Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Системное программирование

Группа 18Б.10-мм

Черников Артем Александрович

Реализация операций линейной алгебры на графическом процессоре с использованием F#

Отчёт по учебной практике

Научный руководитель:
к.ф.-м.н., доцент кафедры информатики СПбГУ Григорьев С. В.

Санкт-Петербург
2021

Оглавление

Введение

Граф — одна из важнейших абстрактных структур данных в информатике. Алгоритмы, использующие графы, применяются в таких областях, как биоинформатика [6], компьютерные сети и социальные сети [9]. Было показано, что из-за своей простоты и общности графы являются мощными инструментами для моделирования сложных задач [5]. Поэтому алгоритмы на графах стали одними из фундаментальных единиц в теоретической информатике[1], предоставив информационные ресурсы для исследований в различных областях, таких как, например, комбинаторная оптимизация, теория сложности и топология. Алгоритмы на графах были адаптированы и реализованы военными, коммерческими предприятиями и исследователями в академических кругах и стали незаменимыми для управления электросетью, телефонными системами и, конечно же, компьютерными сетями.

Параллельные алгоритмы на графах, как известно, сложно реализовать и оптимизировать [3]. Нерегулярные шаблоны доступа к данным и высокий показатель CCR[2] в алгоритмах на графах говорят о том, что даже у лучших алгоритмов будет параллельная эффективность, уменьшающаяся с увеличением количества процессоров [1].

Общий интерфейс обработки графов даёт полезный инструмент для оптимизации как программного, так и аппаратного обеспечения, чтобы создавать высокопроизводительные приложения на графах. Двойственность между каноническим представлением графов в виде абстрактных множеств вершин и рёбер и представлением в виде матрицы была частью теории графов с момента её создания [10]. Матричная алгебра была признана полезным инструментом в теории графов на протяжении примерно такого же количества времени [7]. Современное описание двойственности между алгоритмами на графах и матричной математикой (или разреженной линейной алгеброй) широко освещалось в ли-

[1]Под термином ”информатика” в данном случае подразумевается англоязычный термин ”computer science”

[2]CCR (Communication to Computation Ratio) — соотношение между количеством обменов данными между параллельными задачами и количеством вычислений, производимых ими

тературе и было кратко изложено в цитируемом тексте [8]. Этот текст послужил толчком к развитию стандарта математической библиотеки GraphBLAS[3], который был разработан в серии публикаций [11] и реализаций [13].

Стандарт GrapBLAS определяет операции над разреженными матрицами и векторами над расширенной алгеброй полуколец. Эти операции полезны для создания широкого диапазона алгоритмов на графах. Кепнер и Гильберт [8] предоставляют фреймворк для понимания как могут быть выражены алгоритмы на графах в терминах вычислений над матрицами. Например, рассмотрим произведение двух матриц: $C = AB$. Пусть A и B — разреженные булевы матрицы смежности n на n двух неориентированных графов с общими вершинами. Если переопределить матричное произведение так, что вместо умножения скаляров использовать логическое И, а вместо сложения — логическое ИЛИ, то C будет разреженной булевой матрицей смежности ориентированного графа, у которого есть дуга (i, j) тогда и только тогда, когда у вершины i в A и вершины j в B есть общая смежная вершина. Пара ИЛИ-И формирует алгебраическое полукольцо, соответственно многие подобные операции над графами могут быть лаконично представлены матричными операциями над различными полукольцами и различными числовыми типами. GraphBLAS описывает большой набор встроенных типов и операторов и позволяет пользователю создавать свои собственные. Выражение алгоритмов на графах в терминах линейной алгебры обеспечивает:

- мощный способ выразить алгоритмы на графах с помощью большого количества операций над матрицами смежности,
- поддержку композиции операций над графами,
- более простые алгоритмы на графах в пользовательском коде,
- высокую производительность.

