

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Направление: 02.03.02 «Фундаментальная информатика и информационные технологии»

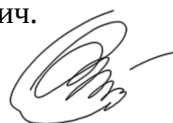
ООП: Большие данные и распределенная цифровая платформа

ОТЧЕТ О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

Тема: Распределенный инференс глубоких нейросетевых моделей.

Выполнил: Колосков Виктор Юрьевич, 22.Б15-пу

Руководитель научно-исследовательской работы: доцент кафедры фундаментальной информатики и распределенных система, кандидат физико-математических наук, Першин Антон Юрьевич.



2024.12.25

Санкт-Петербург

2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	3
Постановка задачи.....	4
1. Модельный параллелизм.....	5
1.1. Конвейерный параллелизм.....	5
1.2. Тензорный параллелизм.....	7
2. Методы оптимизации распределенного инференса.....	9
2.1. Разреженные коммуникации.....	9
2.2. Мультиплексирование и пространственно-временное планирование графических процессоров.....	10
3. Инструменты для распределенного инференса.....	11
3.1. Возможности платформы.....	11
3.1.1 Ручной параллелизм.....	12
3.1.2. Полуавтоматический параллелизм.....	12
3.1.3. Автоматический параллелизм.....	13
3.2. Практические эксперименты.....	14
ЗАКЛЮЧЕНИЕ.....	15
Список использованной литературы.....	16

ВВЕДЕНИЕ

В современную эпоху активного развития искусственного интеллекта и глубокого машинного обучения большие нейросетевые модели становятся неотъемлемой частью решения множества сложных задач, таких как распознавание образов, обработка естественного языка и автономное управление.

Инференс нейросетевых моделей — это процесс использования уже обученной нейронной сети для предсказания результатов на новых, ранее невидимых данных. В отличие от обучения, где модель настраивает свои веса на основе данных, во время инференса модель уже фиксирована, и ее задача — эффективно применять накопленные знания для решения поставленной задачи.

Однако увеличение сложности и размеров моделей часто приводит к проблемам, связанным с чрезмерно большими временными затратами на инференс или с невозможностью разместить модель в памяти графического ускорителя (GPU). Одним из решений этой проблемы является распределение нейросети между несколькими GPU. Такой способ называется параллелизм на уровне модели

Вопросы эффективного распределения нейросетевых моделей по вычислительным устройствам становятся особенно актуальными в условиях растущих требований к вычислительным ресурсам и необходимости выполнения вычислений в реальном времени. Распределённый инференс позволяет использовать несколько устройств (GPU или CPU) для преодоления ограничений по памяти и вычислительной мощности, а также для сокращения времени отклика системы.

При распределенном инференсе необходимость разделить матрицы весов модели между различными устройствами. Такой подход требует создания определенных "разрезов" нейронной сети, которые будут распределены по доступным ресурсам. Существует два способа "разреза" модели: горизонтальный (межслойный) и вертикальный (внутрислойный). Первый способ достаточно прост в реализации, однако может являться не очень эффективным в связи с тем, что при инференсе обработка данных на каждой группе слоев происходит только на 1 видеокарте, вследствие чего возможны простои остальных и, как следствие, неэффективное использование вычислительных ресурсов. Второй способ позволяет использовать одновременно несколько видеокарт, что сильно может повысить эффективность вычислений. Однако достаточно сложен в реализации.

В данной работе исследуются различные способы распределенного инференса глубоких нейросетевых моделей, а также инструменты, реализующие их.

Постановка задачи

Цель: исследовать различные способы распределенного инференса и инструменты их реализации.

Задачи:

1. Исследование тензорного параллелизма
2. Исследование конвейерного параллелизма
3. Изучение методов оптимизации распределенного инференса
4. Изучение инструментов для распределенного инференса

1. Модельный параллелизм

1.1. Конвейерный параллелизм

Конвейерный параллелизм — параллелизм модели, при котором каждый слой или группа последовательных слоев размещается на отдельном узле. Последовательные пакеты данных (batches) подаются в конвейер, чтобы он оставался заполненным. Обмен данными между узлами ограничивается активациями слоев, где происходит разбиение модели, и соответствующими градиентами активации. [10]

Основным ограничением данного метода является то, что из-за межслойного разделения модели некоторые устройства могут простаивать в ожидании получения данных от предыдущих слоев. Это приводит к появлению “пузырей” конвейера как при прямом, так и при обратном проходе. На рисунке 1 представлена наивная реализация конвейерного параллелизма с прямым и обратным проходом. При таком типе конвейеризации в каждый промежуток времени работает только одна видеокарта:

