DOI: 10.15514/ISPRAS-2026-38(5)-12

Эффективное масштабирование графовых нейронных сетей за счёт реализации алгоритмов, учитывающих устройство графических ускорителей

Ф.С. Великонивцев, ORCID: 0000-0002-1830-8340 <fvelikonivtsev@hse.ru>

Национальный Исследовательский Университет Высшая Школа Экономики,
Россия, 109028, г. Москва, ул. Покровский Бульвар, д. 11.

Аннотация. Производительность графовых нейронных сетей (GNN) ограничена разреженным нерегулярным доступом к памяти. Популярные библиотеки, такие как DGL и PyTorch Geometric, поддерживают обобщённую передачу сообщений, однако сложные слои нередко материализуют промежуточные тензоры по рёбрам, увеличивая трафик памяти и ограничивая масштабируемость на больших графах. Мы применяем подход, ориентированный на операции ввода-вывода и арифметическую интенсивность, и показываем, что широко используемые слои классифицируются на три семейства: свёртки на основе умножения разреженной матрицы на плотную (Sparse-Dense Matrix Multiplication, SpMM), агрегации на основе редукции и слои на основе внимания (GATv2/граф-трансформер). Для каждого семейства мы разрабатываем ядра для GPU, снижающие перемещение данных, улучшающие локальность и остающиеся устойчивыми на реалистичных графах. Мы также исследуем перестановку вершин графа и обнаруживаем, что её влияние зависит от отображения ядра: она устойчивее помогает ядрам с параллелизмом по соседям (в которых доминирует gather), чем проектам с параллелизмом по признакам. Эмпирически наши слитые ядра внимания достигают ускорения до 3,9 раз для граф-трансформера (медиана 1,6 раз), а варианты с тензорными ядрами Tensor Cores (блочно-разреженные) – до 7,3x на локально плотных графах; для слоя GATv2 мы достигаем ускорения до 8,5 раз (медиана 2,0 раз) при снижении пикового потребления памяти до 76 раз (медиана 6 раз). Наши ядра редукции с учётом степеней достигают ускорения до 10 раз (медиана 2,6 раз). Для слоёв на основе SpMM корректно кэшированный cuSPARSE (библиотека NVIDIA для разреженной линейной алгебры) достигает ускорения до 8 раз над DGL (самой популярной библиотекой для графового машинного обучения Deep Graph Library) и превосходит оцениваемые пользовательские базовые решения в большинстве оценок. Мы выпускаем наши реализации как готовые замены для поддержки воспроизводимого, учитывающего оборудование ускорения GNN.

Ключевые слова: графовые нейронные сети; ядра, учитывающие ввод-вывод; ускорение на GPU; умножение разреженной матрицы на плотную; слои на основе внимания; слияние ядер; ограниченные памятью вычисления; арифметическая интенсивность; перестановка вершин графа.

Для цитирования: Великонивцев Ф.С. Эффективное масштабирование графовых нейронных сетей за счёт реализации алгоритмов, учитывающих устройство графических ускорителей. Труды ИСП РАН, 2026, том 38, вып. 5, с. 195-216. DOI: 10.15514/ISPRAS-2026-38(5)-12

On Efficient Scaling of GNNs via IO-Aware Layers Implementations

F.S. Velikonivtsev, ORCID: 0000-0002-1830-8340 <fvelikonivtsev@hse.ru>

National Research Institute Higher School of Economics,
10, Pokrovsky Bulvar, Moscow, 109028, Russia.

Abstract. Graph Neural Networks (GNNs) are bottlenecked by sparse, irregular memory access. Popular frameworks such as DGL and PyTorch Geometric support general message passing, but complex layers often materialize edge-wise intermediates, increasing memory traffic and limiting scalability on large graphs. We take an I/O- and arithmetic-intensity–centric view and show that widely used layers fall into three kernel families: SpMM-based convolutions, reduction-based aggregations, and attention-based layers (GATv2/Graph Transformer). For each family, we develop GPU kernels that reduce data movement, improve locality, and remain robust across realistic graphs. We also study graph reordering and find that its impact depends on the kernel mapping: it benefits neighbor-parallel (gather-dominated) kernels more consistently than feature-parallel designs. Empirically, our fused attention kernels reach up to 3.9 times speedup for Graph Transformer (median 1.6 times), with Tensor Core (block-sparse) variants up to 7.3 times on locally dense graphs; for GATv2 we reach up to 8.5 times speedup (median 2.0 times) while reducing peak memory by up to 76 times (median 6 times). Our degree-aware reduction kernels achieve up to 10 times speedup (median 2.6 times). For SpMM-based layers, properly cached cuSPARSE achieves up to 8 times speedup over DGL and outperforms evaluated custom baselines in the majority of evaluations. We release our implementations as drop-in replacements to support reproducible, hardware-aware GNN acceleration.

Keywords: graph neural networks; IO-aware kernels; GPU acceleration; sparse matrix multiplication; attention layers; kernel fusion; memory-bound workloads; arithmetic intensity; graph reordering.

For citation: Velikonivtsev F.S. On Efficient Scaling of GNNs via IO-Aware Layers Implementations. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 5, pp. 195-216. DOI: 10.15514/ISPRAS-2026-38(5)-12

1. Введение

Графовые нейронные сети (GNN) – один из основных инструментов обучения на реляционных и неструктурированных данных, находящий применение в рекомендательных системах и выявлении мошенничества, пространственно-временно́м прогнозировании, научном моделировании и многих других областях. Несмотря на быстрый прогресс в проектировании моделей, производительность обучения и вывода GNN на современных GPU нередко отстаёт от плотных нагрузок глубокого обучения. Ключевая причина в том, что доминирующие операторы GNN разреженны и нерегулярны: агрегация по соседям и внимание обходят рёбра графа с зависящими от данных, инвариантными к перестановкам шаблонами доступа, что приводит к низкой арифметической интенсивности и ограниченной локальности.

Этот разрыв усиливается аппаратными тенденциями. У современных GPU наблюдается всё возрастающее расхождение между пиковой пропускной способностью вычислений и пропускной способностью памяти HBM (High-Bandwidth Memory, основная память GPU), что переводит всё больше ядер в режим, ограниченный памятью, если только они не выполняют значительный объём вычислений на каждый перемещённый байт. В терминах модели roofline точка гребня (арифметическая интенсивность, необходимая для перехода в режим, ограниченный вычислениями) смещается к бóльшим значениям. Как следствие, одно лишь сокращение числа операций (FLOPs) зачастую не приводит к ускорению по фактическому времени. Это несоответствие хорошо задокументировано для внимания в трансформерах, где алгоритмы, учитывающие ввод-вывод, явно сокращающие трафик по шине памяти, дают значительный практический выигрыш. Мы утверждаем, что аналогичный недостающий принцип препятствует масштабируемому ускорению GNN: многие реализации отдают приоритет универсальности операторов, недооптимизируя перемещение данных и

материализацию промежуточных результатов, из-за чего производительность всё сильнее определяется трафиком памяти.

Существующие программные стеки для GNN, как правило, следуют одному из двух подходов. Библиотеки, такие как Deep Graph Library (DGL) и PyTorch Geometric, предоставляют универсальный интерфейс передачи сообщений и оптимизированные разреженные примитивы, однако типовые слои по-прежнему могут материализовать большие промежуточные тензоры по рёбрам, увеличивая как трафик памяти, так и пиковый объём активаций. На другом полюсе – специализированные реализации операторов, которые могут требовать значительной инженерной проработки для каждого слоя и не всегда стабильно выигрывают от новых аппаратных возможностей на разнообразных графах и при различных размерах признаков.

Недавние работы устраняют эти проблемы за счёт адаптивных сред выполнения и планирования, более эффективных реализаций отдельных слоёв, а также форматов структурированной разреженности, отображающих разреженные нагрузки на тензорные ядра Tensor Cores. Тем не менее эти подходы нередко нацелены лишь на подмножество операторов, требуют нетривиального препроцессинга или пользовательских форматов и могут быть чувствительны к структуре графа и распределению степеней.

В данной работе вычисления GNN рассматриваются с точки зрения операций ввода-вывода и арифметической интенсивности, и показывается, что широко используемые слои группируются в три семейства ядер с различным аппаратным поведением:

(i) свёртки на основе умножения разреженной матрицы на плотную (SpMM) (например, GCN/GraphConv),

(ii) агрегации на основе редукции (например, min/max и связанные сегментные редукции),

(iii) сложные вычислительные конвейеры, такие как слои на основе внимания (например, GATv2 и граф-трансформеры).

Для каждого семейства анализируется поведение памяти типовых реализаций, а предложенные проекты валидируются путём профилирования на реалистичных графах с различной плотностью и распределением степеней.

Это даёт три практических результата. Во-первых, наши эксперименты показывают, что для свёрток на основе SpMM вендорские примитивы остаются весьма конкурентоспособными: cuSPARSE нередко соответствует библиотечным базовым решениям или превосходит их, а кэширование препроцессинга графа и транспонированной матрицы смежности улучшает сквозную задержку обратного прохода. Во-вторых, эмпирически обнаруживается, что для слоёв на основе редукции плиточное разбиение с учётом степеней вершин принципиально важно при тяжёлых хвостах распределения степеней: оно повышает пропускную способность на вершинах высокой степени, не увеличивая объём памяти, тогда как для вершин низкой степени достаточно простой коалесцированной компоновки. В-третьих, для слоёв на основе внимания предлагаются вдохновлённые FlashAttention ядра, учитывающие ввод-вывод, которые сливают вычисление по соседям, избегают материализации промежуточных тензоров по рёбрам и опционально задействуют тензорные ядра Tensor Cores через блочно-разреженные компоновки.

Кроме того, пересматривается роль перестановки вершин графа и показывается, что её эффективность зависит от стратегии параллелизации ядра и распределения степеней: проекты с параллелизмом по соседям выигрывают больше на графах высокой степени, тогда как на графах низкой степени (например, дорожных сетях) влияние ограничено, а производительность определяется накладными расходами на каждую вершину. Наконец, выпускается реализация с открытым исходным кодом с минимальными зависимостями (CUDA, PyTorch, cuSPARSE) и интерфейсом PyTorch в качестве готовой замены широко используемых слоёв GNN.

