МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«УЖГОРОДСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ»
ФАКУЛЬТЕТ МАТЕМАТИКИ ТА ЦИФРОВИХ ТЕХНОЛОГІЙ
Кафедра системного аналізу і теорії оптимізації

ВОГАР АНДРІЙ ЮРІЙОВИЧ

ОПТИМІЗАЦІЯ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ ДЛЯ РЕСУРС-ОБМЕЖЕНИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ

124 Системний аналіз

Дипломна робота на здобуття освітнього ступеня бакалавра

Науковий керівник:
Корник Олександр Володимирович
асистент каф. САТО

Ужгород – 2026

АНОТАЦІЯ

Це дослідження присвячене аналізу та розробці методів підвищення обчислювальної ефективності великих мовних моделей (LLM) для їхнього розгортання на пристроях з обмеженими апаратними ресурсами.

У роботі розглянуто сучасний стан розвитку нейронних мереж та виявлено основні бар'єри, що заважають локальному використанню потужних моделей на мобільних і вбудованих системах. Детально розглянуто та систематизовано основні методи стиснення моделей: квантування (Quantization); прунінг (Pruning); а також використання ефективних архітектур, таких як BitNet та дистильовані моделі.

Об'єктом дослідження є процеси оптимізації великих мовних моделей. Предметом дослідження виступають алгоритми квантування, прунінгу та архітектурні модифікації, що дозволяють зменшити обчислювальні витрати без критичної втрати точності генерації. Практична цінність роботи полягає у визначенні найбільш ефективних конфігурацій оптимізації для забезпечення балансу між швидкістю роботи моделі (tokens per second) та її інтелектуальними можливостями на слабких обчислювальних системах.

Ключові слова: великі мовні моделі (LLM), оптимізація нейронних мереж, квантування, прунінг, ресурс-обмежені системи, BitNet, штучний інтелект на пристроях (Edge AI).

ABSTRACT

This research is dedicated to the analysis and development of methods for increasing the computational efficiency of Large Language Models (LLMs) for their deployment on devices with limited hardware resources.

The study examines the current state of neural network development and identifies the primary barriers preventing the local use of powerful models on mobile and embedded systems. Key model compression methods are detailed and systematized: quantization (Post-Training Quantization and Quantization-Aware Training); pruning (structured and unstructured approaches); and the implementation of efficient architectures, such as BitNet and distilled models.

The object of the study is the process of optimizing large language models. The subject of the study involves quantization algorithms, pruning techniques, and architectural modifications that allow for the reduction of computational overhead without a critical loss in generation accuracy. The practical value of the work lies in determining the most effective optimization configurations to achieve a balance between inference speed (tokens per second) and the model's intellectual performance on resource-constrained computing systems.

Keywords: Large Language Models (LLM), neural network optimization, quantization, pruning, resource-constrained systems, BitNet, Edge AI.

ЗМІСТ

ВСТУП

Великі мовні моделі (Large Language Models, LLMs) у сучасному світі стали невід'ємною частиною життя для багатьох людей, що використовують штучний інтелект. Вони вкючають у себе обробку природньої мови, машинний переклад, генерацію тексту та інше. Проте, через величезну кількість параметрів, що входять у мовну модель створюються великі обчислювальні вимоги. Запуск таких моделей на ресурс-обмежених пристроях, таких як мобільні телефони або вбудовані системи, стає викликом.

Аналіз та оптимізація мовних моделей є критично важливим для забезпечення їхньої доступності, масштабованості та ефективності на різних платформах. Квантування (Qantinization), прунінг (pruning) та розробка ефективних архітектур є основними у цій галузі, які дозволяють зменшити розмір самих моделей, зменшити вимоги до заліза та за рахунок цього покращити їх продуктивність без значної втрати якості виводу моделі.

Кінцевою метою є оптимізація вже наявних мовних моделей для роботи на слабких пристроях при одночасному збереженні їхньої функціональності, точності та якості, що сприятиме широкому впровадженні штучного інтелекту в повсякденне життя.

Об'єкт дослідження: Процеси оптимізації великих мовних моделей.

Предмет дослідження: методи квантування, прунінгу та розробки ефективних архітектур для запуску великих мовних моделей на ресурс-обмежених пристроях.

Мета дослідження: Розробка та впровадження ефективних методів оптимізації великих мовних моделей для їхнього запуску на слабких пристроях.