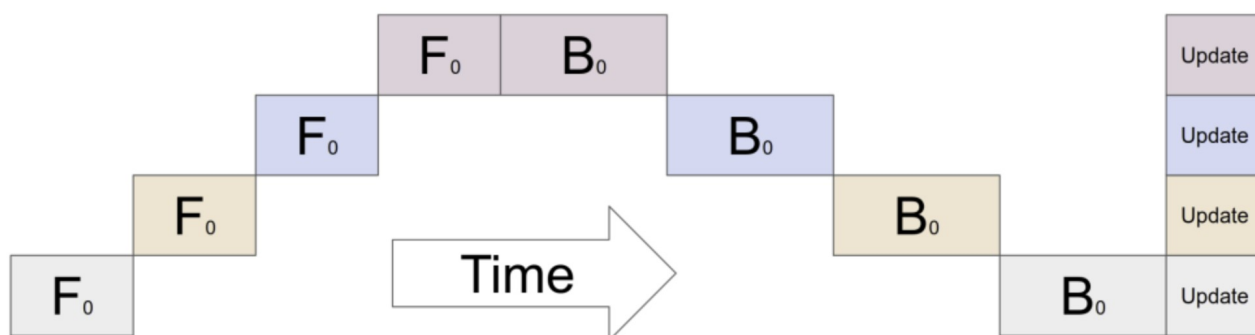


Рисунок 1. Проблема конвейерного параллелизма [4]

Эта проблема может быть решена путем разделения одного батча входных данных на несколько, благодаря чему появляется возможность уменьшить исходный “пузырь” и уменьшить время обучения и инференса. На рисунке 2 представлена реализация конвейерного параллелизма с разделением одного батча данных на несколько микробатчей.

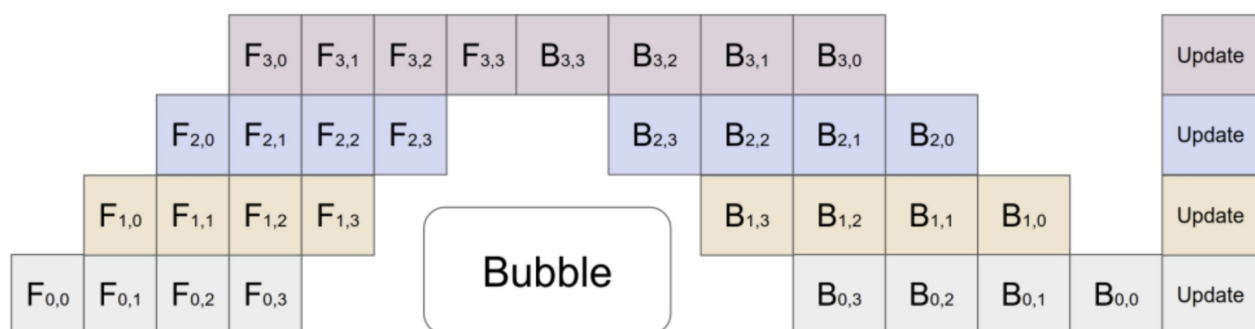


Рисунок 2. Реализация конвейерного параллелизма с разделением батча на несколько микробатчей [4]

Такой способ оптимизации впервые использовался в фреймворке GPipe [4]. Также в статье, посвященной этому фреймворку приводится формула по подсчету оптимального количества микробатчей (1).

$$M \geq 4K(1)$$

Где M — количество микробатчей, на которое необходимо разделить батч, K – количество ускорителей в системе.

При использовании этого метода удалось значительно повысить производительность и скорость обучения относительно наивной реализации. В некоторых случаях удалось практически достигнуть линейного ускорения, относительно одного ускорителя. В дальнейшем многие работы используют этот метод, однако изменяют алгоритмы планирования конвейера для еще большего уменьшения “пузырей” [6, 7, 11]. Однако эти работы в основном могут быть использованы только для ускорения обучения моделей.

В фреймворке PipeDream [6] предлагается использовать конвейерный параллелизм вместе с параллелизмом данных, благодаря чему удастся значительно повысить скорость модели. На каждом этапе конвейера существует возможность использовать параллелизм данных для ускорения как прямого, так и обратного прохода.

Для поиска эффективного разделения модели PipeDream сначала использует профилирование модели, затем алгоритм, основанный на динамическом программировании с целью поиска эффективного разделения нейронной сети.

На первом этапе для каждого слоя l вычисляется T_l и W_l^m — общее время вычислений на прямом и обратном проходе для слоя и время синхронизации веса для слоя соответственно. Для определения этих значений PipeDream профилирует короткий прогон модели с использованием 1000 пакетов на одной из машин.

