Санкт-Петербургский государственный университет

ПАНФИЛЁНОК Дмитрий Викторович

Выпускная квалификационная работа

Brahma.FSharp как основа для обобщённой разреженной линейной алгебры на GPGPU

Уровень образования: бакалавриат

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

Основная образовательная программа *СВ.5006.2018 «Математическое обеспечение и администрирование информационных систем»*

Профиль *Системное программирование*

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

Рецензент:
инженер-программист, ООО «ИнтеллиДжей Лабс» А. В. Бережных

Санкт-Петербург
2022

Saint Petersburg State University

Dmitriy Panfilyonok

Bachelor's Thesis

Brahma.FSharp as a tool for GPGPU-based generic linear algebra algorithms development

Education level: bachelor

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

Programme *CB.5006.2018 «Software and Administration of Information Systems»*

Profile: *System Programming*

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

Reviewer:
Software engineer, «IntelliJ Labs Co. Ltd.» A. V. Berezhnykh

Saint Petersburg
2022

Оглавление

Введение

Данные, представимые в виде графов, встречаются повсеместно: при анализе компьютерных и социальных сетей, в биоинформатике и статическом анализе кода [5]. Графы, возникающие в этих областях, могут содержать миллионы узлов и ребер, поэтому существует необходимость в высокопроизводительных инструментах анализа больших графов. В связи с этим многообещающей становится идея использования графических ускорителей общего назначения — GPGPU. Существующие решения уже сейчас доказывают, что использование GPGPU может повысить производительность алгоритмов анализа графов [1] [6], однако ценой служит более сложная модель программирования [10].
Иным является подход к организации вычислений над графами, основанный на стандарте GraphBLAS [5], который определяет базовые примитивы для построения графовых алгоритмов в терминах линейной алгебры. Свойством такого подхода является способность оперировать богатым набором графов различных типов с помощью небольшого набора матричных операций над полукольцами. Например, умножение матрицы на вектор, как показано на рисунке 1, является шагом в алгоритме поиска в ширину. Решения, основанные на стандарте GraphBLAS, производительны, масштабируемы и имеют более дружественное API [3]. Тем не менее на данный момент нет полноценных инструментов, реализующих стандарт GraphBLAS на графических процессорах общего назначения. Текущие реализации GraphBLAS на GPGPU (например, GraphBLAST [10]) показывают, что использование GPGPU действительно может улучшить производительность инструментов такого рода, однако разработчики сталкиваются не только с проблемами, связанными с реализацией обобщенных операций на графических процессорах с помощью стандартных инструментов языка C++, но и с переносимостью решений, основанных на программно-аппаратной платформе CUDA.

Одним из возможных подходов к реализации GraphBLAS на GPU является использование языка высокого уровня, а также библиотек,

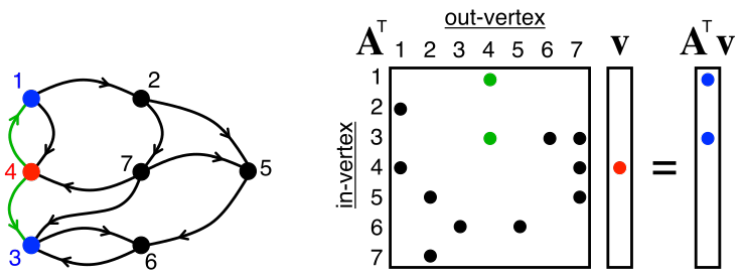

Рис. 1: Вычисление одного шага в алгоритме поиска в ширину[1]

динамически транслирующих конструкции и объекты данного языка в низкоуровневый код, способный исполнятся на графическом процессоре видеокарты. Данный подход был опробован на прототипе GraphBLAS-sharp[2], разработанном на кафедре системного программирования СПбГУ, который показал жизнеспособность этой идеи. В качестве библиотеки для взаимодействия с OpenCL прототип использует библиотеку Brahma.FSharp, которая также разрабатывается на кафедре системного программирования СПбГУ. Библиотека позволяет использовать подмножество языка F# для написания OpenCL ядер и предоставляет интерфейс для работы с ними. В ходе работы над прототипом GraphBLAS-sharp был выявлен ряд недостатков библиотеки Brahma.FSharp, перечисленных ниже.

1. Отсутствие атомарных операций для произвольных типов данных, что не позволяет реализовывать некоторые алгоритмы, которые могут оказаться лучше аналогов.

2. Отсутствие поддержки трансфера пользовательских типов данных из управляемой памяти в видеопамять.

3. Отсутствие возможности вручную управлять выделением памяти на OpenCL устройстве.

4. Отсутствие возможности исполнения нескольких OpenCL ядер параллельно.

[2]Репозиторий библиотеки GraphBLAS-sharp: https://github.com/YaccConstructor/GraphBLAS-sharp. Дата посещения: 09.03.2022

[2]GraphBLAS [Электронный ресурс] // Википедия. Свободная энциклопедия. – URL: https://en.wikipedia.org/wiki/GraphBLAS (дата обращения: 01.01.2022).

1 Постановка задачи

Предметом данной работы является улучшение библиотеки Brahma.FSharp с целью её применения в инструментах обобщённой разреженной линейной алгебры на GPGPU. Для этого были поставлены следующие задачи:

1. реализовать поддержку обобщенных атомарных операций;
2. реализовать поддержку трансфера пользовательских типов данных;
3. обеспечить возможность управления выделением памяти на OpenCL устройстве;
4. обеспечить возможность параллельного исполнения OpenCL ядер;
5. сравнить производительность полученного решения с аналогами.

2 Обзор

2.1 Обзор аналогичных инструментов

Выбор в пользу библиотеки Brahma.FSharp для использования в описанном выше прототипе был обусловлен рядом причин. При поиске инструмента для взаимодействия с GPGPU критериями отбора являлись перечисленные ниже параметры.

- **Возможность описывать ядра для GPGPU на языке высокого уровня.** Такая возможность может позволить достичь уровня переносимости, необходимого в задачах обобщенной линейной алгебры. Многие библиотеки, представляющие собой высокоуровневые обертки над вызовами к API GPU, такие как pyopencl[3], pycuda[4], OpenCL.NET[5], Cloo[6], JavaCL[7], JOCL[8] и другие, не удовлетворяют этому критерию и не подходят для использования в описанном контексте.