Це дослідження може бути корисним для дослідників та пересічної зацікавленої людини у галузі штучного інтелекту, машинного навчання, роботи мовних моделей. Також для інженерів, які працюють та шукають шляхи для впровадженням великих мовних моделей у реальні застосунки на малопотужних пристроях.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ОПТИМІЗАЦІЇ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

1.1 Концепція та архітектурні особливості великих мовних моделей

1.1.1 Визначення та еволюція мовних моделей

LLM (Large Language Model) - це велика мовна модель, яка є типом штучного інтелекту, навченого розуміти та генерувати людську мову [1, с. 4].

Історію розвитку великих мовних моделей доцільно розділити на кілька ключових етапів відповідно до технологічних підходів, що домінували у певні періоди:

1. Статистичні методи та прості вектори (до 2013 р.):

    - Bag-of-Words (BoW): ранні підходи представляли текст як «мішок слів», підраховуючи частоту їх появи. Це дозволяло перетворити текст на числові дані, проте втрачався порядок слів та їхній семантичний контекст [1, с. 6].
    - N-gram: моделі передбачали наступне слово, враховуючи лише N попередніх одиниць тексту. Це було ефективно для вузьких завдань, але не дозволяло опрацьовувати довгі контекстні зв'язки [1, с. 88].

2. Векторні представлення слів (2013 р.):

    - Word2Vec: технологічна революція, що дозволила створювати вбудовування (embeddings) – щільні вектори, де слова зі схожим значенням розташовані поруч у математичному просторі. Наприклад, векторні операції дозволяли розв'язувати аналогії типу «король – чоловік + жінка = королева» [1, с. 8]. Проте кожне слово мало лише один

фіксований вектор (наприклад, слово «bank» мало однакове значення і для фінансової установи, і для берега річки) [1, с. 11].

3. Рекурентні мережі та контекст (2014–2016 рр.):

   - RNN та LSTM: ці архітектури обробляли текст послідовно, зберігаючи «пам'ять» про попередні токени. Це дозволило враховувати порядок слів і локальний контекст [1, с. 11].
   - Увага (Attention): механізм, що дозволив моделям фокусуватися на найбільш релевантних словах у реченні при виконанні завдань (наприклад, машинного перекладу), замість стиснення всього речення в один вектор [1, с. 13].

4. Ера Трансформерів (2017 р. – теперішній час):

   - Transformer (2017): публікація статті «Attention Is All You Need» представила архітектуру, яка повністю відмовилася від рекурентності на користь механізму самоуваги (Self-Attention). Це дозволило обробляти всі слова паралельно, що значно пришвидшило навчання на великих масивах даних та заклало фундамент для сучасних LLM [1, с. 15].

Сучасні LLM є переважно генеративними моделями (decoder-only) та побудовані на архітектурі трансформерів (Transformers). Вони працюють за принципом авторегресії: модель отримає на вхід текст і намагається передбачити, яке слово (або частина слова) має йти наступним [1, с. 12].

Сприйняття тексту моделлю відрізняється від людського: на вхід подається не символьний ряд, а послідовність числових значень. Для них текст розбивається на менші одиниці - токени (слова, частини слів). Кожен токен перетворюється на числовий вектор, який модель може обробляти математично. Процес перетворення тексту на токени називається ембеддингом (embedding) [1, с. 6]. Розглянемо приклад речення «Я люблю пити Кока-Колу», яке розбивається на окремі токени:

Слід зауважити, що ідентифікатори токенів залежать від словника конкретної моделі (наприклад, GPT-5 або Llama-4).

Текст	Я	люблю	пити	Кока	-	Колу
Токен	[251]	[14321]	[842]	[3210]	[12]	[4512]

Табл. 1.1: Приклад токенізації речення

Завдяки механізму уваги (self-attention), модель здатна враховувати котекст усього речення, абзацу та більше, розуміючи зв'язки між словами, навіть якщо вони далеко одне від одного [1, с. 16].

Процес навчання у мовних моделей називається попереднім навчанням (pretraining), оскільки вона не навчається в процесі роботи, а тільки перед початком використання. Модель навчається на величезних масивах тексту (книги, веб-сайти), вивчаючи граматику, факти про світ та логічні зв'язки без участі вчителя (self-supervised learning) [1, с. 26].

1.1.2 Архітектура Transformer та механізм самоуваги (Self-Attention)

Архітектура Transformer, представлена у 2017 році в статті «Attention Is All You Need», стала фундаментом для сучасних великих мовних моделей (LLM) [2]. Вона замінила рекурентні мережі (RNN), дозволивши тренувати моделі паралельно та значно ефективніше обробляти довгі послідовності даних.

Архітектура Трансформера сладається з двох блоків: енкодера (encoder) та декодера (decoder) складених один на одного, як показано на рис. 1.1.