Алгоритм секционирования берет выходные данные шага профилирования и вычисляет: разделение слоев на этапы, коэффициент репликации для каждого слоя и оптимальное количество микропакетов. Алгоритм секционирования пытается свести к минимуму общее время обучения модели. Для конвейерной системы эта проблема эквивалентна минимизации времени, затрачиваемого на самый медленный этап конвейера. Эта задача может быть разбита на подзадачи: конвейер, который максимизирует пропускную способность при заданном количестве компьютеров, состоит из подконвейеров, которые максимизируют пропускную способность при меньшем количестве машин. Пусть $A(j, m)$ – время, затраченное на самую медленную стадию в оптимальном конвейере между стадиями 1 и j . Пусть $T(i \rightarrow j, m)$ – время, затраченное на одну стадию, охватывающую слои от i до j , реплицированную на m машинах.

$$T(i \rightarrow j, m) = \frac{1}{m} \max \left(\sum_{l=i}^j T_l, \sum_{l=i}^j W_l^m \right) (2)$$

Оптимальный конвейер, состоящий из слоев от 1 до j с использованием m машин, может быть либо одним этапом, повторяемым m раз (параллелизм данных), либо состоять из нескольких этапов (параллелизм данных вместе с конвейерным).

В первом случае:

$$A(j, m) = T(1 \rightarrow j, m) (3)$$

Во втором случае оптимальный конвейер содержит более одного этапа. В этом случае его можно разбить на оптимальный подконвейер, состоящий из слоев от 1 до i с $m - m'$ машинами, за которыми следует один этап со слоями от $i+1$ до j , реплицированных на m' машин. Общая задача сведена к подзадаче поиска оптимального подконвейера:

$$A(j, m) = \min_{1 \leq i < j} \min_{1 \leq m' < m} \max(A(i, m - m'), 2C_i, T(i+1 \rightarrow j, m')) \quad (4)$$

Где C_i — время, затраченное на передачу активаций от уровня i к $i+1$.

Алгоритм позволяет достигать практически линейного ускорения по сравнению с одной видеокартой.

1.2. Тензорный параллелизм

Тензорный параллелизм - метод параллельного разбиения модели, который распределяет тензор параметров каждого слоя на несколько процессоров.

Впервые тензорный параллелизм представлен в статье по Megatron-LM [12] как способ обучения больших языковых моделей на значительном количестве видеокарт. В статье для обучения моделей BERT и GPT-2 использовалось 512 видеокарт. В статье используется модель трансформера, за которой следует двухслойный перцептрон. На рисунке 3 представлен способ разделения перцептрона.

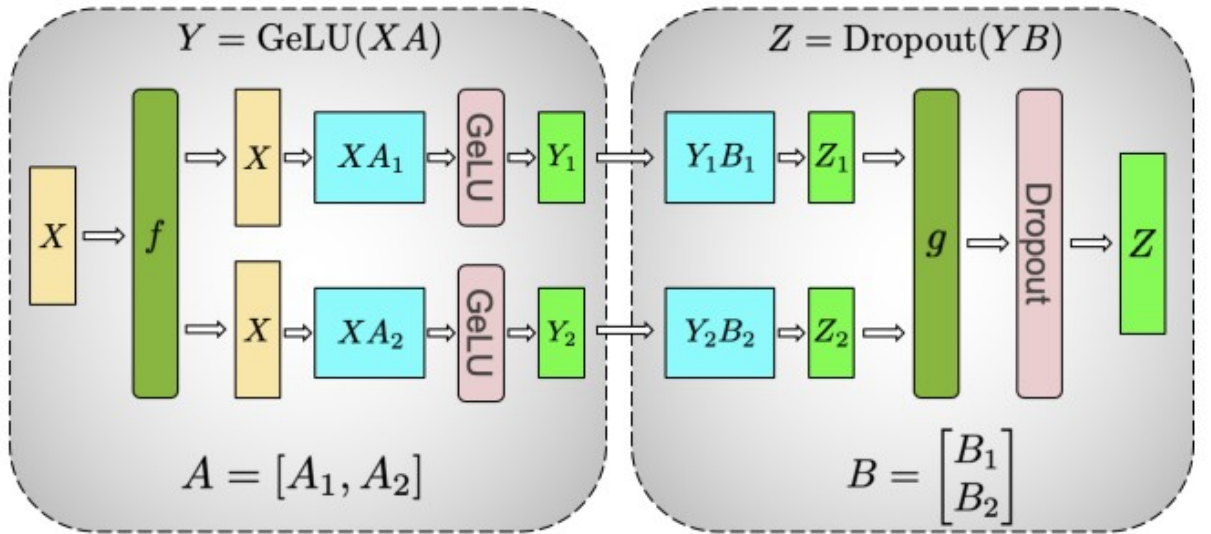


Рисунок 3. Способ разделения перцептрона

Y может быть представлен следующим образом:

$$Y = \text{GeLU}(XA) \quad (5)$$

Можно разделить матрицу весов по столбцам: $A = [A_1, A_2]$. Тогда Y представима в следующем виде:

$$Y = [\text{GeLU}(XA_1), \text{GeLU}(XA_2)] \quad (6)$$

Так как следующую операцию умножения можно также провести в разделенном для Y виде (6), нет необходимости на этом этапе добавлять какие-либо точки синхронизации. Следующая матрица весов разделяется по строкам. Таким образом матрицу Z можно представить в виде:

$$Z = \text{Dropout}(Z_1 + Z_2) = \text{Dropout}(Y_1 B_1 + Y_2 B_2) \quad (7)$$

Следовательно, для получения матрицы Z необходимо добавить точку синхронизации и применить оператор $\text{all-reduce}(g)$ задача которого состоит в том, чтобы собрать полученные матрицы со всех процессоров и произвести операцию редукции. При обратном проходе также необходимо произвести операцию $\text{all-reduce}(f)$.

