Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра суперкомпьютеров и квантовой информатики

Агаджанян Грант Минасович

Исследование методов обучения свёрточных нейронных сетей с использованием графических ускорителей

Курсовая работа

Научный руководитель:
к.ф.-м.н. Попова Нина Николаевна

Москва, 2025

Аннотация

Исследование методов обучения свёрточных нейронных сетей с использованием графических ускорителей

Агаджанян Грант Минасович

В текущей работе освещены стратегии распределённого обучения свёрточных нейронных сетей, а также проведены эксперименты для трёх подходов с выбранными сетью и датасетом на одном узле суперкомпьютера МГУ-270. Результаты анализируются, после чего формируются некоторые рекомендации к организации распараллеливания процесса обучения.

Abstract

Содержание

1 Введение

В последние годы глубокие нейронные сети (DNN) демонстрируют впечатляющие результаты в широком спектре задач. Рост объёмов данных и моделей сетей опережает увеличение вычислительных возможностей, так сложилось исторически. Обучение на одном устройстве, например GPU, становится затратным с точки зрения времени и памяти. Например, обучение ResNet-50 на одном GPU V100 занимает примерно 29 часов. В связи с этим фактором крупномасштабное распределённое обучение на системах высокопроизводительных вычислений(HPC-системах), а также на GPU-кластерах активно используется в случае крупных моделей и работы с высокоразмерными данными для ускорения времени сходимости и преодоления ограничений по объёму памяти. Для каждой конкретной архитектуры DNN существуют свои подходы и тонкости к организации распараллеливания. В этой работе освещены основные стратегии для свёрточных нейронных сетей(CNN) в контексте производительности и масштабируемости. Даны формальные описания каждой стратегии, а затем представлены результаты экспериментов, выполненных на одном узле суперкомпьютера МГУ-270, с выбранной моделью CNN и датасетом.

2 Постановка задачи

Задачи работы:

1. Рассмотреть существующие подходы к реализации обучения CNN на нескольких GPU.
2. Выбрать подмножество стратегий для экспериментальных исследований.
3. Реализовать обучение на одном узле суперкомпьютера МГУ-270.
4. Исследовать эффективность реализованных версий.

3 Обзор проблематики по теме работы

Основными стратегиями параллелизации фазы обучения в DNN являются параллелизм данных(data parallelism) и модельный параллелизм(model parallelism), которые предполагают разбиение данных и частей модели(как целых слоёв, так и отдельных тензоров) соответственно по вычислительным устройствам. При этом в определённый этап развития нейронных сетей параллелизма данных было достаточно: он прост для понимания и реализации, при этом довольно эффективен с точки зрения масштабирования. Однако, как писалось ранее, масштабирование в этом случае может быть ограничено объёмом памяти (модель слишком большая и не помещается в памяти устройства, а учитывая количество параметров современных моделей - это весьма актуально) и коммуникационными накладными расходами(когда коммуникации составляют значительную долю от общего времени обучения - для коллетивной операции Allreduce, которая является основным узким местом). Учитывая указанные проблемы параллелизма данных и растущие масштабы обучения, исследователи сосредоточены на устранении различных узких мест, связанных с компонентами распределённого обучения DNN(методы оптимизации в алгоритмах обучения, предобученные модели, оптимизация операции Allreduce, оптимизация по памяти и т.д.) Несмотря на все эти усилия, переход к модельному параллелизму всё же неизбежен.

Распределённое обучение DNN можно разделить на четыре этапа:

1. (**IO**) Ввод/вывод, предобработка данных.
2. (**FB**) Прямое распространение(forward phase(, при котором выборки проходят через всю сеть, за которым следует обратный распространение(backpropagation) для вычисления градиентов.
3. (**GE**) Обмен градиентами(если требуется).
4. (**WU**) обновление весов.

Выборки случайным образом разбиваются на мини-батчи (mini-batch) размера B.

Процесс обучения выполняется итеративно на этих мини-батчах с использованием алгоритмов оптимизации, таких как стохастический градиентный спуск (SGD). Веса обновляются с учётом скорости обучения ρ по формуле:

$$w^{\text{iter}+1} \leftarrow w^{\text{iter}} - \rho \frac{1}{B} \sum_{i \in \text{Batch}} \left(\frac{dL}{dw} \right)_i .$$

Процесс повторяется эпохами (до сходимости), причём порядок подачи данных в сеть рандомизируется на каждой эпохе.

Введём некоторые основные обозначения, которые будут использоваться далее:

Обозначение	**Описание**
D	Размер набора данных
B	Размер мини-батча
I	Количество итераций на эпоху, $I = \frac{D}{B}$
E	Количество эпох
G	Количество слоев
x_l	Вход слоя l
y_l	Выход (активация) слоя l
w_l	Вес слоя l
b_l	Смещения слоя l
W_l / H_l	Ширина / высота входа слоя l
C_l	Количество входных каналов слоя l
F_l	Количество выходных каналов слоя l (например, фильтров в сверточном слое)
FW_l / BW_l	Прямое / обратное распространение слоя l
X_l^d	Кортеж размерности d, представляющий входной канал. В 2D свертке $X_l^2 = W_l \times H_l$
Y_l^d	Выходной канал размерности d
K_l^d	Фильтр размерности d. В 2D свертке $K_l^2 = K \times K$
p	Общее количество обрабатывающих устройств (PE)
S	Количество сегментов в конвейерном параллелизме
α	Время, необходимое для установления соединения между устройствами
β	Время для отправки 1 байта между устройствами.
δ	Количество байт на элемент (например, входные данные, активации, веса)
γ	Коэффициент повторного использования памяти

Таблица 1: Обозначения

В модели CNN с G слоями в свёрточном слое l используются следующие тензоры:

- Вход слоя l:

  N образцов, каждый содержит C_l каналов, где каждый канал представляет собой кортеж d-мерных данных:

  $$x_l[N, C_l, X_l^d].$$

  Для 2D слоя X_l^d заменяется на $[W_l, H_l]$, то есть:

  $$x_l[N, C_l, W_l \times H_l].$$

Если контекст понятен, то индекс слоя l и размерность d можно опустить, например:

$$x[N, C, X].$$

- Выход(активация) слоя l:

  N образцов и F_l выходных каналов:

$$y_l[N, F_l, Y_l^d].$$

- Веса слоя l:

  F_l фильтров, каждый с C_l каналами и размером ядра K_l^d:

$$w_l[C_l, F_l, K_l^{\mathrm{d}}]$$

  В некоторых случаях размер фильтра ядра можно опустить, например:

$$w_l[C_l, F_l].$$

- Смещение:

$$bi_l\,[F_l]$$

- Градиенты активации:

$$\frac{\partial L}{\partial y_l}[N, F_l, Y_l^d].$$

- Градиенты весов:

$$\frac{\partial L}{\partial w_l}[C_l, F_l, K_l^d].$$

- Градиенты входа:

$$\frac{\partial L}{\partial x_l}[N, C_l, X_l^d].$$

Адаптация для не-свёрточных слоёв:

- Для послойных операций (пулинг, пакетная нормализация) адаптация не требуется.

