НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет програмних систем та прикладної математики
Кафедра системного програмування і спеціалізованих комп'ютерних систем

До захисту допущено:

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою

«Системне програмування та спеціалізовані комп'ютерні системи»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Комп'ютерна система розпаралелення алгоритмів машинного навчання мовою C++»

Виконав:
студент IV курсу, групи КВ-23
Марінченко Михайло Олексійович _____

Керівник:
доц. каф. СП і СКС, к.т.н
Марченко Олександр Іванович _____

Консультант з нормконтролю:
доц. каф. СП і СКС, к.т.н
Клятченко Ярослав Михайлович _____

Рецензент:
доц. каф. ПЗКС, к.т.н
Заболотня Тетяна Миколаївна _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2026 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет програмних систем та прикладної математики
Кафедра системного програмування і спеціалізованих
комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 123 «Комп'ютерна інженерія»
Освітньо-професійна програма «Системне програмування та спеціалізовані комп'ютерні системи»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
«___»_____20__ р.

ЗАВДАННЯ
на дипломний проєкт студенту
Марінченку Михайлу Олексійовичу

1. Тема проєкту «Комп'ютерна система розпаралелення алгоритмів машинного навчання мовою C++», керівник проєкту Марченко О.І., кандидат технічних наук, доцент кафедри системного програмування і спеціалізованих комп'ютерних систем, затверджені наказом по університету від «29» травня 2026р. № N1808-С
2. Термін подання студентом проєкту _____
3. Вихідні дані проєкту: див. Технічне завдання.
4. Зміст пояснювальної записки
   - аналіз існуючих рішень;
   - розробка системи розпаралелення алгоритмів на 3 рівні;
   - тестування;
   - керівництво користувача.

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій, тощо)

- загальна структура системи
- UML-діаграма класів локального рівня
- алгоритм локального балансування навантаження
- робота балансувальника мережевого рівня в межах одної ітерації.

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Клятченко Я.М., к.т.н., доцент каф. СП і СКС		

1. Дата видачі завдання: 15 вересня 2025р.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Терміни виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проєкту	15.11.2025	
2.	Розроблення та узгодження технічного завдання	24.11.2025	
3.	Аналіз існуючих рішень	06.05.2026	
4.	Підготовка матеріалів першого розділу дипломного проєкту	10.05.2026	
5.	Підготовка матеріалів другого розділу дипломного проєкту	20.05.2026	
6.	Розробка програмного забезпечення	22.05.2026	
7.	Підготовка матеріалів третього розділу дипломного проєкту	24.05.2026	
8.	Підготовка графічної частини дипломного проєкту	25.05.2026	

9.	Підготовка матеріалів четвертого розділу дипломного проєкту	28.05.2026	
10.	Підготовка матеріалів п’ятого розділу дипломного проєкту	29.05.2026	
11.	Оформлення документації дипломного проєкту	30.05.2026	
12.	Попередній огляд матеріалів диплому на кафедрі	30.05.2026	

Студент _____ Михайло МАРІНЧЕНКО
(підпис)

Керівник проєкту _____ Олександр МАРЧЕНКО
(підпис)

АНОТАЦІЯ

Бакалаврський дипломний проєкт включає пояснювальну записку (72 стор., 16 рис., 2 табл., список використаної літератури з 7 найменувань, 4 додатки).

Об’єкт розробки - створення програмної системи для розпаралелення алгоритмів навчання нейронних мереж, з можливістю дослідницької діяльності.

Дана програмна система дозволяє:

- забезпечити паралельне виконання алгоритму навчання нейронної мережі на трьох рівнях архітектури: мережевому, апаратному та багатопоточності;
- масштабування обчислень, завдяки паралельному виконанню;
- динамічний розподіл даних;
- математично точна синхронізація;
- проводити порівняльний аналіз динамічного балансування навантаження між центральним та графічним процесорами
- аналізувати вплив мережевих затримок на роботу кластеру
- оцінювати ефективність багаторівневого розгалуження алгоритму навчання з лінійним варіантом навчання

В процесі розробки були використані стандарти та інструменти мови C++ (C++17/20), зокрема бібліотеки для багатопоточності (<thread>, <mutex>), роботи з файловою системою (<filesystem>), а також функціональні обгортки та лямбда-вирази для реалізації архітектури зворотних викликів (callback).

Для побудови системи збирання та керування залежностями застосовано кросплатформні інструменти CMake та vcpkg.

Також була використана стороння бібліотека:

- LibTorch (C++ API for PyTorch) – для виконання оптимізованих тензорних обчислень та роботи з архітектурами нейронних мереж на базі апаратного прискорення (NVIDIA CUDA).

- ZeroMQ (cppzmq) – для організації високонадійної асинхронної мережевої взаємодії між вузлами кластера за патернами REQ/REP (рукостискання) та PUB/SUB (мережеві бар'єри).

В ході розробки:

− проведено аналіз існуючих рішень;

Ключові слова:

СИСТЕМА, РОЗПАРАЛЕЛЕННЯ, РІВНІ, КЛАСТЕРИ, CPU, GPU, KERNEL, РОЗПОДІЛЕНЕ ОБЧИСЛЕННЯ, НЕЙРОННІЙ МЕРЕЖІ, C++, СИНХРОНІЗАЦІЯ ГРАДІЄНТІВ, LIBTORCH, ZEROMQ, CPPZMQ.

ABSTRACT

The Bachelor's thesis comprises an explanatory note (72 pages, 16 figures, 2 tables, a bibliography of 7 references, and 4 appendices).

The aim of the project is to develop a software system for the parallel training of neural networks, with the capability to support research activities.

This software system allows:

- parallel execution of the neural network training algorithm across three architectural levels: network, hardware and multithreading;
- scalability of computations, thanks to parallel execution;
- dynamic data distribution;
- mathematically precise synchronisation;
- conduct a comparative analysis of dynamic load balancing between the central and graphics processors;
- analyse the impact of network delays on cluster performance;
- evaluate the effectiveness of multi-level branching of the training algorithm compared to a linear training variant

During development, C++ standards and tools (C++17/20) were utilised, specifically libraries for multithreading (<thread>, <mutex>), file system operations (<filesystem>), as well as functional wrappers and lambda expressions for implementing callback architecture.

The cross-platform tools CMake and vcpkg were used to build the dependency collection and management system.

A third-party library was also used:

- LibTorch (C++ API for PyTorch) – for performing optimised tensor computations and working with hardware-accelerated neural network architectures (NVIDIA CUDA).
- ZeroMQ (cppzmq) – for organising highly reliable asynchronous network communication between cluster nodes using the REQ/REP (handshake) and PUB/SUB (network barriers) patterns.

During development:

- an analysis of existing solutions was carried out;

Keywords:

SYSTEM, PARALLELISATION, LEVELS, CLUSTERS, CPU, GPU, KERNEL, DISTRIBUTED COMPUTING, NEURAL NETWORKS, C++, GRADIENT SYNCHRONISATION, LIBTORCH, ZEROMQ, CPPZMQ.

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045490.002 ТЗ	Комп’ютерна система розпаралелення машинного навчання мовою С++ Технічне завдання	4	А4	
	А4	ІАЛЦ.045490.003 ТП	Комп’ютерна система розпаралелення машинного навчання мовою С++ Відомість технічного проєкту	1	А4	
	А4	ІАЛЦ.468300.004 ПЗ	Комп’ютерна система розпаралелення машинного навчання мовою С++ Пояснювальна записка	72	А4	

Змін.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.045490.001 ОА		
Розробив		Марінченко М.О.			Комп’ютерна система розпаралелення алгоритмів машинного навчання мовою С++ **Опис альбому**	Літ.	Аркуш 1 / Аркушів 2
Перевірив		Марченко О.І.					
Консульт.						КПІ ім. Ігоря Сікорського, ФПСПМ, КВ-23	
Н. контроль		Клятченко Я.М.					
Зав. каф.		Романкевич В.О.					

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045490.005 Д1	Загальна структура	1	А4	
			системи			
			Схема структурна			
	А4	ІАЛЦ.045490.006 Д2	UML-діаграма класів	1	А4	
			локального рівня			
			Схема стурктурна			
	А4	ІАЛЦ.045490.007 Д3	Алгоритм локального	1	А4	
			балансування			
			навантаження			
			Схема алгоритму			
	А4	ІАЛЦ.045490.008 Д4	Робота балансувальника	1	А4	
			мережевого рівня			
			в межах одної ітерації.			
			Схема алгоритму			
		Диск CD-ROM	Текст пояснювальної			
			записки			
			Графічний матеріал			

Змін	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.045490.001 ОА	Арк. 2

ЗМІСТ

1. НАЙМЕНУВАННЯ І ОБЛАСТЬ ЗАСТОСУВАННЯ 2
2. ПІДСТАВА ДЛЯ РОЗРОБКИ..... 2
3. ЦІЛЬ І ПРИЗНАЧЕННЯ РОБОТИ 2
4. ДЖЕРЕЛА РОБОТИ 2
5. ТЕХНІЧНІ ВИМОГИ..... 3
6. ЕТАПИ РОЗРОБКИ 4

1. НАЙМЕНУВАННЯ І ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування проєкту - “комп'ютерна система розпаралелення алгоритмів машинного навчання мовою C++”.

Область застосування: інформаційні технології.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп’ютерних систем Національного технічного університету України “Київський Політехнічний Інститут імені Ігоря Сікорського”.

3. ЦІЛЬ І ПРИЗНАЧЕННЯ РОБОТИ

Метою дипломного проєкту є розробка комп’ютерної системи для розпаралелення алгоритмів машинного навчання мовою C++, з можливістю пришвидшити довільний алгоритм навчання, і дослідити його ефективність.

4. ДЖЕРЕЛА РОБОТИ

Джерелами інформації для розроблення є технічна література, публікації у періодичних виданнях та Інтернет ресурси.

5. ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до системи, що розробляється

- можливість розподіленого навчання нейронних мереж на кількох комп'ютерах;
- можливість динамічного та симетричного балансування навантаження між CPU та GPU;
- можливість масштабування обчислень через підключення нових вузлів (Worker);
- можливість прозорого відкладеного завантаження великих датасетів (Lazy Loading) з диска;
- можливість гнучкого налаштування мережевих параметрів, сіда (seed) та збору метрик.

5.2 Рекомендовані вимоги до апаратного забезпечення

- процесор: AMD Ryzen 5 / Intel Core i5 або подібний;
- відеоадаптер: сімейство відеоадаптерів NVIDIA з підтримкою CUDA;
- оперативна пам’ять: 8 ГБ (рекомендовано 16 ГБ);
- простір на диску: 10 ГБ (обов'язково SSD для швидкого читання тензорів);
- мережа: Gigabit Ethernet або WiFi маршрутизатор.

5.3 Вимоги до мінімального програмного забезпечення

- операційна система: Windows 10/11;
- компілятор: MSVC 2022 (для Windows) з підтримкою C++17;
- інструменти збирання: CMake, пакетний менеджер vcpkg;
- фреймворки та бібліотеки: LibTorch (PyTorch C++ API), ZeroMQ (cppzmq).

6. ЕТАПИ РОЗРОБКИ

№ з/п	Назва етапів виконання дипломного проєкту	Терміни виконання етапів проєкту
1.	Вивчення літератури за тематикою проєкту	15.11.2025
2.	Розроблення та узгодження технічного завдання	24.11.2025
3.	Аналіз існуючих рішень	06.05.2026
4.	Підготовка матеріалів першого розділу дипломного проєкту	10.05.2026
5.	Підготовка матеріалів другого розділу дипломного проєкту	20.05.2026
6.	Розробка програмного забезпечення	22.05.2026
8.	Підготовка матеріалів третього розділу дипломного проєкту	24.05.2026
9.	Підготовка графічної частини дипломного проєкту	25.05.2026
10.	Підготовка матеріалів четвертого розділу дипломного проєкту	28.05.2026
11.	Підготовка матеріалів п’ятого розділу дипломного проєкту	29.05.2026
11.	Оформлення документації дипломного проєкту	30.05.2026
12.	Попередній огляд матеріалів диплому на кафедрі	30.05.2026

ВІДОМІСТЬ ТЕХНІЧНОГО ПРОЄКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4	ІАЛЦ.045490.000	Завдання на дипломний проєкт	1	
2	А4	ІАЛЦ.045490.001 ОА	Опис альбому	2	
3	А4	ІАЛЦ.045490.002 ТЗ	Технічне завдання	4	
4	А4	ІАЛЦ.045490.003 ТП	Відомість технічного проєкту	1	
5	А4	ІАЛЦ.045490.004 ПЗ	Пояснювальна записка	72	
6	А4	ІАЛЦ.045490.005 Д1	Загальна структура системи	1	
7	А4	ІАЛЦ.045490.006 Д2	UML-діаграма класів локального рівня	1	
8	А4	ІАЛЦ.045490.007 Д3	Алгоритм локального балансування навантаження	1	
9	А4	ІАЛЦ.045490.008 Д4	Робота балансувальника мережевого рівня в межах одної ітерації.	1	

				ІАЛЦ.045490.003 ТП		
	ПІБ	Підп.	Дата			
Розробн.	Марінченко М.О.			Комп'ютерна система розпаралелення алгоритмів машинного навчання мовою С++ **Відомість технічного проєкту**	Лист	Листів
Керівн.	Марченко О.І.				1	1
Консульт.					КПІ ім. Ігоря Сікорського Каф. СП і СКС Гр. КВ-23	
Н/контр.	Клятченко Я.М.					
Зав.каф.	Романкевич В.О.					

Пояснювальна записка до дипломного проєкту

на тему: Комп'ютерна система розпаралелення алгоритмів машинного навчання мовою C++

Київ – 2026

ЗМІСТ

ВСТУП 5
1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ 6
1.1 Огляд сучасних методів розпаралелення машинного навчання 6
1.1.1 Паралелізм за даними (Data Parallelism) 6
1.1.2 Паралелізм за моделлю (Model Parallelism) 8
1.2 Особливості паралельних обчислень у машинному навчанні 10
1.2.1 Мережеве пляшкове горлечко 11
1.2.2 Проблема гетерогенного обладнання 12
1.2.3 Ефект відсталого вузла 13
1.3 Порівняльний аналіз існуючих рішень 15
1.3.1 PyTorch DistributedDataParallel 15
1.3.2 DeepSpeed 16
1.3.3 Federeted Learning 17
1.4 Висновки до першого розділу 18
2. ПРОЄКТУВАННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ РОЗПАРАЛЕЛЕННЯ АЛГОРИТМІВ 20
2.1 Загальна структура системи 20
2.1 Кластерний рівень 22
2.3 Локальний рівень 24
2.4 Рівень багатопоточності 26
2.4.1 Управління потоками на рівні розробленої системи 26
2.4.2 Внутрішній паралелізм LibTorch 27
2.5 Проєктування програмного інтерфейсу користувача 28
2.6 Висновки до другого розділу 30
3. РЕАЛІЗАЦІЯ 31
3.1 Реалізація локального рівня та модуля розподілення навантаження 31
3.1.1 Апаратний обчислювач (клас Worker) 34
3.1.2 Алгоритм динамічного балансування (клас LoadBalancer) 35
3.1.3 Реалізація прихованої багатопотокової бар'єрної синхронізації 39

3.1.4 Зважене усереднення градієнтів та синхронізація моделей 41
3.2 Реалізація кластерної взаємодії між вузлами системи 43
3.2.1 Структура мережевої взаємодії 43
3.2.2 Глобальне балансування навантаження на мережевому рівні 44
3.2.3 Синхронізація генераторів та серіалізація тензорів 47
3.2.4 Зважене математичне усереднення глобальних градієнтів 47
3.2.5 Бар'єрна синхронізація 48
3.3 Реалізація та використання програмного інтерфейсу користувача 49
3.3.1 Інтеграція нейронної мережі та набору даних 50
3.3.2 Конфігурація параметрів навчання та мережі 51
3.3.3 Моніторинг процесу (Callback API) 53
3.3.4 Ініціалізація, впровадження залежностей та запуск навчання 54
3.3.5 Ізольоване локальне виконання та вибір режимів апаратного прискорення 55
3.4 Висновки до третього розділу 56
4. ТЕСТУВАННЯ ТА ОЦІНКА ПРОДУКТИВНОСТІ 57
4.1 Тестування коректності роботи системи 57
4.2 Оцінка продуктивності послідовного виконання 58
4.3 Дослідження продуктивності паралельного виконання 60
4.4 Порівняльний аналіз 65
4.5 Висновки до четвертого розділу 66
5. КЕРІВНИЦТВО КОРИСТУВАЧА 67
ВИСНОВКИ 69
СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ 71

ДОДАТКИ

Додаток 1. Копії графічних матеріалів

- ІАЛЦ.045490.005 Д1. Загальна структура системи. Схема структурна
- ІАЛЦ.045490.006 Д2. UML-діаграма класів локального рівня. Схема структурна
- ІАЛЦ.045490.007 Д3. Алгоритм локального балансування навантаження. Схема алгоритму

- ІАЛЦ.045490.008 Д4. Робота балансувальника мережевого рівня в межах одної ітерації.. Схема алгоритму

Додаток 2. Фрагменти програмного коду

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

API (Application Programming Interface) — інтерфейс програмування додатків, набір готових функцій та процедур, які надає програмна система для використання в інших програмах.

