Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційних радіотехнологій і технічного захисту інформації
(повна назва)

Кафедра Радіотехнологій інформаційно-комунікаційних систем
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів забезпечення можливостей інференса для LLM за допомогою NVIDIA Triton
(тема)

Виконав:
студент 1 курсу, групи АПСм-22-1
Верколаб Г.С.
(прізвище, ініціали)

Спеціальність 126 Інформаційні системи та технології
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Архітектурне проектування інформаційних систем
(повна назва освітньої програми)

Керівник проф. Кузьомін О.Я.
(посада, прізвище, ініціали)

Допускається до захисту

В.о. зав. кафедри _____ Зарудний О.А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційних радіотехнологій і технічного захисту інформації

Кафедра Радіотехнологій інформаційно-комунікаційних систем

Рівень вищої освіти другий (магістерський)

Спеціальність 126 Інформаційні системи та технології
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Архітектурне проектування інформаційних систем
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«____»_____ 20 ___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Верколабу Глібу Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів забезпечення можливостей інференса для llm за допомогою nvidia triton

затверджена наказом університету від 3 жовтня 2023 р. № 1295 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 8 січня 2024 р.

3. Вихідні дані до роботи аналіз великих мовних моделей, аналіз методів покращення інференса, розгортання моделі з nvidia triton

4. Перелік питань, що потрібно опрацювати в роботі системи, що надають можливості виводу для великих мовних моделей (LLM) за допомогою технологій NVIDIA Triton Inference Server

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) ____
Слайди комп'ютерної презентації

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	Дата
Основна частина	проф. Кузьомін О.Я.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Ознайомлення із завданням, ТЗ.	4.10 - 19.10.2023	Виконано
2	Підбір літератури за темою роботи	20.10 - 24.10.2023	Виконано
3	Аналіз великих мовних моделей	25.10 - 2.11.2023	Виконано
4	Огляд технологій оптимізації інференсу	03.11 - 10.11.2023	Виконано
5	Вибір придатних технологій	11.11 - 19.11.2023	Виконано
6	Написання програмного коду	20.11 - 13.12.2023	Виконано
7	Оформлення пояснювальної записки	14.12 - 25.12.2023	Виконано
8	Підготування презентаційного матеріалу	26.12.2023 - 13.01.2024	Виконано
9	Представлення роботи на кафедру	14.01.2024	Виконано

Дата видачі завдання __4__ __жовтня__ 2023 р.

Студент _____
(підпис)

Керівник роботи _____ ____проф. Кузьомін О.Я._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка містить: 78 сторінок, 9 рисунків, 3 таблиці, 23 джерел посилання та 3 додатки.

СИСТЕМА. МОДЕЛЬ. ІНФЕРЕНС. ОПТИМІЗАЦІЯ. СЕРВЕР.

Актуальність. У процесі розробки системи з кодами на Python для великих мовних моделей (LLM) з використанням NVIDIA Triton Inference Server у виробничому середовищі використовується масштабованість та ефективність Triton Inference Server, що дозволиляє обробляти велику кількість запитів на інференс LLM, мінімізуючи затримки. Система є універсальною, підтримує різні архітектури LLM, що робить її пристосованою до різноманітних завдань розуміння та генерації природної мови. Процес розгортання спрощений, і наша система легко інтегрується з існуючими програмами на Python. Пріоритетом є безпека з комплексним моніторингом і веденням логів, при цьому пропонуючи можливості для кастомізації та оптимізації. Реальне застосування в різних галузях підкреслює універсальність системи, а її автоматизація та ефективність приносять відчутні переваги, знижуючи операційні витрати та покращуючи досвід користувачів. Загалом, система дає змогу організаціям використовувати можливості LLM у виробничому середовищі, що є значним кроком на шляху до інтелектуальної, автоматизованої та ефективної обробки природної мови.

Постановка задачі. Система, яка використовує коди Python та NVIDIA Triton Inference Server для виведення LLM у виробничому середовищі, забезпечує масштабованість, ефективність, універсальність, безпеку та можливості моніторингу. Вона спрощує розгортання та управління LLM, що робить її цінним інструментом для широкого спектру додатків для обробки природної мови, від чат-ботів і віртуальних асистентів до аналізу настроїв і генерації контенту, серед багатьох інших.

Практична цінність дослідження. Системи, які надають можливості виводу

для великих мовних моделей (LLM) за допомогою таких технологій, як NVIDIA Triton Inference Server, є цінними та корисними для різних людей, організацій та галузей.

Підтримка та залучення клієнтів: Компанії можуть використовувати магістрів права для розробки чат-ботів і віртуальних асистентів для забезпечення цілодобової підтримки клієнтів і взаємодії з ними на веб-сайтах і в додатках.

Створення контенту: Підприємства можуть використовувати LLM для автоматизації генерації контенту для маркетингу, звітів, описів продуктів тощо.

Клінічна документація: Медичні установи можуть використовувати LLM для автоматичного створення клінічної документації та медичних звітів, зменшуючи адміністративні витрати.

Аналіз ризиків: Фінансові установи можуть використовувати LLM для аналізу ризиків, виявлення шахрайства та автоматизованої підтримки клієнтів за поширеними запитами.

Аналіз тексту: Дослідники та науковці можуть використовувати магістерські програми для розуміння природної мови, аналізу настроїв та узагальнення текстів.

Корисність LLM і систем, що забезпечують можливості виведення, дуже широка і може бути адаптована до широкого спектру галузей і застосувань. Автоматизуючи і вдосконалюючи різні аспекти бізнес-операцій і послуг, ці системи можуть підвищити ефективність, знизити витрати і забезпечити кращий досвід для клієнтів і користувачів.

ABSTRACT

The explanatory note contains: 78 pages, 9 figures, 3 tables, 23 references and 3 appendices.

SYSTEM. MODEL. INFERENCE. OPTIMIZATION. SERVER.

Relevance. In the process of developing a system with Python codes for large language models (LLMs) using NVIDIA Triton Inference Server in a production environment, the scalability and efficiency of Triton Inference Server is used to process a large number of LLM output requests while minimising delays. The system is versatile, supporting different LLM architectures, making it adaptable to a variety of natural language understanding and generation tasks. The deployment process is simplified, and our system can be easily integrated with existing Python applications. Security is a priority with comprehensive monitoring and logging, while offering opportunities for customisation and optimisation. Real-world applications in various industries highlight the system's versatility, and its automation and efficiency bring tangible benefits by reducing operating costs and improving user experience. Overall, the system enables organisations to leverage the power of LLM in a production environment, which is a significant step towards intelligent, automated and efficient natural language processing.

Keywords – Large Language Models, LLM inference, TensorFlow, TensorRT, Triton Inference Server.

Subject of study. A system that uses Python and NVIDIA Triton Inference Server codes to infer LLMs in a production environment provides scalability, efficiency, versatility, security, and monitoring capabilities. It simplifies the deployment and management of LLM, making it a valuable tool for a wide range of natural language processing applications, from chatbots and virtual assistants to sentiment analysis and content generation, among many others.

Practical value. Systems that provide inference capabilities for large language

models (LLMs) using technologies such as NVIDIA Triton Inference Server are valuable and useful to a variety of people, organisations, and industries.

Customer support and engagement: Companies can use LLMs to develop chatbots and virtual assistants to provide round-the-clock customer support and interaction on websites and applications.

Content creation: Businesses can use LLMs to automate the generation of content for marketing, reports, product descriptions, and more.

Clinical documentation: Healthcare providers can use LLM to automatically generate clinical documentation and medical reports, reducing administrative costs.

Risk analysis: Financial institutions can use LLM for risk analysis, fraud detection, and automated customer support for common queries.

Text analysis: Researchers and scientists can use LLM programmes to understand natural language, analyse sentiment and summarise texts.

The usefulness of LLMs and systems that provide inference capabilities is very broad and can be adapted to a wide range of industries and applications. By automating and improving various aspects of business operations and services, these systems can increase efficiency, reduce costs and provide a better experience for customers and users.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

GPT – Generative Pre-trained Transformer.

LLM – Large Language Models.

BERT – Bidirectional Encoder Representations from Transformers.

RoBERTa – Robustly Optimized BERT Approach.

MPT – MosaicML Pretrained Transformer.

LLaMA – Large Language Model Meta AI.

NLP – Natural Language Processing.

DL – Deep learning.

ШІ – штучний інтелект.

ONNX – Open Neural Network Exchange.

TTFT – Time To First Token.

TPOT – Time Per Output Token.

FP – floating-point.

KV – key-value.

MBU – Model Bandwidth Utilization.

MFU – Model Flops Utilization.

QPS – Queries per second.

OOM – Out-of-Memory.

GQA – Grouped Query Attention

MQA – Multi-Query-Attention.

MHA – masked multi-head attention.

CUDA – Compute Unified Device Architecture.

ВСТУП

Останніми роками великі мовні моделі (LLM), такі як GPT-3, привернули до себе багато уваги завдяки своїй здатності генерувати відповіді на текстові рядки, подібні до людських. Однією з ключових особливостей, яка робить ці моделі такими потужними, є їхня здатність робити відповіді на основі контексту та вхідних даних.

Що таке інференс у великій мовній моделі?

У контексті великої мовної моделі інференс – це здатність моделі генерувати передбачення або відповіді на основі контексту та вхідних даних. Коли модель отримує підказку, вона використовує своє розуміння мови та контексту, щоб згенерувати релевантну та доречну відповідь.

Наприклад, якщо користувач вводить «Я зголоднів, може, мені замовити...», а LLM генерує завершення «піцу», він зробив інференс на основі контексту вхідного тексту. Аналогічно, у завданнях перекладу модель використовує своє розуміння контексту та синтаксису вхідної мови, щоб створити відповідний переклад цільовою мовою.

Як LLM робить інференс?

LLM використовують різні методи, щоб робити виведення на основі вхідних даних. Один із ключових прийомів, який використовують ці моделі, називається увагою. Увага дозволяє моделі зосередитися на певних частинах вхідного тексту під час генерації відповіді. Це може допомогти моделі краще зрозуміти контекст і генерувати більш точні відповіді.

Інша техніка, яку використовують LLM, відома як архітектура трансформера. Ця архітектура дозволяє моделі обробляти вхідний текст ієрархічно, що може допомогти їй краще зрозуміти взаємозв'язки між різними

частинами тексту. Це може допомогти моделі генерувати більш точні та контекстуально релевантні відповіді.

Чому інференс важливий в LLM?

Інференс, LLM-виведення або «генеративна відповідь» важлива складова у великих мовних моделях, оскільки він дозволяє моделі генерувати відповіді, які є більш релевантними та доречними на основі контексту вхідного тексту. Це може бути особливо корисно в таких завданнях, як переклад і обробка природної мови, де значення вхідного тексту може бути складним і багатозначним.

Крім того, здатність робити виведення дозволяє LLM генерувати відповіді, більш схожі на людські за своєю природою. Це може бути важливо в таких додатках, як чат-боти та віртуальні асистенти, де метою є створення більш природного та інтуїтивно зрозумілого користувацького досвіду.

Виведення інформації, що являється інференсом, є ключовою особливістю великих мовних моделях, таких як GPT-3. Ці моделі використовують різноманітні методи, щоб робити виведення на основі контексту та вхідних даних. Здатність робити точні та релевантні виведення має важливе значення в таких завданнях, як переклад і обробка природної мови, і може допомогти створити більш схожі на людські відповіді в таких додатках, як чат-боти і віртуальні асистенти.

