УДК 519.68 doi 10.26089/NumMet.v21r325

РАЗРАБОТКА ПРОТОТИПА ВЫСОКОПРОИЗВОДИТЕЛЬНОГО ГРАФОВОГО ФРЕЙМВОРКА ДЛЯ ВЕКТОРНОЙ АРХИТЕКТУРЫ NEC SX–AURORA TSUBASA

И. В. Афанасьев[1,2]

В данной статье описан подход к созданию прототипа графового фреймворка VGL (Vector Graph Library), нацеленного на эффективную реализацию графовых алгоритмов для современной векторной архитектуры NEC SX–Aurora TSUBASA. Современные векторные системы позволяют значительно ускорять приложения, интенсивно использующие подсистему памяти, подклассом которых являются графовые алгоритмы. Однако подходы к эффективной реализации графовых алгоритмов для векторных систем на сегодняшний день исследованы крайне слабо: вследствие сильно нерегулярной структуры графов реального мира, эффективно задействовать векторные особенности целевых платформ затруднительно. В работе показано, что разработанные на основе предложенного фреймворка VGL реализации графовых алгоритмов не уступают в производительности оптимизированным “вручную” аналогам за счет инкапсуляции большого числа оптимизаций графовых алгоритмов, характерных для векторных систем. Вместе с этим предложенный фреймворк позволяет значительно упростить процесс разработки графовых алгоритмов для векторных систем, на порядок сокращая объем кода реализуемых алгоритмов и скрывая от пользователя особенности программирования систем данного класса.

Ключевые слова: NEC SX–Aurora TSUBASA, векторные архитектуры, графовые алгоритмы, графовый фреймворк, графовый API, поиск кратчайших путей в графе, поиск в ширину в графе.

1. Введение. Разработка эффективных реализаций графовых алгоритмов является чрезвычайно важной проблемой современной информатики, поскольку графы крайне удачно моделируют многие объекты реального мира из различных прикладных областей. Так, обработка графов используется при анализе социальных сетей и веб-графов, решении инфраструктурах задач, социально-экономическом моделировании, решении биологических задач, и многих других.

При решении графовых задач традиционно используются суперкомпьютерные системы как с общей, так и с распределенной памятью. Неоспоримым преимуществом систем с распределенной памятью является возможность обработки графов существенно больших размеров, что, однако, достигается ценой значительного снижения производительности [1]. В то же время многие актуальные графовые задачи недостаточно велики, чтобы оправдать распределенные решения с использованием кластеров. Так, например, граф, моделирующий связи между друзьями в социальной сети Facebook, занимает лишь 1,5 ТБ в несжатом виде [2], что позволяет разместить данный граф в памяти многих систем с общей памятью.

Однако далеко не все системы с общей памятью позволяют одинаково быстро и эффективно решать графовые задачи. Известным фактом является то, что большинство графовых задач относится к классу data-intensive, то есть сильно нагружающих подсистему памяти целевых платформ. Вследствие этого, для решения графовых задач больше всего подходят системы с быстрой (HBM) памятью, имеющей пропускную способность порядка 1 ТБайт/с. На сегодняшний день быстрая память устанавливается преимущественно в векторные архитектуры (важным представителем которых является семейство векторных компьютеров NEC SX), либо в архитектуры с векторными расширениями (наборы векторных инструкций AVX–512, AltiVec), так как векторный подход к вычислениям позволяет эффективно задействовать высокую пропускную способность памяти целевых платформ за счет коллективных обращений к подсистеме памяти. Другим важным классом архитектур с быстрой памятью являются графические

[1] Научно-исследовательский вычислительный центр МГУ им. М. В. Ломоносова, Ленинские горы, 119991, Москва, техник, afanasiev_ilya@icloud.com

[2] Московский центр фундаментальной и прикладной математики, afanasiev_ilya@icloud.com

© Научно-исследовательский вычислительный центр МГУ им. М. В. Ломоносова

ускорители NVIDIA GPU, которые, однако, так же обладают векторными особенностями вычислений (варпы).

