Санкт-Петербургский государственный университет

Гарбар Кирилл Анатольевич

Выпускная квалификационная работа

Библиотека операций разреженной линейной алгебры с использованием Brahma.FSharp

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5162.2020 «Технологии программирования»*

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., С. В. Григорьев

Рецензент:
Инженер-разработчик графики, ООО «ФАРВАТЕР», Е. С. Орачев

Санкт-Петербург
2024

Saint Petersburg State University

Kirill Garbar

Bachelor's Thesis

Sparse linear algebra operation library using Brahma.FSharp

Education level: bachelor

Speciality *02.03.03 «Software and Administration of Information Systems»*

Programme *CB.5162.2020 «Programming Technologies»*

Profile: *System Programming*

Scientific supervisor:
C.Sc., docent S.V. Grigorev

Reviewer:
Engineer-developer of graphics at "ФАРВАТЕР", E.S. Orachev

Saint Petersburg
2024

Оглавление

Введение

Одной из самых известных структур данных информатики является граф. Графы активно применяются в рекомендательных системах и СУБД, а также в разных науках: биологии [17], физике [9] и химии [3], как и в математике — теории групп [2], топологии [20], теории вероятностей [4]. Следует отметить, что алгоритмы анализа графов получили широкое распространение в анализе социальных сетей [15].

Классическое представление графа в виде множеств вершин и дуг не всегда удобно для программирования алгоритмов анализа графов, поэтому граф часто представляется иным образом. Одним из способов представить граф в памяти машины является матрица смежности — квадратная матрица размера $N \times N$, где N — количество вершин графа. Ячейка (i, j) матрицы отвечает за наличие дуги между вершинами i и j и имеет значение 0 (дуги нет) или 1 (дуга есть). Алгоритмы обработки графов, представленных в виде матриц смежности, создаются на языке линейной алгебры и широко освещены в литературе [10].

Граф, имеющий небольшое количество дуг относительно количества вершин, разумно представлять в виде разреженной, а не плотной матрицы. Разреженная матрица — это матрица с преимущественно нулевыми элементами. Одним из наиболее активно используемых форматов разреженных матриц является CSR-формат[1], сжимающий хранимые строковые индексы. Помимо CSR-формата используется симметричный ему CSC-формат, а также координатный формат, хранящий индексы и значения элементов в виде троек.

Развитие идей о связи графов и разреженной линейной алгебры послужило толчком к созданию стандарта GraphBLAS [16], определяющего алгоритмы обработки графов на языке линейной алгебры с использованием разреженных матриц и векторов, заданных над полукольцами.

Стандарт GraphBLAS обладает некоторыми проблемами и ограничениями, активно обсуждаемыми в сообществе [7]. Примером таких про-

[1]Описание форматов разреженных матриц — https://en.wikipedia.org/wiki/Sparse_matrix (дата обращения: 1.05.2024)

блем могут служить явные и неявные нули[2].

Кроме того, подавляющее большинство библиотек, предоставляющих операции линейной алгебры для алгоритмов анализа графов, написано на низкоуровневых C/C++, что может быть не всегда удобно. Есть несколько причин для разработки на C/C++. Программы на этих языках отличаются более высокой производительностью, что критически важно при обработке больших разреженных графов. Также, многие такие библиотеки используют совместимые с C/C++ фреймворки OpenCL и CUDA для вычислений на графических процессорах. Чтобы использовать такие библиотеки было удобнее, создаются библиотеки-обёртки на языках более высокого уровня, таких как Python[3], однако разработка используемых библиотек всё ещё осуществляется на C/C++.

Библиотеки, основанные на OpenCL и CUDA, устроены таким образом, что большая часть интенсивных вычислений происходит на графическом процессоре, в то время как остальной код направлен на отправку задач и ожидание результата от вычисляющего устройства. Поэтому можно предположить, что библиотека, написанная на языке более высокого уровня, но также полагающаяся на OpenCL или CUDA, не должна уступать в производительности библиотекам на C/C++, но будет обладать всеми преимуществами разработки на языке высокого уровня. Вдобавок, такая библиотека потенциально может использовать индивидуальные особенности языка для решения существующих на других платформах проблем, например с применением функциональной парадигмы программирования.

Существует проект под названием GraphBLAS-sharp[4], вдохновлённый стандартом GraphBLAS, в рамках которого реализуются операции линейной алгебры для алгоритмов анализа графов. GraphBLAS-sharp основывается на библиотеке Brahma.FSharp, осуществляющей трансля-

[2]Обсуждение проблемы явных нулей с одним из основоположников стандарта GraphBLAS — https://github.com/GraphBLAS/LAGraph/issues/28 (дата обращения: 1.05.2024)

[3]Сайт проекта PyOpenCL — https://pypi.org/project/pyopencl (дата обращения: 1.05.2024)

[4]Репозиторий проекта GraphBLAS-sharp — https://github.com/YaccConstructor/GraphBLAS-sharp (дата обращения: 1.05.2024)

цию кода на языке F# в OpenCL C.

На данный момент в GraphBLAS-sharp уже имеется значительная часть базовых операций, таких как сортировка или префиксная сумма, поэтому существует возможность убедиться в работоспособности проекта, реализовав с их помощью операции линейной алгебры и некоторые алгоритмы анализа графов. Помимо этого, предыдущие результаты экспериментального исследования имеющихся операций показали низкую производительность некоторых из алгоритмов, по этой причине существует необходимость выяснить причины низкой производительности и по возможности их устранить.

1. Постановка задачи

Целью работы является разработка библиотеки операций разреженной линейной алгебры на платформе .NET с использованием транслятора в OpenCL и экспериментальное исследование полученных результатов. Для выполнения этой цели были поставлены следующие задачи.

1. Доработать существующие операции в GraphBLAS-sharp и реализовать перечисленные ниже алгебраические операции.
    - Поэлементное сложение разреженных матриц.
    - Умножение разреженной матрицы на плотный и разреженный вектор.
2. С помощью разработанной библиотеки реализовать некоторые алгоритмы анализа графов, такие как обход графа в ширину и ранжирование веб-страниц с помощью PageRank.
3. Провести исследование производительности библиотеки и реализованных алгоритмов анализа графов, осуществить сравнение с аналогичными библиотеками на различных платформах.

2. Обзор

Обзор используемых в работе технологий и библиотек, некоторых из существующих аналогичных проектов, а также операций линейной алгебры и алгоритмов анализа графов, касающихся данной работы, представлен в этом разделе.

2.1. Вычисления на графическом процессоре

Помимо решения задач компьютерной графики, видеокарты смогли хорошо себя зарекомендовать в области обобщённых вычислений. Техника вычислений общего назначения на видеокартах получила название GPGPU computing — General-purpose computing on graphics processing units. Популярность видеокарт в области высокоинтенсивных параллельных вычислений обусловлена их архитектурой.

2.1.1. Архитектура графического процессора

Современный графический процессор состоит из вычислительных единиц, называемых Streaming Multiprocessor (SM), на каждом из которых в стиле SIMD(Single instruction, multiple data) или SPMD(Single program, multiple data) исполняется множество потоков. Потоки на одном SM могут коммуницировать между собой с помощью быстрой разделяемой памяти. Схема архитектуры типичного графического процессора описана на рисунке 1.

Программы, исполняемые на видеокарте, называются ядрами. Ядро состоит из последовательности инструкций, описывающих работу одного потока. Как правило, графические процессоры работают совместно с центральным процессором, на котором ядра подготавливаются и отправляются для исполнения на видеокарту.

2.1.2. Вычисления с помощью OpenCL

На данный момент существует несколько фреймворков для работы с графическими процессорами. Одним из таких фреймворков является

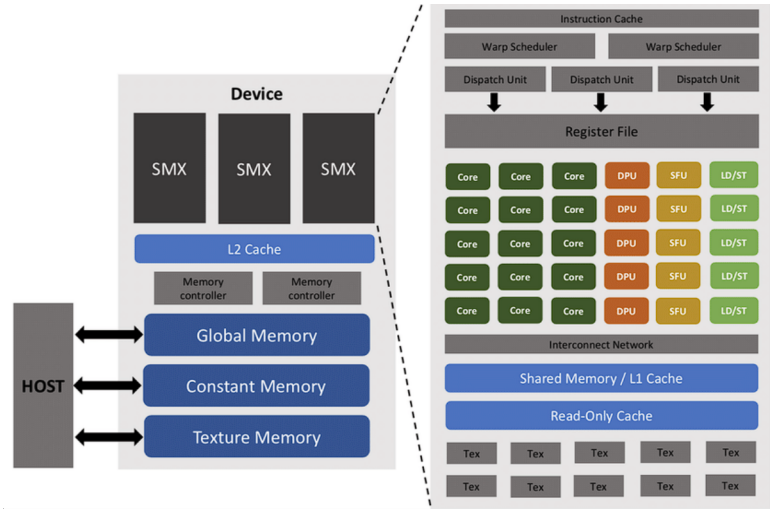

Рис. 1: Архитектура типичного графического процессора. Источник: https://mavink.com/explore/GPU-Nodes

OpenCL, предоставляющий программный интерфейс для языка C.

Пользователь пишет ядра на расширении языка C, управляет выделением ресурсов, таких как память, и отправляет ядра для исполнения на графический или центральный процессор, поддерживающий OpenCL. Исполнение ядер синхронизируется с помощью создаваемых пользователем очередей событий. В самом простом случае ядра исполняются в порядке их запуска.

Ключевой особенностью OpenCL выступает кросс-платформенность. Производители AMD, NVIDIA и Intel поддерживают OpenCL на своих устройствах, чем не могут похвастаться такие фреймворки как NVIDIA CUDA.

2.2. Библиотеки для работы с графическими процессорами

На платформе .NET существует множество инструментов для работы с графическими процессорами. Такие проекты как OpenCL.NET[5] являются лишь обёртками для OpenCL, поэтому код, исполняемый на устройстве, всё ещё должен быть написан на расширении языка C.

[5]Репозиторий OpenCL.NET — https://github.com/cass-support/opencl.net (дата обращения: 1.05.2024)

Помимо обыкновенных обёрток для OpenCL существуют проекты, позволяющие писать весь код на высокоуровневых языках платформы .NET. Библиотеки GPU.NET[6] и Cudafy [11] преобразуют код на языке C# в ядра для CUDA, которые исполняются на графических процессорах NVIDIA. Недостатки этих проектов в том, что они не поддерживают устройства от AMD и Intel. Для работы с OpenCL существует Brahma[7], использующая синтаксис LINQ для написания программ, однако разработка Brahma, GPU.NET и Cudafy была прекращена.

К разрабатываемым по сей день проектам можно отнести ILGPU[8], поддерживающий трансляцию из C# в CUDA или OpenCL, а также Brahma.FSharp — транслирующий функции на языке F# в ядра OpenCL.

