Анализ существующих фреймворков и библиотек, имеющих API на языке Python для распределенного и параллельного использования графических процессорных устройств с целью их внедрения в LightAutoML

Сколковский институт науки и технологий
CDISE

Лаборатория HPC

автор: Ришат Загидуллин
научный руководитель: Сергей Рыкованов, Ph.D.

2021

Цель и задачи работы

Цель данной работы — найти наиболее оптимальный технологический инструментарий для внедрения распределенных ГПУ вычислений в LightAutoML — фреймворк для построения составных моделей машинного обучения и автоматической обработки данных. Для определения оптимальности существующих решений выставлены следующие критерии:

- активная разработка и поддержка со стороны разработчиков;
- прямой доступ к данным и возможность выполнять распределенные расчеты;
- актуальность фреймворка в сфере высокопроизводительных вычислений;
- степень сложности написания распределенных алгоритмов машинного обучения;
- возможность регулировать режим использования памяти;
- степень сложности внедрения кастомной логики обработки данных;
- минимизация написания кода не из коробки.

Критерии выделены на основе задач, поставленных заказчиком в рамках исполнения проекта по добавлению опции ГПУ расчетов в LightAutoML:

- должна быть опция обработки данных распределенно на одном или нескольких ГПУ;
- должны быть внедрены версии алгоритмов регрессии и бустинга деревьев, работающих на одном или нескольких ГПУ;
- не должно быть лишних копирований данных между ГПУ и ЦПУ;
- ГПУ версии функционала должны быть бесшовно добавлены в LightAutoML с сохранением логики пользования фреймворка;
- должна быть возможность обработки данных и исполнения модели на одном ГПУ в случае, когда все данные не помещаются в его память;
- по итогам проекта LightAutoML должен быть представлен на конференции (уровня) GTC.

Данный отчет является первым шагом этого проекта, и в рамках данного шага будут решаться следующие задачи:

- определить самые распространенные библиотеки с API на языке Python, которые дают доступ к выполнению расчетов на ГПУ;

- определить самые распространенные библиотеки с API на языке Python, которые позволяют писать код для распределенных систем с использованием ГПУ;
- сравнить эти библиотеки и фреймворки на основе критериев указанных выше;
- изучить готовые инструменты, предлагающие функционал и использующие ГПУ вычисления и схожие с LightAutoML, проанализировать их преимущества и недостатки, продумать их учет при выполнении данного проекта.

Доступ к ГПУ на Python

Cython и Pybind

Доступ к ГПУ на Python может быть осуществлен на разных уровнях. Самый низкий — создание библиотеки с помощью низкоуровневых API и языков и последующее их подключение к Python с помощью таких инструментов как Cython или Pybind. Низкоуровневыми API являются Vulkan, OpenGL, DirectX, OpenCL, CUDA C. Первые три — самое низкоуровневое, и используется преимущественно разработчиками компьютерной графики, в рамках данного проекта Vulkan, OpenGL и DirectX не являются практичными. OpenCL — кроссплатформенный фреймворк для написания кода на ГПУ. CUDA C является инструментом аналогичным OpenCL но оптимизированным под видеокарты от Nvidia и работающем только на них. При необходимости можно писать код для данного проекта на данном уровне при помощи CUDA C.

Разработчиками Numba был создан CUDA Array Interface, который стал стандартным инструментом для доступа к массиву данных на ГПУ от Nvidia из Python. На момент написания данного отчета данный интерфейс не поддерживается Pybind (ссылка). Это значит, что с Pybind невозможно удовлетворить критерий о возможности доступа к данным на ГПУ напрямую без копирований на ЦПУ, поэтому тестировался только Cython. Далее представлен листинг, показывающий как можно обратиться к массиву данных прямо на ГПУ (это делалось через каст указателя в тип size_t, что вообще говоря не является безопасным, но при необходимости будут исследованы более безопасные варианты).

```
cdef extern from "dev_bucket_sort.hh":
    void fill_na_median_d(double * d_v, size_t N)

cdef fillnamedian_wrapper(size_t d_v, size_t N):
    fill_na_median_d(<double*>d_v, N)

def fillnamedian(d_v, size_t N):
    vPtr = d_v.data.ptr
    fillnamedian_wrapper(vPtr, N)
```

Листинг 1: Интерфейс для вызова из Python трансформера FillnaMedian написанного на CUDA C и C++.