2. Предварительные сведения

2.1 Обозначения и парадигма передачи сообщений в графе

Рассматривается однородный граф $G = (\mathcal{V}, \mathcal{E})$ с $|\mathcal{V}| = N$ вершинами и $|\mathcal{E}| = M$ ориентированными рёбрами. Признаки вершин на слое ℓ образуют матрицу $\boldsymbol{H}^{(\ell)} \in \mathbb{R}^{N \times D_\ell}$, с $\boldsymbol{H}^{(0)} = \boldsymbol{X}$. В основном используется формат сжатого разреженного хранения по строкам (Compressed Sparse Row, CSR). В оставшейся части работы индекс слоя ℓ нередко опускается, когда это ясно из контекста, поскольку анализируется один представительный слой, а аргументы на уровне ядер одинаковы для всех слоёв.

Большинство слоёв GNN выражаются через шаблон нейронной сети с передачей сообщений (Message Passing Neural Networks, MPNN):

$$\boldsymbol{m_i^{(\ell)}} = \mathbf{AGG}^{(\ell)}(\{\,\mathbf{MSG}^{(\ell)}(\boldsymbol{h_i^{(\ell)}}, \boldsymbol{h_j^{(\ell)}}, \boldsymbol{e_{ij}}) : \boldsymbol{j} \in \boldsymbol{\mathcal{N}(i)}\,\}),$$

$$h_i^{(\ell+1)} = \mathrm{UPDATE}^{(\ell)}(h_i^{(\ell)}, m_i^{(\ell)}),$$

где AGG – произвольная перестановочно-инвариантная функция (например, min/max/среднее и так далее), а MSG и UPDATE – произвольные обучаемые функции [1-2]. С аппаратной точки зрения доминирующая стоимость обычно приходится на агрегацию по окрестности над рёбрами, что порождает разреженные и нерегулярные шаблоны доступа к памяти [3-4].

2.2 Три семейства операторов слоёв в графовых нейросетях

Мы рассматриваем три семейства операторов, охватывающих широко используемые слои GNN и обнажающих различные аппаратные узкие места.

(i) Свёртки на основе SpMM. Эти слои выражаются как умножение разреженной матрицы на плотную (SpMM) между разреженной (как правило, нормированной) матрицей смежности $\widetilde{\boldsymbol{A}}$ и плотной матрицей $\boldsymbol{C}$, где $\boldsymbol{C} = \boldsymbol{H}^{(\ell)}\boldsymbol{W}^{(\ell)}$[1]:

$$\boldsymbol{H^{(\ell+1)}} = \boldsymbol{\sigma}(\widetilde{\boldsymbol{A}}\,\boldsymbol{C}) = \boldsymbol{\sigma}\big(\widetilde{\boldsymbol{A}}\,\boldsymbol{H}^{(\ell)}\boldsymbol{W}^{(\ell)}\big).$$

Например, в GCN (графовой свёрточной сети) $\tilde{a}_{ij} = 1/\sqrt{\deg(i)\deg(j)}$ [5].

(ii) Агрегации на основе редукции. Некоторые слои (пулинг, «сегментные» редукции) вычисляют порёберные значения с последующей редукцией по соседям[2]:

$$\boldsymbol{h_i^{(\ell+1)}} = \mathbf{UPDATE}^{(\ell)}\left(\boldsymbol{h_i^{(\ell)}}, \mathbf{REDUCE}_{\boldsymbol{j} \in \boldsymbol{\mathcal{N}(i)}} \boldsymbol{h_j^{(\ell)}}\right),$$

где REDUCE – как правило, max, min [6]. В отличие от операторов на основе SpMM, данные операторы реализуются с помощью примитивов scatter/gather и специализированных ядер редукции.

(iii) Слои на основе внимания. Операторы этого семейства используют механизм внимания для взвешивания сообщений от соседей на фазе обновления (для простоты обозначений индекс головы опущен):

$$\boldsymbol{h_i^{(\ell+1)}} = \sum_{\boldsymbol{j} \in \boldsymbol{\mathcal{N}(i)} \cup \{\boldsymbol{i}\}} \boldsymbol{\alpha_{ij}}\, \boldsymbol{v_j},$$

$$\alpha_{ij} = \mathrm{softmax}_{j \in \mathcal{N}(i) \cup \{i\}}(e_{ij}), e_{ij} = \mathrm{MSG}^{(\ell)}(i, j).$$

[1] Отметим, что $\boldsymbol{C} = \boldsymbol{H}^{(\ell)}\boldsymbol{W}^{(\ell)}$ — стандартная операция матричного умножения двух плотных матриц и поэтому её вычисление не является предметом оптимизации в данной работе.

[2] В данной группе рассматриваются операторы, которые *нельзя* выразить как SpMM, в отличие от, например, порёберного вычисления среднего или суммы.

Члены семейства отличаются главным образом параметризацией предсофтмаксных оценок e_{ij} и векторов значений v_j. Типичные примеры: GAT, GATv2 и слои граф-трансформера [7-10]. Для GATv2 оценки вычисляются как $e_{ij} = a^{\top}\text{LeakyReLU}(\boldsymbol{W}_{\ell} h_i + \boldsymbol{W}_r h_j)$, а значения – как $v_j = \boldsymbol{W}_r h_j$. Слои граф-трансформера используют точечно-произведённое внимание [11]:

$$\boldsymbol{q_i} = \boldsymbol{W_Q h_i}, \quad \boldsymbol{k_j} = \boldsymbol{W_K h_j}, \quad \boldsymbol{v_j} = \boldsymbol{W_V h_j}, \quad \boldsymbol{e_{ij}} = \frac{\boldsymbol{q_i^{\top} k_j}}{\sqrt{\boldsymbol{D}}},$$

где $\alpha_{ij} = \text{softmax}_{j \in \mathcal{N}(i)}(e_{ij})$ и $h_i^{(\ell+1)} = \sum_{j \in \mathcal{N}(i)} \alpha_{ij}\, v_j$. Ряд архитектурных вариантов был предложен: Graphormer [12] дополняет механизм кодированиями центральности степеней и длин кратчайших путей; GPS (General, Powerful, Scalable) [13] сочетает локальный слой MPNN с глобальным слоем само-внимания. Однако глобальное внимание имеет сложность $\mathcal{O}(N^2)$, что мотивирует использование локального варианта для соседей в данной работе, сохраняющего разреженность матрицы смежности.

2.3 Характеристики вычислителя и вопросы производительности

Понимание производительности трёх описанных семейств операторов требует знания релевантных концепций GPU. Основное внимание уделяется NVIDIA GPU, как доминирующей платформе для обучения и вывода современных нейросетей, в том числе и графовых.

Модель выполнения CUDA и иерархия памяти GPU. Программа GPU называется ядром: она запускается с CPU и выполняется параллельно многими потоками на GPU. Потоки группируются в блоки потоков, каждый из которых планируется на потоковый мультипроцессор (SM) – базовую единицу выполнения и вычислений GPU. Внутри блока потоков каждые 32 последовательных потока образуют группу потоков, выполняющуюся синхронно в модели SIMT (Single Instruction, Multiple Threads – "одна инструкция, множество потоков"): все потоки этой группы выполняют одну и ту же инструкцию на каждом такте. Потоки одного блока могут общаться и совместно использовать промежуточные результаты через быструю разделяемую память на кристалле (SMEM), недоступную между блоками.

Когда потоки одной группы потоков идут по разным ветвям кода (например, из-за зависящих от данных условных переходов), группа потоков сериализует обе ветви – неактивные дорожки маскируются и выполняются как холостые операции, – прежде чем воссоединиться в единое исполнение. Агрегация по окрестности GNN – естественный источник такого расхождения групп потоков и дисбаланса нагрузки: если одна группа потоков обрабатывает вершины с разными степенями, потоки с вершинами малой степени заканчивают свой цикл агрегации рано, тогда как потоки с вершинами высокой степени продолжают итерировать по соседям. Группа потоков ждёт самого медленного потока, тратя циклы выполнения впустую.

Современные GPU имеют глубокую иерархию памяти [14]. HBM (память с высокой пропускной способностью) – основная память GPU: наибольшая ёмкость, наибольшая задержка, основное бутылочное горлышко операций, ограниченных скоростью чтения/записи. Далее по иерархии – L2-кэш, общий для всех SM, позволяющий повторное использование данных между SM без управления программистом. Ещё ближе – L1/разделяемая память на SM: на порядки более низкая задержка, чем у HBM, меньшая ёмкость; может конфигурироваться как разделение между кэшем L1 и программно управляемой SMEM. Наконец, регистры – частные для каждого потока, наименьшая ёмкость, наибольшая скорость.

Критическим фактором для пропускной способности HBM является слитность доступа: когда 32 потока этой группы обращаются к последовательным адресам памяти, оборудование обслуживает все 32 обращения в одной транзакции. При неслитных – разрозненных – обращениях каждое может потребовать отдельной транзакции, умножая трафик памяти.

Нерегулярные шаблоны доступа GNN могут демонстрировать именно эту проблему: индексы соседей зависят от данных и не последовательны, поэтому сбор строки признаков соседа порождает HBM-доступ, который нельзя предсказать или слить, и повторные сборы быстро насыщают пропускную способность памяти.

Roofline-модель и арифметическая интенсивность. Удобный способ рассуждать о производительности GPU – Roofline-модель [15], связывающая достижимую производительность с арифметической интенсивностью (FLOPs на байт, перемещённый из HBM). Ядро является ограниченным вычислениями, если оно выполняет достаточно арифметики на байт, чтобы вычислительные блоки GPU стали бутылочным горлышком; ограниченным памятью, если перемещает больше данных относительно выполняемой арифметики. Граница между этими режимами – точка гребня – определяется соотношением пиковой производительности и пиковой пропускной способности HBM. Высокоинтенсивные операции, такие как умножение плотных матриц, как правило, ограничены вычислениями, тогда как разреженные и редукционные примитивы GNN имеют низкую арифметическую интенсивность и часто ограничены памятью; материализация промежуточных тензоров по рёбрам дополнительно увеличивает перемещение данных и ухудшает интенсивность. Снижение FLOPs само по себе не улучшает фактическое время выполнения для ограниченных памятью ядер.

Слияние ядер и материализация промежуточных результатов. На GPU каждый запуск ядра читает входные данные из HBM и записывает результаты обратно в HBM. Слияние ядер решает эту проблему, объединяя несколько операций в одно ядро, так что промежуточные результаты хранятся в регистрах или SMEM и никогда не записываются в HBM [16-17]. Это особенно эффективно для цепочек операций, ограниченных пропускной способностью. Применение слияния к вычислениям GNN обсуждается далее в разделе 2.