- Полносвязный слой с входом $x[N, C, W \times H]$ и F выходами представляется как свёрточный слой с фильтрами размером $W \times H$(padding=0, stride=1), что даёт выход:

$$y[N, F, 1 \times 1].$$

- Для поэлементных слоёв (например, ReLU) $F = C$.

- Для слоёв без весов (пулинг, ReLU) вес формально задаётся как:

$$w[C, F, 0].$$

Последовательная реализация CNN включает следующие шаги для каждого слоя:

- (**IO**) $x[N, C, X] \leftarrow \text{IO}(\text{dataset}, B)$ - загрузка данных для первого слоя.

- (**FB**) $y[N, F, Y] \leftarrow \text{FW}(x[N, C, X], w[C, F, K])$ - прямое распространение.

- (**FB**) $\frac{\partial L}{\partial x}[N, C, X] \leftarrow \text{BW}_{\text{data}}\left(\frac{\partial L}{\partial y}[N, F, Y], w[C, F, K]\right)$ - обратное распространение для входных градиентов.

- (**FB**) $\frac{\partial L}{\partial w}[C, F, K] \leftarrow \text{BW}_{\text{weight}}\left(\frac{\partial L}{\partial y}[N, F, Y], x[N, C, X]\right)$ - обратное распространение для градиентов весов.

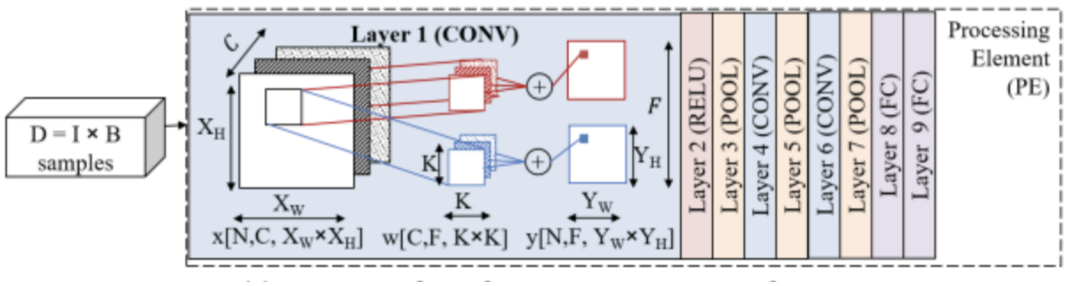

Рис. 1: Иллюстрация последовательной реализации.

Обновление весов на каждой итерации с учётом скорости обучения ρ :

$$(\textbf{WU}) \quad w[C, F, K] \leftarrow \text{WU}\left(\frac{\partial L}{\partial w}[C, F, K], \rho\right)$$

- В тензорах (например, x, y, w) символ $*$ обозначает измерения, значения которых реплицируются между процессами.

- Измерения, разделяемые между PE, помещаются числом процессов p. Например, при параллелизме данных

  $$x[p, *, *]$$

  означает, что вход x разделён по измерению N (число выборок) между p PE, а измерения C и X реплицированы.

- Стрелка

  $\xrightarrow{\text{Allreduce}}$ обозначает операцию Allreduce.

Базовые стратегии различаются способом разделения данных и модели:

1. Распределение выборок данных между PE (параллелизм данных).

2. Разделение выборки данных по пространственным измерениям (ширина/высота) (пространственный параллелизм).

3. Вертикальное разделение нейронной сети по глубине (параллелизм слоёв) с перекрытием вычислений между слоями (также известный как конвейерный параллелизм).

4. Горизонтальное разделение каждого слоя по числу входных и/или выходных каналов (параллелизм каналов и фильтров).

Комбинация двух или более стратегий называется гибридным параллелизмом (например, Data+Filter или Data+Spatial, сокращённо *df* и *ds*).

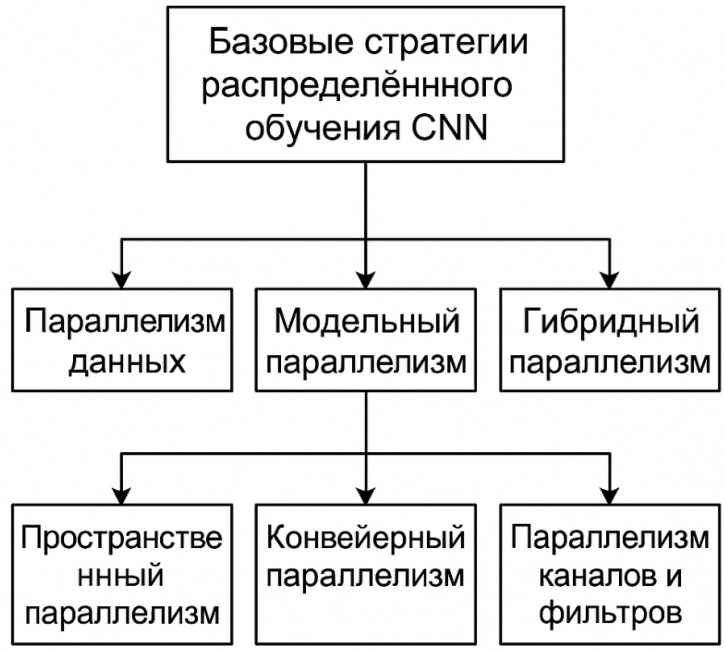

Рис. 2

Параллелизм данных

Вся модель реплицируется на p PE (например, GPU, рис. 3, а набор данных разбивается на поднаборы для каждого PE. Прямое и обратное распространение выполняются независимо на каждом PE с мини-пакетом:

$$B' = \frac{B}{p}.$$

На этапе обмена градиентами выполняется операция `Allreduce` для агрегации градиентов весов:

$$\sum_{i=1}^{p} \left(\frac{dL}{dw}\right)_i.$$

Рис. 3: Иллюстрация параллелизма данных.