Numba, Cupy, Pycuda

Доступ в данным ГПУ и возможность писать ядра на ГПУ предоставляется библиотеками Numba, Cupy и Pycuda. Первые две библиотеки являются основными инструментами, с помощью которых пишется код на Python, использующий ГПУ. Pycuda менее распространена. Но все три библиотеки имеют одинаковый принцип работы с ГПУ в плане написания кастомной логики: пишется ядро на CUDA C, далее библиотека сама компилирует код, после чего пользователь имеет возможность вызова откомпилированной функции на массиве данных на ГПУ. Numba и Cupy также имеют шаблоны для некоторых распространенных операций, которые упрощают написание ядра на CUDA C (например поэлементные операции, редукция).

Также нужно отметить, что Numba и Cupy совместимы. Ядро, написанное на Numba будет работать на массивах из Cupy, так как обе библиотеки поддерживают CUDA Array Interface.

Rapids

Rapids является фреймворком, который с одной стороны находится на том же уровне по доступу к ГПУ что и Cupy, так как на на Rapids также можно писать ядра для работы с ГПУ (тут они зовутся user-defined functions), но с другой стороны имеет более высокоуровневую структуру, так как в ней присутствуют, например, реализации алгоритмов машинного обучения, таких как Regression, Lasso, RidgeRegression, ElasticNet. Это предоставляет гибкость в пользовании библиотекой, так как разработчиками сохраняется доступ к работе с ГПУ на разных уровнях.

В рамках Rapids создан тип хранения данных на ГПУ cudf.DataFrame, с интерфейсом аналогичным Pandas. Данный класс имеет свойство .values, в котором хранится массив Cupy. Это позволяет также пользоваться ядрами, написанными на Numba, Cupy, а также обертками в Cython для обработки cudf.Dataframe. Можно ознакомиться со ссылкой для более подробного изучения.

Еще одним плюсом библиотеки является наличие модуля rmm, дающего возможность разработчикам возможность настроить модель пользования памяти ГПУ во время работы программы.

Tensorflow и Pytorch

Фреймворки Tensorflow и Pytorch дают доступ к ГПУ на самом высоком уровне посредством множества готовых функций. Но также есть возможность добавления кастомной логики на CDUA C/C++ и компилирования при помощи Cython/Pybind или при помощи встроенных инструментов (Tensorflow Op).

Также оба фреймворка поддерживают CUDA Array Interface, что позволяет конвертировать соответствующие тензоры в формат Cupy без копирования данных. К слову,

популярна библиотека Dlpack, которая упрощает процесс конвертации для разработчиков.

Тесты

Библиотеки для доступа к ГПУ на Python поддерживают единый интерфейс, поэтому их можно совмещать между собой. Это большой плюс, так как выбор определенного фреймворка не влечет за собой сужение набора возможностей разработки. Можно пользоваться преимуществами каждой библиотеки.

Кстати говоря, разработчики из Nvidia планируют стандартизовать доступ к их ГПУ через Python посредством библиотеки NVRTC (ссылка). Но разработка находится только на начальных стадиях.

Для анализа производительности определенных выше инструментов были произведены тесты — расчет трансформера FillnaMedian с использованием разных фреймворков на разных данных. Результаты расчетов можно видеть на Рис. 1,2 и Таблице 1. Можно видеть, что на малых данных использование ГПУ не дает преимущества перед версиями алгоритма, написанных на ЦПУ, оно появляется лишь на больших данных. При этом все варианты, написанные на ГПУ имеют между собой незначительную разницу во времени исполнения. Таблица 1 показывает время исполнения трансформера FillnaMedian на данных в несколько колонок. Можно наблюдать заметный разрыв между Torch и Rapids версиями алгоритмов, вызванный разницей размещения массива данных в памяти (по строкам и по столбцам). Это нужно учитывать при реализации ГПУ версии данного трансформера для LightAutoML.