2.4 Реализации GNN и предшествующие работы по ускорению

Два наиболее широко используемых библиотеки машинного обучения на графах для PyTorch – Deep Graph Library (DGL) и PyTorch Geometric (PyG). DGL предоставляет граф-центричный API и реализует слои GNN через два обобщённых разреженных примитива: обобщённый SpMM (gSpMM) для поузловой агрегации и обобщённое выборочное умножение плотной матрицы (gSDDMM) для порёберных вычислений. В своей CUDA-реализации DGL типично параллелизует gSpMM по целевым вершинам, а gSDDMM – по рёбрам; нативный внутренний формат – CSR [18]. PyG следует PyTorch-нативному дизайну и обычно реализует передачу сообщений через scatter/gather редукции и лёгкие CUDA-расширения; нативный формат хранения рёбер – COO [19].

Хотя DGL предоставляет эффективные ядра на уровне операторов (через dgl.ops), более сложные слои нередко выражаются как композиция gSDDMM и gSpMM, что приводит к материализации промежуточных сообщений по рёбрам и последующей их редукции в обеих библиотеках, увеличивая трафик памяти и пиковое потребление памяти активациями. Более того, слияние таких конвейеров средствами автоматической компиляции нетривиально, поскольку вычисление чередует порёберные и поузловые фазы, что затрудняет одновременное сохранение локальности и балансировки нагрузки на всём протяжении слоя [4].

Предшествующие работы по ускорению GNN на GPU охватывают вендорские библиотеки, разреженные примитивы ядер, системы слияния и графовые/форматные преобразования. На стороне вендора cuSPARSE предоставляет высоко оптимизированную разреженную линейную алгебру (в первую очередь SpMM) и служит сильным базовым решением для слоёв GNN на основе SpMM. Однако операторы редукции и внимания не покрываются обобщённым SpMM, а их производительность сильно зависит от выбора параллелизации, промежуточной материализации и локальности памяти. В экосистеме RAPIDS библиотека

cuGraph аналогично предоставляет GPU-примитивы для графов и ориентированные на GNN компоненты.

Помимо вендорных библиотек, GE-SpMM предлагает CSR-нативные ядра в стиле SpMM, адаптированные к использованию в GNN, и вводит слитное кэширование строк и техники на уровне групп потоков для улучшения поведения памяти [20]. Адаптивное разреженное плиточное разбиение улучшает локальность для SpMM/SDDMM в CSR за счёт внутристрочной перестановки, позволяющей плиточное разбиение и повторное использование данных [21]. Классические анализы разреженных GPU-ядер (например, CSR SpMV на GPU с кэшем) подчёркивают роль локальности, параметризованных отображений и (авто)настройки, что остаётся актуальным при интерпретации эффектов кэша и перестановки графа в современных рабочих нагрузках GNN [22].

Дополнительное направление нацелено на сквозные графы выполнения и слияние. FuseGNN, Fused3S и родственные конвейеры снижают накладные расходы на запуск ядер и материализацию промежуточных тензоров за счёт слияния общих последовательностей операторов (например, SDDMM–softmax–SpMM для разреженного внимания) [23-24]. DF-GNN адаптирует слияние и расписание потоков к структуре операции и перекосу степеней [25]. GNNAdvisor изучает вычислительные графы через согласованную призму вычислений, ввода-вывода и памяти и предлагает реорганизацию операторов, унифицированное отображение потоков для слияния порёберных и поузловых фаз, а также перевычисление для снижения объёма активаций [26]. Cached Operator Reordering исследует альтернативные порядки выполнения с кэшированием, хотя публичные реализации доступны не всегда [27]. MaxK-GNN представляет совместное проектирование алгоритма и системы, модифицирующее представления для снижения трафика памяти и включения специализированных ядер [3].

Наконец, ряд работ задействует специализированные блоки GPU через форматные/графовые преобразования. TC-GNN и cuTeSpMM исследуют плотные тензорные ядра Tensor Cores для SpMM через структурированную блокировку и вводят зависящие от структуры критерии полезности Tensor-Core-ускорения [28-29]. RT-GNN нацелен на разреженные тензорные ядра Sparse Tensor Cores, переставляя графы для лучшего соответствия ограничениям структурированной разреженности; его практическая польза зависит от стоимости перестановки, ограничений совместимости и того, насколько близко оцениваемые постановки соответствуют реальным сквозным конвейерам [30].

2.5 Перестановка вершин графа для локальности

Перестановка вершин графа применяет перестановку π к индексам вершин для улучшения локальности кэша за счёт размещения часто совместно используемых вершин ближе в памяти. Классические методы включают разбиение в стиле METIS и порядки уменьшения заполнения [31]. В GNN перестановка может снижать трафик памяти и улучшать общее время [32]. В данной работе показывается, что её польза не универсальна: эффект зависит от шаблона доступа к памяти ядра и структуры графа. Условия, при которых перестановка помогает, исследуются экспериментально в разделе 4.

2.6 Проблемы при работе с графовыми нейросетями

Рабочие нагрузки GNN сочетают разреженный нерегулярный доступ к графу с плотными поузловыми/порёберными вычислениями признаков, что порождает ряд специфических для GPU трудностей.

Во-первых, агрегация по окрестности требует сбора признаков из непоследовательных индексов вершин. Даже при слитных загрузках по измерению признаков последовательность базовых адресов соседей может быть весьма нерегулярной, снижая эффективность кэша и

увеличивая давление на DRAM. Кроме того, в формулировках передачи сообщений промежуточные сообщения по рёбрам (нередко размерности $O(M \cdot D\_l)$) могут материализоваться до агрегации, что дополнительно увеличивает трафик памяти и пиковое потребление.

Во-вторых, слои GNN по своей природе требуют порёберных вычислений с последующими поузловыми редукциями. Хотя современные компиляторы глубокого обучения, такие как torch.compile, весьма мощны, эффективное слияние затруднено в этом чередующемся рабочем процессе [4,33–35].

Эти характеристики мотивируют проектирование ядер с учётом ввода-вывода, которые (i) снижают промежуточную материализацию, (ii) увеличивают повторное использование данных на кристалле через плиточное разбиение и слияние, где возможно, и (iii) явно учитывают перекос степеней и чувствительность к компоновке при отображении работы на потоки GPU. В разделе 3 каждая из этих проблем решается через учитывающие ввод-вывод проекты ядер, адаптированные к трём семействам операторов.

3. GPU-ядра, учитывающие структуру ускорителя

В данном разделе представлены четыре учитывающих ввод-вывод проекта ядер: ядра редукции с учётом степеней вершин, слитое CSR-внимание с онлайн-softmax, блочно-разреженные компоновки для тензорных ядер Tensor Cores через формат WSB и SpMM на основе cuSPARSE с кэшированием структуры графа. Более новаторские вклады представлены первыми; свёртки на основе SpMM, где основным строительным блоком является вендорский примитив с кэшированием, рассматриваются в конце.

3.1 Агрегация на основе редукции с учётом степеней вершин

Реальные графы имеют тяжёлые хвосты распределения степеней [36,37], что вызывает серьёзный дисбаланс нагрузки при обработке каждой целевой вершины одним блоком потоков. Для решения этой проблемы целевые вершины разбиваются на подмножества light (лёгкие) и heavy (тяжёлые) через квантиль порогового значения распределения степеней, установленный на верхний квантиль.

Адаптивное разбиение по степени. Лёгкие вершины обрабатываются ядром с параллелизмом по признакам: один блок на вершину сканирует список соседей с слитными чтениями, сохраняя состояние потока в регистрах. Для тяжёлых вершин список соседей разбивается на непрерывные фрагменты рёбер, каждый обрабатывается отдельным блоком потоков, вычисляющим локальную частичную редукцию; частичные результаты по фрагментам затем объединяются атомарными операциями над упакованными 64-битными парами (value, index), позволяя одной атомарной операцией одновременно обновлять текущий минимум и соответствующий индекс исходной вершины. Прямой проход сохраняет эти argmin-индексы; обратный использует их для распространения градиентов как разреженное scatter-add по выбранным источникам, избегая повторного редукционного прохода. Пороговый квантиль степени и параметры запуска ядра выбираются автонастройщиком для каждого графа и размерности признаков.

Агрегации на основе редукции имеют очень низкую арифметическую интенсивность и фундаментально ограничены пропускной способностью. Для графа с M рёбрами и размерностью признаков D оба варианта реализации читают $O(M \cdot D)$ байт из HBM при прямом проходе; разбиение не изменяет этой асимптотической оценки. Выигрыш – в задержке: параллелизация сканирования соседей тяжёлых вершин по нескольким блокам увеличивает параллелизм и снижает хвостовую задержку.

3.2 Слитное CSR-внимание

Слои GNN на основе внимания, как правило, материализуют порёберные предсофтмаксные оценки или сообщения размерности $O(M \cdot H)$ или $O(M \cdot H \cdot D)$, где H – число голов. Это увеличивает трафик HBM и пиковое потребление памяти. Разрабатываются слитые CSR-ядра, выполняющие вычисление весов, нормировку softmax и агрегацию значений за один потоковый проход по соседям, исключая материализацию промежуточных тензоров по рёбрам. Предоставляются слитые ядра как для точечно-произведённого внимания (граф-трансформер), так и для GATv2.

Слитный прямой проход – онлайн-softmax и выполнение с учетом степеней. Прямой проход сливает вычисление оценок, нормировку softmax и агрегацию значений в одно потоковое CSR-ядро, исключая материализацию промежуточных тензоров по рёбрам. Каждый блок потоков обрабатывает одну пару (индекс вершины, индекс головы) и потоково обходит CSR-окрестность с векторизованными загрузками. Следуя алгоритму онлайн-softmax [38], каждая группа потоков хранит в регистрах текущий максимум m и текущий нормировщик ℓ, обновляя их инкрементально для каждого обработанного соседа. Когда m обновляется с m_{old} до m_{new}, аккумулятор выхода масштабируется корректирующим множителем $\exp(m_{\text{old}} - m_{\text{new}})$, и взвешенный вклад значений свёртывается в тот же проход – без необходимости хранить предсофтмаксные оценки или веса внимания по рёбрам. Для обработки перекоса степеней, аналогично агрегации на основе редукции (раздел 3), применяется выполнение с учётом степеней: для вершин высокой степени несколько групп потоков назначаются одной паре (индекс вершины, индекс головы) для увеличения параллелизма; их частичные онлайн-softmax состояния объединяются примитивами на уровне групп потоков, а затем блок-уровневой редукцией через разделяемую память для получения финального нормированного выхода.