Архітектура сучасних моделей базується на послідовному накладанні однакових структурних елементів. Основними будівельними одиницями виступають блоки трансформера (Transformer Blocks), які повторюються у структурі моделі велику кількість разів. Наприклад, у моделі GPT-3 кількість таких блоків (шарів) сягає 96 [3].

Кожен такий блок складається з двох основних частин:

1. Шар уваги (Attention Layer) - дозволяє моделі фокусуватися на різних частинах вхідної послідовності.

2. Нейронна мережа прямого поширення (Feedforward Neural Network - FFN/MLP) - здійснює обробку отриманих векторів.

Рис. 1.1: Спрощена схема взаємодії Encoder та Decoder в архітектурі Transformer

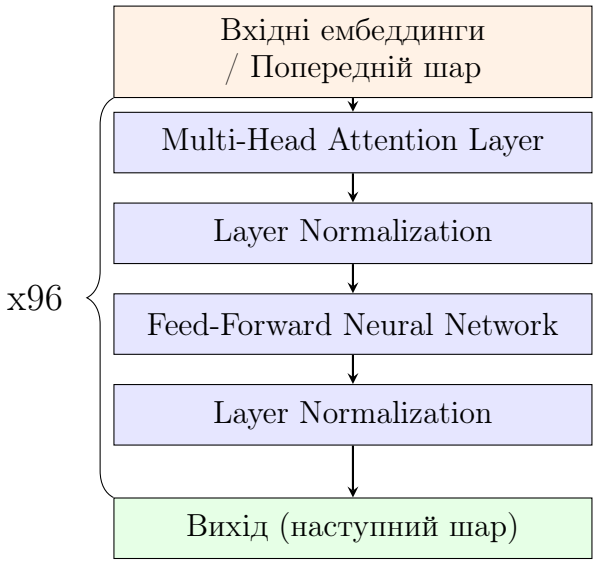

Рис. 1.2: Схематичне представлення ієрархічної структури блоків у моделі GPT-3

Механізм самоуваги (Self-Attention) дозволяє моделі фокусуватися на різних частинах вхідної послідовності під час обробки кожного токена. Це забезпечує глибоке розуміння контексту: наприклад, у реченні «Собака побіг за білкою, бо вона...» механізм уваги допомагає визначити, чи стосується займенник «вона» собаки чи білки, аналізуючи синтаксичні та семантичні зв'язки [1, с. 88].

Математично цей процес базується на трьох проекційних матрицях, що створюються під час навчання: Запит (Q – Query), Ключ (K – Key) та Значення (V – Value). Основна операція обчислення уваги виражається

формулою:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1.1}$$

де d_k – розмірність векторів ключів, що використовується для масштабування, а функція softmax нормалізує отримані бали, перетворюючи їх на ймовірності, сума яких дорівнює одиниці.

Функція softmax відіграє ключову роль у механізмі уваги, оскільки вона перетворює вектор довільних дійсних чисел z у вектор ймовірностей, де кожен елемент знаходиться в діапазоні $(0, 1)$. Математично вона визначається як:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{1.2}$$

де z_i – вхідний бал для i-го токена, а знаменник є сумою експонент усіх вхідних балів K токенів у послідовності. Це забезпечує те, що модель приділяє найбільшу «увагу» найбільш релевантним елементам, пригнічуючи шум від менш важливих токенів.

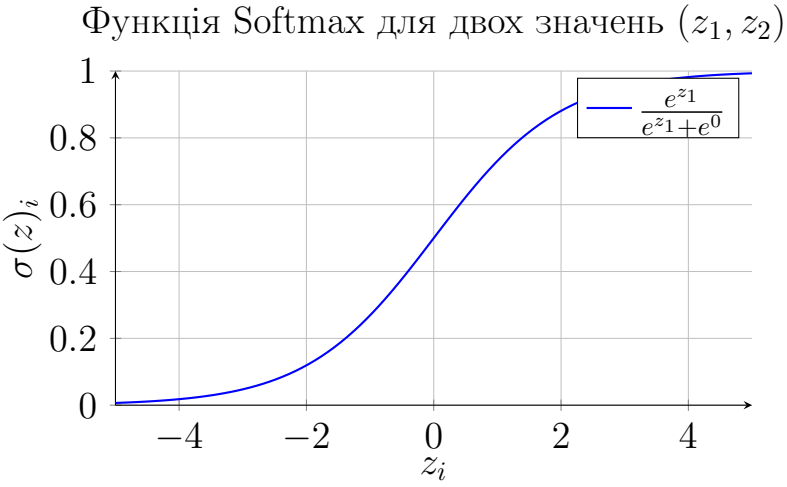

Рис. 1.3: Графічна інтерпретація функції Softmax

Для того щоб модель могла вловлювати різні типи зв'язків одночасно (наприклад, граматичні та смислові), використовується багатоголова увага (Multi-Head Attention). У цьому підході механізм самоуваги дублюється декілька разів. Кожна «голова» працює паралельно з власним набором матриць Q, K, V, після чого результати об'єднуються (конкатенуються) [1, с. 90].