1 АНАЛІЗ І ПОРІВНЯННЯ АРХІТЕКТУР LLM

1.1 Порівняння LLM (Large Language Models)

Моделі такі як GPT-3, BERT, RoBERTa, MPT, LLaMa v2, передбачають розгляд різних аспектів, таких як архітектура, цілі попереднього навчання, розмір моделі, навчальні дані та типові випадки використання. Нижче наводиться порівняльний аналіз цих моделей:

GPT-3 (Generative Pre-trained Transformer 3):

Архітектура: використовує архітектуру трансформатора з унікальним «авторегресійним» підходом, де модель генерує текст по одному токену за раз, передбачаючи ймовірність кожного токену на основі попередніх. Явного кодування вхідного тексту не виконується.

Розмір моделі: GPT-3 одна з найбільших мовних моделей, що містить 175 млрд параметрів.

Навчальні дані: тренується на великому масиві текстових даних, включаючи книги, статті та веб-сайти, з акцентом на максимізацію ймовірності наступного токена в послідовності.

Продуктивність: демонструє вражаючу продуктивність у різних завданнях генерації природної мови, таких як заповнення тексту, відповіді на запитання та чат-боти. Однак він може не впоратися із завданнями, що вимагають точного контролю над синтаксисом і семантикою.

Обчислювальні ресурси: вимагає значних обчислювальних ресурсів, зокрема потужного обладнання та великого обсягу пам'яті.

Інференс: через свою авторегресійну природу GPT-3 має повільніший час виведення порівняно з іншими моделями. Однак згенерований нею текст може бути більш якісним і зв'язним.

BERT (Bidirectional Encoder Representations from Transformers):

Архітектура: використовує багатошаровий двонаправлений кодер трансформер для створення контекстних зображень слів у вхідному тексті. Ці

представлення потім тонко налаштовуються для конкретних подальших завдань NLP.

Завдання попереднього навчання: BERT навчається на моделі мови з маскою, де він вчиться передбачати пропущені слова у реченні, враховуючи як лівий, так і правий контекст.

Розмір моделі Моделі BERT можуть бути різних розмірів, при цьому оригінальна модель BERT має 110 млн параметрів.

Навчальні дані: BERT навчається на комбінації BookCorpus та англійської Вікіпедії, навчений на великому масиві текстових даних, включаючи книги, статті та веб-сайти. Налаштована на конкретні подальші завдання NLP.

Продуктивність: досягає найсучасніших результатів у широкому спектрі завдань NLP, включаючи відповіді на запитання, аналіз настроїв, розпізнавання іменованих об'єктів і класифікацію текстів.

Обчислювальні ресурси: потребує помірних обчислювальних ресурсів, зокрема графічного процесора з щонайменше 4 ГБ VRAM.

Інференс: досягає швидкого часу виведення завдяки попередньо розрахованим вбудовуванням та архітектурі нейронної мережі з поверхневою структурою.

RoBERTa (Robustly Optimized BERT Pretraining Approach):

Архітектура: спирається на BERT, додаючи додаткові методи, такі як доповнення даних, змагальне навчання та більший розмір моделі, що призводить до покращення продуктивності при виконанні деяких завдань NLP(Natural Language Processing).

Мета попереднього навчання: подібно до BERT, RoBERTa використовує масковану мету мовної моделі для навчання двонаправленим уявленням.

Розмір моделі: моделі RoBERTa розрізняються за розміром, причому «базова» модель має 125 млн. параметрів.

Навчальні дані: навчений на великому масиві текстових даних, включаючи книги, статті та веб-сайти, з використанням комбінації маскованого мовного моделювання та завдань передбачення наступного речення. Налаштовується на

конкретні подальші завдання NLP.

Продуктивність: перевершує BERT у виконанні деяких завдань NLP, особливо тих, що вимагають більш далеких залежностей і більш тонкого розуміння контексту.

Обчислювальні ресурси: вимагає помірних обчислювальних ресурсів, подібних до BERT.

Інференс: Час виведення трохи повільніший, ніж у BERT, через більший розмір моделі та складнішу архітектуру.

MPT (MosaicML Pretrained Transformer):

Архітектура: використовує ієрархічну архітектуру, що складається з кластерних трансформаторних кодерів, за якими слідує серіалізований трансформаторний декодер. Кодери об'єднані в кластери, кожен з яких містить кілька шарів, а вихідні дані проходять через серію лінійних перетворень перед подачею на декодер.

Розмір моделі: розроблений для обчислювальної ефективності, з меншою кількістю параметрів і простішою архітектурою. Це призводить до швидшого навчання та виведення висновків, але може обмежувати його здатність охоплювати складні лінгвістичні явища. Має моделі різних розмірів.

Навчальні дані: навчання відбувалося за допомогою комбінації маскованого мовного моделювання, передбачення наступного речення та контрастної навчальної задачі. Модель була попередньо навчена на наборі даних обсягом 50 ГБ тексту та доопрацьована на наступних завданнях.

Продуктивність: демонструє конкурентоспроможну продуктивність на різних завданнях NLP, зберігаючи при цьому нижчі обчислювальні вимоги порівняно з іншими моделями. На бенчмарку GLUE MPT досягає середнього показника F1 93,5%.

Обчислювальні ресурси: потребує менше обчислювальних ресурсів порівняно з GPT-3 і RoBERTa, з можливістю запуску на CPU або GPU з обмеженою VRAM.

Інференс: оптимізовано швидкість виведення та використання пам'яті, але

з акцентом на мінімізацію обчислювальних вимог без шкоди для продуктивності. Досягає низьких затримок на NVIDIA Triton Inference Server, з затримками від 2 мс для партій до 64.

LLaMA(Large Language Model Meta AI) v2:

Архітектура: Використовує модульну архітектуру, яка об'єднує кілька трансформаторних кодерів і декодерів в одну модель. Кожна пара кодерів і декодерів утворює «модуль», який можна незалежно масштабувати та оптимізувати.

Розмір моделі: має більшу кількість параметрів (приблизно 1,4 мільярда) порівняно з MPT (близько 130 мільйонів). Ця підвищена складність дозволяє LLaMA v2 вловлювати більш тонкі закономірності в мові, але також вимагає більше обчислювальних ресурсів для навчання та висновків. Також має моделі інших розмірів.

Навчальні дані: навчання відбувалося за допомогою комбінації маскованого мовного моделювання, передбачення наступного речення та навчальної задачі з кількома пострілами під назвою «LAMA». Модель була попередньо навчена на наборі даних обсягом 75 ГБ тексту і допрацьована на наступних завданнях.

Продуктивність: досягає найсучаснішої продуктивності на різних тестових наборах даних, таких як GLUE, SuperGLUE і LAMA. У тесті SuperGLUE LLaMA v2 досягає середнього показника F1 95,5% для всіх завдань.

Обчислювальні ресурси: вимагає менше обчислювальних ресурсів у порівнянні з GPT-3 і RoBERTa, але все одно потребує графічного процесора з принаймні 4 ГБ VRAM.

Інференс: також оптимізовано за швидкістю виведення та використанням пам'яті, що робить його придатним для розгортання на периферійних пристроях і мобільних платформах. Досягає швидкого часу виведення на сервері виведення NVIDIA Triton Inference Server із затримками від 3 мс для партій розміром до 64.

Заключення аналізу:

Розмір моделі: GPT-3 – найбільша, великий розмір моделі GPT-3 дозволяє

їй генерувати більш зв'язковий та контекстуально насичений текст, проте це пов'язано з більшими обчислювальними витратами.

Тонка настройка: Моделі BERT та RoBERTa можуть бути тонко налаштовані на вирішення конкретних завдань, що робить їх універсальними для різних програм.

Зрештою вибір між цими моделями залежить від конкретної задачі NLP, доступних обчислювальних ресурсів та компромісу між розміром моделі та її продуктивністю. GPT-3 і RoBERTa можуть бути гарним вибором, якщо потрібна модель, яка може впоратися з широким спектром завдань NLP і генерувати високоякісний текст. BERT і LLaMA v2 – хороші варіанти, якщо потрібна модель, яка може впоратися з різноманітними завданнями NLP і має менший обчислювальний слід. MPT може бути хорошим вибором, якщо потрібна модель, яка може обробляти різноманітні завдання NLP, але при цьому має низькі обчислювальні вимоги.

1.2 Формат моделі DL, сумісний з triton inference server

NVIDIA Triton Inference Server підтримує кілька форматів моделей, включаючи TensorFlow, ONNX та PyTorch. Вибір формату залежить від різних факторів, таких як існуючий формат моделі, простота конвертації та специфічні вимоги до розгортання. Наведемо порівняння цих форматів:

TensorFlow плюси:

– Triton Inference Server має вбудовану підтримку моделей TensorFlow, що полегшує їх розгортання.

– TensorFlow широко використовується, і багато попередньо навчені моделі та фреймворки доступні у форматі TensorFlow, наприклад TensorRT.

– TensorFlow пропонує інструменти оптимізації моделей для виведення, такі як TensorFlow Serving.

Мінуси:

Якщо модель має інший формат, вам може знадобитися перетворити її на

TensorFlow, що може бути додатковим кроком.

Приклад використання:

TensorFlow – це оптимальний вибір, якщо у вас є готові до розгортання моделі TensorFlow або якщо ви плануєте використовувати попередньо навчені моделі TensorFlow.

ONNX (Open Neural Network Exchange) плюси:

– ONNX це відкритий стандарт, призначений для взаємодії різних фреймворків глибокого навчання, що робить його універсальним вибором.

– Triton Inference Server має вбудовану підтримку моделей ONNX.

– ONNX дозволяє конвертувати моделі з різних фреймворків глибокого навчання у загальний формат.

Мінуси:

Засоби конвертації ONNX можуть підтримувати не всі архітектури та операції над моделями, тому складні моделі можуть вимагати додаткової роботи.

Приклад використання:

ONNX – хороший варіант, якщо вам потрібно перетворити моделі з кількох фреймворків на загальний формат для розгортання в Triton.

PyTorch плюси:

– Якщо ваші моделі представлені у форматі PyTorch, їх розгортання без конвертації може бути зручним.

– PyTorch популярний для досліджень та розробок, тому у вас можуть бути доступні моделі в цьому форматі.

Мінуси:

Triton Inference Server не підтримує моделі у форматі PyTorch. Може знадобитися перетворення моделей PyTorch на сумісний формат, наприклад, ONNX або TensorFlow.

Приклад використання:

Якщо у вас є моделі PyTorch, але ви не хочете конвертувати їх, ви можете виконати перетворення в TensorFlow або ONNX перед їх розгортанням в Triton.

Загалом вибір формату залежить від конкретного сценарію. Якщо вже є моделі у певному форматі, то практичнішим буде вибір формату, який відповідає існуючим моделям. Однак, якщо необхідно перетворити моделі, слід враховувати такі фактори, як простота перетворення та можливості інструментів перетворення, доступних для конкретних моделей. TensorFlow є гарним вибором, якщо пріоритетом є сумісність різних фреймворків, але ONNX та PyTorch також є надійними варіантами, якщо вони відповідають існуючому робочому процесу та моделям.

