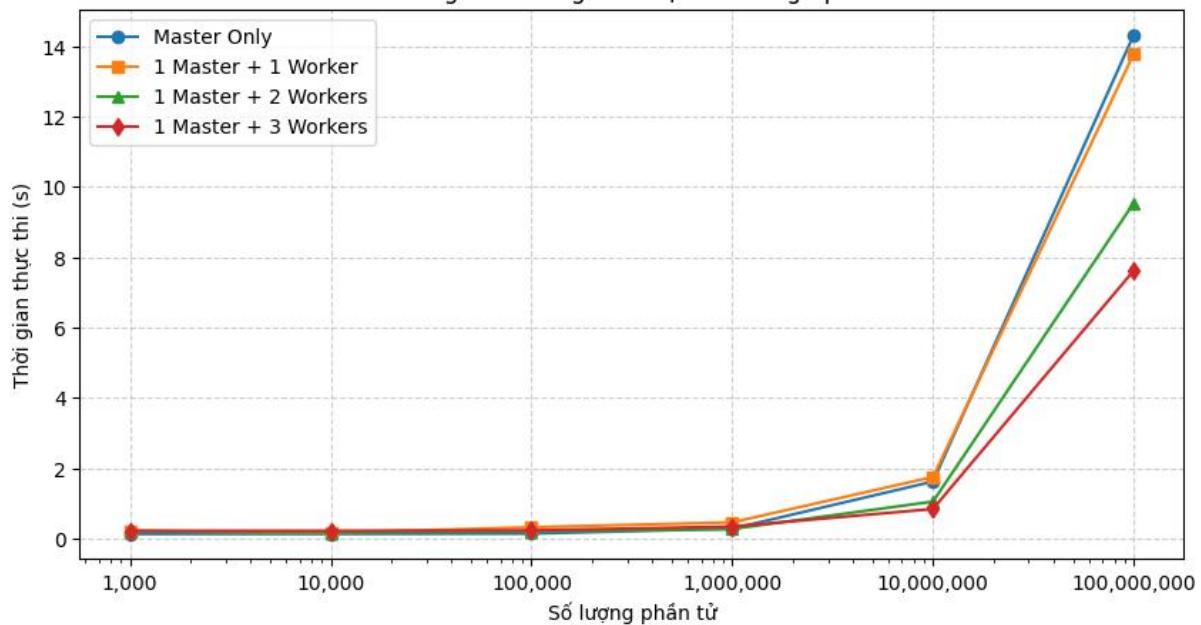
- 1000000000 phần tử

**Kết quả:**

Bảng 5.2. Kết quả thời gian chạy Histogram của các cấu hình Spark (Giây).

| Size      | Result  | Master only | 1 Master 1 Worker | 1 Master 2 Worker | 1 Master 3 Worker |
|-----------|---------|-------------|-------------------|-------------------|-------------------|
| 1000      | Correct | 0.13        | 0.21              | 0.21              | 0.22              |
| 10000     | Correct | 0.13        | 0.16              | 0.17              | 0.22              |
| 100000    | Correct | 0.14        | 0.32              | 0.2               | 0.23              |
| 1000000   | Correct | 0.28        | 0.46              | 0.27              | 0.34              |
| 10000000  | Correct | 1.62        | 1.75              | 1.05              | 0.84              |
| 100000000 | Correct | 14.32       | 13.78             | 9.54              | 7.62              |

Thời gian Histogram thực thi trong Spark



Hình 5.24. Biểu đồ thời gian chạy Histogram của các cấu hình Spark.

Từ bảng 5.2, ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như hình 5.24, ta có nhận xét như sau:

**Nhận xét:**

**Đối với tập dữ liệu nhỏ từ 1.000 - 1.000.000 phần tử:**

- Thời gian thực thi của các trường hợp không có sự khác biệt rõ ràng. Nguyên nhân có thể là do overhead của Spark, bao gồm quá trình khởi tạo job, phân chia dữ liệu, truyền dữ liệu giữa các node và tổng hợp kết quả. Khi kích thước dữ liệu còn nhỏ, overhead này chiếm phần lớn thời gian thực thi, làm lu mờ lợi ích của việc phân tán.

- Với lượng dữ liệu nhỏ, Spark có thể thực thi phần lớn công việc trên driver node mà không cần nhiều tài nguyên từ worker nodes, dẫn đến thời gian xử lý giàn như tương đương giữa các trường hợp.

#### **Đối với tập dữ liệu lớn từ 10.000.000 – 100.000.000 phần tử:**

- Thời gian thực thi bắt đầu có sự khác biệt đáng kể khi số lượng phần tử tăng lên, đặc biệt đối với 100.000.000 phần tử. Nguyên nhân chính là do khi dữ liệu lớn hơn, Spark bắt đầu tận dụng tốt hơn khả năng phân tán và xử lý song song trên nhiều worker nodes.
- Khi có nhiều worker hơn, dữ liệu histogram được chia nhỏ và xử lý đồng thời trên các node, giúp giảm tải cho từng node và cải thiện hiệu suất chung.
- Trường hợp "Master Only" có thời gian xử lý cao nhất, do toàn bộ quá trình tính toán chỉ diễn ra trên một node duy nhất, gây ra tắc nghẽn tài nguyên (CPU, bộ nhớ) và khiến thời gian xử lý tăng mạnh khi dữ liệu lớn.

#### **5.4.3. Bài toán Sum Reduce**

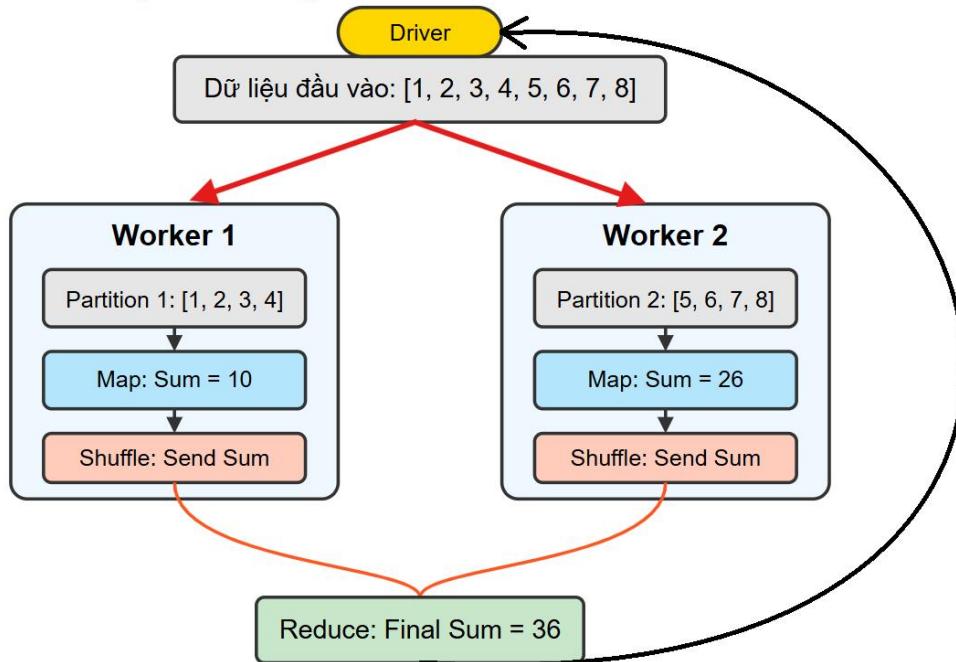
##### **a) Cơ chế xử lý**

- Khởi tạo SparkSession
  - Trước khi thực hiện bất kỳ tính toán nào với Spark, cần khởi tạo SparkSession – đây là đối tượng chính để tương tác với Spark.
  - SparkSession cung cấp một giao diện duy nhất để thao tác với RDD, DataFrame và thực hiện các phép toán phân tán.
- Tạo dữ liệu đầu vào và phân tán lên các Worker
  - Dữ liệu đầu vào là một danh sách gồm từ 1 nghìn đến 10 triệu số nguyên ngẫu nhiên, mỗi số nằm trong khoảng từ 1 đến 100. Thay vì xử lý toàn bộ danh sách trên một máy, Spark chuyển danh sách này thành RDD để phân tán lên các Worker bằng lệnh parallelize(), giúp phân tán dữ liệu lên các Worker trong cluster.
  - Khi gọi parallelize(data), Spark tự động chia nhỏ danh sách này thành nhiều "partition". Mỗi partition chứa một phần dữ liệu của danh sách ban đầu, các partition này sẽ được gửi đến các Worker, giúp tận dụng song song hóa để xử lý nhanh hơn.

- Nếu có 2 Worker, Spark có thể chia RDD thành 2 phần, mỗi Worker sẽ tính toán trên một phần của dữ liệu. Nếu có 4 Worker, dữ liệu sẽ được chia nhỏ hơn nữa.
- Quá trình này giúp tận dụng tính toán song song, vì mỗi Worker chỉ xử lý một phần nhỏ của danh sách thay vì để một máy duy nhất thực hiện, giúp giảm thời gian tính toán so với cách xử lý trên một máy/worker duy nhất.

Thực hiện phép Reduce để tính tổng: Trong Spark, phép Reduce được sử dụng để tổng hợp dữ liệu từ nhiều Worker về Master.

### Cách Apache Spark xử lý bài toán Reduce Sum với 2 worker.



**Hình 5.25.** Minh họa bài toán Sum Reduce với 2 worker.

Dựa vào **hình 5.25** ta thấy cơ chế xử lý bài toán Sum Reduce trong Spark như sau:

- Ánh xạ và gom nhóm dữ liệu tại các Worker
  - Mỗi Worker nhận được một partition của RDD và thực hiện tính tổng cục bộ trên phần dữ liệu đó.
  - Ví dụ, giả sử danh sách gốc có 8 phần tử [1, 2, 3, 4, 5, 6, 7, 8], Spark có thể chia thành 2 partition như sau:
    - Partition 1 (Worker 1): [1, 2, 3, 4]
    - Partition 2 (Worker 2): [5, 6, 7, 8]
  - Mỗi Worker thực hiện tính tổng trên partition của mình:
    - Worker 1:  $1 + 2 + 3 + 4 = 10$

- Worker 2:  $5 + 6 + 7 + 8 = 26$
- Giai đoạn Shuffle:
  - Sau khi hoàn thành tính toán cục bộ, các kết quả trung gian (10 và 26) cần được tổng hợp
  - Quá trình shuffle diễn ra - dữ liệu được truyền qua mạng giữa các worker
- Tổng hợp kết quả lên Master: Sau khi mỗi Worker hoàn thành tính toán, các kết quả cục bộ sẽ được gửi về Master. Master thực hiện phép Reduce lần cuối để tính tổng toàn bộ danh sách. Kết quả này được trả về Driver node:
 
$$10 (\text{tổng từ Worker 1}) + 26 (\text{tổng từ Worker 2}) = 36$$
- Cơ chế Tree Reduce trong Spark: Thay vì gửi tất cả dữ liệu về Master ngay lập tức, Spark tổ chức tính tổng theo mô hình cây (Tree Reduce). Ở mỗi bước, Spark gộp các kết quả lại từng cấp, giúp giảm số lượng dữ liệu phải gửi về Master. Điều này giúp giảm đáng kể chi phí truyền dữ liệu qua mạng và tối ưu tốc độ xử lý. Ví dụ:
  - Bước 1: Worker 1 tính tổng phần dữ liệu của nó (18), Worker 2 tính tổng phần của nó (26).
  - Bước 2: Master chỉ cần cộng  $18 + 26 = 44$ , thay vì phải nhận toàn bộ danh sách gốc.