На рисунке 4 представлен способ разделения блока self-attention.

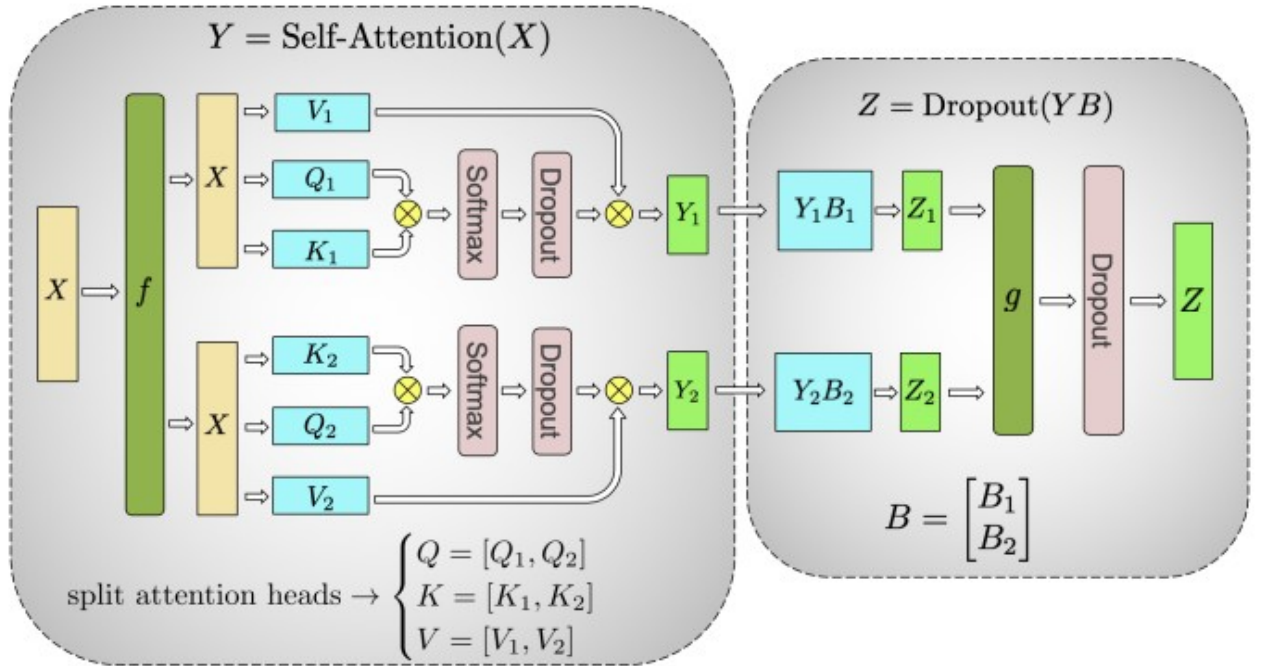


Рисунок 4. разделение блока self-attention

В этом случае разделяются матрицы Q, K, V по столбцам и вычисления производятся независимо на каждом GPU. После этого получаем матрицу $Y = [Y_1, Y_2]$.

Второй этап аналогичен второму этапу для многослойного перцептрона.

Таким образом удастся значительно увеличить скорость обучения и инференс модели, однако данный способ требует поиска эффективного способа разделения матриц весов.

Стоит отметить, что способ параллелизма модели, представленной Megatron-LM можно сочетать с параллелизмом конвейера, вследствие чего появляется возможность обучать и использовать модели с огромным количеством параметров с достаточно низкими для таких сетей временными затратами.

2. Методы оптимизации распределенного инференса

2.1. Разреженные коммуникации

Одним из способов уменьшения времени распределенного инференса является уменьшение коммуникационных связей между GPU. Для этого необходимо найти коммуникационные связи, при удалении которых, инференс нейронной сети не будет ухудшен или ухудшен незначительно.

Так в статье [5] представлен фреймворк DISCO, основная идея которого заключается в следующем: сначала обучается полносвязная сеть, затем определяются ненулевые веса, соответствующие разреженным признакам для связи, и постепенно упрощается коммуникация, при этом оставшиеся веса настраиваются более тонко.