2 ОГЛЯД ІНФЕРЕНСА ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

2.1 Розуміння генерації текстів LLM

Великі мовні моделі (LLM) генерують текст у два етапи: «попереднє заповнення», коли «токени» (лексеми) у вхідному запиті обробляються паралельно, і «декодування», коли текст генерується по одному токену за раз в авторегресійному режимі. Кожен згенерований токен додається до вхідних даних і повертається в модель для генерації наступного токену. Генерація зупиняється, коли LLM виводить спеціальний стоп-токен або коли виконується визначена користувачем умова (наприклад, згенеровано деяку максимальну кількість токенів). По суті, генеративний інференс LLM генерує текстові результати на основі вхідних даних шляхом ітеративного передбачення наступного токена в послідовності.

Токени можуть бути словами або підсловами; точні правила розбиття тексту на токени варіюються від моделі до моделі.

Ці моделі зазвичай приймають на вхід послідовність цілих чисел, які представляють послідовність токенів (слів/частин слів), і генерують передбачення наступного токена, який буде видан. Нижче наведено простий приклад, який ілюструє це в коді:

```
# Vocabulary of tokens the model can parse. The position of each token in the
# vocabulary is used as the token_id (an integer representing that token)
vocab = ["having", "I", "fun", "am", "learning", ".", "Neuron"]
# input token_ids: list of integers that represent the input tokens in this
# case: "I", "am", "having", "fun"
input_token_ids = [1, 3, 0, 2]
# The LLM gets a vector of input token_ids, and generates a probability-distribution
# for what the output token_id should be (with a probability score for each token_id
# in the vocabulary)
output = LLM(input_token_ids)
# by taking argmax on the output, we effectively perform a 'greedy sampling' process,
# i.e. we choose the token_id with the highest probability. Other sampling techniques
# also exist, e.g. Top-K. By choosing a probabilistic sampling method we enable the
model
```

```
# to generate different outputs when called multiple times with the same input.
next_token_id = np.argmax(output)
# map the token_id back into an output token
next_token = vocab[next_token_id]
```

Щоб генерувати повні речення, програма ітеративно закликає LLM для генерації передбачення наступної лексеми, і на кожній ітерації ми додаємо передбачений токен назад до вхідних даних:

```
def generate(input_token_ids, n_tokens_to_generate):
  for _ in range(n_tokens_to_generate): # decode loop
    output = LLM(input_token_ids) # model forward pass
    next_token_id = np.argmax(output) # greedy sampling
    if (next_token_id == EOS_TOK_ID)
      break # break if generated End Of Sentence (EOS)
    # append the prediction to the input, and continue to the next out_token
    input_token_ids.append(int(next_token_id))
  return input_token_ids[-n_tokens_to_generate :] # only return generated token_ids
input_token_ids = [1, 3] # "I" "am"
output_token_ids = generate(input_tokens_ids, 4) # output_token_ids = [0, 2, 4, 6]
output_tokens = [vocab[i] for i in output_token_ids] # "having" "fun" "learning" "Neuron"
```

Цей процес передбачення майбутніх значень (регресія) і додавання їх назад до вхідних даних (авто) і називають авторегресією[1].

Можна порівняти, як моделі Llama розбивають текст на токени з моделями OpenAI (рисунки 2.1,2.2).

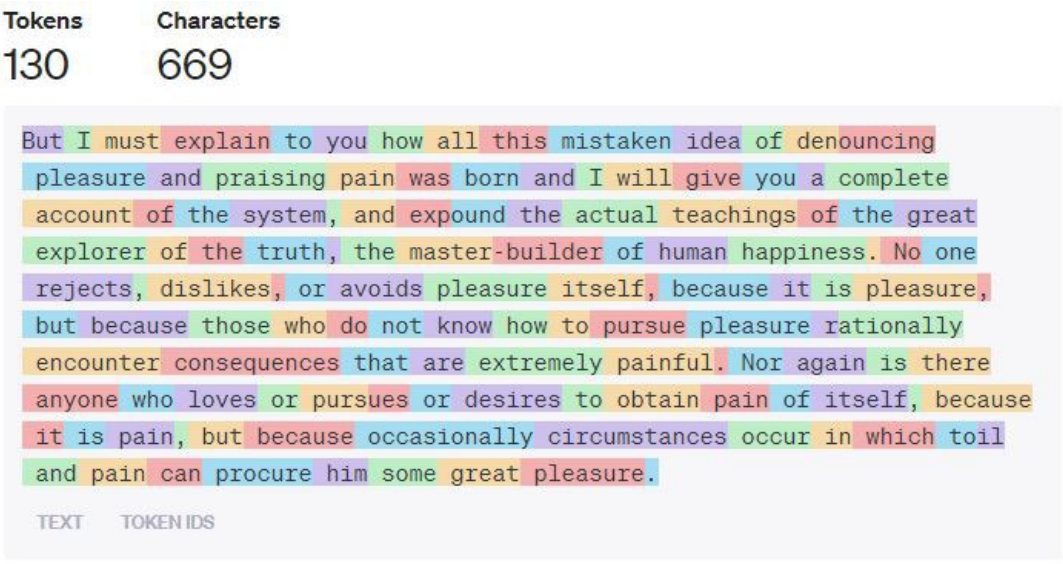

Рисунок 2.1 – Токенізація тексту моделей OpenAI

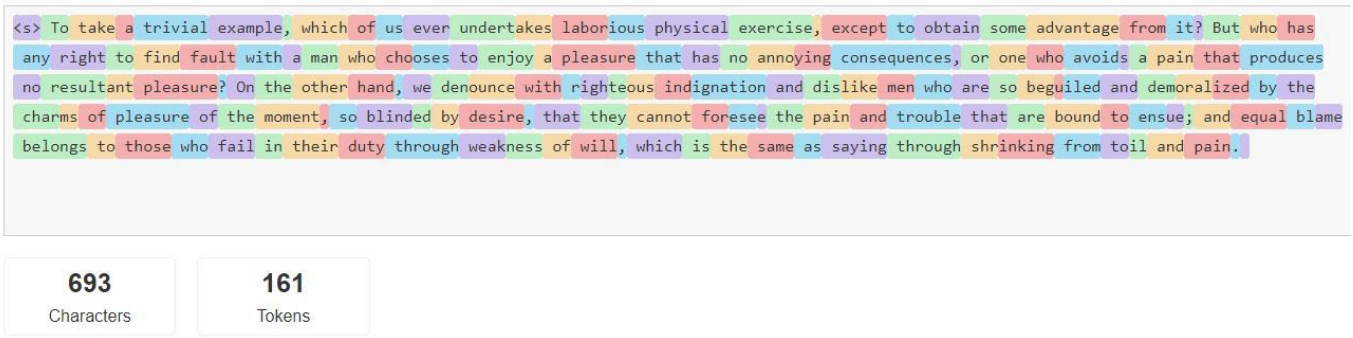

Рисунок 2.2 –Токенізація тексту моделі Llama

Хоча постачальники LLM-інференсу часто говорять про продуктивність у метриці на основі токенів (наприклад, токенів/секунду), ці цифри не завжди можна порівняти між типами моделей, враховуючи ці варіації. Наприклад, команда Anyscale виявила, що токенізація Llama 2 займає на 19% більше часу, ніж токенізація ChatGPT (але все одно має набагато меншу загальну вартість). А дослідники з HuggingFace також виявили, що Llama 2 потребує на ~20% більше токенів для навчання на тому ж обсязі тексту, що і GPT-4.

2.2 Важливі показники для LLM-сервісу

Отже, як саме потрібно думати про швидкість виведення? Розробники з MosaicML пропонують чотири ключові метрики для обслуговування LLM:

1) Час до першого токена (Time To First Token, TTFT): як швидко користувачі починають бачити результати роботи моделі після введення запиту. Низький час очікування відповіді важливий для взаємодії в реальному часі, але менш важливий для офлайн-навантажень. Ця метрика визначається часом, необхідним для обробки запиту і генерації першого токена результату.

2) Час на вихідний токен (Time Per Output Token, TPOT): час генерації вихідного токену для кожного користувача, який звертається до нашої системи. Ця метрика відповідає тому, як кожен користувач буде сприймати «швидкість» моделі. Наприклад, TPOT 100 milisecnd/tok означає 10 токенів на секунду на

користувача, або ~450 слів на хвилину, що швидше, ніж звичайна людина може прочитати.

3) Час затримки: загальний час, який потрібен моделі, щоб згенерувати повну відповідь для користувача. Загальну затримку відповіді можна розрахувати, використовуючи дві попередні метрики: затримка = (TTFT) + (TPOT) * (кількість токенів, які потрібно згенерувати).

4) Пропускна здатність: кількість вихідних токенів в секунду, які може генерувати сервер виведення для всіх користувачів і запитів.

Для оптимізації метою будуть найшвидший час до першого токена, найвища пропускна здатність і найшвидший час на один вихідний токен. Іншими словами, ми хочемо, щоб моделі генерували текст якомога швидше для якомога більшої кількості користувачів.

Слід зазначити, що існує компроміс між пропускною здатністю та часом генерації одного токена: якщо ми обробляємо 16 запитів користувачів одночасно, ми матимемо вищу пропускну здатність порівняно з послідовним виконанням запитів, але нам знадобиться більше часу, щоб згенерувати токени для кожного користувача.

Якщо існує точна ціль щодо загального часу затримки інференсу, є приклади корисних методів для оцінювання моделей:

– Довжина вихідного токену домінує над загальною затримкою відгуку: для визначення середньої затримки зазвичай можна просто взяти очікувану/максимальну довжину вихідного токена і помножити її на загальний середній час на вихідний токен для моделі.

– Довжина вхідних даних не має значення для продуктивності, але важлива для вимог до апаратного забезпечення: Додавання 512 вхідних токенів збільшує затримку менше, ніж виробництво 8 додаткових вихідних токенів у моделях MPT. Однак необхідність підтримувати довгі вхідні дані може ускладнити обслуговування моделей. Наприклад, ми рекомендуємо використовувати A100-80GB (або новіші) для обслуговування MPT-7B з максимальною довжиною контексту 2048 токенів.

– Загальний час затримки сублінійно залежить від розміру моделі: на однаковому обладнанні більші моделі працюють повільніше, але співвідношення швидкості не обов'язково збігається зі співвідношенням кількості параметрів. Затримка MPT-30B у ~2.5 рази більша, ніж у MPT-7B. Затримка Llama2-70B у ~2 рази більша, ніж у Llama2-13B.

3 МЕТОДИ ОПТИМІЗАЦІЇ ІНФЕРЕНСА LLM

3.1 Задачі забезпечення інференса LLM

Оптимізація інференсу LLM має користь від загальних методів, таких як:

– Злиття операторів: об'єднання різних сусідніх операторів разом часто призводить до зменшення затримки.

– Квантизація: активатори та ваги [2] стискаються, щоб використовувати меншу кількість бітів.

– Стиснення: розрідженість або дистиляція.

– Паралелізація: тензорний паралелізм на декількох пристроях або конвеєрний паралелізм для великих моделей.