### b) Cài đặt Sum Reduce trong Spark

Mã giả:

```

1. FUNCTION SumReduce(n)
2. spark ← InitializeSparkSession()
3. data ← GenerateRandomIntegers(1, 100, n)
4. rdd ← spark.parallelize(data) // Phân tán dữ liệu
5. sum ← rdd.reduce((a, b) → a + b) // Tính tổng phân tán
6. RETURN sum
7. END FUNCTION

```

Cách hoạt động của từng cấu hình:

Quá trình phân chia:

- Sử dụng hàm parallelize(data) để chuyển danh sách thành RDD.
- Spark tự động chia dữ liệu thành các partition, mỗi partition chứa một phần của danh sách ban đầu.
- Số partition phụ thuộc vào khối lượng dữ liệu và cấu hình (Spark sử dụng số partition mặc định nếu không chỉ định).

## Phân phối lên Worker:

- Ở chế độ cluster, Master phân phối các partition cho các Worker.
  - Với 2 Worker: Nếu có 16 partition, Master phân chia đều, mỗi worker nhận khoảng 8 partition (các log cho thấy 2 executor hoạt động, mỗi executor giải quyết khoảng 8 task, hoặc tổng thể 16 task được xử lý song song).
  - Với 1 Worker: Toàn bộ partition được gửi đến worker duy nhất, số task xử lý được giới hạn bởi số core của worker.
- Ở Local Mode, toàn bộ partition được xử lý trên node duy nhất theo số core có sẵn.

## Cấu hình 1 Master – 3 Worker:

### Cấu hình Spark:

```
spark = SparkSession.builder \
 .appName("SumReduce") \
 .master("spark://spark-master:7077") \
 .config("spark.executor.cores", "2") \
 .config("spark.executor.memory", "2g") \
 .config("spark.driver.memory", "2g") \
 .getOrCreate()
```

### Lệnh:

```
docker exec -it spark-master spark-submit --master spark://spark-
master:7077 /app/sum_reduce.py
```

### Cơ chế hoạt động:

- Hệ thống có 1 master và 3 worker, mỗi worker chạy 1 executor.
- Tổng số executor: 3.
- Mỗi executor được cấp 2 core, tổng số core sử dụng: 6 core.
- Task và dữ liệu được phân chia đều trên 3 executor để tăng hiệu suất xử lý.

### Chi tiết xử lý:

- Với 20 triệu, 10 triệu phần tử:
  - Log: "6/6 partition (6 TID)" (vì có 3 executor, mỗi executor xử lý 2 TID).
  - Công việc diễn ra trong 1 stage duy nhất.
  - Tổng cộng 6 task (TID) được khởi tạo và phân phối đều cho 3 executor.
- Với 1 triệu, 100.000, 10.000, 1.000 phần tử:
  - Log: "3/3 partition (3 TID)".
  - Spark tối ưu, có thể chỉ sử dụng 1 hoặc 2 executor, vì dữ liệu nhỏ không đủ để tận dụng hết tài nguyên.

- Xử lý diễn ra trong 1 stage duy nhất.

### Cấu hình 1 Master – 2 Worker:

Cấu hình Spark:

```
spark = SparkSession.builder \
 .appName("SumReduce") \
 .master("spark://spark-master:7077") \
 .config("spark.executor.cores", "2") \
 .config("spark.executor.memory", "2g") \
 .config("spark.driver.memory", "2g") \
 .getOrCreate()
```

Lệnh chạy:

```
docker exec -it spark-master spark-submit --master spark://spark-
master:7077 /app/sum_reduce.py
```

Cơ chế:

- Mỗi worker chạy 1 executor, chia sẻ dữ liệu và task.
- Executors: 2 (mỗi worker 1 executor)
- Cores sử dụng: tổng số lõi CPU của cả 2 worker, mỗi worker có 2 core, tổng 4 core.
- Task: chia thành nhiều partition, phân bổ cho 2 executor
- Phân phối: Master phối hợp công việc, chia đều cho 2 worker

Với 20 và 10 triệu phần tử:

- Log: “4/4 partition (4 TID)”
- Có 2 executor, mỗi executor xử lý 2 TID (dữ liệu lớn nên tận dụng được cả 2 worker)
- Toàn bộ công việc diễn ra trong 1 stage.

Với 1 triệu, 100.000, 10.000, 1.000 phần tử:

- Log: “2/2 partition || task (2 TID)”
- Chỉ có 1 executor thực hiện xử lý (Spark tối ưu số task khi dữ liệu quá nhỏ).
- Diễn ra trong 1 stage.

### Cấu hình 1 Master – 1 Worker:

Cấu hình Spark:

```
spark = SparkSession.builder \
 .appName("SumReduce") \
 .master("spark://spark-master:7077") \
 .config("spark.executor.cores", "2") \
 .config("spark.executor.memory", "2g") \
 .config("spark.driver.memory", "2g") \
```

```
.getOrCreate()
```

Lệnh chạy:

```
docker exec -it spark-master spark-submit --master spark://spark-master:7077 /app/sum_reduce.py
```

Cơ chế:

- Mỗi khi tạo job, Master yêu cầu worker chạy executor.
- Mặc định, Spark tạo đúng 1 executor trên worker (do chỉ có 1 worker), với toàn bộ core khả dụng trên máy worker.
- Cores sử dụng: toàn bộ lõi CPU của worker (2 core cho 1 worker)
- Task: Tùy thuộc partition mặc định (thường bằng số core hoặc số partition Spark quy định)
- Phân phối: Master gửi task sang worker, worker chạy và trả kết quả về driver

Với 20 triệu, 10 triệu, 1 triệu, 100.000, 10.000, 1.000 phần tử:

- Log: “2/2 partition (2 TID)”
- Toàn bộ xử lý được thực hiện bởi 1 executor trên worker.
- 1 stage duy nhất.

### Local Mode:

Cấu hình Spark:

```
spark = SparkSession.builder \
 .appName("SumReduce") \
 .master("local[2]") \
 .config("spark.executor.cores", "2") \
 .config("spark.executor.memory", "2g") \
 .config("spark.driver.memory", "2g") \
 .getOrCreate()
```

Lệnh chạy:

```
docker exec -it spark-master spark-submit --master local[2]
/app/sum_reduce.py
```

Cơ chế:

- Spark chỉ chạy trên container `spark-master`, không cần các worker.
- Spark sẽ dùng 2 core của máy cục bộ (local[2]) để tạo các task.
- Tùy vào số core của container, toàn bộ các task vẫn do một tiến trình (executor driver) xử lý
- Cores sử dụng: phụ thuộc vào số lõi CPU của container `spark-master` (khi dùng local[2]), nghĩa là sử dụng 2 core
- Task: tạo ra theo số partition mặc định (thường bằng số core)

- Phân phối dữ liệu: không qua mạng, tất cả xử lý trên cùng container

Với 10 triệu, 1 triệu, 100.000, 10.000, 1.000 phần tử:

- Log: “2/2 partition (2 TID)”
- Toàn bộ task được xử lý bởi executor driver trên node local.
- 1 stage duy nhất.

### c) Kết quả thực nghiệm

#### Môi trường thực nghiệm:

Cấu hình một Spark Cluster gồm 1 master và nhiều worker trên môi trường Docker với cấu hình cụ thể như sau:

- Docker Image: bitnami/spark:3.5.1
- Cấu hình Spark Master: 2 vCPU, 2GB RAM
- Cấu hình Spark Worker: 2 vCPU, 2GB RAM

Tiến hành thực nghiệm trên các trường hợp sau:

- 1 master 0 worker
- 1 master 1 worker
- 1 master 2 worker
- 1 master 3 worker

#### Dữ liệu

Các mảng một chiều số nguyên với các kích thước sau:

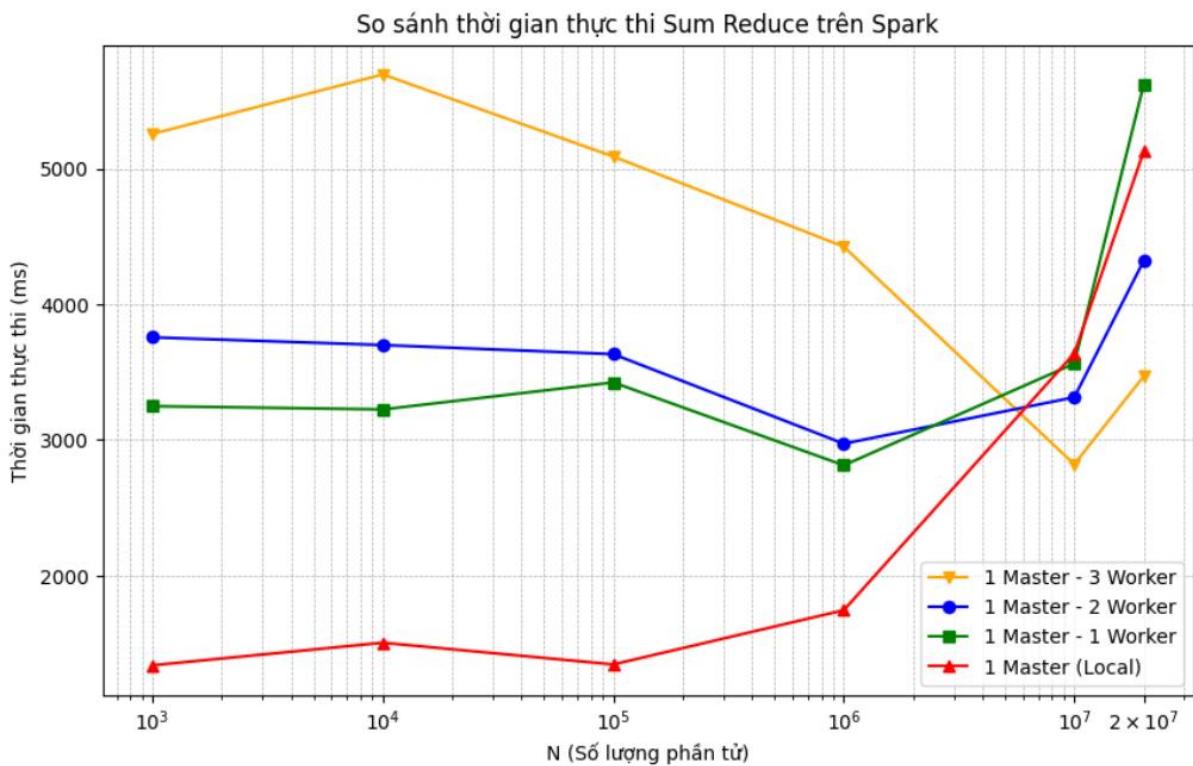
- 1000 phần tử
- 10000 phần tử
- 100000 phần tử
- 1000000 phần tử
- 10000000 phần tử
- 20000000 phần tử

