Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б10-мм

Реализация Triton ядер для ускорения работы нейронных сетей

Береснёв Руслан Анатольевич

Отчёт по учебной практике
в форме «Эксперимент»

Научный руководитель:
ассистент кафедры СП, Спирин Е.С.

Санкт-Петербург
2024

Оглавление

Введение

Для генерации текстов, изображений, речи используются нейронные сети (ChatGPT [7], Midjourney [5] и т.д.). В основе их архитектуры лежат модели, обучение которых может занимать месяцы, поэтому для ускорения обучения производятся всевозможные оптимизации. Ускорение даже небольших частей модели может существенно улучшить эксперименты.

Оптимизация нейронных сетей осуществляется засчет написания вычислительных ядер, позволяющих увеличить скорость выполнения определённых операций. Вычислительное ядро [13] — это процедура, скомпилированная для ускорителей с высокой пропускной способностью (CPU, GPU), способная, например, сочетать в себе сразу несколько операций над тензорами (числами, векторами, матрицами) и оптимально выполняться.

Обычно все оптимизации производятся на графическом процессоре: блоки данных копируются с CPU на GPU, параллельнно обрабатываются несколькими потоками, а затем результаты синхронизируются и записываются обратно на CPU. Такие операции являются достаточно низкоуровневыми и выполняются с помощью специальных библиотек (например, CUDA [6] или OpenCL [3]). Для того, чтобы разработчикам не приходилось задумываться обо всех нюансах работы с GPU, появились специальные компиляторы, позволяющие производить оптимизации на видеокартах без глубоких знаний в области многопоточного программирования, синхронизации, работы с памятью и в целом работы с графическими процессорами.

Одним из инструментов для создания оптимизированных ядер является Triton [9] — open-source язык-компилятор на основе Python от OpenAI, который позволяет писать высокоэффективный код, исполняемый на GPU. Целью Triton является полная автоматизация работы с GPU таким образом, чтобы можно было сфокусироваться на написании высокоуровневых параллельных программ, практически не задумываясь об отдельных операциях, связанных с многопоточностью.

1. Постановка задачи

1.1. Цель

Целью данной учебной практики стало написание Triton-ядер для ускорения архитектуры конкретной нейронной сети, а также сравнение производительности с уже реализованными ядрами в других библиотеках.

1.2. Задачи на осенний семестр

- Описать архитектуру и механизм работы языка-компилятора Triton;
- найти библиотеки, реализующие ядра для ускорения работы нейронных сетей;
- провести обзор существующих Triton-ядер в данных библиотеках;
- описать план реализации и проведения экспериментов.

1.3. Задачи на весенний семестр

- Выбрать определенную архитектуру нейронной сети;
- провести обзор существующих оптимизаций для данной архитектуры;
- ускорить работу нейронной сети, создав для каждой ее части Triton-ядра;
- реализовать окружение для тестирования новых и уже использующихся ядер;
- сравнить производительность собственной реализации и уже готовых оптимизаций;

2. Обзор

2.1. Язык-компилятор Triton

В данном разделе рассмотрены общее представление и применение Triton, а также его технические аспекты: принцип работы, детали архитектуры, этапы компиляции в машинный код и совместимость с различными операционными системами и компонентами.

2.1.1. Краткий обзор

Triton представлен в виде устанавливаемой библиотеки для Python [10], которая является альтернативой программно-аппаратной архитектуре CUDA, позволяющей легче писать оптимизированные ядра на GPU. Triton используется для оптимизации алгебраических операций, таких как сложение векторов и перемножение матриц, а также для оптимизации архитектур DNN — например, ускорения работы функций активации.

2.1.2. Принцип работы

Вычислительные ядра в Triton определяются как декорированные Python-функции и помечаются с помощью декоратора @triton.jit. Ядро с разными входными данными запускается из отдельной функции, которая распараллеливает выполнение программы с помощью 3D SPMD-сетки запуска (Single Program Multiple Data), состоящей из блоков — каждый блок создает «экземпляр» ядра, который обрабатывает свой небольшой подмассив (тайл) в исходном тензоре. Размерности тайла должны быть степенями двойки (32, 64, 128 и т.д.).