Кроссплатформенность OpenCL вместе с потенциальной возможностью внедрить некоторые особенности языка F# в реализацию библиотеки стали основными причинами выбора Brahma.FSharp в качестве инструмента для реализации библиотеки.

2.3. Форматы представления матриц и векторов

Чтобы эффективно работать с матрицами и векторами, они должны быть представлены в памяти определённым образом. Самый простой способ представить плотные матрицы и векторы — в виде двумерного и одномерного массива соответственно.

Разреженный вектор как правило представляется в координатном формате. Такой вектор содержится в двух массивах. В первом хранятся индексы ненулевых элементов вектора, а во втором соответствующие значения этих элементов.

Форматы представления разреженных матриц разнообразнее [14]. Самым простым является координатный (COO) формат. Матрица в COO-формате представляется в виде трёх массивов. Первые два хра-

[6]Обзор GPU.NET — https://www.itwriting.com/blog/4702-gpu-programming-for-net-tidepowerds-gpu-net-gets-some-improvements-more-needed.html (дата обращения: 1.05.2024)

[7]Репозиторий Brahma — https://github.com/aquavit/Brahma (дата обращения: 1.05.2024)

[8]Главная страница проекта ILGPU — https://ilgpu.net/ (дата обращения: 1.05.2024)

нят индекс строки и столбца элемента, а третий — его значение.

На практике чаще применяется compressed sparse row (CSR) формат. Отличия CSR-формата от COO в массиве, отвечающем за строки. Формат CSR сжимает этот массив, храня по индексу i количество ненулевых элементов в строках с номером меньше i. Аналогично устроены compressed sparse column (CSC) матрицы.

2.4. Операции линейной алгебры

В данном разделе описаны операции разреженной линейной алгебры, которые будут реализованы в результате этой работы, а также некоторые вспомогательные операции, необходимые для реализации алгебраических операций.

2.4.1. Поэлементное сложение и умножение матриц

Результатом сложения двух матриц одинакового размера является третья матрица, в ячейках которой лежит результат применения скалярной операции к двум соответствующим элементам складываемых матриц. Операции поэлементного сложения и умножения матриц отличаются в том, что результатом сложения ненулевого элемента с нулевым при умножении может быть только нулевой элемент, а при сложении — ненулевой. Далее обе эти операции будут называться сложением матриц.

В случае, если матрицы разреженные, они хранятся в одном из специальных форматов. Самым часто используемым является формат CSR. Сложение CSR-матриц происходит в три этапа.

Сперва происходит построчное слияние соответствующих массивов столбцов. Эффективным алгоритмом для параллельного слияния массивов является MergePath [8]. В результате слияния соответствующие элементы двух матриц оказываются в двух соседних ячейках массива.

После этого соответствующие элементы складываются и результат сохраняется в новый массив на позицию, соответствующую одному из двух элементов в сложении. Такой массив содержит много пробелов

между значениями.

На третьем этапе создаётся результирующий массив, в который кладутся все ненулевые элементы предыдущего массива.

2.4.2. Умножение матрицы на плотный вектор

Одной из наиболее используемых операций линейной алгебры является умножение разреженной матрицы на плотный вектор. Описываемый алгоритм представлен в литературе [6]. Алгоритм включает в себя два этапа.

1. Умножение всех значений матрицы на соответствующие им значения из вектора.
2. Построчное сложение полученных на предыдущем этапе значений.

2.4.3. Умножение матрицы на разреженный вектор

Если вектор хранит небольшое количество ненулевых элементов относительно своего размера, для экономии ресурсов можно использовать алгоритм умножения на разреженный вектор [19]. Алгоритм включает в себя следующие этапы.

1. Отбор строк матрицы, которые будут участвовать в умножении.
2. Сортировка полученных элементов матрицы по столбцам.
3. Умножение элементов матрицы на соответствующий элемент вектора.
4. Суммирование полученных после умножения значений для каждого столбца.

2.4.4. Параллельная префиксная сумма массива

Для реализации операций линейной алгебры требуются базовые операции над массивами, такие как префиксная сумма. Результатом операции префиксной суммы массива служит массив, хранящий в ячейке

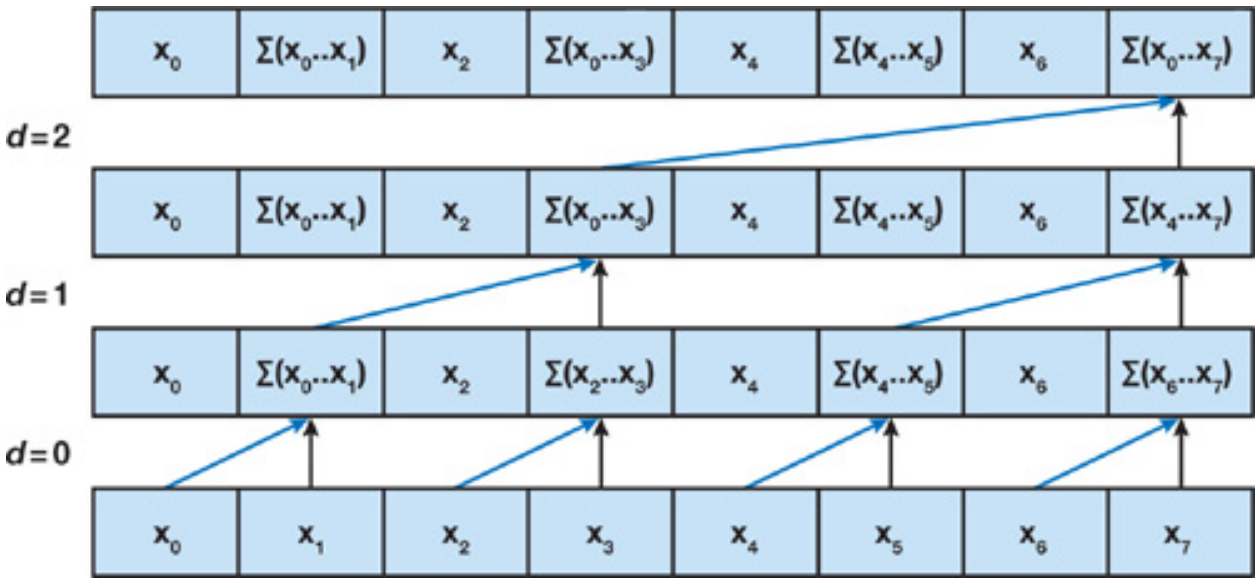

Рис. 2: Первая фаза алгоритма префиксной суммы. Источник: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

i сумму элементов исходного массива с номерами меньше i. В данной работае используется параллельная версия алгоритма [12], специально предназначенная для графических процессоров.

На первом этапе алгоритма считаются частичные суммы элементов массива, как показано на рисунке 2. Эту фазу также называют фазой редуцирования массива, потому что последняя ячейка хранит сумму всех его элементов. После этого в каждую ячейку массива помещается необходимая префиксная сумма, согласно рисунку 3.

2.5. Алгоритмы анализа графов

GraphBLAS-sharp предоставляет набор достаточно обобщённых алгебраических операций. Несмотря на это, проект сильно вдохновлён стандартом GraphBLAS и нацелен на использование имеющихся операций в области анализа графов.

2.5.1. Обход в ширину в терминах линейной алгебры

Обход в ширину это широко известный и простой алгоритм [1]. Задача обхода состоит в том, чтобы посетить каждую вершину графа,

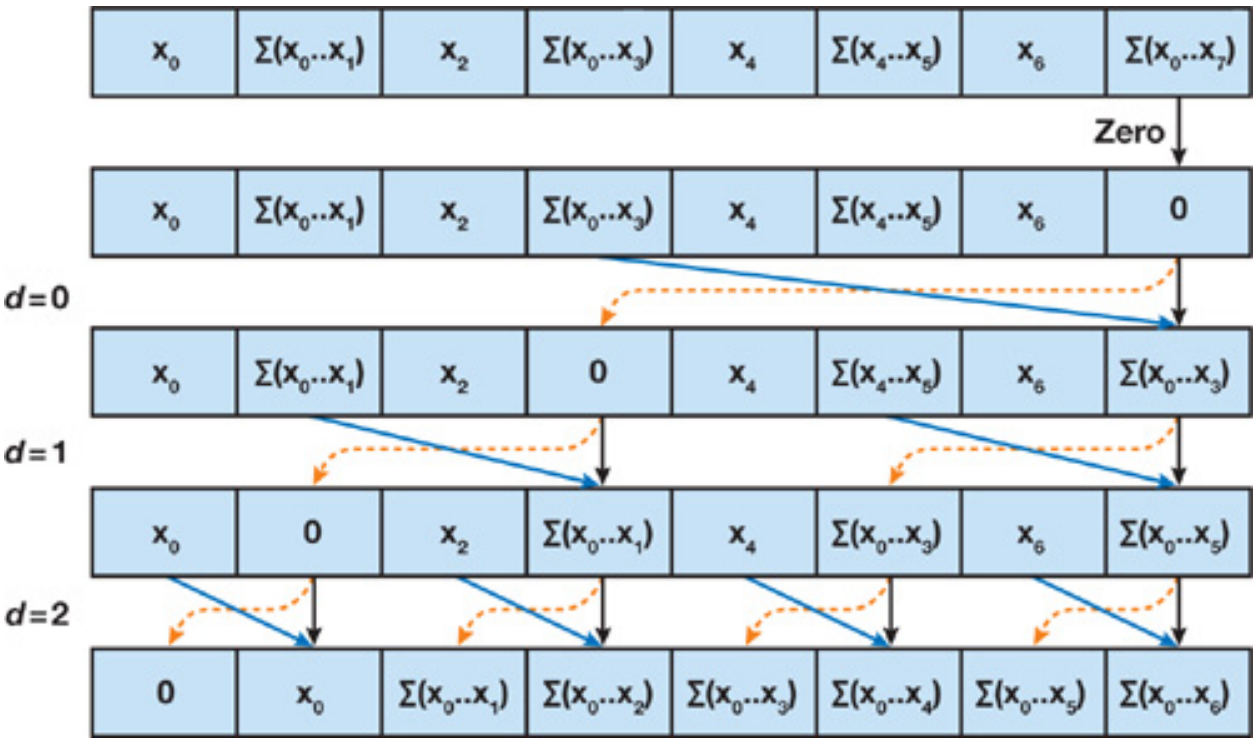

Рис. 3: Вторая фаза алгоритма префиксной суммы. Источник: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

которая достижима из заданной стартовой вершины. Порядок обхода вершин определяется расстоянием вершин от стартовой, выраженным в количестве рёбер. Таким образом, обход осуществляется по уровням или фронтам — множествам вершин, равноудалённым от стартовой.