CPU (Central Processing Unit) — центральний процесор, основний обчислювальний пристрій комп’ютера.

CUDA (Compute Unified Device Architecture) — програмно-апаратна архітектура паралельних обчислень від NVIDIA, що дозволяє використовувати графічні процесори для обчислень загального призначення.

DDP (Distributed Data Parallel) — метод паралельного навчання нейронних мереж за даними, що використовується у фреймворках машинного навчання.

GPU (Graphics Processing Unit) — графічний процесор, спеціалізований блок, оптимізований для паралельних обчислень.

LibTorch — бібліотека (C++ API для PyTorch), що використовується для виконання оптимізованих тензорних обчислень та роботи з нейронними мережами.

MSVC (Microsoft Visual C++) — компілятор мови C++ від компанії Microsoft.

RAM (Random Access Memory) — оперативна пам’ять пристрою. SSD (Solid State Drive) — твердотільний накопичувач даних. ZeroMQ (cppzmq) — високонадійна асинхронна бібліотека для організації мережевої взаємодії між вузлами.

ВСТУП

В наш час стрімкий розвиток нейронних мереж вимагає все більшої обробки великих обсягів даних, даний перехід для ефективного навчання вимагає переходу до паралельних обчислювань. На сьогоднішній день існує достатня частина інструментів для цієї задачі, проте значна більшість цих інсрументів відноситься до enterprise-сегменту. Вони вимагають складного розгортання інфраструктури, використання контейнеризації та специфічних налаштувань мережі. Пункти перераховані вище створюють високий поріг входу для використання інструментів та робить проблематичним швидке розгортання у простих умовах для маленьких команд розробників.

Частіше розробники при створенні власної реалізації для даних задач стикаються з двома головними проблемами. По-перше, виникає складність із синхронізацією градієнтів між декількома робочими станціями без перенавантаження мережевого трафіку. По-друге, вкрай неефективне використання апаратного забезпечення кожного робочого вузла, бо у стандартних скриптах для навчання нейронних мереж весь обсяг операцій по навчанню бере на себе графічний процесор (GPU), в той же час багатоядерний центральний процесор (CPU) виконує лише допоміжні функції, що і призводить до значного часу його простою. Всі перелічені факти призводять до значної втрати обчислювальної потужності.

Завданням дипломного проєкту є розробка легковісної системи мовою C++, який вирішує поставлені задачі шляхом багаторівневого розпаралелювання алгоритмів навчання нейронних мереж. Розроблена система забезпечить динамічний розподіл даних на всіх трьох рівнях, математично точну синхронізацію процесу навчання нейронних мереж. Крім того програмний комплекс значно спростить використання та налаштування розподіленої системи обчислень в середовищі C++.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1 Огляд сучасних методів розпаралелення машинного навчання

Останні роки виділяються швидким зростанням обсягів та складності обчислення в моделях машинного навчання. В усіх типах нейронних мереж кількість параметрів, які треба оптимізувати, збільшилась з мільйонів до мільярдів. Через обмеження обчислювального потенціалу одного пристрою призвело до того, що таке обчислення стало недоцільно і економічно невигідно.

З оглядом на фізичні обмеження, вся індустрія переходить від вертикального масштабування процесу обчислення до горизонтального, тобто перехід до об’єднання окремих ресурсів для збільшення обчислювального потенціалу.

На наш час у світовій практиці сформувалися декілька базових парадигм до розпаралелення алгоритмів навчання, кожен з них вирішує окрему специфічну апаратну проблему.

1.1.1 Паралелізм за даними (Data Parallelism)

Паралелізм за даними є найпоширенішим методом для паралельного навчання нейронних мереж, коли є можливість розмістити модель на робочому вузлі, але сам набір даних є досить великим, через що час навчання може становити неймовірну величину.

Процес навчання за даним методом є циклічним, і він складається з 4 основних фаз:

1) прямий прохід: кожен робочий вузол пропускає через локальну мережу призначений йому фрагмент даних, на цьому етапі кожен вузол працюю окремо і отримує локальне значення функції втрат;

2) зворотній прохід: за допомогою алгоритму зворотнього поширення помилки кожен робочій вузол обчислює свої локальні градієнти для ваг мережі на основі обробленого свого фрагменту даних;

3) етап агрегації градієнтів: кожен вузол призупиняє обчислення та обмінюються отриманими градієнтами для математичного усереднення і формують один глобальний градієнт, який ґрунтується на всіх оброблених даних;

4) оновлення ваг: глобальний градієнт застосовуються для ваг кожної локальної мережі, оскільки початкові ваги були ідентичні та на кожному вузлі застосовується єдиний градієнт, всі локальні моделі залишаються абсолютно синхронізованими для наступної ітерації;

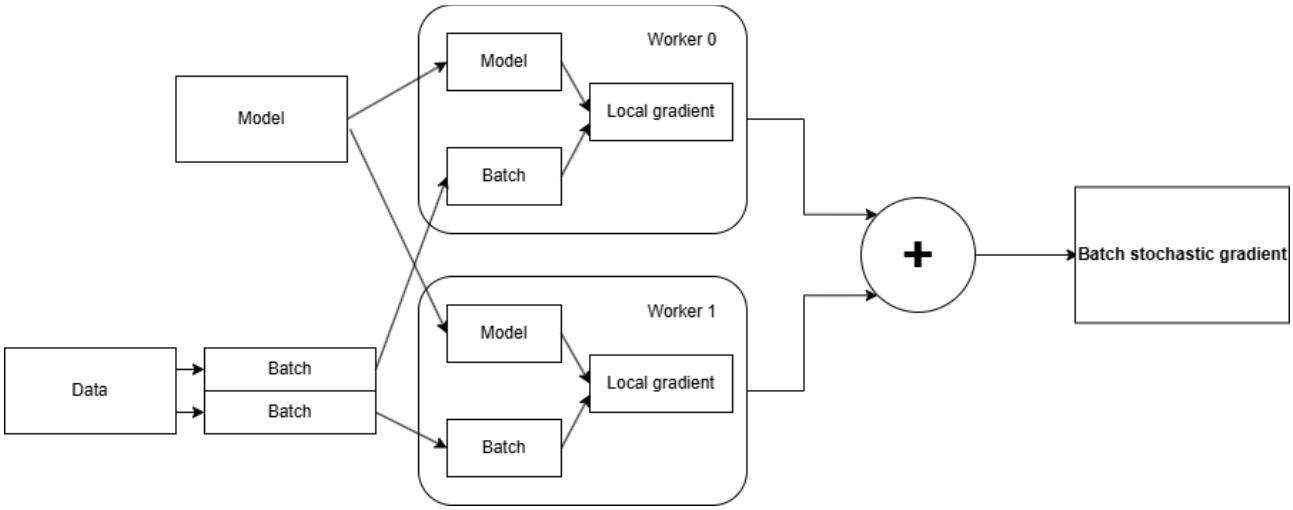

Рисунок — 1.1 Діаграма роботи парадигми паралельності за даними

Важливою характеристикою методу паралелізму за даними є його строга еквівалентність послідовному навчання. З точки зору стохастичного градієнтного

спуску, навчання на K паралельних вузлах з розміром фрагменту даних *G*, повністю відповідає навчанню на одному пристрої з розміром сегменту даних, виділених на навчання в ітерації, K×G. Це і є гарантією того, що паралельне обчислення впливає лише на швидкість обчислювань і гарантує їх точність.

1.1.2 Паралелізм за моделлю (Model Parallelism)

Незважаючи на високу ефективність попередньо розглянутого методу, він має значне обмеження в апаратному плані, воно вимагає, щоб повна архітектура моделі, включаючи її ваги, градієнти та інші параметри, повністю могла бути розміщенню в локальну пам’ять одного робочого вузла, який виконує обчислювання. З розвитком глибинного навчання та появою дуже масивних моделей за обсягом даних наприклад, моделі сімейства GPT. Кількість всіх використовуваних параметрів зросла в кількості до мільярдів, сучасний фізичний обсяг апаратного забезпечення дуже часто не є достатнім для моделей такого рівня. Для подолання цього бар’єру була розроблена концепція паралелізму за моделлю.

В основі даної парадигми використовується фрагментація архітектури нейронної мережі, а не набору даних використовуваних для навчання. Модель розрізають на логічні сегменти, які фізично завантажуються на два окремих обчислювальні вузли. Наприклад, у моделі, що має в собі X послідовних шарів, при використанні двох робочих вузлів поділяється між ними таким чином, що шари з 1 по $\frac{X}{2}$ ініціалізуються виключно на першому вузлі, а шари з $\frac{X}{2}+1$ до X на другому. В цьому варіанті сегмент даних для навчання не поділяється на фрагменти, а повністю подається на перший вузол.

При використання даної парадигми весь процес навчанні моделі по своїй суті можна спів-ставити з конвеєрною стрічкою. Життєвий цикл одного сегменту даних має наступний вигляд:

1) перший вузол виконує прямий прохід по своїх шарах моделі за отриманими даними, після чого отримує проміжні результати (сирі дані після проходу шару моделі);

2) вузол серіалізує отримані результати та передає на другий вузол;

3) другий вузол використовує отримані дані для проходу вже своїх шарів, саме цим закінчує прямий прохід та обчислює глобальну функцію втрат;

4) починається зворотній прохід, другий вузол обчислює градієнти для шарів, які були ініціалізовані на ньому, і передає градієнти на перший вузол.

Можна виділити таку особливість методу паралелізму за моделлю: для забезпечення постійної роботи обчислювального процесу, вузли повинні швидко обмінюватися великими обсягами даних (тензорами активації) під час кожного кроку прямих та зворотних проходів, що робить даний метод залежним від апаратної топології обладнання між обчислювальними вузлами.

З усього вище сказаного, можна зробити висновок, що паралелізм за даними та за моделлю використовують два фундаментально різні підходи для масштабування процесу навчання нейронних мереж. Проте кожна з цих парадигм стискається з різними проблемами при практичній реалізації на реальному обладнанні.

1.2 Особливості паралельних обчислень у машинному навчанні

Теоретична ефективність будь-яких розподілених системи обчислень базується на фундаментальному законі Амдала, згідно цього закону, максимальне очікуване прискорення від розпаралелювання задачі жорстко обмежується послідовною частиною або найповільнішою в системі.

Для задачі паралельного навчання нейронних мереж за стохастичним градієнтним спуском, цей закон набуває критично важливого значення. За цим алгоритмом процес навчання складається з двох основних фаз:

1) паралельна фаза: обчислення градієнтів ізольовано на кожному вузлі;

2) послідовна фаза: синхронізація, передача даних та оновлення ваг глобальної моделі.

Тому, навіть при нескінченно великому збільшенні робочих вузлів, ніколи час ітерації не стане меншим за послідовну частину навчання нейронної мережі.

Також у середовищах, де робочу вузли не відповідають один одному за обчислювальним потенціалом, виникає дисбаланс між високошвидкісним обчислення на графічних прискорювачах, центральним процесором та мережевою передачею даних. Цей дисбаланс формує проблему “пляшкового горла”, це стає проблемним місцем обчислювальної системи, де найвищу межу продуктивності, всієї обчислювальної системи, диктує мережева пропускна здатність або найслабший вузол за обчислювальним потенціалом.

1.2.1 Мережеве пляшкове горлечко

Однією з головних проблем систем розподіленого обчислення є велика різниця в швидкості обчислень графічних прискорювачів та передачі даних каналами зв’язку між ними. Такий ефект називають “мережевим пляшковим горлом”.

Навчання масивних моделей, які не можуть вміститися в пам’ять одного вузла для обчислень, потребують використання паралелізму за моделюю, але такий метод навчання в свою чергу вимагає безперервного обміну даними, тензорами активації, між вузлами, щоб не виникало простоювання обчислювальних можливостей. Великі дата-центри для рішення цієї проблеми використовують технології NVLink або мережеві рішення по стандарту InfiniBand. NVLink - це високошвидкісна технологія обміну даними напряму між графічними прискорювачами, яка мінує доволі повільну шину PCIe, як для цієї задачі. InfiniBand - це високошвидкісна комутована технологія мережевої взаємодії, також вона дозволяє отримати наднизьку затримку. Такі технології дозволяють отримати високу пропускну здатність на рівні близькому до 900Гб/с з мінімальної затримкою. Все вище сказане, дозволяє працювати всім робочим вузлам, як єдиний монолітний суперкомп'ютер.

На відміну від корпоративних рішень, об’єднання пристроїв наприклад, домашні персональні комп’ютери та ноутбуки, для використання в робочих лабораторіях та офісах проходить за допомогою Gigabit Ethernet(LAN) або Wi-Fi, дані способи здатні забезпечити обмін даними зі швидкість до 50-110 Мб/с, дана швидкість в тисячі разів менша за швидкість, яку використовують в дата-центрах за допомогою спеціалізованих шин обміну даними та мережевими стандартами. Побутові маршрутизатори, які використовуються для з’єднання робочих вузлів в даних обчислювальних кластерах, схильні до втрати пакетів та вносять значну затримку.

З вище сказаного, може зазначити, що спроба реалізація паралелізму за моделлю в умовах звичайної локальної мережі призведе до повної втрати обчислювального потенціалу, тому що передача гігабайтів тензорів активації в такій мережі займе більше часу , ніж відповідні обчислення, це призведе до значного простоювання апаратної частини для обчислень.

З перелічених аргументів, можемо зазначити, що вибір припадає на використання парадигми паралелізму за даними, оскільки він мінімізує використання мережевого трафіку, через обмін не гігабайтами тензорів активації, а лише усередненими градієнтами.

Але вибір даної парадигми не вирішує проблему оптимізації пляшкового горла повністю. В умовах використання вузлів з різними обчислювальними можливостями цей метод стискається з ще одною критичною проблемою, вона пов'язана з десинхронізацією роботи між вузлами.

1.2.2 Проблема гетерогенного обладнання

В розподілених системах усі обчислювальні кластери класифікують за апаратною та архітектурною ознакою на гетерогенні та гомогенні, кожен з цих класів напряму впливає на реалізацію роботи обчислювального кластеру.

Гомогенні системи - це системи, в яких усі робочі вузли є абсолютно ідентичними один одному, це включає однакові центральні процесори, графічні прискорювачі, кількість оперативної пам'яті та однакове з’єднання між всіма вузлами, яке забезпечить однакову пропускну здібність, все це дає кожному вузлу обчислювати однакові порції даних за один той самий час. Вище сказане, дає можливість використовувати найпростіші алгоритми для балансування навантаження.

Гетерогенні системи ж , навпаки, складаються з вузлів з різним обчислювальним потенціалом, тобто мають різну апаратну частину та архітектури, це призводить до того, що час виконання однакових задач на кожному вузлі буде різний. Також на це будуть впливати реальні умови, в яких використовуються подібні обчислювальні кластери, бо зазвичай вони можуть використовувати різні технології мережевого з’єднання, наприклад інтерфейси Ethernet або Wi-Fi.

З усього вище сказаного, можемо зробити висновок, що такі системи потребують більш складного підходу для розподілення навантаження між робочими вузлами, щоб не викликати простою більш потужних вузлів кластеру. Спроба застосувати класичні підходи, як для гомогенних систем, викличуть проблему, так звану “ефект відсталого вузла”.

1.2.3 Ефект відсталого вузла

Ефект відсталого вузла - це системна проблема в паралельних обчислюваннях, яка дуже гостро стає відчутною за використання стохастичного градієнтного спуску, бо загальний час виконання ітерації жорстко обмежується часом виконання найслабшим вузлом кластеру.

Пов'язано дане обмеження з роботою методу навчання нейронних мереж, бо він не може зробити глобальне оновлення ваг без отриманих результатів обчислень кожного вузла кластеру, це створює точку бар'єрної синхронізації, бо кожен вузол, який завершив свої обчислення, буде повинен перейти в режим очікування, та простоювати, поки всі інші вузли закінчать обробляти свої порції даних.

Вплив цього ефекту добре демонструється на прикладі паралельних обчислень за допомогою графічного прискорювача, наприклад Nvidia RTX 4060, та

центрального процесора AMD Ryzen 5 5600, коли зображення для обробки діляться навпіл. Наприклад, відеокарта опрацьовує свою частину за 0.5 секунд, але в той же час процесор вимагає 5 секунд. Це призводить до значного простою найпотужнішого модуля системи.