[3]Спецификация GraphBLAS API — https://people.eecs.berkeley.edu/~aydin/GraphBLAS_API_C_v13.pdf (дата обращения: 2021-06-08)

Существует эталонная реализация стандарта GraphBLAS — SuiteSparse[4]. Разумеется, уже создано некоторое количество реализаций помимо вышеназванной, например GraphBLAST[5], GBTL[6], pggraphblas[7]. Тем не менее, наиболее релевантные решения по тем или иным причинам поддерживают некоторые из вышеперечисленных пунктов с определёнными ограничениями, что не совсем полностью раскрывает потенциал GraphBLAS.

Например, использование графических процессоров общего назначения позволяет достичь более высокой производительности [2], тогда как многие реализации поддерживают исполнение только на центральных процессорах. При этом достаточно обоснованно ожидать возможности исполнять код на широком спектре устройств, что может обеспечить поддержка платформы OpenCL[8].

[4]Авторское описание реализации SuiteSparse — https://people.engr.tamu.edu/davis/suitesparse.html (дата обращения: 2021-06-08)

[5]Репозиторий библиотеки GraphBLAST — https://github.com/gunrock/graphblast (дата обращения: 2021-06-08)

[6]Репозиторий библиотеки GBTL — https://github.com/cmu-sei/gbtl (дата обращения: 2021-06-08)

[7]Репозиторий библиотеки pggraphblas — https://github.com/michelp/pggraphblas (дата обращения: 2021-06-08)

[8]Ресурс с обзором платформы OpenCL — https://www.khronos.org/opencl (дата обращения: 2021-11-12)

1. Постановка задачи

Целью данной учебной практики является реализация на языке F# GraphBLAS API для матриц, представленных в координатном формате, с поддержкой OpenCL.

Для достижения данной цели были поставлены следующие задачи.

- Реализовать на языке F# архитектуру некоторой части описанных в стандарте GraphBLAS примитивов линейной алгебры, а также набор базовых операций линейной алгебры над матрицами и векторами, представленными в координатном формате, с поддержкой OpenCL.
- Произвести сравнение по производительности с существующими аналогами.

2. Обзор

В данном разделе представлены обзор одних из самых релевантных существующих реализаций стандарта GraphBLAS, выводы и комментарии к нему, а также обзор используемой библиотеки.

2.1. Существующие решения

Обзор существующих решений был произведён с целью выявить у них ограничения, накладываемые на предоставляемые ими возможности и спроектировать решение, минимизирующее эти ограничения и расширяющее некоторые возможности. Критерии обзора перечислены ниже.

- Высокая релевантность по запросам, касающимся темы реализации GraphBLAS, в поисковой системе Google.
- Ссылаемость из форума[9], посвящённому стандарту GraphBLAS.
- Ссылаемость из репозитория[10] на веб-сервисе GitHub, в котором перечислено множество ресурсов, связанных со стандартом GraphBLAS.

Вышеназванные форум и репозиторий имеют также высокую релевантность на данный момент по запросу "GraphBLAS" в поисковой системе Google.

Было произведено сравнение найденных решений по следующим критериям:

- переносимость (возможность выполнять код и на центральных, и на графических процессорах; насколько широк диапазон поддерживаемых графических процессоров),
- количество поддерживаемых операций линейной алгебры,

[9]Форум, посвящённый стандарту GraphBLAS — https://graphblas.github.io/ (дата обращения: 2021-06-08)

[10]Репозиторий со ссылками на ресурсы, связанные с GraphBLAS — https://github.com/GraphBLAS/GraphBLAS-Pointers (дата обращения: 2021-06-08)

Критерии сравнения	**Название библиотеки**			
	SuiteSparse	*GraphBLAST*	*GBTL*	*pggraphblas*
Переносимость	Только CPU	Только GPU, поддержка CUDA	Только CPU, в разработке GPU	Только CPU
Выбор операций	Широкий	Широкий	Достаточно широкий	Достаточно широкий
Гибкость	Низкая	Низкая	Низкая	Низкая

Таблица 1: Сравнение существующих решений

- гибкость с точки зрения создания пользователем собственных типов и операций над ними.

Результаты обзора приведены в таблице 1.