#### Kết quả:

*Bảng 5.3. Kết quả thời gian chạy Sum reduce của các cấu hình Spark (Giây).*

| Kích thước<br>mảng (N) | 3 Workers (s) | 2 Workers (s) | 1 Worker (s) | Master/Local<br>(s) |
|------------------------|---------------|---------------|--------------|---------------------|
| 1,000                  | 5.255         | 3.755         | 3.248        | <b>1.338</b>        |
| 10,000                 | 5.694         | 3.699         | 3.224        | <b>1.504</b>        |

|            |              |       |       |              |
|------------|--------------|-------|-------|--------------|
| 100,000    | 5.088        | 3.632 | 3.423 | <b>1.344</b> |
| 1,000,000  | 4.424        | 2.971 | 2.812 | <b>1.744</b> |
| 10,000,000 | <b>2.819</b> | 3.313 | 3.562 | 3.636        |
| 20,000,000 | <b>3.468</b> | 4.319 | 5.617 | 5.125        |



**Hình 5.26.** Biểu đồ thời gian chạy Sum Reduce của các cấu hình Spark.

Từ bảng 5.3, ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như **hình 5.26**, ta có nhận xét như sau:

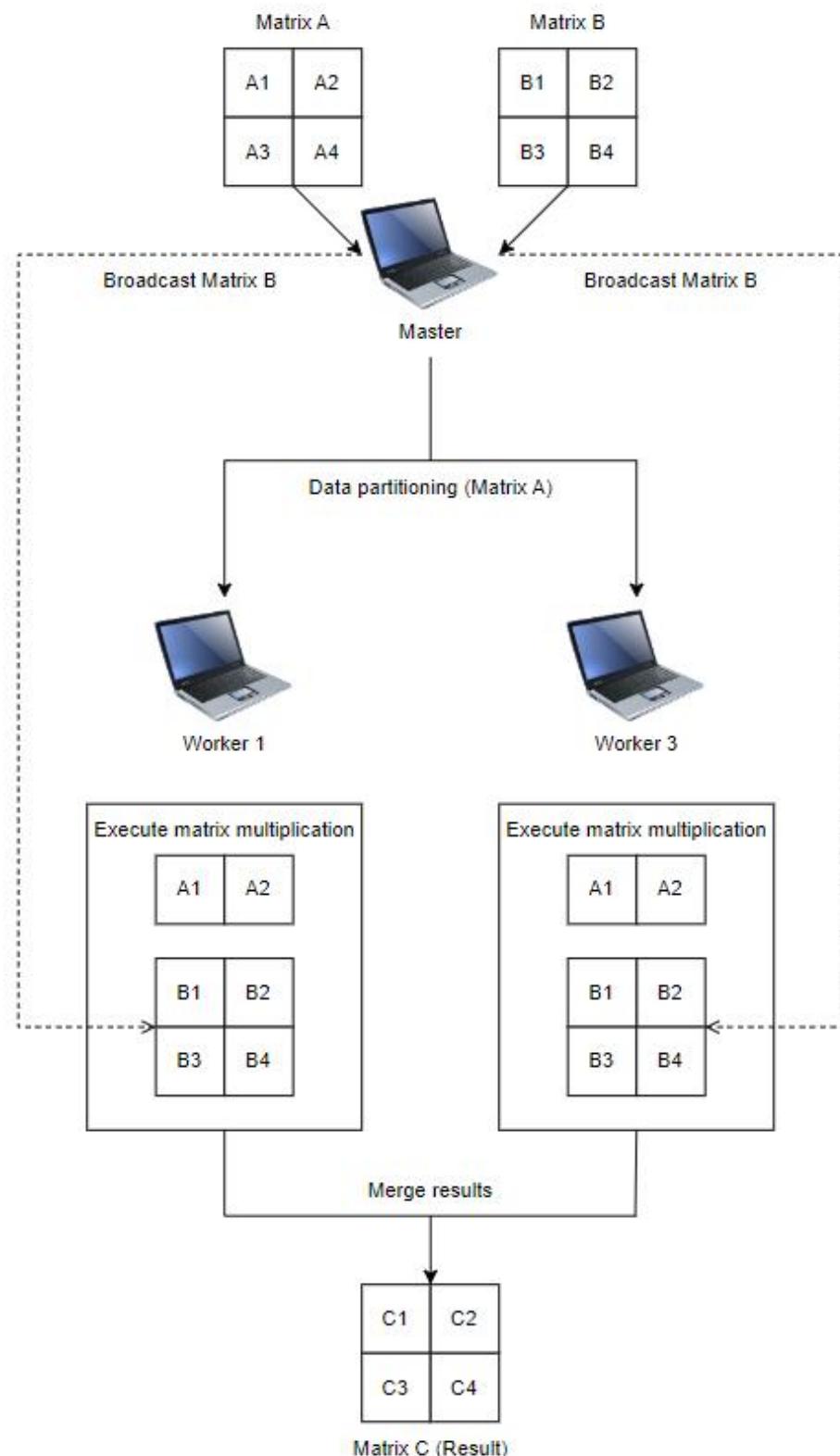
- Giai đoạn 1.000 – 1.000.000 phân tử
  - Local mode nhanh nhất, do không có chi phí truyền dữ liệu giữa master và worker. Khi dữ liệu nhỏ, overhead này lớn hơn lợi ích của phân tán, dẫn đến local chạy nhanh hơn. Với 1.000 phân tử, Local Mode chỉ mất 1.338s, nhanh hơn 2.4 lần so với 1 Worker (3.248s) và gần 4 lần so với 3 Worker (5.255s). Khi tăng lên 1.000.000 phân tử, Local Mode vẫn nhanh nhất (1.744s) so với 1 Worker (2.812s) và 2 Worker (2.971s).
  - 1 Worker và 2 Worker có thời gian gần nhau, nhưng 1 Worker vẫn nhanh hơn một chút do dữ liệu nhỏ và gây chi phí phân tán cao đối với 2 worker. Việc chia nhỏ dữ liệu cho nhiều Worker khi dữ liệu quá nhỏ lại làm tăng

overhead, khiến 2 Worker không thực sự có lợi thế rõ ràng. Với 1.000.000 phần tử: 1 Worker (2.812s) nhanh hơn 2 Worker (2.971s).

- Giai đoạn 10.000.000 – 20.000.000 phần tử
  - Local mode bắt đầu chậm lại nhanh chóng, do không thể tận dụng nhiều core như cluster. Với 10.000.000 phần tử Local mất 3.636s, lâu hơn 3 Worker (2.819s), 20.000.000 phần tử Local mất 5.125s, lâu hơn 3 Worker (3.468s).
  - 1 Worker vẫn chậm hơn local do không thực sự được phân tán và bị giới hạn tài nguyên (mỗi worker cấu hình 2 core) và do chi phí truyền tải giữa master và worker. Do 1 Worker vẫn phải giao tiếp với Master, dẫn đến chi phí truyền tải dữ liệu giữa Master và Worker. Với 20.000.000 phần tử, 1 Worker mất 5.617s, lâu hơn cả Local Mode (5.125s).
  - 2 Worker và 3 Worker vẫn là nhanh nhất khi dữ liệu lớn, do việc chia tải dữ liệu tốt hơn, tận dụng được lợi ích của phân tán. Vì dữ liệu lớn nên càng nhiều Worker thì sẽ càng nhanh nên 3 Worker sẽ nhanh hơn 2 Worker. Với 10.000.000 phần tử, 3 Worker (2.819s) nhanh hơn 2 Worker (3.313s) và 1 Worker (3.562s). Với 20.000.000 phần tử, 3 Worker (3.468s) vẫn nhanh hơn 2 Worker (4.319s) và 1 Worker (5.617s).
- Tổng kết:
  - Local Mode nhanh nhất khi dữ liệu nhỏ ( $\leq 1.000.000$  phần tử) do không có overhead truyền dữ liệu giữa các Worker.
  - 1 Worker nhanh hơn 2 Worker khi dữ liệu nhỏ do việc chia task khi dữ liệu ít gây thêm overhead.
  - Khi dữ liệu lớn ( $\geq 10.000.000$  phần tử), 3 Worker và 2 Worker nhanh nhất vì có thể phân tán workload hiệu quả hơn.
  - 1 Worker bị giới hạn tài nguyên, đôi khi còn chậm hơn Local Mode khi dữ liệu lớn.

#### 5.4.4. Bài toán nhân ma trận

##### a) Cơ chế xử lý



**Hình 5.27.** Hình ảnh minh họa thuật toán nhân ma trận phân tán.

Ý tưởng nhân ma trận như **hình 5.27** phân tán trên Spark được thực hiện qua các bước:

- Phân tán dữ liệu:

- Ma trận đầu vào A được chia thành các hàng (row-wise) và lưu trữ dưới dạng RDD.
- Ma trận B được truyền tới tất cả các worker thông qua broadcast để tránh overhead truyền dữ liệu lặp lại.
- Tính toán song song:
  - Mỗi worker nhận một tập hợp hàng của ma trận A và thực hiện nhân từng hàng với toàn bộ ma trận B.
  - Sử dụng mapPartitions để tối ưu hiệu suất.
- Tổng hợp kết quả:
  - Sau khi hoàn thành phép nhân, Spark thu thập kết quả từ tất cả các worker và kết hợp lại để tạo thành ma trận kết quả.