Обратный проход с эффективным использованием памяти. Наивный обратный проход потребовал бы повторного чтения $O(M \cdot H)$ весов внимания, материализованных при прямом проходе, что доминирует в трафике HBM на плотных графах. Следуя стратегии FlashAttention-2 [39], при прямом проходе кэшируются только компактные поузловые logsumexp-статистики $L \in \mathbb{R}^{N \times H}$; при обратном проходе каждый блок перевычисляет веса внимания на лету из кэшированного L и уже находящихся в регистрах признаков соседей, устраняя чтение $O(M \cdot H)$ активаций. Это сохраняет пиковое потребление памяти на уровне $O(N \cdot H)$ независимо от размера графа – именно это отвечает за большие сокращения памяти, наблюдаемые на практике (например, 44x на twitch-views). Та же стратегия перевычисления применяется и для граф-трансформера, и для GATv2.

Слитое выполнение устраняет доминирующий источник накладных расходов HBM в слоях внимания. Декомпозированный базовый вариант записывает и затем перечитывает предсофтмаксные оценки или веса внимания $\alpha \in \mathbb{R}^{M \times H}$ в нескольких фазах; обобщённые реализации передачи сообщений могут материализовывать полные сообщения по рёбрам $\boldsymbol{E} \in \mathbb{R}^{M \times H \times D}$; оба этих расхода масштабируются с числом рёбер. Наши слитые ядра записывают только компактную поузловую статистику размерности $O(N \cdot H)$ при прямом проходе (табл. 1). Поскольку $M \gg N$ для многих реальных графов (социальные, цитационные сети), выигрыш растёт с плотностью и является основным фактором наблюдаемых значительных сокращений памяти.

3.3 Блочно-разреженные представления для тензорных ядер

Инструкции WMMA тензорных ядер Tensor Cores на современных GPU NVIDIA разработаны для умножений матриц 16×16; на архитектуре Ampere реализация нацелена на точность BF16. Размер окна строк фиксируется в 16 целевых вершинах для соответствия этой форме инструкции, и используется блочно-разреженный формат из [23] (Fused3S), упаковывающий невзвешенное разреженное внимание в плитки 16×16 для выполнения на тензорных ядрах.

Табл. 1. Сохранённые активации для слоёв внимания при обучении. Наши слитые ядра избегают хранения пореберных выходов softmax и вместо этого кэшируют компактную поузловую статистику. L – поузловой logsumexp; $\Delta \in \mathbb{R}^{N \times H}$ – поузловое скалярное произведение $\langle O_{i,h,:}, dO_{i,h,:} \rangle$, кэшируемое для обратного прохода граф-трансформера; $G \in \mathbb{R}^{N \times H}$ – аналогичный поузловой нормировщик для GATv2.
Table 1. Stored activations for attention layers during training. Our fused kernels avoid storing softmax edge outputs and instead cache compact node-wise statistics. L is the node-wiselogsumexp; $\Delta \in \mathbb{R}^{N \times H}$ is the node-wise dot product $\langle O_{i,h,:}, dO_{i,h,:} \rangle$, cached for the backward pass of the graph transformer; $G \in \mathbb{R}^{N \times H}$ – is the analogous node-wise normalizer for GATv2.

Метод	Тензоры по рёбрам	Тензоры по вершинам	Пиковый размер
Общий MP (сообщения)	$\boldsymbol{E} \in \mathbb{R}^{M \times H \times D}$	–	$O(MHD)$
Ванильный (веса по рёбрам)	$\alpha \in \mathbb{R}^{M \times H}$	$L \in \mathbb{R}^{N \times H}$ (опц.)	$O(MH)$
Наш (слитый, граф-трансформер)	–	$L, \Delta \in \mathbb{R}^{N \times H}$	$O(NH)$
Наш (слитый, GATv2)	–	$L, G \in \mathbb{R}^{N \times H}$	$O(NH)$

Стандартные свёртки GNN требуют скалярного веса матрицы смежности на каждое ребро, что Fused3S не поддерживает; формат расширяется добавлением плотной плитки весов 16×8 на каждый блок тензорных ядер (TCB), получая формат *Weighted Sparse-Block* (WSB). Каждый TCB хранит: (i) 8 глобальных индексов столбцов (col_idx), (ii) компактное 128-битное битовое поле, кодирующее, какие из 16×8 потенциальных рёбер присутствуют, и (iii) для SpMM – плотная плитка весов 16×8 с нулевым заполнением для отсутствующих рёбер. Препроцессинг выполняется однократно на граф и амортизируется по итерациям обучения. Предоставляются Triton-ядра как для SpMM, так и для внимания граф-трансформера над форматом WSB.

Выполнение на тензорных ядрах, компромиссы и взгляд с точки зрения HBM. Для SpMM каждый программный экземпляр охватывает одно окно строк из 16 вершин и итерирует по парам последовательных TCB. Два TCB (16×8 каждый) объединяются в плитку 16×16; соответствующие 16 столбцов признаков исходных вершин собираются, и тензорное ядро выполняет $Y_{\text{win}} += W_{\text{full}} \cdot X_{\text{tile}}$ как WMMA умножение-накопление. Для внимания граф-трансформера ядро загружает $Q_{\text{win}} \in \mathbb{R}^{16 \times D}$ и итерирует по парам TCB, вычисляя $S = Q_{\text{win}} K_{\text{blk}}^{\top}$ через WMMA, применяя маску битового поля к отсутствующим рёбрам и выполняя онлайн-softmax на уровне плитки, накапливая взвешенные значения без материализации порёберных предсофтмаксных оценок. В обоих случаях каждый загруженный вектор признаков участвует в скалярных произведениях со всеми 16 строками окна, обеспечивая существенно более высокую арифметическую интенсивность, чем скалярный CSR-поток.

WSB наиболее эффективен, когда окна строк имеют достаточную локальную плотность: стоимость сбора амортизируется плотным вычислением 16×16, и преимущество пропускной способности тензорных ядер доминирует. Для очень разреженных графов накладные расходы на маскированные вычисления и заполнение могут свести на нет или обратить вспять преимущество [23,29]. Обратный проход менее благоприятен: столбцы исходных вершин могут встречаться в нескольких окнах строк, поэтому обновления dK и dX требуют атомарного накопления; обратный проход поэтому вычисляется через стандартный SpMM-примитив на предварительно вычисленной транспонированной матрице смежности. С точки зрения HBM, WSB является оптимизацией арифметической интенсивности, а не чистой техникой сокращения ввода-вывода: и CSR, и WSB должны загружать одни и те же признаки вершин, но плиточное разбиение WSB означает, что каждый загруженный вектор участвует в целой плитке скалярных произведений, а не в единственном скалярном порёберном вычислении, и может слегка увеличивать трафик признаков при сборе заполненных столбцов.

FLOPs и арифметическая интенсивность. Для точного обоснования аргумента об арифметической интенсивности анализируются FLOPs прямого прохода граф-трансформера и трафик HBM для обоих форматов. Пусть N – число вершин, nnz – число ненулевых рёбер, D – суммарная размерность представления ($D = H \cdot d$, объединяя число голов H в D для краткости).

CSR. Для каждого ненулевого ядро выполняет скалярное произведение запрос–ключ ($2D$ FLOPs), нормировку softmax (2 FLOPs) и взвешенное умножение-накопление значений ($2D$ FLOPs):

$$\mathbf{FLOPs_{CSR}} \approx \mathbf{nnz} \cdot (\mathbf{4}\boldsymbol{D} + \mathbf{2}).$$

Q, K, V загружаются по одному разу ($6ND$ байт в BF16), один INT32 индекс столбца читается на каждый ненулевой (4 байта), выход записывается один раз ($2ND$ байт), давая $\mathrm{Bytes_{CSR}} \approx 4\,\mathrm{nnz} + 8ND$. Арифметическая интенсивность:

$$\mathbf{AI_{CSR}} \approx \frac{\mathbf{nnz} \cdot (\mathbf{4}\boldsymbol{D} + \mathbf{2})}{\mathbf{4\,nnz} + \mathbf{8}\boldsymbol{ND}}.$$

Поскольку CSR-ядра выполняются на скалярных CUDA-ядрах, достижимая производительность ограничена первым из двух пределов:

$$\boldsymbol{P_{CSR}} = \mathbf{min}(\boldsymbol{P_{CUDA}},\ \mathbf{AI_{CSR}} \cdot \boldsymbol{B_{mem}}),$$

где P_{CUDA} – пиковая скалярная производительность CUDA-ядер, а B_{mem} – пропускная способность памяти. При типичных размерах графов и размерностях представлений $\mathrm{AI_{CSR}}$ мало, и ядро ограничено памятью.