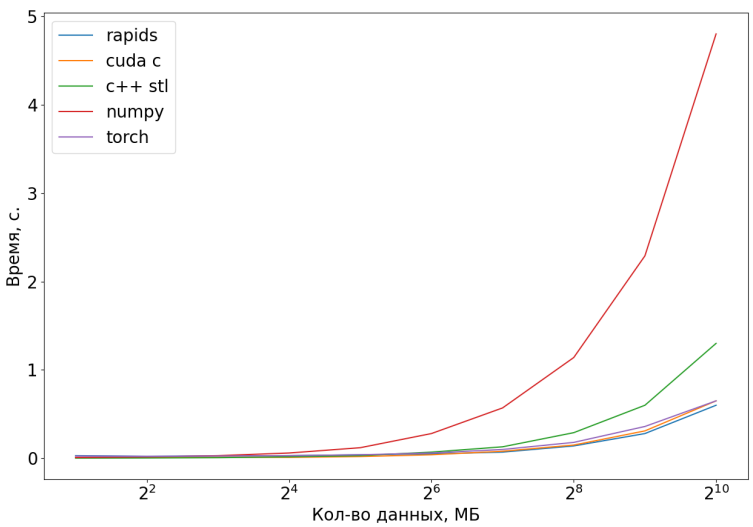

Рис. 1: Время исполнения разных версий FillnaMedian на 1 колонке данных от 2 МБ до 1 ГБ. Описание легенды на следующей картинке

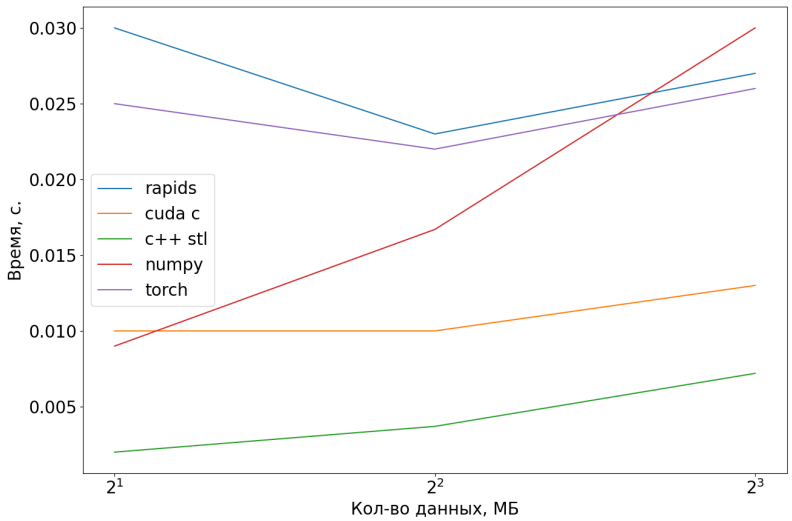

Рис. 2: Время исполнения разных версий FillnaMedian на 1 колонке данных от 2 МБ до 8 МБ. Rapids — реализация с использованием функций из фреймворка Rapids, CUDA C — реализация, написанная на CUDA C и привязанная через Cython, C++ stl — реализация, написанная на C++ и библиотеки STL и привязанная через Cython, Numpy — реализация в Numpy, Torch — реализация в Torch.

Реализация	Размеры	Вес	Время
rapids	16384 x 1	1 МБ	0.03 с
torch	16384 x 1	1 МБ	0.01 с
rapids	32768 x 32	8 МБ	0.08 с
torch	32768 x 32	8 МБ	0.02 с
rapids	65536 x 64	32 МБ	0.17 с
torch	65536 x 64	32 МБ	0.03 с
rapids	131072 x 256	256 МБ	0.74 с
torch	131072 x 256	256 МБ	0.07 с

Таблица 1: Время исполнения FillnaMedian, реализованных на Rapids и Torch на нескольких колонках данных. Зеленый цвет — реализация работает быстрее аналогичной на ЦПУ, красный цвет – медленее.

Распределенные вычисления на Python поддерживающие доступ к ГПУ

В данной секции будут рассмотренны следующие библиотеки для распределенных вычислений на нескольких ГПУ: dask, ray, pyspark.

Нужно указать, что есть еще относительно низкоуровневая библиотека mpi4py, которая поддерживает CUDA Array Interface, поэтому распределенные вычисления на ГПУ

можно делать и на ней. Но разработка на данном уровне будет избегаться из практических соображений, так как по сравнению с представленными ранее альтернативами объем работ будет много выше. Однако, для частных кейсов можно прибегать к разработке на данном уровне в случае необходимости.