Пусть у нас есть 2 графических процессора. Для слоя l тензор входных данных разделен поровну. Но для следующей операции необходимо, чтобы на каждом GPU находился весь тензор входных данных, однако это крайне невыгодно с точки зрения коммуникаций. Тогда с одного процессора на другой можно передать только часть данных, которые наиболее значительно влияют на инференс нейронной сети. Эта задача эквивалента удалению некоторых весов из слоя модели. Тем самым задача удаление коммуникаций между процессорами эквивалента задачи прунинга (рисунок 5).

Прунинг — это процесс оптимизации модели, при котором удаляются малозначимые элементы структуры нейронной сети, с целью уменьшения ее размера, ускорения работы или сокращения вычислительных затрат, без значительного ухудшения качества модели.

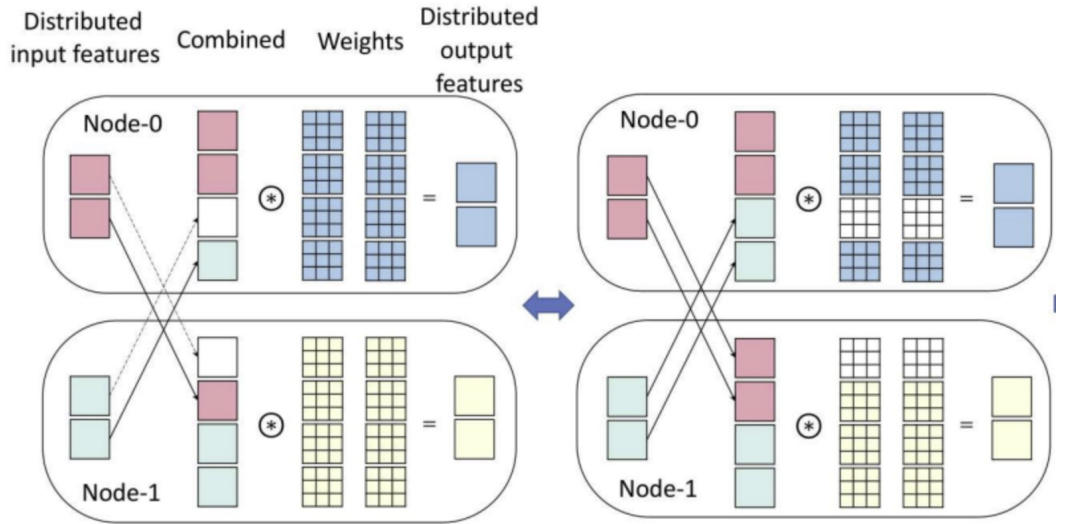


Рисунок 5. Эквивалентность удаления связей и удаления весов для сверточного слоя.

Общая задержка системы зависит от задержки вычислений и коммуникаций. Коммуникационные задержки вычисляются по формуле (8):

$$L_{comm} = \frac{F_s(1 - S_{comm})}{B} \quad (8)$$

где F_s – общий размер входных параметров (байт); B – пропускная способность сети (байт/с). S_{comm} – разреженность связей для каждого узла.

Вычислительный задержки вычисляются по формуле (9):

$$L_{comp} = \frac{C_c(1 - S_{comp})}{NC} \quad (9)$$

где C_c – число операций с плавающей запятой (FLOPs); N – количество процессоров; C – число выполняемых операций в секунду (FLOPS); S_{comp} – разреженность связей для каждого узла.

Можно доказать, что S_{comp} и S_{comm} связаны формулой (10):

$$S_{comm} = \frac{N}{N-1} S_{comp} \quad (10)$$

Вычисления состоят из небольшой части сообщаемых данных и большей части локальных данных. Таким образом, асинхронные вычисления и обмен данными могут быть конвейеризированы, чтобы «скрыть» задержку друг друга. Например, один узел может одновременно начать обработку свертки признаков и собственные веса и получение признаков от другого узла. Следовательно, в режиме «конвейера» задержка вывода для слоя i может быть вычислена по формуле (11):

$$L(i) = \max(L_{comp}(i), L_{comm}(i)) \quad (11)$$

Из уравнения (11) следует, что задержка слоя определяется более медленной задержкой между L_{comm} и L_{comp} . Исходя из уравнений (8), (9) и (10), как L_{comm} , так и L_{comp} уменьшаются за счет увеличения S_{comm} . Следовательно, существует точка равновесия S_{comm}^{eq} где $L_{comm} = L_{comp}$. Она может быть вычислена по формуле (12):

$$S_{comm}^{eq} = \frac{AN - N}{AN - N + 1} \quad (12)$$

$$\text{где } A = \frac{NC}{C_c} \frac{F_s}{B}.$$

2.2. Мультиплексирование и пространственно-временное планирование графических процессоров

Для увеличения пропускной способности инференса нескольких нейронных сетей, находящихся на одном процессоре, существует метод мультиплексирования. Идея метода состоит в том, что ресурсы GPU делятся на части, и задачи выполняются параллельно, используя разные части графического ускорителя. Для этого необходимо понять, какое количество SM ядер наиболее оптимально отдать под вычисления при инференсе каждой сети. Для этого фреймворк D-STACK [3] для каждой сети с K_{max} различными ядрами максимизирует ее работу, вычисляемую по формуле (13):

$$A = \frac{1}{E_t S} \quad (13)$$

где S – количество SM ядер на видеокарте; E_t – время выполнения инференса DNN, которая вычисляется как сумма времени выполнения не распараллеленных и распараллеленных операций.