Пошаговый алгоритм:

- (**IO**) $(x)_i[p, *, *] \leftarrow \mathrm{IO}(\text{sub-dataset}_i, B')$ в первом слое.
- (**FB**) $(y)_i[p, *, *] \leftarrow \mathrm{FW}((x)_i[p, *, *], w[*, *, *])$.
- (**FB**) $\frac{\partial L}{\partial x}_i[p, *, *] \leftarrow \mathrm{BW}_{\mathrm{data}}\left(\frac{\partial L}{\partial y}_i[p, *, *], w[*, *, *]\right)$.
- (**FB**) $\frac{\partial L}{\partial w}_i[*, *, *] \leftarrow \mathrm{BW}_{\mathrm{weight}}\left(\frac{\partial L}{\partial y}_i[p, *, *], (x)_i[p, *, *]\right)$.
- (**GE**) $\frac{\partial L}{\partial w}[*, *, *] \overset{\text{Allreduce}}{\longleftarrow} \sum_{j=1}^{p}\left(\frac{\partial L}{\partial w}_j[*, *, *]\right)$.
- (**WU**) $w[*, *, *] \leftarrow \mathrm{WU}\left(\frac{\partial L}{\partial w}[*, *, *]\right)$.

Пространственный параллелизм

В начале один PE, именуемый ведущим, принимает мини-батч на каждой итерации, после чего делит его(по сути тензоры) между остальными обрабатывающими элементами по пространственным измерениям(H, W, D). Каждый PE выполняет этапы прямого и обратного прохода локально.

Для свёрточных слоёв необходимы так называемые гало обмены(halo exchange), так как фильтру(ядру свёртки) размера KxK на границе обрабатываемой области потребуются данные, находящиеся на других PEs. Размер передаваемых данных зависит от того, как разделены пространственные измерения, и длины шага фильтра(stride).

На этапе обратного прохода вычисление $\frac{\partial L}{\partial x_i}$ требует обмена граничными данными для соответствующего $\frac{\partial L}{\partial y_i}$. Для вычисления градиентов весов требуется $(x)_{i+}$, однако повторного обмена граничными данными не требуется, так как ранее переданные значения $(x)_i$ могут быть повторно использованы. На этапе обновления весов выполняется операция Allreduce для суммы $\frac{\partial L}{\partial w}$.

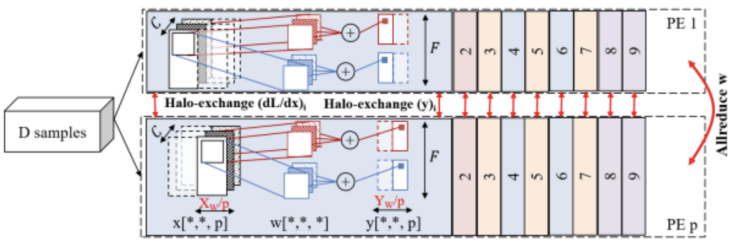

Рис. 4: Иллюстрация пространственного параллелизма.

Пошаговый алгоритм:

$$
\begin{aligned}
&(\mathbf{IO}) \quad x[*,*,*] \leftarrow \mathrm{IO}(\mathrm{dataset}, B) \\
&(\mathbf{IO}) \quad (x)_i[*,*,p] \overset{\text{Scatter}}{\longleftarrow} x[*,*,*] \text{ in the first layer.} \\
&(\mathbf{FB}) \quad (x)_{i+}[*,*,p] \overset{\text{halo}}{\longleftarrow} (x)_i[*,*,p] \\
&(\mathbf{FB}) \quad (y)_i[*,*,p] \leftarrow \mathrm{FW}((x)_{i+}[*,*,p], w[*,*,*]) \\
&(\mathbf{FB}) \quad \left(\frac{\partial L}{\partial y}\right)_{i+}[*,*,p] \overset{\text{halo}}{\longleftarrow} \left(\frac{\partial L}{\partial y}\right)_i[*,*,p] \\
&(\mathbf{FB}) \quad \left(\frac{\partial L}{\partial x}\right)_i[*,*,p] \leftarrow \mathrm{BW}_{\mathrm{data}}\left(\left(\frac{\partial L}{\partial y}\right)_{i+}[*,*,p], w[*,*,*]\right) \\
&(\mathbf{FB}) \quad \left(\frac{\partial L}{\partial w}\right)_i[*,*,*] \leftarrow \mathrm{BW}_{\mathrm{weight}}\left(\left(\frac{\partial L}{\partial y}\right)_i[*,*,p], (x)_{i+}[*,*,p]\right) \\
&(\mathbf{GE}) \quad \frac{\partial L}{\partial w}[*,*,*] \overset{\text{Allreduce}}{\longleftarrow} \sum_{i=1}^{p}\left(\frac{\partial L}{\partial w}\right)_i[*,*,*] \\
&(\mathbf{WU}) \quad w[*,*,*] \leftarrow \mathrm{WU}\left(\frac{\partial L}{\partial w}[*,*,*]\right)
\end{aligned}
$$

Параллелизм фильтров и каналов(горизонтальный)

Идея заключается в том, что каждый слой сети делится поровну по количеству входных или выходных каналов(в случае выходных каналов равносильно делению по количеству фильтров) между PEs. Каждый PE хранит часть весов данного слоя и частично вычисляет выходные данные как на этапе прямого, так и на этапе обратного прохода. Так как каждый PE выполняет обновление весов только для своей части, этап обмена градиентами весов пропускается.