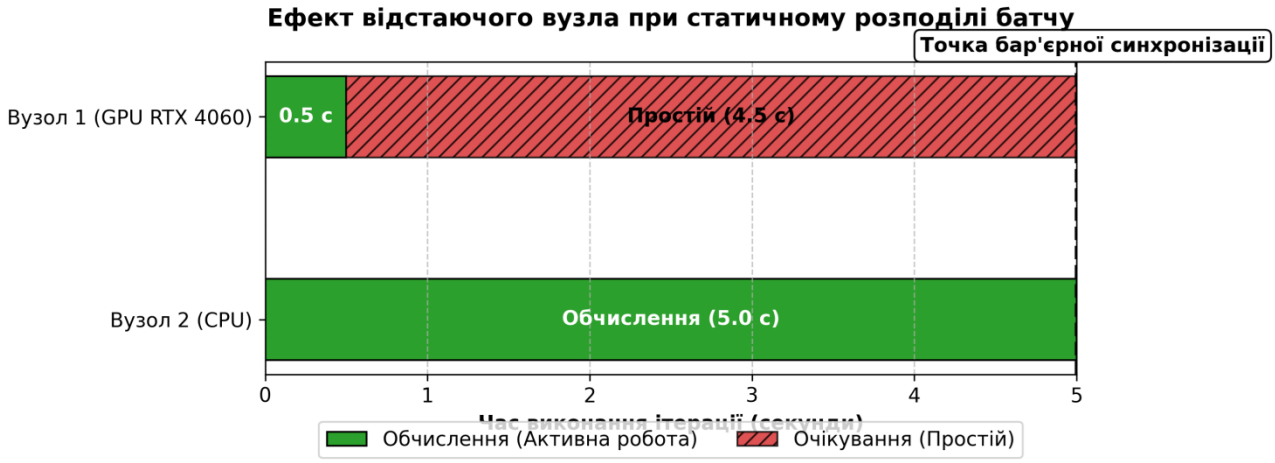

Рисунок — 1.2 Візуалізація ефекту відсталого вузла

Як видно з наведеної діаграми, використання підходу для гомогенних систем створює ситуацію, в якій високопродуктивний вузол, GPU RTX 4060, простоює значну частину часу, це призводить до глобальної деградації всієї системи обчислень.

Слідую-чи з вище сказаного, ефект відсталого вузла є головним бар'єром до масштабування систем обчислень в домашніх або офісних локальних мережах. Аналіз архітектурних та апаратних можливостей дає змогу зрозуміти, що для ефективної роботи розробленого програмного комплексу необхідно відмовитися від класичного статичного паралелізму, на користь динамічного балансування навантаження, це дозволить мінімізувати час простою та забезпечите більш синхронну роботу усієї системи обчислень.

1.3 Порівняльний аналіз існуючих рішень

На сьогоднішній день існує значна кількість готових рішень для паралельного навчання нейронних мереж, але всі вони розроблялися під різні умови роботи обчислювального кластеру, для аналізу обрано декілька найпопулярніших рішень. Кожне з них ми будемо аналізувати крізь призму умов нашої задачі, для цього було виділено декілька основних параметрів:

- підтримка гетерогенних обчислень;
- наявність вбудованого механізму для динамічного балансування;
- можливість низькорівневої інтеграції;
- витрати часу на обмін даними між вузлами.

Дані критерії допоможуть оцінити ефективність засобів, які будуть розглянуті далі, для вирішення поставленої проблеми. Для аналізу було обрано:

- PyTorch DistributedDataParallel (DDP);
- DeepSpeed;
- Federeted Learning.

1.3.1 PyTorch DistributedDataParallel

Алгоритм DDP працює за принципом сурової синхронізації, на початку ітерації навчання дані діляться на рівномірні частини та діляться між усіма вузлами обчислювальної системи. Кожен вузол незалежно виконує прямий та зворотній прохід, обчислені локальні градієнти передаються по мережі для їх об’єднання в глобальний градієнт, під час цього всі вузли зупиняють свою роботу

в очікуванні на оновленні ваги моделі. Якщо оцінювати даний алгоритм за нашими критеріями, ми маємо наступні результати:

- підтримка гетерогенних обчислювань є низькою, бо фреймворк архітектурно розрахований на гомогенні системи;
- динамічне балансування відсутнє, алгоритм ділить дані на рівні порції, що викличе ефект відсталого вузла і призведе до деградації обчислювальних можливостей всієї системи;
- витрати на обмін даними між вузлами є високоефективними, але лише для мереж спеціалізованого стандарту, як InfiniBand, для локальних мереж обмін даними кожну ітерації викличе перевантаження локальної мережі;
- низькорівнева інтеграція є складною, хоча фреймворк використовує ядро, яке реалізовано на мові програмування C++ (LibTorch), механізм саме розподіленого навчання є повністю інкапсульованим в Python-обгортки.

1.3.2 DeepSpeed

DeepSpeed - це передовий фреймворк із відкритим кодом, створений компанією Microsoft. Він створений для підвищення ефективності та оптимізації розподіленого навчання. Головною технологією є ZeRo (Zero Redundancy Optimizer), вона допомагає усувати надмірне використання пам’яті у кожному вузлі, замість звичайного дублювання, ZeRo динамічно фрагментує стані оптимізатора, градієнти, параметри моделі між усіма пристроями кластеру. Під час роботи з масивними обчислювальними графами, модуль ZeRo-Offload дозволяє залучати для обробки ресурси центрального процесора та оперативної пам’яті для зменшення навантаження на відео-пам’ять графічного процесора. Незважаючи на всі свої переваги для навчання гігантських моделей, DeepSpeed

має декілька критичних обмежень, оцінюємо даний фреймворк за нашими критеріями:

– підтримка гетерогенних обчислювань: незважаючи на те, що технологія ZeRo-Offload залучає процесор до роботи, але вона використовує його лише, як буфер пам’яті, а не окремий обчислювальний вузол, базова логіка роботи все одно лишається орієнтовною на гомогенні кластери з графічних процесорів;

– динамічне балансування: DeepSpeed використовується виключно для оптимізації використання пам’яті, він позбавлений можливості динамічно розподіляти порції даних між різними вузлами для обчислення, що призводить в гетерогенних системах до простоювання обчислювальних можливостей;

– витрати на обмін даними між вузлами: Через фрагментацію частин моделі між вузлами, під час кожного проходу, як прямого та зворотного, вузли будуть змушені обмінюватися даними, тому фреймворк потребує використання надшвидких інтерфейсів, як InfiniBand;

– низькорівнева інтеграція є неможливою, оскільки фреймворк розроблявся, як високорівнева надбудова над PyTorch, і він виключно функціонує в python-середовищі. Це повністю блокує його можливість інтеграції в унікальні та високопродуктивні C++ системи, які базуються навіть на LibTorch.

1.3.3 Federeted Learning

Метод федеративного навчання працює асинхронно, центральний сервер розсилає кожному вузлу копію моделі. На відмінну від інших методів синхронізація при федеративному навчанні відбувається не кожну

ітерацію, а раз на декілька епох. Кожен вузол виконує обчислення, наприклад 3 епохи, а тільки потім відправляє результати на сервер , також в цьому випадку результат буде не градієнти, а вже оновлені ваги моделі. Головний вузол використовує алгоритм усереднення для всіх отриманих результатів, але є один нюанс, якщо якийсь вузол був занадто слабкий і не встиг надіслати дані у відповідний час, сервер його ігнорує і продовжує роботу з вузлами, які надіслали результати вчасно. Оцінка за критеріями:

– архітектура спеціально створена для підтримки гетерогенних систем, вона дозволяє об’єднувати різнорідні пристрої, від смартфонів до серверів;

– динамічне балансування реалізовано асинхронністю та відкидання відсталих вузлів, цій метод запобігає простоюванню швидкого обладнання;

– через використання асинхронного методу навчання, рідкісну синхронізацію та пересилання мережею повних ваг моделі навчання проходить значно повільніше ніж при використанні методу синхронного градієнтного спуску, це викликає критично високу затримку на моменті збіжності моделі;

– низькорівнева інтеграція відсутня, бо сам фреймворк реалізовано переважно високорівневими засобами абстракції і погано підходить до створення легковагових та оптимізованих додатків для локальних мереж на базу C++.

1.4 Висновки до першого розділу

У першому розділі проведено детальний аналіз існуючих методів розпаралелення алгоритмів машинного навчання та виявлено ключові проблеми, що виникають при їх застосуванні у гетерогенних обчислювальних кластерах. Встановлено, що найпоширеніші Enterprise-рішення (PyTorch DDP, DeepSpeed)

оптимізовані переважно для гомогенного обладнання та спеціалізованих високошвидкісних мереж (наприклад, InfiniBand). Це робить їх неефективними у стандартних локальних мережах через виникнення ефекту відсталого вузла та мережевого пляшкового горлечка. На основі порівняльного аналізу обґрунтовано необхідність розробки власної програмної системи мовою C++, яка базуватиметься на парадигмі паралелізму за даними з механізмами динамічного балансування навантаження, що дозволить ефективно використовувати обчислювальні потужності різнорідного обладнання без критичних затримок.

2. ПРОЄКТУВАННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ РОЗПАРАЛЕЛЕННЯ АЛГОРИТМІВ

2.1 Загальна структура системи

Комп'ютерна система розпаралелення алгоритмів розроблена з орієнтацією на алгоритми машинного навчання з реалізацією мовою C++. Розроблена система базується на парадигмі паралелізму за даними, що передбачає наявність повної ідентичної копії нейронної мережі та набору даних для навчання на кожному обчислювальному вузлі кластеру. Розпаралелення підлягає лише процес навчання, а саме кожен вузол обробляє лише виділену йому підмножину дата-сету, після завершення математичних операцій, їх результати синхронізуються для оновлення моделі та глобального стану системи. Даний підхід дозволяє уникнути передачі мережею між вузлами сирих даних під час навчання, що мінімізує мережеві затримки.

Архітектурно розроблена система є строго ієрархічною і поділяється на три взаємодіючи між собою рівні (рис. 2.1):

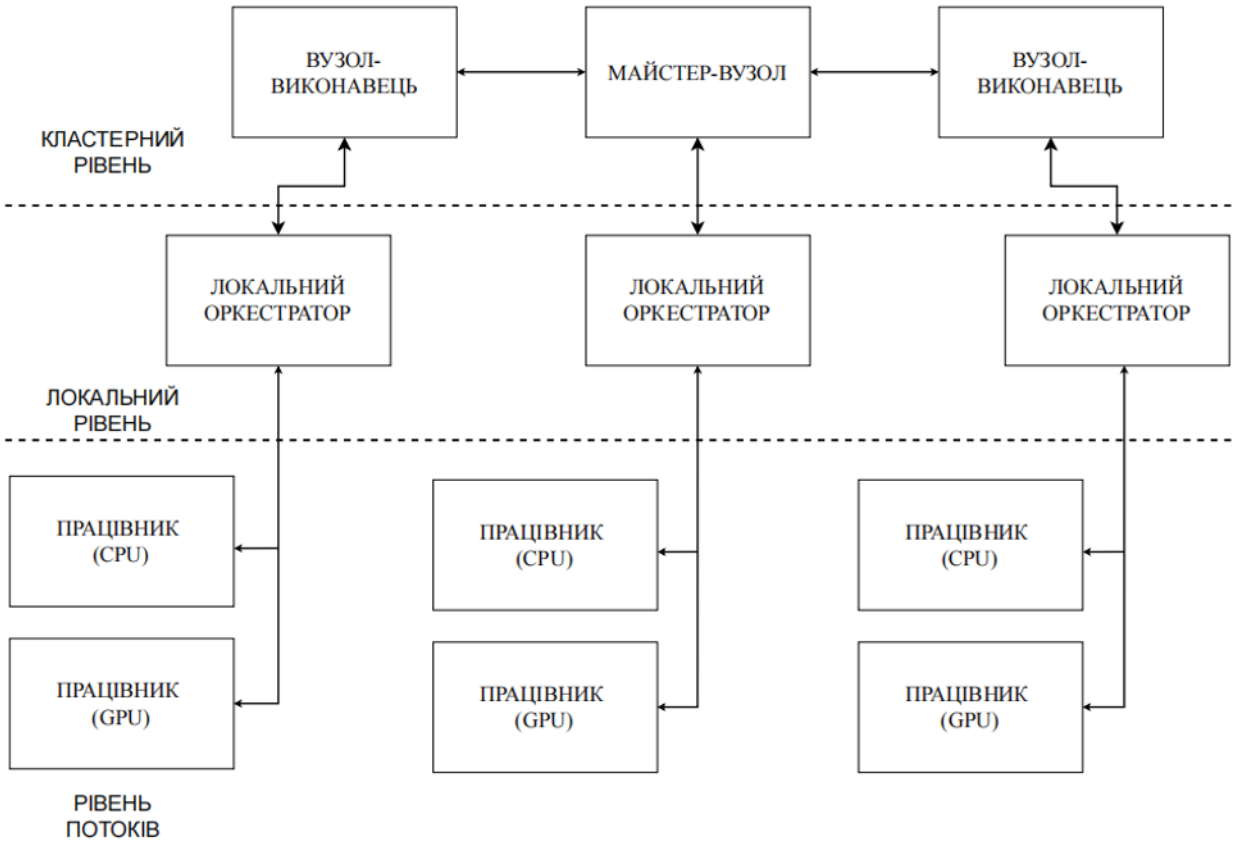

Рисунок 2.1 — Структурна схема роботи розробленої системи

Згідно структурної схеми, відповідальність за роботу системи розмежовується між окремими модулями відповідно за рівнями. Така архітектурна будова забезпечує відповідні переваги

– кластерний рівень - є відповідальним за глобальну координацію між вузлами та балансування навантаження в мереж;

– локальний рівень - управляє системними потоками та розподіленням даними та навантаженням між CPU та GPU на одному вузлі;

– рівень багатопоточності (обчислювальний рівень) - відокремлює низькорівневу математику, відповідає за паралелізм тензорних операцій між ядрами процесора або cuda-ядрами відео-прискорювача;

– високий рівень абстракції, бо кожен модуль працює незалежно від модуля, вищого за рівнем. Наприклад, кластерний рівень оперує лише

мережевими вузлами, не маючи інформації про наявність графічного прискорювача у вузлів;

– подвійне балансування навантаження на систему, через наявність двох окремих балансувальників навантаження в системі, мережевий та локальний, досягається висока продуктивність системи розподіленого навчання, за рахунок уникання ефекту відсталого вузла;

– бар’єрна синхронізація, завдяки її використанню досягається безпечне поєднання градієнтів між усіма вузлами системи, без ризику виникнення стану гонитви або взаємних блокувань.

Система за даною структурною схемою дозволяє гнучко масштабувати процес навчання, від запуску на одному комп’ютері до створення гетерогенного кластеру в локальний мережі.

2.1 Кластерний рівень

Кластерний рівень є найвищим абстрактним рівнем в розробленій системи. Його діяльність полягає в розширені навчання на декілька окремих фізичних пристроїв (вузлів), які поєднанні між собою локальної мережею. На цьому рівні використовується топологія Master-Worker, вона керує процесом розподіленого навчання. Кожен вузол працює з локальною копією моделі та дата-сету, це дозволяє уникнути пересилання великих об’ємів даних мережею між вузлами, натомість вузли в процесі навчання обмінюються лише результати обчислень (градієнти моделі) та мета даними, які потрібні для правильного функціонування системи та мережевого балансувальника, даний підхід дозволяє запобігти перенавантаженню мережевого каналу.

Основними складовими кластерного рівня є:

1. Глобальна організація: для коректного навчання, з математичного боку, кожен вузол має починати з однаковими станами, це досягається за рахунок першої ініціалізації та першого рукостискання між майстер-вузлом та працівником-вузлом, під час яких майстер відправляє глобальні параметри та криптографічне зерно. Це є гарантією побітової ідентичності між моделями на різних вузлах та однакового порядку перемішування даних між епохами навчання.

2. Мережевий балансувальник навантаження: оскільки вузли в кластері можуть мати різну обчислювальну потужність, розподіл даних за статичним принципом призвів би до простою системи, через очікування найповільнішого вузла в системі. Для вирішення даної проблеми було створено мережевий балансувальник, який вирішує дану задачу шляхом динамічної генерації меж, за якими кожен вузол бере дані для навчання. Генерація проходить на основі телеметрії (часу обробки попереднього сегменту даних), зібраної з кожного вузла, балансувальник вираховує співвідношення часу та продуктивності. Після, кожному вузлу відправляються, замість даних, лише індекси початку та кінця діапазону, за таким принципом швидший вузол отримує більш великий діапазон, повільніший навпаки, завдяки чому всі вузли майже одночасно завершують роботу над своїми сегментами даних.