- **Использование OpenCL в качестве платформы исполнения.** Решения, основанные на платформе OpenCL, более переносимы, чем аналогичные решения для других платформ, таких как NVIDIA CUDA и Metal. По этой причине из рассмотрения были исключены некоторые инструменты, такие как, например, Numba[9].

- **Возможность описывать собственные ядра.** Такая возможность является необходимой для реализации многих алгоритмов,

[3]Репозиторий библиотеки pyopencl: https://github.com/inducer/pyopencl. Дата обращения: 17.03.2022

[4]Репозиторий библиотеки pyopencl: https://github.com/inducer/pycuda. Дата обращения: 17.03.2022

[5]Репозиторий библиотеки OpenCL.NET: https://github.com/dgsantana/OpenCL.NET. Дата посещения: 01.06.2021

[6]Репозиторий библиотеки Cloo: https://github.com/clSharp/Cloo. Дата посещения: 01.06.2021

[7]Репозиторий библиотеки JavaCL: https://github.com/nativelibs4java/JavaCL. Дата посещения: 01.06.2021

[8]Репозиторий библиотеки JOCL: https://github.com/gpu/JOCL. Дата посещения: 01.06.2021

[9]Репозиторий библиотеки Numba: https://github.com/numba/numba. Дата посещения: 09.03.2022

требуемых в библиотеке обобщенной разреженной линейной алгебры. По причине несоответствия этому критерию из рассмотрения была исключена, например, библиотека Accelerate[10].

- **Возможность использовать данные произвольных (или почти произвольных) типов, в том числе обобщенных.** Такая возможность необходима, так как стандарт GraphBLAS не накладывает ограничения на используемые типы данных. По этому критерию были исключены из рассмотрения такие библиотеки, как Aparapi[11] и SPOC[12].

- **Возможность использования на платформе .NET или Java.** Эти платформы достаточно популярны, чтобы потенциальный инструмент для обобщенной линейной алгебры был более востребован.

- **Инструмент написан на функциональном языке программирования.** Для функциональных языков программирования могут быть доступны некоторые техники суперкомпиляции и оптимизации, например kernel fusion [7] и deforestation [9], которые могли бы улучшить производительность разрабатываемого инструмента. По причине несоответствия этому критерию из рассмотрения были исключены, например, библиотеки ILGPU[13] и Alea GPU.

В конечном итоге среди инструментов, подходящих под заданные требования, остались FSCL[14] — библиотека для языка программирования F#, позволяющая работать с OpenCL, и библиотека Brahma.FSharp.

[10]Репозиторий библиотеки Accelerate: https://github.com/AccelerateHS/accelerate. Дата посещения: 09.03.2022

[11]Репозиторий библиотеки Aparapi: https://github.com/aparapi/aparapi. Дата посещения: 09.03.2022

[12]Репозиторий библиотеки SPOC: https://github.com/mathiasbourgoin/SPOC. Дата обращения: 09.03.2022

[13]Репозиторий библиотеки ILGPU: https://github.com/m4rs-mt/ILGPU. Дата посещения: 09.03.2022

[14]Репозиторий библиотеки FSCL: https://github.com/FSCL/FSCL.Runtime. Дата посещения: 09.03.2022

Библиотека FSCL, несмотря на преимущества в виде продвинутой поддержки исполнения программ в multi-GPU режиме, оказалось недостаточно гибкой из-за особенной модели программирования. К тому же, библиотека давно не поддерживается. По этой причине была выбрана библиотека Brahma.FSharp, разрабатывающаяся на кафедре системного программирования СПбГУ.

2.2 Платформа OpenCL и модель исполнения

Центральным элементом платформы OpenCL [4] является понятие хоста — устройства, которое координирует вычисления и осуществляет взаимодействие с пользователем. Команды, переданные с хоста, выполняются на OpenCL устройствах, которые могут быть представлены во множественном числе и отличаться по типу (CPU, GPU, FPGA) и платформе (AMD, Intel и так далее). OpenCL устройства логически делятся на вычислительные модули (compute units), которые в свою очередь делятся на обрабатывающие элементы (processing elements). Схематическое изображение модели платформы показано на рисунке 2. Между хостом и OpenCL устройствами существуют отличия в том числе в модели программирования. OpenCL программа, по сути, состоит из двух частей — кода ядер, которые реализуют параллельные вычисления и запускаются на OpenCL устройствах, и программы хоста, которая управляет ресурсами устройств, координирует исполнение OpenCL ядер и так далее.

Взаимодействие между хостом и OpenCL устройством происходит посредством команд, помещенных в очередь команд (command queue). Хост отправляет команды в очередь, после чего они устанавливаются планировщиком для исполнения на устройствах. Команды могут отвечать за выполнение ядер, управление памятью и синхронизацию выполнения команд в очереди. Они исполняются асинхронно между хостом и устройством. Команды могут выполняться последовательно (in-order execution) или внеочередно (out-of-order execution). Координация команд исполнения ядер и команд управления памятью может быть осу-

ществленна в том числе за счет объектов-событий, которые генерируют эти команды.

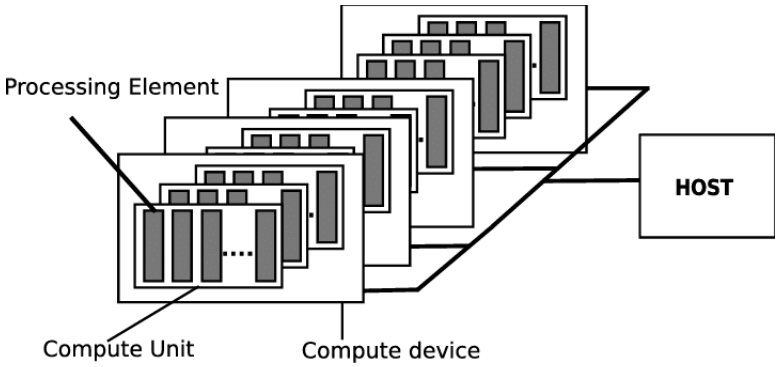

Рис. 2: Архитектура платформы OpenCL [8]

Кроме того, OpenCL определяет индексное пространство (NDRange), которое может иметь от 1 до 3 измерений. Индексное пространство содержит множество точек, называемых рабочими элементами (work-items). Каждый рабочий элемент имеет уникальный индекс в глобальном индексном пространстве и занимается параллельным исполнением ядра, код которого является общим для всех элементов. Рабочие элементы ко всему прочему сгруппированы в рабочие группы (work-groups) заранее определенного размера. Соответственно, рабочий элемент обладает также локальным индексом — уникальным идентификатором внутри рабочей группы. Элементы внутри одной рабочей группы могут синхронизироваться, а также имеют доступ к общей локальной памяти. Схематическое изображение индексного пространства показано на рисунке 3.