Для максимизации работы необходимо найти максимум ее производной по E_t (14):

$$A'_{E_t} = \frac{1}{d E_t} \left(\frac{1}{E_t S} \right) = - \frac{1}{E_t^2 S} \quad (14)$$

3. Инструменты для распределенного инференса

На сегодняшний день существует множество различных фреймворков, в которых есть инструменты для распределенного обучения и инференса моделей. Каждый из них создавался для решения определенных задач и имеет свои преимущества и недостатки. Одним из наиболее популярным и универсальным решением является MindSpore. **MindSpore** — платформа глубокого обучения для всех сценариев, направленная на упрощение разработки, эффективное выполнение и унифицированное развертывание для всех сценариев [1]. Фреймворк поддерживает как обучение на одном устройстве, так и распределенное обучение на кластере устройств, включая CPU, GPU и специализированные чипы Ascend. Также MindSpore имеет возможность работать с библиотеками, направленными на оптимизацию вычислений, например, TensorRT

3.1. Возможности платформы

Программный код для MindSpore пишется на языке Python, однако сам код выполняется на C++. При запуске программы исходный код переводится в MindIR (рисунок 6).

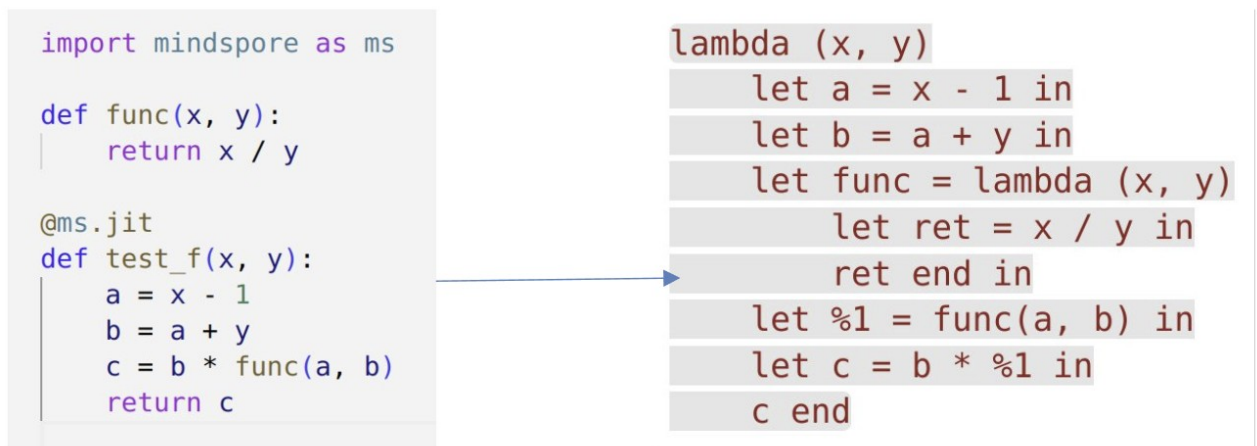


Рисунок 6. Пример перевода python кода в промежуточное представление MindIR

MindIR является промежуточным представлением программы между исходным и целевым языками, которое облегчает анализ и оптимизацию программы для компилятора. Также благодаря нему происходит автоматическое дифференцирование. По семантике код MindIR близок к коду на функциональных языках программирования.

В MindSpore весь параллелизм моделей разделяется на три вида: ручной параллелизм, полуавтоматический параллелизм и автоматический параллелизм.

3.1.1 Ручной параллелизм

При ручном параллелизме пользователю самому необходимо распределять вычисления и данные между видеокартами при помощи распределенных операторов, таких как *AllReduce*, *AlltoAll*, *Broadcast* и др [1]. Для реализации этих операций на GPU фреймворк применяет библиотеку NCCL.

3.1.2. Полуавтоматический параллелизм

При использовании полуавтоматического параллелизма пользователю необходимо задать разбиение вычислений на процессоры, а MindSpore автоматически свяжет вычисления. Ручной параллелизм в фреймворке, в основном, представлен двумя видами параллелизма: тензорным параллелизмом и конвейерным параллелизмом.