Подходы к эффективной реализации графовых алгоритмов для векторных систем на сегодняшний день исследованы крайне слабо. Вследствие сильно нерегулярной структуры графов реального мира, эффективно задействовать векторные особенности целевых платформ затруднительно. Из-за значительной сложности создания эффективных реализаций графовых алгоритмов для современных векторных систем возникает необходимость в разработке графовых фреймворков (программных сред для решения графовых задач). Так, например, реализации графовых алгоритмов, описанные в работах [3, 4], насчитывают до 1000 строк кода каждая, причем в каждой из них применяется большое число различных микроархитектурных оптимизаций, позволяющих эффективно использовать аппаратные ресурсы целевых архитектур.

Графовые фреймворки позволяют пользователям реализовывать широкие подклассы графовых алгоритмов на основе несложных последовательных фрагментов кода, описывающих основную вычислительную логику обработки графов. Данный подход к реализации графовых алгоритмов значительно менее трудозатратен, так как не требует от пользователя знания детальных особенностей целевой архитектуры, а лишь знания API (Application Programming Interface) используемого фреймворка. С точки зрения процесса суперкомпьютерного кодизайна графовые фреймворки позволяют скрывать от пользователя сложные вопросы микроархитектурной оптимизации, выбора форматов представления графов и вспомогательных данных, реализации эффективного распараллеливания, использования векторных особенностей систем, и многие другие, позволяя пользователям, не искушенным в вопросах разработки эффективных параллельных программ, создавать высокопроизводительные реализации графовых алгоритмов.

На сегодняшний день востребованность в разработке графовых фреймворков крайне велика: для многоядерных центральных процессоров существуют широко известные фреймворки Ligra [5] и Galois [6], в то время как для графических ускорителей NVIDIA GPU разработаны фреймворки Gunrok [7], CuSHA [8], Medusa [9], Enterprise [10] и др. Однако ни один из существующих на сегодняшний день фреймворков не нацелен на решение графовых задач на современных векторных системах, таких как архитектуры NEC SX–Aurora TSUBASA, A64FX или Intel KNL. В настоящей работе описан прототип разработанного фреймворка VGL (Vector Graph Library), нацеленного как раз на эффективное решение графовых задач на современных векторных системах различных классов. Основной целевой архитектурой данного фреймворка является векторная система NEC SX–Aurora TSUBASA, однако в будущем планируется расширение данного фреймворка для работы на других векторных архитектурах и графических ускорителях NVIDIA GPU.

2. Векторная архитектура NEC SX–Aurora TSUBASA. NEC SX–Aurora TSUBASA на данный момент является новейшей векторной архитектурой семейства SX [11, 12]. В отличие от своих предшественников из серии SX [13, 14], архитектура SX–Aurora TSUBASA состоит из ускорителя vector engine (VE), оснащенного векторным процессором, а также векторного хоста (VH), имеющего в своей основе стандартный узел x86. VE является основным процессором для выполнения векторных приложений, в то время как VH используется в качестве вторичного процессора для выполнения функций базовой операционной системы (ОС), которые выгружаются с VE. VE имеет восемь мощных векторных ядер, каждое из которых обеспечивает производительность 537,6 ГФлоп/с при вычислениях с одинарной точностью, благодаря чему пиковая производительность всего VE достигает 4,3 ТФлоп/с.

Каждое векторное ядро SX–Aurora состоит из трех компонентов: блока скалярной обработки (SPU), блока векторной обработки (VPU) и подсистемы памяти. Большинство вычислений выполняется на векторном блоке, в то время как скалярный блок имеет функциональные возможности типичного скалярного центрального процессора. Поскольку VE SX–Aurora — не типичный графический ускоритель, а самодостаточный процессор, SPU предназначен для обеспечения относительно высокой производительности при скалярных вычислениях. VPU каждого векторного ядра имеет свой собственный относительно простой конвейер команд, предназначенный для декодирования и переупорядочения векторных команд, поступающих из SPU. Декодированные векторные инструкции внутри VPU выполняются на векторно-параллельных конвейерах (VPP). Для хранения результатов промежуточных вычислений каждое векторное ядро оснащено 64 векторными регистрами с общей емкостью регистра, равной 128 КБ. Каждый регистр предназначен для хранения вектора из 256 элементов двойной точности (DP). Подсистема памяти vector engine состоит из шести модулей HBM, что обеспечивает SX–Aurora пропускную способность памяти в 1,22 ТБ/с [13]. Столь высокая пропускная способность памяти способствует более высокой производительности на задачах, интенсивно использующих память, к которым относятся в том числе графовые алгоритмы.