Ví dụ: Ta có 2 ma trận đầu vào như sau:

$$\begin{aligned} A &= [[1, 2], \\ &\quad [3, 4], \\ &\quad [5, 6]] \end{aligned}$$

$$\begin{aligned} B &= [[7, 8], \\ &\quad [9, 10]] \end{aligned}$$

Cụm phân tán gồm 1 máy master và 2 máy worker. Quá trình nhân ma trận phân tán diễn ra như sau:

- Phân chia dữ liệu:
  - Worker 1: Nhận hàng [1, 2] và [3, 4]
  - Worker 2: Nhận hàng [5, 6]
- Nhân từng hàng với ma trận B:
  - Worker 1 thực hiện:
    - $[1, 2] * B = [1 \times 7 + 2 \times 9, 1 \times 8 + 2 \times 10] = [25, 28]$
    - $[3, 4] * B = [3 \times 7 + 4 \times 9, 3 \times 8 + 4 \times 10] = [57, 64]$
  - Worker 2 thực hiện:
    - $[5, 6] * B = [5 \times 7 + 6 \times 9, 5 \times 8 + 6 \times 10] = [89, 100]$
- Tổng hợp kết quả:

$$\begin{aligned} C &= [[25, 28], \\ &\quad [57, 64], \end{aligned}$$

[89, 100]]

### b) Cài đặt nhân ma trận phân tán trên Spark

- Cấu hình một Spark cluster gồm 1 master và nhiều worker sử dụng Docker:

```
version: '3.7'
services:
 spark-master:
 image: bitnami/spark:3.5.1
 command: bin/spark-class org.apache.spark.deploy.master.Master
 ports:
 - 8080:8080
 - 7077:7077
 environment:
 - SPARK_MODE=master

 spark-worker:
 image: bitnami/spark:3.5.1
 command: bin/spark-class org.apache.spark.deploy.worker.Worker
 spark://spark-master:7077
 depends_on:
 - spark-master
 environment:
 - SPARK_MODE=worker
 - SPARK_MASTER_URL=spark://spark-master:7077
```

- Viết mã PySpark để thực hiện merge sort trên Spark RDD5.3.5. Bài toán sắp xếp (Merge Sort)

Mã giả:

```
Input: Ma trận A (kích thước m×n), ma trận B (kích thước n×p)
Output: Ma trận kết quả C (kích thước m×p)
```

- Khởi tạo Spark
- Phân chia ma trận A thành P phần vùng dưới dạng RDD
- Broadcast ma trận B đến tất cả các Worker
- Áp dụng mapPartitions để nhân từng hàng của A với B
- Thu thập kết quả và tập hợp thành ma trận C
- Dừng Spark

### c) Kết quả thực nghiệm

#### Môi trường thực nghiệm:

Cấu hình một Spark Cluster gồm 1 master và nhiều worker trên môi trường Docker với cấu hình cụ thể như sau:

- Docker Image: bitnami/spark:3.5.1
- Cấu hình Spark Master: 2 vCPU, 2GB RAM
- Cấu hình Spark Worker: 2 vCPU, 2GB RAM

Tiến hành thực nghiệm trên các trường hợp sau:

- 1 master 0 worker
- 1 master 1 worker
- 1 master 2 worker
- 1 master 3 worker

#### Dữ liệu:

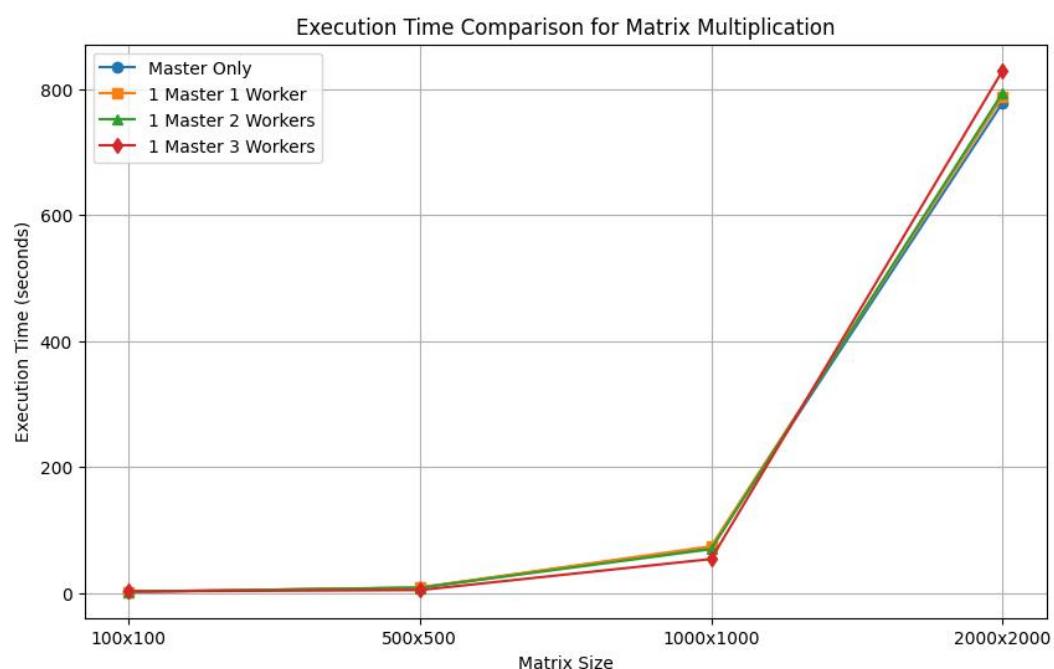
Gồm các ma trận vuông số nguyên A và B có kích thước như sau

- 100x100
- 500x500
- 1000x1000
- 2000x2000

#### Kết quả:

*Bảng 5.4. Kết quả thời gian chạy Merge Sort của các cấu hình Spark (Giây).*

| Matrix Size | Result  | Master Only     | 1 master 1 worker | 1 master 2 worker | 1 master 3 worker |
|-------------|---------|-----------------|-------------------|-------------------|-------------------|
| 100x100     | Correct | <b>1.8682</b>   | 1.9400            | 1.9772            | 2.5697            |
| 500x500     | Correct | 8.4562          | 8.6111            | 8.1621            | <b>4.5617</b>     |
| 1000x1000   | Correct | 72.5301         | 74.2507           | 69.5383           | <b>53.9535</b>    |
| 2000x2000   | Correct | <b>778.9456</b> | 788.2258          | 794.5794          | 829.7778          |



*Hình 5.28. Biểu đồ thời gian chạy nhân ma trận của các cấu hình Spark.*

Từ **bảng 5.4**, ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như **hình 5.28**, ta có nhận xét như sau:

- Hiệu năng với ma trận nhỏ ( $100 \times 100$ ):
  - Khi chạy trên Master Only, thời gian là 1.8682s.
  - Khi thêm 1 hoặc 2 Worker, thời gian tăng nhẹ (1.9400s - 1.9772s), cho thấy lợi ích phân tán không rõ ràng với kích thước nhỏ.
  - Khi có 3 Worker, thời gian tăng đáng kể (2.5697s), có thể do overhead trong việc chia nhỏ và thu thập kết quả.
- Hiệu năng với ma trận trung bình ( $500 \times 500, 1000 \times 1000$ ):
  - Với  $500 \times 500$ , hiệu năng cải thiện rõ ràng khi tăng từ 1 lên 2 Worker (8.1621s so với 8.6111s).
  - Khi tăng lên 3 Worker, thời gian giảm đáng kể (4.5617s), cho thấy sự phân tán bắt đầu có lợi.
  - Với  $1000 \times 1000$ , xu hướng tương tự: thêm Worker giúp giảm thời gian tính toán (69.5383s với 2 Worker so với 74.2507s với 1 Worker).
- Hiệu năng với ma trận lớn ( $2000 \times 2000$ ):
  - Khi tăng số Worker từ 1 đến 3, thời gian không giảm mà còn tăng (788.2258s lên 829.7778s).
  - Điều này do tiêu tốn chi phí để broadcast dữ liệu lớn đến các worker
  - Spark có overhead lớn khi làm việc với dữ liệu lớn, đặc biệt nếu không tối ưu hóa cách phân chia dữ liệu.
- Tóm lại:
  - Spark có lợi thế khi làm việc với ma trận vừa phải ( $500 \times 500$  đến  $1000 \times 1000$ ), đặc biệt khi tăng số Worker.
  - Với ma trận nhỏ, phân tán không hiệu quả do chi phí khởi tạo và truyền dữ liệu.
  - Với ma trận lớn, overhead giao tiếp giữa các Worker có thể làm giảm hiệu suất, cần tối ưu hóa chiến lược phân tán dữ liệu.