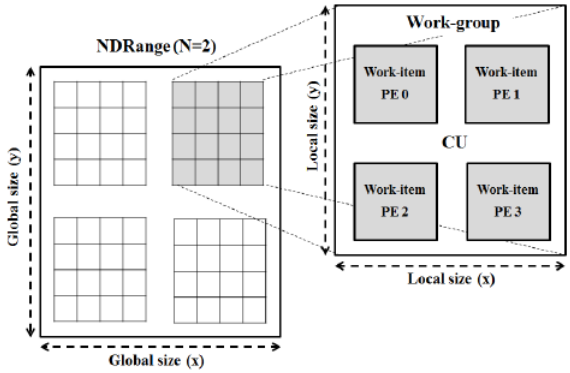

Рис. 3: Модель индексного пространства OpenCL [2]

2.3 Язык программирования F#

F# — это мультипарадигменный язык программирования из семейства языков платформы .NET. Язык позволяет писать типобезопасные программы в функциональном стиле благодаря строгой типизации и поддержке функциональной парадигмы. Среди характеристик языка можно выделить особенности, описанные ниже.

- **Размеченные объединения.** Размеченное объединение является типом, представляющим собой конечный набор альтернатив. Примером размеченного объединения является тип `Option`, имеющий 2 альтернативы: `Some` — наличие значения, и `None` — его отсутствие.

- **Цитирование (F# Quotations).** Механизм цитирования позволяет представить блок кода на языке F# в виде абстрактного синтаксического дерева. Благодаря этому программист может проводить анализ и преобразование такого дерева во время исполнения. Среди доступных применений этой особенности стоит выделить возможность преобразовывать код на языке программирования F# в код на другом языке. Для того чтобы создать цитату выражения языка F#, его нужно обернуть в специальные кавычки `<@ ... @>`. Выражение, объявленное таким образом, компилируется в значение типа `Quotations.Expr`, представляющее собой абстрактное синтаксическое дерево этого выражения.

- **Вычислительные выражения.** Вычислительные выражения представляют собой особый механизм описания вычислений. С их помощью могут быть описаны, например, монады, моноиды, аппликативыне функторы. С их помощью можно описывать недетерминированные вычисления, асинхронные вычисления, вычисления с побочными эффектами.

2.4 Библиотека Brahma.FSharp

Brahma.FSharp — это библиотека, которая позволяет использовать возможности OpenCL в проектах на языке F#. Её особенности перечислены ниже.

- Позволяет использовать подмножество языка F# для написания кода ядер OpenCL.
- Позволяет проверять некоторые ошибки на этапе компиляции.
- Поддерживает трансляцию пользовательских типов данных.
- Обеспечивает переносимость за счет кодогенерации.

Библиотека состоит из двух основных компонентов:

- транслятора из F# в OpenCL;
- API хоста для запуска ядер и управления памятью.

2.4.1 Описание OpenCL ядер

Для описания ядер OpenCL используется механизм цитирования F# кода. Ядро описывается лямбда-функцией с типом `#NDRange -> 'a`, где первым аргументом передаются данные об индексном пространстве ядра. При этом ядро может быть полиморфно относительно типа данных, с которыми оно работает. Внутри ядра возможно использовать почти любые выражения языка F#, а также некоторые специально определенные функции из библиотеки Brahma.FSharp. К таким функциям относятся, например: `barrierLocal()` для создания синхронизирующего барьера внутри рабочей группы или `localArray<'a>` для выделения локальной памяти. Пример описания простейшего ядра приведен в листинге 1.

2.4.2 Транслятор

В библиотеке используется машинно-зависимый транслятор, состоящий из трёх частей:

- модуля, который занимается преобразованием абстрактного синтаксического дерева кода на F#;
- модуля, который преобразует абстрактное синтаксическое дерево кода на F# в дерево языка ядер OpenCL;
- модуля, который по дереву языка ядер OpenCL генерирует OpenCL код.

Модель трансляции изображена на рисунке 4.

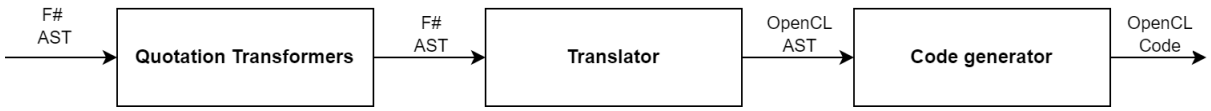

Рис. 4: Модель трансляции, используемая в библиотеке Brahma.FSharp

Среди его особенностей можно выделить следующие:

- поддерживает возможность использования вложенных функций;
- поддерживает возможность печати в консоль изнутри ядра;
- поддерживает возможность использования в коде ядра пользовательских типов данных.

Листинг 1: Пример описания простейшего ядра на языке F#

```
let kernel =
    <@
        fun (range: Range1D) (buffer: 'a clarray) (value: 'a) →
            let gid = range.GlobalID0
            buffer.[gid] ← value
    @>
```

3 Реализация

3.1 Поддержка обобщенных атомарных операций

Когда речь идёт о библиотеке обобщенных вычислений на GPGPU, проблема использования произвольных атомарных операций встает весьма остро. Их отсутствие существенно ограничивает спектр доступных алгоритмов, в том числе необходимых в задачах линейной алгебры. Стандарт OpenCL позволяет использовать некоторый набор атомарных функций над небольшим множество примитивных типов, однако для реализации обобщенных алгоритмов этого оказывается недостаточно. К сожалению, общепринятого решения этой проблеме найти не удалось. Среди аналогичных инструментов только ILGPU имеет возможность конструировать и использовать произвольные атомарные операции на основе неатомарных. Для этого в библиотеке есть статический метод `MakeAtomic`, контракт которого приведен в листинге 2. В качестве реализации используется цикл активного ожидания.