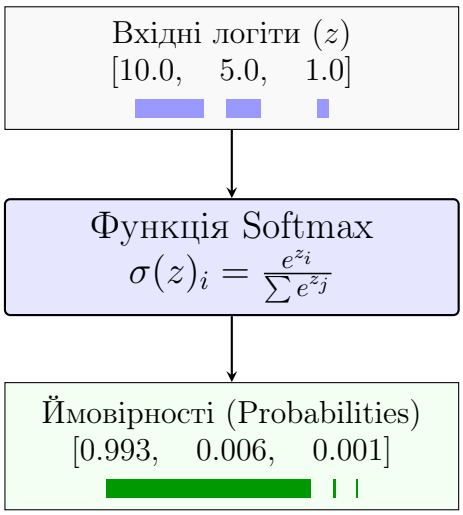

Рис. 1.4: Схема перетворення логітів у ймовірнісний розподіл функцією Softmax

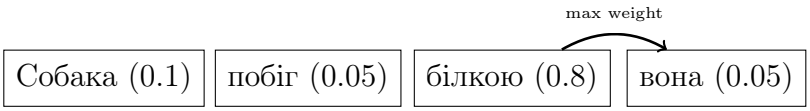

Рис. 1.5: Візуалізація розподілу ваг уваги після функції Softmax для слова «вона»

1.1.3 Параметри моделі та вагові коефіцієнти

Параметри — це внутрішні змінні моделі, які вона оптимізує в процесі навчання на масивах даних. Вони визначають функціональне перетворення вхідних даних у вихідні передбачення [1, с. 8]. Математично параметри сучасних архітектур складаються з двох основних компонентів: ваг (w) та зміщень (b).

Вагові коефіцієнти (Weights, w) становлять переважну більшість параметрів нейронної мережі. У сучасних архітектурах типу Transformer процес їх функціонування та зберігання має такі особливості [1, с. 8]:

- Принцип роботи: ваги організовані у вигляді проекційних матриць, зокрема матриць Запиту (W_Q), Ключа (W_K), Значення (W_V) та матриць повнозв'язних шарів прямого поширення (Feed-Forward layers). Вхідні дані у вигляді токенів множаться на ці матриці для формування нових контекстних представлень (рис. 1.6).

- Формати зберігання: традиційно ваги зберігаються як числа з плаваючою комою (floating-point numbers), наприклад, у

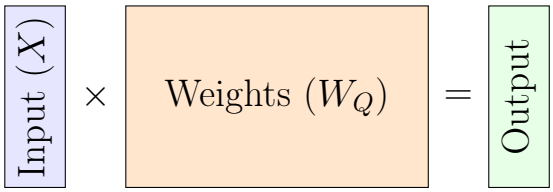

Рис. 1.6: Візуалізація процесу лінійного перетворення вхідних даних через матрицю ваг

форматах FP32 (32-біт) або FP16 (16-біт). Це забезпечує високу точність обчислень, проте вимагає значного обсягу пам'яті та обчислювальних потужностей.

Зміщення (Biases, b) — додаткові змінні, які дозволяють зміщувати результат активації нейрона, аналогічно до вільного члена в лінійному рівнянні $y = wx + b$ [4].

Масштаб сучасних моделей вимагає колосальних обчислювальних ресурсів. Наприклад, якщо GPT-1 налічувала 117 мільйонів параметрів, то GPT-3 вже має 175 мільярдів [1, с. 20]. Популярні моделі середнього розміру (наприклад, Llama-3-7B) містять близько 7 мільярдів параметрів. Саме ця кількість визначає інтелектуальні можливості моделі, але прямо пропорційно впливає на вимоги до обсягу оперативної пам'яті (ОЗП). Для зберігання моделі 7B у форматі FP32 потрібно близько 28 ГБ відеопам'яті, що робить її недоступною для більшості споживчих пристроїв без методів оптимізації.