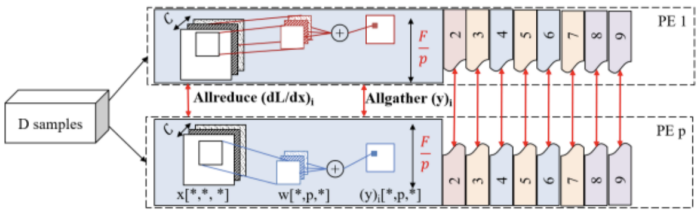

Рис. 5: Иллюстрация параллелизма фильтров.

Пошаговый алгоритм:

$$(\mathbf{IO}) \quad x[*,*,*] \leftarrow \mathrm{IO}(\mathrm{dataset}, B)$$

$$(\mathbf{IO}) \quad (x)_i[*,*,*] \xleftarrow{\text{Bcast}} x[*,*,*] \text{ in the first layer.}$$

$$(\mathbf{FB}) \quad (y)_i[*,p,*] \leftarrow \mathrm{FW}((x)_i[*,*,*], w[*,p,*])$$

$$(\mathbf{FB}) \quad y[*,*,*] \xleftarrow{\text{Allgather}} \bigcup_{i=1}^{p} (y)_i[*,p,*]$$

$$(\mathbf{FB}) \quad \left(\frac{\partial L}{\partial x}\right)_i [*,*,*] \leftarrow \mathrm{BW}_{\mathrm{data}} \left(\left(\frac{\partial L}{\partial y}\right)_i [*,p,*], w[*,p,*] \right)$$

$$(\mathbf{FB}) \quad \frac{\partial L}{\partial x}[*,*,*] \xleftarrow{\text{Allreduce}} \sum_{i=1}^{p} \left(\frac{\partial L}{\partial x}\right)_i [*,*,*]$$

$$(\mathbf{FB}) \quad \frac{\partial L}{\partial w}[*,p,*] \leftarrow \mathrm{BW}_{\mathrm{weight}} \left(\left(\frac{\partial L}{\partial y}\right)_i [*,p,*], (x)_i[*,*,*] \right)$$

$$(\mathbf{WU}) \quad w[*,p,*] \leftarrow \mathrm{WU}\left(\frac{\partial L}{\partial w}[*,p,*]\right)$$

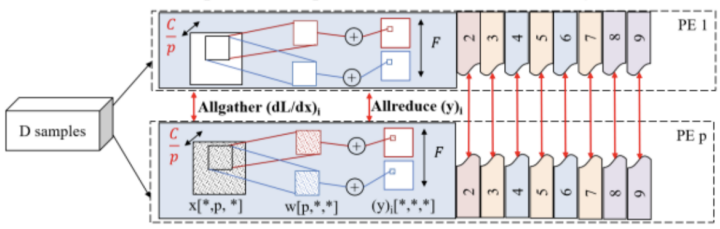

Рис. 6: Иллюстрация параллелизма каналов.

Пошаговый алгоритм:

$$(\mathbf{FB}) \quad (y)_i[*,*,*] \leftarrow \mathrm{FW}((x)_i[*,p,*], w[p,*,*])$$

$$(\mathbf{FB}) \quad y[*,*,*] \xleftarrow{\text{Allreduce}} \sum_{i=1}^{p} (y)_i[*,*,*]$$

$$(\mathbf{FB}) \quad \left(\frac{\partial L}{\partial x}\right)_i [*,p,*] \leftarrow \mathrm{BW}_{\mathrm{data}} \left(\left(\frac{\partial L}{\partial y}\right)_i [*,*,*], w[p,*,*] \right)$$

$$(\mathbf{FB}) \quad \frac{\partial L}{\partial x}[*,*,*] \xleftarrow{\text{Allgather}} \bigcup_{i=1}^{p} \left(\frac{\partial L}{\partial x}\right)_i [*,p,*]$$

Параллелизм слоёв(вертикальный)

Данный подход к параллелизму модели заключается в разделении CNN по её глубине (количество слоев G) на $p \leq G$ составных слоев, при этом каждый составной слой назначается одному PE. Рассматривается конвейерная реализация данного параллелизма модели.

Мини-батч делится на S сегментов размером $\frac{B}{S}$. На каждом этапе прямое вычисление составного слоя i-го на сегменте данных s выполняется одновременно с вычислением составного слоя $(i+1)$-го на сегменте данных $s-1$ и далее. Обратное вычисление осуществляется в обратном порядке.

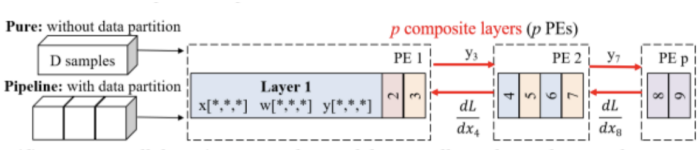

Рис. 7: Иллюстрация параллелизма слоёв.

Гибридный вариант

Гибридный вариант - смесь нескольких рассмотренных выше стратегий. Можно рассмотреть filter+data, при котором p обрабатывающих элементов (PE) организованы в $p1$ групп, каждая из которых состоит из $p2 = \frac{p}{p1}$ элементов. В рамках этой стратегии реализуется параллелизм фильтров внутри каждой группы и параллелизм данных между группами. Для PE с индексом $1 \leq i \leq p2$ внутри группы $1 \leq j \leq p1$

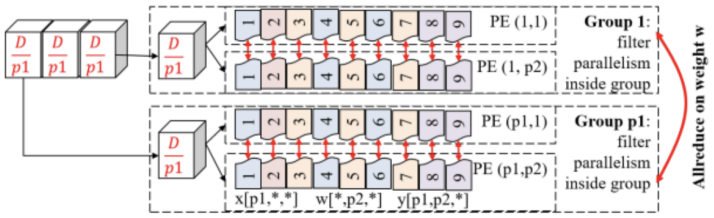

Рис. 8: Иллюстрация гибридного параллелизма(data+filter).