Окрім цих методів, існує багато важливих оптимізацій, специфічних для трансформерів. Яскравим прикладом цього є кешування KV (key-value). Механізм «Уваги»[3] у моделях трансформерах, що базуються на декодері, є неефективним з точки зору обчислень. Кожен токен звертає увагу на всі раніше бачені токени, і, таким чином, повторно обчислює багато тих самих значень, коли генерується кожен новий токен. Наприклад, під час генерації n-го токена, (n-1)-й токен звертається до (n-2)-го, (n-3)-го, ... 1-го токенів. Аналогічно, під час генерації (n+1)-го токену, увага для n-го токену знову повинна бути звернена на (n-1)-й, (n-2)-й, (n-3)-й, ... 1-й токени. Кешування KV, тобто збереження проміжних ключів/значень для шарів уваги, використовується для того, щоб зберегти ці результати для подальшого використання, уникаючи повторних обчислень.

3.2 Смуга пропускання пам'яті

Ще один фактор – це ширина смуги пропускання пам'яті, або смуга пропускання частот. В процесах обчислення в LLM в основному переважають операції множення матриці на матрицю, ці операції з невеликою розмірністю, як

правило, обмежені шириною смуги пам'яті на більшості апаратних засобів. При генерації токенів авторегресійним способом один з розмірів матриці активації (визначається розміром партії та кількістю токенів у послідовності) є малим при малих розмірах партії. Тому швидкість залежить від того, як швидко ми можемо завантажити параметри моделі з пам'яті GPU в локальні кеші/регістри, а не від того, як швидко ми можемо обчислювати на завантажених даних. Доступна та досягнута смуга пропускання пам'яті в апаратному забезпеченні виводу є кращим предиктором швидкості генерації токенів, ніж їхня пікова обчислювальна продуктивність.

Використання апаратного забезпечення дуже важливе з точки зору вартості обслуговування. Графічні процесори дорогі, і нам потрібно, щоб вони виконували якомога більше роботи. Сервіси спільного виводу обіцяють знизити витрати за рахунок об'єднання робочих навантажень від багатьох користувачів, заповнення окремих прогалин і партійної обробки запитів, що перетинаються. Для великих моделей, таких як Llama2-70B, ми досягаємо гарного співвідношення ціна/продуктивність лише при великих розмірах партій. Наявність системи обслуговування виведення, яка може працювати з великими партіями, має вирішальне значення для економічної ефективності. Однак, велика партія означає більший розмір кешу KV, а це, в свою чергу, збільшує кількість графічних процесорів, необхідних для обслуговування моделі. Тут відбувається перетягування канату, і операторам спільних сервісів доводиться йти на певні компроміси з точки зору витрат і впроваджувати системну оптимізацію.

3.3 Використання ширини смуги пам’яті моделі для оптимізації

Як пояснювалося раніше, виведення для LLM при невеликих розмірах партій, особливо під час декодування, залежить від того, наскільки швидко ми можемо завантажити параметри моделі з пам'яті пристрою в обчислювальні блоки. Смуга пропускання пам'яті диктує, як швидко відбувається рух даних. Щоб виміряти використання апаратного забезпечення, з'являється нова метрика,

яка називається «Model Bandwidth Utilization» (MBU) – використання ширини смуги пам’яті моделі. MBU визначається як (досягнута ширина смуги пам'яті) / (пікова ширина смуги пам'яті), де досягнута ширина смуги пропускання – це ((загальний розмір параметрів моделі + KV розмір кешу) / TPOT).

Припустимо, коли параметр розміром 7B, що працює з 16-бітною точністю, має TPOT рівним 14 мс, то він переміщує 14 ГБ параметрів за 14 мс, що означає використання ширини смуги в 1 ТБ/с. Якщо пікова ширина смуги машини становить 2 ТБ/с, ми працюємо з MBU 50%. Для простоти в цьому прикладі ігнорується розмір кешу KV, який є невеликим для менших розмірів партій і коротших послідовностей. Значення MBU, близькі до 100%, означають, що система виведення ефективно використовує доступну ширину смуги пам'яті. MBU також корисний для порівняння різних систем виводу (апаратне + програмне забезпечення) в нормалізованому вигляді. MBU доповнює метрику «Model Flops Utilization» (MFU)[4], яка є важливою в умовах обмежених обчислень.

На рисунку 3.1 показано графічне представлення MBU у вигляді діаграми, схожої на графік roofline [5].

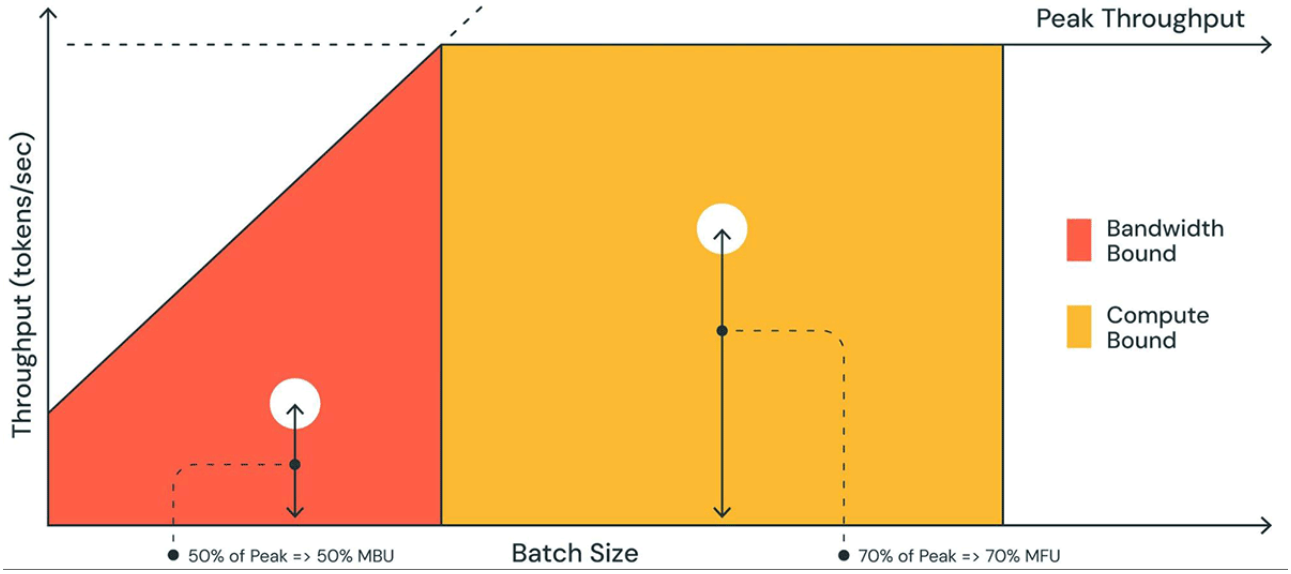

Рисунок 3.1 – Частки піків MBU та MFU, досягнутих в областях, обмежених пам'яттю та обчисленнями

Суцільна похила лінія помаранчевої області показує максимально можливу пропускну здатність, якщо смуга пропускання пам'яті повністю насичена на 100%. Однак, в дійсності для малих розмірів партій (біле коло) спостережувана продуктивність є нижчою за максимальну, що можна виміряти за допомогою MBU. Для великих розмірів партій (жовта область) система обмежена в обчисленнях, і досягнута пропускна здатність як частка від максимальної можливої пропускної здатності вимірюється як Model Flops Utilization (MFU).

MBU та MFU визначають, наскільки більше місця доступно для збільшення швидкості інферена на даному апаратному забезпеченні. На рисунку 3.2 показано виміряні MBU для різних ступенів тензорного паралелізму з сервером виведення на основі TensorRT-LLM. Пікове використання ширини смуги пам'яті досягається при передачі великих суміжних блоків пам'яті. Коли менші моделі, такі як MPT-7B, розподілені між декількома GPU, ми спостерігаємо менший MBU, оскільки ми переміщуємо менші фрагменти пам'яті на кожному GPU.

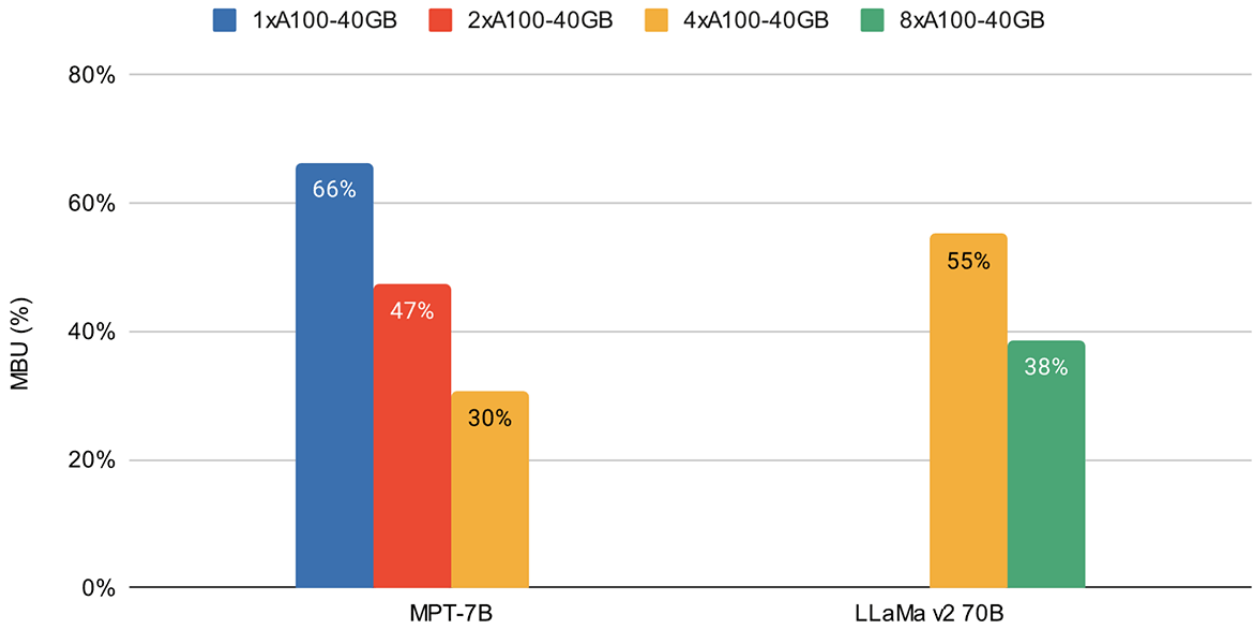

Рисунок 3.2 – Емпіричні спостереження MBU для різних ступенів тензорного паралелізму з TensorRT-LLM на графічних процесорах A100-40G. Запити: послідовності з 512 вхідних токенів з розміром партії 1

На рисунку 3.3 показано емпірично спостережувані MBU для різних ступенів тензорного паралелізму та розмірів партій на графічних процесорах NVIDIA H100. MBU зменшується зі збільшенням розміру партії. Однак при масштабуванні графічних процесорів відносне зменшення MBU є менш значним. Також варто відзначити, що вибір апаратного забезпечення з більшою шириною смуги пам'яті може підвищити продуктивність з меншою кількістю графічних процесорів. При розмірі партії 1 ми можемо досягти вищого показника MBU на 60% на 2xH100-80 ГБ у порівнянні з 55% на 4xA100-40 ГБ GPU (див. рисунок 3.2).