Листинг 2: Контракт метода MakeAtomic

```
public T MakeAtomic<T, TOperation, TCompareExchangeOperation>(
    ref T target,
    T value,
    TOperation operation,
    TCompareExchangeOperation compareExchangeOperation)
```

Существует и альтернативный способ расширить множество доступных для использования атомарных операций. Этот способ использует знание о том, что устройства, реализующие стандарт OpenCL, обеспечивают поддержку атомарных операций над 64-битными целочисленным типом long, в то время как поддержка других 64-битных типов (например, double) не гарантируется. Это ограничение можно обойти за счет использования union структуры, объединяющей любой 64-битный тип с типом long[15]. Очевидным недостатком такого решения является

[15]Ссылка на предлагаемое решение: https://stackoverflow.com/questions/31863587/atomic-operations-with-double-opencl/31865819#31865819. (Дата посещение: 27.05.2022)

то, что оно подходит только для типов размером 64 бита.

В данной работе поддержка произвольных атомарных операций выполнена за счет генерации функций, реализующих спинлок. Для этого язык ядра был расширен функцией `atomic` с сигнатурой, приведенной в листинге 3. Несмотря на то, что это обыкновенная функция языка F#, её применение внутри кода ядра ограниченно. Так, существуют ограничения на используемые в качестве первого параметра выражения — это должна быть либо переменная, либо обращение к массиву по индексу, а также ограничены возможности ее частичного применения. Обработкой этой функции занимается соответсвующий шаг на этапе преобразования синтаксического дерева (как изображено на рисунке 5), который генерирует спинлок и заменяет вызов функции `atomic` на вызов функции, реализующей этот спинлок. Кроме того, он преобразует код ядра таким образом, чтобы обеспечить возможность выделения памяти под мьютексы — для каждого массива, доступ к которому необходимо ограничить взаимным исключением, выделяется дополнительный массив 32-битных целых чисел соответствующей длины, заполненный нулями. Таким образом, например, код ядра на языке F#, приведенный в листинге 4, преобразуется в код на языке OpenCL C, приведенный в листинге 5. Стоит также отметить, что в случае, когда вызов функции atomic можно преобразовать в вызов нативной атомарной функции OpenCL, спинлок не генерируется.

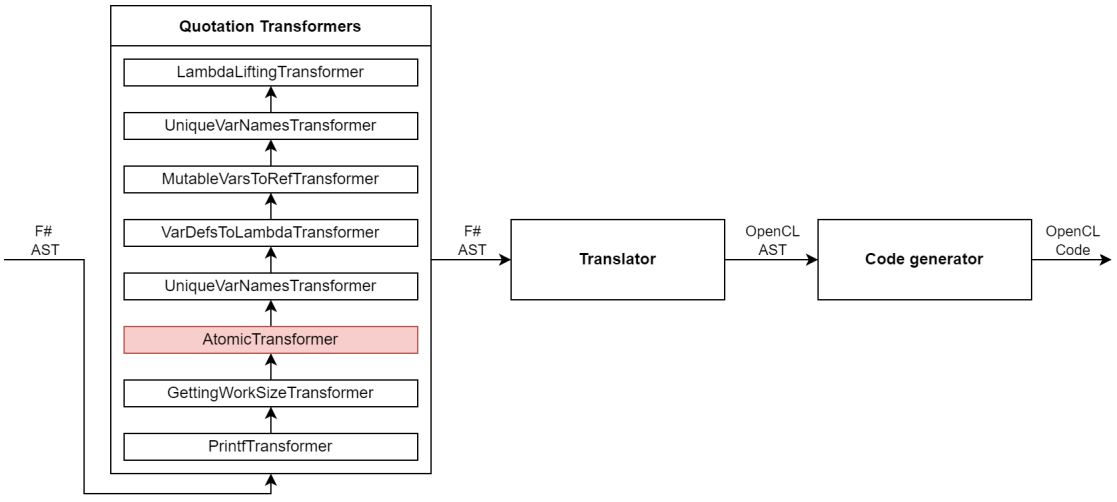

Рис. 5: Этап преобразования функции atomic в модели трансляции

Листинг 3: Сигнатура функции atomic

```
val atomic : ('a → 'b) → ('a → 'b)
```

Листинг 4: Код OpenCL ядра на языке F#

```
<@
    fun (range: Range1D) (buf: ClArray<int>) →
        atomic (fun x → x + 1) buf.[0] ▷ ignore
@>
```

Листинг 5: Код OpenCL ядра после трансляции в C

```
#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics: enable
#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics: enable

int baseFunc(private int x)
{
    return (x + 1);
}

int atomicFunc(__local int* localAccMutex, __local int* x) {
    int oldValue;
    bool flag = 1;
    while (flag) {
        int old = atom_xchg(&localAccMutex[0], 1);
        if (old == 0) {
            oldValue = *x;
            *x = baseFunc(*x);
            atom_xchg(&localAccMutex[0], 0);
            flag = 0;
        };
        barrier(CLK_LOCAL_MEM_FENCE);
    };
    return oldValue;
}
```

Предложенное решение имеет и свои недостатки. Так, например, при использовании сгенерированной атомарной функции выделяется дополнительная память под массив мьютексов, что может быть неочевидно для пользователя. Кроме того, решение обладает низкой переносимостью из-за особенностей реализации OpenCL на каждом конкретном устройстве — как показывает практика, нет гарантий того, что генерируемый код будет вести себя единообразно на всех возможных устройствах.