Пошаговый алгоритм:

(**IO**) $(x)_j[p1, *, *] \leftarrow \mathrm{IO}(\text{sub-dataset}_j, B')$

(**IO**) $(x)_{ij}[p1, *, *] \overset{\text{Bcast}}{\longleftarrow} (x)_j[p1, *, *]$ в первом слое

Параллелизм фильтров внутри группы из $p2$ PEs:

(**FB**) $(y)_{ij}[p1, p2, *] \leftarrow \mathrm{FW}((x)_{ij}[p1, *, *], w[*, p2, *])$

(**FB**) $(y)_j[p1, *, *] \overset{\text{Allgather}}{\longleftarrow} \bigcup_{i=1}^{p2} (y)_{ij}[p1, p2, *]$

(**FB**) $\left(\frac{\partial L}{\partial x}\right)_{ij}[p1, *, *] \leftarrow \mathrm{BW}_{\text{data}}\left(\left(\frac{\partial L}{\partial y}\right)_{ij}[p1, p2, *], w[*, p2, *]\right)$

(**FB**) $\left(\frac{\partial L}{\partial x}\right)_{j}[p1, *, *] \overset{\text{Allreduce}}{\longleftarrow} \sum_{i=1}^{p2} \left(\frac{\partial L}{\partial x}\right)_{ij}[p1, *, *]$

Параллелизм данных в $p1$ группах:

(**FB**) $\left(\frac{\partial L}{\partial w}\right)[*, p2, *] \leftarrow \mathrm{BW}_{\text{weight}}\left(\left(\frac{\partial L}{\partial y}\right)_{ij}[p1, p2, *], (x)_{ij}[p1, *, *]\right)$

(**GE**) $\frac{\partial L}{\partial w}[*, p2, *] \overset{\text{Allreduce}}{\longleftarrow} \sum_{j=1}^{p1} \left(\frac{\partial L}{\partial w}\right)_j [*, p2, *]$

(**WU**) $w[*, p2, *] \leftarrow \mathrm{WU}\left(\frac{\partial L}{\partial w}[*, p2, *]\right)$

4 Исследование и построение решения задачи

1. Определимся со стратегиями, которые мы будем использовать в рамках эксперимента: параллелизм данных, конвейерный параллелизм, тензорный параллелизм входных/выходных каналов.

2. Выберем достаточно крупную (по числу параметров и слоёв) модель CNN, например, `ResNet-152`.

3. Подберём объёмный датасет (ведь нам выделяется несколько вычислительных устройств, их надо «нагрузить»): `Tiny-ImageNet-200`.

4. Осуществим поиск инструментов для практической реализации обучения в популярных фреймворках DNN. В `PyTorch` есть всё необходимое для проведения эксперимента в рамках выбранных нами стратегий.

5. Каждая из алгоритмов нуждается в индивидуальной ручной настройке параметров — от этого зависит эффективность масштабирования! Поэтому выполняем серии запусков, варьируя ключевые параметры.

6. По результатам экспериментов делаем выводы об эффективности той или иной стратегии.

5 Описание практической части

5.1 Архитектура ResNet-152

Для экспериментов выбрана CNN ResNet-152 - вариант с 152 слоями, построенный на *bottleneck*-блоках.

5.1.1 Основные компоненты

- **Начальный блок**:
  - Свёрточный слой 7×7, 64 фильтра, шаг 2.
  - MaxPool 3×3, шаг 2.
- **Bottleneck-блок**:
  1. 1×1 свёртка для уменьшения размерности.
  2. 3×3 свёртка.
  3. 1×1 свёртка для восстановления размерности.
  4. Пропускная (*identity*) или проекционная (*projection*) связь.
- **Конфигурация слоёв**: четыре стадии, каждая со своим числом блоков:

Стадия	**Выходной размер**	**Число bottleneck-блоков**
conv2_x	56×56	3
conv3_x	28×28	8
conv4_x	14×14	36
conv5_x	7×7	3

- **Завершающие слои**:
  - Global Average Pooling.
  - Полносвязный слой (1000 нейронов для ImageNet).
  - Softmax.

5.2 Датасет Tiny-ImageNet-200

В качестве датасета для эксперимента выбрана Tiny-ImageNet-200 — уменьшенная версия ImageNet, часто используемая для исследований в области обучения представлений и больших архитектур.

5.2.1 Основные характеристики

- **Число классов**: 200 (подмножество классов ImageNet).
- **Размер изображений**: 64×64 пикселя, цветные (RGB).
- **Разметка и разделение**:
  - *Train*: 200 классов $\times$ 500 изображений = 100 000 образцов.
  - *Validation*: 200 классов $\times$ 50 изображений = 10 000 образцов.
  - *Test*: 10 000 изображений без публичных меток (по 50 на класс).

5.3 Параллелизм данных

5.3.1 Описание реализации параллелизма данных

В файле `DDP.py` параллелизм данных реализован с помощью механизма **Distributed Data Parallel** (DDP) из PyTorch. Основные шаги:

1. **Инициализация распределённой среды.** Каждый процесс, запущенный через `torchrun -nproc_per_node`, вызывает

```
torch.distributed.init_process_group(
    backend='nccl',
    init_method='env://',
    world_size=...,
    rank=...
)
```

и привязывает себя к локальному GPU через `torch.cuda.set_device(local_rank)`.

2. **DistributedSampler.** Для обучающего датасета создаётся

```
DistributedSampler(train_dataset,
num_replicas=world_size, rank=rank, shuffle=True)
```

который разбивает данные на равные куски по числу процессов и гарантирует независимое перемешивание через `sampler.set_epoch(epoch)`.

3. **Обёртка модели.** Модель (ResNet-152) конвертируется в `SyncBatchNorm`, копируется на локальный GPU и оборачивается:

```
model.to(device)
ddp_model = DistributedDataParallel(model, device_ids=[local_rank])
```

DDP автоматически выполняет broadcast начальных параметров и регистрирует хуки all-reduce для градиентов.

4. **Forward / backward + синхронизация.** В каждой итерации:

```
outputs = ddp_model(inputs.to(device))
loss = criterion(outputs, labels.to(device))
loss.backward()
```

При `loss.backward()` для каждого параметра выполняется `all_reduce`.

5. **Optimizer.step() и поддержание синхронности.** После усреднения градиентов каждый процесс локально вызывает `optimizer.step()`, благодаря чему обновлённые параметры везде совпадают. Вывод логов и сохранение модели выполняются только на процессе `rank==0`.