3. Существующие графовые фреймворки и формирование API для векторных систем. Основой любого графового фреймворка является API — программный интерфейс, предоставляющий пользователю возможность реализовывать графовые алгоритмы на основе некоторого заданного фреймворком набора абстракций работы с графами и сопутствующими структурами данных. Перед разработкой удобного и эффективного API графового фреймворка для векторных систем крайне важно провести обзор существующих графовых API для архитектур NVIDIA GPU и multicore CPU, поскольку, как показано в работе [15], современные векторные системы сочетают в себе свойства обоих классов данных архитектур. Далее будут подробно рассмотрены API и основные принципы работы следующих наиболее широко известных графовых фреймворков: Gunrock, Ligra, CuSHA и Medusa.

3.1. Gunrock [7]. В отличие от многих других графовых фреймворков, API Gunrock опирается на абстракции представления данных, а не подхода к вычислениям. Использование абстракций на основе данных позволяет данному фреймворку крайне эффективно производить вычисления на массивно-параллельной архитектуре NVIDIA GPU, создание программ для которой как раз и предполагает описание параллелизма по данным. Вычисления в Gunrock организованы согласно BSP (Bulk Synchronous Parallel) модели, а реализуемые алгоритмы предпочтительно итеративны. Основной абстракцией данных в Gunrock является фронт (frontier) вершин или ребер графа. Фронтом называется некоторое подмножество вершин или ребер графа, активно участвующих в вычислениях на текущем шаге (итерации) алгоритма. Тремя основными примитивами, на основе которых строятся графовые алгоритмы в Gunrock, являются: advance, compute и filter. Наиболее важным является примитив advance, генерирующий новый фронт на основе заданного посредством обхода вершин, смежных к изначальному фронту. При этом как на входе, так и на выходе примитива advance могут быть фронты различных типов — вершин и ребер. Примитив compute выполняет заданную пользователем операцию над каждым элементом заданного фронта вершин или ребер. Примитив Filter генерирует новый фронт вершин или ребер путем удаления части вершин из изначального фронта согласно заданному пользователем критерию. Таким образом, каждую итерацию графового алгоритма, реализованного на основе фреймворка Gunrock, можно описать следующим уравнением:

$$f_out = Op(f_in, G, P), \tag{1}$$

где f_in, f_out — входной и выходной фронты, состоящие из вершин или ребер, а $Op(f, G, P)$ — некоторая операция над входным фронтом f и графом G, а также некоторыми дополнительными, определяемыми пользователем структурами данных P (например, массивом кратчайших расстояний до каждой из вершин графа). Важными атрибутами примитивов являются функторы-обработчики, вызываемые при обходе различных элементов графа. В Gunrock используются следующие функторы: CondEdge, ApplyEdge, CondVertex, ApplyVertex. Cond-функторы возвращают переменную типа bool и используются для фильтрации фронтов вершин и ребер, в то время как Apply-операции позволяют определить действия над вершинами и ребрами, производимые при обходе графа при помощи примитива advance или compute. Хранение графов в Gunrock осуществляется на основе CSR (Compressed Sparse Row) формата для операций, работающих с фронтами вершин, а также на основе формата списка ребер для операций, работающих с фронтами ребер. Хранение всех данных осуществляется на основе SoA (Structure of Arrays — структуры массивов) для осуществления coalesced-доступов в память.

3.2. Ligra [5]. API фреймворка Ligra так же опирается на абстракции представления данных. Основной абстракцией данных в Ligra является подмножество вершин (vertexSubset). Подмножество вершин в Ligra аналогично фронту из вершин в Gunrock, однако их представление и реализация достаточно сильно отличаются: так, в Ligra используются два типа представления подмножеств вершин — плотные и разреженные. Плотные подмножества вершин реализованы на основе массива флагов типа bool, в то время как разреженные — на основе списков из номеров вершин. Кроме того, в Ligra нет понятия фронта ребер: все операции определяются над подмножествами вершин и смежными с ними ребрами. Над подмножествами вершин в Ligra определены операции двух типов: edgeMap и vertexMap. EdgeMap является аналогом примитива advance Gunrock и осуществляет обход всех ребер графа, смежных к заданному на входе подмножеству вершин. EdgeMap так же имеет две версии — EdgeMapSparse и EdgeMapDense для плотных и разреженных подмножеств вершин соответственно. Аналогично, операция vertexMap осуществляет обход вершин графа (или из заданного подмножества) — аналог примитива compute из Gunrock.