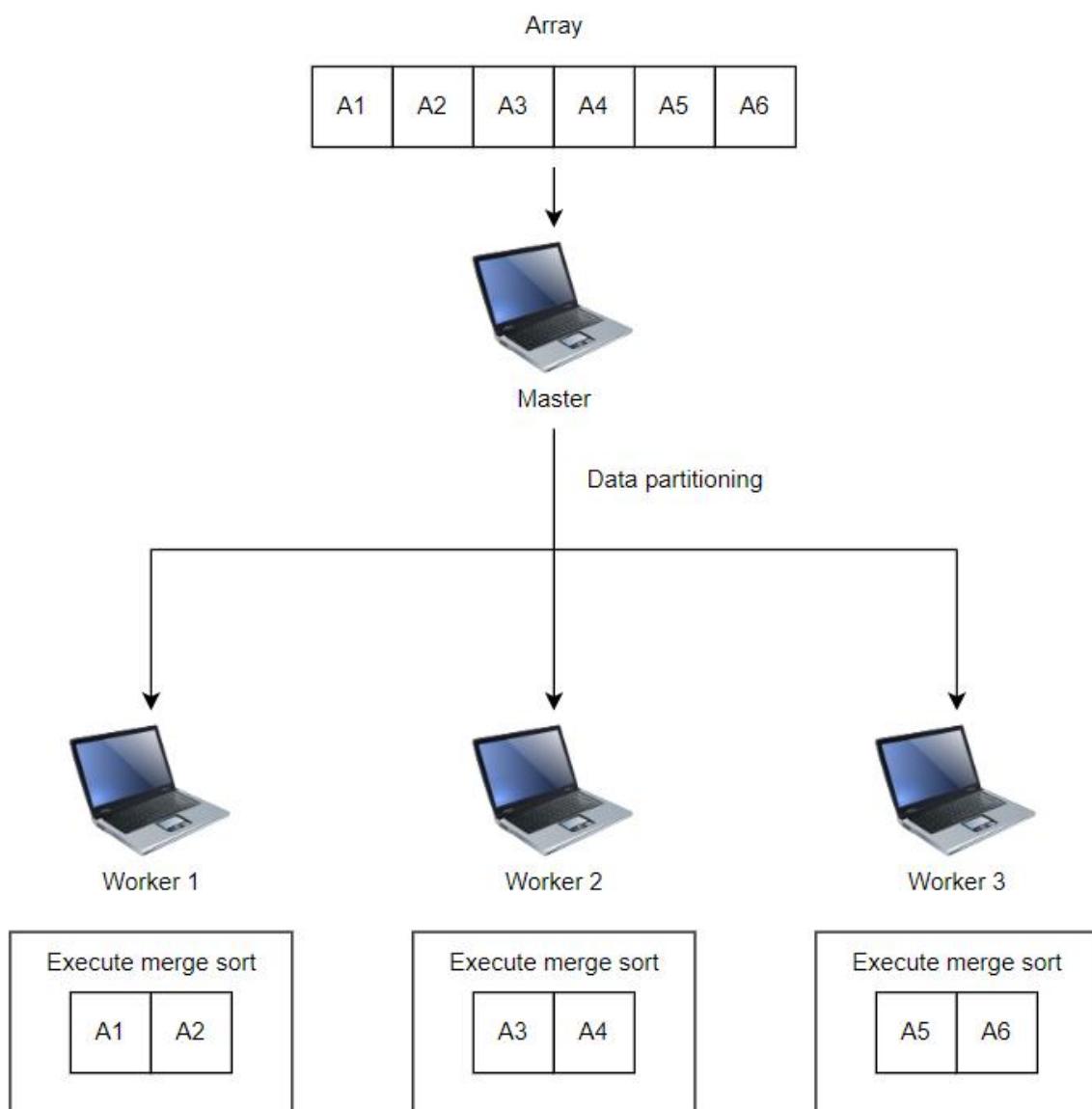
#### **5.4.5. Bài toán sắp xếp (Merge Sort)**

##### **a) Cơ chế xử lý**

Merge sort trên Spark sẽ được thực hiện theo các bước sau:

- Phân tán dữ liệu:

- Mảng đầu vào được chia thành nhiều phân vùng (partitions) và lưu trữ dưới dạng RDD (Resilient Distributed Dataset).
- Mỗi partition sẽ chứa một phần của dữ liệu cần sắp xếp.
- Sắp xếp cục bộ trong từng partition
  - Mỗi partition thực hiện Merge Sort cục bộ, tức là áp dụng thuật toán sắp xếp trên phần dữ liệu của mình.
  - Quá trình này có thể thực hiện bằng cách gọi mapPartitions để áp dụng Merge Sort trên từng partition.
- Hợp nhất dữ liệu giữa các partitions
  - Sau khi các partitions được sắp xếp cục bộ, Spark tiến hành hợp nhất dữ liệu giữa các partitions.



**Hình 5.29.** Hình ảnh minh họa thuật toán Merge Sort phân tán.

Dựa theo ý tưởng Merge sort trong **hình 5.29**, giả sử ta có một mảng các số nguyên dương gồm 6 phần tử như sau: Array = [8, 4, 5, 1, 7, 3]. Cụm phân tán gồm 1 máy master và 3 máy worker. Ta sẽ thực hiện merge sort phân tán như sau:

- Chia mảng ban đầu thành 3 partition như sau:
  - Partition 1: [8, 4]
  - Partition 2: [5, 1]
  - Partition 3: [7, 3]
- Các worker thực hiện sắp xếp các partition:
  - Worker 1: sắp xếp [8, 4] → [4, 8]
  - Worker 2: sắp xếp [5, 1] → [1, 5]
  - Worker 3: Sắp xếp [7, 3] → [3, 7]
- Master thực hiện hợp nhất các partition [4, 8], [1, 5], [3, 7] → [1, 3, 4, 5, 7, 8]

### b) Cài đặt Merge Sort trong Spark

- Cấu hình một Spark cluster gồm 1 master và nhiều worker sử dụng Docker:

```
version: '3'
services:
 spark-master:
 image: bitnami/spark:3.5.1
 container_name: spark-master
 environment:
 - SPARK_MODE=master
 ports:
 - "7077:7077"
 - "8080:8080"

 spark-worker:
 image: bitnami/spark:3.5.1
 environment:
 - SPARK_MODE=worker
 - SPARK_MASTER_URL=spark://spark-master:7077
 depends_on:
 - spark-master
```

- Viết mã PySpark để thực hiện merge sort trên Spark RDD

Mã giả:

|                                                                  |
|------------------------------------------------------------------|
| Input: Mảng A với kích thước N<br>Output: Mảng A đã được sắp xếp |
|------------------------------------------------------------------|

1. Khởi tạo Spark
2. Chuyển mảng A thành RDD và phân tán
3. Áp dụng merge sort cho các phần dữ liệu được phân tán
4. Gộp các kết quả thành một mảng đã sắp xếp

- |                     |
|---------------------|
| 5. Thu thập kết quả |
| 6. Dừng Spark       |

### c) Kết quả thực nghiệm

#### Môi trường thực nghiệm:

Cấu hình một Spark Cluster gồm 1 master và nhiều worker trên môi trường Docker với cấu hình cụ thể như sau:

- Docker Image: bitnami/spark:3.5.1
- Cấu hình Spark Master: 2 vCPU, 2GB RAM
- Cấu hình Spark Worker: 2 vCPU, 2GB RAM

Tiến hành thực nghiệm trên các trường hợp sau:

- 1 master 0 worker
- 1 master 1 worker
- 1 master 2 worker
- 1 master 3 worker

#### Dữ liệu:

Gồm các mảng số nguyên dương với các kích thước sau:

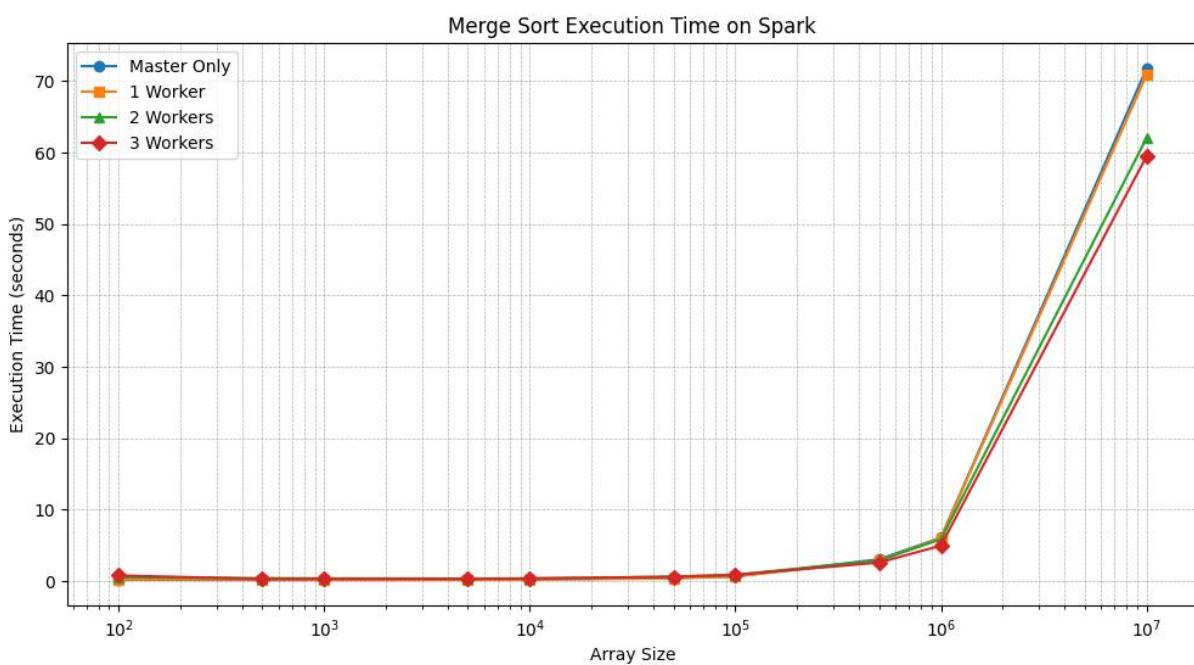
- 100 phần tử
- 500 phần tử
- 1000 phần tử
- 5000 phần tử
- 10000 phần tử
- 50000 phần tử
- 100000 phần tử
- 500000 phần tử
- 1000000 phần tử
- 10000000 phần tử

#### Kết quả:

*Bảng 5.5. Kết quả thời gian chạy Merge Sort của các cấu hình Spark (Giây).*

| Array Size | Result  | Master only | 1 Master 1 Worker | 1 Master 2 Worker | 1 Master 3 Worker |
|------------|---------|-------------|-------------------|-------------------|-------------------|
| 100        | Correct | 0.1932      | 0.2299            | 0.5433            | 0.7814            |
| 500        | Correct | 0.1508      | 0.2162            | 0.4049            | 0.2878            |

|          |         |         |         |         |         |
|----------|---------|---------|---------|---------|---------|
| 1000     | Correct | 0.1584  | 0.1701  | 0.3713  | 0.2967  |
| 5000     | Correct | 0.1844  | 0.1822  | 0.3373  | 0.2996  |
| 10000    | Correct | 0.2174  | 0.1992  | 0.3481  | 0.3609  |
| 50000    | Correct | 0.4302  | 0.3806  | 0.6131  | 0.5905  |
| 100000   | Correct | 0.6555  | 0.6434  | 0.8817  | 0.8615  |
| 500000   | Correct | 3.0382  | 2.8658  | 2.8851  | 2.5922  |
| 1000000  | Correct | 6.0849  | 6.0353  | 5.8597  | 4.9624  |
| 10000000 | Correct | 71.7671 | 70.8961 | 62.0422 | 59.4969 |



**Hình 5.30.** Biểu đồ thời gian chạy Merge Sort của các cấu hình Spark.

Từ **bảng 5.5**, ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như **hình 5.30**, ta có nhận xét như sau:

Hiệu suất trên tập dữ liệu nhỏ ( $< 100,000$  phần tử):

- Khi kích thước mảng nhỏ ( $\leq 100,000$  phần tử), cấu hình "Master only" có hiệu suất tốt nhất hoặc gần tương đương với các cấu hình phân tán.
- Các cấu hình có nhiều worker không mang lại lợi ích đáng kể, thậm chí có thể có hiệu suất kém hơn do overhead của việc phân chia và tổng hợp dữ liệu.
- Trường hợp "1 Master 2 Worker" và "1 Master 3 Worker" có thời gian xử lý cao hơn so với "Master only", minh họa rõ ràng tác động của chi phí phân tán và tổng hợp dữ liệu trong Spark.