WSB. Те же операции организуются в блоки 16×16. Пусть μ – среднее число ненулевых на блок ($\mu \le 256$) и $n_b = \mathrm{nnz}/\mu$ – число блоков. Для каждого блока SpDDMM $Q_{\mathrm{win}}(16 \times D) \cdot K_{\mathrm{blk}}^{\top}(D \times 16)$ – это полное умножение $16 \times 16 \times D$ ($512D$ FLOPs); онлайн-softmax по плитке предсофтмаксных оценок 16×16 стоит 512 FLOPs; агрегация значений ещё $512D$ FLOPs, итого:

$$\mathbf{FLOPs_{WSB}} \approx \boldsymbol{n_b}(\mathbf{1024}\boldsymbol{D} + \mathbf{512}).$$

Глобальная загрузка Q, K, V и запись выхода остаются $8ND$ байт; каждый блок загружает 256 BF16-элементов столбцов (512 байт) и 128-битное битовое поле (16 байт): $\mathrm{Bytes_{WSB}} \approx 528\,n_b + 8ND$. Арифметическая интенсивность:

$$\mathbf{AI_{WSB}} \approx \frac{\boldsymbol{n_b}(\mathbf{1024}\boldsymbol{D} + \mathbf{512})}{\mathbf{528}\,\boldsymbol{n_b} + \mathbf{8}\boldsymbol{ND}}.$$

Подставляя $n_b = \mathrm{nnz}/\mu$ и сокращая общий множитель $(2D + 1)$, получаем замкнутое сравнение:

$$\frac{\mathbf{AI_{WSB}}}{\mathbf{AI_{CSR}}} = \frac{\mathbf{1024}(\mathbf{nnz} + \mathbf{2}\boldsymbol{ND})}{\mathbf{528\,nnz} + \mathbf{8}\boldsymbol{ND\mu}}.$$

Поскольку $\mu \le 256$, знаменатель удовлетворяет $528\,\mathrm{nnz} + 8ND\mu \le 1024(\mathrm{nnz} + 2ND)$, поэтому отношение всегда ≥ 1: WSB имеет арифметическую интенсивность не меньше, чем CSR, при любом размере графа или плотности. Два предельных случая показательны: когда доминируют рёбра ($\mathrm{nnz} \gg ND$), отношение приближается к $1024/528 \approx 1{,}94$; когда доминируют признаки вершин ($ND \gg \mathrm{nnz}$), оно приближается к $2048ND/(8ND\mu) = 256/\mu = 1/\rho$, где $\rho = \mu/256$ – средняя доля заполнения плитки. Поскольку WSB использует тензорные ядра Tensor Cores, потолок производительности возрастает до P_{TC}:

$$\boldsymbol{P_{WSB}} = \mathbf{min}(\boldsymbol{P_{TC}},\ \mathbf{AI_{WSB}} \cdot \boldsymbol{B_{mem}}).$$

Ключевой вывод, таким образом, двоякий: WSB имеет строго более высокую арифметическую интенсивность, чем CSR, во всех режимах, *и* его потолок производительности равен $P_{\mathrm{TC}} \gg P_{\mathrm{CUDA}}$. За это приходится платить: WSB выполняет полное плотное умножение 16×16 на каждый блок независимо от истинной плотности плитки ρ, раздувая фактические FLOPs в $\sim 1/\rho$ раз относительно истинного числа ненулевых.

Преимущество тензорных ядер окупается, пока $P_{\mathrm{TC}}/P_{\mathrm{CUDA}} \gg 1/\rho$, что выполняется для умеренно плотных окон на графах с кластеризованной структурой.

3.4 Свёртки на основе SpMM

Учитывая зрелость разреженной линейной алгебры, предоставляемой NVIDIA, в качестве основной реализации используется SpMM из NVIDIA cuSPARSE. Нормировка матрицы смежности обрабатывается как этап препроцессинга: степени вершин и нормированные значения рёбер предварительно вычисляются и передаются как буфер значений CSR. Дополнительно поддерживается автоподбор вариантов алгоритма cuSPARSE для выбора наилучшей конфигурации под каждый граф и размерность признаков.

Кэш структуры графа и переиспользование для обратного прохода. При обучении GNN граф фиксирован, а SpMM вызывается многократно за эпоху. Для устранения повторных накладных расходов на инициализацию кэшируются метаданные, специфичные для графа: дескрипторы cuSPARSE, нормированные значения рёбер, выбранный алгоритм и буферы рабочего пространства. Для обратного прохода ($\boldsymbol{dX} = \boldsymbol{A}^{\top}\boldsymbol{dY}$) опционально предварительно вычисляется и кэшируется транспонированная матрица смежности, сводя оба прохода к почти чистым вызовам библиотеки.

SpMM на тензорных ядрах через WSB. В дополнение к cuSPARSE предоставляются специфичные для WSB ядра SpMM с тензорными ядрами Tensor Cores, использующие взвешенный блочно-разреженный формат, описанный выше (раздел 3). Число FLOPs практически совпадает со стандартным SpMM, но структура плиток WSB позволяет задействовать тензорные ядра Tensor Cores, обеспечивая значительно более высокую пиковую производительность по сравнению со скалярными CUDA-ядрами.

4. Эксперименты

4.1 Описание протокола экспериментов

Оценка проводится на GraphLand [40] – наборе тестовых наборов данных с разнообразными структурами графов, на ogbn-arxiv и ogbn-products из OGB [41], на стандартных цитационных сетях Cora, CiteSeer и PubMed [42] как проверочных наборах малого масштаба, а также на графах дорожных сетей из коллекции GraphLand (city-roads-M и city-roads-L). В табл. 2 приведена структурная статистика всех оцениваемых графов.

Ряд наборов данных GraphLand использует один и тот же базовый граф, но отличается задачей предсказания: наборы artnet-views и artnet-exp используют один социальный граф; hm-prices и hm-categories – один граф совместных покупок; web-fraud, web-topics и web-traffic – один ориентированный веб-граф. В эксперименты включён один представитель от каждой группы. Задача предсказания не влияет на оценку: ускоряется сам слой GNN, а числовое соответствие прямого и обратного выходов с эталонной реализацией гарантирует идентичную сходимость.

Все эксперименты используют полно-пакетное обучение на одном GPU NVIDIA A100 80GB SXM4. В качестве представителей семейств оцениваются GCN для слоёв на основе SpMM, min-агрегация для слоёв на основе редукции, а также граф-трансформер и GATv2 для слоёв на основе внимания; предложенные техники оптимизации остаются широко применимыми и к исходной архитектуре GAT. Для слоёв на основе SpMM и редукции диапазон размерностей признаков $D \in \{64,128,256,512\}$. Для GATv2 число голов $H \in \{2,4,8\}$ и размерность на голову $D \in \{32,64,128,256\}$ (так что суммарная скрытая размерность равна $H \cdot D$). Для граф-трансформера суммарная скрытая размерность фиксирована, варьируются пары (H, D). Оцениваются перестановки METIS с размерами разбиений $k \in \{128,512,\ldots,65536\}$ против исходного порядка. Параметры ядер настраиваются автоматически. Измеряются задержка прямого/обратного прохода и пиковое потребление памяти GPU.

Базовые реализации. В качестве основного базового метода используется DGL; все ускорения приводятся относительно его поведения. Для свёрток на основе SpMM дополнительно используются cuSPARSE SpMM с форматом CSR, TC-GNN[3], FuseGNN, PyG и ванильные разреженные операции PyTorch. Для слоёв на основе редукции дополнительно используется cuGraph. Для граф-трансформера при наличии сравнивается DF-GNN (используется вариант DF-GNN-hyper как единственный с доступной реализацией обратного прохода). Дополнительно оцениваются собственные ядра на тензорных ядрах Tensor Cores для SpMM и граф-трансформера (обозначаются как реализация WSB/TC).

Не все сторонние системы удалось оценить на каждом наборе данных из-за дополнительных допущений и инженерных ограничений. В частности, DF-GNN успешно выполнился лишь на 8 графах нашего набора. Fused3S не смог построить требуемое блочно-разреженное представление для нескольких больших графов web-topics, а даже при успешном построении наблюдался незаконный доступ к памяти, приводящий к сбоям во время выполнения. Более того, выпущенный код Fused3S не предоставляет реализации обратного прохода для обучения, поэтому мы полагаем, что он не пригоден в качестве сквозной альтернативы для обучения. Наконец, операторы внимания cuGraph требуют двудольного графа для поддержки слоёв с раздельными левыми/правыми признаками вершин (как в GATv2) и для соответствия интерфейсу граф-трансформера; поскольку наши тестовые наборы данных для внимания используют недвудольные графы, базовые решения на основе внимания cuGraph не приводятся.

Табл. 2. Статистика наборов данных (вычислена на обработанных нами графах).
Table 2. Statistics of data sets (calculated on the graphs we processed).

Набор данных	#вершин	#рёбер	плотность	ср. степень
hm-categories	46,563	21,461,990	9.90×10^{-3}	460.92
tolokers-2	11,758	1,038,000	7.51×10^{-3}	88.28
avazu-ctr	76,269	21,968,154	3.78×10^{-3}	288.04
cora	2,708	10,556	1.44×10^{-3}	3.90
citeseer	3,327	9,104	8.23×10^{-4}	2.74
twitch-views	168,114	13,595,114	4.81×10^{-4}	80.87
pubmed	19,717	88,648	2.28×10^{-4}	4.50
artnet-exp	50,405	560,696	2.21×10^{-4}	11.12
city-reviews	148,801	2,330,830	1.05×10^{-4}	15.66
city-roads-M	57,073	132,570	4.07×10^{-5}	2.32
ogbn-arxiv	169,343	1,166,243	4.07×10^{-5}	6.89
ogbn-products	2,449,029	123,718,280	2.06×10^{-5}	50.52
city-roads-L	142,257	279,061	1.38×10^{-5}	1.96
pokec-regions	1,632,803	30,622,564	1.15×10^{-5}	18.75
web-traffic	2,890,331	12,895,369	1.54×10^{-6}	4.46

Представления графов и накладные расходы на их получение. Разные реализации требуют разных кодировок и препроцессинга графа. В табл. 3 приведено потребление памяти (МБ) кэшированными/препроцессированными объектами каждой реализации. CSR-

[3] Мы заметили, что TC-GNN содержит ошибку в обратном проходе для неориентированных графов в GCN и требует транспонирования матрицы смежности в реальном времени; добавлено исправление в исходный код метода.

реализации (DGL, PyG, cuSPARSE и наши CSR-слитые ядра) хранят смещения строк и индексы столбцов с опциональными значениями рёбер. Кэширование транспонированной матрицы для обратного прохода в cuSPARSE увеличивает потребление памяти представлением: например, с 184,36 МБ до 203,72 МБ на ogbn-arxiv и с 496,16 МБ до 626,39 МБ на тестовом графе pokec-regions.

Табл. 3. Потребление памяти (МБ) представлениями графов, требуемыми разными реализациями. Значения измерены для кэшированных/препроцессированных объектов графа, используемых каждой реализацией; N/A – недоступные измерения из-за ошибки построения блочно-разреженного представления в Fused3S.
Table 3. Memory consumption (MB) of graph representations required by different backends. Values are measured for cached/preprocessed graph objects used by each backend; N/A – measurements are unavailable due to an error in constructing the block-sparse representation in Fused3S.