Хранение графов в Ligra осуществляется на основе CSR формата, причем для ориентированных графов хранятся как входящие, так и исходящие дуги, что позволяет эффективно реализовать операцию транспонирования графа посредством изменения ролей данных массивов. Так же в Ligra реализо-

вано хранение взвешенных графов, причем веса хранятся парами с ID вершин для улучшения локальности.

3.3. CuSHA [8]. API фреймворка CuSHA основан на вычислительной абстракции GAS (Gather–Apply–Scatter), в которой каждая итерация алгоритма состоит из этапов gather/read, update/compute и scatter/write. На этапе gather/read каждая из вершин получает данные о состоянии соседних вершин, на этапе compute/update обрабатывается собранная информация и производятся необходимые вычисления, после чего на этапе scatter/write информация рассылается по соседним вершинам. Данная модель применима как для систем с распределенной, так и общей памятью: для распределенных систем на этапах gather/scatter производится пересылка сообщений, в то время как для систем с общей памятью на этапах read/write коммуникации производятся с использованием основной памяти.

Хранение графов в CuSHA осуществляется на основе форматов G-Shards и Concatenated Windows (CW), что позволяет данному фреймворку оптимизировать шаблоны доступа в память. Однако используемая вычислительная абстракция GAS может одинаково эффективно применяться и для графов, хранимых в CSR формате.

3.4. Medusa [9]. API фреймворка Medusa основан на модели пересылки сообщений между вершинами графа. Данный фреймворк использует механизмы буферов для отправки и приема сообщений между различными вершинами вдоль ребер, а также набор вычислительных функций для агрегации сообщений и сопутствующих вычислений.

На момент написания статьи Gunrock имеет наиболее высокую производительность из рассмотренных фреймворков для графических ускорителей. Абстракции данных, используемые в Gunrock, позволяют фреймворку эффективно задействовать массивный параллелизм GPU, а также сократить накладные расходы на поддержание сложных структур данных, необходимых для других моделей (например, пересылки сообщений). Абстракции фреймворка Gunrock (как вычислительные, так и данных) потенциально хорошо подходят в том числе и для векторных процессоров, имеющих схожую модель вычислений с графическими ускорителями. Кроме того, для реализации данных примитивов может быть эффективно задействована векторная обработка данных.

Вследствие этого, предложенный в данной работе фреймворк VGL использует API, достаточно сильно схожий с API Gunrock. Основными вычислительными абстракциями фреймворка VGL так же являются функции advance и compute, однако с некоторыми принципиальными обобщениями, уточнениями и дополнениями для более эффективной работы на целевых векторных системах. При этом как внутренние особенности вычислительных примитивов и структур данных, так и значительная часть функциональности разработанного фреймворка VGL имеют кардинальные отличия от реализации в Gunrock.

Важно отметить, что помимо рассмотренных в данном обзоре графовых фреймворков, существуют также и графовые библиотеки — NVGraph, Boost, Galois и др. Однако данные библиотеки не предоставляют API, позволяющий пользователям реализовывать широкий класс произвольных графовых алгоритмов.

4. Описание и характеристики основных функций фреймворка VGL. В данном разделе приведено подробное описание основных вычислительных абстракций и абстракций над данными предлагаемого фреймворка VGL. Центральными абстракциями над данными в разрабатываемом фреймворке являются непосредственно входной граф и подмножество (фронт) вершин графа. Основными вычислительными абстракциями являются следующие примитивы: *advance, generate_new_frontier, compute, filter, reduce.* Каждый из данных примитивов оперирует с одной или несколькими абстракциями данных — графом или же фронтом вершин.