В Triton каждый блок в сетке запуска ассоциирован ровно с одним потоком, выполняющим обработку своего тайла. Это позволяет достичь автоматического параллелизма при выполнении программы: не нужно задумываться о синхронизации общей памяти, межпоточном взаимодействии и т.д., что упрощает разработку сложных программ на графическом процессоре. В CUDA, например, используется тот же ме-

ханизм параллельного выполнения ядер, однако в каждом блоке может быть запущено одновременно до 1024 потоков (см. рис. 1).

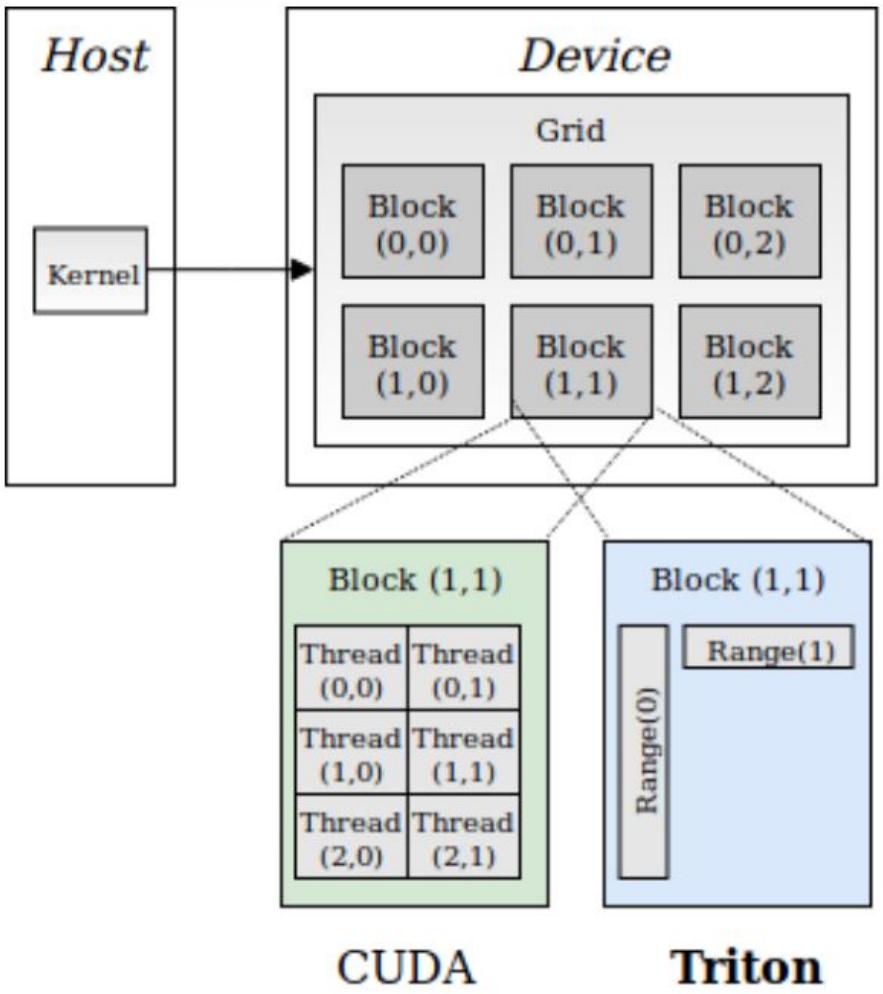

Рис. 1: Разница между моделями выполнения ядер на Triton и CUDA

2.1.3. Архитектура

Архитектура Triton является модульной и состоит из трех основных частей, что позволяет улучшить производительность:

Python

Высокоуровневое представление на языке Python, с помощью которого реализуются все операции для работы с тензорами при написании вычислительных ядер, а также происходит параллельный запуск ядер.

Triton-IR

Промежуточное представление для Triton, основанное на LLVM-IR — абстрактном типизированном представлении LLVM [4] в виде

байт-кода для оптимизации и генерации кода для различных архитектур. Triton-IR состоит из независимо компилируемых модулей, которые включают в себя:

- Глобальные переменные;
- константы;
- функции.

Стоит отметить, что функции состоят из имени, возвращаемого типа и списка аргументов, за которыми следует тело функции, представляемое списком базовых блоков, образующих граф потока управления программы. Каждый такой блок должен быть простой последовательностью кода, заканчивающейся условным/безусловным переходом или инструкцией завершения функции.