Из таблицы 1 видно, что переносимость (в широком смысле, упомянутом в критериях сравнения) в полной мере не поддерживается ни в одной из рассмотренных реализаций. С точки зрения производительности, лучшее решение из представленных — GraphBLAST, которое позволяет производить вычисления на GPU. Однако это решение поддерживает только архитектуру CUDA, то есть запускать код можно только на графических процессорах фирмы Nvidia. Касательно предоставляемых встроенных операций, показатели хорошие для всех обозреваемых решений, однако собственные пользовательские типы в них задать сложно.

Таким образом, было принято решение создать отдельную библиотеку, в которой будут устранены недостатки существующих аналогичных решений. Переносимость и производительность будут достигнуты благодаря поддержке платформы OpenCL, которая поддерживается не только графическими процессорами фирмы Nvidia, но и, например, AMD. Высокоуровневый интерфейс с гибкой системой типов будет осуществлён посредством языка F#.

2.2. Используемая библиотека Brahma.FSharp

Было принято решение использовать библиотеку Brahma.FSharp[11], так как она позволяет транслировать нативный код на языке F# в OpenCL, что обеспечивает высокоуровневое управление параллельными вычислениями. Благодаря данной библиотеке вкупе с языком F# пользователю становится возможным писать достаточно абстрактный, гибкий и высокоуровневый код, который может выполняться на графическом процессоре.

[11]Репозиторий библиотеки Brahma.FSharp — https://github.com/YaccConstructor/Brahma.FSharp (дата обращения: 2021-06-08)

3. Описание решения

В данном разделе представлено описание реализации предлагаемого решения. Код доступен в репозитории[12], размещённом на веб-сервисе GitHub.

Общая архитектура решения приведена на рисунке 1.

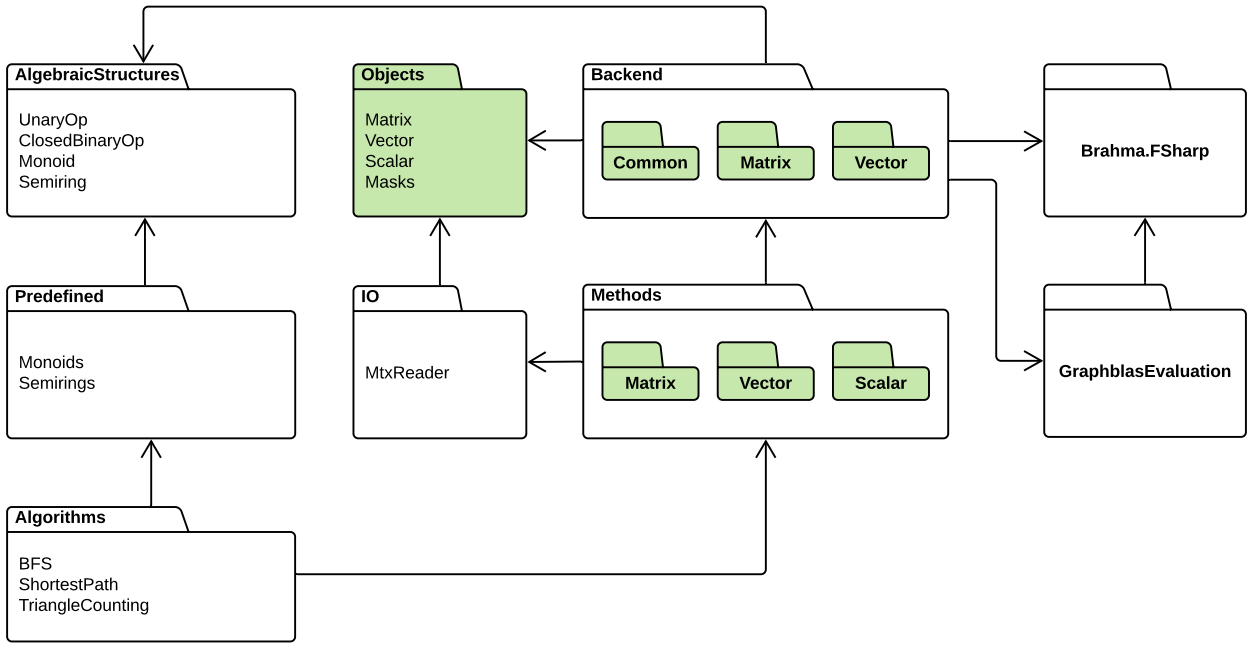

Рис. 1: Общая архитектура предлагаемого решения. Зелёным цветом выделены пакеты, над которыми работал автор.