4.1. Абстракции данных: граф и его представление. Формат хранения графа в фреймворке VGL основан на формате VectCSR, описанном в работе [3]. Хранение графа производится в формате CSR с векторным расширением, что позволяет организовать эффективную обработку заданного подмножества вершин и ребер графа и, кроме того, гарантировать эффективный шаблон доступа к памяти с использованием векторных инструкций максимальной длины при обходе вершин и ребер за счет использования векторного расширения. Фреймворк VGL поддерживает хранение как ориентированных, так и неориентированных графов. При работе с неориентированными графами каждая из дуг хранится в виде двух ориентированных дуг различных направлений, причем дуги для каждой вершины хранятся единым списком (как входящие, так и исходящие). Для ориентированных графов для каждой из вершин хранятся только лишь списки исходящих вершин, вследствие чего для эффективной работы pull-based [16] алгоритмов на данный момент необходимо предварительное транспонирование графа. Оптимизация формата хранения ориентированных графов для эффективной поддержки реализации push-based [16] и pull-based графовых

алгоритмов является одним из приоритетных направлений дальнейшей работы. Так же фреймворк VGL поддерживает хранение взвешенных графов, причем веса и id направляющих вершин ребер хранятся в виде структуры массивов для эффективной загрузки данных о ребрах векторными устройствами.

4.2. Абстракции данных: фронт (подмножество) вершин графа. Фронт (подмножество) вершин графа в фреймворке VGL реализован при помощи класса Frontier, конструктор которого принимает на вход единственный параметр — максимальное количество вершин в данном фронте, обычно равное числу вершин в графе, с которым осуществляются вычисления. Фронт вершин может иметь один их трех типов:

- *ALL_ACTIVE_FRONTIER* — все вершины графа принадлежат данному фронту. Данный тип важен для "all-active" графовых алгоритмов, которые эффективно реализуются на векторных системах. Применение данного типа минимизирует накладные расходы на поддержание списков активных вершин.
- *DENSE_FRONTIER* — доля вершин, принадлежащих фронту, от общего числа вершин в графе достаточно велика. Хранение вершин фронта данного типа осуществляется при помощи массива флагов принадлежности к фронту (*IN_FRONTIER_FLAG, NOT_IN_FRONTIER_FLAG*).
- *SPARSE_FRONTIER* — фронт вершин состоит лишь из небольшого количества вершин, доля которых от общего числа вершин в графе мала. Вершины фронта данного типа хранятся в виде списка индексов принадлежащих к фронту вершин.

При работе с вычислительными примитивами, изменяющими состав фронта (*filter, generate_new_frontier*), во фреймворке VGL реализовано автоматическое переключение между плотным и разреженным типами фронта согласно заложенным во фреймворк критериям: фронт вершин считается разреженным, если в нем находится менее 20% от общего числа вершин графа. Пользователь фреймворка может также задавать произвольный критерий переключения между разреженным и плотными типами фронтов, в том числе основанный на количестве исходящих (или входящих) из (в) фронта ребер. Стандартный критерий разреженности фронта выбран как наиболее эффективный на основе экспериментов с предварительной реализацией алгоритмов BFS (Breadth-First Search), SSSP (Single Source Shortest Paths), PR (Page Rank) и CC (Connected Components) на основе разработанного фреймворка.

Важно отметить, что во фреймворке VGL не предусмотрено объекта отдельного типа, отвечающего за хранение подмножества ребер графа. Это обусловлено тем, что операция advance обхода ребер, смежных к заданному фронту вершин, на основе графа в формате VectCSR для векторных систем значительно эффективнее операции обхода подмножества ребер графа в формате списка ребер (*edges_list*). Кроме того, генерация разреженных подмножеств вершин осуществляется значительно быстрее генерации разреженных подмножеств ребер.

4.3. Вычислительные абстракции: *advance*. Примитив *advance* является основным способом обхода графа в фреймворке VGL. Для многих графов реального мира поток вычислений данного примитива имеет крайне нерегулярную структуру, вследствие чего его эффективная реализация для векторных архитектур представляет собой наибольшую трудность. Примитив *advance* принимает на вход граф и заданное подмножество его вершин (*frontier*), а также несколько определяемых пользователем функций-обработчиков: *vertex_preprocess_op, edge_op, vertex_postprocess_op*. Для каждой из вершин фронта вначале выполняется операция *vertex_preprocess_op*, затем для каждого исходящего из данной вершины ребра выполняется операция *edge_op*, после чего для каждой из вершин выполняется завершающая операция *vertex_postprocess_op*. Важно отметить параллельный характер применения операций *edge_op* для смежных ребер заданной вершины. Общая схема применения данных операций к вершинам фронта графа представлена на рис. 1.