5.3.2 Анализ экспериментальных результатов DDP

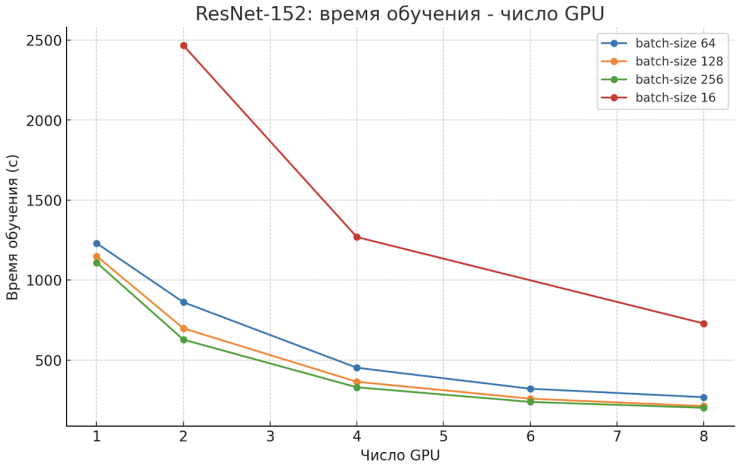

Рис. 9

На графике зависимости времени обучения ResNet-152 от числа GPU для batch-size = 16, 64, 128, 256 видно, что при росте числа устройств ускорение далеко от идеального (линейного) и зависит от размера батча.

1. Масштабируемость и коммуникационные издержки. Обмен градиентами и статистиками BatchNorm реализован через NCCL all-reduce. Время all-reduce растёт с числом процессов (обычно $\mathcal{O}(\log P)$ или $\mathcal{O}(P)$), поэтому доля коммуникации увеличивается, а выигрыш от добавления GPU убывает (закон Амдала).

2. Влияние размера батча.

- *Малые батчи* (16): GPU быстро считаю градиенты и простаивают, дожидаясь обмена. Коммуникация доминирует, масштабирование близко к плато.
- *Большие батчи* (128, 256): время вычислений преобладает над коммуникацией, данные хорошо насыщают GPU, а относительные накладные расходы на all-reduce снижаются. При малом числе GPU наблюдается почти линейное ускорение (2×, 4×), при большем — убывающая отдача, но реальный выигрыш всё ещё заметен.

3. Другие узкие места. Нагрузка на I/O и CPU (data loading, аугментации) возрастает с числом процессов. При 8 GPU и 32 loader-потоках возможны задержки диска/-файловой системы.

4. Вывод. Результаты подтверждают классический компромисс: добавление GPU ускоряет обучение, но эффективность падает из-за коммуникационных и I/O издержек. Оптимальное сочетание batch-size и числа GPU достигается тогда, когда время на вычисления значительно превосходит накладные расходы на синхронизацию.

5.4 Конвейерный параллелизм

5.4.1 Описание реализации конвейерного параллелизма

В файле PIPELINE.py для ускорения обучения используется *конвейерный параллелизм* (pipeline parallelism) при распределении слоёв ResNet-152 по нескольким GPU. В качестве модели взята ResNet-152 из TorchVision, адаптированная для 200 классов (Tiny-ImageNet-200) заменой выходного слоя на

```
model.fc = nn.Linear(model.fc.in_features, 200)
```

Для конвейера применяется класс `torch.distributed.pipeline.sync.Pipe`, который автоматически разбивает модель на сегменты, отправляет их на разные устройства и обрабатывает входные данные порциями (микробатчами).

Разбиение модели на сегменты Модель делится на восемь последовательных сегментов:

1. входной блок: `conv1` + `bn1` + `relu` + `maxpool`;
2. `layer1`;
3. `layer2`;
4. первая половина `layer3`;
5. вторая половина `layer3`;
6. `layer4`;

7. `avgpool` + `Flatten`;

8. классификатор `fc`.

Затем эти сегменты равномерно распределяются по `num_stages` GPU, так что каждый девайс получает сопоставимое число слоёв. Перенос на устройства осуществляется с помощью метода `.to(device)`.

Конвейер с микробатчами При создании `Pipe` модель собирается обратно в один `nn.Sequential`, но разделяется на `chunks` микробатчей:

```
pipeline = Pipe(
    nn.Sequential(*all_segments),
    balance=balance,
    devices=devices,
    chunks=args.chunks,
    checkpoint='never'
)
```

Значение `chunks` определяет глубину конвейера: чем больше микробатчей, тем лучше перекрываются вычисления на разных этапах и меньше «пузырей» простоя. При `chunks`=1 конвейер вырождается в последовательную обработку одного большого батча.

Цикл обучения

1. Изображения перемещаются на GPU первого этапа.

2. Вызывается `output = pipeline(images)`, который разбивает батч на микробатчи и прогоняет их по конвейеру.

3. Метки классов перемещаются на тот же GPU, что и выходы, и вычисляется `CrossEntropyLoss`.

4. Обратное распространение через `loss.backward()` автоматически передаёт градиенты между этапами.

5. Обновление параметров выполняется `optimizer.step()`.

Инициализация RPC (Remote Procedure Call) и её завершение (`rpc.init_rpc` / `rpc.shutdown`) обеспечивает взаимодействие между этапами в многопроцессном режиме, однако в локальной конфигурации с одним процессом и несколькими GPU достаточно параметров `devices` и `balance` в `Pipe`.

5.4.2 Анализ экспериментальных результатов PIPELINE

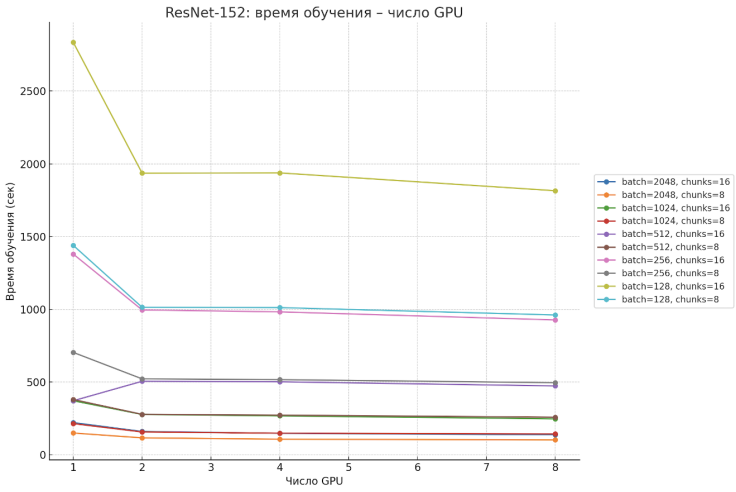

Рис. 10

На рисунке показано время обучения в зависимости от числа GPU при различных комбинациях `batch_size` и числа микробатчей `chunks`.