Набор данных	DGL	PyG	cuGraph	cu SPARSE	cu SPARSE + $\widetilde{A^T}$	Torch sparse	**Наш (CSR fused)**	TC-GNN	DF-GNN	Fused3S	**Наш (WSB/ TC)**
artnet-exp	24.28	24.28	59.63	17.48	20.00	46.62	21.11	21.57	26.03	34.97	49.72
avazu-ctr	413.17	413.17	1681.86	252.57	336.95	1681.86	245.86	329.11	497.38	1343.80	1681.86
citeseer	47.22	47.22	47.96	47.09	47.15	47.30	47.17	47.15	47.23	47.09	47.82
city-reviews	155.40	155.56	298.91	127.59	137.62	189.67	138.18	144.84	163.16	143.78	266.75
city-roads-L	120.84	120.84	147.46	116.56	118.71	124.24	119.25	118.18	120.82	116.12	134.01
city-roads-M	18.30	18.30	29.59	16.35	17.29	19.46	17.51	17.15	8.37	16.12	23.78
cora	15.04	15.04	15.82	14.89	14.96	15.11	14.97	14.97	15.06	14.90	15.63
hm-categories	350.69	350.68	1640.98	246.32	328.55	1640.98	187.12	268.68	432.84	1311.87	1640.98
ogbn-arxiv	124.16	125.45	224.84	184.36	203.72	224.84	197.98	147.29	208.97	156.08	224.84
ogbn-products	4791.77	4811.92	13996.19	10582.25	12072.83	13996.19	11582.20	7599.92	13771.59	8511.87	13996.19
pokec-regions	858.82	858.38	2717.03	496.16	626.39	2461.85	632.62	723.00	962.47	1871.63	2461.85
pubmed	39.86	39.86	46.22	38.70	39.19	40.43	39.26	39.30	40.05	38.77	45.03
tolokers-2	17.00	16.99	80.09	12.06	16.11	80.09	9.12	12.90	21.07	63.92	80.09
twitch-views	229.77	228.48	1053.04	159.14	212.29	1053.04	124.91	174.45	279.05	831.86	1053.04
web-traffic	13281.56	13281.56	14206.88	13112.91	13184.15	13363.83	13197.13	13202.41	13314.69	N/A	13860.13

Наш путь с тензорными ядрами Tensor Cores использует взвешенную блочно-разреженную компоновку (WSB), которая обменивает дополнительную память структуры на более высокую арифметическую интенсивность. Следовательно, WSB может быть существенно дороже CSR на плотных или локально плотных графах, например hm-categories (246,32 МБ у cuSPARSE против 1640,98 МБ у нашего WSB/TC) и tolokers-2 (12,06 МБ против 80,09 МБ). На очень больших разреженных графах накладные расходы более умеренны относительно уже большого CSR-следа, например web-traffic (13112,91 МБ у cuSPARSE против 13860,13 МБ у нашего WSB/TC). Наконец, для некоторых наборов данных Fused3S не может построить требуемое блочно-разреженное представление (отмечено как N/A в табл. 3), что подчёркивает: требования формата влияют не только на память, но и на устойчивость конвейера оценки.

4.2 Основные результаты

В данном разделе обобщаются основные экспериментальные результаты и приводятся наиболее показательные измерения производительности.

cuSPARSE по-прежнему сложно превзойти для свёрток на основе SpMM. Как показано в табл. 4, cuSPARSE остаётся сильным базовым решением для слоёв на основе SpMM. Например, с кэшированными дескрипторами cuSPARSE достигает ускорения 2,30x прямого прохода на ogbn-arxiv и 3,24x/1,51x на city-roads-L (прямой/обратный), тогда как кэширование предвычисленной $\tilde{A}^{\top}$ для обратного прохода повышает ускорения обратного прохода до 1,64x и 1,74x на тех же графах (и до 1,60x на web-traffic и web-topics) ценой дополнительной памяти. Компоновка WSB может быть конкурентоспособной в некоторых режимах (например, 2,59x прямого прохода на tolokers-2 и 2,35x на city-roads-L), но зависит от структуры и несёт накладные расходы формата, поэтому рассматривается как опциональная оптимизация, а не универсальная замена CSR+cuSPARSE. Ряд специализированных базовых решений менее устойчив на этом масштабе: TC-GNN достигает 4,99x/1,94x на city-roads-L, но показывает почти нулевую пропускную способность на web-traffic и исчерпывает память на ogbn-arxiv.

Свёртки на основе редукции: разбиение с учётом степеней даёт значительные прямые выигрыши. Как показано в табл. 5, для конфигурации с $D = 512$ наше разбиение существенно ускоряет прямой проход по сравнению с DGL на всех графах, достигая до 8,31x на web-traffic и 8,24x на web-topics, при сохранении устойчивых улучшений обратного прохода (например, 2,09x на ogbn-arxiv). Кроме того, наша реализация снижает пиковое потребление памяти по сравнению с базовым решением DGL как в прямом, так и в обратном проходе, с типичными коэффициентами в диапазоне 1,43–1,97x на основных графах. Ускорение наиболее заметно на графах с высокой средней степенью и менее выражено на разреженных графах с малой средней степенью, таких как city-roads-L (средняя степень 1,96).

Слои на основе внимания: слитые CSR-ядра улучшают задержку и существенно снижают пиковое потребление памяти. Как показано в табл. 6, для GATv2 наши слитые CSR-ядра обеспечивают значительные ускорения прямого прохода на всех основных графах при базовой конфигурации (например, 9,70x/1,71x на city-roads-L, 6,68x/2,02x на pokec-regions и 5,59x/1,89x на tolokers-2). Принципиально, слитая формулировка исключает материализацию промежуточных тензоров по рёбрам и даёт существенные сокращения пиковой памяти (например, 32,59x/28,55x на tolokers-2 и 44,17x/31,71x на twitch-views). Для большей скрытой размерности ($H = 8$, размерность головы $= 128$) DGL не масштабировался; наша реализация остаётся стабильной, при этом демонстрируя ещё более высокие прямые выигрыши (до 9,78x на tolokers-2) и значительное сокращение памяти (например, 69,74x/44,80x на tolokers-2).

Для граф-трансформера (табл. 7) наше слитое CSR-ядро обеспечивает стабильные прямые и обратные выигрыши (например, 1,27x/1,24x на ogbn-arxiv и 2,00x/1,10x на tolokers-2), тогда как DF-GNN не работает на некоторых графах в этой конфигурации (N/A). Наше ядро WSB/TC с тензорными ядрами Tensor Cores может дополнительно улучшить прямую производительность на некоторых графах (например, 2,69x на tolokers-2), но обратный проход менее благоприятен из-за атомарного накопления по плиткам.

Распределение групп потоков с учётом степеней для ядер внимания. Разбиение с учётом степеней, введённое в разделе 3, естественно применяется к ядрам внимания. При высокой внутриграфовой дисперсии степеней блоки потоков для вершин высокой степени работают значительно дольше, создавая хвостовую задержку и недозагрузку SM. В табл. 8 приведено ускорение данного расширения над исходным ядром внимания без учёта степеней для граф-трансформера и GATv2 (размерность головы $= 512$, $H = 4$). Ускорение коррелирует с асимметрией распределения степеней.

Ядра обратного прохода с учётом направленности для неориентированных графов. Для неориентированных графов обратный проход может обходить ту же CSR-структуру, что и прямой, полностью устраняя дорогостоящие операции разбрасывания и атомарного накопления. Разрабатываются специализированные ядра обратного прохода с учётом

направленности для обеих реализаций (CSR-слитый и WSB). В табл. 9 приведено ускорение над направленным базовым обратным проходом (размерность головы $= 128$, $H = 4$). CSR-реализация GT выигрывает 5–21% от этой оптимизации; WSB достигает медианного ускорения 7,4x, поскольку устранение конкуренции scatter/атомарных операций полностью восстанавливает преимущество тензорных ядер Tensor Cores в обратном проходе.

Табл. 4. Результаты для SpMM с GCN (скрытая размерность $D = 512$); cuSPARSE переиспользует нормированные значения, CSR-дескриптор, рабочее пространство; «+ $\widetilde{A^{\top}}$» дополнительно кэширует предвычисленную $\widetilde{A^{\top}}$ для обратного прохода.

Table 4. Results for SpMM with GCN (latent dimension $D = 512$); reuses normalized values, CSR descriptor, and workspace; «+ $\widetilde{A^{\top}}$» additionally caches precomputed $\widetilde{A^{\top}}$ for the backward pass.

Реализация	ogbn-arxiv	web-traffic	twitch-views	pokec-regions	tolokers-2	city-roads-L
Ускорение задержки (t_{DGL}/t), прямой/обратный						
cuSPARSE	2.30/0.97	2.44/0.94	1.26/0.61	1.34/0.66	2.33/0.65	3.24/1.51
cuSPARSE + $\widetilde{A^{\top}}$	2.27/1.64	2.44/1.60	1.26/1.18	1.34/1.28	2.46/1.32	3.27/1.74
Наш (WSB/TC)	1.56/1.41	0.87/1.35	1.80/1.07	1.99/1.20	2.59/0.79	2.35/1.55
TC-GNN	2.63/OOM	<0.01/<0.01	0.08/0.09	1.76/2.05	0.71/0.50	4.99/1.94
FuseGNN	0.27/1.44	0.19/0.09	0.32/0.37	0.92/1.20	0.35/0.58	0.53/1.80
PyG	0.20/0.18	OOM/OOM	0.06/OOM	OOM/OOM	0.11/0.07	0.36/0.36
Коэффициент пиковой памяти ($\mathrm{mem}_{\mathrm{DGL}}/\mathrm{mem}$), прямой/обратный						
cuSPARSE	0.49/0.54	1.47/1.14	0.64/0.66	1.92/1.23	0.06/0.08	1.82/1.19
cuSPARSE + $\widetilde{A^{\top}}$	0.23/0.29	1.42/1.12	0.30/0.35	1.50/1.09	0.02/0.03	1.80/1.18
Наш (WSB/TC)	1.49/0.99	1.31/1.00	1.20/0.96	1.44/0.99	1.24/0.98	1.51/1.00
TC-GNN	1.93/OOM	1.48/1.15	1.36/0.84	1.78/1.45	1.92/1.17	2.38/1.48
FuseGNN	0.21/0.20	0.11/0.14	0.27/0.23	0.22/0.22	0.22/0.22	0.22/0.21
PyG	0.01/0.01	OOM/OOM	0.03/OOM	OOM/OOM	0.01/0.01	0.12/0.12

Табл. 5. Результаты для свёрток на основе редукции (оператор $\min$-агрегация) на выбранных графах (скрытая размерность $D = 512$).

Table 5. Results for reduction-based convolutions ($\min$-aggregation operator) on selected graphs (hidden dimension $D = 512$).