В Triton-IR используется форма Static Single Assignment (SSA) [16], в которой каждая переменная в модуле должна быть объявлена до того, как будет использована, а также содержать единожды присвоенное значение. Таким образом, если в переменной меняется значение, то создается версия данной переменной с новым значением (например, для x это x_1, x_2, ...).

Triton-JIT

JIT (Just-In-Time) компиляция [14] — это динамическая компиляция, которая в отличие от статической производится во время исполнения программы, а не перед ее запуском. JIT позволяет повысить производительность интерпретируемых программ: пока интерпретатор построчно выполняет программу, JIT-компилятор преобразует часто используемые функции из промежуточного представления в машинный код, оптимизированный для данной аппаратной архитектуры. Таким образом, JIT-компиляция совмещает в себе скорость статической компиляции и гибкость интерпретации, однако требует выделения дополнительной памяти.

В Triton JIT-компилятор производит компиляцию и параллельный запуск на GPU ядер, помеченных декоратором @triton.jit. Triton-JIT ис-

пользуется для упрощения и компиляции представления Triton-IR в эффективный машинный код. Процесс компиляции состоит из машинно-независимых и машинно-зависимых проходов, которые будут рассмотрены далее.

Машинно-независимые проходы включают в себя такие оптимизации, как pre-fetching (предварительная загрузка инструкций или данных в память процессора, чтобы избежать задержек во время выполнения программы) и peephole optimizations (замена небольшого набора сгенерированных компилятором инструкций эквивалентным набором, имеющим более высокую производительность);

Машинно-зависимые проходы в Triton-JIT осуществляются засчет следующих оптимизационных стратегий:

- **Иерархическое разбиение тайлов** — обычные тайлы разбиваются на микро-тайлы и нано-тайлы. Это позволяет обрабатывать большой тайл частями, соответствуя аппаратным вычислительным возможностям, а также как можно более плотному распределению памяти. Микро-тайлам соответствуют процессорные единицы SIMD, а нано-тайлам — операции LD/ST внутри них (Load-Store) (см. рис. 2)

- **Оптимизация обращений к памяти** — обращения к памяти являются объединенными, если потоки, исполняющие одинаковые инструкции, одновременно извлекают из памяти рядом находящиеся данные. Так как обычно из DRAM данные считываются большими блоками, то в плане оптимизации количества обращений к памяти важно, чтобы эти обращения были как можно чаще объединенными, и не происходило так, что некоторая часть загруженных из памяти данных не используется ни одним потоком. Так как выполняемые программы в промежуточном представлении Triton являются однопоточными, и таким образом автоматически распараллеленными, это позволяет компилятору по возможности организовать расположение потоков так, чтобы избежать необъединенных обращений к памяти.