Для векторной архитектуры NEC SX–Aurora TSUBASA крайне важен принцип обращений к памяти внутри одной векторной инструкции. При обходе графа с использованием примитива *advance* обработка вершин векторными инструкциями производится двумя принципиально различными способами: вершины с низкой степенью обрабатываются коллективно (одна векторная инструкция на 256 вершин), в то время как каждая из вершин с высокой степенью связанности обрабатывается индивидуально (одна или несколько векторных инструкций на одну вершину). Для различных типов вершин шаблон доступа к памяти может кардинально отличаться. К примеру, обновление состояния коллективно обрабатываемых вершин реализуется записью различных элементов векторной инструкции в различные ячейки памяти

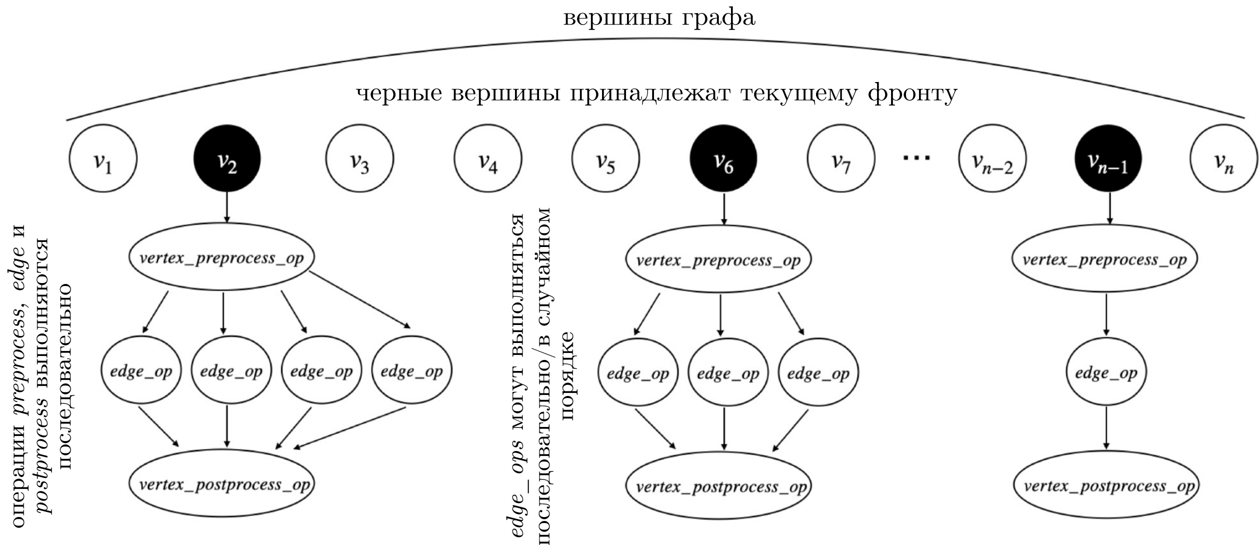

Рис. 1. Схема применения операций *vertex_preprocess_op, edge_op, vertex_postprocess_op* примитива *advance* к активным вершинам фронта (отмечены черным цветом)

(шаблон записи без конфликтов), в то время как для индивидуально обрабатываемых вершин — в одну и ту же (шаблон записи с конфликтом). Поэтому во фреймворке VGL пользователю предоставлена возможность определять различные обработчики для двух классов вершин (коллективно и индивидуально обрабатываемых), вследствие чего примитив *advance* имеет прототипы, приведенные в листинге 1.

Каждая из операций, передаваемых в примитив *advance*, может определяться пользователем либо при помощи lambda-функций, либо с использованием функторов языка C++. Данные инструменты языка C++ позволяют захватывать как некоторые переменные и данные фреймворка, характеризующие структуру графа и фронта вершин, так и произвольное количество данных пользователя (механизм lambda capture). Прототипы операции над ребрами и вершинами графа, передаваемые в примитив *advance*, приведены в листинге 2.