- При росте GPU с 1 до 2–4 наблюдается резкое падение времени — модель успешно распараллеливается, и каждый GPU получает сопоставимый объём работы.
- Начиная примерно с 4–6 устройств кривая выходит на *плато*: дальнейшее добавление GPU даёт всё меньший выигрыш.
- Основные причины насыщения:

  1. **Пузыри конвейера** — периоды простоя на старте и в конце обработки батча, особенно заметные при небольшом числе микробатчей.
  2. **Коммуникационные издержки** — передача активаций между этапами при каждом forward/backward.

3. **Дисбаланс сегментов** — если один этап значительно „тяжелее“ других, он задаёт скорость всего конвейера.

4. **Слишком маленький общий батч** — при фиксированном `batch_size` на большее число GPU приходится мало данных, и устройства недозагружены.

- `batch_size` и `chunks` сильно влияют:
  - Большой `batch_size` обеспечивает достаточную загрузку каждого GPU.
  - Большое число микробатчей (`chunks` $\gtrsim$ `num_stages`) сокращает пузыри и повышает перекрытие вычислений.

Таким образом, конвейерный параллелизм даёт близкое к линейному ускорение до некоторого числа GPU, после чего масштабируемость ограничивается коммуникациями, дисбалансом и накладными расходами. Для дальнейшего улучшения требуется либо рост объёма вычислений (бóльшие батчи или крупнее модель), либо оптимизация коммуникаций и расписания конвейера.

5.5 Тензорный параллелизм

5.5.1 Описание реализации тензорного параллелизма

В файле TENSOR.py реализован **тензорный параллелизм** — разбиение вычислений слоя по каналам тензоров на несколько GPU. Для этого написаны специальные классы: `ParallelConv2d` и `ParallelLinear`, которые заменяют стандартные слои `nn.Conv2d` и `nn.Linear` в модели ResNet-152. Ключевые моменты реализации следующие:

- **Разделение весов и каналов.** В конструкторе каждого параллельного слоя вычисляется, какую часть каналов (входных или выходных) должен обрабатывать данный процесс. Для режима `out` каждый GPU хранит свой набор выходных каналов, для `in` — часть входных каналов.

- **Коллективные операции.**

- `out`: после локального вычисления выходных каналов используется кастомная дифференцируемая функция `AllGatherGrad`, в которой в `forward` выполняется `dist.all_gather`, а в `backward` — `dist.all_reduce` и разбиение градиента.
- `in`: после локальной свёртки/линейного слоя выполняется `dist.all_reduce` суммы частичных результатов, а затем (при наличии bias) добавляется смещение.

- **Автоматическая замена слоёв.** Функция `replace_layers_with_parallel` рекурсивно обходит ResNet-152 и подменяет все `Conv2d`/`Linear` на их параллельные аналоги, копируя в них соответствующие части исходных весов.

5.5.2 Режимы параллелизма

В коде поддерживаются три режима, задаваемые аргументом `-parallel_mode`:

`out` Разбиение по *выходным* каналам. Применимо, если out_channels $\geq N_{\text{GPU}}$. После локального расчёта части выхода используется `all_gather` (сборка каналов) через `AllGatherGrad`.

`in` Разбиение по *входным* каналам. Применимо, если in_channels $\geq N_{\text{GPU}}$. Каждый GPU считает вклад по своему фрагменту входа, затем выходы суммируются через `all_reduce`.

`hybrid` Гибридный режим: для каждого слоя автоматически выбирается `in` (если достаточное число входных каналов) или `out` (если входных каналов меньше числа GPU). Такой подход сочетает достоинства обоих способов и предотвращает появление нулевых размерностей.