3.2 Поддержка трансфера пользовательских типов данных

В задачах, связанных с обработкой графов, метки ребер могут иметь произвольный тип. В связи с этим стандарт GraphBLAS не налагает ограничений на тип элементов матрицы. Эталонная реализация GraphBLAS на CPU — SuiteSparse — позволяет использовать любою структуру фиксированного размера. Поэтому возникает необходимость обеспечить поддержку использования пользовательских структур данных. Кроме структур, интерес вызывает возможность использования размеченных объединений в ядрах OpenCL. Так, с помощью типа данных `Option` естественно выражается наличие в графе ребра с некоторым весом или его отсутствие. Использование типа `Option` для реализации GraphBLAS может решить проблему «явных нулей»[16], которая вызывает дискуссии в сообществе до сих пор.

Несмотря на то, что транслятор уже умел транслировать некоторые пользовательские типы данных (кортежи, структуры, размеченные объединения), трансфер данных таких типов из управляемой памяти в видеопамять и наоборот реализован не был. Среди аналогичных инструментов лишь ILGPU и FSCL поддерживают использование пользовательских типов данных. Рантайм библиотеки ILGPU способен оперировать любыми типами, которые удовлетворяют ограничению

[16]Обсуждение необходимости фильтровать явные нули: https://github.com/GraphBLAS/LAGraph/issues/28. Дата обращения: 09.03.2022

unmanaged[17]. На самом деле, множество доступных типов еще более ограничено непреобразуемым (blittable) типами — такими типами, которые представлены одинаково в управляемой и неуправляемой памяти. Благодаря этому ограничению становится возможным не копировать данные из управляемой памяти в неуправляемую, а использовать их сразу, защитив предварительно от сборщика мусора с помощью метода `GCHandle.Alloc` с параметром `GCHandleType.Pinned`. Несмотря на то, что ограничение unmanaged позволяет использовать широкий набор примитивных типов, а также обобщенные структуры, некоторые преобразуемые типы, например bool, а также размеченные объединения недоступны для использования в ILGPU. Рантайм FSCL обеспечивает поддержку преобразуемых типов, однако данные таких типов явно копируются из управляемой памяти в неуправляемую, что негативно сказывается на производительности.

Система, реализованная в данной работе, способна обрабатывать как преобразуемые типы данных, так и непреобразуемые. В случае преобразуемых типов данные при трансфере явно копируются из управляемой памяти в неуправляемую. В противном случае копирования не происходит, и данные доступны сразу. Для обеспечения трансфера данных был реализован класс `CustomMarshaler`, который содержит методы, описанные ниже.

- `IsBlittable(Type): bool` — определяет, является ли тип данных непреобразуемым.
- `WriteToUnmanaged('a[]): int` — копирует данные из управляемой памяти в неуправляемую.
- `ReadFromUnmanaged(IntPtr, length: int): unit` — копирует данные из неуправляемой памяти в управляемую.

На данный момент реализована поддержка трансфера следующих типов данных:

[17]Документация, описывающая unmanaged типы: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/unmanaged-types (Дата обращение: 27.05.2022)

- кортежей;
- записей;
- размеченных объединений;
- пользовательских структур.

3.3 Улучшение модели управления памятью

В прежней версии библиотеки выделение памяти на OpenCL устройстве происходило неявно. Такой подход имел ряд существенных недостатков:

- было невозможно выделять память напрямую на OpenCL устройстве;
- было невозможно выставлять нужные флаги буфера при создании;
- было невозможно автоматически освобождать выделенную память.

Все эти факторы негативно сказывались на производительности. Большинство же аналогичных библиотек предоставляют возможности для более тонкого управления памятью и временем жизни выделенных объектов. Так, например, библиотека ILGPU имеет абстракцию верхнего уровня над массивом данных на OpenCL устройстве — `ArrayView`.

Для решения описанных выше проблем были введены абстракции, которые помогли обеспечить необходимый уровень контроля за управлением памятью. Они перечислены ниже.

- `ClBuffer` — класс, который абстрагирует участок памяти на OpenCL устройстве. Согласно идиоме RAII, он инкапсулирует владение ресурсом (памятью в данном случае). Захват ресурса происходит в конструкторе, а освобождение — в методе Dispose интерфейса IDisposable.

- `ClArray` — высокоуровневая обертка над `ClBuffer`, предоставляющая стандартный интерфейс взаимодействия с одномерным массивом данных.

- `ClCell` — высокоуровневая обертка над `ClBuffer`, предоставляющая интерфейс взаимодействия с одноэлементным множеством.

Более подробная иерархия классов изображена на рисунке 6.

3.4 Общая архитектура решения

В настоящей работе была также переработана общая архитектура решения. Необходимость этих изменений продиктована не только связью с обновлением механизмов управления памятью, но также с требованием реализовать нативную поддержку параллельного исполнения OpenCL команд. Ранее такая возможность почти полностью отсутствовала.

Иерархия классов предлагаемого решения изображена на рисунке 7. В предлагаемой архитектуре можно выделить 3 слоя. Самый нижний — слой абстракций внешней библиотеки OpenCL.NET. Эта библиотека предоставляет интерфейс для вызова нативных функций OpenCL.

Над ним располагается слой, целью которого является следующее:

1. обеспечить возможность трансляции F# кода в OpenCL, а также трансфера данных из управляемой памяти в видеопамять;

2. предоставить более удобный интерфейс для взаимодействия с OpenCL;

3. сохранить гибкость используемых абстракций.

В этом слое расположены классы, необходимые для непосредственного взаимодействия F# и OpenCL — транслятор из F# в OpenCL `FSQuotationToOpenCLTranslator` и маршаллер `CustomMarshaler`. Абстракции этого слоя почти полностью повторяют набор абстракций слоя OpenCL.NET, обеспечивая таким образом необходимую гибкость,

однако добавляют также ряд особенностей, призванных упростить взаимодействие с пользователем. Так, например, асинхронная очередь команд `MailboxProcessor<Msg>`, которая является абстракцией более высокого уровня над `OpenCL.Net.CommandQueue`, позволяет удобнее организовать контроль над временем жизни объекта в памяти OpenCL устройства, а также позволяет синхронизировать выполнение операций в нескольких параллельных очередях. За счет этого и достигается требуемая параллельность на уровне исполнения команд.