3.1. Примитивы линейной алгебры

Из примитивов линейной алгебры реализованы матрица и вектор, представленные в координатном формате, одномерная и двумерная маски, а также скаляр. Представления вышеперечисленных примитивов описаны в пакете Objects.

[12]Ответвление от исходного репозитория библиотеки с окончательной версией реализации стандарта GraphBLAS — https://github.com/artemgl/GraphBLAS-sharp (дата обращения: 2021-06-08). Исходный репозиторий этой библиотеки — https://github.com/YaccConstructor/GraphBLAS-sharp (дата обращения: 2021-06-08)

Координатный формат представления разреженной матрицы[13] подразумевает хранение элементов в виде цепочки троек (i, j, a), где i — номер строки, j — номер столбца, a — значение, находящееся в i-той строке и в j-том столбце. Для вектора хранение организовано аналогично. Матрица и вектор, представленные в координатном формате, описаны записями COOMatrix и COOVector соответственно. Запись COOMatrix имеет поля Rows, Columns и Values. В поле Rows хранится массив номеров строк, в Columns — номеров столбцов, а в Values — значений. Длины этих массивов всегда равны. Таким образом, отдельную тройку (i, j, a) можно получить, взяв по одному значению из каждого массива, расположенному по фиксированному для каждой тройки индексу.

Полиморфизм относительно конкретной реализации матрицы или вектора реализован посредством размеченных объединений. Конкретная реализация каждой операции линейной алгебры описана в пакетах Backend и Methods. Реализация операций подробно описана в разделе 3.2.

Одномерная маска $\mathbf{m} = \langle N, \{i\}\rangle$ определяется размером $N > 0$ и множеством $\mathbf{ind(m)}$ индексов $\{i\}$, где $0 \le i < N$. Двумерная маска $\mathbf{M} = \langle M, N, \{(i, j)\}\rangle$ определяется количеством строк $M > 0$, столбцов $N > 0$ и множеством $\mathbf{ind(M)}$ пар $\{(i, j)\}$, где $0 \le i < M$, $0 \le j < N$. Нетрудно заметить, что одномерные и двумерные маски аналогичны векторам и матрицам соответственно, за исключением того, что маски не имеют значений. Также, на маске определена унарная операция — взять дополнение. Для одномерной маски $\mathbf{m}$ дополненная маска $\mathbf{\neg m}$ определяется как маска такого же размера, такая что $\mathbf{ind(\neg m)} = \{i : 0 \le i < N, i \notin \mathbf{ind(m)}\}$, то есть её множество индексов содержит все возможные элементы, не содержащиеся во множестве индексов изначальной маски. Для двумерной маски дополнение определяется аналогично.

Маска используется для того, чтобы обозначить, какие элементы нужно обрабатывать в операции, а какие — игнорировать. Применение

[13]Статья о разреженной матрице, раздел "Coordinate list (COO)" — https://en.wikipedia.org/wiki/Sparse_matrix (дата обращения: 2021-06-07)

маски позволяет избежать вычислений значений, которые не требуются в рамках текущей задачи, что обеспечивает оптимизацию вычислений. Также маска позволяет обращаться к конкретным элементам матрицы или вектора, требующимся в данном контексте. Операции линейной алгебры, поддерживающие маску, могут быть выполнены либо с ней, либо без неё. В первом случае вычисления будут производиться над всеми элементами результирующего вектора или матрицы, во втором — только над отмеченными в данной маске.