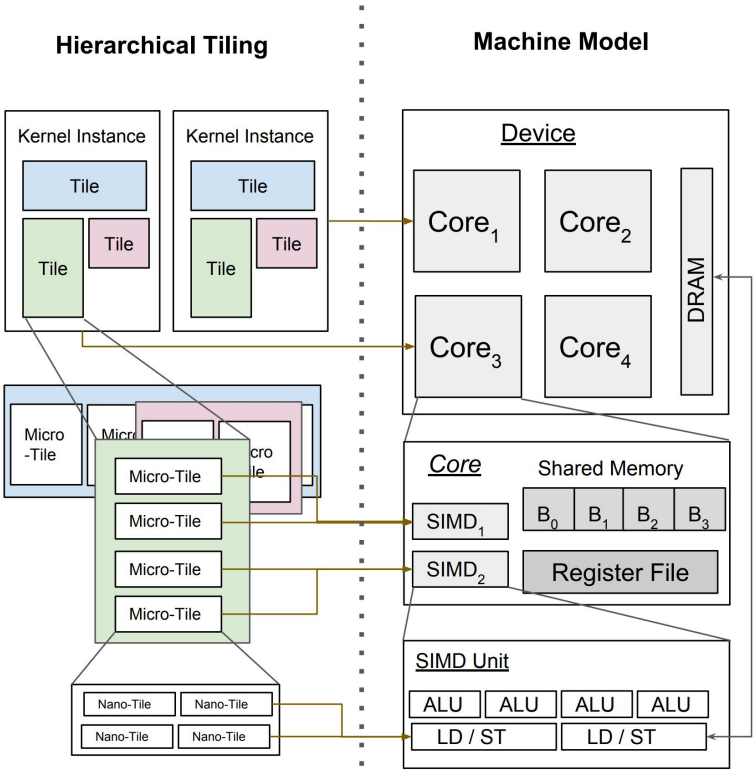

Рис. 2: Модель иерархического разбиения тайлов

- **Распределение общей памяти** — механизм данной итерации заключается в том, чтобы определить, где и когда тайл должен быть временно сохранен в общую память с быстрым доступом. Сколько памяти нужно определенной переменной и на какой промежуток времени ее выделить, определяет алгоритм диапазона жизни (см. рис. 3). Благодаря динамическому распределению общей памяти с быстрым доступом между переменными могут быть оптимизированы операции с высокой степенью арифметических вычислений (например, перемножение матриц).

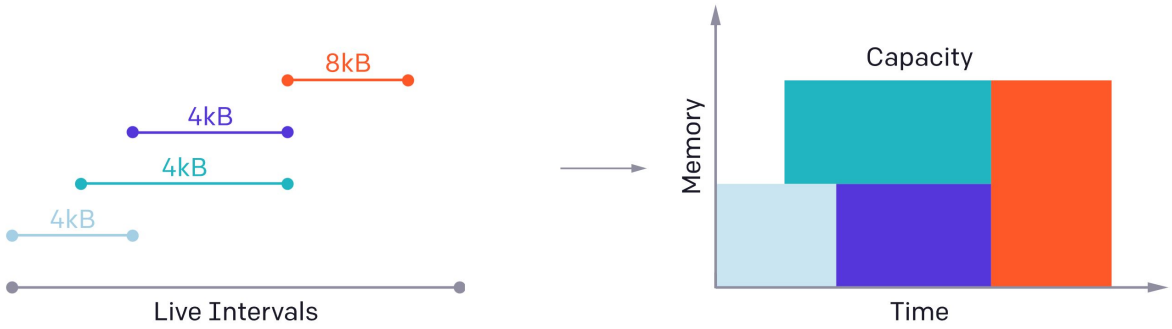

Рис. 3: Динамическое распределение общей памяти между переменными с помощью алгоритма диапазона жизни

2.1.4. Этапы компиляции

Компиляция каждой высокоуровневой Python-функции, помеченной декоратором @triton.jit, в низкоуровневый машинный код, исполняемый на GPU, происходит в несколько этапов [8] (см. рис. 4).

Рис. 4: Схема этапов компиляции функции на Python в машинное представление

Сначала создается абстрактное синтаксическое дерево исходной функции, происходит его обход и генерация представления Triton-IR с использованием формы SSA. Затем код в Triton-IR упрощается и оптимизируется засчёт машинно-зависимых и машинно-независимых проходов, и преобразутеся JIT-компилятором Triton в промежуточное представление LLVM (LLVM-IR). Из промежуточного представления библиотека LLVM конвертирует код в PTX [15] — ассемблерный язык, разработанный NVIDIA для программирования графических процессоров, которые поддерживают архитектуру CUDA. Наконец, код PTX преобразуется компилятором CUDA в исполняемый на GPU машинный код.

2.1.5. Совместимость

Triton разработан для исполнения на GPU от NVIDIA. На данный момент возможность работы с CPU или с другими GPU (например от

AMD), не поддерживается, однако находится на стадии разработки.

Также для установки Triton и написания ядер на нем нужна операционная система Linux; с Windows и Mac без дополнительных утилит библиотека triton не совместима.

2.2. Библиотеки с реализацией собственных Triton-ядер

В данном разделе рассмотрены open-source библиотеки, которые реализуют собственные Triton-ядра для оптимизации различных алгебраических операций и архитектур нейронных сетей. Цель обзора — понять, какие библиотеки являются общими и могут быть использованы для ускорения модулей различных нейронных сетей, а какие специально созданы для оптимизации конкретных архитектур.

2.2.1. PyTorch [11]

Библиотека PyTorch используется для машинного обучения на языке программирования Python и применяется в областях компьютерного зрения, обработки естественного языка, генерации текста и изображений. PyTorch предоставляет две высокоуровневые особенности:

- Тензорные вычисления (например, как в NumPy) с сильным ускорением благодаря исполнению на графическом процессоре;
- работу с глубокими нейронными сетями (DNN), использующими автоматическое дифференцирование (Autograd system).