В терминах линейной алгебры граф можно представить матрицей смежности $A \in \mathbb{B}^{n \times n}$, а фронт — вектором $x \in \mathbb{B}^n$. В качестве операций сложения и умножения возьмём логические OR и AND. В таком случае новый фронт получается из старого путём умножения транспонированной матрицы A^T на вектор x с удалением ранее посещённых вершин. Результатом обхода может служить вектор, хранящий в компоненте с номером i расстояние от источника обхода до вершины с номером i, выраженное в количестве рёбер. Итерация обхода изображена на рисунке 4.

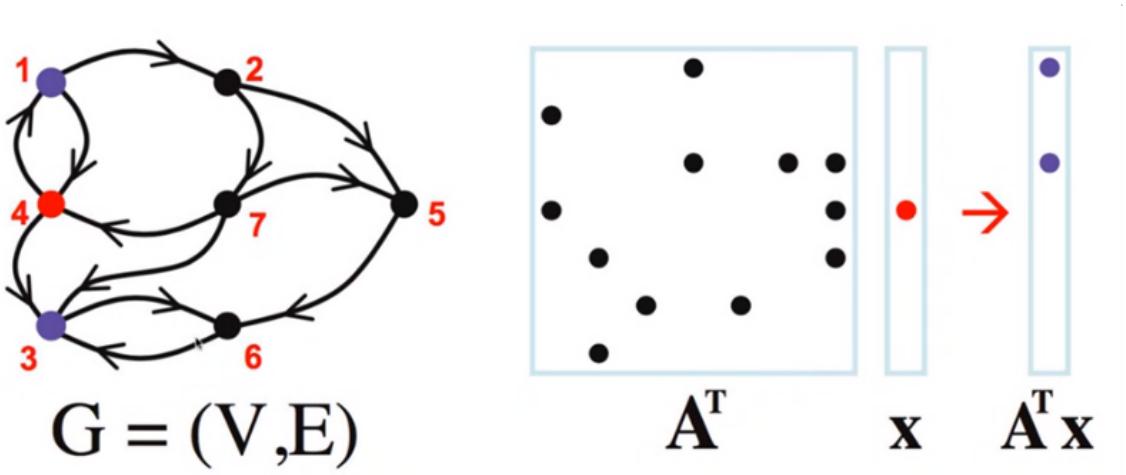

Рис. 4: Итерация обхода в ширину в терминах линейной алгебры. Источник: https://en.wikipedia.org/wiki/GraphBLAS

2.5.2. Ранжирование веб-страниц с помощью PageRank

Задачей алгоритма PageRank является ранжирование веб-страниц по качеству и количеству ведущих на них ссылок.

Идея алгоритма состоит в том, что о важности веб-страницы можно судить по ссылающимся на неё веб-страницам. Таким образом, чем больше ссылок имеется на данную веб-страницу, тем она релевантнее. Кроме того, разные ссылки имеют разный вес, зависящий от ссылающейся веб-страницы.

Формулировка алгоритма в матрично-векторных терминах следующая. Имеется матрица $G \in \mathbb{R}^{n \times n}$, представляющая граф, вершинами которого являются веб-страницы, а дугами — ссылки между страницами. Веса дуг, исходящих из каждой данной вершины одинаковы и в сумме дают единицу. Результатом работы алгоритма служит вектор $rank \in \mathbb{R}^n$, сопоставляющий каждой вершине её важность, выраженную числом. В начале алгоритма важность каждой вершины $1/n$.

На каждой итерации алгоритма все вершины получают вес от ссылающихся на них вершин. На языке линейной алгебры это выражается как умножение матрицы G на вектор $rank$. Так, на первой итерации алгоритма все вершины получают вес от своих прямых соседей, на второй — от соседей своих соседей и так далее. После некоторого количества итераций $rank$ практически перестаёт изменяться и алгоритм останав-

ливается. Чтобы подсчитать, насколько сильно меняется *rank*, можно использовать норму разности векторов. Псевдокод алгоритма изображён на рисунке 1.

Algorithm 1 Ранжирование веб-страниц с помощью PageRank

```
Input: Matrix G, accuracy
n ← G.Size
error ← accuracy
rank, prev ← createVector(G.NumberOfRows, 1/n)

while error ≥ accuracy do
    rank ← mxv(G, prev)
    error ← computeError(rank, prev)
    prev ← rank
end while

return rank
```

2.6. Аналогичные библиотеки

Наличие спроса на операции с разреженными матрицами породило множество реализаций на самых разных платформах. В этом разделе будут рассмотрены некоторые из таких библиотек. Помимо библиотек с матричными операциями будут упомянуты библиотеки анализа графов, так как задачи анализа графов тесно связаны с операциями над матрицами и векторами.

2.6.1. Библиотеки операций линейной алгебры

Упомянутый ранее стандарт GraphBLAS, описывающий строительные блоки для алгоритмов анализа графов в терминах линейной алгебры, породил множество реализаций различной степени строгости и полноты, предоставляющих набор алгебраических операций над матрицами и векторами.

Самой известной и эталонной библиотекой, в наиболее полной мере реализующей стандарт GraphBLAS является

SuiteSparse:GraphBLAS[9] [5].

SuiteSparse поддерживает вычисления только на центральных процессорах, поэтому производительность библиотеки может уступать некоторым её аналогам, работающим с видеокартами. Одна из наиболее высокопроизводительных библиотек, реализующих стандарт, это GraphBLAST [18], работающий на устройствах NVIDIA с помощью CUDA.

Рассмотренные библиотеки достаточно строго реализуют стандарт GraphBLAS. Существуют также библиотеки, не стремящиеся в полной мере реализовать стандарт, но также реализующие операции линейной алгебры для анализа графов. Одной из таких библиотек является Spla [13]. Spla поддерживает высокопроизводительные математические вычисления с помощью OpenCL на устройствах различных производителей.

Алгоритмы анализа графов это не единственная область применения алгебраических операций. Библиотека Math.NET Numerics на платформе .NET предоставляет набор обобщённых математических операций и в том числе операций линейной алгебры. В качестве аналога Math.NET на платформе CUDA можно рассмотреть CUSP[10].

2.6.2. Библиотеки анализа графов

Все рассмотренные ранее библиотеки, имеющие отношение к стандарту GraphBLAS, позволяют работать с графами в терминах абстракций линейной алгебры. Однако это не единственный подход к обработке графов. Существуют библиотеки, работающие с графами в терминах дуг и вершин. Примером такой библиотеки на платформе .NET служит QuikGraph[11], предоставляющий механизмы работы с обобщёнными графовыми структурами.

[9]Описание SuiteSparse и ссылки на связанные материалы — https://people.engr.tamu.edu/davis/suitesparse.html (дата обращения: 1.05.2024)

[10]О библиотеке CUSP —https://cusplibrary.github.io/index.html (дата обращения: 1.05.2024)

[11]Репозиторий проекта QuikGraph — https://github.com/KeRNeLith/QuikGraph (дата обращения: 1.05.2024)

2.6.3. Сравнение библиотек

Чтобы оценить полученное решение, необходимо сравнить его с уже существующими проектами. Рассмотренные ранее проекты работают на различных платформах и устройствах. В таблице 1 приведено сравнение библиотек по платформе и устройствам, на которых они работают.

Поддержку вычислений на графических устройствах от AMD и Intel предоставляют только проекты, основанные на OpenCL. Также заметно, что среди библиотек, связанных со стандартом GraphBLAS преобладают реализации на языках C/C++.

Название	CPU	GPU Nvidia	GPU AMD/Intel	Язык
GraphBLAS-sharp	Да	Да	Да	F#
SuiteSparse	Да	Нет	Нет	C
GraphBLAST	Нет	Да	Нет	C++
Spla	Да	Да	Да	C++
CUSP	Нет	Да	Нет	C++
Math.NET Numerics	Да	Нет	Нет	C#
QuikGraph	Да	Нет	Нет	C#

Таблица 1: Сравнение библиотек операций линейной алгебры и анализа графов по поддержке устройств различных производителей и языку программирования.

3. Архитектура библиотеки

В данном разделе будет рассмотрена архитектура разработанной библиотеки, ее основные компоненты и главный сценарий использования.

3.1. Основные компоненты

GraphBLAS-sharp разбивается на четыре компоненты, как показано на рисунке 5.

Транслятор Brahma.FSharp предоставляет обёртку OpenCL с помощью OpenCL.NET и осуществляет трансляцию ядер из библиотеки операций с языка F# на OpenCL C. Для взаимодействия с очередью команд OpenCL в Brahma.FSharp предусмотрено два интерфейса — асинхронный на основе каналов .NET и синхронный с доступом к очереди напрямую.

Библиотека операций линейной алгебры объединяет типы, представляющие объекты линейной алгебры — матрицы и векторы в различных форматах, алгебраические операции над этими объектами и базовые операции над массивами, такие как сортировка. Для предоставления операций пользователю выделяется отдельный модуль, содержащий обёртки операций библиотеки. Помимо этого в GraphBLAS-sharp также имеются реализованные с помощью библиотеки алгоритмы анализа графов, которые служат примером использования библиотеки.

Для тестирования библиотеки есть отдельный проект — GraphBLAS-sharp.Tests, в котором содержатся тесты для всех базовых операций библиотеки и реализованных алгоритмов анализа графов. В качестве фреймворка для тестов была выбрана Expecto — одна из наиболее популярных библиотек тестирования программ, написанных на F#.

Задачей GraphBLAS-sharp.Benchmarks является выполнение замеров производительности операций и алгоритмов библиотеки с помощью BenchmarkDotNet[12] — популярной библиотеки для бенчмарков на плат-

[12]Обзор инструмента BenchmarkDotNet — https://benchmarkdotnet.org/articles/overview.html (дата обращения: 1.05.2024)

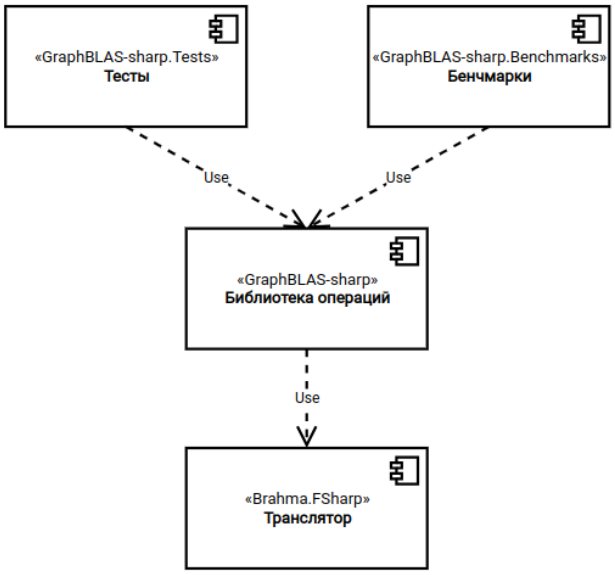

Рис. 5: Архитектура библиотеки GraphBLAS-sharp

форме .NET. GraphBLAS-sharp.Benchmarks содержит сценарии для замеров, конфигурационные файлы и скрипты для автоматического контроля производительности.