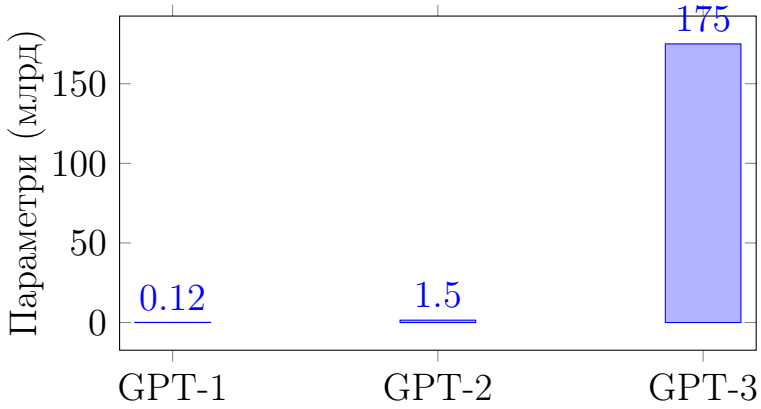

Рис. 1.7: Порівняння кількості параметрів у різних поколіннях моделей GPT

1.1.4 Представлення даних: від FP32 до цілочисельних типів

Перехід від чисел з плаваючою комою високої розрядності до цілочисельних типів зниженої точності є основою методів квантування (Quantization). Це критично важливо для оптимізації пропускної здатності пам'яті та енергоефективності при розгортанні LLM на пристроях з обмеженими ресурсами.

Еволюцію та градацію форматів представлення даних можна структурувати наступним чином:

1. FP32 (Single-Precision Floating-Point): стандарт для етапу навчання. Кожен параметр займає 4 байти (32 біти), що дозволяє зберігати значення з високою точністю та широким динамічним діапазоном [1, с. 201]. Проте для моделі з 7 млрд параметрів це потребує близько 28 ГБ пам'яті, що перевищує можливості більшості споживчих систем.

2. FP16 та BF16 (Half-Precision): проміжний етап, де розмір параметра зменшується до 2 байтів (16 бітів). Формат bfloat16 (Brain Floating Point) став стандартом для сучасних LLM, оскільки він зберігає той самий динамічний діапазон, що й FP32, жертвуючи лише точністю мантиси.

3. INT8 (8-бітні цілі числа): квантування дозволяє відобразити (map) континуум значень FP32 у дискретний діапазон цілих чисел від -128 до 127. Це зменшує об'єм моделі у 4 рази порівняно з FP32. При цьому цілочисельна арифметика виконується процесорами (CPU/NPU) значно швидше за арифметику з плаваючою комою [4].

4. INT4 та спеціалізовані формати (QLoRA): зниження розрядності до 4 біт дозволяє зберігати ваги моделі 7B у межах 3.5–4 ГБ VRAM [1, с. 367]. У методах типу QLoRA використовується нормалізований 4-бітний формат (NF4), що дозволяє підтримувати високу якість генерації при радикальному стисненні.

5. Тернарні та бінарні формати (BitNet 1.58b): екстремальний підхід,

де ваги обмежуються значеннями $\{-1, 0, 1\}$. Це потребує лише 1.58 біта на параметр і замінює енерговитратне множення матриць на операції додавання та віднімання, що є революційним для мобільних систем [6].

Для наочного порівняння ефективності різних форматів нижче наведено таблицю 1.2, що демонструє вимоги до пам'яті для моделі об'ємом 7 мільярдів параметрів.

Табл. 1.2: Вплив формату даних на споживання пам'яті моделі 7B

Тип даних	Біти	Об'єм (ГБ)	Сфера застосування
FP32	32	∼28.0	Навчання (Training)
FP16 / BF16	16	∼14.0	Стандартний інференс
INT8	8	∼7.0	Квантування для серверів
INT4	4	∼3.5	Споживчі ПК та смартфони
BitNet	1.58	∼1.4	Спеціалізовані пристрої

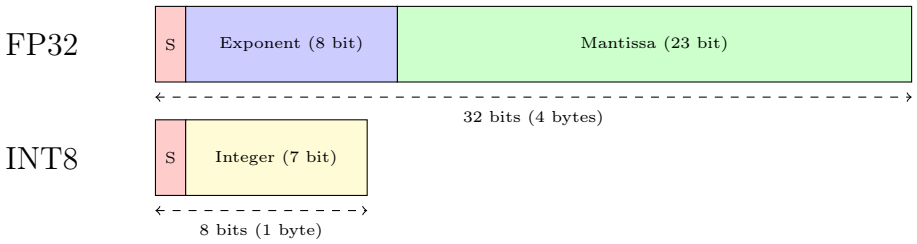

Рис. 1.8: Порівняння фізичного об'єму пам'яті форматів FP32 та INT8

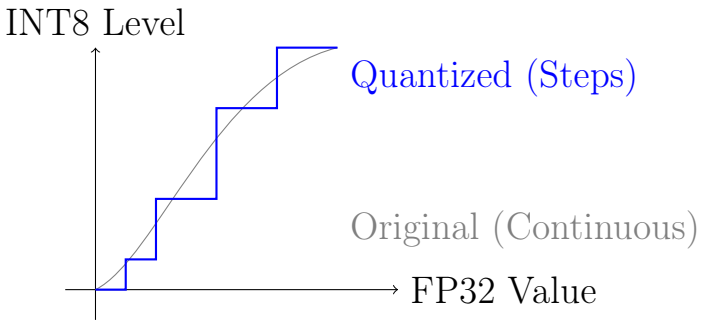

Рис. 1.9: Візуалізація дискретизації значень при переході до цілочисельного формату