На самом верхнем уровне располагается слой, который призван упростить модель параллельного программирования, скрыв абстракции OpenCL за функциональным интерфейсом. Для этого было реализовано вычислительное выражение `ClTask`, которое хранит контекст вычислений, а также функции `runSync` и `inParallel`, запускающие вычисления синхронно или параллельно.

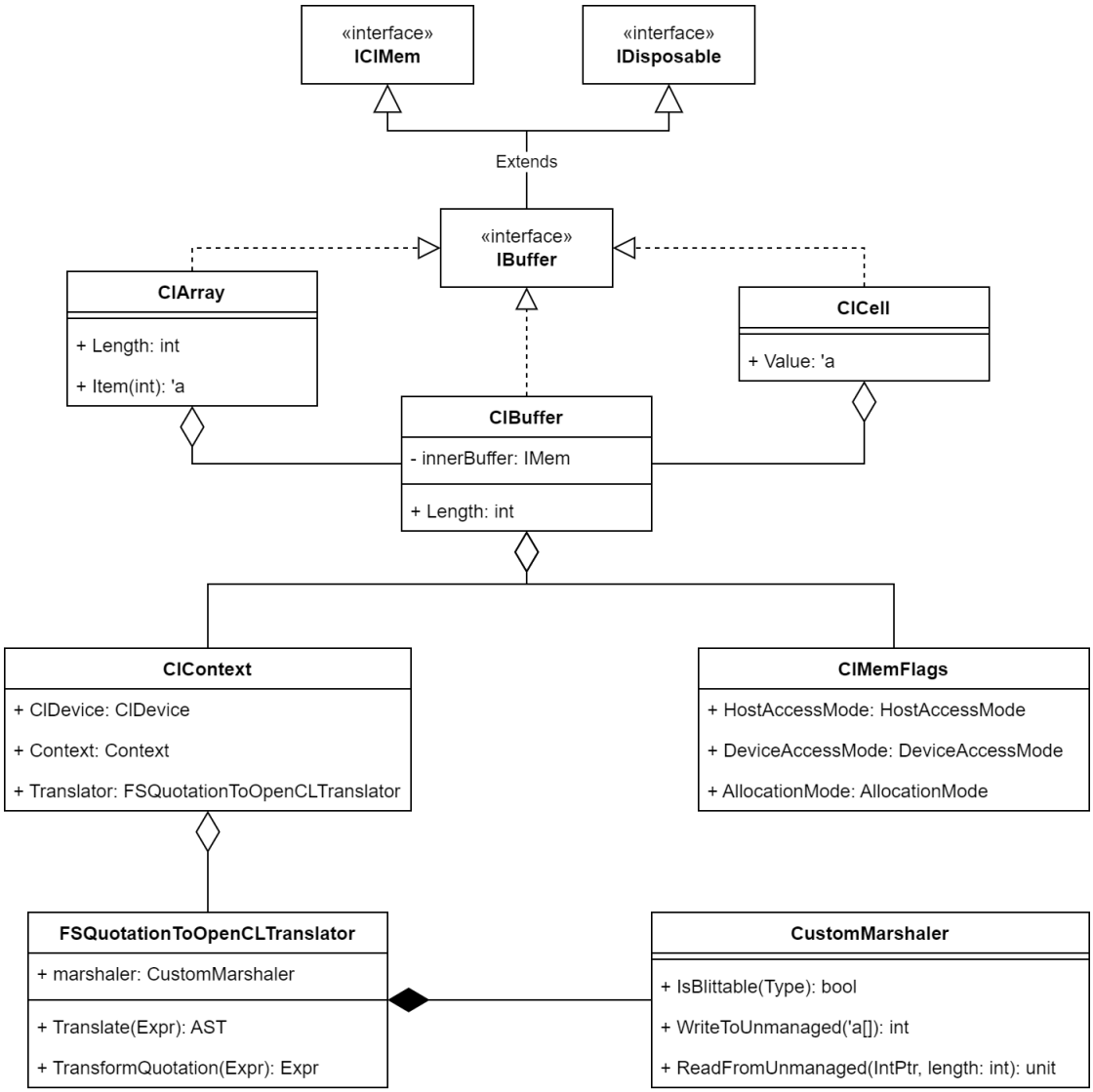

Рис. 6: Иерархия абстракций в модели управления памятью

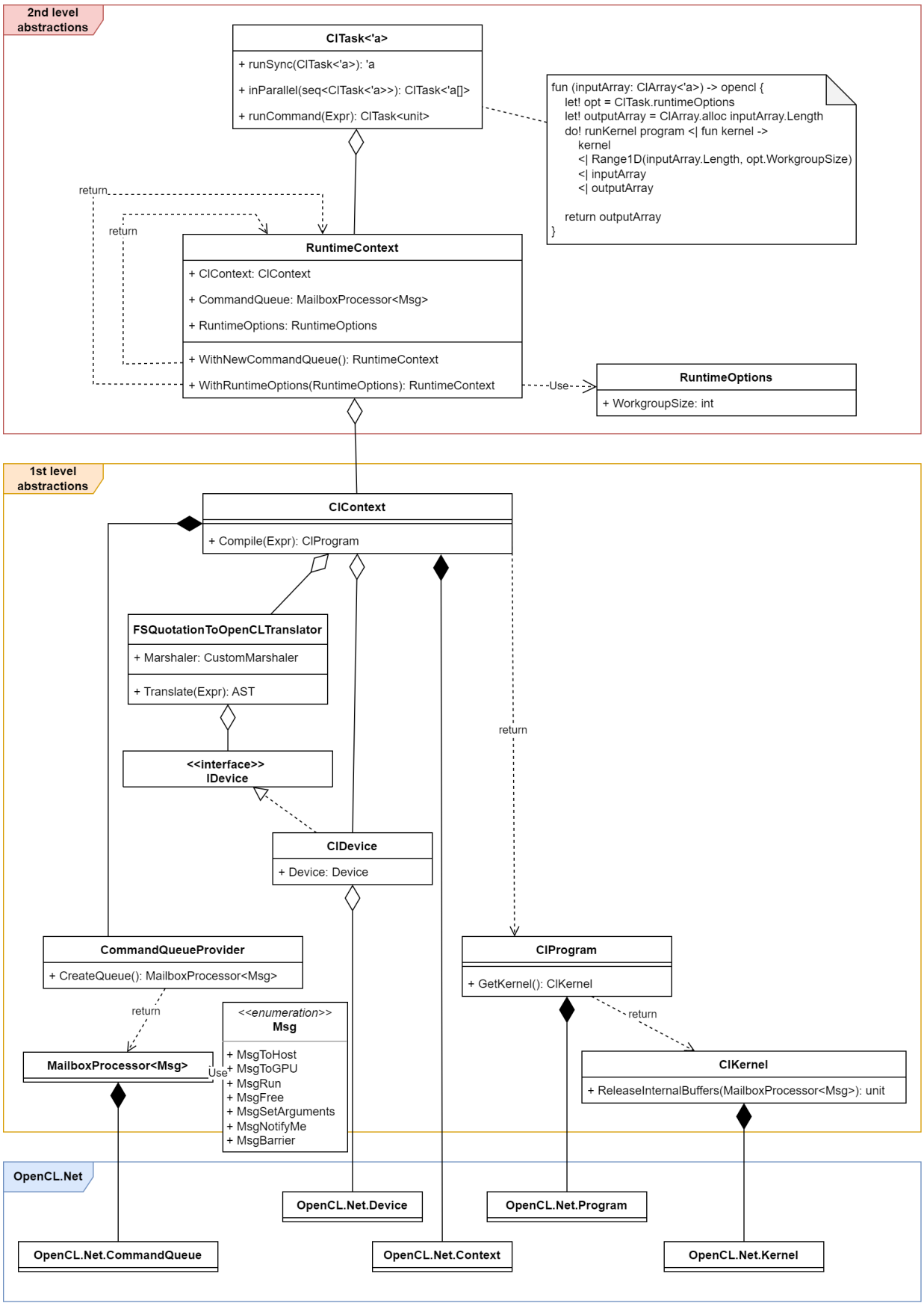

Рис. 7: Общая архитектура решения

4 Эксперимент

В данном разделе приведены описание и результаты экспериментов по оценке производительности предложенных решений в сравнении с аналогами, в которых реализована соответствующая функциональность. Для сравнения были выбраны библиотеки FSCL и ILGPU, поскольку они наиболее близки к предлагаемому решению по набору особенностей. Их сравнение приведено в таблице 1.

Характеристика	**Brahma.FSharp**	**FSCL**	**ILGPU**
Поддержка непреобразуемых типов	Да	Да	Да
Поддержка обобщенных структур данных	Да	*Нет*	Да
Поддержка преобразуемых типов	Да	*Нет*	*Нет*
Поддержка произвольных атомарных операций	Да	*Нет*	Да
Поддержка параллельного исполнения команд	Да	Да	Да

Таблица 1: Сравненение

4.1 Условия эксперимента

Сравнение поводилось с помощью библиотеки BenchmarkDotNet на ПК с операционной системой Ubuntu 20.04 в конфигурации Intel core i7-4790 CPU, 3.6GHz, DDR4 32Gb RAM и GeForce GTX 2070 GPU, 8Gb VRAM.

4.2 Оценка затрат на трансфер данных

Для оценки времени, затрачиваемого на трансфер данных между хостом и OpenCL устройством, были проведены эксперименты, сравнивающие в отдельности следующие характеристики:

1. запись данных в видеопамять;

Библиотека	**Длина массива**	**Среднее, мкс**	**Стандартная ошибка, мкс**
Brahma.FSharp	1000	47.28	20.312
ILGPU	1000	45.87	2.561
FSCL	1000	1,506.0	152.3
Brahma.FSharp	1 000 000	1,214.76	310.448