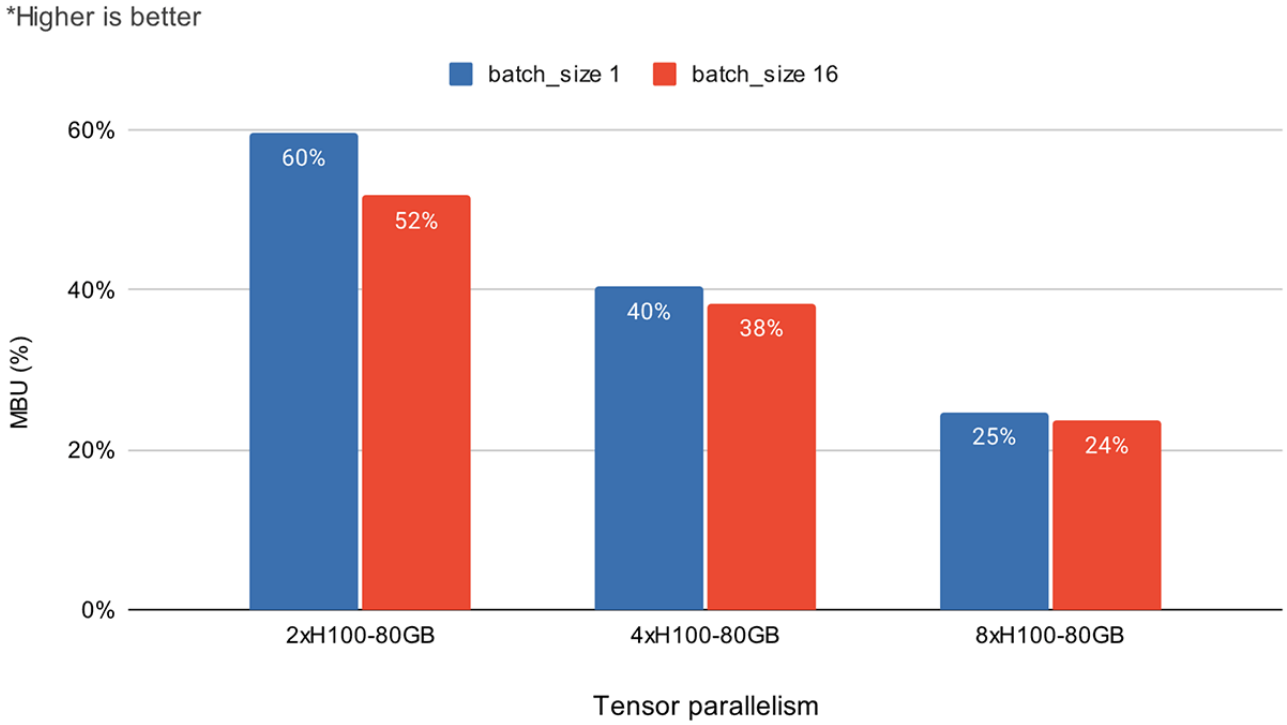

Рисунок 3.3 – Емпіричні спостереження MBU для різних розмірів партій (Llama v2 70B fp16) та режимів тензорного паралелізму на графічних процесорах H100-80ГБ. Запити: послідовності з 512 вхідних токенів

4 ПОРІВНЯЛЬНИЙ АНАЛІЗ ОПТИМІЗАЦІЙ НА МОДЕЛЯХ

4.1 Порівняння часу затримки

Порівняння часу до першого токена (TTFT) (таблиця 4.1) та час на вихідний токен (TPOT) для моделей MPT-7B та Llama2-70B з різним ступенем тензорного паралелізму. Зі збільшенням тривалості вхідних запитів час генерації першого токена починає займати значну частину загальної затримки. Розпаралелювання тензора на декілька графічних процесорів допомагає зменшити цей час затримки.

На відміну від навчання моделі, масштабування на більшу кількість графічних процесорів дає значне зменшення користі затримки виведення. Наприклад, для Llama2-70B перехід від 4x до 8x GPU зменшує затримку лише на 0.7x при невеликих розмірах партії. Однією з причин цього є те, що вищий паралелізм має менший MBU. Інша причина полягає в тому, що тензорний паралелізм збільшує накладні витрати на зв'язок між вузлами GPU.

Таблиця 4.1 – Час до першого токена для запитів на введення довжиною 512 токенів з розміром партії 1

	Час до першого токена (мс)			
Модель	1xA100-40GB	2xA100-40GB	4xA100-40GB	8xA100-40GB
MPT-7B	46	34	26	-
Llama2-70B	-	-	154	114

Більші моделі, такі як Llama2 70B, потребують щонайменше 4xA100-40B графічних процесорів для розміщення в пам'яті.

При великих розмірах партій вищий тензорний паралелізм призводить до більш значного відносного зменшення затримки токенів. На рисунку 4.1 показано, як змінюється час на один вихідний токен для MPT-7B. При розмірі партії 1, перехід від 2x до 4x зменшує затримку токенів лише на ~12%. При розмірі партії 16 затримка при 4x на 33% менша, ніж при 2x. Це узгоджується з

попереднім спостереженням, що відносне зменшення MBU є меншим при вищих ступенях тензорного паралелізму для розміру партії 16 порівняно з розміром партії 1.

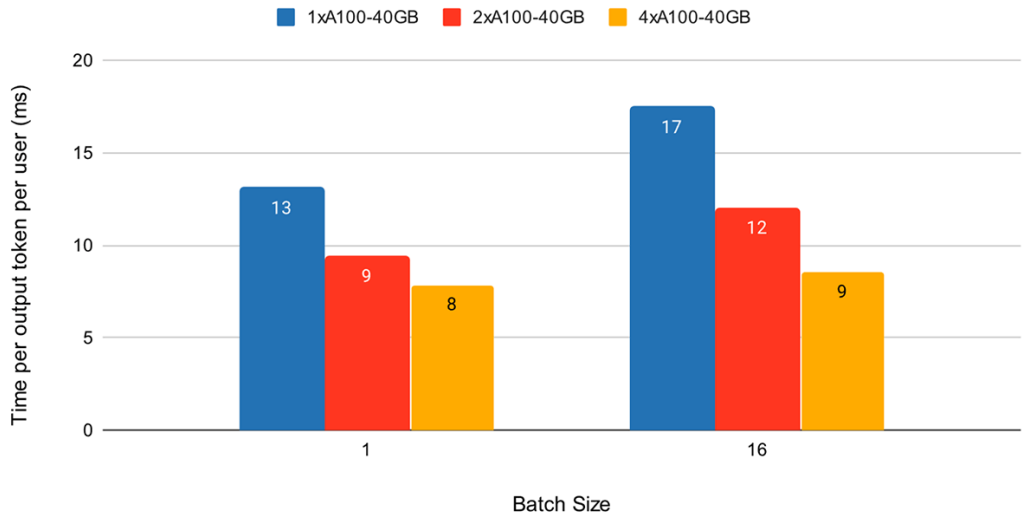

Рисунок 4.1 – Час на вихідний токен на користувача при масштабуванні MPT-7B(bf16) на графічних процесорах A100-40GB

Затримка не масштабується лінійно зі збільшенням кількості графічних процесорів. Запити: послідовності з 128 вхідних і 64 вихідних токенів.

Рисунок 4.2 показує схожі результати для Llama2-70B, за винятком того, що відносне покращення між 4x та 8x є менш вираженим.

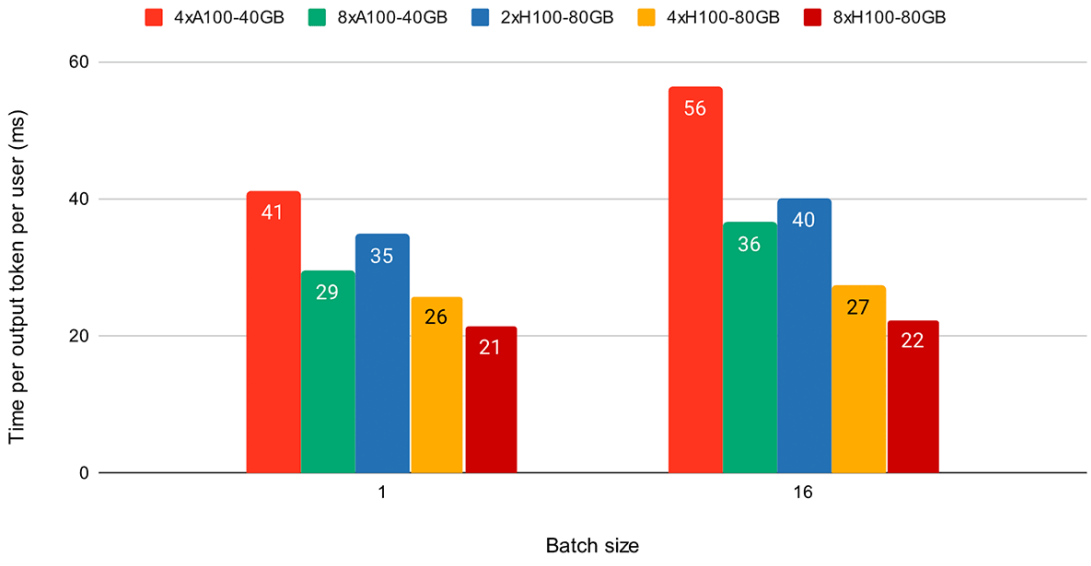

Рисунок 4.2 – Час на вихідний токен на користувача при масштабуванні Llama-v2-70B на декілька графічних процесорів

Значення для графічних процесорів 1x40ГБ, 2x40ГБ та 1x80ГБ відсутні, оскільки Llama-v2-70B (у форматі float16) не підходить для цих систем.

Також порівнюється масштабування графічного процесора на двох різних апаратних засобах. Оскільки H100-80ГБ має в 2,15 рази більшу ширину смуги графічної пам'яті порівняно з A100-40ГБ, ми бачимо, що затримка на 36% менша при розмірі партії 1 і на 52% менша при розмірі партії 16 для 4x систем.

4.2 Пропускна здатність

Ми можемо досягти компромісу між пропускною здатністю та часом обробки одного токена, об'єднавши запити в партію(групу). Партіювання запитів під час обчислень на GPU збільшує пропускну здатність у порівнянні з послідовною обробкою запитів, але кожен запит буде виконуватися довше (без урахування ефекту черги).

Існує кілька поширених методів для партіювання запитів на виведення, як показано на рисунку 4.3.

Статична партійна обробка: клієнт упаковує кілька запитів у запити, і відповідь повертається після того, як всі послідовності в партії будуть виконані.

Динамічна партійна обробка: запити об'єднуються в партії «на льоту»[4] всередині сервера. Зазвичай цей метод працює гірше, ніж статична обробка, але може наблизитися до оптимального, якщо відповіді короткі або однакової довжини. Погано працює, коли запити мають різні параметри.

Безперервне партіювання[5]: Ідея розбиття запитів на партії по мірі їх надходження являється методом SOTA. Замість того, щоб чекати, поки всі послідовності в партії закінчать роботу, він групує послідовності разом на рівні ітерацій. Це дозволяє досягти в 10-20 разів кращої пропускної здатності, ніж динамічне партіювання.

Безперервне партіювання зазвичай є найкращим підходом для спільних служб, але бувають ситуації, коли два інших підходи можуть бути кращими. У

середовищах з низьким QPS(Queries per second) динамічна партійна обробка може перевершити безперервну.