Классы Mask1D и Mask2D описывают одномерную и двумерную маски соответственно. В классе Mask1D содержится информация о размере одномерной маски, а также множество индексов, представленное в виде массива. В классе Mask2D хранение организовано аналогично в соответствии с её определением. Если реализовывать операцию "взять дополнение" явно, то применение данной операции к маске, построенной по разреженной матрице, будет возвращать в качестве результата маску, внутри которой будет храниться большое количество избыточной информации. По этой причине вместо этого в классы Mask1D и Mask2D был добавлен флаг isComplemented, отражающий необходимость взять дополнение данной маски перед использованием. Экземпляры классов Mask1D и Mask2D создаются вызовом функций mask или complemented для вектора и матрицы соответственно, реализованных в пакете Methods. Поддержка отсутствия маски реализована с использованием идеи перегрузки функций. Так как перегрузка в явном виде не поддерживается в F#, были созданы отдельные функции для каждой операции, поддерживающей маску, не принимающие её в качестве аргумента.

Некоторые операции принимают или возвращают единичное значение определённого типа, которым параметризованы эти операции. При этом реализовывать оптимизированные операции, работающие непосредственно с переменной, принадлежащей данному типу, не представляется возможным, так как это потенциально может требовать частого копирования единичного значения из видеопамяти. По этой причине был реализован класс Scalar, позволяющий хранить все единичные зна-

чения в видеопамяти и копировать их только по требованию программиста.

3.2. Операции линейной алгебры

В этом разделе будет подразумеваться координатный формат представления матрицы и вектора, если формат не будет уточняться.

Операции могут принимать матрицу, вектор, моноид (запись Monoid из пакета AlgebraicStructures), полукольцо (запись Semiring из того же пакета) и маску. Запись Monoid описывает множество (тип), определённую на нём бинарную ассоциативную операцию и нейтральный относительно этой операции элемент, принадлежащий этому множеству, то есть имеющий данный тип. Запись Semiring описывает пару: коммутативный моноид и мультипликативная операция. Таким образом, записи Monoid и Semiring определяют контекст, в котором требуется выполнить операцию.

3.2.1. Поэлементное сложение матриц и векторов

В рамках поэлементного сложения матрица и вектор, представленные в координатном формате хранения, могут быть рассмотрены как ничем не отличающиеся друг от друга примитивы линейной алгебры. Как уже было упомянуто ранее, матрица представляется в виде трёх массивов: номеров строк, номеров столбцов и значений. Пару, состоящую из первых двух массивов, можно воспринимать как один массив индексов. При этом форма представления ничем не будет отличаться от представления вектора. Поэтому рассмотрим далее только поэлементное сложение векторов.

Работа алгоритма состоит из следующих этапов.

1. Слить массивы индексов векторов в один с сохранением упорядоченности элементов.

2. Отметить повторяющиеся индексы в получившемся массиве.

3. Посчитать итоговую позицию каждого неповторяющегося индекса в получившемся массиве.

4. Перенести неповторяющиеся данные в результирующий массив индексов.

Таким образом, в результате получится упорядоченный массив, представляющий собой объединение двух исходных массивов как объединение множеств. Разделение алгоритма на вышеперечисленные этапы позволяет достичь параллелизма при его исполнении.

Помимо массива индексов требуется получить результирующий массив значений. Если значение встречается в паттерне только одного из векторов, оно будет складываться с нулём. Для избежания лишних вычислений, по аддитивному свойству нуля ($\forall a \quad a + 0 = 0 + a = a$) это значение сразу может быть записано в результирующий массив. Таким образом, складывать друг с другом требуется только пары значений под одинаковыми индексами. Над массивами значений производятся те же операции, что и над массивами векторов, за исключением того, что во время второго этапа дополнительно производится сложение пар элементов под одинаковыми индексами. При этом результат сложения записывается вместо того значения в паре, которое будет записано в результирующий массив.

Первый этап сложения представляет собой алгоритм Merge Path [12]. Этот алгоритм можно рассматривать как модификацию типичного однопоточного слияния двух упорядоченных массивов в один упорядоченный. При этом он обладает массовым параллелизмом. Идея этого алгоритма проиллюстрирована на рисунке 2. На осях расположены два массива A и B, которые требуется слить. На пересечениях i-той строки с j-тым столбцом находится 1, если $A[i] < B[j]$, или 0 иначе. Упорядоченность обоих массивов порождает, неформально говоря, лестницу, отделяющую клетки со значением 1 от клеток со значением 0. Заметим, что количество диагоналей в таблице равно сумме размеров массивов A и B. Пронумеруем диагонали таблицы. По тому, где и как i-тая диагональ пересекается с полученной лестницей, можно однозначно опреде-

лить, каким будет i-тое значение результирующего массива. Заметим также, что для каждой конкретной диагонали не требуется никакой информации о других диагоналях. Таким образом, для каждой диагонали параллельно можно найти бинарным поиском место пересечения с лестницей и, следовательно, значение в результирующем массиве.