ILGPU	1 000 000	542.95	40.386
FSCL	1 000 000	4,567.0	885.3

Таблица 2: Оценка времени записи данных в видеопамять, непреобразуемый тип данных, int

Библиотека	**Длина массива**	**Среднее, мкс**	**Стандартная ошибка, мкс**
Brahma.FSharp	1000	49.66	19.347
ILGPU	1000	45.80	2.730
FSCL	1000	1,664.0	220.4
Brahma.FSharp	1 000 000	5,327.74	1,262.829
ILGPU	1 000 000	2,320.40	40.209
FSCL	1 000 000	13,429.0	2,06.0

Таблица 3: Оценка времени записи данных в видеопамять, непреобразуемый тип данных, структура

2. выделение памяти на OpenCL устройстве.

Сущность экспериментов заключалась в оценке времени работы соответствующих операций применительно к некоторому массиву данных. Параметрами эксперимента послужили следующие свойства:

- тип данных массива (непреобразуемый или преобразуемый);
- длина массива;
- используемая библиотека (Brahma.FSharp, FSCL или ILGPU).

4.2.1 Запись данных в видеопамять

Результаты эксперимента приведены в таблицах 4 и 5.

Библиотека	**Длина массива**	**Среднее, мкс**	**Стандартная ошибка, мкс**
Brahma.FSharp	1000	39.47	3.677
ILGPU	1000	45.07	2.172
Brahma.FSharp	1 000 000	2,507.69	292.934
ILGPU	1 000 000	1,082.90	94.608

Таблица 4: Оценка времени записи данных в видеопамять, непреобразуемый тип данных, ValueOption<int>

Библиотека	**Длина массива**	**Среднее, мкс**	**Стандартная ошибка, мкс**
Brahma.FSharp	1000	2,685.18	418.878
Brahma.FSharp	1 000 000	2,044,113.55	48,098.780

Таблица 5: Оценка времени записи данных в видеопамять, преобразуемый тип данных, bool

4.2.2 Выделение памяти на OpenCL устройстве

Результаты эксперимента приведены в таблицах 8 и 9.

4.2.3 Выводы

Из приведенных результатов видно, что предлагаемое решение сравнимо по производительности с аналогичными инструментами в операциях выделения памяти и записи данных непреобразуемых типов. При этом в операциях записи данных преобразумых типов решение существенно медленнее.