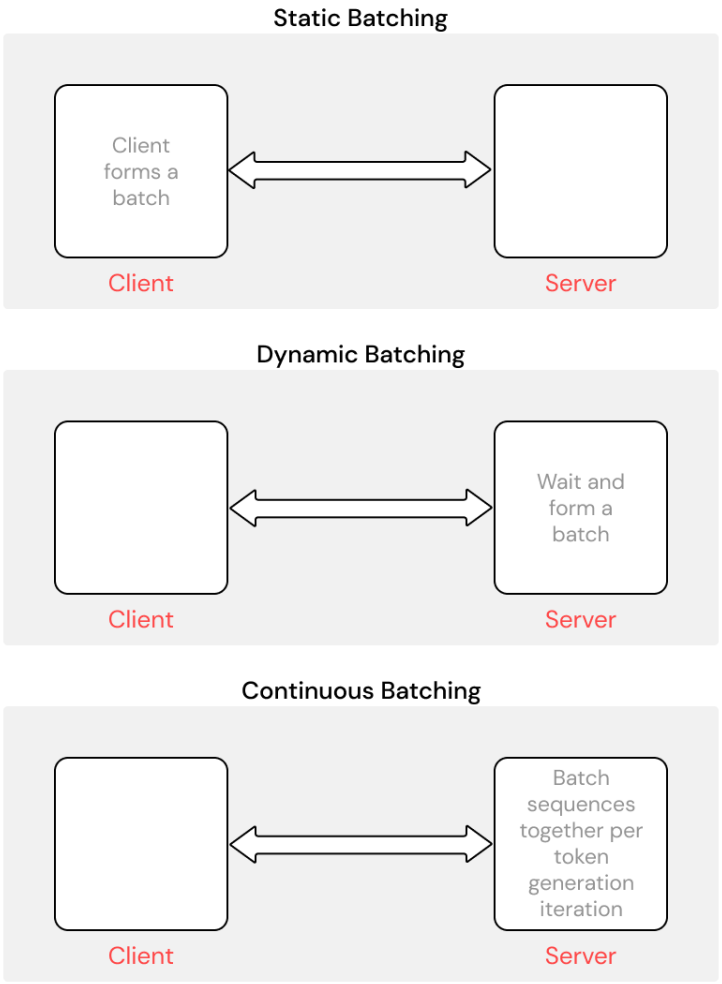

Рисунок 4.3 – Різні типи партійного інференсу подачі LLM. Партійне виведення є ефективним способом підвищення ефективності інференсу

Іноді легше реалізувати низькорівневу оптимізацію графічного процесора в простішому фреймворку партійної обробки. Для робочих навантажень партійного інференсу в автономному режимі статичне партіювання дозволяє уникнути значних накладних витрат і досягти кращої пропускної здатності.

4.3 Розміри партій

Ефективність роботи партійної обробки сильно залежить від потоку запитів. Можна отримати верхню межу його продуктивності, провівши

бенчмаркінг статичного партіювання з однорідними запитами. Результати бенчмаркінгу можна побачити на таблиці 4.2.

Таблиця 4.2 – Пікова пропускна здатність MPT-7B (запитів/сек) зі статичним партіюванням і бекендом на базі FasterTransformers.

	Розмір партії						
Апаратне забезпечення	1	4	8	16	32	64	128
1 x A10	0.4	1.4	2.3	3.5	OOM error	OOM error	OOM error
2 x A10	0.8	2.5	4.0	7.0	8.0	OOM error	OOM error
1 x A100	0.9	3.2	5.3	8.0	10.5	12.5	OOM error
2 x A100	1.3	3.0	5.5	9.5	14.5	17.0	22.0
4 x A100	1.7	6.2	11.5	18.0	25.0	33.0	36.5

Запити: 512 вхідних і 64 вихідних токенів. Для більших вхідних даних межа OOM(Out-of-Memory) буде при менших розмірах партій.

4.4 Оптимізація часу затримки

Час затримки запиту збільшується зі збільшенням розміру партії. Наприклад, з одним графічним процесором NVIDIA A100, якщо ми максимізуємо пропускну здатність при розмірі партії 64, затримка збільшується в 4 рази, тоді як пропускна здатність зростає в 14 разів. Спільні сервіси інференсу зазвичай вибирають збалансований розмір партії. Користувачі, які розміщують власні моделі, повинні самі вирішувати, який компроміс між затримкою та пропускною здатністю підходить для їхніх додатків. У деяких додатках, таких як чат-боти, низька затримка для швидких відповідей є головним пріоритетом. В інших додатках, таких як партійна обробка неструктурованих PDF-файлів, ми можемо пожертвувати затримкою при обробці окремого документа заради швидкої паралельної обробки всіх документів.

На рисунку 4.4 показано криву залежності пропускної здатності від затримок для моделі 7B. Що дозволяє користувачам вибрати апаратну конфігурацію, яка відповідає їхнім вимогам до пропускної здатності в умовах обмеження затримок.

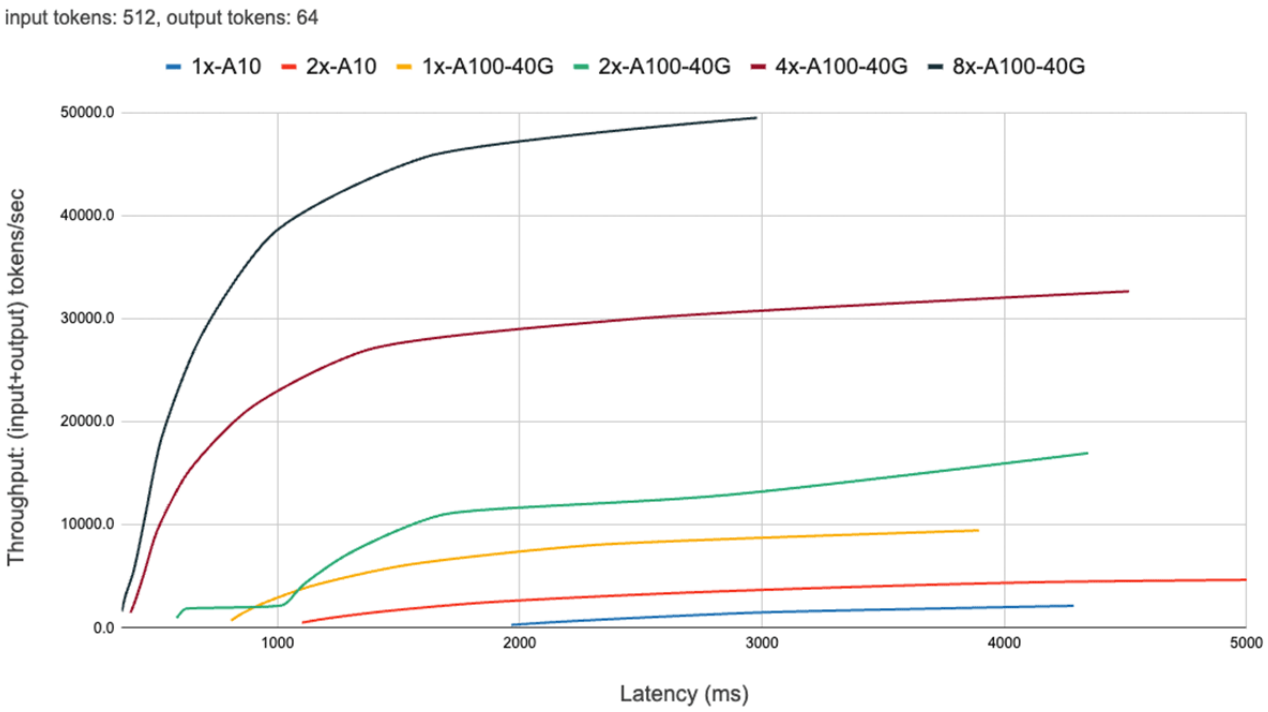

Рисунок 4.4 – Залежність пропускної здатності від затримки для моделі MPT-7B

Кожна лінія на цій кривій отримана при збільшенні розміру партії від 1 до 256. Це корисно для визначення того, наскільки великим ми можемо зробити розмір партії, враховуючи різні обмеження на затримку. Після певного розміру партії, тобто коли ми переходимо до режиму обмеження обчислень, кожне подвоєння розміру партії лише збільшує затримку, не збільшуючи пропускну здатність.

При використанні паралелізму важливо розуміти низькорівневі апаратні деталі. Наприклад, не всі екземпляри 8xA100 однакові в різних хмарах. Деякі сервери мають високошвидкісні з'єднання між усіма графічними процесорами, інші об'єднують графічні процесори в пари, а з'єднання між парами мають меншу ширину смуги. Це може призвести до виникнення «вузьких місць», через що реальна продуктивність значно відхилятиметься від наведених вище кривих.

4.5 Оптимізація за допомогою квантизації

Квантизація є поширеною технікою, яка використовується для зменшення вимог до апаратного забезпечення для LLM-виведення. Зменшення точності ваг моделі та активацій під час виведення може значно знизити вимоги до апаратного забезпечення. Наприклад, перехід від 16-бітових ваг до 8-бітових може вдвічі зменшити кількість необхідних графічних процесорів у середовищах з обмеженою пам'яттю (наприклад, Llama2-70B на A100s). Зменшення ваг до 4-бітових дозволяє виконувати виведення на споживчому обладнанні (наприклад, Llama2-70B на Macbook).

На практиці, квантизацію слід застосовувати з обережністю. Наївні методи квантування можуть призвести до значного погіршення якості моделі. Вплив квантування також залежить від архітектури моделі та розміру.

Експериментуючи з такими методами, як квантизація, рекомендовано використовувати LLM-тести якості, наприклад, Mosaic Eval Gauntlet[6], щоб оцінити якість системи виведення, а не тільки якість моделі ізольовано. Крім того, важливо дослідити глибшу оптимізацію системи. Зокрема, квантизація може зробити кеш KV набагато ефективнішим.

Як згадувалося раніше, при авторегресійній генерації токенів, минулі ключі/значення (KV) з шарів уваги кешуються замість того, щоб перераховувати їх на кожному кроці. Розмір кешу KV залежить від кількості послідовностей, що обробляються за один раз, і довжини цих послідовностей. Крім того, під час кожної ітерації наступного покоління токенів до існуючого кешу додаються нові елементи KV, що робить його більшим, оскільки генеруються нові токени. Тому ефективне керування кеш-пам'яттю KV при додаванні цих нових значень є критично важливим для хорошої продуктивності виведення. У моделях Llama2 використовується варіант уваги, який називається «групова увага до запитів», GQA (Grouped Query Attention). Коли кількість заголовків KV дорівнює 1 – GQA є тим самим, що й мультизапитова увага MQA (Multi-Query-Attention). GQA

допомагає зменшити розмір кешу KV за рахунок спільного використання ключів/значень. Формула для обчислення розміру кешу KV наступна:

*batch_size * seqlen * (d_model/n_heads) * n_layers * 2 (K and V) * 2 (bytes per Float16) * n_kv_heads*

Таблиця 3 показує розмір кешу GQA KV, розрахований для різних розмірів партій при довжині послідовності 1024 токени. Для порівняння, розмір параметра для моделей Llama2 становить 140 ГБ (Float16) для моделі 70B. Квантизація кешу KV є ще одним методом (на додаток GQA/MQA) для зменшення розміру кешу KV, вплив чого оцінюється на якість генерації.