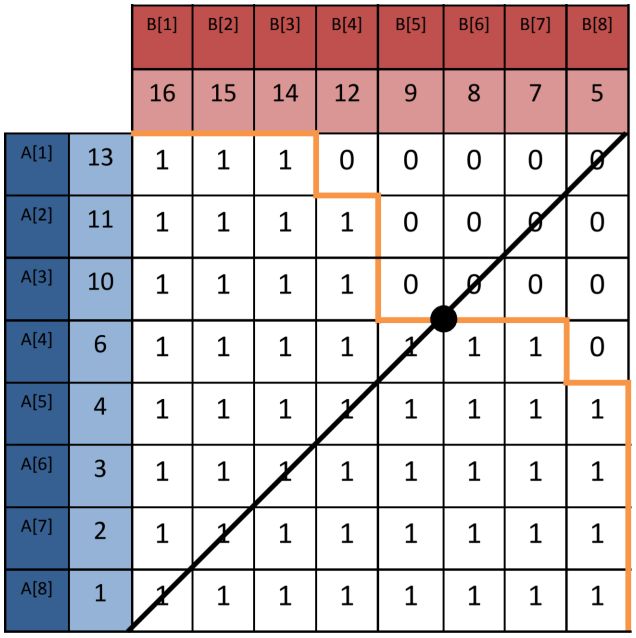

Рис. 2: Демонстрация концепции алгоритма Merge Path. Изображение взято из статьи [12]

Второй этап заключается в проверке повторяющихся элементов получившегося массива C. Создаётся вспомогательный массив, в котором на i-той позиции записывается 1, если $C[i] \neq C[i-1]$, и 0 иначе.

В третьем этапе к вспомогательному массиву применяется алгоритм вычисления префиксных сумм[14]. Для вспомогательного массива $\langle a_0, a_1, a_2, a_3, \dots \rangle$ результатом третьего этапа будет массив $\langle 0, a_0, a_0 + a_1, a_0 + a_1 + a_2, \dots \rangle$.

После третьего этапа из вспомогательного массива получается массив E, такой что i-тый элемент результирующего массива будет равен $D[E[i]]$, где D — массив, полученный на первом этапе. В соответствии с этой формулой на четвёртом этапе заполняются результирующие мас-

[14]Статья с описанием алгоритма вычисления префиксных сумм — https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda (дата обращения: 2021-06-08)

сивы.

3.2.2. Сокращение вектора при помощи аддитивной операции

Операция сокращения вектора возвращает скаляр, хранящий в себе сумму всех элементов вектора. Таким образом, для её реализации достаточно взять массив значений вектора и вычислить сумму всех его элементов.

Вычисление суммы всех элементов массива было реализовано с помощью упрощения алгоритма вычисления префиксных сумм массива путём отбрасывания лишних инструкций. Изначально массив делится на множество участков, в каждом из которых эффективно вычисляется локальная сумма этого участка и записывается в дополнительный массив. Таким образом, после этого этапа остаётся решить ту же задачу, но уже для вспомогательного массива, длина которого в n раз меньше, чем у изначального, где n — количество элементов в одном участке. Этот же шаг применяется итеративно к каждому новому массиву, пока длина очередного массива не станет равной 1. На основе указателя на первое значение в этом массиве создаётся запись Scalar и возвращается в качестве выходной переменной. В целях экономии памяти, после того, как количество дополнительных массивов становится равным двум, новые массивы не создаются. Вместо этого значениями заполняется старый, после чего указатели на два дополнительных массива меняются местами.

3.2.3. Запись вектора и скаляра в вектор через маску

Алгоритм записи вектора (вектор-аргумент) через маску в другой вектор (вектор-цель) состоит из следующих этапов.