3.2. Сценарий использования

Все операции библиотеки реализуются в определённом стиле. Пример реализации библиотечной операции map, применяющей переданную операцию ко всем элементам массива, изображён на листинге 6. Сперва операции из GraphBLAS-sharp необходимо подготовить, передав им параметры, необходимые для трансляции кода в OpenCL, такие как OpenCL-контекст и размер рабочей группы. Некоторым операциям также требуются арифметические бинарные операции или предикаты, в данном случае это op. Для задания функций, которые будут транслироваться в OpenCL, используется инструмент метапрограммирования F# — code quotations[13]. Можно заметить, что переданная в аргументы операция op встраивается в map с помощью оператора %. После

[13]Документация и примеры использования code quotations — https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/code-quotations (дата обращения: 01.05.2024)

```
let mapOp<'a, 'b> (op: Expr<'a -> 'b>) (clContext: ClContext)
↪  workGroupSize =
        // Функция, которая будет транслирована в OpenCL
        let map =
            <@ fun (ndRange: Range1D) length (inputArray: ClArray<'a>) ->
                let gid = ndRange.GlobalID0
                if gid < length then
                    inputArray.[gid] <- (%op) inputArray.[gid] @>
        // Трансляция
        let kernel = clContext.Compile map
        // После передачи в map,'a, 'b> аргументов, будет возвращена эта
        ↪  функция
        fun (processor: RawCommandQueue) (inputArray: ClArray<'a>) ->
            // Получение экземпляра ядра
            let kernel = kernel.GetKernel()
            // Установка аргументов ядра
            let ndRange = Range1D.CreateValid(inputArray.Length,
            ↪  workGroupSize)
            kernel.KernelFunc ndRange inputArray.Length inputArray result
            // Передача ядра в очередь исполнения
            processor.RunKernel kernel
```

Рис. 6: Реализация операции map в GraphBLAS-sharp.

трансляции в OpenCL из mapOp возвращается безымянная функция, содержащяя транслированное в OpenCL ядро. Это делается для того, чтобы возвращаемую функцию можно было переиспользовать, не допуская повторения относительно длительной трансляции. В результате передачи в функцию очереди комманд OpenCL и массива, происходит подготовка ядра к запуску и передача его в очередь на исполнение.

Главным сценарием использования библиотеки является реализация пользователем алгоритмов анализа графов с применением широкого набора библиотечных операций. Пример реализации простейшего алгоритма — обхода графа в ширину — представлен в листинге на рисунке 7. Сперва происходит подготовка операций из GraphBLAS-sharp, как в прошлом примере. Чтобы получить необходимые для алгоритма операции из базовых операций библиотеки, некоторым операциям требуется предикат или бинарная операция. Например, чтобы получить операцию проверки фронта на пустоту, в Vector.exists передается предикат, по которому произойдёт поиск нунелувого элемента в массива.

В результате получается функция containsNonZero, которая содержит готовое к исполнению OpenCL ядро и ожидает в качестве параметров очередь команд OpenCL и вектор. Когда операции готовы, можно приступать к реализации алгоритма. Сперва происходит инициализация необходимых векторов для обхода в ширину, а потом итерации обхода повторяются до тех пор, пока фронт не пустой. Результатом обхода является вектор чисел, соответствующих уровням вершин графа при обходе.

```
let BFS
    (clContext: ClContext)
    (wokrGroupSize: int)
    (queue: RawCommandQueue)
    (matrix: ClMatrix<bool>)
    (source: int) =
    // Подготовка операций GrapBLAS-sharp
    let spMVInPlace =
        Operations.SpMVInPlace add mul clContext wokrGroupSize
    let zeroCreate =
        Vector.zeroCreate clContext wokrGroupSize
    let ofList = Vector.ofList clContext wokrGroupSize
    let maskComplementedInPlace =
        Vector.map2InPlace Mask.complementedOp clContext wokrGroupSize
    let fillSubVectorTo =
        Vector.assignByMaskInPlace Mask.assign clContext wokrGroupSize
    let containsNonZero =
        Vector.exists Predicates.isSome clContext wokrGroupSize

    // Реализация алгоритма
    // Создание результирующего массива
    let levels =
        zeroCreate queue DeviceOnly matrix.RowCount Dense
    // Инициализация фронта обхода
    let front =
        ofList queue DeviceOnly Dense matrix.RowCount [ source, true ]
    let mutable level = 0
    let mutable stop = false

    while not stop do
        level <- level + 1
        // Заполнение levels значением level по маске front
        fillSubVectorTo queue levels front level
        // Получение фронта следующей итерации
        spMVInPlace queue matrix front
        // Удаление посещённых вершин
        maskComplementedInPlace queue front levels

        stop <-
            not
            <| (containsNonZero queue front).ToHostAndFree queue
    levels.Free()
    return levels
```

Рис. 7: Реализация обхода в ширину с помощью GraphBLAS-sharp на языке F#.

4. Реализация и оптимизации

В этом разделе будут описаны детали реализации алгебраических операций и алгоритмов анализа графов, относящихся к работе.

4.1. Поэлементное сложение CSR-матриц

В GraphBLAS-sharp уже был реализован один из стандартных алгоритмов сложения матриц, основанный на уже готовом сложении COO-матриц. Такой алгоритм также применяется в CUSP. Он состоит из следующих шагов.

1. Конвертация матриц из CSR-формата в координатный.
2. Поэлементное сложение матриц в координатном формате.
3. Конвертация результирующей матрицы обратно в CSR-формат.

Замеры производительности показали низкую эффективность данного алгоритма. Причиной этому стала дорогостоящая конвертация из CSR в COO-формат и высокий расход памяти на промежуточные представления матриц. Поэтому было решено складывать матрицы сразу в CSR-формате.

Для этого потребовалось реализовать версию алгоритма MergePath, осуществляющую параллельное слияние соответствующих массивов переданных CSR-матриц. Отличием от уже имеющейся версии для COO-матриц стало то, что теперь алгоритм слияния запускается по строкам матрицы, где каждая группа потоков осуществляет слияние своей строки. Раньше из-за наличия массива индексов строк у COO-матриц слияние можно было осуществлять целиком, без разделения на строки.

В результате слияния матриц получается массив, содержащий в соседних ячейках соответствующие элементы матриц. После этого требуется применить к элементам матриц заданную бинарную операцию. На этом этапе, в некоторых библиотеках, таких как SuiteSparse:GraphBLAS, реализуются два варианта алгоритма — для поэлементного сложения и умножения матриц. С использованием option

```
let mulOp =
    fun (left: int option) (right: int option) ->
        match left, right with
        | Some l, Some r -> Some (l * r)
        | Some l, None -> None
        | None, Some r -> None
        | None, None -> None
```

Рис. 8: Бинарная операция умножения в GraphBLAS-sharp

типов языка F# удалось обобщить данную операцию. Таким образом, если задать бинарную операцию, показанную на рисунке 8, можно выразить операцию поэлементного умножения матриц.

4.2. Умножение CSR-матрицы на плотный вектор

Первый этап алгоритма умножения реализуется самым наивным образом — поток с идентификатором i записывает результат умножения элемента матрицы с номером i на элемент вектора, соответствующий столбцу элемента i матрицы.

Единственным нетривиальным этапом умножения на плотный вектор является суммирование полученных на прошлом шаге значений. Этот эта полагается на использование локальной памяти графического процессора. Размер локальной памяти ограничен и часто недостаточно велик, чтобы поместить туда весь массив суммируемых значений. По этой причине в начале алгоритма рассчитывается объём свободной локальной памяти и массив загружается в локальную память максимально возможными частями. Процесс повторяется пока массив не будет обработан полностью. Когда элементы загружены в локальную память, необходимо выполнить сложение элементов, лежащих в одной строке. Для этого был выбран самый простой описанный в статье алгоритм. В рамках этого алгоритма каждый поток последовательно суммирует элементы в своей строке.

4.3. Умножение CSR-матрицы на разреженный вектор

Для умножения матрицы на разреженный вектор необходима сортировка элементов матрицы по столбцам. Для этого была выбрана уже имеющаяся в GraphBLAS-sharp операция битонной сортировки.

На этапе умножения элементов требуется по элементу матрицы получить соответствующий элемент вектора. Так как вектор разреженный, сделать это за константное время невозможно. В свою очередь, создание плотного представления вектора слишком дорогостоящая операция, поэтому для поиска элемента вектора по номеру используется бинарный поиск.

Для последующего сложения элементов по столбцам для каждого столбца используется отдельный поток, суммирующий элементы последовательно.

В некоторых случаях, после операции умножения на результат накладывается маска, фильтрующая элементы вектора. По этой причине добавлена возможность передать маску в операцию умножения, чтобы заранее определить, для каких элементов вектора производить расчёты не требуется.

Одним из алгоритмов, которые используют умножение матрицы на вектор, является обход графа в ширину. Иногда при обходе графа не важно, какие значения находятся в его вершинах, поэтому в таких случаях можно пропустить этап суммирования умноженных значений. Для таких алгоритмов была добавлена отдельная версия умножения, не считающая лишнюю сумму.

4.4. Оптимизация префиксной суммы и сортировки

В GraphBLAS-sharp уже имеется реализация алгоритма префиксной суммы и битонной сортировки, однако они обладает рядом недостатков, такими как излишнее выделение памяти и большое количество вызовов ядер. Всё это негативно сказывается на производительности, поэто-

му алгоритмы были переделаны так, чтобы вместо вызвов нескольких небольших ядер вызывалось как можно меньшее количество больших ядер, объединяющих старые. Кроме того, был реализован дополнительный вариант алгоритма префиксной суммы, не считающий итоговую сумму массива, так как она требуется не во всех алгоритмах. Это позволило избежать ещё одной дорогостоящей аллокации памяти.

4.5. Оптимизация Brahma.FSharp

Для взаимодействия с очередью команд OpenCL в Brahma.FSharp использовалась асинхронная очередь на основе MailboxProcessor. При обработке сообщений MailboxProcessor осуществлял необходимые для запуска OpenCL ядер действия, такие как синхронизация и установка аргументов, и вызывал команды OpenCL.NET.

В обработчике сообщений были обнаружены лишние синхронизации очереди OpenCL, а также выяснилось, что использование MailboxProcessor влечёт достаточно сильные накладные расходы на обмен сообщениями. По этой причине вместо MailboxProcessor теперь используются каналы .NET, а также предоставлен доступ к очереди OpenCL напрямую, без обмена сообщениями. Лишние синхронизации также были удалены.

```
let maskOp =
    fun (left: 'a option) (right: 'b option) ->
        match right with
        | None -> left
        | Some value -> None
```

Рис. 9: Операция наложения дополненной маски в GraphBLAS-sharp

5. Реализация алгоритмов анализа графов

Для реализации были выбраны одни из самых известных алгоритмов, которые можно выразить набором имеющихся в GraphBLAS-sharp операций: обход графа в ширину, поиск кратчайшего пути из заданной вершины, и ранжирование веб-страниц с помощью PageRank.