3. Обмін даними реалізовано на базі високопродуктивної бібліотеки ZeroMQ, процес кожної ітерації на мережевому рівні можна розділити на два основні етапи:

   а) збір результатів (REQ/REP) з кожного вузла кластеру: після закінчення обчислень, вузли відправляють свої локальні градієнти до головного вузла. Мережевий оркестратор розраховує глобальні градієнти за зваженим середнім;

b) надсилання оновлених всім робочим вузлам (PUB/SUB): коли головний вузол виконає крок оптимізації, усереднені градієнти розсилаються широкомовно вузлам кластеру, далі весь кластер почне працювати із синхронізованим станом.

4. Контрольоване переривання: через розробку системи з використанням жорсткої бар'єрної синхронізації, раптова втрата зв'язку з одним із вузлів кластеру могла б призвести до вічного зависання системи, рішення для цієї проблеми використання мережевих тайм-аутів на базі сокетів ZeroMQ. Якщо час очікування результатів від іншого вузла є вищим за встановлений проміжок часу, головний вузол ініціює процедуру контрольованого переривання навчання: мережеві з'єднання розриваються, всі використовувані ресурси звільняються, а процес навчання зупиняється, даний підхід захищає систему від непередбачуваної поведінки мережі та дає змогу користувачу перезапустити навчання з нуля з активними робочими вузлами.

2.3 Локальний рівень

Локальний рівень системи відповідає за розподілення навчання в межах одного фізичного пристрою (комп'ютера або ноутбука), який має центральний процесор та графічний прискорювач (гетерогенна архітектура). Використовується архітектурний шаблон Master-Worker, для забезпечення ізольованого обчислення та ефективності синхронізації градієнтів.

Локальний рівень логічно поділяється на три основні модулі:

1. Головний оркестратор: це головний модуль локальної системи він відповідає за:

a) управління даними: отримання глобального масиву даних та його динамічний розподіл між працівниками, на основі метрик отриманих від балансувальник;

b) поєднання результатів обчислень: після завершення роботи працівників, збирає результати їх обчислень (градієнти) та виконує їх зважене усереднення, також на основі метрик, за якими глобальна порція даних розподілялася між працівниками;

c) глобальне оновлення ваг моделі: головний оркестратор за допомогою єдиного екземпляра оптимізатор, після об'єднання градієнтів, виконує крок оптимізації та синхронізує ваги між моделями працівників. Використання єдиного екземпляру оптимізатора виключає проблему десинхронізації через внутрішні стани, які використовуються в деяких оптимізаторах.

2. Ізольовані обчислення: працівники системи є ізольованими від інформації про загальний стан системи. Кожен працівник місить власну копію моделі, яка розміщується у відповідному типі пам'яті. Задача працівників є, за отриманими даними, виконати прямий та зворотній, отримання градієнтів і сигналізація оркестратору, що результати готові до поєднання.

3. Адаптивний балансувальник навантаження: ключовий модуль локального рівня системи, він використовується для динамічного розподілу навантаження між працівниками. Балансувальник отримує час обчислення попередньої порції даних кожним працівником, аналізує отриманні дані і повертає відповідний за ними розподіл навантаження. Алгоритм використовує метод "розділених зон нечутливості":

a) зона зміни: при незначному розбіжності в часі виконання обчислені, балансувальник робить плавне зміщення в бік швидшого пристрою;

b) зона критичного відставання: Якщо один працівник (зазвичай CPU) за часом виконання, стає причиною критичного падіння продуктивності, де різниця між пристроями стає більше за відповідний поріг, балансувальник повністю відключає відповідний вузол, тимчасово переводячи систему для обчислень на швидшому працівнику.

2.4 Рівень багатопоточності

Ефективність паралельного навчання залежить від правильного використання ресурсі, рівень багатопоточності в системі поділяється на два типи: управління потоками на рівні системи засобами мови C++ та внутрішній паралелізм базового фреймворку LibTorch.

2.4.1 Управління потоками на рівні розробленої системи

На цьому рівні за допомогою стандартних засобів управління потоками мови C++ вирішується задача глобального управління гетерогенними пристроями. Розроблена система не займається безпосередньо математичними операціями, а виконує роль диспетчера, для забезпечення виконання даної задачі, система використовує стандарти багатопоточності(std::thread, std::mutex, std::condition_varible). Головним завдання для системи на цьому рівні є забезпечення:

– асинхронний старт обчислень: після розрахунків балансувальника та розділення масиву даних між працівниками, основний потік (Оркестратор) створює незалежні потоки для центрального процесора та графічного прискорювача;

– бар’єрна синхронізація: передача даних через шину PCIe та поєднання градієнтів потребують суворої послідовності, тому для синхронізації використовуються: м’ютекси, які забезпечують ексклюзивний доступ до пам’яті, та умовні змінні, які дозволяють головному потоку зупинити своє виконання та очікувати сигналу від потоків працівників про завершення обчислень та отримання результату;

– ізоляція пам’яті: Стандарти C++ гарантують, що потоки не перетнуться в пам’яті до моменту виклику функції поєднання, що робить неможливим появи стану гонитви (Data race) і не призведе до пошкодження параметрів моделі.

2.4.2 Внутрішній паралелізм LibTorch

В той час, як розроблена система керує розподілом масиві даних між пристроями, розпаралелювання відповідних математичних операцій передається ядру LibTorch. Фреймворк автоматично оптимізує розрахунки, застосовуючи різні стратегії залежно від апаратної частини:

– центральний процесор (CPU): Фреймворк керує пулом потоків операційної системи, для цього він використовує два підходи. Міжопераційний паралелізм, який дозволяє одночасно обчислювати незалежні гілки нейронної мережі, та внутрішньо операційний паралелізм, який в свою чергу розбиває одну велику задачу на фрагменти і завантажує всі вільні ядра;

– графічний прискорювач (GPU): Фреймворк напряму працює з архітектурою CUDA, завдяки цьому він не створює системні потоки, а транслює математичні операції напряму у спеціальні мікро-задачі, які

асинхронно виконуються на тисячах мультипроцесорах відеокарти (CUDA-ядра).

Завдяки адаптивності LibTorch, досягається максимальна ефективність системи, головний потік забезпечує безперервне завантаження нових даних для обчислень, в той же час LibTorch гарантує для кожного пристрою повне використання ресурсів без простою.

2.5 Проєктування програмного інтерфейсу користувача

Розроблену систему можна класифікувати, як інструментальний фреймворк, важливим етапом проєктування виступає створення програмного інтерфейсу користувача (API). Головним архітектурним викликом API є приховування складної розподіленої мережевої взаємодії, низькорівневе балансування навантаження та багатопоточності, від кінцевого користувача системи (розробника моделей).

За концепцією розроблений інтерфейс системи проєктувався з використанням інкапсуляції та інверсії управління, які базуються на інкапсуляції низькорівневої логіки, єдине джерело конфігурації для системи, асинхронний моніторинг за допомогою зворотних викликів.

Інкапсуляція низькорівневої логіки: головна ідея, що кінцевий користувач не повинен оперувати інструментами, котрі використовуються для розпаралелення алгоритму, наприклад: мережеві сокети, системні пули потоків та м’ютекси. API передбачає, що користувачу достатньо передати в систему абстрактні сутності: об’єкт будови нейронної мережі та дата-сет, набір даних для навчання. Після виклику відповідної функції, оркестратор бере на себе повне управління з створення мережевого з’єднання, балансування навантаженням на апаратну частину та синхронізацію навчання між декількома вузлами.

Єдине джерело конфігурації: одною з важливіших проблем, які можуть виникати під час розпаралелення навчання нейронних мереж, є розбіжність налаштувань навчання. Щоб запобігти ситуаціям, коли вузли використовують різний шаг навчання або розмір батчу, система використовує лише виділені гіперпараметри, які були ініціалізовані на майстер-вузлі, система бере на себе відповідальність серіалізувати їх та доставити до кожного працівника кластеру.

Асинхронний моніторинг через зворотні виклики: оскільки розроблена система виносить обчислення в паралельні потоки, синхронний метод повернення результатів є не ефективним, з огляду для забезпечення прозорості навчання, API спроєктований для використання зворотних викликів (callback-функцій). Під час циклу навчання система збирає комплексну структуру метрик, вона поєднує дані з трьох рівнів:

– рівень багатопоточності: поточна помилка, епоха, швидкість навчання;

– локальний рівень: час виконання ітерації, відсоток розподілення батчу, поточна пропускна здатність;

– кластерний рівень: частка даних вузла у загальному масиві, мережевий ідентифікатор.

Вся вище, перерахована інформація, передається користувачеві у вигляді єдиної структури даних, завдяки цьому підходу, розроблена система стає незалежною від візуалізації телеметрії, даючи змогу користувачу, як саме обробляти отриману інформації, зберігати в файл, інтегрувати в графічні даш-борди або виводити у консоль.

Спираючись на вище описане проєктування API, для користувача використання гетерогенного кластеру для розпаралелення алгоритму машинного навчання стає таким ж зрозумілим, як і запуск стандартного послідовного алгоритму на одному пристрої.

2.6 Висновки до другого розділу

У даному розділі було описано загальну концептуальну архітектуру системи, яка розробляється, для розпаралелення алгоритмів машинного навчання. Було детально розглянуто трирівневу ієрархію системи та алгоритми балансування навантаження на мережевому і локальному рівнях, які дають змогу запобігати ефекту відсталого вузла при обчисленнях за допомогою гетерогенного обладнання.

3. РЕАЛІЗАЦІЯ

У даному розділі детально розглядається програмна реалізація розробленої комп'ютерної системи для розпаралелення алгоритмів машинного навчання. Система була розроблена мовою програмування C++ (Стандарт C++17). Використання стандарту C++17 зумовлено потребою в використанні сучасних можливостей керуванням пам'яттю (розумні вказівники std::unique_ptr, std::shared_ptr), багатопотоковості (std::thread, std::condition_variable, std::mutex), та даних опціонального типу (std::optional).

Основним інструментом для обчислення тензорних операцій та побудови обчислювальних графів є LibTorch, вона є нативним C++ API для найпопулярнішого фреймворку PyTorch. Мережевий обмін між незалежними вузлами кластеру було побудовано за допомогою бібліотеки обміну повідомленнями ZeroMQ (C++ обгортка zmq.hpp), з її допомогою забезпечено мінімальні затримки при обміні даними між робочими вузлами. Проєкт збирається за допомогою системи збірки CMake, що сприяє швидкому розгортанню системи.

3.1 Реалізація локального рівня та модуля розподілення навантаження

Локальний рівень системи є відповідальним за низькорівневе управління процесом обчислення в межах одного фізичного пристрою (персональний комп'ютер або ноутбук). Головною точкою входу є клас LocalTrainer, даний клас виконує роль локального оркестратору, в той же час самі математичні операції передаються в роботу об'єктам класу Worker. Для збереження інкапсуляції та дотримання принципів об'єктно-орієнтовного програмування зокрема, принципу єдиної відповідальності, даний рівень було поділено на три основні компоненти, діаграма класів локального рівня наведено на рисунку 3.1.

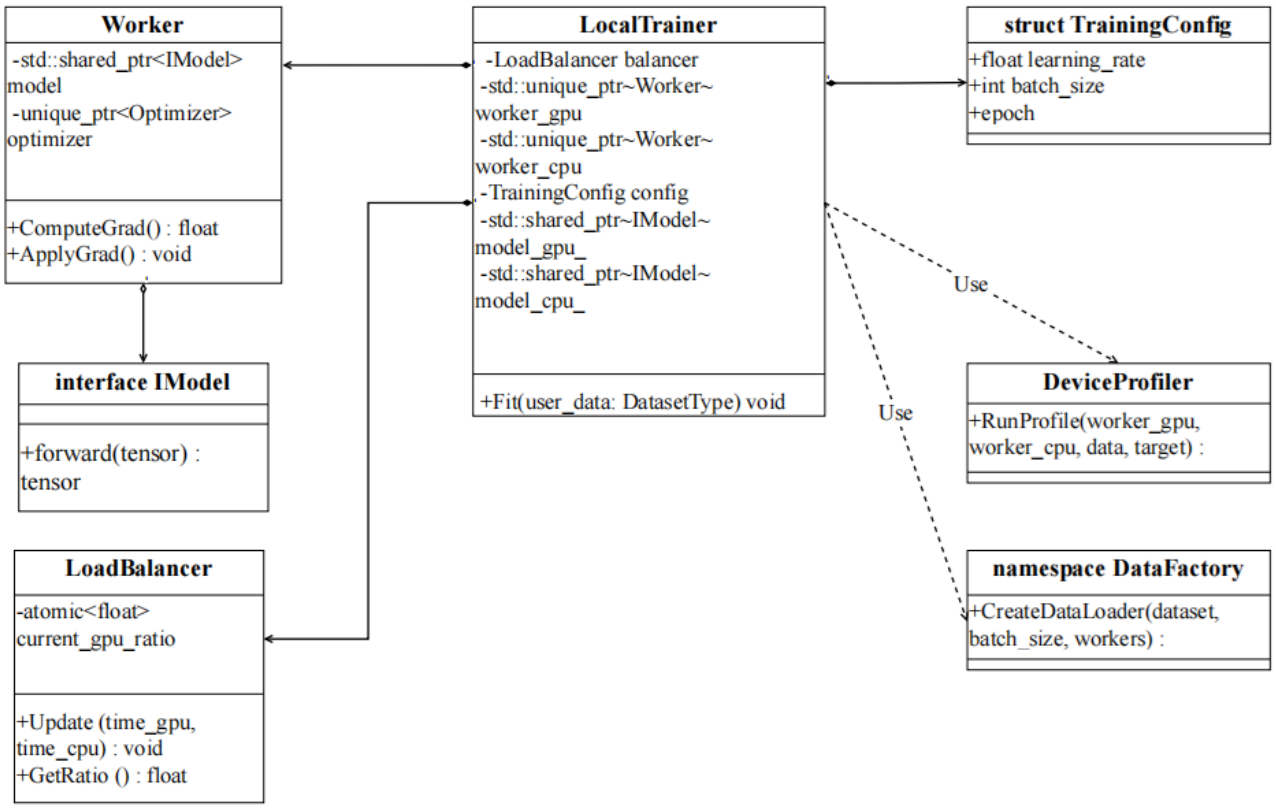

Рисунок — 3.1 UML-діаграма класів локального рівня системи

Як видно з зображеного з рисунка 3.1, структура локального рівня створена за монолітною будовою, де точкою управління є клас LocalTrainer, який за допомогою публічного методу Fit() запускає глобальний процес навчання на одному пристрої. Його взаємодія з іншими модулями локального рівня поділяється за таким принципом:

1. Суворі зв’язки: На UML-діаграмі вони позначаються чорним ромбом з суцільною лінією, вони зв’язують LocalTrainer з об’єктами, для яких він виступає власником життєвого циклу:

   а) обчислювачі Worker: LocalTrainer ініціалізує два екземпляри відповідного класу (worker_gpu, worker_cpu) за допомогою розумних вказівників std::unique_ptr, які використовуються для гарантованого автоматичного знищення об’єктів та очищення пам’яті при виході з зони видимості або знищенні об’єкта-власника;

b) модуль LoadBalancer – це апаратний балансувальник (balancer), який є невід'ємною частиною усього процесу навчання, з його допомогою динамічно змінюється навантаження між CPU та GPU;

c) TrainingConfig – це структура, яка використовується для інкапсуляції всіх гіперпаратметрів навчання (learning_rate, epochs, batch_size, optimizer_type та інші).

2. Розподілене володіння: На діаграмі дані зв'язки позначаються, як білий ромб з суцільною лінією, відповідні зв'язки ілюструють відносини між Worker та інтерфейсом нейронної мережі IModel. Через те, що об'єкти (model_gpu_, model_cpu_) нейронних мереж створюються в конструкторі LocalTrainer, та є потреба щоб об'єкти працівників отримали до них доступ, але без глибокого копіювання важких вагових матриць, використовується std::shared_ptr, це забезпечує безпечний спільний доступ до моделей.

3. Слабкі зв'язки: вони позначаються, як пунктирні лінії зі стрілкою, вона вказує на об'єкти, які локальний оркестратор не зберігає у пам'яті весь час, а лише викликає їх у певні моменти часу:

a) DeviceProfiler використовується для виклику статичного методу RunProfile() для проведення стартового визначення обчислювальних потужностей між центральним процесором та графічним прискорювачем;

b) простір імен DataFactory викликається для створення фонових потоків (DataLoader), для розбиття користувацького дата-сету згідно batch_size та для завантаження даних з жорсткого носія даних у відповідну пам'ять (RAM, VRAM), де розміщена модель нейронної мережі.