1. Фильтрация вектора-аргумента через маску.

2. Слияние полученного вектора с вектором-целью.

Для того чтобы получить вектор, профильтрованный через маску, нужно отбросить у него те элементы, индексы которых не содержатся

в маске. Это реализовано аналогично алгоритму поэлементного сложения с той лишь разницей, что в последнем используется объединение индексов как множеств, в отличие от данного алгоритма, в котором используется их пересечение как множеств. Пересечение получится, если взять те элементы, которые отбрасывались в алгоритме поэлементного сложения. При этом элементы, содержащиеся в маске, но не содержащиеся в векторе, также добавляются в профильтрованный вектор со специальной меткой, так как эти элементы должны быть удалены из вектора-цели.

Второй этап записи вектора также аналогичен алгоритму поэлеметного сложения. Отличие состоит в том, что вместо операции сложения производится операция записи значения из отфильтрованного вектора-аргумента. Помимо этого значение со специальной меткой удаляется из результирующего вектора, то есть помечается нулем во вспомогательном массиве, для которого вычисляются префиксные суммы.

Запись скаляра в вектор через маску является частным случаем записи вектора. Таким образом, реализация этой операции аналогична вышеописанной.

4. Результаты

В данном разделе описаны результаты сравнения с существующим аналогом и сделанные на их основе выводы.

4.1. Сравнение с аналогом

В качестве аналога для сравнения была выбрана библиотека SuiteSparse. Сравнение производилось на примере поэлементного сложения двух матриц. В качестве контекста было выбрано полукольцо с типом, представляющим собой 32-битные числа с плавающей запятой, и стандартным сложением для этого типа. Матрицы, на которых тестировались алгоритмы, были выбраны из множества, предоставляемого SuiteSparse Matrix Collection [4]. Каждая матрица, будучи квадратной, складывалась с собой, возведённой заранее в квадрат. Характеристики выбранных матриц отражены в таблице 2. Характеристики ПК, на котором производилось сравнение: Ubuntu 20.04, Intel core i7-4790 CPU, 3.6GHz, DDR4 32Gb RAM и GeForce GTX 1070 GPU, 8Gb VRAM. Сравнение проводилось с помощью библиотеки BenchmarkDotNet[15]. Для каждой матрицы алгоритм запускался по 5 итераций. Учитывалось время копирования всех данных в видеопамять.

Название	Размер	Количество ненулевых элементов	Количество ненулевых элементов у возведённой в квадрат
luxembourg_osm	114599	119666	4582
belgium_osm	1441295	1549970	148316
wiki-Talk	2394385	5021410	42937
cit-Patents	3774768	16518948	1222

Таблица 2: Матрицы, на которых производилось сравнение

Результаты сравнения приведены в таблицах 3 и 4.

[15]Репозиторий библиотеки BenchmarkDotNet — https://github.com/dotnet/BenchmarkDotNet (дата обращения: 2021-11-13)

Название	Среднее, мс	Стандартное отклонение, мс
luxembourg_osm	37.2	0.25
belgium_osm	80.98	0.24
wiki-Talk	125.85	1.92
cit-Patents	504.51	3.4

Таблица 3: Результаты замеров для библиотеки GraphBLAS-sharp

Название	Среднее, мс	Стандартное отклонение, мс
luxembourg_osm	9.81	5.49
belgium_osm	46.59	4.52
wiki-Talk	53.51	4.22
cit-Patents	263.47	48.02

Таблица 4: Результаты замеров для библиотеки SuiteSparse

Из результатов видно, что предлагаемое решение на данном этапе на больших матрицах проигрывает аналогичному примерно в 2 раза.

Было выяснено, что библиотека Brahma.FSharp не позволяет выделять память сразу в видеопамяти, а требует выделять память на основном CPU, используемом для настройки выполнения, и затем копировать ее в видеопамять. В связи с этим образовываются накладные расходы на копирование вспомогательных данных в видеопамять. Время, затрачиваемое на копирование данных, представлено в таблице 5.