Таблиця 4.3 – Розмір кешу KV для Llama-2-70B при довжині послідовності 1024

Розмір партії	Кеш-пам'ять GQA KV (FP16)	Кеш-пам'ять GQA KV (Int8)
1	0.312 ГіБ	0.156 ГіБ
16	5 ГіБ	2.5 ГіБ
32	10 ГіБ	5 ГіБ
64	20 ГіБ	10 ГіБ

Як згадувалося раніше, генерація токенів за допомогою LLM при малих розмірах партій є проблемою, пов'язаною з пропускною здатністю пам'яті GPU, тобто швидкість генерації залежить від того, наскільки швидко параметри моделі можуть бути переміщені з пам'яті GPU до кешу на чипі. Перетворення ваги моделі з FP16 (2 байти) в INT8 (1 байт) або INT4 (0.5 байт) вимагає переміщення меншого обсягу даних і, таким чином, прискорює генерацію токенів. Однак квантизація може негативно вплинути на якість генерації моделі.

5 УЗАГАЛЬНЕННЯ З ОПТИМІЗАЦІЙ ЗАБЕЗПЕЧЕННЯ ІНФЕРЕНСУ

Кожен з факторів, впливає на те, як будуються та розгортаються моделі. Треба використовувати ці результати для прийняття рішень на основі даних, які враховують тип обладнання, стек програмування, архітектуру моделі та типові патерни використання.

5.1 Яку техніку паралелізму слід використовувати

Тензорний паралелізм покращує час затримки за рахунок збільшення внутрішньошарової комунікації. Таким чином, як правило, краще використовувати найменший ступінь тензорного паралелізму, який відповідає вимогам до затримок, а потім використовувати паралелізм конвеєра/даних.

Якщо затримка не є основною проблемою у відповідному застосунку (наприклад, оцінка моделі), а основною метою є максимізація пропускної здатності (тобто, мінімізація загальної вартості на токен), то найефективніше використовувати конвеєрний паралелізм і збільшувати розмір пакету наскільки це можливо.

У затримки такі складові, як час до першого токена для інтерактивних застосунків визначає, наскільки швидко буде реагувати сервіс, а час на вихідний токен – наскільки швидко він буде працювати.

Ширина смуги пам'яті є ключовим фактором: Генерування першого токена, як правило, пов'язане з обчисленнями, тоді як подальше декодування – це операція, пов'язана з пам'яттю. Оскільки LLM-виведення часто працює в умовах обмеженої пам'яті, MBU є корисною метрикою для оптимізації і може використовуватися для порівняння ефективності систем виведення.

5.2 Який розмір партії слід використовувати

Послідовна генерація токенів генеративного виведення LLM, це робоче навантаження, як правило, дуже обмежене в пам'яті. Це означає, що пропускна

здатність (а отже, і вартість виведення) значно покращиться за рахунок партійної обробки.

Одночасна обробка декількох запитів має вирішальне значення для досягнення високої пропускної здатності та ефективного використання дорогих графічних процесорів. Для використання спільних онлайн-сервісів безперервна партійна обробка є необхідною, в той час як робочі навантаження з партійного виведення в офлайні можуть досягти високої пропускної здатності за допомогою простіших методів партійної обробки.

Як правило, краще збільшувати розмір партій до максимальної величини, яка вписується в бюджет затримок (до batch=256, більший розмір пакетів зазвичай не сприяє підвищенню продуктивності).

Важливо, що KV-кеш зростає лінійно зі збільшенням розміру партії, і може зростати до моменту, коли пам'ять закінчується (зазвичай позначається як OOM). Якщо дозволяє бюджет затримок, рекомендується збільшувати розмір партій до максимального значення, яке не призводить до OOM.

Також можна розглянути можливість конвеєризації моделі понад необхідного для того, щоб вмістити параметри моделі / KV-кеш на пристроях, щоб звільнити місце в пам'яті пристрою і, таким чином, дозволити збільшити розмір партій, не спричиняючи проблем з переповненням пам'яті.

Стандартні методи оптимізації виведення є важливими (наприклад, злиття операторів, вагове квантування) для LLM, але важливо дослідити глибші методи оптимізації систем, особливо ті, що покращують використання пам'яті. Одним із прикладів є квантизація кешу KV.

5.3 Конфігурації обладнання

Тип моделі та очікуване робоче навантаження слід використовувати для вибору апаратного забезпечення для розгортання. Наприклад, при масштабуванні на декілька графічних процесорів MBU падає набагато швидше для менших моделей, таких як MPT-7B, ніж для більших моделей, таких як

Llama2-70B. Продуктивність також має тенденцію до сублінійного масштабування з вищим ступенем тензорного паралелізму. Тим не менш, високий ступінь тензорного паралелізму може мати сенс для менших моделей, якщо трафік високий або якщо користувачі готові платити більше за наднизьку затримку.

5.4 Прийняття рішень на основі даних

Розуміння теорії є важливим, ліпше завжди вимірювати наскрізну продуктивність сервера. Існує багато причин, чому розгортання інференса може працювати гірше, ніж очікувалося. MBU може бути несподівано низьким через неефективність програмного забезпечення, або відмінності в апаратному забезпеченні між хмарними провайдерами можуть призвести до сюрпризів – може бути двократна різниця в затримці між серверами різних провайдерів.

6 РЕАЛІЗАЦІЯ ЗАБЕЗПЕЧЕННЯ ІНФЕРЕНСА З NVIDIA TRITON

Повторюючись, LLM – великі. Цей факт може зробити їх дорогими і повільними у виконанні без правильних методів.

Для вирішення цієї проблеми існує багато методів оптимізації, від оптимізації моделей, таких як злиття ядер[7] і квантизація, до оптимізації часу виконання, наприклад, реалізацій на C++, кешування KV, безперервного партіювання «на льоту»[8] та «уваги сторінок»[9]. Може бути важко вирішити, які з них підходять для використання, і орієнтуватися у взаємодії між цими методами та їхніми, іноді, несумісними реалізаціями.

TensorRT-LLM від NVIDIA, комплексна бібліотека для компіляції та оптимізації LLM для виведення, включає в себе всі ці оптимізації, надаючи інтуїтивно зрозумілий Python API для визначення та побудови нових моделей.

Бібліотека TensorRT-LLM з відкритим вихідним кодом прискорює продуктивність виведення на новітніх LLM на графічних процесорах NVIDIA. Вона використовується як основа оптимізації для LLM виводу в NVIDIA NeMo[14], комплексному фреймворку для створення, налаштування та розгортання додатків генеративного ШІ у виробництво. NeMo надає повні контейнери, включаючи TensorRT-LLM і NVIDIA Triton, для розгортання генеративного ШІ.

TensorRT-LLM налічує компілятор глибокого навчання TensorRT і включає найновіші оптимізовані ядра, створені для найсучасніших реалізацій FlashAttention[15] та маскованої багатоголової уваги MHA (masked multi-head attention) для виконання LLM.

Також TensorRT-LLM має етапи попередньо та пост-обробки та примітиви багатопроцесорної/багатовузлової комунікації в простому API Python з відкритим вихідним кодом для покращення продуктивності виведення LLM на графічних процесорах.

6.1 Початок встановлення

Першим кроком буде клонування та збірка бібліотеки TensorRT-LLM. Найпростіший спосіб зібрати TensorRT-LLM і отримати всі її залежності – це скористатися Docker-файлом, що входить до комплекту поставки:

```
git lfs install
git clone -b release/0.5.0 https://github.com/NVIDIA/TensorRT-LLM.git
cd TensorRT-LLM
git submodule update --init --recursive
make -C docker release_build
```

Ці команди витягують базовий контейнер і встановлюють усі залежності, необхідні для TensorRT-LLM, всередині контейнера. Після цього збирається і встановлюється сам TensorRT-LLM у контейнер.

6.2 Отримання ваг моделі

TensorRT-LLM – це бібліотека для LLM-висновку, тому для її використання потрібно надати набір навчених ваг. Можна використовувати власні ваги моделі, навчені у фреймворку на кшталт NVIDIA NeMo, або витягнути набір попередньо навчених ваг з репозиторіїв на кшталт HuggingFace Hub[16].

Команди автоматично витягують файли ваг і токенізатор для налаштованого на чат варіанту моделі Llama 2 з параметром 7B з HuggingFace Hub. Також можна завантажити ваги самостійно, щоб використовувати їх в автономному режимі за допомогою наступної команди. Потрібно оновити шляхи в наступних командах, щоб вони вказували на цю директорію:

```
git lfs install
git clone https://huggingface.co/meta-llama/Llama-2-7b-chat-hf
```

Використання цієї моделі відноситься до окремої ліцензії платформи «Meta». Щоб завантажити необхідні файли, треба погодитися з умовами та авторизуватись за допомогою Hugging Face.

6.3 Компіляція моделі

Наступним кроком у процесі є компіляція моделі в рушій TensorRT. Для цього знадобляться ваги моделі, а також визначення моделі, написане на Python API TensorRT-LLM.

Репозиторій TensorRT-LLM містить широкий спектр попередньо визначених архітектур моделей. У цьому матеріалі використовується включене визначення моделі Llama замість того, щоб писати власне.

Це приклади деяких оптимізацій, доступних у TensorRT-LLM (Додаток А):

```
# Launch the Tensorrt-LLM container
make -C docker release_run LOCAL_USER=1
# Log in to huggingface-cli
# You can get your token from huggingface.co/settings/token
huggingface-cli login --token *****
# Compile model
python3 examples/llama/build.py \
    --model_dir meta-llama/Llama-2-7b-chat-hf \
    --dtype float16 \
    --use_gpt_attention_plugin float16 \
    --use_gemm_plugin float16 \
    --remove_input_padding \
    --use_inflight_batching \
    --paged_kv_cache \
    --output_dir examples/llama/out
```

Коли створюється визначення моделі за допомогою API TensorRT-LLM, будується граф операцій з примітивів NVIDIA TensorRT, які формують шари

нейронної мережі. Ці операції відповідають певним ядрам – заздалегідь написаним програмам для графічного процесора.

Компілятор TensorRT може переглядати граф, щоб вибрати найкраще ядро для кожної операції та доступного графічного процесора. Важливо, що він також може ідентифікувати шаблони в графі, де кілька операцій є хорошими кандидатами для об'єднання в одне ядро. Це зменшує необхідний об'єм пам'яті та накладні витрати на запуск декількох ядер GPU.

TensorRT також компілює граф операцій в єдиний граф CUDA (Compute Unified Device Architecture)[17], який може бути запущений одночасно, що ще більше зменшує накладні витрати на запуск ядра.

Компілятор TensorRT є надзвичайно потужним для злиття шарів і збільшення швидкості виконання, але існують деякі складні злиття шарів, такі як FlashAttention, які передбачають чергування багатьох операцій разом, і які не можуть бути виявлені автоматично. У таких випадках можна явно замінити частини графа плагінами під час компіляції.

У цьому прикладі використовується плагін gpt_attention, який реалізує ядро злитої уваги, подібне до FlashAttention, і плагін gemm, який виконує матричне множення з накопиченням FP32. Також треба вказати бажану точність для повної моделі як FP16, що відповідає точності ваг за замовчуванням, які були завантажені з HuggingFace.

Ось що створить цей скрипт після завершення його виконання. У /examples/llama/out тепер повинні бути такі файли:

– Llama_float16_tp1_rank0.engine: Основний вивід скрипту збірки, що містить виконуваний графік операцій з вбудованими вагами моделі.

– config.json: Містить детальну інформацію про модель, таку як її загальна структура та точність, а також інформацію про те, які плагіни було включено до рушія.