3.1.1 Апаратний обчислювач (клас Worker)

З вище сказаного тексту, зрозуміло, що безпосередньо виконанням матричних операцій займається клас локального рівня Worker. Для коректної роботи системи в гетерогенному середовищі, кожен об’єкт даного класу маж свій екземпляр алгоритму оптимізації (std::unique_ptr<Optimizer>), це рішення обумовлене тим, що якби системні потоки мали змогу використовувати один спільний оптимізатор , це призвело би до руйнування внутрішніх станів оптимізатора (моменти у алгоритмі Adam) через конкуруючий доступ до пам’яті. Однією з ключових складностей при створенні класу Worker для розпаралелення навчання стала необхідність розділення стандартного циклу машинного навчання на дві незалежні одна від одної фази, програмну реалізацію наведено у рисунку 3.2.

```
float ComputeGrad(torch::Tensor data, torch::Tensor target) {
        model->train();
        optimizer->zero_grad();
        torch::Tensor prediction = model->forward(data);
        torch::Tensor loss = loss_function(prediction, target);
        loss.backward();
        return loss.item<float>();
}
void ApplyGrad() {
        if (config.clip_grad_norm.has_value()) {
                torch::nn::utils::clip_grad_norm_(model->parameters(), config.clip_grad_norm.value());
        }
        optimizer->step();
}
```

Рисунок 3.2 — Демонстрація розділення циклу навчання на дві фази

Як видно з публічного методу, класу Worker, ComputeGrad виконує алгоритм зворотного поширення помилки (loss.backward()), проте навмисно не викликає optimizer→step(), це розділення необхідно для того, щоб оркестратор мав проміжок часу між обчислення та застосуванням градієнтів до ваг моделі, LocalTrainer використовує цей проміжок для збору градієнтів з CPU та GPU, виконує зважене усереднення градієнтів, лише після синхронізації їх викликається метод ApplyGrad().

3.1.2 Алгоритм динамічного балансування (клас LoadBalancer)

Оскільки час, який потребують центральний процесор та графічний прискорювач для обчислень, є кардинально різним, статичний поділ даних між ними призвів би до неефективного використання ресурсів: швидше обладненя більшість часу простоювало б в очікуванні на завершення обчислень від повільнішого вузла. Для вирішення цієї проблеми було розроблено модуль LoadBalancer, який керує атомарною змінною current_gpu_ratio.

Робота балансувальника створена за алгоритмом зі зворотнім зв’язком, кожну ітерацію він аналізує час, який витратили на обчислення CPU та GPU, та на його основі приймає рішення про перерозподіл даних на наступну ітерацію навчання. Блок-схему розробленого алгоритму наведено на рисунку 3.3

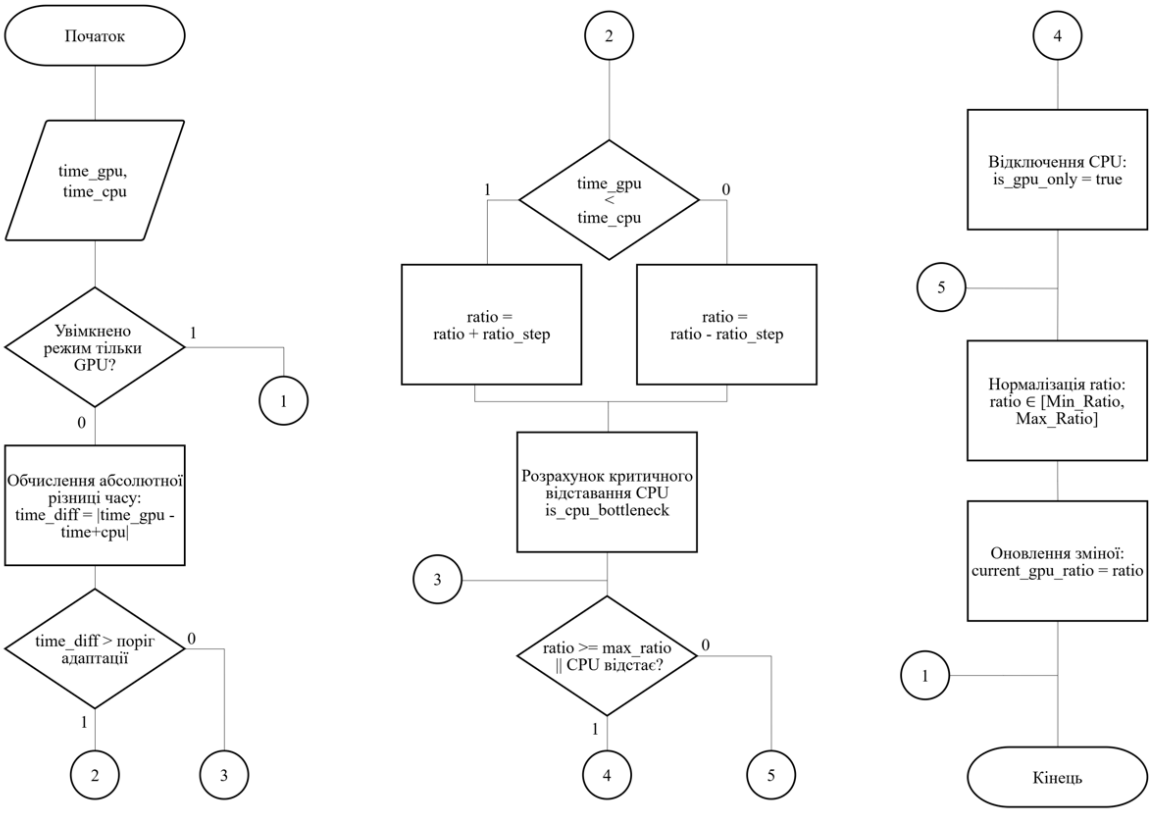

Рисунок 3.3 — Блок-схема алгоритму балансування навантаження

За наведеною блок-схемою, алгоритм балансування навантаження поділяється на декілька логічних етапів: перевірка поточного стану системи, зона адаптації поділу даних між CPU та GPU, визначення критичного відставання та фінальну нормалізацію розрахованого коефіцієнту поділу навантаження. Програмну реалізацію даного алгоритму наведено на рисунку 3.4.

```
void LoadBalancer::Update(double time_gpu, double time_cpu) {
    if (is_gpu_only) return;
    double time_diff = std::abs(time_cpu - time_gpu);
    float ratio = current_gpu_ratio.load();
    if (time_diff > config.min_time_diff_ms_adjust) {
        if (time_gpu < time_cpu) {ratio += config.ratio_step;}
        else {ratio -= config.ratio_step;}
    }
```

```
        bool is_cpu_bottleneck = (time_cpu > time_gpu * config.cpu_slowdown_factor) &&
                ((time_cpu - time_gpu) > config.min_time_diff_ms_kill);
        if (ratio >= config.max_gpu_ratio || is_cpu_bottleneck) {
                std::cout << "[Balancer] CPU is slow. Turn on GPU only." << std::endl;
                is_gpu_only = true;
                test_counter = 0;
        }
        ratio = std::max(config.min_gpu_ratio, std::min(config.max_gpu_ratio, ratio));
        current_gpu_ratio.store(ratio);
}
```

Рисунок 3.4 — Демонстрація програмної реалізації алгоритму балансування навантаження

Як видно з наведеної демонстрації програмної реалізації та блок-схеми алгоритму, робота починається з перевірки стану системи, за допомогою змінної is_gpu_only, яка вказує в якому режимі триває навчання, процесор разом з відеокартою або лише відеокарта. Якщо центральний процесор вже було усунуто від обчислень на попередніх ітераціях циклу навчання через його низьку продуктивність, метод негайно повертає керування локальному оркестратору, це дозволяє мінімізувати витрати головного потоку на виконання непотрібних перевірок та операцій, що звільняє процесорний час для наступних ітерацій.

Натомість при активному гібридному режимі навчання, алгоритм переходить до розрахунку абсолютної різниці між часом виконання попереднього батчу на CPU та GPU (time_diff). Для прийняття рішення, щодо розділення даних на наступну ітерацію, використовується концепція зон нечутливості. На системний планувальник потоків впливають різні сторонні фактори:

- фонові системні процеси;
- переривання;
- температурний тротлінг.

Через вище, сказані фактори, час обробки одного батчу може коливатися протягом усього навчання моделі. Наступним кроком алгоритм порівнює time_diff з встановленим конфігураційним порогом (min_time_diff_ms_adjust), якщо різниця в часі менша за поріг, коефіцієнт розподілу даних батчу між процесором та графічним прискорювачем (ratio) не змінюється, таке рішення обумовлене захистом процесу навчання від нескінчених мікроколивань, лишу у випадку, коли різниця в часі стає більш відчутною, змінна ratio лінійно змінюється на конфігураційний крок (ratio_step), використання лінійного зсуву, запобігає різким стрибкам навантаження та гарантує плавний перехід.

Найбільш критичним елементом алгоритму є механізм жорсткого відключення CPU. Логічний вираз, результат якого записується в зміну is_cpu_bottleneck, розроблено, щоб аналізувати одразу два фактори. Відносне відставання, перевіряє у скільки разів CPU витрачає більше часу, ніж GPU (cpu_slowdown_factor), але цієї умови недостатньо для коректного відключення процесора від навчання, якщо батч був занадто малим, та відеокарта обчислила його за 1мс, а CPU витратив 3 мс, в сухих числах процесор відстає у три рази, але витрата у 2 мс не є критичною для системи. Тому створено другу умову (min_time_diff_ms_kill), вона спрацьовує лише в момент, коли абсолютна затримка завдає реальної шкоди процесу навчання. При спрацюванні обох умов, балансувальник активує пропорець is_gpu_only та обнуляє змінну test_counter, скидання лічильника потрібно, щоб система раз на декілька ітерацій перевіряла швидкодію процесора, це забезпечує самовідновлення алгоритму.

На фінальному етапі, алгоритм нормалізує відкориговане значення за допомогою композиції стандартних функцій std::max та std::min, це забезпечує безпечний доступ до пам’яті, без використання відповідного обмеження алгоритм міг би розрахувати значення індексу поділу вище за 1.0 (100%), це призвело б до помилки сегментації при спробі нарізати тензор LibTorch поза його фізичні межі.

Хоча структура системи, гарантує, що метод Update буде викликатися в момент, коли обчислювачі вже закінчили свою роботу, змінна current_gpu_ratio все одно реалізована, як атомарний тип (std::atomic<float>), це рішення обумовлене захистом системи, в майбутньому вона може стати потрібною для фонового логування, тому використання атомарних методів load та store гарантує, що паралельний процес, який звернеться до цієї змінної, отримає значення без необхідності зупиняти головний потік за допомогою std::mutex.

3.1.3 Реалізація прихованої багатопотокової бар'єрної синхронізації

Після розподілення навантаження за допомогою локального балансувальника, система має забезпечувати паралельне обчислення матричних операцій на центральному та графічному процесорах. Найочевидніший підхід є створення системних потоків, але при створенні нових потоків кожну ітерацію та їх знищення є неефективним підходом, бо життєвим циклом потоку керує системне ядро, та при постійних перемиканнях створюються великі витрати часу, в свою ж чергу їх час може перевищувати час, який витрачається на розрахунок математичних операцій.

Для уникнення цієї проблеми було реалізовано в класі LocalTrainer розроблено статичний пул потоків із стійкою бар'єрною синхронізацією, в основі, якої лежать умовні змінні (std::condition_variable) та м'ютекси (std::mutex). Ініціалізацію потоку для GPU наведено на рисунку 3.5.

```
void InitThreadPool() {
    thread_gpu = std::thread([this]() {
        torch::DeviceGuard device_guard(torch::Device(torch::kCUDA, 0));
        while (true) {
            std::unique_lock<std::mutex> lock(mtx);
```

```
                cv_start.wait(lock, [this] { return gpu_has_work && (state == State::Working || state ==
State::Stop); });
                if (state == State::Stop) break;
                gpu_has_work = false;
                lock.unlock();
                auto start = std::chrono::high_resolution_clock::now();
                auto gpu_loss = worker_gpu->ComputeGrad(shared_gpu_data, shared_gpu_target);
                torch::cuda::synchronize();
                auto end = std::chrono::high_resolution_clock::now();
                time_gpu = std::chrono::duration<double, std::milli>(end - start).count();
                lock.lock();
                current_gpu_loss = gpu_loss;
                workers_done++;
                if (workers_done == 2) cv_done.notify_one();
}});}
```

Рисунок 3.5 — Програмна реалізація пулу потоків та бар’єрної синхронізації GPU

З наведеної реалізації можемо виділити, що алгоритм базується на об’єкті cv_start умовної змінної, після одного запуску при старті програми, потік переходить до стану сну, через використання нескінченого циклу та виклику wait(lock), коли система перевела потік до стану сну, він не споживає ресурсів CPU, в той же час локальний оркестратор нарізає нову порцію даних, по закінченню він викликає метод cv_start_notify_all(), що миттєво виводить зі стану сну потоки CPU та GPU для страта паралельної обробки нарізаних даних.

Важливою деталлю в цьому алгоритмі є виклик torch::cuda::synchronize(), він потрібен, бо за стандартами архітектури CUDA, усі звернення до графічного процесора встають у власну апаратну чергу, після цього керування повертається до системного потоку. Якби не було відповідного виклику, таймер, який рахує час виконання матричних множень зберіг лише час надсилання команди для початку обчислень, а не фізичний час самих розрахунків. Це викликало би отримання

модулем LoadBalancer хибних даних для розрахунку коефіцієнту поділу даних, що зруйнувало би логіку балансування. При виклику torch::cuda::synchronize() , потік очікує завершення всіх операції у відео-пам’яті, тільки після цього захоплює м’ютекс, збільшує лічильник працівників, які завершили обчислення (workers_done) та повідомляє головний потік про готовність до наступної ітерації (cv_done.notify_one()).

3.1.4 Зважене усереднення градієнтів та синхронізація моделей

Після завершення роботи потоків CPU та GPU, виникає розбіжність між їх локальними моделями, оскільки вони опрацьовували різні частини даних, було згенеровано різні вектори градієнтів цільової функції, якщо на цьому етапі відразу перейти до кроку оптимізації, параметри (ваги) моделей почнуть розходитися у просторі ознак, що призведе до навчання двох окремих моделей паралельно, а не до очікуваного результату, розпаралелення алгоритму навчання для однієї моделі, також втратиться будь-яка можливість їх синхронізувати.

Задля вирішення цієї проблеми, перед фізичним оновлення параметрів, система повинна звести градієнти отриманні від обох потоків до єдиного еталону, через розроблений балансувальник, дані розподіляються нерівномірно, тому використання звичайного усереднення не дасть коректного результату. До прикладу, при розподіленні даних до початку ітерації, відеокарта отримала 90%, а процесор лише 10%, ваги їхніх градієнтів у бажане усереднення мають бути строго пропорційні попередньому діленню, яке зробив балансувальник. Тому в системі було реалізовано алгоритм зваженого усереднення.

Момент злиття багатовимірних тензорів супроводжується деякими проблемами на рівні управління пам’яттю та апаратними шинами, які треба врахувати, реалізація виконує низьку важливих етапів:

1. Блокування автоматичного графу диференціювання: першим кроком є вимкнення механізму автоматичного обчислення похідних, бо математичний рушій фреймворку динамічно прослідковує за будь-якими математичними операціями над тензорами в пам’яті, які записує до графу обчислень (Autograd). Якщо система почала додавати градієнти один до одного з увімкненим графом диференціювання, рушій розпізнав би цю операцію, як новий шар нейронної мережі і назавжди зберіг її у пам’яті, це призвело би до аварійного завершення програми. Тому система відокремлює цю операцію у безпечний контекст, через який вказується рушію, що наступні операції є тільки системними маніпуляціями з пам’яттю.

2. Проблема розташування даних: одною з проблем є розташування даних фізично, градієнти, обчислені CPU знаходяться у оперативній пам’яті, а розрахунки GPU у VRAM. Між ними неможливі прямі операції, для цього алгоритм переносить дані процесора до відеопам’яті, де і буде відбуватися злиття градієнтів.

3. Злиття градієнтів на місці: після другого етапу, ми отримали дані від CPU та GPU розташовані у VRAM, що дає змогу алгоритму виконати зважене усереднення безпосередньо у відеопам’яті. Ця операція виконується прямо на місці перезаписуючи старі значення, це оптимізує процес усереднення градієнтів.

Після завершення третього етапу, модель, яка розташована у пам’яті відеокарти містить математично точний вектор градієнтів, який пропорційно враховує всі отриманні знання після виконання гібридного батчу, після цього алгоритм оптимізації виконує крок навчання (зміщує параметри моделі за обчисленими градієнтами), після чого система синхронізує ваги між двома моделями, що гарантує їх побітову ідентичність перед наступною ітерацією.

3.2 Реалізація кластерної взаємодії між вузлами системи

Після створення локального рівня , виникає потреба в масштабуванні на рівень кластера, тому комп’ютерна система була розроблена з можливістю розпаралелення алгоритму машинного навчання між декількома незалежними пристроями, управління цим процесом інкапсульовано у класи високого рівня, які відповідають за його оркестрацію та балансування навантаження в мережі.