Несложно видеть, что почти все принимаемые данными операциями переменные характеризуют какую-либо из характеристик графа — индекс обрабатываемой вершины и ее степень связанности, индекс обрабатываемого ребра, и др. Однако данные операции также принимают два дополнительных параметра, позволяющих производить более эффективную векторизацию. Параметр *vector_index* характеризует текущий индекс внутри векторных инструкций, что позволяет организовать бесконфликтную работу с промежуточными данными, хранение которых осуществляется на векторных регистрах. Примером может служить операция обработки ребра в алгоритме поиска кратчайших путей Дейкстры, приведенная в листинге 3. Векторизация данного примера будет производиться компилятором неэффективно, так как при коллективной обработке различных ребер одной векторной инструкцией неизбежно будет возникать конфликт при записи в скалярную переменную *changes*, отвечающую за наличие произошедших изменений на текущей итерации алгоритма.

Исправить ситуацию позволяет переменная *vector_index*, позволяющая организовать бесконфликтную запись в векторные массивы-регистры; API для работы с которыми описан в последующих разделах.

Листинг 1. Варианты прототипов примитива advance

```
void advance(Graph<_TVertexValue, _TEdgeWeight> &_graph,
             Frontier &_frontier,
             EdgeOperation &&edge_op,
             VertexPreprocessOperation &&vertex_preprocess_op,
             VertexPostprocessOperation &&vertex_postprocess_op,
             CollectiveEdgeOperation &&collective_edge_op,
             CollectiveVertexPreprocessOperation &&collective_vertex_preprocess_op,
             CollectiveVertexPostprocessOperation &&collective_vertex_postprocess_op);
```

Листинг 2. Прототипы операций, принимаемых примитивом advance, определяемые lambda-функциями. Для операций vertex_preprocess_op и vertex_postprocess_op используется идентичный формат.

```
auto EDGE_OP = [] (int src_id,
                   int dst_id,
                   int local_edge_pos,
                   long long int global_edge_pos,
                   int vector_index,
                   DelayedWriteNEC &delayed_write);

auto VERTEX_OP = [] (int src_id,
                     int connections_count,
                     int vector_index,
                     DelayedWriteNEC &delayed_write);
```

Листинг 3. Пример конфликта по записи при векторизации в алгоритме поиска кратчайших путей Дейкстры

```
if(dst_weight > src_weight + weight)
{
    _distances[dst_id] = src_weight + weight;
    changes = true;
}
```

Листинг 4. Пример бесконфликтной записи в алгоритме поиска кратчайших путей Дейкстры

```
if(dst_weight > src_weight + weight)
{
    _distances[dst_id] = src_weight + weight;
    reg_changes[vector_index] = 1;
}
```

Пример бесконфликтной реализации данного алгоритма приведен в листинге 4: запись о наличии изменений производится в специализированный векторный регистр, с последующей его редукцией в скаляр по завершению итерации алгоритма с использованием специализированных функций API для работы с векторными регистрами.

Специализированная структура *DelayedWriteNEC* позволяет избежать аналогичных конфликтов при доступе к глобальной памяти. Так, если бы в двух предыдущих примерах запись в массив дистанций производилась по индексу *src_id* вместо *dst_id*, то для многих вершин графа (обрабатываемых не коллективно) происходил бы конфликт из-за записи данных в одну ячейку глобальной памяти. *DelayedWriteNEC* позволяет производить аккумуляцию промежуточных результатов при обходе ребер во временных структурах данных, с последующей отложенной бесконфликтной записью внутри операции *vertex_postprocess_op*.

4.4. Вычислительные абстракции: *generate_new_frontier*. Примитив *generate_new_frontier* позволяет генерировать новый фронт (подмножество) вершин графа на основании заданного пользователем условия. Данный примитив принимает на вход граф вместе с заданным пользователем условием cond, на выходе же генерирует фронт вершин, для которых условие cond возвращает флаг *IN_FRONTIER_FLAG*. Прототип примитива *generate_new_frontier* вместе с прототипом условия *cond* принадлежности вершины фронту приведены в листинге 5.

Примитив *generate_new_frontier* осуществляет генерацию фронта вершин в три этапа. На первом этапе для каждой из вершин заполняется массив флагов принадлежности вершин к фронту на основании заданного пользователем условия cond, причем производится обход всех вершин графа. На втором этапе производится оценка числа вершин в сгенерированном фронте, после чего принимается решение о принадлежности фронта к плотному или разреженному типу на основании критерия, заданного по умолчанию во фреймворке или пользователем. В случае, если фронт вершин — разреженный, для него дополнительно генерируется список индексов вершин, принадлежащих фронту с использованием оптимизированной операции *scan* (префиксной суммы).