– model.cache: Кешує деяку інформацію про час та оптимізацію, отриману під час компіляції моделі, що пришвидшує наступні збірки.

6.4 Запуск моделі

Файл рушія містить інформацію, необхідну для виконання моделі, але використання LLM на практиці вимагає набагато більше, ніж один прямий прохід через модель. TensorRT-LLM включає високо оптимізоване середовище виконання на C++ для виконання побудованих LLM-рушіїв та керування такими процесами, як вибірка токенів з виводу моделі, керування кешем KV та партійне виконання запитів.

Можна використати це середовище виконання безпосередньо для локального виконання моделі, або використовується бекенд середовища виконання TensorRT-LLM для NVIDIA Triton Inference Server, щоб обслуговувати модель для декількох користувачів.

Щоб запустити модель локально, використовується наступна команда:

python3 examples/llama/run.py --engine_dir=examples/llama/out --max_output_len 100 --tokenizer_dir meta-llama/Llama-2-7b-chat-hf --input_text " How do I count to nine in French?"

6.5 Розгортання за допомогою Triton Inference Server

Окрім локального виконання, також можна використовувати NVIDIA Triton Inference Server для створення готового до виробництва розгортання LLM.

У новому бекенді NVIDIA Triton Inference Server для TensorRT-LLM, який використовує середовище виконання TensorRT-LLM C++ для швидкого виконання виводу і включає в себе такі технології, як партійна обробка в польоті та кешування KV на сторінках. Triton Inference Server з бекендом TensorRT-LLM доступний у вигляді готового контейнера через каталог NGC.

Спочатку створіть репозиторій моделі, щоб Triton Inference Server міг прочитати модель і будь-які пов'язані з нею метадані. Сховище tensorrtllm_backend містить каркас відповідного сховища моделей у каталозі all_models/inflight_batcher_llm/, який ви можете використовувати. У цьому

каталозі є чотири підкаталоги, які містять артефакти для різних частин процесу виконання моделі:

– /preprocessing та /postprocessing: Містять скрипти для бекенду Triton Inference Server Python для токенізації текстових вхідних даних та детокенізації вихідних даних моделі для перетворення між рядками та ідентифікаторами токенів, з якими працює модель.

– /tensorrt_llm: Місце, де ви розміщуєте рушій моделі, який ви раніше скомпілювали.

– /ensemble: Визначає комплексне середовище моделі, який пов'язує попередні три компоненти разом і вказує Triton Inference Server, як передавати дані через них.

Відкривши сховище прикладів моделей, треба скопіювати туди модель, яка була скомпільована на попередньому кроці:

```
# After exiting the TensorRT-LLM docker container
cd ..
git clone -b release/0.5.0 \
https://github.com/triton-inference-server/tensorrtllm_backend.git
cd tensorrtllm_backend
cp ../TensorRT-LLM/examples/llama/out/* all_models/inflight_batcher_llm/tensorrt_llm/1/
```

Потім треба змінити деякі конфігураційні файли зі скелету сховища, додавши до них наступну інформацію:

– Де знаходиться скомпільований движок моделі;

– Який токенізатор використовувати;

– Як працювати з виділенням пам'яті для кешу KV при виконанні виводу в партіях.

```
python3 tools/fill_template.py --in_place \
    all_models/inflight_batcher_llm/tensorrt_llm/config.pbtxt \
    decoupled_mode:true,engine_dir:/all_models/inflight_batcher_llm/tensorrt_llm/1,\
```

```
max_tokens_in_paged_kv_cache:,batch_scheduler_policy:guaranteed_completion,kv
_cache_free_gpu_mem_fraction:0.2,\
max_num_sequences:4

python tools/fill_template.py --in_place \
    all_models/inflight_batcher_llm/preprocessing/config.pbtxt \
    tokenizer_type:llama,tokenizer_dir:meta-llama/Llama-2-7b-chat-hf

python tools/fill_template.py --in_place \
    all_models/inflight_batcher_llm/postprocessing/config.pbtxt \
    tokenizer_type:llama,tokenizer_dir:meta-llama/Llama-2-7b-chat-hf
```

Тепер відкривши Docker-контейнер запустіть сервер Triton. Треба вказати «world size», тобто кількість графічних процесорів, для яких було побудовано модель, та вказати на model_repo, що було створено.

```
docker run -it --rm --gpus all --network host --shm-size=1g \
-v $(pwd)/all_models:/all_models \
-v $(pwd)/scripts:/opt/scripts \
nvcr.io/nvidia/tritonserver:23.10-trtllm-python-py3
# Log in to huggingface-cli to get tokenizer
huggingface-cli login --token *****
# Install python dependencies
pip install sentencepiece protobuf
# Launch Server
python /opt/scripts/launch_triton_server.py --model_repo
/all_models/inflight_batcher_llm --world_size 1
```

6.6 Надсилання запитів

Для надсилання запитів та взаємодії з запущеним сервером можливо скористатися однією з клієнтських бібліотек Triton Inference Server або

надсилати HTTP-запити до кінцевої точки generate. Розширення generate надає просту текстоорієнтовану схему кінцевих точок для взаємодії з великими мовними моделями. Щоб почати можна скористатися наступною командою curl:

```
curl -X POST localhost:8000/v2/models/ensemble/generate -d \
'{"text_input": "How do I count to nine in French?",
"parameters": {
"max_tokens": 100,
"bad_words":[""],
"stop_words":[""]}}'
```

ВИСНОВКИ

Разом TensorRT-LLM і Triton Inference Server надають незамінний набір інструментів для оптимізації, розгортання та ефективного запуску LLM. З випуском TensorRT-LLM як бібліотеки з відкритим вихідним кодом на GitHub, організаціям і розробникам додатків стало простіше, ніж будь-коли, використовувати потенціал цих моделей.

Основні особливості TensorRT-LLM включають наступне:

– Підтримка LLM, таких як Llama 1 і 2, ChatGLM, Falcon, MPT, Baichuan і Starcoder;

– Партіювання на льоту і увага сторінок;

– Багатовузловий інференс на декількох графічних процесорах (MGMN);

– Рушій NVIDIA Hopper tansformer з FP8;

– Підтримка архітектури NVIDIA Ampere, архітектури NVIDIA Ada Lovelace та графічних процесорів NVIDIA Hopper;

– Підтримка Windows (бета-версія).

Протягом останніх 2 років NVIDIA тісно співпрацює з провідними компаніями, що займаються LLM, включаючи Anyscale, Baichuan, Cohere, Deci, Grammarly, Meta, Mistral AI, MosaicML, яка зараз є частиною Databricks, OctoML, Perplexity AI, Tabnine, Together.ai, Zhipu та багатьма іншими, щоб прискорити та оптимізувати LLM виведення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Llama 2 is about as factually accurate as GPT-4 for summaries and is 30X cheaper. URL: https://www.anyscale.com/blog/llama-2-is-about-as-factually-accurate-as-gpt-4-for-summaries-and-is-30x-cheaper. (дата звернення 20.12.2023)

2. Post: Llama2 tokenizer gives you 20% more tokens. URL: https://twitter.com/Thom_Wolf/status/1701206627859206450?s=20. (дата звернення 20.12.2023)

3. GPT in 60 Lines of NumPy. URL: https://jaykmody.com/blog/gpt-from-scratch/. (дата звернення 20.12.2023)

4. Introduction to neural networks — weights, biases and activation. URL: https://medium.com/mlearning-ai/introduction-to-neural-networks-weights-biases-and-activation-270ebf2545aa. (дата звернення 20.12.2023)

5. Effective Approaches to Attention-based Neural Machine Translation. URL: https://arxiv.org/abs/1508.04025. (дата звернення 20.12.2023)

6. PaLM: Scaling Language Modeling with Pathways. URL: https://arxiv.org/abs/2204.02311. (дата звернення 20.12.2023)

7. Roofline model. URL: https://en.wikipedia.org/wiki/Roofline_model. (дата звернення 20.12.2023)

8. NVIDIA TensorRT-LLM Supercharges Large Language Model Inference on NVIDIA H100 GPUs. URL: https://developer.nvidia.com/blog/nvidia-tensorrt-llm-supercharges-large-language-model-inference-on-nvidia-h100-gpus/. (дата звернення 26.12.2023)

9. Orca: A Distributed Serving System for Transformer-Based Generative Models. URL: https://www.usenix.org/conference/osdi22/presentation/yu. (дата звернення 20.12.2023)

10. LLM Evaluation Scores. URL: https://www.mosaicml.com/llm-evaluation. (дата звернення 20.12.2023)

11. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. URL: https://arxiv.org/abs/2307.08691. (дата звернення 20.12.2023)

12. A Distributed Serving System for Transformer-Based Generative Models. URL: https://www.usenix.org/conference/osdi22/presentation/yu. (дата звернення 20.12.2023)

13. Efficient Memory Management for Large Language Model Serving with PagedAttention. URL: https://arxiv.org/pdf/2309.06180.pdf. (дата звернення 20.12.2023)

14. NVIDIA NeMo. URL: https://www.nvidia.com/en-us/ai-data-science/generative-ai/nemo-framework/. (дата звернення 26.12.2023)

15. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. URL: https://arxiv.org/abs/2307.08691. (дата звернення 20.12.2023)

16. Llama 2 on Hugging Face. URL: https://huggingface.co/meta-llama/Llama-2-7b-chat-hf. (дата звернення 26.12.2023)

17. Getting Started with CUDA Graphs. URL: https://developer.nvidia.com/blog/cuda-graphs/. (дата звернення 20.12.2023)

18. Кузьомін О. Дослідження методів нейронних мовних моделей для перекладу української мови з використанням штучного інтелекту. // Попов І., Кузьомін О. UINVERSUM. 2023. № 2. С. 95–99. URL: https://doi.org/10.36074/universum.2.2023

19. O. Kuzomin. Decision support procedures for decision making in a COVID condition / Boboyorov Sardor Uchqun o‘g‘li, O. Kuzomin, V. Lyashenko // Multidisciplinary Journal of Science and Technology, 3(2), 2023. P. 74–80.

20. Kuzomin O. CREATION OF INTELLIGENT SYSTEMS FOR ANALYZING SUPERMARKET VISITORS TO IDENTIFY CRIMINAL ELEMENTS /Berkovskyi , D., Kuzomin O. Collection of Scientific Papers «SCIENTIA», (May 5, 2023; Sydney, Australia), 113–118.

21. Кузьомін О. Я. Спільний автокодер із порогом виявлення аномалії суглоба на виробничих лініях / А. Є. Шустрова, науковий керівник: Шустрова А. Є. Кузьомін О. // Я. Розвиток наукової думки постіндустріального суспільства: сучасний дискурс : матеріали III Міжнародної наукової конференції, 28 квітня

2023 р., Львів. — Вінниця : «Європейська наукова платформа», 2023. – С. 120-125.

22. Верколаб Г. С. Дослідження методів забезпечення можливостей інференса для LLM за допомогою NVIDIA Triton/ Г.С. Верколаб // Студентський науковий журнал UNIVERSUM №3, грудень 20, 2023. – Вінниця, Україна. – pp. 148-156.

23. Методичні вказівки до оформлення пояснювальних записок кваліфікаційних робіт [Електронне видання] / Упорядн. О.М. Бітченко, Л.Ф. Сайківська, О.А. Зарудний. – Харків: ХНУРЕ, 2022. – 34 с.