3.2.1 Структура мережевої взаємодії

Для розподілених систем машинного навчання існує дві основні топології на мережевому рівні: децентралізована (Ring-AllReduce, де дані передаються по колу між вузлами) та централізована (Master-server), для розробленою системи було обрано централізований тип, оскільки кластер буде гетерогенним.

В структурі самого проєкту топологія було реалізована за принципом Master-Worker, при її використанні один виділений комп’ютер бере на себе роль головного мережевого окрестратора (Master), він відповідає за ініціалізацію стартових гіперпараметрів, зборі телеметрії, розрахунку розподілу даних та оновлені глобальної моделі. Усі інші машини виступають у ролі обчислювальних вузлів (Worker), їх завдання отримати індекси свого проміжку даних, виконати обчислення за допомогою свого апаратного устаткування та повернути результат (градієнти) до головного вузла мережею.

Однієї з проблем при розробці мережевого рівня кластера є подолання проблеми Network Bottleneck, моделі може містити мільйони параметрів, що буде еквівалентно десяткам гігабайтів даних, які буде потрібно передавати кожну ітерацію, а використання стандартних високорівневих протоколів (HTTP, gRPC) є недоцільним, оскільки вони використовують текстову серіалізацію (JSON,

XML) та вимагають підтримувати сесії з’єднання, в решті ми отримуємо ситуацію, де накладні витрати на парсинг можуть займати більше часу ніж одна ітерація математичних обчислень. З іншого боку, використання низькорівневих TCP-сокетів операційної системи вимагає розробки власної системи управління буферами пам’яті, контролю фрагментації пакетів та логіки передачі повторних повідомлень.

Для вирішення проблеми транспортування, транспортний модуль було побудовано на основі високоефективної бібліотеки ZeroMQ. Вона абстрагує розробника від управління TCP-сокетами, натомість вона оперує поняттями повідомлень і забезпечує маршрутизацію безпосередньо у коді. В коді було використано два патерни:

– паттерн REQ/REP: використовується для індивідуального підключення робочих вузлів та отримання початкових параметрів навчання. Це гарантований асинхронний канал, через який працівники надсилають свої дані головному вузлу;

– паттерн PUB/SUB: використовується для миттєвої широкомовної розсилки (broadcast). Головний вузол відкриває сокет pub, на який підписуються всі працівники кластеру, для миттєвого отримання усереднених глобальних градієнтів, це дозволяє мережевому оркестратору відправляти дані у мережу лише один раз.

3.2.2 Глобальне балансування навантаження на мережевому рівні

Головною проблемою гетерогенних кластерів є ефект відсталого вузла, оскільки комп’ютери мають різні технічні характеристики та різну пропускну здатність, статичний поділ дата-сету призвів би до зниження продуктивності

усього кластеру. Більш потужний вузол обробляв би свою частину, а потім очікував на завершення обчислень іншим вузлом.

Для забезпечення динамічного розподілу, який базується на швидкості обчислень кожного окремого вузла, було розроблено модуль мережевого балансувальника. Його завдання розділяти глобальний батч кластеру на фрагменти для кожної робочої станції кластеру, його життєвий цикл проілюстровано на блок-схемі на рисунку 3.6.

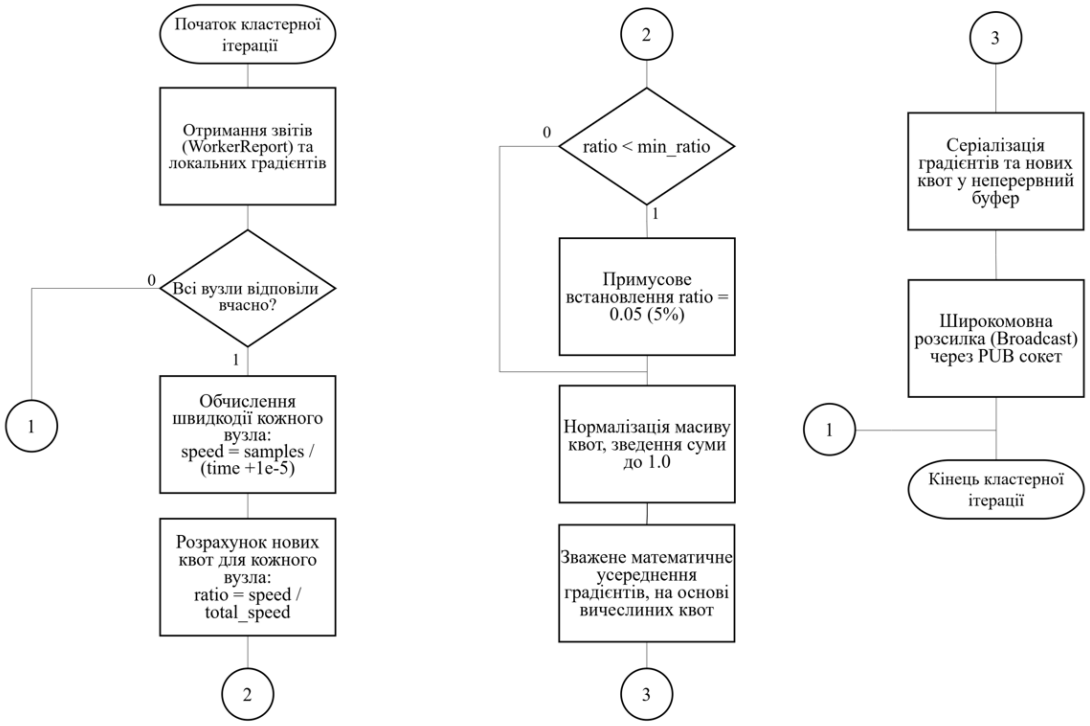

Рисунок 3.6 — Робота балансувальника мережевого рівня в межах одної ітерації.

Відповідно до наведеної блок-схеми, процес балансування на мережевому рівні поділяється на декілька базових етапів:

1. Збір даних та контроль роботи вузлів: алгоритм починає роботу з бар’єрного очікування на результати обчислень від інших вузлів кластеру,

під час якого головний вузол отримує масив структур WorkerReport та локальні градієнти. На цьому етапі проходить перевірка, чи всі вузли закінчили свою роботу та надіслали дані вчасно? При випадку, коли система фіксує тайм-аут одного з вузлів, алгоритм аварійно завершує роботу системи, миттєво перериваючи навчання, щоб починати його з робочими вузлами, якщо всі звіти отримано вчасно, алгоритм переходить до обчислення абсолютної швидкодії кожного вузла (speed = rep.processed_samples / (rep.compute_time_ms + 1e-5)), для запобігання діленню на нуль, бо існує можливість неправильно обчисленої телеметрії вузлом, втрата під час передачі або надшвидкої обробки батчу на вузлі, примусово додається константа 1e-5. Після цього розраховуються квоти, які будуть передані кожному вузлу кластера.

2. Нормалізація: наступним етапом іде перевірка розрахованої пропорції з лімітами встановлені в системі. Він використовується, щоб запобігти отримання вузлами нульових квот, бо можливі тимчасові зависання через стрибки пінгу або операційне фонове завантаження системи, тому алгоритм видає відповідному вузлу мінімальну квоту в 5%, щоб вузол не був назавжди виключений з роботи кластеру. Після застосування всіх обмежень, до масиву квот застосовується нормалізація, для зведення його сумарного значення до 1.0 (100%).

3. Трансляція даних: фінальний етап алгоритму складається з зваженого усереднення градієнтів, отриманих від робочих вузлів, з врахуванням обсягу даних оброблених кожним вузлом, еталонний утворений масив градієнтів серіалізується у єдиний неперервний буфер. Цей буфер відправляється через PUB-сокет у вигляді широкомовної розсилки. Робочі вузли пілся отримання цього пакету розблоковує їхніх локальних потоків для подальшого обчислення з новими вагами моделями, цей етап завершує поточну кластерну ітерацію.

3.2.3 Синхронізація генераторів та серіалізація тензорів

Для коректної роботи системи моделі на кожному вузлі повинні мати ідентичні стартові ваги, бо алгоритми ініціалізації стартових ваг нейронних мереж базуються на псевдовипадкових генераторах чисел, якщо моделі на різних комп’ютерах почнуть навчання з різними початковими вагам, спроба об’єднати їх градієнти призведе до руйнування матриць і до неможливості сходження глобальної моделі.

Вирішення цієї проблеми полягає в створенні головним вузлом випадкового числа (Random seed), який передається з початковими параметрами навчання на інші вузли кластеру. Вони застосовують його безпосередньо до математичного рушія LibTorch, тепер завдяки застосуванню random seed моделі гарантовано при створенні будуть мати побітову ідентичність ваг ще до початку передачі першого батчу.

Другою критичною проблемою є передача багатовимірних тензорів. Оскільки бібліотека ZeroMQ підтримує роботу з неперервними ділянками пам'яті, система імплементує концепцію Zero-copy. Замість використання повільних бібліотек серіалізації, градієнти лінеаризуються (розгладжуються) та копіюються прямим побітовим перенесенням (std::memcpy). Звіт обчислювача та градієнти об'єднуються в єдиний бінарний пакет (Payload). Це дозволяє мережевій карті відправляти дані напряму з оперативної пам'яті без створення проміжних копій у буферах ОС, зменшуючи затримку до мікросекунд.

3.2.4 Зважене математичне усереднення глобальних градієнтів

Детального інженерного роз'яснення потребує етап інтеграції градієнтів, який фігурує у фінальній частині алгоритму балансування. Оскільки

розроблений модуль нарізає дата-сет асиметрично, застосування простого середнього арифметичного (сума градієнтів, поділена на кількість вузлів) є критичною помилкою, що призвело би до втрати можливості до сходження моделі.

Якщо потужний вузол обробив 900 зображень, а слабкий лише 100, їхні градієнти не є рівнозначними для глобальної моделі. Градієнт сервера містить статистично більш вірну інформацію про генеральну сукупність даних, тоді як градієнт слабкого вузла має високий рівень шуму через малу вибірку. Просте усереднення призвело б до зміщення напрямку оптимізації в бік шуму.

Тому система реалізує алгоритм глобального зваженого усереднення. Вектор розгладжених градієнтів кожного вузла помножується на скаляр — кількість зразків, які цей вузол реально обробив. Після того, як Master-вузол сумує ці масштабовані масиви в єдиний глобальний буфер, результат ділиться на загальний обсяг батчу. Такий підхід робить внесок кожної машини у зміну еталонних ваг нейронної мережі суворо пропорційним обсягу виконаної роботи.

3.2.5 Бар'єрна синхронізація

Розподілені системи знаходяться під постійним впливом ризиків апаратних та мережевих збоїв. Транспортний рівень може втратити пакет, може розірватися з'єднання, а робоча станція може вимкнутися через перегрів. Якщо один із комп'ютерів кластера безслідно зникне з мережі, оркестратор назавжди залишиться заблокованим в операції recv. Це спричинить стан вічного блокування (Deadlock) всього кластера.

Для забезпечення відмовостійкості на транспортному рівні ZeroMQ було встановлено жорсткі тайм-аути на отримання пакетів. Як було зазначено в описі

алгоритму балансування, система реагує на такі збої за патерном суворого швидкого аварійного завершення.

У системах розподіленого машинного навчання агрегація неповних, фрагментарних знань (продовження навчання без градієнтів вузла, що впав) є набагато гіршим сценарієм, ніж повна зупинка. Втрата частини градієнтів може непомітно змістити розподіл даних і назавжди зіпсувати модель (С. Тому, якщо Master-вузол фіксує тайм-аут, система активує критичний виняток і миттєво перериває виконання програми. Таке рішення гарантує математичну коректність: глобальна еталонна модель ніколи не буде оновлена пошкодженими або десинхронізованими даними.

Хоча у випадку критичного мережевого збою поточна архітектура вимагає повного перезапуску процесу навчання з самого початку, цей вибір є абсолютно виправданим, бо набагато безпечніше отримати очевидну зупинку системи і пожертвувати поточним прогресом, ніж дозволити алгоритму мовчки продовжити роботу з втратами та згенерувати некоректну, зміщену модель, яка може мати некоректну роботу при її використанні. На стороні Worker-вузлів реалізовано комплементарний жорсткий бар'єр: після надсилання обчислених градієнтів комп'ютери призупиняють виконання потоків. Їхня діяльність відновлюється виключно після успішного отримання Broadcast-повідомлення від головного вузла, що забезпечує повну синхронність ітерацій у кластері.

3.3 Реалізація та використання програмного інтерфейсу користувача

Головною метою розробки будь-якого фреймворку є приховування складної логіки від кінцевого користувача. Розроблена система інкапсулює у собі складні процеси управління мережевими сокетами ZeroMQ, системними потоками операційної системи, бар'єрною синхронізацією та апаратним балансуванням між центральним процесором та відеокартою. Завдяки цьому

кінцевий користувач може зосередитися виключно на проєктуванні архітектури нейронної мережі та підготовці даних.

Для взаємодії з системою було розроблено об'єктно-орієнтований програмний інтерфейс користувача (API). Процес інтеграції користувацького коду з фреймворком складається з п'яти логічних етапів: визначення моделі та даних, конфігурація параметрів, налаштування моніторингу, ініціалізація середовища та вибір цільового обладнання.

3.3.1 Інтеграція нейронної мережі та набору даних

Основою будь-якого процесу машинного навчання є математична модель та дані. Фреймворк не обмежує користувача у виборі архітектури: це може бути як простий багатошаровий перцептрон (MLP), так і складна згорткова мережа (CNN). Єдиною вимогою API є успадкування користувацького класу від базового інтерфейсу IModel. Приклад наведено на рисунку 3.7

```
struct Net : public IModel {
    Net() {
        fc1 = register_module("fc1", torch::nn::Linear(2, 1028));
        fc2 = register_module("fc2", torch::nn::Linear(1028, 1));
    }
    torch::Tensor forward(torch::Tensor x) override {
        x = torch::relu(fc1->forward(x));
        x = fc2->forward(x);
        return x;
    }
    torch::nn::Linear fc1{ nullptr }, fc2{ nullptr };
};
```

Рисунок 3.7 — Демонстрація користувацької моделі

Як видно з наведеного прикладу, користувач використовує стандартні примітиви бібліотеки LibTorch. Виклик функції register_module є критично важливим: він реєструє матриці ваг кожного шару у внутрішньому графі обчислень (Autograd), що дозволяє фреймворку автоматично витягувати градієнти під час мережевої синхронізації. Метод forward перевизначається для опису прямого проходу (Forward Pass) тензорів через функції активації (наприклад, torch::relu).

Для роботи з даними API фреймворку використовує патерн "Адаптер". Користувач повинен створити клас, що успадковується від torch::data::Dataset, і реалізувати два базових методи: get(size_t index) для отримання конкретного зразка та size() для визначення загального обсягу вибірки. Такий підхід робить систему абсолютно незалежною від формату вхідних даних: користувач може завантажувати зображення формату JPEG, текст, або готові бінарні тензори формату .pt безпосередньо з жорсткого диска.

3.3.2 Конфігурація параметрів навчання та мережі

Щоб забезпечити гнучкість налаштувань та уникнути виклику функцій з надмірною кількістю аргументів, налаштування системи реалізовано через структури даних (Configuration Structs). Всі параметри розділені на дві логічні категорії: конфігурація машинного навчання (TrainingConfig) та мережева конфігурація кластера (NetworkConfig).

Структура TrainingConfig інкапсулює стандартні параметри, необхідні для процесу оптимізації. Користувач може задати швидкість навчання (learning_rate), кількість епох (epochs), розмір батчу (batch_size) та кількість фонових потоків ОС для завантаження даних (num_workers).

Для вибору частини параметрів було реалізовано систему типізації. Замість використання текстових рядків, вибір алгоритмів реалізовано через безпечні перелічення (Enums). Наприклад, вибір оптимізатора здійснюється через OptimizerType::Adam або SGD, а функція втрат — через LossType::MSE або CrossEntropy. Також структура містить вкладений об'єкт BalancerConfig, який дозволяє тонко налаштувати поведінку гібридного режиму (мінімальну та максимальну квоту GPU, поріг відключення процесора cpu_slowdown_factor та крок адаптації).

Для ініціалізації кластера використовується структура NetworkConfig:

```
NetworkConfig net_config;
net_config.role = NetworkConfig::Role::Master;
net_config.expected_workers = 1;
net_config.port_req = 5555;
net_config.port_pub = 5556;
net_config.local_device_type = TrainingConfig::DeviceType::CPU;
```

Рисунок 3.8 — Демонстрація налаштування мережевого конфігу

Основною властивістю API є те, що один і той самий скомпільований виконуваний файл може виступати як Головним вузлом (Master), так і обчислювачем (Worker), залежно від переданого параметра role. Для Master-вузла критичним є параметр expected_workers. Система використовує його для створення жорсткого бар'єра ініціалізації: оркестратор заблокує виконання програми і не розпочне процес навчання, доки не отримає підтвердження підключення від заданої кількості обчислювальних машин.