Dask

Dask — изначально библиотека для распределенных вычислений только на ЦПУ на Python. Позднее добавилась возможность выполнять расчеты на ГПУ в рамках библиотеки Rapids. Проект ведется со стороны Nvidia, и модуль dask_cudf является составной частью фреймворка Rapids.

Из преимуществ данной библиотеки стоит отметить активное внедрение Dask в другие билиотеки. Например, распределенные расчеты с помощью XGBoost на ГПУ реализованы при помощи Dask (ссылка). Также разработчики LightGBM выражают заинтересованность в добавлении возможности запуска их алгоритма на dask_cudf. (ссылка).

Из недостатков можно указать, что ГПУ версия фреймворка еще не реализована полностью (отсутствуют некоторые метрики для валидации модели, также некоторые модели не работали во время тестов). Однако, как уже было сказано ранее, она активно разрабатывается.

Pyspark

Spark — очень популярный фреймворк для обработки данных, имеющий обертку на Python. Если говорить о ЦПУ версии, то по сравнению с Dask данный фреймворк более тяжеловесен и имеет больше уже готового функционала. Однако в плане расчетов на ГПУ Spark отстает от Dask. Использование ГПУ также является актуальной и активно разрабатывается (тоже командой из Nvidia). В качестве бэкэнда применяется тот же Rapids. Но на данный момент в рамках Pyspark нет возможности работать с данными на ГПУ напрямую (ссылка). Pyspark сам определяет будет ли алгоритм быстрее работать на ГПУ и при необходимости пересылает туда данные.

Также Nvidia имеет библиотеку для распределенных вычислений на ГПУ для XGBoost (ссылка).

Ray

Использование фреймворка Ray напоминает клиент-серверную архитектуру. Создается класс для хранения состояния, затем элементы класса можно параллельно менять с помощью созданных рабочих. Доступ к данным на ГПУ имеется, так как Ray не контролирует содержание классов, а лишь предлагает подход к их обработке.

Отдельно авторы библиотеки написали библиотеку для XGBoost с возможностью использования нескольких ГПУ.

В отличие от предыдущих двух фреймворков, Ray не имеет большого количества готового нужного функционала, поэтому использование этого подхода будет занимать больше времени, чем Dask или Pyspark.

Тесты

С целью проверки доступа к данным на распределенных ГПУ был написан тест, выполняющий расчеты по следующему пайплайну:

Input -> FillnaMedian -> GB Trees -> Select Features -> StdScale -> Lasso

Используемый стек: XGBoost, Catboost, Rapids, Dask. Тест показал, что Dask+Rapids действительно дают прямой доступ к данным на ГПУ без копирования данных на ЦПУ. Кроме того, расчеты выполнялись на нескольких ГПУ. Данные по времени исполнения представленны на Рис. 3-5.

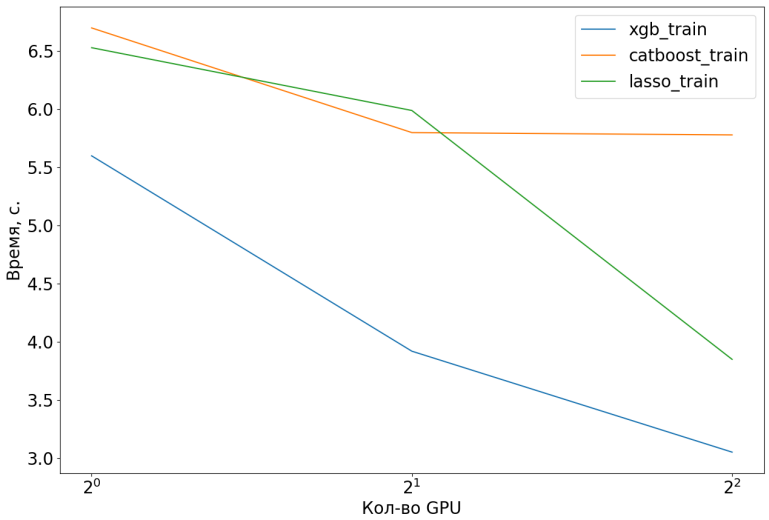

Рис. 3: Время исполнения обучения некоторых алгоритмов из пайплайна.