В PyTorch реализованы Triton-ядра для следующих операций:

- Нахождение максимума и минимума из двух чисел;
- генерация случайного целого числа;
- нахождение значения в списке с помощью алгоритма бинарного поиска;

- перемножение матриц;
- реализация функции активации softmax.

PyTorch является общеприменимой библиотекой и позволяет как производить сложные вычисления над тензорами с использованием математических функций, так и создавать и обучать различные нейронные сети.

2.2.2. Kernl [12]

Библиотека Kernl разработана для ускорения моделей трансформеров, созданных на PyTorch, с помощью выполнения на GPU. Kernl является механизмом логических выводов (inference engine) и сопоставляет факты и данные из базы знаний с некоторыми правилами, чтобы сделать логические выводы. Данная библиотека используется для обработки естественного языка и генерации текстов.

В Kernel реализованы следующие ядра Triton:

- Для механизма Attention;
- для скалярного произведения векторов и транспонирования матриц;
- для таких функций активации, как: гиперболический тангенс, ReLU, GELU;
- для перемножения матриц;
- для алгоритма Layer Normalization.

2.2.3. Flash-attn [1]

Flash-attn [2] — библиотека на Python, которая реализует улучшенный алгоритм для стандартного механизма внимания засчет уменьшения чтений/записей памяти между GPU HBM (High Bandwidth Memory) и статической оперативной памятью SRAM. Оптимизация обращений к памяти важна, так как механизм внимания квадратично по

времени и используемой памяти зависит от длины входной последовательности. Механизм внимания является ключевым элементом в архитектуре таких глубоких нейронных сетей как трансформеры, которые используются в обработке естественного языка (NLP) и классификации изображений.

В Flash-attn Triton-ядра используются для реализации самого механизма Flash Attention, а также для вычисления:

- Произведения матриц;
- перекрестной энтропии;
- функций гиперболического тангенса и гиперболического косинуса;
- функций активации ReLU, Leaky ReLU, GELU;
- алгоритма Layer Normalization.

Библиотека flash-attn заточена под реализацию механизма Flash Attention и его расширения Block-Sparse Flash Attention, и была протестирована на двух датасетах, содержащих длинные текстовые документы. Были получены следующие результаты по скорости выполнения и использованию памяти по сравнению с реализациями механизма внимания в некоторых других библиотеках (см. рис. 5):

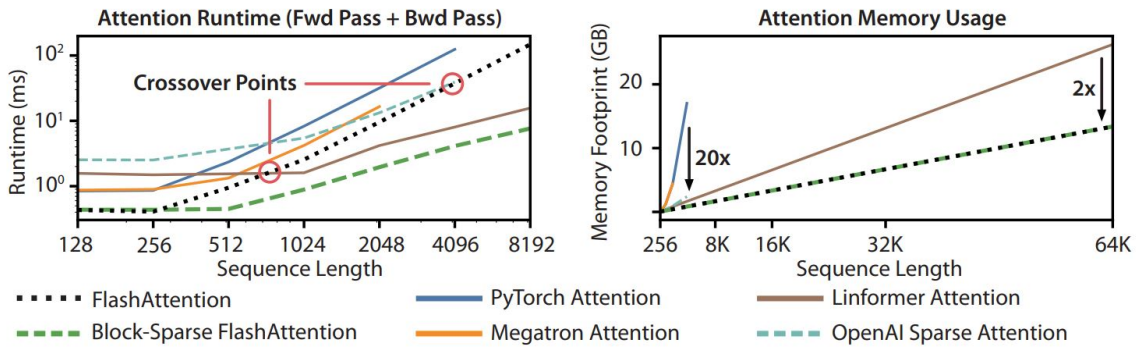

Рис. 5: Сравнение Flash Attention и Block-Sparse Flash Attention с некоторыми другими реализациями механизма внимания

2.2.4. xFormers [17]

xFormers так же применяется для ускорения моделей трансформеров и используется в областях компьютерного зрения и обработки естественного языка. Данная библиотека состоит из модулей, которые могут быть расширены для реализации конкретных задач на трансформерах.

В xFormers реализованы Triton-ядра для различных механизмов внимания:

- Memory-efficient exact attention;
- sparse attention;
- block-sparse attention.

Также используются ядра для функций активации fused softmax и fused SwiGLU, алгоритмов fused layer normalization и fused dropout.