3.3.3 Моніторинг процесу (Callback API)

Однією з проблем закритих фреймворків є відсутність прозорості та неможливість інтеграції зі сторонніми утилітами. Щоб користувач міг відстежувати процес навчання у реальному часі, розроблений API імплементує патерн проєктування Спостерігач через механізм зворотних викликів (Callbacks).

Замість того, щоб вбудовувати виведення тексту у консоль глибоко в надра математичних алгоритмів, система після кожної визначеної ітерації (задається параметром log_interval_batches) формує комплексну структуру TrainingMetrics і передає її у користувацьку функцію.

Структура TrainingMetrics є місцем збору всієї телеметрії та містить три блоки даних:

1) метрики машинного навчання: поточна епоха, сирі значення функції втрат (current_loss) та згладжене значення втрат (ema_loss) для побудови трендів;

2) метрики локального обладнання: витрачений час на обробку батчу (batch_time_ms), абсолютна швидкість у зразках на секунду (samples_per_second) та поточне навантаження на локальний GPU (local_gpu_ratio);

3) глобальні мережеві метрики: ідентифікатор вузла, його поточна відсоткова частка від загального мережевого навантаження (cluster_load_share) та фізична кількість призначених зображень (local_chunk_size).

Завдяки методу orchestrator.SetLogCallback(), користувач отримує повний контроль над обробкою телеметрії. У найпростішому випадку дані можуть форматуватися та виводитися у консоль. Проте ця архітектура дозволяє легко інтегрувати структуру TrainingMetrics з інструментами візуалізації (TensorBoard)

або відправляти телеметрію на віддалений сервер моніторингу (Grafana) без жодних змін у ядрі фреймворку.

3.3.4 Ініціалізація, впровадження залежностей та запуск навчання

Фінальним етапом є ініціалізація оркестратора та запуск процесу обчислень. Враховуючи багатопотокову природу системи, пряма передача готового екземпляра моделі в ядро є небезпечною, оскільки різні обчислювачі (наприклад, локальні CPU та GPU) вимагають незалежних об'єктів графа обчислень для уникнення стану гонитви.

Тому API використовує патерн Впровадження залежностей у вигляді фабричного методу:

```
auto model_builder = []() {
   return std::make_shared<Net>();
};
auto my_dataset = MyDataSet("inputs.pt", "targets.pt");
NetworkTrainer orchestrator(train_config, net_config);
orchestrator.Start(model_builder, std::move(my_dataset));
```

Рисунок 3.9 — Демонстрація впровадження залежностей та створення окрестратора

Замість моделі, користувач передає лямбда-вираз model_builder, який описує процедуру створення нової нейромережі. Фреймворк самостійно викличе цей будівельник необхідну кількість разів. Використання семантики переміщення std::move(my_dataset) гарантує передачу прав власності на масиви даних усередину фреймворку, усуваючи необхідність глибокого копіювання дорогоцінної оперативної пам'яті.

Також в користувацькому інтерфейсі приділено увагу до стійкості до відмов. Життєвий цикл навчання обгорнуто у блок перехоплення винятків (try-catch), який чітко розмежовує типи помилок. Перехоплення c10::Error сигналізує про фатальні помилки на рівні ядра LibTorch (наприклад, вихід за межі відеопам'яті CUDA Out Of Memory), тоді як std::exception обробляє помилки налаштувань фреймворку або критичні мережеві тайм-аути ZeroMQ.

3.3.5 Ізольоване локальне виконання та вибір режимів апаратного прискорення

Для використання розробленої системи не є обов’язковим мати декілька робочих станцій, бо API спроєктовано таким чином, щоб у користувача залишалася можливість запускати навчання в ізольованому режимі (локально) на одній машині (комп’ютер або ноутбук).

Управління локальними обчисленнями та вибір цільового обладнання здійснюється через параметр type у структурі TrainingConfig, який приймає значення безпечного перелічення DeviceType. Доступні три режими виконання:

1. DeviceType::CPU — виконання матричних операцій виключно на центральному процесорі. Використовується як базовий режим для тестування або на пристроях без дискретних відеокарт.
2. DeviceType::GPU — ексклюзивне використання графічного прискорювача. Фреймворк направляє 100% даних у відеопам'ять, звільняючи процесор від розрахунків.
3. DeviceType::Hybrid — інноваційний режим локальної роботи, що активує розроблений динамічний балансувальник навантаження. Система адаптивно розподіляє батч між центральним процесором та відеокартою в межах однієї робочої станції.

Для тонкого налаштування гібридного режиму користувач отримує доступ до структури BalancerConfig. Інтерфейс дозволяє втручатися у поведінку математичного алгоритму під конкретне обладнання: задавати стартову пропорцію розподілу даних (test_start_ratio), встановлювати жорсткий ліміт навантаження для відеокарти (max_gpu_ratio), а також налаштовувати агресивність механізму відсікання за допомогою параметрів cpu_slowdown_factor та min_time_diff_ms_kill.

Такий архітектурний підхід робить розроблений програмний інтерфейс користувача універсальним інструментом: фреймворк здатний однаково ефективно масштабувати процес навчання як вертикально (використовуючи одночасно CPU та GPU), так і горизонтально (об'єднуючи десятки машин через ZeroMQ).

3.4 Висновки до третього розділу

У третьому розділі детально розглянуто програмну реалізацію системи мовою C++ із використанням бібліотеки для роботи з тензорами (LibTorch) та комунікаційної бібліотеки ZeroMQ. Реалізовано локальний рівень із пулом потоків та прихованою бар'єрною синхронізацією, що дозволяє безпечно поєднувати градієнти від центрального процесора та графічного прискорювача. Створено комунікаційний модуль кластерної взаємодії, який застосовує патерни REQ/REP та PUB/SUB, а також використовує концепцію Zero-copy для мінімізації мережевих затримок під час передачі багатовимірних тензорів. Важливим здобутком є імплементація математично точного алгоритму зваженого усереднення градієнтів, що враховує асиметричний розподіл обсягу даних між вузлами кластеру, зберігаючи коректність оптимізації моделі.

4. ТЕСТУВАННЯ ТА ОЦІНКА ПРОДУКТИВНОСТІ

4.1 Тестування коректності роботи системи

Найважливішим етапом тестування розробленої системи (AutoBalancedTrainer) є підтвердження того, що ядро системи (алгоритм прямого та зворотного поширення помилки, оптимізатор) працює математично коректно, а модель дійсно має змогу навчатися.

Для перевірки здатності системи забезпечити навчання моделі було запущено процес навчання згорткової нейронної мережі (MNIST_CNN) на підготовлених бінарних даних, створених з дата-сету MNIST. Головним критерієм правильності роботи системи є стабільне падіння функції втрат (loss) протягом процесу навчання. Логування під час цього процесу за допомогою метрики експоненційної ковзної середньої помилки, яка демонструє загальну тенденцію зміни помилки, що після кожного кроку оптимізатора Adam, розбіжність між прогнозами моделі поменшується, тенденція показана на рисунку 4.1

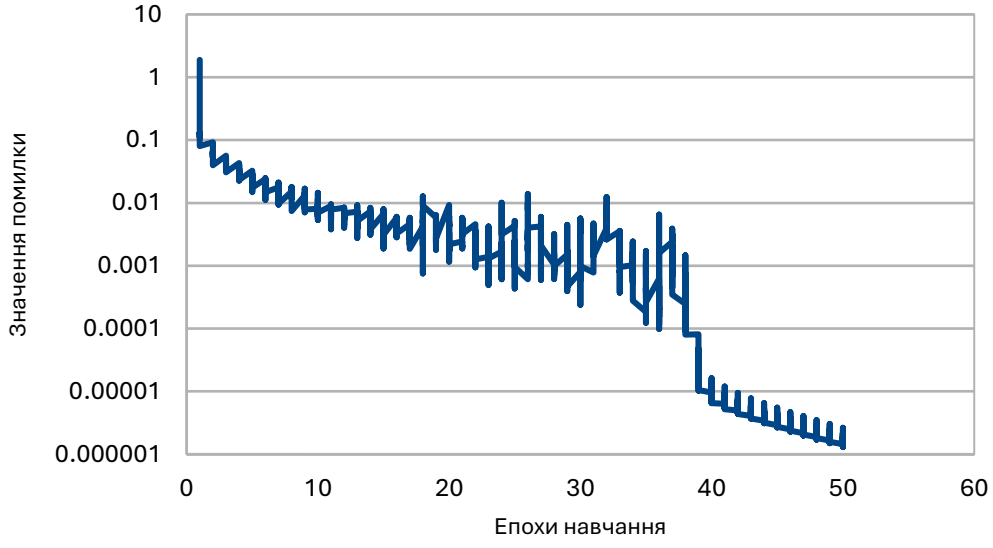

Рисунок — 4.1 Графік зменшення помилки моделі

На цьому графіку ми точно можемо побачити, як значення помилки зменшується з кожною наступною епохою, але щоб остаточно довести, що модель не просто завчила тренувальний дата-сет, було створено тест, де фінальні дані моделі були збережені до файлу mnist_weights.pt та модель було використана у Python-середовище, де для тестування використовувалися нові для моделі дані, намальовані користувачем, відповідні зображення модель не бачила під час тренування, результат цього тестування представлено на рисунку 4.2.

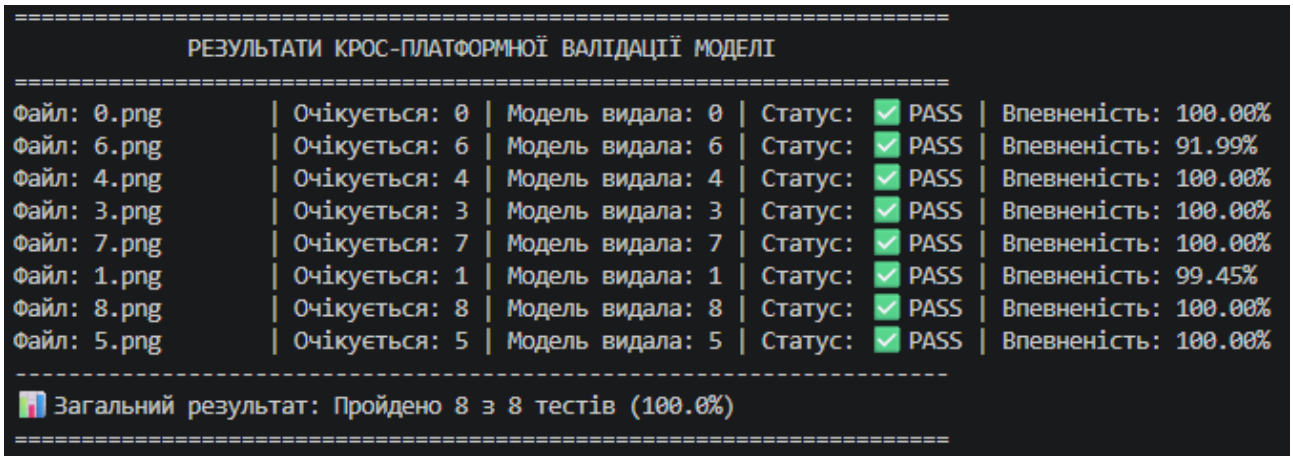

Рисунок 4.2 — Успішна обробка моделлю тестових ізольованих даних

Успішне проходження даного тесту формує розуміння, що фреймворк повністю працездатний та забезпечує коректне обчислення градієнтів і формування ваг моделі.

4.2 Оцінка продуктивності послідовного виконання

Для ефективної оцінки роботи розпаралелення системою алгоритму навчання, необхідно встановити базову ефективність навчання, для цього система запускається в послідовному режимі навчання (expected_worker = 0).

Метою цього експерименту є оцінка швидкості навчання на двох принципово різних типах апаратного устаткування, CPU та GPU, тестування проводиться на двох різних моделях, легка — MLP та важка — CNN, при фіксованих розмірах батча і кількості епох. Система була переведена для роботи в відповідні режими роботи. Основними метриками оцінки ефективності є середній час обробки батчу, середня швидкість обробки даних протягом навчання та сам його час. Результати наведені в таблиці 4.1.

Таблиця 4.1 — Результати оцінки продуктивності послідовного виконання

Модель	Режим навчання	Середній час батчу, мс	Швидкість обробки, зразків/с	Загальний час навчання
MLP	CPU	94.4848, мс	89928.6, зразків/с	2.3219, секунд
MLP	GPU	94.5879, мс	58996.5, зразків/с	2.9349, секунд
CNN	CPU	115.9244, мс	4425.2, зразків/с	681.5711, секунд
CNN	GPU	8.5289, мс	60480.3, зразків/с	52.8073, секунд

Аналіз даних, наведених у таблиці, дозволяє зробити висновки, що для навчання не завжди доцільно використовувати лише графічні прискорювачі, тому, що ми бачимо на прикладі MLP процесор випереджає відеокарту, це пояснюється необхідним відеокарті часом на копіювання тензорів у пам’ять та ініціалізацією CUDA-ядер. В той же час, через невелику складність самої моделі, CPU випереджає GPU.

Ситуація кардинально змінюється на прикладі CNN, це спричинено високою паралелізацією згортки, яка ідеально обробляється на архітектурі CUDA.

4.3 Дослідження продуктивності паралельного виконання

Основним етапом тестування розробленої системи є оцінка алгоритмів динамічного розподілу навантаження під час роботи системи в режимі кластера. Для проведення відповідного експерименту було розгорнуто кластер, який складається з трьох обчислювальних вузлі, які кардинально відрізняються за своїми обчислювальними потужностями:

1) майстер-вузол (Флагман): оснащений високопродуктивною відеокартою — RTX 4060;

2) воркер 1: використовує графічний прискорювач попердніх поколінь — GTX 1650;

3) воркер 2: обчислення виконуються виключно на потужностях багатоядерного центрального процесора — Ryzen 7 7735HS.

Усі метрики під час навчання зберігалися вузлами в локальні лог-файли, на базі яких в подальшому буде проводитися аналіз роботи.

Під час аналізу перших логів роботи кластера було виявлено локальну варіативність функції втрат. Як видно з консольних виводів, зображених на рисунку 4.3, з нього можемо побачити, що значення помилки отримані в межах однакових епох відрізняються на майстрі та воркерах, також треба звернути увагу, що помилка на вузлі, який використовує лише центральний процесор є стабільно нижче за показники вузлів, які працюють в гібридному режимі.

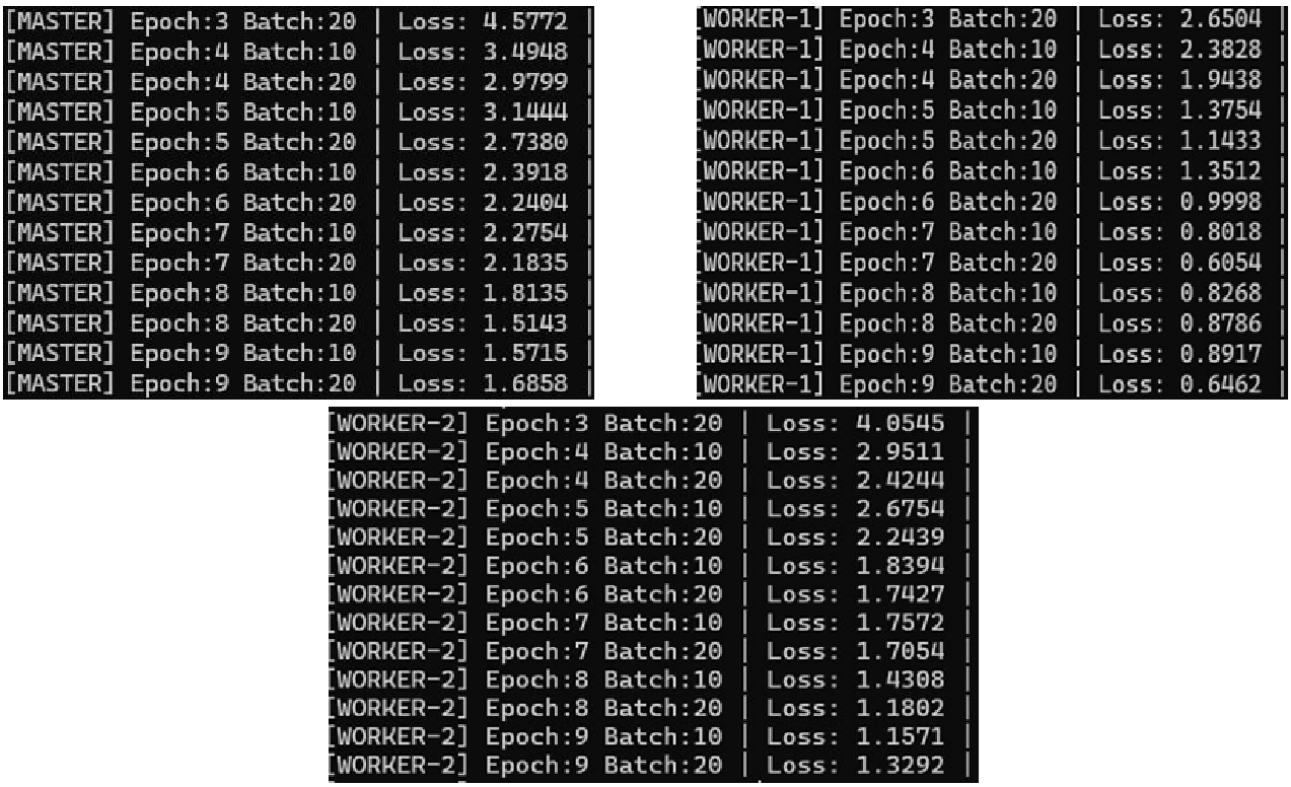

Рисунок 4.3 — Динаміка локальної функції втрат під час навчання MLP у кластері

Ця розбіжність є показником роботи алгоритму динамічного балансування в обчислювальному середовище, її основні причини полягають у наступному:

1. Динамічний розмір вибірки : оскільки для моделі MLP центральні процесори демонструють високу швидкість обчислень, алгоритм балансування на кожному батчі перераховує квоти для вузлів, бо кожен вузол має шанс завершити свою частину вибірки раніше за інших.