Hiệu suất trên tập dữ liệu lớn ( $\geq 500,000$  phần tử):

- Khi kích thước mảng tăng lên, lợi thế của việc phân tán trở nên rõ ràng. Ở kích thước 500,000 phần tử, "1 Master 3 Worker" có hiệu suất tốt hơn "Master only" (~14.7% nhanh hơn).
- Ở kích thước 10,000,000 phần tử, "1 Master 3 Worker" nhanh hơn "Master only" khoảng 17.1%, cho thấy lợi ích thực sự của việc xử lý phân tán khi khối lượng dữ liệu đủ lớn để bù đắp chi phí overhead.

Điểm uốn hiệu suất:

- Ngưỡng khoảng 500,000 - 1,000,000 phần tử đánh dấu sự chuyển đổi, khi các cấu hình phân tán dần trở nên hiệu quả hơn "Master only".
- Điều này phù hợp với mô hình hiệu suất của Spark, khi overhead phân tán ban đầu lớn nhưng dần trở nên ít ảnh hưởng hơn khi dữ liệu đủ lớn.

Tóm tắt:

- Dữ liệu nhỏ (<100,000 phần tử): Xử lý tập trung (Master only) là tối ưu nhất.
- Dữ liệu trung bình (500,000 - 1,000,000 phần tử): Hiệu suất giữa các cấu hình bắt đầu cân bằng, và xử lý phân tán bắt đầu thể hiện lợi thế.
- Dữ liệu lớn ( $\geq 10,000,000$  phần tử): Cấu hình nhiều worker có hiệu suất tốt hơn rõ rệt so với "Master only", phản ánh đúng mô hình mở rộng phân tán của Spark.

#### **5.4.6. Bài toán Inverted Indexing**

##### **a) Cơ chế xử lý**

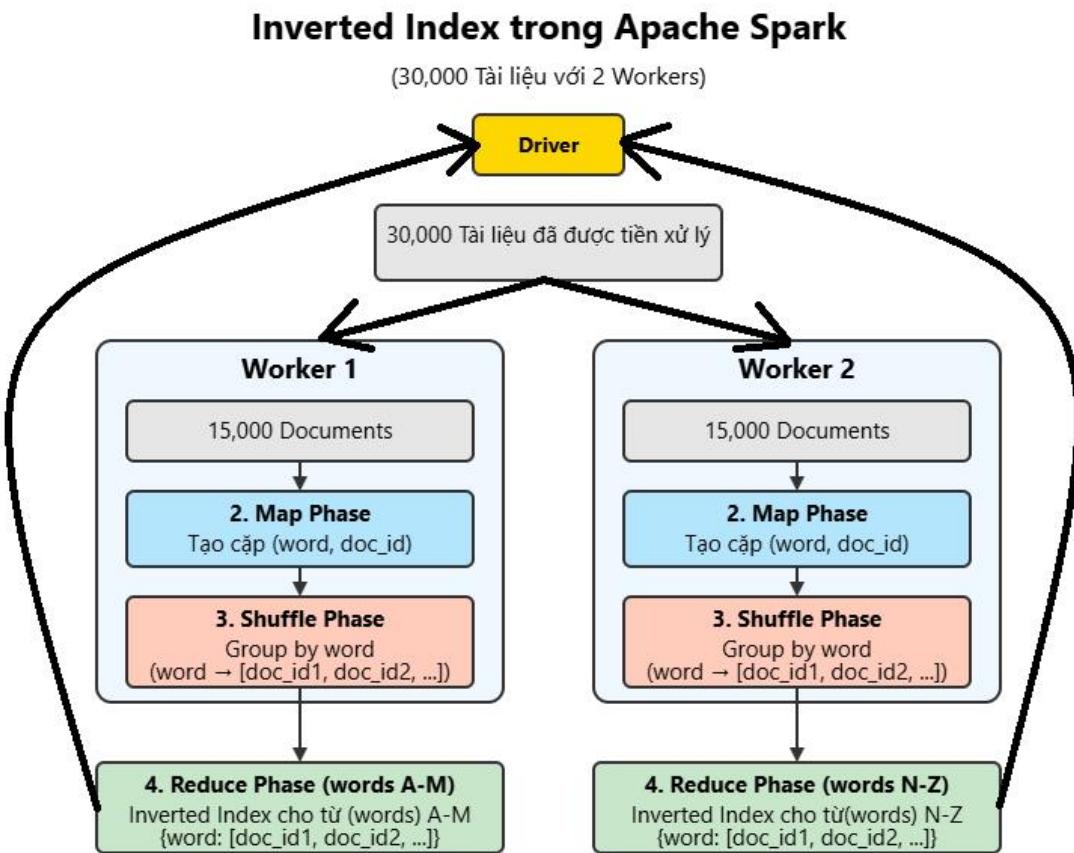
Phân tán dữ liệu lên các Worker:

- Dữ liệu sau đầu vào được chia nhỏ và gửi đến các Worker để xử lý song song.
- Spark sử dụng một cơ chế gọi là RDD (Resilient Distributed Dataset) để quản lý dữ liệu trên nhiều Worker.
- Khi dữ liệu được chuyển thành RDD, Spark tự động phân chia nó thành nhiều phần nhỏ (partition).
- Mỗi Worker sẽ nhận một số partition để xử lý, thay vì để một máy tính (worker) duy nhất làm toàn bộ công việc.

Quá trình xây dựng Inverted Index trên Spark bao gồm các bước chính sau:

- Khởi tạo và phân tán dữ liệu
- Tạo cặp Từ-Tài liệu (Map Phase)
- Shuffle và Group (Shuffle Phase)

- Xây dựng chỉ mục cuối cùng (Reduce Phase)
- Thu thập kết quả (Collection Phase)



*Hình 5.31. Minh họa bài toán Inverted Indexing với 2 worker.*

Chi tiết từng bước cho **hình 5.31**:

Bước 1: Khởi tạo và phân tán dữ liệu

Khi khởi tạo RDD (Resilient Distributed Dataset) từ tập tài liệu đầu vào:

- Spark phân chia dữ liệu thành nhiều partition (phân vùng)
- Số lượng partition mặc định thường được xác định dựa trên số lõi xử lý sẵn có trong cụm máy
- Mỗi partition được gửi đến một Worker để xử lý độc lập
- Spark duy trì thông tin về các partition và cách chúng được phân phối trên các Worker

Chi tiết kỹ thuật:

- Khi không chỉ định số partition, Spark sẽ sử dụng giá trị spark.default.parallelism hoặc số lõi sẵn có
- Mỗi partition có thể chứa nhiều tài liệu, nhưng một tài liệu chỉ thuộc về một partition duy nhất

- Việc phân chia dữ liệu thành các partition giúp tải công việc được cân bằng giữa các Worker

#### Bước 2: Tạo cặp Từ-Tài liệu (Map Phase)

Trong giai đoạn này:

- Mỗi Worker xử lý các tài liệu trong partition của mình
- Các tài liệu được chuẩn hóa và tách thành các từ khóa (tokenization)
- Với mỗi từ khóa, Worker tạo ra một cặp ((word, doc\_id), 1)
  - o word: từ khóa được trích xuất
  - o doc\_id: ID của tài liệu chứa từ khóa đó
  - o 1: đại diện cho một lần xuất hiện của từ khóa trong tài liệu

#### Bước 3: Shuffle và Group (Shuffle Phase)

Đây là giai đoạn quan trọng nhất trong quá trình xây dựng Inverted Index:

- Shuffle Operation: Spark tái phân phối dữ liệu giữa các Worker dựa trên giá trị khóa
  - Tất cả các cặp có cùng từ khóa và document ID sẽ được gửi đến cùng một Worker
  - Quá trình này yêu cầu truyền dữ liệu qua mạng giữa các Worker (network I/O)
- ReduceByKey Operation: Tính tổng số lần xuất hiện của mỗi cặp (từ khóa, tài liệu)
  - Kết quả là các cặp ((word, doc\_id), frequency)
- GroupByKey Operation: Gom nhóm các tài liệu theo từ khóa
  - Chuyển từ ((word, doc\_id), frequency) thành (word, [(doc\_id1, freq1), (doc\_id2, freq2), ...])