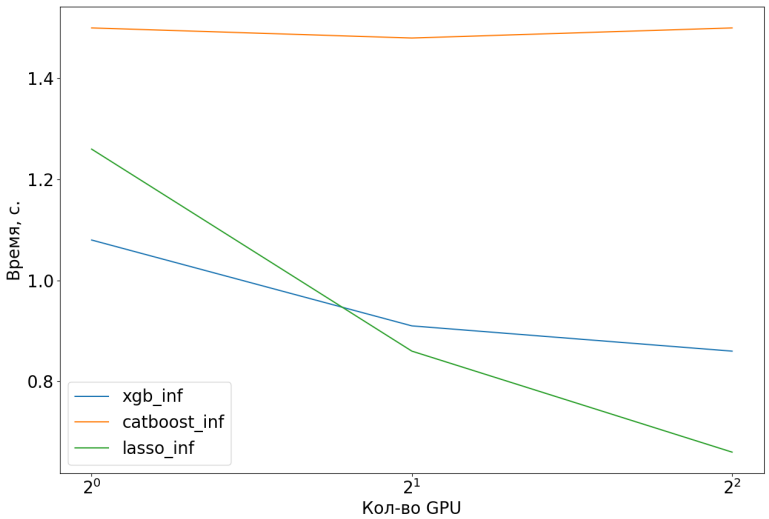

Рис. 4: Время исполнения инференса некоторых алгоритмов из пайплайна.

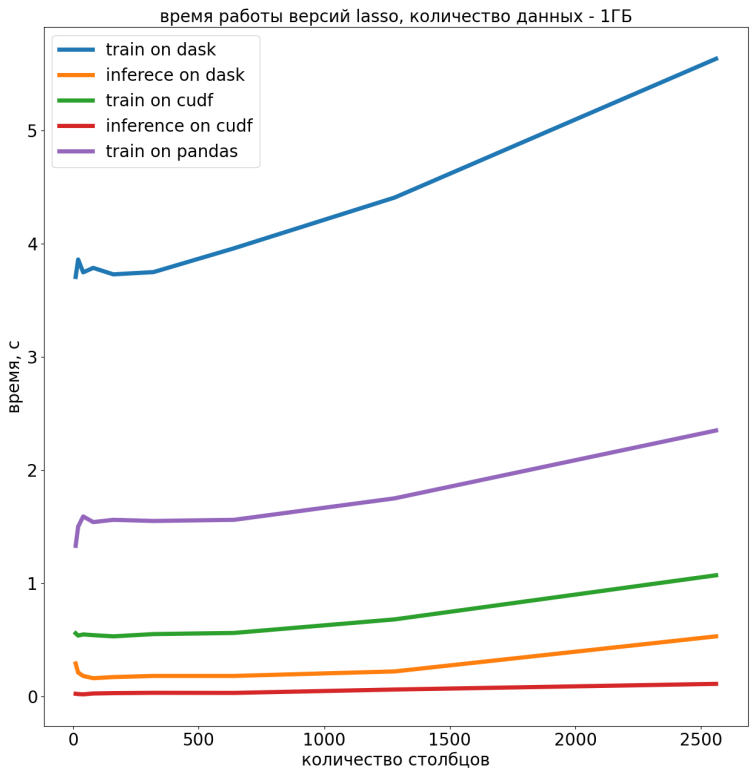

Рис. 5: Зависимость времени исполнения разных реализаций Lasso от количества фичей в модели (при одинаковом весе датасета — около 1 ГБ).

В рамках данного проекта требуется наличие возможности проведения распределенных вычислений на ГПУ для алгоритмов бустинга деревьев решений. Ниже приведена таблица с информацией о поддержке ГПУ наиболее распространенных алгоритмов:

Реализация	ГПУ/несколько ГПУ	Ray	PySpark	Dask	прямой доступ к ГПУ
XGBoost	да	да	да	да	да
LightGBM	да	нет	да	нет	нет
Catboost	да	нет	да	нет	нет
ThunderGBM	да	нет	нет	нет	нет

Таблица 2: Характеристики различных реализаций бустинга деревьев

В случае, когда доступ в распределенным ГПУ вычислениям не предоставлен из коробки, нужно будет пробовать разработать собственную распределенную реализацию.