Реализация	ogbn-arxiv	web-traffic	web-topics	pokec-regions	tolokers-2	city-roads-L
Ускорение задержки (t_{DGL}/t), прямой/обратный						
Наш	4.84/2.09	8.31/1.52	8.24/1.50	2.59/1.41	2.54/1.32	4.14/1.63
cuGraph	2.87/1.45	0.06/2.17	0.06/2.11	1.42/1.14	0.44/0.87	3.50/1.38
Коэффициент пиковой памяти ($\mathrm{mem}_{\mathrm{DGL}}/\mathrm{mem}$), прямой/обратный						
Наш	1.71/1.37	1.43/1.28	1.57/1.33	1.77/1.40	1.97/1.51	1.66/1.35
cuGraph	1.19/1.14	1.19/1.13	1.24/1.15	1.27/1.16	1.19/1.12	1.28/1.17

Табл. 6. Ускорение задержки и коэффициент пиковой памяти для GATv2 на выбранных графах (головы $H = 2$, размерность головы $D = 64$); выше лучше. Нижний блок – ускорения для большей скрытой размерности (головы $H = 8$, размерность головы $= 128$); приводятся графы со стабильными измерениями на DGL.
Table 6. Latency speedup and peak memory ratio for GATv2 on selected graphs (heads $H = 2$, head dimension $D = 64$); The bottom panel shows speedups for higher latent dimensions (heads $H = 8$, head dimension $D = 128$); graphs with stable measurements on DGL are shown.

Ускорение задержки	ogbn-arxiv	twitch-views	artnet-exp	pokec-regions	tolokers-2	city-roads-L
Наш	2.82/0.98	5.32/1.30	5.45/2.04	6.68/2.02	5.59/1.89	9.70/1.71
PyG	0.95/0.95	0.57/0.67	0.39/0.51	OOM/OOM	0.45/0.70	1.54/0.62
Коэффициент пиковой памяти	ogbn-ariv	twitch-views	artnet-ep	pokec-regions	tolokers-2	city-roads-L
Наш	4.67/3.46	44.17/31.71	6.90/5.34	12.74/8.47	32.59/28.55	1.77/1.51
PyG	0.58/0.69	0.61/0.68	0.59/0.72	OOM/OOM	0.61/0.71	0.54/0.72
Ускорение ($H = 8$)	ogbn-ariv	city-reviews	artnet-ep	city-roads-M	tolokers-2	city-roads-L
Наш	3.11/1.16	4.96/1.48	4.73/1.83	2.19/1.38	9.78/2.18	2.01/1.34
PyG	0.46/0.54	0.31/0.40	0.36/0.47	0.42/0.59	0.35/0.44	0.42/0.58
Память ($H = 8$)	ogbn-ariv	city-reviews	artnet-ep	city-roads-M	tolokers-2	city-roads-L
Наш	6.73/4.11	14.09/8.62	10.38/6.37	2.72/1.76	69.74/44.80	2.37/1.53
PyG	0.55/0.64	0.58/0.66	0.57/0.66	0.49/0.62	0.60/0.67	0.48/0.61

Табл. 7. Результаты для граф-трансформера (головы $H = 4$, размерность головы $D = 128$). N/A – DF-GNN столкнулся с ошибкой незаконного доступа к памяти; выше лучше.
Table 7. Results for the graph-transformer (heads $H = 4$, head dimension $D = 128$). N/A – DF-GNN encountered an illegal memory access error; higher is better.

Реализация	ogbn-arxiv	artnet-exp	city-roads-M	pokec-regions	tolokers-2	city-roads-L
Ускорение задержки (t_{DGL}/t), прямой/обратный						
Наш (CSR fused)	1.27/1.24	1.17/1.15	1.15/1.19	1.25/1.15	2.00/1.10	1.15/1.16
DF-GNN	1.62/1.19	1.19/1.21	1.12/1.20	N/A	N/A	1.13/1.18
Наш (WSB/TC)	1.56/0.77	1.29/0.78	1.16/1.07	1.50/0.70	2.69/0.71	1.15/1.03
Коэффициент пиковой памяти ($\mathrm{mem}_{\mathrm{DGL}}/\mathrm{mem}$), прямой/обратный						
Наш (CSR fused)	1.19/1.01	1.22/1.02	1.17/1.00	1.27/1.04	1.57/1.13	1.16/1.00
DF-GNN	1.18/1.14	1.19/1.08	1.16/1.08	N/A	N/A	1.16/1.13
Наш (WSB/TC)	1.08/1.21	1.08/1.21	1.08/1.21	1.10/1.21	1.23/1.20	1.07/1.21

Табл. 8. Ускорение распределения групп потоков с учётом степеней над ядром внимания без учёта степеней (GT и GATv2, размерность головы $= 512$, $H = 4$), прямой/обратный.
Table 8. Speedup of warp distribution with respect to degrees over the attention core without respect to degrees (GT and GATv2, head dimension $= 512$, $H = 4$), forward/backward.

Набор данных	Асимметрия	GT пр.	GT обр.	GATv2 пр.	GATv2 обр.
web-traffic	738.1	5.83x	6.34x	5.45x	7.34x
ogbn-arxiv	110.8	3.72x	1.09x	3.91x	2.49x
pokec-regions	71.8	1.07x	1.02x	1.13x	1.10x
twitch-views	42.4	2.10x	1.45x	2.23x	2.08x
city-reviews	35.0	2.07x	1.40x	2.24x	1.78x
ogbn-products	17.5	1.07x	1.02x	1.19x	1.11x
avazu-ctr	9.4	2.48x	1.37x	2.30x	2.04x
hm-categories	7.3	1.79x	1.26x	1.62x	1.65x

Табл. 9. Ускорение обратного прохода ядер для неориентированных графов над направленным базовым (размерность головы $= 128$, $H = 4$). GT (CUDA) и GATv2 (CUDA) используют CSR-слитые ядра; GT (WSB) – путь с блочно-разреженными тензорными ядрами Tensor Cores. Графы преобразованы в неориентированные.
Table 9. Speedup of backward kernel traversal for undirected graphs over a directed baseline (head dimension $= 128$, $H = 4$). GT (CUDA) and GATv2 (CUDA) use CSR-fused kernels; GT (WSB) uses block-sparse Tensor Cores. Graphs are converted to undirected.

Набор данных	Ср. ст.	GT (CUDA)	GATv2 (CUDA)	GT (WSB)
avazu-ctr	289	1.21x	1.12x	8.59x
hm-categories	461	1.19x	1.08x	8.16x
tolokers-2	89	1.01x	1.05x	7.57x
twitch-views	81	1.03x	1.01x	9.36x
ogbn-products	51	1.11x	1.14x	7.03x
pokec-regions	28	1.11x	0.97x	6.76x
ogbn-arxiv	14	1.13x	1.09x	7.73x
web-traffic	9	1.06x	1.21x	10.74x
city-roads-L	4	1.10x	1.01x	2.75x

Стоимость препроцессинга WSB и амортизация построения WSB. Построение WSB требует разбиения целевых вершин на окна строк и группировки их окрестностей в TCB, что влечёт единовременную стоимость препроцессинга на структуру графа. Эта стоимость амортизируется по итерациям обучения; WSB-представление может быть сериализовано на диск для повторного использования. В табл. 10 приведены время построения WSB, задержка обучения за эпоху и число эпох для окупаемости. На локально плотных графах – hm-prices (окупаемость: 65 эпох) и avazu-ctr (окупаемость: 159 эпох) – WSB амортизируется быстро. На 9 из 16 графов, однако, CSR-слитая реализация быстрее за эпоху, а точка окупаемости превышает 1000 эпох.

Табл. 10. Амортизация WSB (GT, 4 слоя, размерность головы $= 128$, $H = 4$). Точка окупаемости – число эпох обучения, необходимых для компенсации единовременной стоимости построения WSB.
Table 10. WSB amortization (GT, 4 layers, head dimension $= 128$, $H = 4$). The breakeven point is the number of training epochs required to offset the one-time cost of constructing the WSB.

Набор данных	Стоимость WSB (с)	DGL (мс/эп)	WSB (мс/эп)	Ускорение	Точка окупаемости (эп.)
hm-prices	16.1	731.2	484.4	1.51x	65
avazu-ctr	14.9	825.5	731.9	1.13x	159
city-roads-L	4.4	296.3	285.5	1.04x	404
city-roads-M	4.3	145.5	139.4	1.04x	693

4.3 Перестановка вершин графа: когда улучшение локальности помогает

Для количественной оценки условий, при которых перестановка помогает, используется чувствительный к кэшу контрольный тест (dot-aggr) с двумя реализациями, имеющими идентичную арифметику, но разные шаблоны доступа к памяти: (i) feature-parallel (слитые загрузки по признакам, соседи сканируются последовательно) и (ii) neighbor-parallel (gather-подобные загрузки по соседям, каждая дорожка обрабатывает признаки разного соседа). *Разрыв перестановки* $\Delta s = s_{\text{neighbor}} - s_{\text{feature}}$, где $s = t_{\text{orig}}/t_{\text{reordered}}$, показывает, какое отображение больше выигрывает от перестановки; положительное Δs означает, что чувствительное к gather отображение выигрывает больше.

Рис. 1 демонстрирует чёткую зависимость от степени вершин. На ogbn-products (ср. степень ≈50) Δs стабильно и значительно положителен: много соседей на вершину означает много одновременных gather-обращений, и упорядоченное размещение превращает разбросанные HBM-чтения в попадания в тёплый кэш. На разреженных графах – city-roads-L (ср. степень ≈2) и web-traffic (ср. степень ≈4) – Δs близко к нулю и в нескольких конфигурациях слегка отрицательно: низкая степень вершин означает небольшой рабочий набор на узел, помещающийся в кэш независимо от порядка, что не оставляет пространства для улучшения локальности перестановкой.