Chi tiết kỹ thuật:

- Quá trình shuffle được tối ưu hóa bởi Spark thông qua hash partitioning
- Mỗi từ khóa được hash để xác định Worker nào sẽ xử lý nó
- reduceByKey() kết hợp cả phép shuffle và reduce, hiệu quả hơn groupByKey() vì nó giảm lượng dữ liệu cần truyền
- Dữ liệu được gom nhóm dựa trên khóa (word) giúp tạo ra cấu trúc inverted index

#### Bước 4: Xây dựng chỉ mục cuối cùng (Reduce Phase)

Trong giai đoạn này:

- Mỗi Worker xử lý một tập hợp các từ khóa đã được phân phối sau quá trình shuffle
- Các cặp (document ID, frequency) được chuyển đổi thành dictionary
- Kết quả cuối cùng cho mỗi từ khóa là một dictionary với:
  - Key: ID của tài liệu
  - Value: Số lần từ khóa xuất hiện trong tài liệu đó

Chi tiết kỹ thuật:

- Hàm mapValues() chỉ biến đổi giá trị của RDD mà không thay đổi khóa
- Dictionary cung cấp độ phức tạp truy cập O(1) tới tần suất xuất hiện của từ khóa trong mỗi tài liệu

Bước 5: Thu thập kết quả (Collection Phase)

Giai đoạn cuối cùng:

- Phương thức collect() gửi yêu cầu từ Driver đến tất cả các Worker để lấy dữ liệu
- Tất cả các phần của inverted index từ các Worker khác nhau được gửi về Driver
- Driver tổng hợp kết quả thành một dictionary Python
- Kết quả cuối cùng là cấu trúc: {word1: {doc\_id1: freq1, doc\_id2: freq2, ...}, word2: {...}, ...}

Chi tiết kỹ thuật:

- Đây là bước tập trung dữ liệu duy nhất trong toàn bộ quá trình
- Có thể gây ra vấn đề bộ nhớ nếu kết quả quá lớn (không vừa trong RAM của Driver)
- Trong các ứng dụng thực tế, kết quả thường được lưu trữ vào hệ thống lưu trữ phân tán (HDFS, S3) thay vì thu thập về Driver

## b) Cài đặt Inverted Indexing trong Spark

Mã giả:

```

1. FUNCTION BuildInvertedIndex(documents)
2. // Khởi tạo và phân tán
3. spark ← InitSparkSession()
4. // Xây dựng chỉ mục
5. rdd ← spark.parallelize(documents)
6. tokenized ← Tokenize(text, stopwords))
7. // Tạo cặp từ-tài liệu phân tán
8. wordDocPairs ← tokenized.flatMap(id, tokens)
9. // Đếm tần suất và nhóm theo từ

```

```

10. frequencies ← wordDocPairs.reduceByKey((a, b) → a + b)
11. invertedIndex ← frequencies.map(((word, docId), freq) → (word,
(docId, freq)))
12. invertedIndex_result ← dict(invertedIndex.collect())
13. RETURN invertedIndex_result
14. END FUNCTION

```

Cách hoạt động của từng cấu hình:

### **Cấu hình 3 Worker – 1 Master**

Cấu hình Spark:

```

spark = SparkSession.builder \
 .appName("InvertedIndex") \
 .master("spark://spark-master:7077") \
 .config("spark.executor.cores", "2") \
 .config("spark.executor.memory", "2g") \
 .config("spark.driver.memory", "2g") \
 .getOrCreate()

```

Lệnh:

```

docker exec -it spark-master spark-submit --master spark://spark-
master:7077 /app/inverted_index.py

```

Cơ chế hoạt động:

- Hệ thống có 1 master và 3 worker, mỗi worker chạy 1 executor, tổng cộng có 3 executor hoạt động đồng thời.
- Mỗi executor được cấp 2 core, tổng cộng hệ thống sử dụng 6 core để xử lý dữ liệu.
- Khi chạy 5.000 tài liệu:
  - Spark tạo 3/3 partition cho dữ liệu.
  - Hệ thống có 4 stage (0,1,2,3).
  - Tổng 12 TID (Task ID) được khởi tạo, mỗi executor xử lý 4 TID.
- Khi chạy 10.000, 20.000, 30.000 tài liệu:
  - Spark tạo 16/16 partition, tổng cộng 64 TID.
  - Hệ thống vẫn có 4 stage.
  - Mỗi executor xử lý 16 TID.

### **Cấu hình 2 Worker – 1 Master**

Cấu hình Spark:

```

spark = SparkSession.builder \
 .appName("InvertedIndex") \
 .master("spark://spark-master:7077") \
 .config("spark.executor.cores", "2") \
 .config("spark.executor.memory", "2g") \

```

```
.config("spark.driver.memory", "2g") \
.getOrCreate()
```

Lệnh:

```
docker exec -it spark-master spark-submit --master spark://spark-
master:7077 /app/inverted_index.py
```

Cơ chế:

- Có 2 worker, mỗi worker chạy 1 executor, chia nhau xử lý.
- Thông kê:
  - Khi xử lý 5k tài liệu: tạo 2/2 partition, 4 stage, tổng 8 TID, 2 executor. Mỗi stage xử lý 2 TID với mỗi executor xử lý 1 TID
  - Với 10k, 20k, 30k tài liệu: 16/16 partition, 4 stage, tổng cộng 64 TID, 2 executor. Mỗi stage xử lý 16 TID với mỗi executor xử lý 8 TID
- Với 5.000, 10.000 tài liệu:
  - Log: “2/2 partition” được tạo ra ở mỗi stage.
  - Số Stage: 4 stage.
  - Executor: Có 2 executor
  - Phân bổ Task: Tổng cộng 8 task (TID) được khởi tạo.
- Với 20.000, 30.000 tài liệu:
  - Log: “4/4 task” (16 TID) được hiển thị, qua 4 stage.
  - Executor: 2 executor (mỗi executor chạy trên một worker). Mỗi worker có 2 core vậy mỗi executor sẽ dùng 2 core

## Cấu hình 1 Worker – 1 Master

Cấu hình Spark:

```
spark = SparkSession.builder \
 .appName("InvertedIndex") \
 .master("spark://spark-master:7077") \
 .config("spark.executor.cores", "2") \
 .config("spark.executor.memory", "2g") \
 .config("spark.driver.memory", "2g") \
 .getOrCreate()
```

Lệnh:

```
docker exec -it spark-master spark-submit --master spark://spark-
master:7077 /app/inverted_index.py
```

Cơ chế:

- Master điều phối, toàn bộ task chạy trên đúng 1 worker (1 executor).
  - 1 executor xử lý.
- Với 30.000, 20.000, 10.000, 5.000 tài liệu:

- Log: “2/2 partition” mỗi stage (8 TID tổng cộng).
- Executor: 1 executor chạy trên worker duy nhất. Sử dụng 2 core (ứng với số core của 1 worker)
- Số Stage: 4 stage.

## Local Mode

Cấu hình Spark:

```
spark = SparkSession.builder \
 .appName("InvertedIndex") \
 .master("local[2]") \
 .config("spark.executor.cores", "2") \
 .config("spark.executor.memory", "2g") \
 .config("spark.driver.memory", "2g") \
 .getOrCreate()
```

Lệnh:

```
docker exec -it spark-master spark-submit --master local[2]
/app/inverted_index.py
```

Cơ chế:

- Tất cả xử lý trên một tiến trình (1 executor trên máy `spark-master`).
- Kết quả tổng hợp khi chạy với 5k, 10k, 20k, 30k tài liệu:
  - Mỗi lần: 8/8 partition, 4 stage (0,1,2,3), tổng 32 TID.
  - 1 executor thực hiện toàn bộ task.
- Với 30.000, 20.000, 10.000, 5.000 tài liệu:
  - Log: “2/2 partition” (8 TID) được tạo ra.
  - Executor: Toàn bộ xử lý diễn ra trên executor driver. Sử dụng 2 core của máy.
  - Số Stage: 4 stage (mỗi stage xử lý 8 TID).

### c) Kết quả thực nghiệm

**Môi trường thực nghiệm:**

Cấu hình một Spark Cluster gồm 1 master và nhiều worker trên môi trường Docker với cấu hình cụ thể như sau:

- Docker Image: bitnami/spark:3.5.1
- Cấu hình Spark Master: 2 vCPU, 2GB RAM
- Cấu hình Spark Worker: 2 vCPU, 2GB RAM

Tiến hành thực nghiệm trên các trường hợp sau:

- 1 master 0 worker

- 1 master 1 worker
- 1 master 2 worker
- 1 master 3 worker

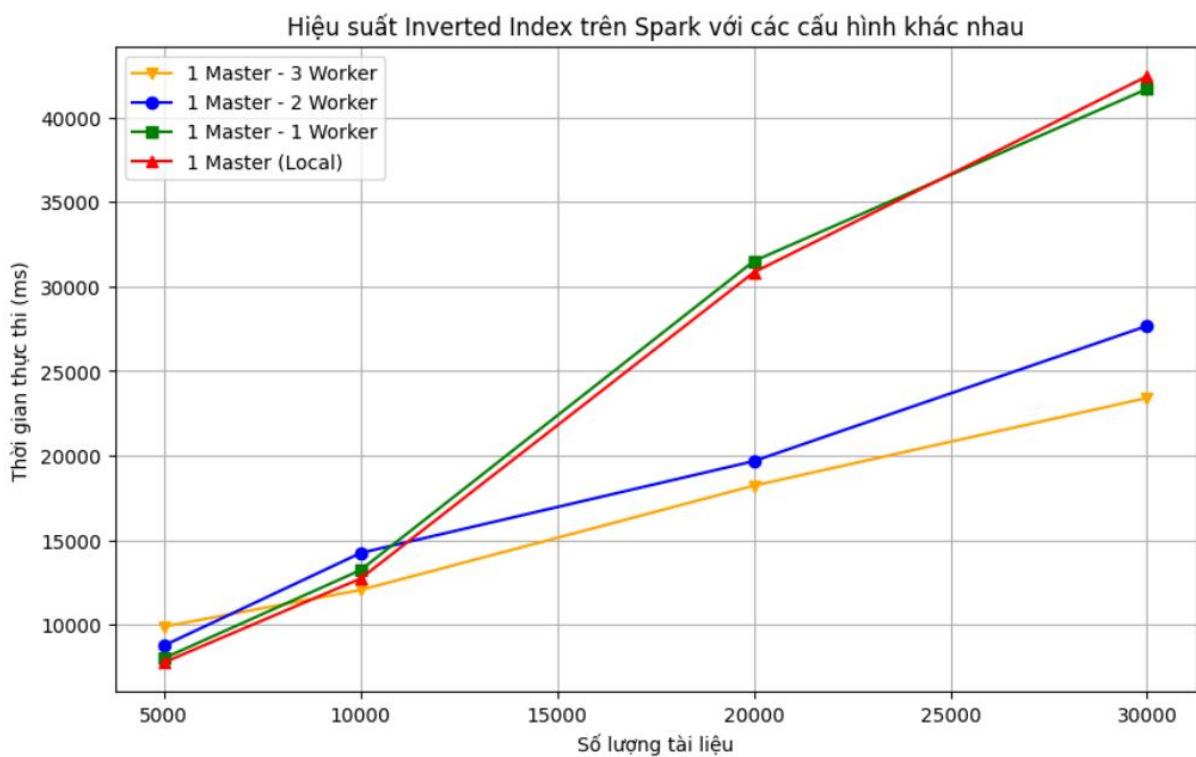
### Dữ liệu:

Sử dụng tập dữ liệu Wikipedia Movie Plots từ Kaggle, tài liệu sẽ có dạng là với mỗi hàng sẽ là 1 document và chỉ lấy cột Plot là nội dung của document đó.