При тензорном параллелизме пользователем задаются разбиения для входных данных и для матриц весов. При необходимости MindSpore автоматически между слоями подставляет распределенные операторы. Для примера пусть для вычислений имеются четыре процессора (рисунок 7), рассмотрим следующие операции перемножения матриц (15):

$$Z = (X \cdot W) \cdot V \quad (15)$$

Пусть для X задано разбиение по строкам, у V такое же разбиение, а W разбит по столбцам. После первой операции матрица Y на процессорах представляется в виде (16):

$$Y = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} [W_1, W_2] = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} \quad (16)$$

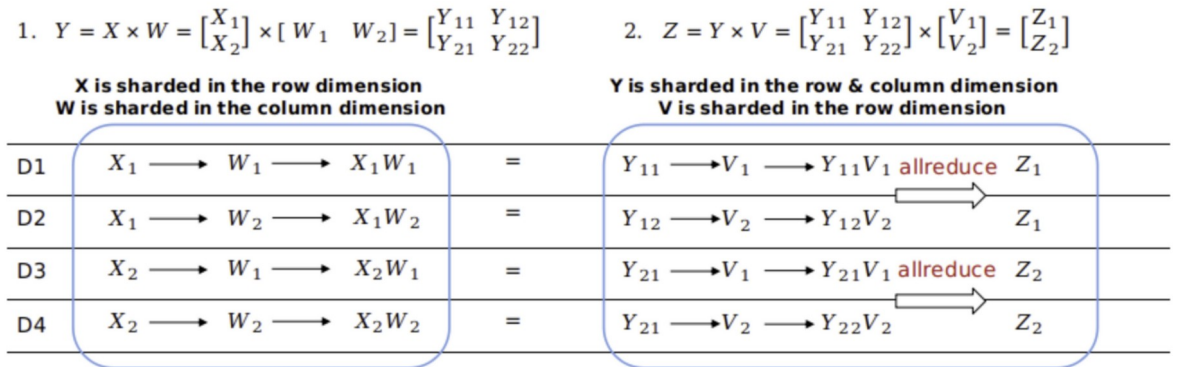


Рисунок 7.

Пример параллелизма тензоров при использовании полуавтоматического параллелизма [1].

При второй операции матрица Z представляется в виде (17):

$$Z = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^2 Y_{1i} V_i \\ \sum_{i=1}^2 Y_{2i} V_i \end{bmatrix} = \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} \quad (17)$$

При выполнении второй операции необходимо после умножений сложить соответствующие матрицы. Для этого MindSpore подставляет оператор *AllReduce*.

Конвейерный параллелизм в MindSpore в своей реализации аналогичен PipeDream [7]. Для его использования необходимо задать, на каких процессорах вычисляются значения каждого слоя, а также количество микробатчей, получаемых из одного батча.

3.1.3. Автоматический параллелизм

При автоматическом параллелизме пользователем задаются только некоторые, ключевые разбиения или не задаются вовсе. MindSpore, при помощи алгоритма поиска стратегии распространения осколков (sharding propagation strategy search algorithm). Этот алгоритм применяется для тензорного параллелизма.

Пусть на некоторых слоях пользователем заданы способы разбиения для входных данных и матриц весов. На основе этих точек для каждого перехода между слоями алгоритм строит таблицу со всевозможными стратегиями разбиения и для каждой стратегии вычисляет стоимость на основе распределенных операторов, которые надо применить при использовании определенной стратегии (рисунок 8). Алгоритм пытается минимизировать стоимость при переходах от одного изначально заданного разбиения к другому.