Основой реализованных алгоритмов служит умножение матрицы на плотный вектор. Чтобы сэкономить ресурсы на инициализацию нового вектора, результат умножения сразу записывается в старый вектор.

Помимо этого также требуются операции, которые будут фильтровать векторы от лишних вершин или записывать в них переданное значение. Примером таких операций может послужить удаление посещённых ранее вершин из фронта в обходе в ширину и обновление расстояний в поиске кратчайших расстояний. Все эти операции выражаются с помощью имеющейся в GraphBLAS-sharp операции *map2*, которая принимает бинарную операцию и два массива. Затем *map2* применяет переданную операцию к соответствующим элементам переданных массивов и сохраняет результат в другой массив. Так, например, передав в *map2* операцию наложения дополненной маски, как на рисунке 9, можно удалить из фронта те вершины, результат для которых покрыт маской, хранящей расстояния от начальной вершины. Также как и с умножением матрицы на вектор, результат записывается в уже существующие массивы там, где это возможно.

В процессе обхода в ширину фронт часто может содержать малое количество вершин относительно своего размера. В таком случае разумной идеей является представление фронта в разреженном виде. С этой целью помимо умножения матрицы на плотный вектор также было

```
module PageRank =
    type PageRankMatrix =
        member Dispose : RawCommandQueue -> unit

    val prepareMatrix : ClContext -> int -> (RawCommandQueue ->
     ↪  ClMatrix<float32> -> PageRankMatrix)

    val run : ClContext -> int -> (RawCommandQueue -> PageRankMatrix
     ↪  -> float32 -> ClVector<float32>)
```

Рис. 10: Сигнатура модуля PageRank с абстрактным типом PageRankMatrix.

реализовано умножение на разреженный вектор. Таким образом, умножение на плотный и разреженный вектор чередуются в зависимости от заполненности фронта, вычисляемой в конце каждой итерации. Такой вариант реализации обхода в ширину также добавлен в GraphBLAS-sharp.

Особенностью PageRank выступило то, что перед началом алгоритма матрицу смежности графа необходимо привести в специальный вид. Чтобы убедиться, что в алгоритм будет передана только матрица в корректном формате, была задействована система типов F#. С помощью задания сигнатуры модуля, изображённой на рисунке 10, удалось добиться того, что единственным способом создать пригодную для алгоритма матрицу остался метод prepareMatrix, благодаря этому алгоритм работает только с корректной неизменяемой матрицей.

6. Исследование производительности

Подход к исследованию производительности операций линейной алгебры и алгоритмов анализа графов, анализ результатов и выводы представлены в данном разделе.

6.1. Постановка экспериментов

Для постановки экспериментов в GraphBLAS-sharp существует отдельный проект под названием GraphBLAS-sharp.Benchmarks, в основе которого лежит инструмент BenchmarkDotNet. В проекте реализована загрузка матриц в формате Matrix Market[14] и их дальнейшие преобразования, выделение необходимой для эксперимента памяти и её чистка, а также замер времени исполнения выбранного алгоритма. Для замеров аналогов использовались graph-bench и SpBench.

Характеристики машины, на которой был поставлен эксперимент, следующие: Ubuntu 22.04, Intel Core i7-4790 CPU, 3.60GHz, DDR4 32GB RAM и GeForce GTX 1070, 8GB VRAM, 1410 MHz. С целью убедиться в работоспособности проекта на машинах с видеокартами AMD, были произведены замеры и на AMD Radeon Vega Frontier Edition, 16 GB VRAM, 1382 MHz.

Матрицы, участвующие в замерах, были взяты из The SuiteSparse Matrix Collection[15]. Были отобраны матрицы разной разреженности, размера и средней степени вершин. Выбранные матрицы приведены в таблице на рисунке 2.

Все замеры происходили по одной и той же схеме. После генерации исходных данных или чтения из файла, конвертации их в необходимый формат и загрузки на видеокарту, происходило 10 разогревочных итераций исследуемого алгоритма, за которыми следовало 100 итераций, по результатом которых высчитывалось среднее значение. Стандартное отклонение ни в одном из замеров не превысило 5 процентов от

[14]Описание Matrix Market формата — https://math.nist.gov/MatrixMarket/formats.html (дата обращения: 1.05.2024)

[15]Источник матриц для экспериментов — https://sparse.tamu.edu/ (дата обращения: 1.05.2024)

среднего значения.

Название	Размер	Количество элементов	Средняя степень
wing	62 032	243 088	3,9
coAuthorsCiteseer	227 320	1 628 238	7,2
coPapersDBLP	540 486	15 245 729	56,4
hollywood-2009	1 139 905	113 891 327	98,9
belgium-osm	1 441 295	3 099 940	2,1
roadNet-CA	1 971 281	5 533 214	2,8
road-central	14 081 816	33 866 826	2,4

Таблица 2: Матрицы, на которых производилось сравнение

6.2. Исследование оптимизаций Brahma.FSharp и базовых операций

В этом разделе будет описано влияние применённых в ходе данной работы оптимизаций на производительность библиотеки GraphBLAS-sharp.

6.2.1. Запуск простого OpenCL ядра

Прежде чем исследовать производительность сложных операций над графами, может быть полезно рассмотреть простейшие операции.

В качестве наиболее простой операции над массивом, которую можно выразить через OpenCL, была выбрана операция прибавления единицы ко всем элементам массива. Так как в данном эксперименте нас больше всего интересуют накладные расходы на запуск ядра, вместо сравнения с аналогичными библиотеками на C/C++, использующими CUDA или OpenCL, может быть достаточно сравнения с чистым OpenCL C.

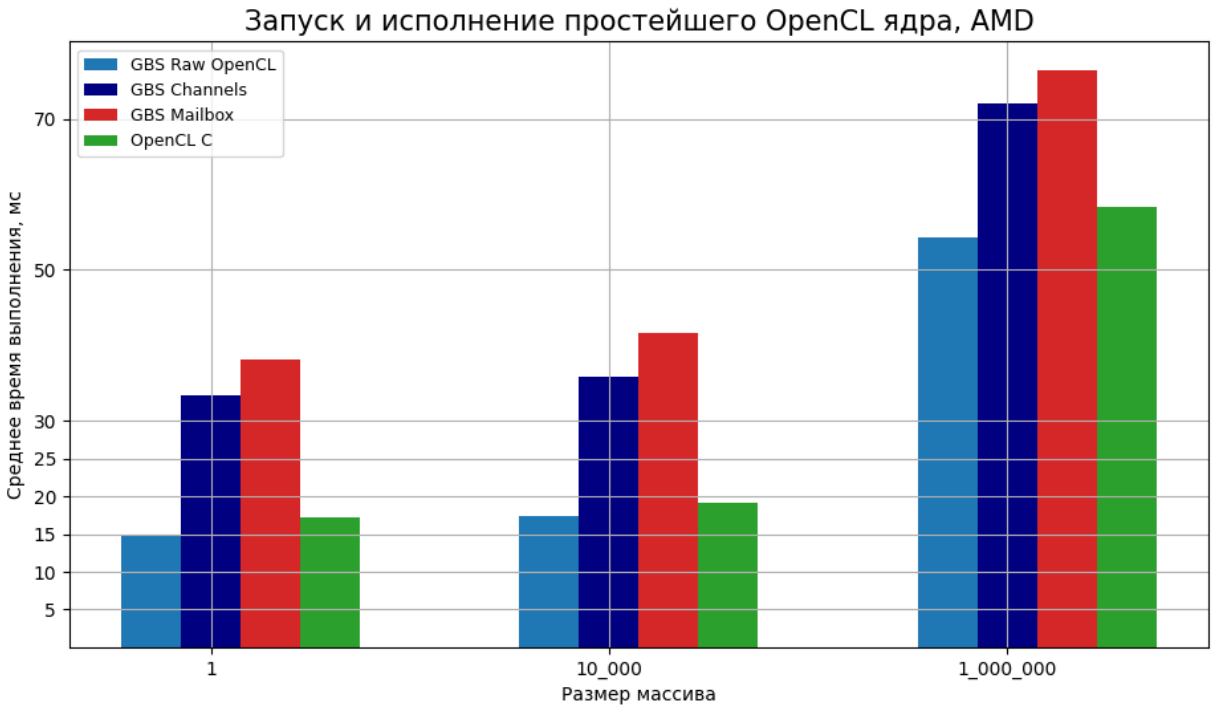

Рис. 11: Среднее время на запуск и исполнение простейшего OpenCL ядра одну тысячу раз в зависимости от размера массива, мс, AMD. Результаты для GraphBLAS-sharp с использованеим MailboxProcessor, .NET каналов и с чистой OpenCL очередью, а также OpenCL на C.

Для замеров в GraphBLAS-sharp и на языке C был написан код, который готовит аргументы для ядра и выставляет их, отправляет ядро на исполнение и дожидается его завершения. Для GraphBLAS-sharp реализовано три варианта алгоритма, с использованием MailboxProcessor, каналов .NET и с обращением напрямую в очередь OpenCL.

Из-за того, что время исполнения всей операции достаточно мало, описанная операция исполнялась одну тысячу раз в каждом замере. График среднего времени на запуск тысячи ядер представлен на рисунке 11.

На графике видно, что накладные расходы на запуск ядра в GraphBLAS-sharp и на OpenCL C практически одинаковы, в то время как использование асинхронной прослойки в виде каналов или MailboxProcessor значительно увеличивает время на запуск ядра, что особенно заметно при обработке небольших объёмов данных.

6.2.2. Префиксная сумма массива

Несколько более сложной операцией является префиксная сумма массива. В процессе работы алгоритма может запуститься два или более ядер, а также может присутствовать аллокация памяти под сумму элементов всего массива. Для сравнения были выбраны 4 версии префиксной суммы — версия до применения оптимизаций, версия с уменьшенным количеством запускаемых ядер как с использованием MailboxProcessor, так и без, а также версия без подсчёта суммы всех элементов массива. Все реализации основаны на том же алгоритме, что и в spla, поэтому имеет смысл провести сравнение с этой библиотекой. В spla итоговая сумма элементов массива не считается.

На графике из рисунка 12 изображено среднее время подсчёта префиксной суммы массива в GraphBLAS-sharp и spla. По результатам видно, что уменьшение количества запускаемых ядер и отказ от использования асинхронных прослоек для общения с очередью OpenCL оказывают значительное влияние на производительность. Также можно заметить, что отказ от подсчёта итоговой суммы массива позволяет увеличить производительность практически в два раза. При этом сам рассчёт происходит практически мгновенно, поэтому можно сделать вывод, что именно выделение дополнительной памяти под сумму повлекло потери производительности.