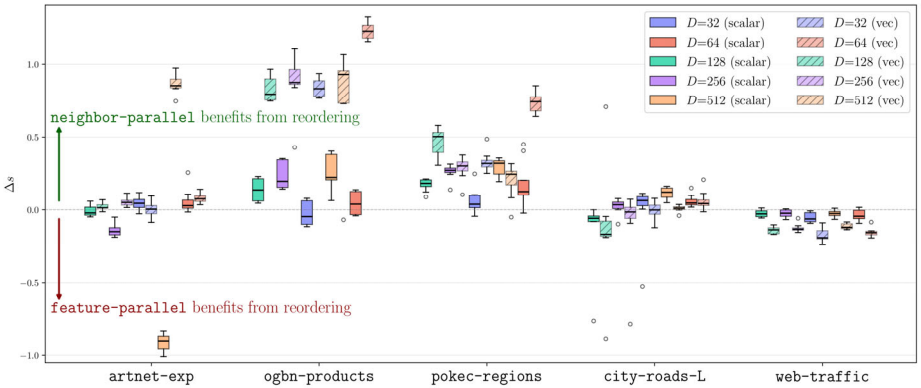

Рис. 1. Разрыв перестановки Δs на выбранных наборах данных (dot-aggr). $\Delta s > 0$ означает, что перестановка больше помогает neighbor-parallel (gather). Сравниваются скалярные загрузки против 16-байтных векторизованных (vec); прямоугольники агрегируют по размерам разбиений.
Fig. 1. Permutation gap Δs on the selected datasets (dot-aggr). $\Delta s > 0$ means that permutation is more beneficial than neighbor-parallel (gather). Scalar loadings are compared against 16-byte vectorized loadings (vec); rectangles are aggregated by partition sizes.

5. Заключение

В данной работе мы изучили производительность GNN с точки зрения операций ввода-вывода и арифметической интенсивности; мы показали, что широко используемые слои группируются в три семейства ядер с различными узкими местами: свёртки на основе SpMM, агрегации на основе редукции и слои на основе внимания. На основе этой таксономии разработаны GPU-реализации, снижающие избыточный трафик HBM, улучшающие локальность и остающиеся устойчивыми на реалистичных графах. В частности, слитые ядра внимания исключают материализацию промежуточных тензоров по рёбрам и существенно снижают пиковое потребление памяти активациями, тогда как плиточное разбиение с учётом степеней улучшает производительность операторов редукции при тяжёлых хвостах распределения степеней. Для слоёв на основе SpMM вендорские примитивы оказались сильным базовым решением: с кэшированием структуры графа и транспонированной матрицы cuSPARSE соответствует или превосходит специализированные системы в нашей постановке. Блочно-разреженный путь с тензорными ядрами Tensor Cores улучшает прямую производительность на локально плотных графах, однако для ориентированных графов обратный проход может быть менее эффективным из-за scatter/атомарного накопления. Наконец, показано, что влияние перестановки вершин графа не является универсальным: оно зависит от стратегии параллелизации ядра и эффективного рабочего набора, при этом чувствительные к gather отображения с параллелизмом по соседям выигрывают более стабильно.

Путь WSB/TCB обменивает нерегулярные разреженные вычисления на плотные умножения плиток 16×16; выигрыш критически зависит от эффективной плотности каждой плитки. При очень разреженных окнах строк ядро всё равно выполняет умножение $QK^{\top}$ для многих замаскированных элементов, что увеличивает расход FLOPs и нередко возвращает ядро в режим, ограниченный памятью или накладными расходами. WSB также несёт накладные расходы препроцессинга и хранения, амортизируемые по итерациям обучения. Обратный проход менее благоприятен, чем прямой: поскольку одни и те же столбцы исходных вершин могут встречаться в нескольких окнах строк, обновления dK/dV нередко требуют атомарного накопления, что может доминировать во времени выполнения.

Несколько перспективных направлений остаются для будущих исследований. Адаптивный автонастройщик, выбирающий между реализациями на основе статистики графа, сделает производительность более предсказуемой. Программный конвейер CUDA и специализация групп потоков предлагают естественный следующий шаг: этот шаблон, показанный для внимания в FlashAttention-3 [44], принесёт пользу ядрам редукции и SpMM. Балансировка нагрузки при тяжёлых хвостах распределения степеней – дополнительная задача: статическое упорядочение по длительнейшему-времени-обработки (как в FlashAttention-4 [45]) совместно с персистентными ядрами [46-47] может снизить простой SM. Наконец, масштабирование на распределённое многогпу-выполнение – естественное расширение для графов, превышающих память одного устройства.

Список литературы / References

[1]. Gilmer J. et al. Neural Message Passing for Quantum Chemistry, 2017.
[2]. Kimura M. et al. On permutation-invariant neural networks, 2024.
[3]. Peng H. et al. MaxK-GNN: Extremely Fast GPU Kernel Design for Accelerating Graph Neural Networks Training. Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024), 2024.
[4]. Zhang H. et al. Understanding GNN Computational Graph: A Coordinated Computation, IO, and Memory Perspective, 2021.
[5]. Kipf T. N., Welling M. Semi-Supervised Classification with Graph Convolutional Networks. International Conference on Learning Representations (ICLR), 2017.

[6]. Hamilton W. L., Ying R., Leskovec J. Inductive Representation Learning on Large Graphs. Advances in Neural Information Processing Systems (NeurIPS), 2017.
[7]. Min E. et al. Transformer for graphs: An overview from architecture perspective. arXiv preprint, 2022. DOI: 10.48550/arXiv.2202.08455.
[8]. Veličković P. et al. Graph Attention Networks. International Conference on Learning Representations (ICLR), 2018.
[9]. Brody S., Alon U., Yahav E. How Attentive are Graph Attention Networks? 2022.
[10]. Shi Y. et al. Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification, 2021.
[11]. Vaswani A. et al. Attention is all you need. Advances in Neural Information Processing Systems, 2017. vol. 30.
[12]. Ying C. et al. Do Transformers Really Perform Bad for Graph Representation? Advances in Neural Information Processing Systems, 2021, vol. 34, pp. 28877-28888.
[13]. Rampášek L. et al. Recipe for a General, Powerful, Scalable Graph Transformer. Advances in Neural Information Processing Systems, 2022, vol. 35, pp. 14501-14515.
[14]. NVIDIA. CUDA Programming Guide. Available at: https://docs.nvidia.com/cuda/cuda-programming-guide/, accessed 25.08.2026.
[15]. Williams S., Waterman A., Patterson D. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM. Association for Computing Machinery, 2009, vol. 52, no. 4, pp. 65-76.
[16]. Wei N. et al. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. Association for Computing Machinery, 2021, pp. 883-898.
[17]. Chen T. et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning, 2018.
[18]. Wang M. et al. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. arXiv preprint, 2019. DOI: 10.48550/arXiv.1909.01315.
[19]. Matthias F. et al. PyG 2.0: Scalable Learning on Real World Graphs. Temporal Graph Learning Workshop @ KDD, 2025.
[20]. Huang G. et al. GE-SpMM: General-purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks, 2020.
[21]. Hong C. et al. Adaptive sparse tiling for sparse matrix multiplication, 2019, pp. 300-314.
[22]. Reguly I. Z., Giles M. Efficient sparse matrix-vector multiplication on cache-based GPUs, 2012, pp. 1-12.
[23]. Li Z., Chandramowlishwaran A. Fused3S: Fast Sparse Attention on Tensor Cores, 2025.
[24]. Chen Z. et al. fuseGNN: accelerating graph convolutional neural network training on GPGPU. Proceedings of the 39th International Conference on Computer-Aided Design. Association for Computing Machinery, 2020.
[25]. Liu J. et al. DF-GNN: Dynamic Fusion Framework for Attention Graph Neural Networks on GPUs, 2024.
[26]. Wang Y. et al. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), 2021, pp. 515-531.
[27]. Bazinska J. et al. Cached Operator Reordering: A Unified View for Fast GNN Training, 2023.
[28]. Wang Y. et al. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs, 2023.
[29]. Xiang L. et al. cuTeSpMM: Accelerating Sparse-Dense Matrix Multiplication using GPU Tensor Cores, 2025.
[30]. Yan J. et al. RT-GNN: Accelerating Sparse Graph Neural Networks by Tensor-CUDA Kernel Fusion. ACM Trans. Archit. Code Optim. Association for Computing Machinery, 2025, vol. 22, no. 1.
[31]. Karypis G., Kumar V. METIS–A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Ordering of Sparse Matrices, 1997.
[32]. Merkel N. et al. Can Graph Reordering Speed Up Graph Neural Network Training? An Experimental Study, 2024.
[33]. Li M. et al. The Deep Learning Compiler: A Comprehensive Survey. IEEE Transactions on Parallel and Distributed Systems. Institute of Electrical; Electronics Engineers (IEEE), 2021, vol. 32, no. 3, pp. 708-727.
[34]. Dao T. et al. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, 2022.
[35]. Paszke A. et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019.
[36]. Albert R., Barabási A.-L. Statistical mechanics of complex networks. Rev. Mod. Phys. American Physical Society, 2002, vol. 74, pp. 47-97.

[37]. Newman M. E. J. The Structure and Function of Complex Networks. SIAM Review. 2003, vol. 45, no. 2, pp. 167-256.
[38]. Milakov M., Gimelshein N. Online normalizer calculation for softmax, 2018.
[39]. Dao T. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning, 2023.
[40]. Bazhenov G., Platonov O., Prokhorenkova L. GraphLand: Evaluating graph machine learning models on diverse industrial data // Advances in Neural Information Processing Systems, 2025.
[41]. Hu W. et al. Open Graph Benchmark: Datasets for Machine Learning on Graphs, 2021.
[42]. Yang Z., Cohen W., Salakhudinov R. Revisiting semi-supervised learning with graph embeddings. International Conference on Machine Learning, 2016, pp. 40-48.
[43]. Fomina D. et al. On Efficient Scaling of GNNs via IO-Aware Layers Implementations, 2026.
[44]. Shah J. et al. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision, 2024.
[45]. Zadouri T. et al. FlashAttention-4: Algorithm and Kernel Pipelining Co-Design for Asymmetric Hardware Scaling, 2026.
[46]. Gupta K., Stuart J., Owens J. D. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. Innovative Parallel Computing (InPar). IEEE, 2012.
[47]. Stankovic A. AutoSAGE: Input-Aware CUDA Scheduling for Sparse GNN Aggregation (SpMM/SDDMM) and CSR Attention, 2025.

Информация об авторах / Information about authors

Фёдор Сергеевич ВЕЛИКОНИВЦЕВ – аспирант, младший научный сотрудник научно-учебной лаборатории компании Яндекс факультета компьютерных наук НИУ ВШЭ. Сфера научных интересов: машинное обучение на графах, методы эффективного глубокого обучения, программирование на графических ускорителях.

Fedor Sergeevich VELIKONIVTSEV – postgraduate student, Junior Research Fellow at the Yandex Research and Education Laboratory, Faculty of Computer Science, HSE University. His research interests include graph machine learning, efficient deep learning methods, and GPU programming.