3. Реализация

В осеннем семестре был проведен обзор языка-компилятора Triton, а также open-source библиотек с Triton-ядрами; реализация планируется на весенний семестр.

В весеннем семестре планируется выбрать определенную архитектуру нейронных сетей и попытаться ускорить ее производительность засчет эффективной реализации на GPU. Это может быть сделано с помощью fused-операторов (сочетание нескольких похожих операций в одной с целью ускорения работы и минимизации обращений к памяти) и оптимизированных вычислений, использующих Triton-ядра.

Планируется разбить архитектуру на несколько частей и для каждой написать по своему Triton-ядру, оптимизирующему определенную функциональность.

4. Эксперимент

В весеннем семестре планируется провести эксперименты и сравнить производительность реализованных Triton-ядер с аналогичными операциями из библиотек с той же функциональностью: одной из таких библиотек может быть PyTorch.

В библиотеке triton.testing есть инструменты для замеров длительности работы функций, а также инструменты для построения графиков, сравнивающих различные реализации. Данные инструменты могут быть использованы для визуализации результатов.

Заключение

В ходе данной учебной практики были получены следующие результаты:

- Описаны архитектура и механизм работы языка-компилятора Triton;
- найдены библиотеки, реализующие ядра для ускорения работы нейронных сетей;
- проведен обзор существующих Triton-ядер в данных библиотеках;
- описан план реализации и проведения экспериментов.

Список литературы

[1] Dao-AILab. FlashAttention repository on GitHub. — URL: https://github.com/Dao-AILab/flash-attention (дата обращения: 28 декабря 2023 г.).

[2] FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness / Tri Dao, Daniel Y. Fu, Stefano Ermon et al. — URL: https://arxiv.org/pdf/2205.14135.pdf (дата обращения: 29 декабря 2023 г.).

[3] Group Khronos. Open standard for parallel programming of heterogeneous systems. — URL: https://www.khronos.org/api/opencl (дата обращения: 18 декабря 2023 г.).

[4] LLVM. The LLVM Compiler Infrastructure. — URL: https://llvm.org/ (дата обращения: 26 декабря 2023 г.).

[5] Midjourney. Making Images with Midjourney. — URL: https://docs.midjourney.com/docs/quick-start (дата обращения: 17 декабря 2023 г.).

[6] Nvidia. CUDA Toolkit - Free Tools and Trainings for Developers. — URL: https://developer.nvidia.com/cuda-toolkit (дата обращения: 18 декабря 2023 г.).

[7] OpenAI. Introducing ChatGPT. — URL: https://openai.com/blog/chatgpt (дата обращения: 17 декабря 2023 г.).

[8] OpenAI. Introducing Triton: Open-source GPU programming for neural networks. — URL: https://openai.com/research/triton (дата обращения: 28 декабря 2023 г.).

[9] OpenAI. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. — URL: https://www.eecs.harvard.edu/~htk/publication/2019-mapl-tillet-kung-cox.pdf (дата обращения: 18 декабря 2023 г.).

[10] OpenAI. Triton repository on GitHub. — URL: https://github.com/openai/triton (дата обращения: 18 декабря 2023 г.).

[11] PyTorch. PyTorch repository on GitHub. — URL: https://github.com/pytorch/pytorch (дата обращения: 28 декабря 2023 г.).

[12] Sarrut Lefebvre. Kernl repository on GitHub. — URL: https://github.com/ELS-RD/kernl (дата обращения: 28 декабря 2023 г.).

[13] Wikipedia. Compute kernel. — URL: https://en.wikipedia.org/wiki/Compute_kernel (дата обращения: 17 декабря 2023 г.).

[14] Wikipedia. Just-in-time compilation. — URL: https://en.wikipedia.org/wiki/Just-in-time_compilation (дата обращения: 28 декабря 2023 г.).

[15] Wikipedia. Parallel Thread Execution. — URL: https://en.wikipedia.org/wiki/Parallel_Thread_Execution (дата обращения: 28 декабря 2023 г.).

[16] Wikipedia. Static single-assignment form. — URL: https://en.wikipedia.org/wiki/Static_single-assignment_form (дата обращения: 26 декабря 2023 г.).

[17] Lefaudeux Benjamin, Massa Francisco, Liskovich Diana et al. xFormers: A modular and hackable Transformer modelling library. — https://github.com/facebookresearch/xformers. — 2022.