6.2.3. Алгоритмы анализа графов

Помимо исследования отдельных операций, может быть полезно рассмотреть, как они работают вместе. Обход графа в ширину включает в себя большое количество базовых операций, поэтому на его примере можно оценить результат оптимизации библиотеки. В качестве базовой версии обхода в ширину был взят обход с плотным фронтом. Далее к нему были применены следующие оптимизации — чередование плотного и разреженного фронта, оптимизация вспомогательных операций, таких как сортировка или префиксная сумма, и отказ от MailboxProcessor.

График из рисунка 13 отображает, во сколько раз быстрее по срав-

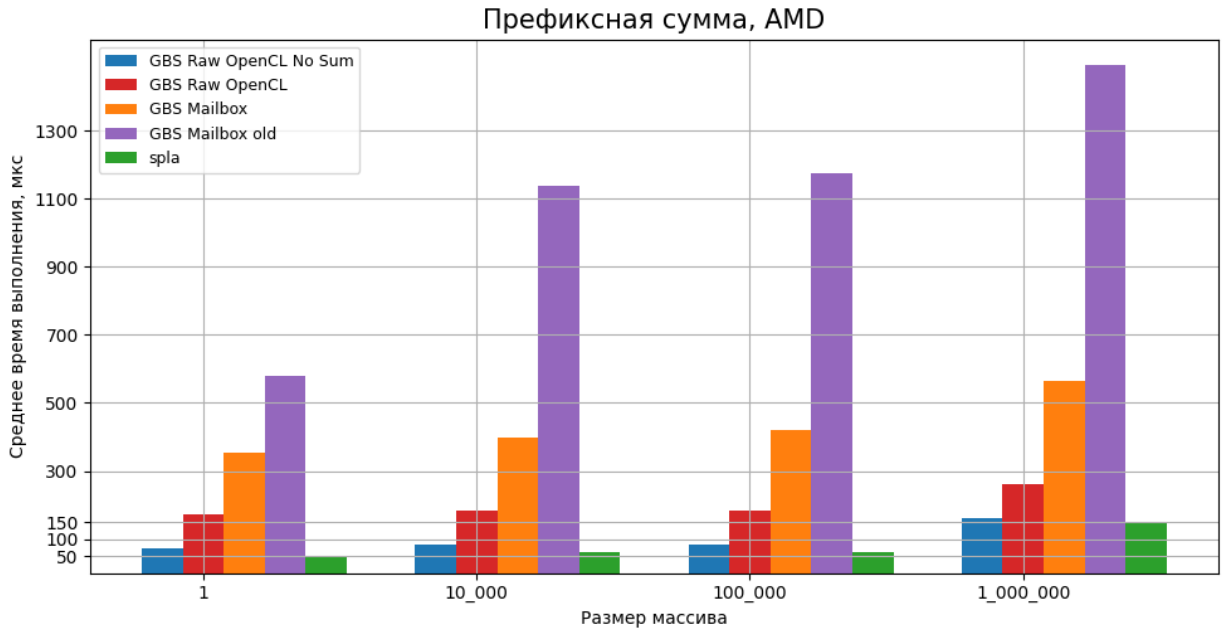

Рис. 12: Среднее время подсчёта префиксной суммы в зависимости от размер массива в GrapBLAS-sharp и spla, мкс, AMD. GBS Mailbox old — старая версия алгоритма, GBS Mailbox — версия с уменьшенным количеством ядер, GBS Raw OpenCL — алгоритм без использования MailboxProcessor, GBS No Sum — версия без подсчёта суммы элементов.

нению с базовой версией становится обход в ширину после применения оптимизаций. Можно заметить, что неоптимизированная версия Push-Pull обхода в половине случаев оказала отрицательное влияние на производительность, однако после оптимизации операций сортировки и префиксной суммы, которые используются только в обходе с разреженным фронтом, удалось добиться ускорения вплоть до 50 раз по сравнению с обходом только с плотным фронтом.

6.3. Сравнение производительности с аналогичными библиотеками

В данном разделе будет произведено сравнение производительности алгоритмов анализа графов и операций GraphBLAS-sharp с реализациями из аналогичных библиотек.

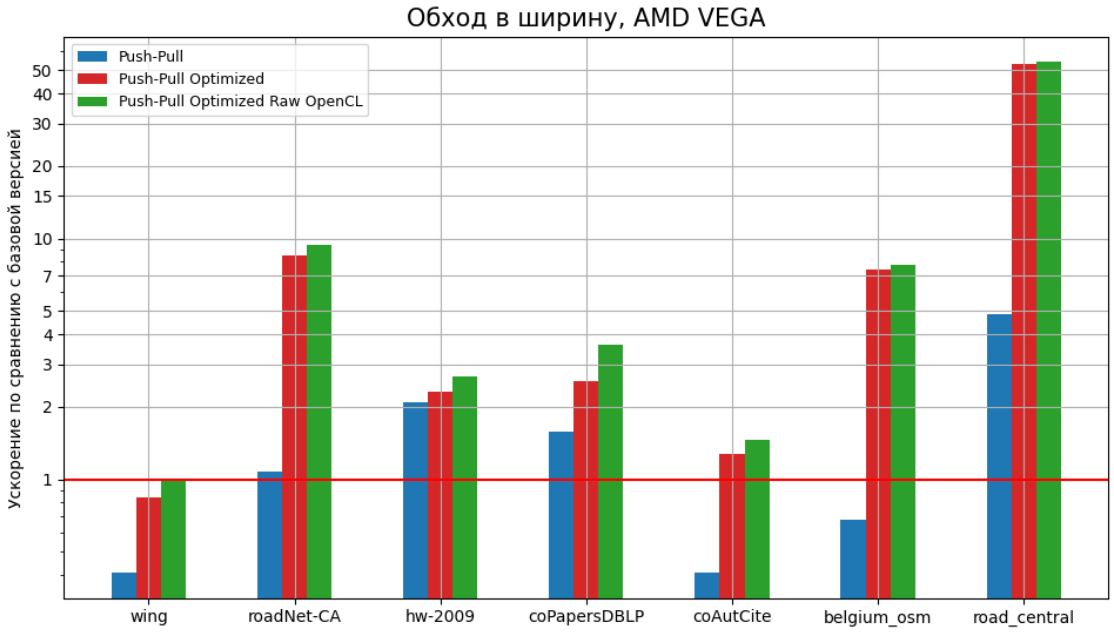

Рис. 13: Во сколько раз быстрее происходит обход графа в ширину в GraphBLAS-sharp после оптимизаций, по сравнению с базовой версией, AMD VEGA. Push-Pull — обход с чередованием плотного и разреженного фронта, Push-Pull Optimized — обход с оптимизированной префиксной суммой и сортировкой, Raw OpenCL — версия обхода после отказа от MailboxProcessor.

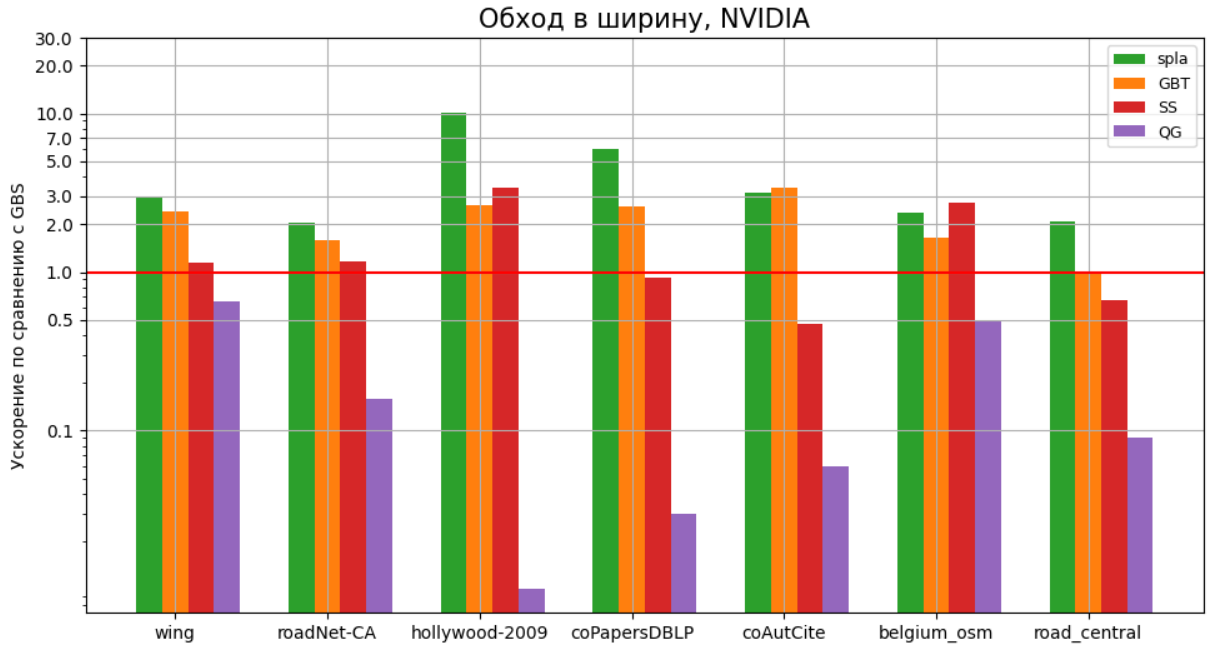

Рис. 14: Во сколько раз быстрее происходит обход графа в ширину в библиотеках-аналогах по сравнению с GraphBLAS-sharp, GTX 1070. Библиотеки: SuiteSpare:GraphBLAS/LAGraph (SS), GraphBLAST (GBT), Spla, QG(QuikGraph).