При выполнении тестов использовались синтетические данные, а также датасет HIGGS (ссылка). Для генерирования синтетических данных написана функция **generate_data**. В качестве аргументов указывается количество строк данных, количество столбцов типа Numeric, количество столбцов типа Category, количество столбцов типа String, количество элементов датасета, являющихся NaN.

Существующие готовые реализации

H2o

Библиотека для распределенного анализа данных и применения алгоритмов машинного обучения. Поддержка ГПУ предоставлена модулем h2o4gpu. Есть реализация XGBoost (с помощью dask-cuda) и моделей регрессии (собственные реализации подключенные через cython).

Нет стабильной LightGBM, Catboost, обработки данных на ГПУ, прямого доступа к данным на ГПУ.

Modin

Библиотека для быстрой обработки данных. Имеет pandas-like интерфейс. Разрабатывается поддержка некоторых алгоритмов машинного обучения (XGBoost). Поддерживается только ЦПУ, но ведется работа по подключению возможностей ГПУ вычислений (с помощью Rapids + Ray) (ссылка).

Turicreate

Библиотека для автоматического построения моделей машинного обучения. Библиотека сама старается подобрать наиболее оптимальную модель и параметры для решения зада-

чи. Есть выборочная поддержка ГПУ через Tensorflow, но без прямого доступа к данным на ГПУ.

Vaex

Аналогичная Modin библиотека с pandas-like интерфейсом. Однако, помимо работы с данными есть алгоритмы машинного обучения, в том числе Catboost, LightGBM и XGBoost. Есть возможность писать кастомную логику на ГПУ через Numba. Прямого доступа к ГПУ нет. Судя по статье (от одного из разработчиков Vaex) исполняет базовые функции быстрее Rapids. Возможно, стоит использовать в качестве baseline при измерении производительности во время разработки.

Выводы

Ниже приведена таблица с описанием соответствия того или иного стека технологий критериям из первой секции:

Критерий	Ray + Rapids	PySpark + Rapids	Dask + Rapids
активная поддержка разработчиками	да	да	да
прямой доступ к ГПУ	да	нет	да
настройки памяти	да	да	да
от Nvidia	нет	да	да
легкость кастомизации	да	да	да
много ли есть функций из коробки	нет	да	да

Таблица 3: Распределение фреймворков по критериям

Кажется, что оптимальной основной комбинацией для разработки LightAutoML на ГПУ является Rapids + Dask. Как уже отмечалось выше, выбор определенного стека в принципе не закрывает доступ к другим фреймворкам. В случае если это целесообразно, можно дополнять код реализациями, использующими альтернативные библиотеки.

Классы для LightAutoML на ГПУ

Ниже будет предварительное описание классов. После более подробного изучения архитектуры LightAutoML могут произойти некоторые изменения.

1. Класс для работы с настройками кластера (GpuClusterConfig, например). Будет передаваться в пайплайн, и весь пайпалайн будет запускаться с этими настройками.

2. Класс для чтения данных на ГПУ (CudfReader, DaskCudfReader).

3. Классы для работы с данными на гпу. Наследуют от LAMLDataset: DaskCudfDataset, CudfDataset, CupyDataset, SparseCupyDataset. Нужно будет посмотреть есть ли необхо-

димость добавить какой-то функционал в интерфейс LAMLDataset (например to_cupy() и to_cudf()).

4. Классы для трансформеров. Наследуют от абстрактного. Имеют такие же названия как и у аналогов на ЦПУ с приставкой _gpu. Так как вызов трансформера пользователями не предусмотрен в рамках фреймворка (насколько я понял), то вызов нужной версии трансформера будет делаться проверкой типа входных данных.

5. Классы для исполнения алгоритмов на ГПУ. Так как они могут быть явно вызваны пользователями, то можно исполнение на ГПУ прописать в уже существующих классах. Тогда пользователям не нужно менять код, кроме части где вызывается reader.

Важные ссылки

- Nvidia developer blog, описывающий работу с текстовыми данными с помощью RAPIDS.
- Nvidia developer blog с туториалом по запуску XGBoost на нескольких ГПУ с помощью Dask + RAPIDS.
- Cloud.Google: как запускать Yarn на Dataproc.
- Статья от разработчиков ThunderGBM с описанием их ГПУ оптимизаций для бустинга деревьев.
- Towardsdatascience: статья с описанием готовых библиотек для обработки данных на ГПУ на Python.