### Kết quả:

*Bảng 5.6. Kết quả thời gian chạy Inverted Indexing của các cấu hình Spark (Giây).*

| Số tài liệu<br>(docs) | Master + 3<br>Workers (s) | Master + 2<br>Workers (s) | Master + 1<br>Worker (s) | Master/Local<br>(s) |
|-----------------------|---------------------------|---------------------------|--------------------------|---------------------|
| 5,000                 | 9.885                     | 8.764                     | 8.014                    | <b>7.763</b>        |
| 10,000                | <b>12.049</b>             | 14.236                    | 13.225                   | 12.724              |
| 20,000                | <b>18.219</b>             | 19.670                    | 31.499                   | 30.837              |
| 30,000                | <b>23.409</b>             | 27.668                    | 41.683                   | 42.435              |



*Hình 5.32. Biểu đồ thời gian chạy Inverted Indexing của các cấu hình Spark.*

Từ **bảng 5.6**, ta có thể biểu diễn trực quan kết quả thực nghiệm dưới dạng đồ thị như **hình 5.32**, ta có nhận xét như sau:

- Trường hợp "1 Master - 3 Worker"

- 3 Worker đạt hiệu suất tốt nhất khi số tài liệu lớn. Khi số tài liệu tăng lên, hiệu suất càng tốt hơn so với các cấu hình khác.
  - 20,000 tài liệu: 3 Worker chạy **18.219 giây**, nhanh hơn 2 Worker (19.670 giây) và 1 Worker (31.499 giây).
- Lý do: Có nhiều worker giúp phân tán khối lượng công việc, giảm tải cho từng node, do đó tốc độ xử lý cao hơn. Khi số tài liệu tăng lên, sự khác biệt về hiệu suất giữa 3 Worker và các cấu hình khác càng rõ rệt.
- Trường hợp "1 Master - 2 Worker"
  - Ban đầu có hiệu suất tương đương hoặc tốt hơn "1 Master - 3 Worker" với số tài liệu nhỏ. Với 5,000 tài liệu 2 Worker chạy **8.764 giây** nhanh hơn 3 Worker (9.885 giây).
  - Khi số lượng tài liệu tăng lên, thời gian chạy cao hơn so với "1 Master - 3 Worker", nhưng vẫn thấp hơn so với các cấu hình ít worker hơn. Với 30,000 tài liệu 2 Worker chạy 27.668 giây lâu hơn 3 Worker (23.409 giây) nhưng vẫn nhanh hơn 1 Worker (41.683 giây) và Local (42.435 giây).
  - Lý do: Việc có 2 worker giúp cải thiện hiệu suất đáng kể so với 1 worker, nhưng khi tài liệu tăng lên, việc thiếu 1 worker so với trường hợp 3 worker dẫn đến hiệu suất bị giảm nhẹ.
- Trường hợp "1 Master - 1 Worker"
  - Ban đầu tương đương với chế độ Local, nhưng khi số lượng tài liệu tăng lên, thời gian xử lý tăng đáng kể.
  - Khi số lượng tài liệu lớn (20,000 - 30,000), tốc độ chậm hơn rõ rệt so với các cấu hình có nhiều worker.
  - Lý do: Chỉ có 1 worker xử lý toàn bộ dữ liệu nên không tận dụng được khả năng phân tán của Spark, dẫn đến quá tải khi tài liệu lớn.
- Trường hợp "1 Master (Local)"
  - Hiệu suất ban đầu tương đương 1 Master – Worker và là cấu hình chạy nhanh nhất, nhưng khi số tài liệu tăng lên, tốc độ sẽ chậm hơn so với 2 Worker và 3 Worker.
    - Với 5,000 tài liệu Local Mode chạy **7.763 giây** nhanh hơn 1 Worker (8.014 giây), 2 Worker (8.764 giây), 3 Worker (9.885 giây)

- Với 20,000 tài liệu: Local Mode (30.837 giây), gần bằng 1 Worker (31.499 giây), 2 Worker chạy 27.668 giây, 3 Worker chạy **23.409 giây.**
  - Lý do: Local mode có thể tối ưu hơn trong việc tận dụng tài nguyên CPU/RAM vì không có chi phí truyền dữ liệu qua mạng. Khi số tài liệu lớn, nhược điểm không có khả năng phân tán xử lý sẽ trở nên rõ ràng.
- Lý do "1 Master - 1 Worker" chậm hơn "1 Master (Local)":
  - Khi chạy trên cluster với 1 worker, Spark vẫn cần truyền dữ liệu giữa master và worker, gây thêm độ trễ.
  - Ở chế độ local, toàn bộ xử lý diễn ra trên một tiến trình duy nhất, không mất chi phí truyền dữ liệu qua mạng.
  - Khi chỉ có 1 worker, lợi ích phân tán tải không đáng kể, nhưng chi phí truyền dữ liệu lại cao hơn, dẫn đến hiệu suất thấp hơn so với Local mode.

## CHƯƠNG 6: KẾT LUẬN

### 6.1. Tổng kết

Qua đề tài, chúng ta đã có một cái nhìn trực quan về cơ chế xử lý dữ liệu song song và phân tán. Cụ thể với các ví dụ minh họa cho một số bài toán đã thu được kết quả thực nghiệm trên cả hai cơ chế xử lý dữ liệu, kết quả cho thấy:

- Xử lý song song trên GPU mang lại hiệu suất vượt trội so với CPU trong hầu hết các trường hợp, đặc biệt khi tận dụng shared memory.
- Xử lý phân tán trên Spark phù hợp với các bài toán có kích thước lớn, nhưng hiệu quả bị ảnh hưởng bởi chi phí truyền dữ liệu giữa các node.
- Hiệu suất của các thuật toán phụ thuộc vào kích thước dữ liệu, số lượng worker, và cách thức quản lý bộ nhớ trong từng môi trường.

Kết quả thực nghiệm đã cung cấp một cái nhìn toàn diện về ưu và nhược điểm của từng phương pháp, từ đó giúp định hướng cho các nghiên cứu tiếp theo nhằm tối ưu hóa xử lý dữ liệu lớn.

### 6.2. Hướng phát triển

Dựa trên những kết quả đạt được, hướng phát triển tiếp theo của đề tài sẽ tập trung vào việc kết hợp xử lý phân tán và song song nhằm tận dụng tối đa hiệu suất của hệ thống:

- Kết hợp GPU vào môi trường xử lý phân tán: Trong các hệ thống phân tán như Spark, việc tích hợp GPU vào từng node có thể cải thiện hiệu suất đáng kể, giảm thiểu thời gian xử lý dữ liệu lớn.
- Ứng dụng vào các bài toán thực tế: Áp dụng phương pháp này vào các lĩnh vực như xử lý hình ảnh y tế, phân tích dữ liệu lớn trong tài chính, AI phân tán, và mô phỏng khoa học.

Với những hướng phát triển này, nghiên cứu có thể tiếp tục mở rộng, mang lại những cải tiến đáng kể trong xử lý dữ liệu hiệu năng cao.

## TÀI LIỆU THAM KHẢO

- [1]. Hillis, W. D., & Steele Jr, G. L. (1986). Data parallel algorithms. *Communications of the ACM*, 29(12), 1170-1183.
- [2]. Đỗ Như Tài. (2014, October). Introduction to CUDA programming [PowerPoint slides]. Trường Đại học Ngoại ngữ - Tin học TP.HCM
- [3]. Jung, S., Chang, D. J., & Park, J. W. (2017). Large scale document inversion using a multi-threaded computing system. *ACM SIGAPP Applied Computing Review*, 17(2), 27-35.
- [4]. Shrivastava, A. (2016, January 15). Crash Course in Map Reduce and GPU Programming [PDF slides]. Rice University, COMP 441.
- [5]. Khan, M. F., Paul, R., Ahmed, I., & Ghafoor, A. (1999). Intensive data management in parallel systems: A survey. *Distributed and Parallel Databases*, 7, 383-414.
- [6]. Prahara, A., Pranolo, A., Anwar, N., & Mao, Y. (2021). Parallel Approach of Adaptive Image Thresholding Algorithm on GPU. *Knowl. Eng. Data Sci.*, 4(2), 69-84.
- [7]. Heo, J., Won, J., Lee, Y., Bharuka, S., Jang, J., Ham, T. J., & Lee, J. W. (2020, March). IIU: Specialized architecture for inverted index search. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 1233-1245).
- [8]. Zhang, Y., Cao, T., Li, S., Tian, X., Yuan, L., Jia, H., & Vasilakos, A. V. (2016). Parallel processing systems for big data: a survey. *Proceedings of the IEEE*, 104(11), 2114-2136.
- [9]. Shetty, N. R., Prasad, N. H., & Nalini, N. (2022). Emerging research in computing, information, communication and applications. Springer Singapore.
- [10]. Ray, U., Hazra, T. K., & Ray, U. K. (2016). Matrix multiplication using Strassen's algorithm on CPU & GPU. *International Journal of Computer Sciences and Engineering*, 98-105.
- [11]. Green, O., McColl, R., & Bader, D. A. (2012, June). GPU merge path: a GPU merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing* (pp. 331-340).

- [12]. Pibiri, G. E., & Venturini, R. (2020). Techniques for inverted index compression. *ACM Computing Surveys (CSUR)*, 53(6), 1-36.
- [13]. Jung, S., Chang, D. J., & Park, J. W. (2017). Large scale document inversion using a multi-threaded computing system. *ACM SIGAPP Applied Computing Review*, 17(2), 27-35.
- [14]. Zaharia, M., et al. (2010). Spark: Cluster Computing with Working Sets.
- [15]. Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (pp. 10-10).
- [16]. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., ... & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (pp. 15-28).
- [17]. Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., & Chun, B. G. (2015). Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (pp. 293-307).
- [18]. Venkataraman, S., Yang, Z., Franklin, M., Recht, B., & Stoica, I. (2016). Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (pp. 363-378).
- [19]. Xu, L., Li, M., Zhang, L., Butt, A. R., Wang, Y., & Hu, Z. (2016). MEMTUNE: Dynamic memory management for in-memory data analytic platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (pp. 383-392).
- [20]. Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., ... & Zaharia, M. (2015). Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (pp. 1383-1394).
- [21]. Armbrust, M., Das, T., Torres, J., Yavuz, B., Zhu, S., Xin, R., ... & Zaharia, M. (2018). Structured streaming: A declarative API for real-time applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (pp. 601-613).

- [22]. Xin, R. S., Wendell, P., & Zaharia, M. (2018). Adaptive execution for batch and streaming analytics. In 2018 IEEE International Conference on Big Data (Big Data) (pp. 1911-1920).