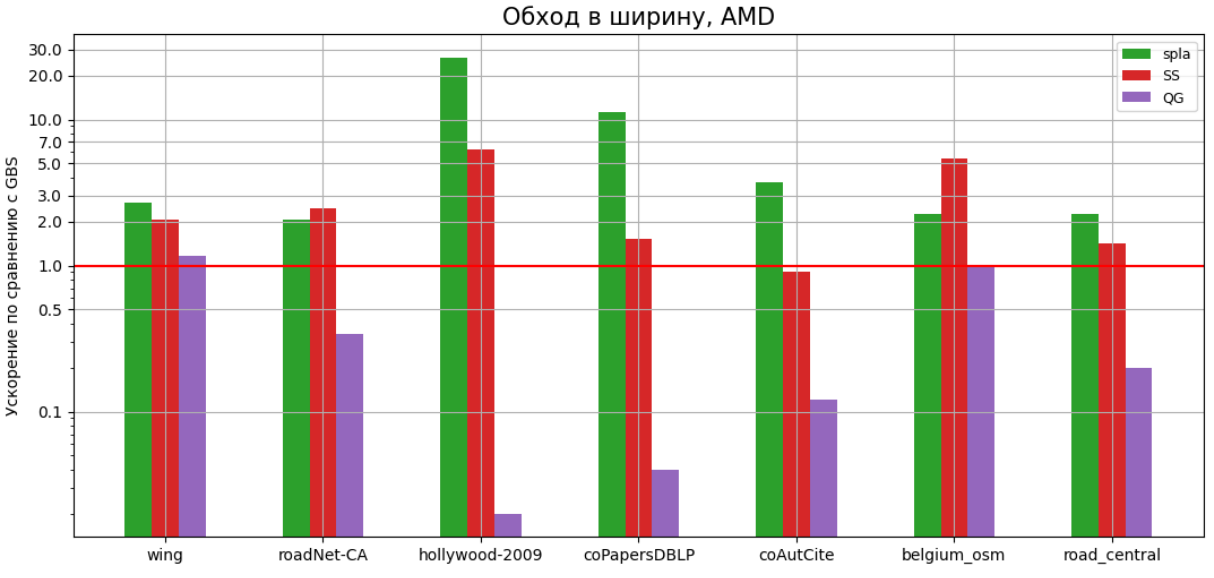

Рис. 15: Во сколько раз быстрее происходит обход графа в ширину в библиотеках-аналогах по сравнению с GraphBLAS-sharp, AMD VEGA. Библиотеки: SuiteSpare:GraphBLAS/LAGraph (SS), Spla, QG(QuikGraph).

6.3.1. Обход графа в ширину

Первым из реализованных в GraphBLAs-sharp и получившим наибольшее количество оптимизаций стал обход графа в ширину. Рисунки 14 15 представляют графики, показывающие, во сколько раз аналогичные библиотеки быстрее по сравнению с самой производительной версией обхода из GraphBLAS-sharp. В результате замеров на NVIDIA быстрее всего оказался spla, обгоняя GraphBLAS-sharp максимум в 10 раз и в среднем в 2-3 раза. Несколько медленнее spla оказался GraphBLAST, на одной из матриц время обхода совпало с GraphBLAS-sharp. SuiteSparse быстрее GraphBLAS-sharp на двух матрицах, одинаково быстр на трёх и медленнее на двух. Медленнее всех оказался QuickGraph, это связано с тем, что он не направлен на высокопроизводительную обработку больших разреженных графов. Можно также заметить, что на AMD отставание увеличивается примерно в 2 раза.

Несмотря на многократное ускорение обхода в ширину в результате оптимизаций, отставание от самых быстрых аналогов всё ещё значительное. Так как производительность простых базовых операций не уступает аналогам, а дополнительные расходы на коммуникацию с OpenCL через MailboxProcessor теперь сведены к нулю, можно полагать, что остались две причины отставания в производительности. Во-первых, как показали эксперименты и исследования с помощью профилировщика, выделение памяти на графическом устройстве занимает значительное время, в некоторых случаях превосходящее время исполнения соответствующего ядра. Некоторые алгоритмы можно переделать таким образом, чтобы минимизировать обращения к аллокатру памяти. В аналогичных библиотеках, например в spla, для ускорения работы с памятью реализуются собственные аллокаторы памяти. Во-вторых, некоторые из применяемых в реализации алгоритмов анализа графов операций могут работать быстрее или медленнее в зависимости от размеров исходных данных, поэтому следует рассмотреть возможность выбора реализации операции с учётом размера входных данных. К примеру, последовательное исполнение может оказаться предпочти-

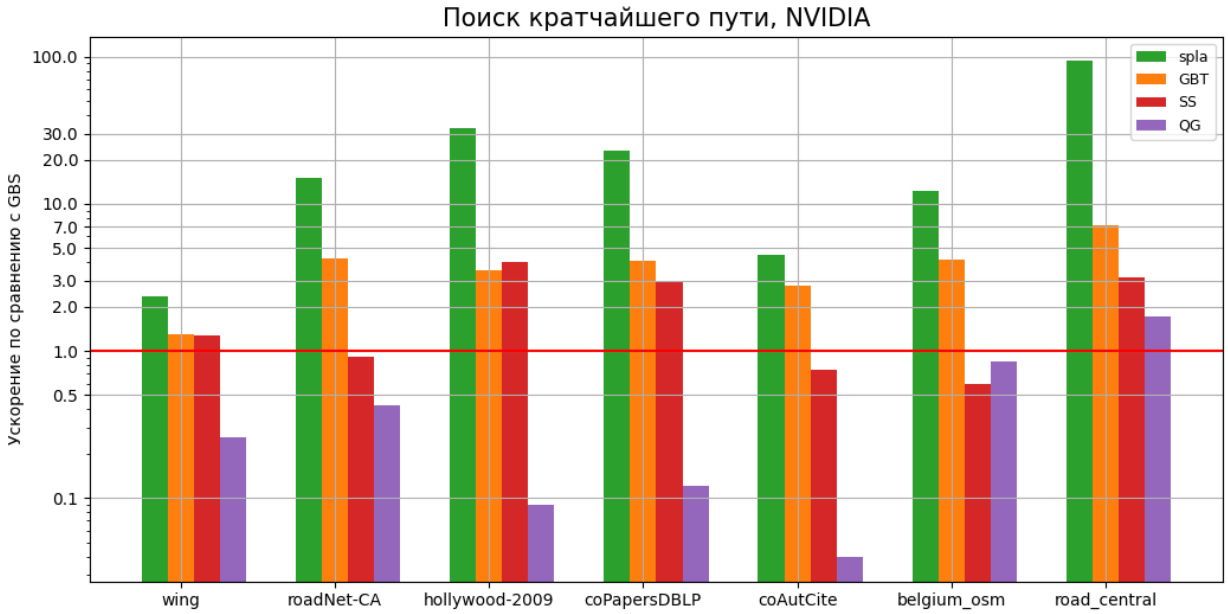

Рис. 16: Во сколько раз быстрее происходит поиск кратчайшего пути в библиотеках-аналогах по сравнению с GraphBLAS-sharp, GTX 1070. Библиотеки: SuiteSpare:GraphBLAS/LAGraph (SS), GraphBLAST (GBT), Spla, QG(QuikGraph).

тельнее при работе с массивом небольшого размера.

6.3.2. Поиск кратчайшего пути и PageRank

Помимо обхода графа в ширину были реализованы базовые версии поиска кратчайшего пути и ранжирования веб-страниц с помощью PageRank. Графики на рисунках 16 17 изображают, во сколько раз аналогичные библиотеки быстрее по сравнению с GraphBLAS-sharp. Можно заметить, что отставание от аналогов для поиска кратчайшего пути усилилось по сравнению с обходом в ширину, ближайшим кандидатом по скорости стал SuiteSparse. Для PageRank наблюдается отставание от аналогов на OpenCL и CUDA примерно на порядок, и выгрыш в производительности в несколько раз по сравнению со SuiteSparse.

6.3.3. Поэлементное сложение матриц

Для сравнения с библиотеками алгебраических операций была выбрана операция поэлементного сложения CSR матриц. Каждая матри-

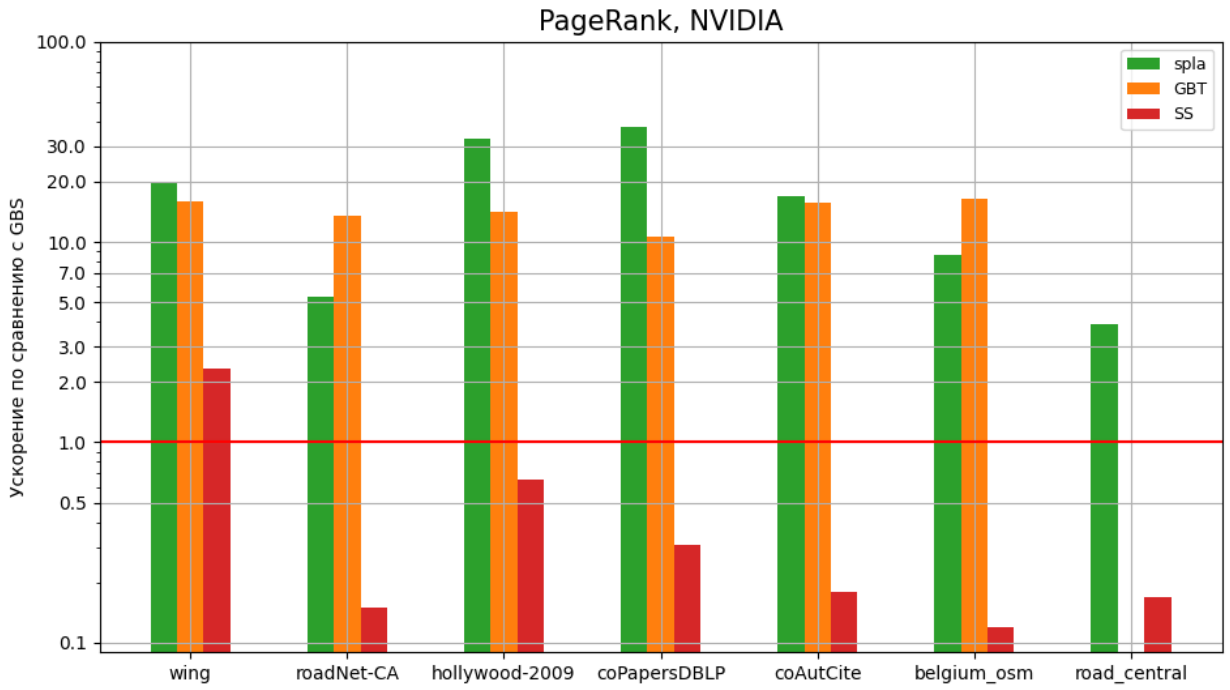

Рис. 17: Во сколько раз быстрее происходит ранжирование веб-страниц с помощью PageRank в библиотеках-аналогах по сравнению с GraphBLAS-sharp, GTX 1070. Библиотеки: SuiteSpare:GraphBLAS/LAGraph (SS), GraphBLAST (GBT), Spla. Отсутствие данных означает ошибку времени выполнения.