5.5.3 Анализ результатов экспериментов

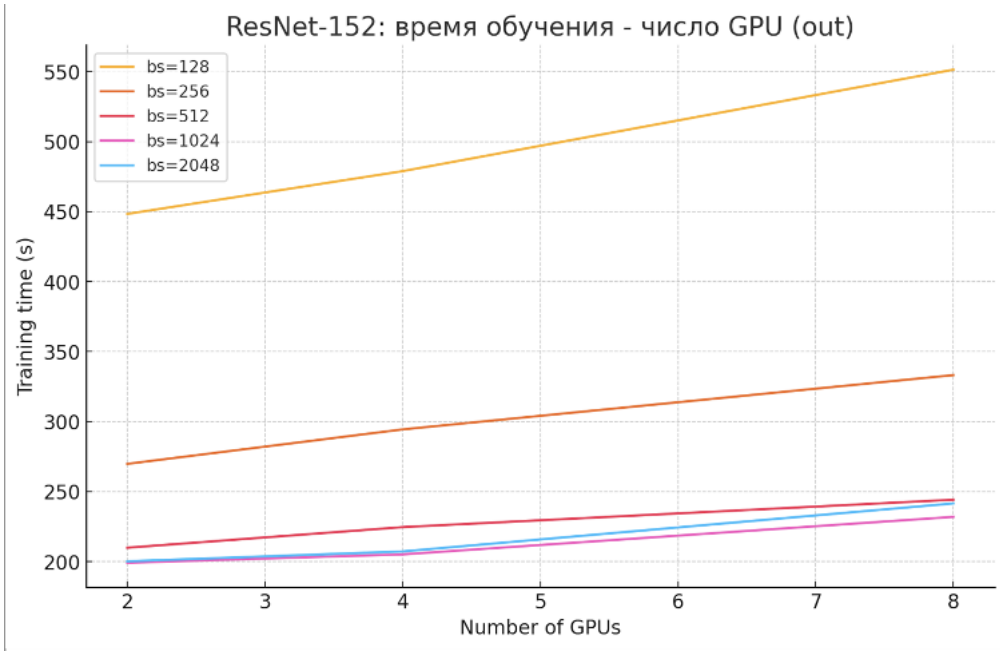

Рис. 11

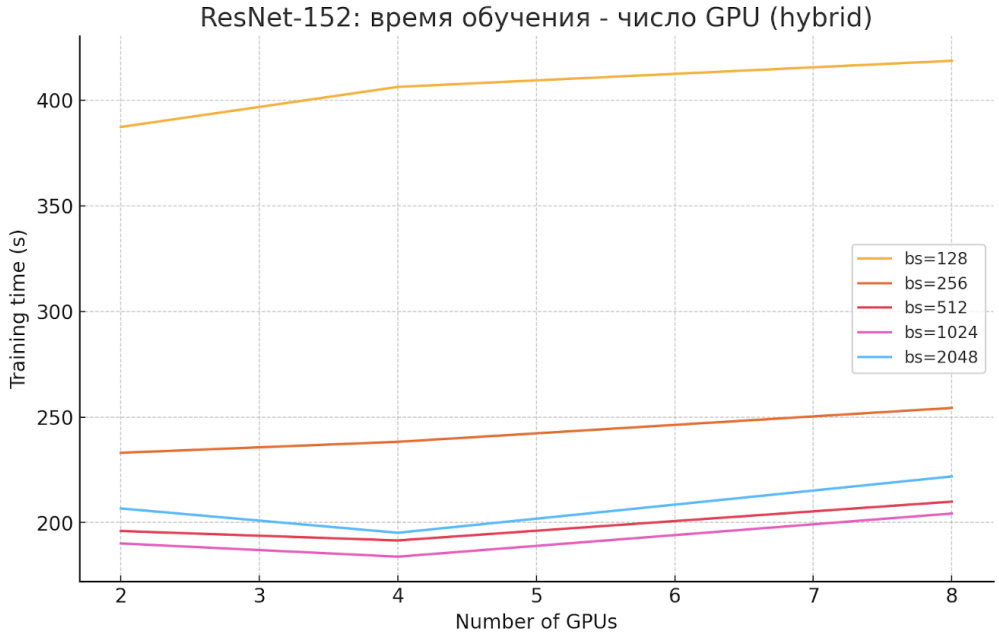

Рис. 12

Выше приведены два графика зависимости суммарного времени обучения ResNet-152 от числа GPU (2, 4, 8) для пяти значений глобального batch size (128, 256, 512, 1024, 2048): сначала для чистого режима `out`, затем для режима `hybrid`.

Выводы по режиму `out`. С ростом числа GPU время обучения растёт (особенно при малых batch). Это объясняется тем, что каждую итерацию выполняется множество коллективных операций `all_gather`/`all_reduce` на всё большее число устройств, и коммуникационный оверхед быстро перевешивает выигрыш от распределения свёртки.

Выводы по режиму `hybrid`. Гибридный режим даёт небольшое ускорение при переходе с 2 на 4 GPU, особенно для больших batch (1024, 2048), поскольку крупные слои переводятся в режим `in` (суммирование вместо конкатенации), что снижает объём передаваемых данных. При дальнейшем увеличении GPU время снова растёт из-за тех же коммуникационных затрат.

Влияние размера batch. Увеличение batch size снижает число итераций, что в целом уменьшает суммарное время обучения, но не отменяет роста коммуникационных накладных расходов при большом числе GPU.

Заключение

1. В главе 3 были рассмотрены основные стратегии распределённого обучения CNN:

   - параллелизм данных;
   - модельный параллелизм (пространственный, конвейерный, параллелизм каналов и фильтров);
   - гибридный параллелизм.

   Модельные и гибридные версии применяются для крупных моделей и высокоразмерных данных, чтобы ускорить сходимость и преодолеть ограничения памяти за счёт разделения модели на несколько устройств.

2. Для экспериментальных исследований были выбраны три стратегии:

   - **Data-Parallel (DDP)**;
   - **Pipeline-Parallel**;
   - **Tensor-Parallel** в трёх режимах (`out`, `in`, `hybrid`).

   В качестве модели использована ResNet-152, обучение проведено на Tiny-ImageNet-200.

3. Эксперименты успешно выполнены на одном узле МГУ-270 с разнообразными конфигурациями (от 1 до 8 GPU, наборы batch size от 16 до 2048, режимы параллелизма).

4. Основные выводы по сравнительному анализу:

   (a) *Data-Parallel (DDP)* показал наилучшую масштабируемость по времени: при росте числа GPU (1→8) общее время обучения заметно уменьшалось для всех размеров batch.

(b) *Pipeline-Parallel* даёт ускорение относительно однопроцессного запуска, но чувствителен к выбору числа чанков (chunks) и балансу между этапами: слишком мало chunks — низкая загрузка; слишком много — высокий коммуникационный оверхед.

(c) *Tensor-Parallel* (`out`, `in`, `hybrid`) снижает требования к памяти на GPU, но чистые режимы (`out` и `in`) плохо масштабируются во времени из-за высокой стоимости коллективных операций на уровне сотен слоёв. Гибридный режим `hybrid` даёт небольшой выигрыш при переходе 2→4 GPU на больших batch, однако при дальнейшем росте числа GPU коммуникационный оверхед вновь преобладает.

5. Рекомендации:

(a) Если модель помещается в память одного GPU и требуется максимальная скорость, предпочтителен *DDP*.

(b) Для очень глубоких или крупномасштабных моделей, когда памяти одного устройства не хватает, эффективно сочетать *pipeline-* и *tensor-parallel* подходы по нескольким уровням параллелизма.

Список литературы

[1] *Julia Gusak Daria Cherniuk, Alena Shilova Alexandr Katrutsa и др.* Survey on Effcient Training of Large Neural Networks / Alena Shilova Alexandr Katrutsa и др. Julia Gusak, Daria Cherniuk. — 2022.

[2] *Albert Njoroge Kahira Truong Thao Nguyen, Leonardo Bautista Gomez и др.* An Oracle for Guiding Large-Scale Model/Hybrid Parallel Training of Convolutional Neural Networks / Leonardo Bautista Gomez и др. Albert Njoroge Kahira, Truong Thao Nguyen. — 2021.

[3] *PyTorch Team.* Distributed Training Overview. — 2024. `https://pytorch.org/tutorials/beginner/dist_overview.html`.