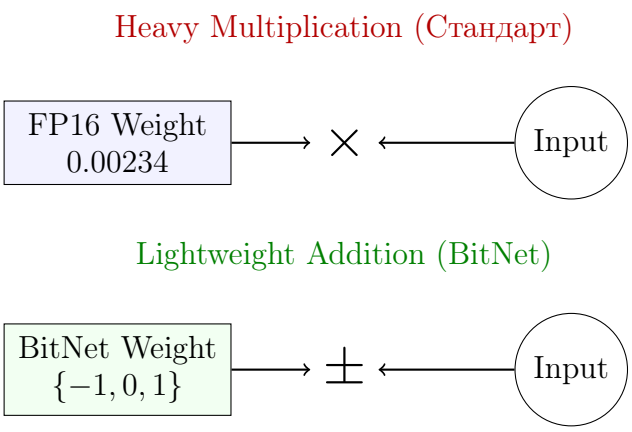

Рис. 1.10: Концептуальна різниця між стандартним обчисленням та тернарним підходом BitNet

1.2 Аналіз потреб LLM у ресурсах обчислювальних систем

1.2.1 Математична модель оцінки ресурсних потреб LLM

Для аналізу ефективності розгортання моделей необхідно попередньо визначити теоретичний об'єм необхідної пам'яті (M) та обчислювальну складність інференсу. Об'єм оперативної пам'яті, необхідний для зберігання ваг моделі, можна розрахувати за формулою:

$$M \approx \frac{P \times B}{8 \times 1024^3} + M_{ctx} \quad [\text{ГБ}] \tag{1.3}$$

де:

- P — кількість параметрів моделі (наприклад, 8×10^9);
- B — кількість бітів на один параметр (FP32 = 32, FP16 = 16, INT4 = 4);
- M_{ctx} — додаткова пам'ять для контекстного вікна (KV-cache), що залежить від довжини вхідної послідовності.

Обчислювальна складність генерації одного токена (в операціях FLOPs) для стандартної архітектури Transformer наближено дорівнює:

$$C \approx 2 \times P \quad [\text{FLOPs/token}] \tag{1.4}$$

Це означає, що для моделі Llama-3-8B генерація одного слова потребує приблизно 16 мільярдів операцій з плаваючою комою. Використання цілочисельного квантування (INT8/INT4) дозволяє замінити операції типу floating-point на менш ресурсомісткі integer операції, що математично виражається через прискорення обчислень у N разів, де N — коефіцієнт ефективності векторних інструкцій процесора (наприклад, AVX-512 або Apple NEON).

1.2.2 Пропускна здатність пам'яті як системне обмеження

Критичним показником для системного аналізу є не лише об'єм, а й швидкість доступу до даних. Теоретична швидкість генерації (T_{gen}) обмежена пропускною здатністю пам'яті (Memory Bandwidth):

$$T_{gen} = \frac{BW}{M_{weights}} \quad [\text{токенів/сек}] \tag{1.5}$$

де BW — пропускна здатність шини пам'яті (ГБ/сек), а $M_{weights}$ — вага моделі в ГБ. Ця формула пояснює, чому квантування в 4 рази (з FP16 до INT4) не лише економить місце, а й потенційно в 4 рази прискорює генерацію тексту на пристроях з повільною RAM.

1.2.3 Методологія та інструментарій дослідження

Експериментальне тестування проводилося на локальній обчислювальній системі з використанням спеціально розробленого програмного інструментарію на мові Python. Скрипт інтегрувався з API сервера Ollama для отримання метаданих моделей та використовував системні бібліотеки psutil та GPUtil для фіксації пікових показників у режимі реального часу.

Об'єктом дослідження виступили моделі різних архітектур (Llama, Qwen, Mistral, Gemma) з об'ємом параметрів від 1B до 8.2B, що використовують 4-бітне квантування (quantization level Q4_K_M або Q4_0).

1.2.4 Експериментальні дані та аналіз результатів

Для збору експериментальних даних та верифікації теоретичних розрахунків було розроблено програмний інструментарій на мові Python. Програма здійснює моніторинг апаратних ресурсів (CPU, GPU, RAM) у реальному часі через взаємодію з системними метриками та API сервера Ollama. Повний текст програмного коду бенчмарку наведено у додатку А.