Библиотека	**Длина массива**	**Среднее, мкс**	**Стандартная ошибка, мкс**
Brahma.FSharp	1000	43.11	11.056
ILGPU	1000	43.75	8.165
Brahma.FSharp	1 000 000	41.61	15.903
ILGPU	1 000 000	88.97	9.878

Таблица 6: Оценка времени выделения памяти на OpenCL устройстве, непреобразуемый тип данных, int

Библиотека	**Длина массива**	**Среднее, мкс**	**Стандартная ошибка, мкс**
Brahma.FSharp	1000	39.30	16.682
ILGPU	1000	37.92	4.928
Brahma.FSharp	1 000 000	53.89	45.284
ILGPU	1 000 000	111.61	26.629

Таблица 7: Оценка времени выделения памяти на OpenCL устройстве, непреобразуемый тип данных, структура

Библиотека	**Длина массива**	**Среднее, мкс**	**Стандартная ошибка, мкс**
Brahma.FSharp	1000	37.89	19.484
ILGPU	1000	38.47	5.896
Brahma.FSharp	1 000 000	30.46	13.185
ILGPU	1 000 000	92.77	10.086

Таблица 8: Оценка времени выделения памяти на OpenCL устройстве, непреобразуемый тип данных, ValueOption<int>

Библиотека	**Длина массива**	**Среднее, мкс**	**Стандартная ошибка, мкс**
Brahma.FSharp	1000	36.19	10.077
Brahma.FSharp	1 000 000	39.71	10.528

Таблица 9: Оценка времени выделения памяти на OpenCL устройстве, преобразуемый тип данных, bool

Реализация	**Размер пространства**	**Среднее, мкс**	**Стд. ошибка, мкс**
Brahma.FSharp (нативная)	1000	138.96	65.74
Brahma.FSharp (кастомная)	1000	96.41	26.31
ILGPU (нативная)	1000	20.40	8.876
ILGPU (кастомная)	1000	631.24	88.88
Brahma.FSharp (нативная)	100 000	126.49	40.55
Brahma.FSharp (кастомная)	100 000	298.12	42.93
ILGPU (нативная)	100 000	14.21	4.548
ILGPU (кастомная)	100 000	541,701.94	7,757.32
Brahma.FSharp (нативная)	10 000 000	366.2	45.58
Brahma.FSharp (кастомная)	10 000 000	14,484.7	4,588.92

Таблица 10: Оценка затрат на использование атомарных операций

4.3 Оценка затрат на использование атомарных операций

Для оценки временных затрат на применение атомарных операций, основанных на использовании спинлока, были проведены эксперименты, сравнивающие время выполнения такого ядра с временем выполнения ядра, использующего аналогичную нативную реализацию. Кроме того было проведено сравнение с аналогичным решением в библиотеке ILGPU. Параметром эксперимента послужил глобальный размер индексного пространства ядра. Результаты эксперимента приведены в таблице 10.

4.3.1 Выводы

Из приведенных результатов видно, что разница в производительности между нативной реализацией атомарных операций и реализацией, с использованием спинлока, в библиотеке Brahma.FSharp незначительна для малых размеров индексного пространства. Однако с ростом числа агентов, которым необходим эксклюзивный доступ к памяти, производительность сильно снижается. Предлагаемое решение производи-

тельнее аналогичного в библиотеке ILGPU и, кроме того, оно не так сильно деградирует с ростом числа агентов. По это причине производительность кастомной атомарной операции ILGPU при размере массива 10 000 000 установить не удалось.

Заключение

В рамках данной работы[18] были получены перечисленные ниже результаты.

- Реализована поддержка обобщенных атомарных операций.
- Реализована поддержка трансфера обобщенных типов данных, таких как структуры, кортежи и размеченные объединения.
- Улучшена модель управления памятью.
- Реализована возможность параллельного исполнения OpenCL ядер.
- Проведено экспериментальное сравнение предложенной реализации с аналогами.

[18]Репозиторий проекта: https://github.com/YaccConstructor/Brahma.FSharp, имя аккаунта: dpanfilyonok.

Список литературы

[1] CuSha: Vertex-Centric Graph Processing on GPUs / Farzad Khorasani, Keval Vora, Rajiv Gupta, Laxmi N. Bhuyan // Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing. — HPDC '14. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 239–252. — URL: https://doi.org/10.1145/2600212.2600227.

[2] Design of OpenCL Framework for Embedded Multi-core Processors / Jung-Hyun Hong, Young-Ho Ahn, Byung-Jin Kim, Ki Chung // Consumer Electronics, IEEE Transactions on. — 2014. — 05. — Vol. 60. — P. 233–241.

[3] Graphs, Matrices, and the GraphBLAS: Seven Good Reasons / Jeremy Kepner, David Bader, Aydın Buluç et al. // Procedia Computer Science. — 2015. — Vol. 51. — P. 2453–2462. — International Conference On Computational Science, ICCS 2015. URL: https://www.sciencedirect.com/science/article/pii/S1877050915011618.

[4] Khronos OpenCL Working Group. — The OpenCL Specification, Version 1.1, 2011. — URL: https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[5] Mathematical foundations of the GraphBLAS / J. Kepner, P. Aaltonen, D. Bader et al. // 2016 IEEE High Performance Extreme Computing Conference (HPEC). — 2016. — P. 1–9.

[6] Multi-GPU Graph Analytics / Yuechao Pan, Yangzihao Wang, Yuduo Wu et al. — 2017. — 05. — P. 479–490.

[7] Optimizing CUDA Code By Kernel Fusion—Application on BLAS / Jiri Filipovic, Matus Madzin, Jan Fousek, Ludek Matyska // CoRR. — 2013. — Vol. abs/1305.1183. — 1305.1183.

[8] Quantifying the Energy Efficiency of FFT on Heterogeneous Platforms / Yash Ukidave, Amir Kavyan Ziabari, Perhaad Mistry et al. — 2013. — 04.

[9] Wadler Philip. Deforestation: transforming programs to eliminate trees // Theoretical Computer Science. — 1990. — Vol. 73, no. 2. — P. 231–248. — URL: https://www.sciencedirect.com/science/article/pii/030439759090147A.

[10] Yang Carl, Buluç Aydin, Owens John D. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU // CoRR. — 2019. — Vol. abs/1908.01407. — 1908.01407.