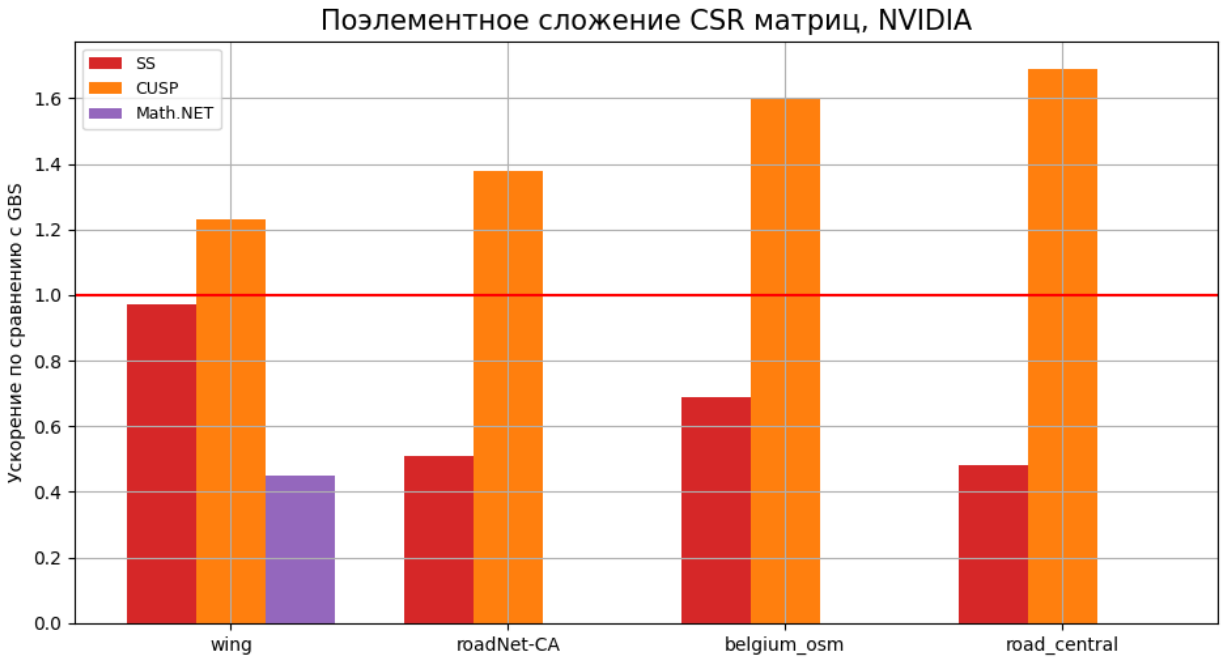

Рис. 18: Во сколько раз быстрее происходит поэлементное сложение CSR матриц в библиотеках-аналогах по сравнению с GraphBLAS-sharp, GTX 1070. Библиотеки: SuiteSpare:GraphBLAS/LAGraph (SS), CUSP, Math.NET Numerics. Отсутствие данных означает, что GraphBLAS-sharp более чем в тысячу раз быстрее.

ца складывалась со своим квадратом. На графике в рисунке 18 изображено, во сколько раз сложение матриц в аналогичных библиотеках быстрее, чем в GraphBLAS-sharp. Math.NET Numerics оказался более чем в тысячу раз медленнее на трёх матрицах. На этих же матрицах SuiteSparse отстал в два раза, но на самой маленькой отработал за то же время, что и GraphBLAS-sharp. Быстрее всех стал CUSP, обгоняя GraphBLAS-sharp примерно в полтора раза.

6.4. Выводы из исследования производительности

В результате сравнения производительности с аналогичными реализациями и исследования влияния оптимизаций на производительность были сделаны следующие выводы.

- Запуск ядра OpenCL это относительно дорогостоящая операция, поэтому следует реализовывать алгоритмы таким образом, чтобы требовалось запустить как можно меньше ядер.

- Влияние на производительность дополнительных прослоек над программным интерфейсом OpenCL ощутимо для небольших операций, но для более сложных операций, таких как обход графа в ширину, ухудшение производительности практически незаметно.
- В некоторых случаях затраты на выделение памяти превосходят затраты на вычисления, поэтому следует избегать дополнительных аллокаций там, где это возможно.
- Производительность рассмотренных базовых операций сравнима с аналогами или реализациями на чистом OpenCL C.
- Несмотря на многократное ускорение алгоритмов анализа графов в результате оптимизаций, они всё ещё ощутимо медленнее аналогов на OpenCL и CUDA, но сравнимы по скорости со SuiteSparse.
- На устройствах NVIDIA производительность библиотеки значительно выше, чем на AMD.

Заключение

В рамках данной работы были достигнуты следующие результаты.

1. Выполнена реализация библиотеки, включая перечисленные ниже возможности.

   - Доработаны и оптимизированы базовые операции библиотеки: сортировка и префиксная сумма массива.
   - Выявлены и устранены ухудшающие производительность недочеты в Brahma.FSharp: асинхронная коммуникация с очередью OpenCL через MailboxProcessor и лишние синхронизации этой очереди при обработке сообщений.
   - Реализованы алгебраические операции: поэлементное сложение разреженных матриц и умножение разреженной матрицы на плотный и разреженный вектор.

2. С помощью имеющихся операций реализованы следующие алгоритмы анализа графов: классический обход графа в ширину (с плотным фронтом), поиск кратчайшего пути, а также ранжирование веб-страниц с помощью PageRank. Реализованы две дополнительные вариации обхода в ширину: обход с чередованием плотного и разреженного фронта, и обход с только разреженным фронтом.

3. Исследована производительность созданной библиотеки. Установлено, что реализованные оптимизации для Brahma.FSharp и базовых операций привели к увеличению производительности не менее чем в 3 раза. Произведено сравнение быстродействия библиотечных операций и графовых алгоритмов с реализациями из аналогичных библиотек на устройствах AMD и NVIDIA. Обнаружено, что производительность рассмотренных базовых операций не уступает аналогичным реализациям, однако наблюдается отставание в производительности алгоритмов анализа графов от аналогов на CUDA и OpenCL.

В планы по дальнейшей разработке проекта входит устранение выявленных в ходе данной работе проблем с производительностью, а также разработка более дружественного пользовательского интерфейса, пополнение набора операций библиотеки и доработка транслятора Brahma.FSharp.

Код проекта GraphBLAS-sharp доступен в репозитории на сервисе github[16]. Имя пользователя: kirillgarbar.

[16]Репозиторий проекта GraphBLAS-sharp — https://github.com/YaccConstructor/GraphBLAS-sharp/tree/dev (дата обращения: 1.05.2024)

Список литературы

[1] Artiles Oswaldo, Saeed Fahad. TurboBFS: GPU Based Breadth-First Search (BFS) Algorithms in the Language of Linear Algebra. — Vol. 2021. — 2021. — 05.

[2] Baginski Czeslaw, Grzeszczuk Piotr. On the generic family of Cayley graphs of a finite group // J. Comb. Theory, Ser. A. — 2021. — Vol. 184. — P. 105495. — URL: https://doi.org/10.1016/j.jcta.2021.105495.

[3] Cayley Graphs of Semigroups Applied to Atom Tracking in Chemistry / Nikolai Nøjgaard, Walter Fontana, Marc Hellmuth, Daniel Merkle // J. Comput. Biol. — 2021. — Vol. 28, no. 7. — P. 701–715. — URL: https://doi.org/10.1089/cmb.2020.0548.

[4] Comparing large-scale graphs based on quantum probability theory / Hayoung Choi, Hosoo Lee, Yifei Shen, Yuanming Shi // Appl. Math. Comput. — 2019. — Vol. 358. — P. 1–15. — URL: https://doi.org/10.1016/j.amc.2019.03.061.

[5] Davis Timothy A. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra // ACM Trans. Math. Softw. — 2019. — dec. — Vol. 45, no. 4. — 25 p. — URL: https://doi.org/10.1145/3322125.

[6] Gao Jiaquan, Qi Panpan, He Guixia. Efficient CSR-Based Sparse Matrix-Vector Multiplication on GPU // Mathematical Problems in Engineering. — 2016. — 01. — Vol. 2016. — P. 1–14.

[7] Gilbert John R. What did the GraphBLAS get wrong? // HPEC GraphBLAS BoF. — 2022. — URL: https://sites.cs.ucsb.edu/~gilbert/talks/talks.htm.

[8] Green Oded, Odeh Saher, Birk Yitzhak. Merge Path - a Visually Intuitive Approach to Parallel Merging. — 2014. — 06.

[9] Jr. G. David Forney. Codes on Graphs: Models for Elementary Algebraic Topology and Statistical Physics // IEEE Trans. Inf. Theory. — 2018. — Vol. 64, no. 12. — P. 7465–7487. — URL: https://doi.org/10.1109/TIT.2018.2866577.

[10] Konig D. Graphen und Matrizen (Graphs and Matrices). — 1931.

[11] Kopp Nick. Using Cudafy for GPGPU Programming in .NET. — 2013. — URL: https://www.codeproject.com/Articles/202792/Using-Cudafy-for-GPGPU-Programming-in-NET.

[12] Nguyen Hubert. Parallel Prefix Sum (Scan) with CUDA // GPU Gems 3. — 2007. — 08. — ISBN: 9780321545428.

[13] Orachev Egor, Grigorev Semyon. Spla: Generalized Sparse Linear Algebra Library with Vendor-Agnostic GPUs Acceleration // EEE High Performance Extreme Computing Virtual Conference. — 2023. — 2 p. — URL: https://ieee-hpec.org/wp-content/uploads/2023/09/78.pdf.

[14] Review on Sparse Matrix Storage Formats With Space Complexity Analysis / Saira Jamalmohammed, Lavanya K., Sumaiya I., Biju V. — 2021. — 01. — P. 122–145. — ISBN: 9781799833376.

[15] Rizi Fatemeh Salehi. Graph Representation Learning for Social Networks : Ph. D. thesis / Fatemeh Salehi Rizi ; University of Passau, Germany. — 2021. — URL: https://opus4.kobv.de/opus4-uni-passau/frontdoor/index/index/docId/921.

[16] Standards for graph algorithm primitives / Tim Mattson, David Bader, Jon Berry et al. // 2013 IEEE High Performance Extreme Computing Conference (HPEC). — 2013. — P. 1–2.

[17] Washietl Stefan, Gesell Tanja. Graph Representations and Algorithms in Computational Biology of RNA Secondary Structure // Structural Analysis of Complex Networks / Ed. by Matthias Dehmer. —

Birkhäuser / Springer, 2011. — P. 421–437. — URL: https://doi.org/10.1007/978-0-8176-4789-6_17.

[18] Yang Carl, Buluç Aydın, Owens John D. GraphBLAST: A High-Performance Linear Algebra-Based Graph Framework on the GPU // ACM Trans. Math. Softw. — 2022. — feb. — Vol. 48, no. 1. — 51 p. — URL: https://doi.org/10.1145/3466795.

[19] Yang Carl, Wang Yangzihao, Owens John. Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU. — 2015. — 05. — P. 841–847.

[20] A novel weighted graph representation-based method for structural topology optimization / Jie Xing, Ping Xu, Shuguang Yao et al. // Adv. Eng. Softw. — 2021. — Vol. 153. — P. 102977. — URL: https://doi.org/10.1016/j.advengsoft.2021.102977.