У ході бенчмаркінгу було зафіксовано показники, наведені у таблиці 1.3.

Табл. 1.3: Результати аналізу споживання системних ресурсів локальними LLM

Модель	Параметри	Квант.	CPU (%)	GPU (%)	VRAM (МБ)
TinyLlama	1B	Q4_0	8.0	90.0	3341
Gemma3	4.3B	Q4_K_M	36.3	40.0	3341
Qwen3	4.0B	Q4_K_M	49.9	97.0	3444
Mistral	7.2B	Q4_K_M	46.9	95.0	3389
Llama 3.1	8.0B	Q4_K_M	49.8	100.0	3339
Qwen3	8.2B	Q4_K_M	59.4	37.0	3494
DeepSeek-R1	8.2B	Q4_K_M	51.4	38.0	3318

1.2.5 Інтерпретація результатів аналізу

На основі отриманих даних (рис. 1.11) було виявлено кілька ключових закономірностей, що мають критичне значення для аналізу:

1. Стабілізація об'єму VRAM: Спостерігається відносна сталість зайнятої відеопам'яті (близько 3.3–3.5 ГБ) незалежно від кількості параметрів. Це вказує на агресивне управління пам'яттю в середовищі Ollama та ефективність 4-бітного квантування, яке дозволяє уніфікувати вимоги до сховища для моделей середнього класу.

2. Ефект «обчислювального порогу» (Offloading): Найбільш показовим є результат моделей Qwen3:8b та DeepSeek-R1:8b. Попри те, що вони належать до того ж вагового класу, що й Llama 3.1, навантаження на GPU різко падає до 37–38%, тоді

як навантаження на CPU зростає до максимуму (59.4%). Це свідчить про автоматичне перенесення обчислювальних шарів у системну RAM через вичерпання ресурсів швидкої пам'яті GPU, що призводить до суттєвого падіння продуктивності.

3. Оптимізація архітектур: Модель Llama 3.1:8b продемонструвала найвищий рівень утилізації GPU (100%), що робить її еталонною з точки зору балансу між кількістю параметрів та ефективністю використання апаратного прискорення.

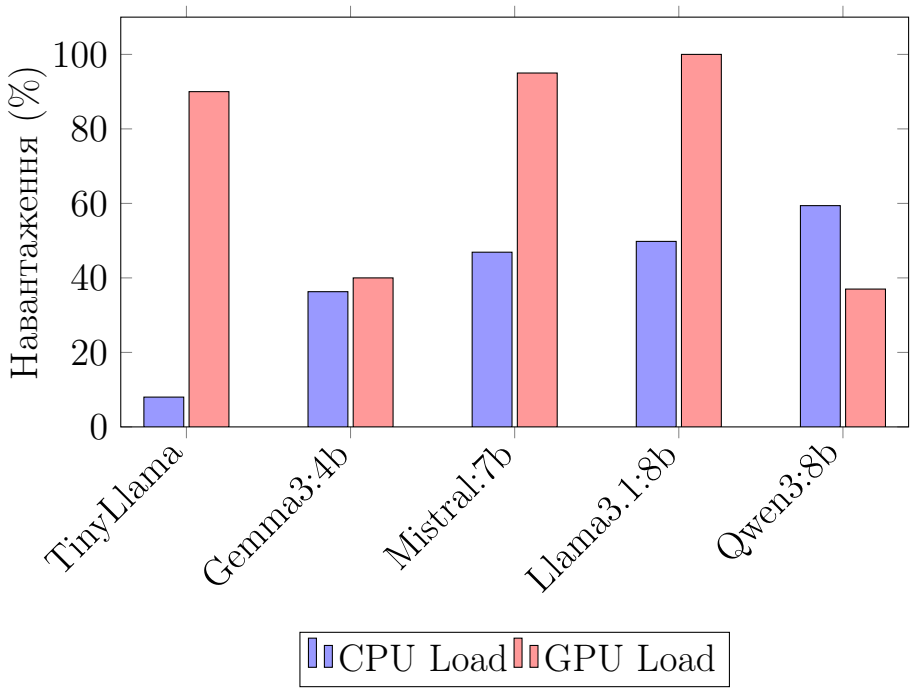

Рис. 1.11: Порівняльний розподіл навантаження на апаратні ресурси системи

Таким чином, проведений аналіз підтверджує, що для успішного розгортання моделей об'ємом понад 8B на споживчих пристроях стандартного квантування (Q4_K_M) недостатньо. Це обґрунтовує необхідність дослідження методів інтенсивнішої оптимізації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Alammar J., Grootendorst M. Hands-On Large Language Models: Language Understanding and Generation. O'Reilly Media, 2024. 412 p.