Название матрицы	Среднее время копирования данных, мс
luxembourg_osm	6.57
belgium_osm	36.04
wiki-Talk	85.52
cit-Patents	353.38

Таблица 5: Результаты замеров времени копирования вспомогательных данных

4.2. Выводы

В результате сравнения с аналогичным решением были сформулированы следующие выводы.

- Снижение накладных расходов осуществимо путём модифицирования библиотеки Brahma.FSharp.
- С учётом переносимости и высокоуровневого интерфейса предлагаемое решение жизнеспособно.

Заключение

В ходе выполнения данной работы были достигнуты следующие результаты.

- Реализована на языке F# архитектура таких описанных в стандарте GraphBLAS примитивов линейной алгебры, как одномерная и двумерная маски, матрица и вектор, представленные в координатном формате, а также скаляр.
- Реализованы на языке F# такие операции линейной алгебры над матрицами и векторами, представленными в координатном формате, с поддержкой OpenCL, как поэлементное сложение матриц/векторов, сокращение вектора при помощи аддитивной операции, запись вектора/скаляра в другой вектор через маску.
- Произведено сравнение с существующим аналогом по производительности и выявлены пути к оптимизации существующего решения.

Код доступен в репозитории[16], размещённом на веб-сервисе GitHub.

[16]Ответвление от исходного репозитория библиотеки с окончательной версией реализации стандарта GraphBLAS — https://github.com/artemgl/GraphBLAS-sharp (дата обращения: 2021-06-08).

Список литературы

[1] A. Azad G. Ballard A. Buluç. Exploiting Multiple Levels of Parallelism in Sparse Matrix-Matrix Multiplication. — 2016. — Режим доступа: https://arxiv.org/pdf/1510.00844.pdf (дата обращения: 2020-12-17).

[2] C. Yang A. Buluç John D. Owens. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. — January 2021. — Режим доступа: https://arxiv.org/pdf/1908.01407.pdf (дата обращения: 2021-11-11).

[3] D. Ediger R. McColl J. Riedy. STINGER: High Performance Data Structure for Streaming Graphs. — 2012. — Режим доступа: http://lovesgoodfood.com/jason/CV/material/hpec12-stinger.pdf (дата обращения: 2020-12-17).

[4] David Timothy A. Hu Yifan. The university of Florida sparse matrix collection. — November 2011. — Режим доступа: https://dl.acm.org/doi/10.1145/2049662.2049663 (дата обращения: 2021-11-11).

[5] E. Bergamini H. Meyerhenke. Approximating Betweenness Centrality in Fully-dynamic Networks. — 2016. — Режим доступа: https://arxiv.org/pdf/1510.07971.pdf (дата обращения: 2020-12-17).

[6] E. Georganas A. Buluç J. Chapman. Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly. — 2014. — Режим доступа: https://crd.lbl.gov/assets/pubs_presos/sc14genome.pdf (дата обращения: 2020-12-17).

[7] Harary F. Graph Theory. — 1969.

[8] J. Kepner J. Gilbert. Graph Algorithms in the Language of Linear Algebra. — Philadelphia, PA : SIAM Press, 2011. — Режим доступа: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719918.fm (дата обращения: 2020-12-17).

[9] J. Riedy D.A. Bader. Multithreaded Community Monitoring for Massive Streaming Graph Data. — 2014. — Режим доступа: http://lovesgoodfood.com/jason/cv/material/mtaap13-streaming-community-monitoring.pdf (дата обращения: 2020-12-17).

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

[11] Mattson T. Workshop on Graph Algorithms Building Blocks. — 2016.

[12] O. Green R. McColl A. Bader. GPU Merge Path - A GPU Merging Algorithm. — November 2014. — Режим доступа: https://www.researchgate.net/publication/254462662_GPU_merge_path_a_GPU_merging_algorithm (дата обращения: 2021-06-08).

[13] P. Zhang M. Zalewski A. Lumsdaine. GBTL-CUDA: Graph Algorithms and Primitives for GPUs. — 2016. — Режим доступа: https://resources.sei.cmu.edu/asset_files/ConferencePaper/2016_021_001_507186.pdf (дата обращения: 2020-12-17).