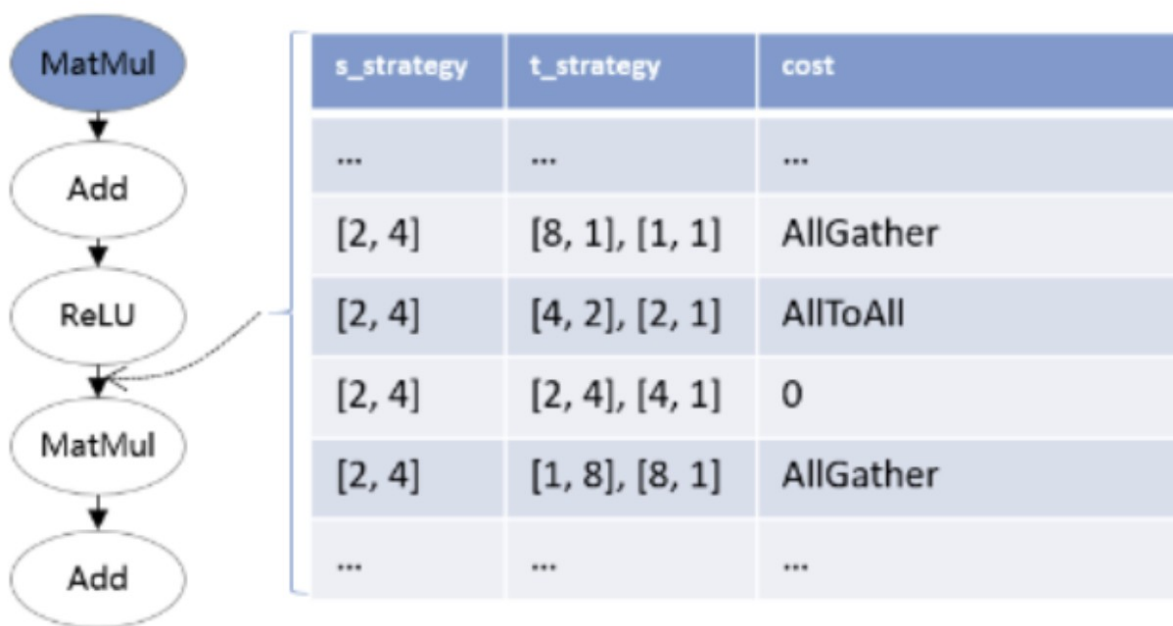


Рисунок 8. Алгоритм поиска стратегии распространения осколков [1]

3.2. Практические эксперименты

Для экспериментов используется платформа с двумя графическими ускорителями. В качестве модели взят трехслойный перцептрон. Первый слой состоит из матрицы весов, размера 784x512 и операции ReLU. Второй слой представлен в виде матрицы 512x512 и ReLU. Третий слой состоит из матрицы размера 512x10. В качестве датасета для обучения использован MNIST. Количество эпох - 10, на каждой эпохе производится 1875 шагов. Тензорный параллелизм может сильно зависеть от стратегии разбиения, поэтому в эксперименте берется разбиение, полученное при использовании алгоритма поиска

стратегии распространения осколков. При конвейерном параллелизме на первом GPU вычисляются данные первого слоя, на втором - второго и третьего слоя соответственно. В таблице 1 представлено время, затраченное на обучение модели в зависимости от типа используемого параллелизма.

Таблица 1. Время, затраченное на обучение, в зависимости от типа параллелизма.

Тип параллелизма	Время (мин)
Конвейерный параллелизм	2.81
Тензорный параллелизм	2.57
Без параллелизма	3.90

Из таблицы можно сделать вывод, что, в среднем, при распараллеливании скорость обучения меньше примерно в 1.45 раза. При этом тензорный параллелизм немного превосходит в скорости конвейерный.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы изучены некоторые способы модельного параллелизма, такие как: тензорный и конвейерный. Исследованы способы модификации распределенного инференса для повышения его скорости. Изучены некоторые инструменты платформы MindSpore для распределенного обучения и инференса и проведены практические эксперименты с реализациями конвейерного и тензорного параллелизма в фреймворке. Показано, что благодаря модельному параллелизму можно уменьшить время обучения сети примерно в 1.45 раза, в зависимости от типа используемого параллелизма.

В дальнейшем планируется углубленное изучение тензорного и конвейерного параллелизма, и поиск новых алгоритмов внутрислойного разбиения нейронной сети и методов оптимизации распределенного инференса. Также предполагается исследование возможностей фреймворков MindSpore, Megatron-LM и DeepSpeed.

Список использованной литературы

1. Distributed Parallel Native // mindspore URL: https://www.mindspore.cn/docs/en/r2.3.1/design/distributed_training_design.html (дата обращения: 15.11.2024).
2. Глубокое обучение // getsomemath URL: http://getsomemath.ru/subtopic/deep_learning/mlp/intro_to_mlp (дата обращения: 15.11.2024).
3. Aditya Dhakal, K. K. Ramakrishnan, Sameer G. Kulkarni D-STACK: High Throughput DNN Inference by Effective Multiplexing and Spatio-Temporal Scheduling of GPUs // arxiv.org. - 2023
4. Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, Zhifeng Chen GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism // arxiv.org. - 2019
5. Minghai Qin, Chao Sun, Jaco Hofmann, Dejan Vucinic Western Digital, Milpitas, California DISCO: DISTRIBUTED INFERENCE WITH SPARSE COMMUNICATIONS // arxiv.org. - 2023
6. Sun Ao, Weilin Zhao, Xu Han, Cheng Yang, Zhiyuan Liu, Chuan Shi, Maosong Sun Seq1F1B: Efficient Sequence-Level Pipeline Parallelism for Large Language Model Training // arxiv.org. - 2023
7. Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, Phil Gibbons PipeDream: Fast and Efficient Pipeline Parallel DNN Training // arxiv.org. - 2018
8. GPU Performance Background User's Guide // nvidia URL: <https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html> (дата обращения: 10.12.2024).
9. Mastering LLM Techniques: Inference Optimization // nvidia URL: <https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/> (дата обращения: 15.12.2024).
10. Andres Rodriguez Deep Learning Systems. - 1-е изд. - San Rafael, California: Morgan And Claypool
11. Penghui Qi, Xinyi Wan, Guangxing Huang & Min Lin ZERO BUBBLE PIPELINE PARALLELISM // arxiv.org. - 2024
12. Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, Bryan Catanzaro Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism // arxiv.org. - 2020