2. Vaswani A., Shazeer N., Parmar N. et al. Attention Is All You Need. arXiv:1706.03762 [cs.CL]. 2017. URL: https://arxiv.org/abs/1706.03762 (дата звернення: 17.02.2026).

3. Transforming agency: On the mode of existence of large language models - Scientific Figure on ResearchGate. URL: https://www.researchgate.net/figure/Schematic-of-the-GPT-3-processing-architecture-as-a-standardized-reference-for fig1_394849833 (дата звернення: 17.02.2026)

4. Jacob B., Kligys S., Chen B. et al. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. arXiv:1712.05877 [cs.LG]. 2017. URL: https://arxiv.org/abs/1712.05877 (дата звернення: 18.02.2026).

5. Cheng Y., Wang D., Zhou P., Zhang T. A Survey of Model Compression and Acceleration for Deep Neural Networks. arXiv:1710.09282 [cs.LG]. 2020. URL: https://arxiv.org/abs/1710.09282 (дата звернення: 18.02.2026).

6. Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, Furu Wei. The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits. arXiv:2402.17764 [cs.CL]. 2024. URL: https://arxiv.org/abs/2402.17764 (дата звернення: 23.02.2026).

ДОДАТОК А

Програмний код модуля моніторингу системних ресурсів

Лістинг 1: Скрипт для автоматизованого бенчмаркінгу LLM

```
import psutil
import GPUtil
import time
import requests
import pandas as pd
import threading

OLLAMA_URL = "http://localhost:11434/api"

def get_system_stats():
    cpu = psutil.cpu_percent()
    ram = psutil.virtual_memory().percent
    gpu_load, vram_used = 0, 0
    gpus = GPUtil.getGPUs()
    if gpus:
        gpu_load = gpus[0].load * 100
        vram_used = gpus[0].memoryUsed
    return cpu, ram, gpu_load, vram_used

def get_model_details(model_name):
    try:
        resp = requests.post(f"{OLLAMA_URL}/show", json={"name":
            ↪ model_name})
        data = resp.json()
        details = data.get('details', {})
        return {
            "size": details.get('parameter_size', 'N/A'),
            "quant": details.get('quantization_level', 'N/A'),
            "family": details.get('family', 'N/A')
        }
    except:
        return {"size": "N/A", "quant": "N/A", "family": "N/A"}

def benchmark():
    models_resp = requests.get(f"{OLLAMA_URL}/tags").json()
    model_list = [m['name'] for m in models_resp['models']]
```

```
    results = []

    for model_name in model_list:
        print(f"\n>>> Testing: {model_name}")
        details = get_model_details(model_name)

        # Container for peak values
        peaks = {"cpu": 0, "ram": 0, "gpu": 0, "vram": 0}
        stop_monitor = False

        def monitor():
            while not stop_monitor:
                c, r, g, v = get_system_stats()
                peaks["cpu"] = max(peaks["cpu"], c)
                peaks["ram"] = max(peaks["ram"], r)
                peaks["gpu"] = max(peaks["gpu"], g)
                peaks["vram"] = max(peaks["vram"], v)
                time.sleep(0.5)

        # Start monitoring in background
        monitor_thread = threading.Thread(target=monitor)
        monitor_thread.start()

        try:
            # Increased timeout to 300s for heavy models like DeepSeek
            requests.post(f"{OLLAMA_URL}/generate",
                          json={"model": model_name, "prompt": "Why is system
                              ↪ analysis important?", "stream": False},
                          timeout=300)
        except Exception as e:
            print(f"    Error: {e}")
        finally:
            stop_monitor = True
            monitor_thread.join()

        print(f"    [{model_name}] Quant: {details['quant']} | Peak GPU:
            ↪ {peaks['gpu']}% | VRAM: {peaks['vram']}MB")

        results.append({
            "Model": model_name,
            "Family": details['family'],
```

```
75              "Params": details['size'],
76              "Quantization": details['quant'],
77              "Peak_CPU_%": peaks["cpu"],
78              "Peak_RAM_%": peaks["ram"],
79              "Peak_GPU_%": peaks["gpu"],
80              "Peak_VRAM_MB": peaks["vram"]
81          })
82
83      # Save results
84      df = pd.DataFrame(results)
85      df.to_csv("llm_system_analysis_results.csv", index=False)
86      print("\n[Done] Data saved to llm_system_analysis_results.csv")
87
88  if __name__ == "__main__":
89      benchmark()
```