2. Варіативність нових даних: на кожній новій ітерації вузли отримують абсолютно нові дані, які модель ще не обробляла, а ваги вона отримала за рахунок обробки попередніх даних, тому параметри моделі не є ще повноцінно адаптованими до специфіки поточних даних, тому при отримані специфічною або більш складною вибірки, локально функція втрат закономірно зростає.

3. Методи агрегації на вузлах: майстер-вузол та воркер-2 працюють у гібридному режимі, де локально помилка розраховується, як зважене середнє між паралельними потоками процесора та відеокарти, з іншої сторони воркер-1 працює у режимі CPU, обчислюючи помилку на єдиному пристрої, ця розбіжність так само вносить відсоток до розбіжності локальної помилки на вузлах.

Незважаючи на динамічне коливання локального значення loss на вузлах, спричиненого факторами, описаними вище, загальне навчання проходить математично коректно, бо аналізуючи отримані логи (рис. 4.3), ми можемо спостерігати виражений, стабільний спад функції втрат на всіх вузлах одночасно. На майстер-вузлі показник знизився з 4.57 до 1.68, на воркері-1 з 2.65 до 0.64 та воркері-2 з 4.05 до 1.32, ця синхронна тенденція до збіжності повністю демонструє успішну роботу розробленої системи в режимі роботи кластеру, поєднання локальних градієнтів в один глобальний та формування правильних ваг моделі.

Після підтвердження коректної роботи кластеру та успішної агрегації градієнтів між копіями моделей на різних вузлах, наступним етапом є дослідження загальної пропускної здатності системи з використанням різних топологій кластеру. Для аналізу було розгорнуто кластер для навчання, вище згадуваних, MLP та CNN моделей. Метрики їх навчання наведено у таблиці 4.2.

Таблиця 4.2— Результати оцінки продуктивності роботи кластеру

Модель	Топологія кластеру	Середній час батчу, мс	Швидкість обробки, зразків/с	Загальний час навчання
MLP	2 гібридні вузли + вузол CPU	34.1162, мс	19224.9, зразків/с	32.22, секунд

Продовження таблиці 4.2

Модель	Топологія кластеру	Середній час батчу, мс	Швидкість обробки, зразків/с	Загальний час навчання
MLP	3 CPU вузли	28.4798, мс	28628.2, зразків/с	27.12, секунд
CNN	2 гібридні вузли + вузол CPU	756.9868, мс	706.7, зразків/с	4 356.17, секунд
CNN	2 GPU вузли	214.5098, мс	2432.5, зразків/с	1252.10, секунд

Отримана телеметрія демонструє ключову проблему розподілених систем, з іншого боку було продемонстровано ефективність розробленої системи. Хоча кластер мав апаратні обмеження, алгоритм балансування навантаження продемонстрував, як з топологію, яка включає два гібридні вузли та один лише з CPU, пропускну здібність в 19224 зразків/с, хоча отримане значення і поступається кластеру, який складається лише з центральних процесорів, але загалом критичного падіння продуктивності не сталося, як могло бути при статичному поділу вибірки даних. Алгоритм балансування перерозподілив навантаження в бік центральних процесорів та утримав продуктивність на достатньому рівні. Перевага CPU з моделлю MLP є очікуваним результатом, оскільки сучасні процесори добре справляються з невеликим матричним множенням.

Показовим є результат, отриманий під час навчання згорткової нейронної мережі (CNN). Кластер з двох відеокарт продемонстрував майже у три рази швидшу роботу ніж повний кластер з гібридними режимами та додатковим CPU вузлом. Ця втрата продуктивності обумовлена накладними витратами та

мережевого пляшкового горличка, оскільки CNN генерує надвеликі масиви матриці градієнтів вагою десятки мегабайтів. Фізичний час серіалізації градієнтів та передача їх локальною мережею критично перевищує час обчислень, це робить процес навчання мережево-залежним. Цей експеримент доводить, що використання суто CPU-вузлів для важких моделей є недоцільний, хоча алгоритм балансування виділяє їм над маленьку вибірку даних, але система в будь-якому випадку змушена передавати глобальні градієнти цьому вузлу, що сумується з повільним обчисленням великих масивів матриць на процесорі та призводить до простоювання найпотужнішого вузла в кластері.

Оскільки апаратне вирішення цієї проблеми шляхом впровадження більш швидких засобів обміну між вузлами суперечить ідеї розроблюваної системи, виникає потреба вирішувати цю проблему програмно. Одним із дієвих засобів є масштабування глобального батчу, обсяг градієнтів, які передаються мережею не зміниться, оскільки він залежить лише від параметрів самої моделі, тож збільшуючи розмір глобальної вибірки, ми збільшуємо корисний час роботи системи та зменшуємо частоту обміну мережею градієнтами без використання надшвидких інтерфейсів для обміну між вузлами.

Для підтвердження, вище описаної програмної оптимізації було проведено додатковий експеримент, в якому розмір батчу було збільшено у 8 разів для кластеру з двох GPU, щоб максимально заповнити пам’ять слабшого вузла (GTX 1650).

Отримані результати демонструють, що середній час обчислення вибірки даних не змінився і склав 211.06 мс проти 214.5 мс, це наочно показує, що велику частку обробки батчу займає саме мережева синхронізація між вузлами. Через збільшення батчу, система за один цикл синхронізації вузлів обробила більше даних, за рахунок цього загальна пропускна здібність зросла до 19700 зразків на секунду, а загальний час навчання скоротився до 156.93 секунд.

Хоча по швидкості кластер не наздогнав еталонний показник навчання на одному ізольованому GPU, але дає розуміння, що при збільшені вибірки, яка обробляється за одну ітерацію, та горизонтальному масштабуванні кластеру система матиме змогу перекрити мережеві затримки.

Результати досліджень дають розуміння, що при правильних налаштуваннях топології та адаптації гіперпараметрів навчання, розроблена система має змогу прискорити процес навчання.

4.4 Порівняльний аналіз

На основі проведених експериментів, від ізольованого навчання до оптимізованого кластеру, здійснено підсумковий порівняльний аналіз та рекомендації щодо вибору топології кластеру від типу задачі.

Аналіз отриманих результатів дає змогу виділити три основні закони систем розпаралелення алгоритмів машинного навчання:

1. Архітектурна залежність від типу нейронної мережі: порівняння моделей MLP та CNN, наочно демонструє, що доцільність розпаралелення алгоритму навчання залежить від обчислювальної складності моделі та обсягу мережевого трафіку.

2. Мережевий парадокс: показовим є порівняння пропускної здібності навчання CNN на одній потужній відеокарті із неоптимізованим кластером з двох відеокарт, який складається з тієї самої відеокарти та додаткової, більш слабшої, фіксувалося критичне падіння продуктивності у другому випадку, що підтвердило проблему мережевого перенасичення. Також було продемонстровано недоцільність залучення центральних процесорів для навчання таких моделей.

3. Ефективність оптимізації та балансування: під час проведення експериментів було продемонстровано, що за допомогою програмних рішень можливо оптимізувати процес навчання моделей, а саме амортизація витрат, в нашому випадку збільшення глобальної вибірки для кластера, щоб оптимізувати часті звернення до мережевої синхронізації. Також використання динамічного балансування навантаження показало, що з її допомогою навіть при об’єднанні обладнання з різних поколінь не призвело до повної деградації системи навчання.

4.5 Висновки до четвертого розділу

У четвертому розділі проведено комплексне тестування коректності та продуктивності розробленої системи. Успішне крос-платформне тестування згорткової моделі (MNIST_CNN) підтвердило математичну коректність роботи ядра системи та здатність до збіжності розподіленої моделі. Під час дослідження продуктивності у гетерогенному кластері (RTX 4060, GTX 1650, Ryzen CPU) доведено ефективність алгоритмів динамічного балансування, які змогли утримати високу пропускну здатність системи без простою потужного обладнання. Виявлено та проаналізовано явище "мережевого парадоксу" при навчанні важких моделей, для вирішення якого було успішно застосовано метод масштабування глобальної вибірки даних, що підтвердило здатність системи адаптуватися до умов локальних мереж.

5. КЕРІВНИЦТВО КОРИСТУВАЧА

Для успішного розгортання та роботи розробленої система, цільова обчислювальна система має працювати під управлінням операційної системи Windows 10 або 11. Базовим середовищем розробки є Microsoft Visual Studio з напередвстановленим пакетом для розробки класичних додатків C++, це забезпечить наявність компілятора MSVC та автоматизованої системи збирання Cmake.

Для використання апаратного прискорення на відеокартах Nvidia необхідно завантажити платформу Nvidia CUDA Toolkit (посилання) версії 12.1 або новіше. Наявність цього інструменту є необхідним для виконання виконання матричних операцій на тензорних ядрах відеокарти.

Ядром для виконання математичних операцій у системі виступає тензорний бекенд LibTorch (посилання), версію якого користувач має обрати залежно від наявності відеокарти (CUDA 12.1+) або використання лише процесорних потужностей (CPU). Після завершення завантаження архіву з офіційного сайту, його слід розпакувати у локальну директорію, шлях до неї в подальшому буде використовуватися інструментом CMake.

Для використання мережевої взаємодії система потребує бібліотеки cppzmq, для її автоматичного підключення знадобиться пакетний менеджер vcpkg (посилання). Треба клонувати репозиторій за посиланням після чого виконати наступні команди у терміналі, які зображені на рисунку 5.1.

```
bootstrap-vcpkg.bat
vckpg integrate install
```

Рисунок 5.1 — Команди необхідні для інтеграції пакетного менеджера vckpg

Завдяки наявності файлу маніфеста vckpg.json в репозиторії фреймворку, система самостійно завантажить та скомпілює код ZeroMQ під цільову архітектуру.

Фінальним етапом є завантаження та збирання проєкту. Користувачу необхідно клонувати або завантажити архів репозиторію розробленої системи (посилання), який містить директорію із заголовковими файлами, конфігурацію CMakeLists.txt та шаблонний файл вихідного коду. Після відкриття директорії з завантаженим репозиторієм в середовищі Visual Studio (можливе також використання Visual Studio Code), система автоматично виявить маніфест vckpg.json, завантажить проєкт. Далі користувачу треба буде вказати шлях до папки libtorch, яка раніше завантажувалася, та в середині неї вказати шлях до libtorch\share\cmake\Torch. Процес встановлення завершується збиранням проєкту, після чого система повністю готова до функціонування.

У даному розділі було детально розглянуто процес розгортання та налаштування розробленої системи у середовищі ОС Windows. Визначено базові програмно-апаратні вимоги, зокрема використання Microsoft Visual Studio з компілятором MSVC, системи автоматизації збирання CMake та платформи Nvidia CUDA Toolkit для забезпечення апаратного прискорення на графічних процесорах. Описано покроковий алгоритм підключення ключових залежностей: тензорного бекенду LibTorch та комунікаційної бібліотеки ZeroMQ. Завдяки використанню сучасного пакетного менеджера vcpkg та конфігураційних файлів CMake, процес підключення бібліотек та компіляції вихідного коду максимально автоматизовано. Це зводить до мінімуму необхідність складних ручних налаштувань і дозволяє користувачу швидко підготувати обчислювальне середовище та розпочати роботу з розподіленою системою.

ВИСНОВКИ

У межах виконання бакалаврського дипломного проєкту було успішно розв’язано задачу з розробки легковісної та високопродуктивної комп’ютерної системи для розпаралелення алгоритмів машинного навчання мовою C++. Розроблена система є протиставленням складним корпоративним фреймворкам, які орієнтовані на спеціалізовані дата-центри і втрачають продуктивність в звичайних локальних мережах із гетерогенним обладнанням.

Фундаментом розробленого рішення стала трирівнева структура, що об'єднує кластерну мережеву взаємодію, локальну оркестрацію та низькорівневу багатопоточність. Відмова від класичного статичного поділу навчальної вибірки на користь динамічного балансування дозволила системі самостійно адаптуватися до швидкодії кожного пристрою. Завдяки безперервному аналізу телеметрії розроблений алгоритм автоматично перерозподіляє навантаження між центральними (CPU), графічними (GPU) процесорами та вузлами кластеру. Це дозволило ефективно протистояти проблемі відсталого вузла та звести до мінімуму час простою найпотужнішого обладнання в кластері.

Для забезпечення математичної цілісності моделі, яка навчається, в системі було імплементовано строгу бар’єрну синхронізацію та метод зваженого усереднення градієнтів, який поєднує градієнти строго пропорційно до обсягу оброблених даних. Високу швидкість обміну багатовимірними тензорами забезпечив транспортний модуль на базі бібліотеки ZeroMQ із застосуванням концепції прямого доступу до пам'яті. Також всю складну мережеву взаємодію та складні алгоритми балансування було приховано під зручним користувацьким API, який дає змогу легко налаштовувати навчання за допомогою конфігів та моніторити процес, завдяки механізму зворотніх викликів.

Результати комплексного тестування на реальному апаратному забезпеченні повністю підтвердили заявлені характеристики системи. Практичні експерименти засвідчили стабільну збіжність моделей, здатність системи до

горизонтального та вертикального масштабування, а також можливість ефективного подолання мережевих обмежень шляхом регулювання глобального батчу. У підсумку створено повноцінний, самодостатній програмний продукт, який дозволяє максимально розкрити обчислювальний потенціал доступного апаратного забезпечення.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Zhang A., Lipton Z. C., Li M., Smola A. J. Dive into Deep Learning. [Електронний ресурс]. Режим доступу: https://d2l.ai/ (дата звернення: 25.05.2026).

2. Model Parallelism Core Concepts. AWS SageMaker Documentation. [Електронний ресурс]. Режим доступу: https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-intro.html (дата звернення: 25.05.2026). (Документація щодо архітектури та концепцій паралелізму моделей).

3. Hennessy J. L., Patterson D. A. Computer Architecture: A Quantitative Approach (6th Edition). – Morgan Kaufmann, 2017. – 984 p. (Фундаментальна праця щодо архітектури сучасних обчислювальних систем та оцінки їхньої продуктивності).

4. Trobec R., Slivnik B., Bulić P., Robič B. Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms. – Springer, 2018. – 268 p. [Електронний ресурс]. Режим доступу: https://ftp.utcluj.ro/pub/users/civan/CPD/3.RESURSE/9.Book_2018_Introduction%20to%20ParallelComputing_Trobek.pdf (дата звернення: 25.05.2026).

5. PyTorch C++ API (LibTorch) Documentation. PyTorch Foundation. [Електронний ресурс]. Режим доступу: https://pytorch.org/cppdocs/ (дата звернення: 25.05.2026). (Офіційне керівництво з тензорних обчислень та розгортання моделей на C++).

6. Hintjens P. ZeroMQ: Messaging for Many Applications. – O'Reilly Media, 2013. – 238 p. [Електронний ресурс]. Режим доступу: https://zguide.zeromq.org/ (дата звернення: 25.05.2026). (Посібник із проектування розподілених мережевих топологій та патернів обміну повідомленнями).

7. Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM. 2008. Vol. 51, No. 1. P. 107–113. DOI: https://doi.org/10.1145/1327452.1327492 (Базова наукова стаття щодо принципів побудови розподілених обчислювальних систем архітектури Master-Worker).