Листинг 5. Прототипы примитива *generate_new_frontier* и условия принадлежности вершины *cond*

```
void generate_new_frontier(Graph<_TVertexValue, _TEdgeWeight> &_graph,
                           FrontierNEC &_frontier,
                           Condition &&cond);

auto FRONTIER_CONDITION = [] (int src_id)->int;
```

Листинг 6. Прототип примитива *compute*

```
void compute(Graph<_TVertexValue, _TEdgeWeight> &_graph,
             FrontierNEC &_frontier,
             ComputeOperation &&compute_op);

auto COMPUTE_OP = [] (int src_id, int connections_count, int vector_index);
```

Потенциальным недостатком данной схемы реализации примитива *generate_new_frontier* является необходимость обхода всех вершин графа для генерации массива флагов принадлежности вершин к фронту, что, вообще говоря, вычислительно не оптимально в случае необходимости генерации большого числа сильно разреженных фронтов, так как для генерации фронта по результатам работы примитива *advance* достаточно обойти лишь все смежные ребра к исходному фронту, подаваемому на вход примитиву *advance.* Для более эффективной обработки данной ситуации в дальнейшем предполагается реализовать вариант примитива advance, на выходе генерирующий фронт вершин вычислительно оптимальным способом для сильно разреженных случаев. Кроме того, в дальнейшем рассматривается возможность полного объединения примитивов advance и *generate_new_frontier*, как это сделано в Gunrock.

4.5. Вычислительные абстракции: *filter*. Примитив *filter* имеет схожую функциональность с примитивом *generate_new_frontier,* но с рядом принципиальных отличий, проиллюстрированных на рис. 2. На вход данный примитив принимает граф, фронт вершин A и задаваемое пользователем условие *cond* (с прототипом варианту для примитива *generate_new_frontier*), на выходе же генерирует новый фронт вершин B, с вершинами, принадлежавшими фронту A и удовлетворяющими условию *cond.* Важным преимуществом данного примитива является тот факт, что вычисления на этапе генерации массива флагов выполняются не над всеми вершинами графа, а только над вершинами, принадлежащими входному фронту A. В результате при использовании примитива *filter* возможен сильный выигрыш в производительности для сильно разреженных фронтов вершин, из которых необходимо исключить часть элементов. В дальнейшем будет более детально исследована возможность объединения примитивов *filter* и *generate_new_frontier* с добавлением некоторых дополнительных флагов, отвечающих за различия в принципах работы данных примитивов (для чего необходимо расширение количества графовых алгоритмов, реализованных на основе фреймворка).

4.6. Вычислительные абстракции: *compute*. Примитив *compute* применяет заданную пользователем операцию *compute_op* к каждой из вершин, принадлежащих заданному фронту. Прототип примитива *compute* и операции *compute_op* приведен в листинге 6, а схема, иллюстрирующая принцип работы примитива *compute*, приведена на рис. 3, *а*.

4.7. Вычислительные абстракции: *reduce*. Примитив *reduce* выполняет редукцию значений, возвращаемых заданной пользователем операции *reduce_cond* на основе операции редукции *reduce_op.* Данный примитив может применяться для оценки числа вершин в последующих фронтах или же вычисления вклада *dangling*-узлов в алгоритме *page rank.* Важно отметить, что в существующих фреймворках (например, Gunrock) функциональность примитива *reduce* может быть получена при помощи комбинации примитивов *compute* и *filter* с использованием атомарных операций. Однако для векторной архитектуры NEC SX–Aurora TSUBASA использование атомарных операций крайне не эффективно, в то время как операция редукции реализуется на них значительно эффективнее, чем на графических ускорителях [15] (вследствие различий в количестве вычислительных ядер, а также в принципах коммуникации и синхронизации между ними), что обуславливает необходимость создания отдельного примитива для данной операции. Схема, иллюстрирующая принцип работы примитива reduce, приведена на рис. 3, *б*.

5. Программная структура фреймворка VGL. Ядро разработанного фреймворка VGL состоит из трех основных компонент: векторного формата хранения графа (и интерфейса работы с ним), векторного

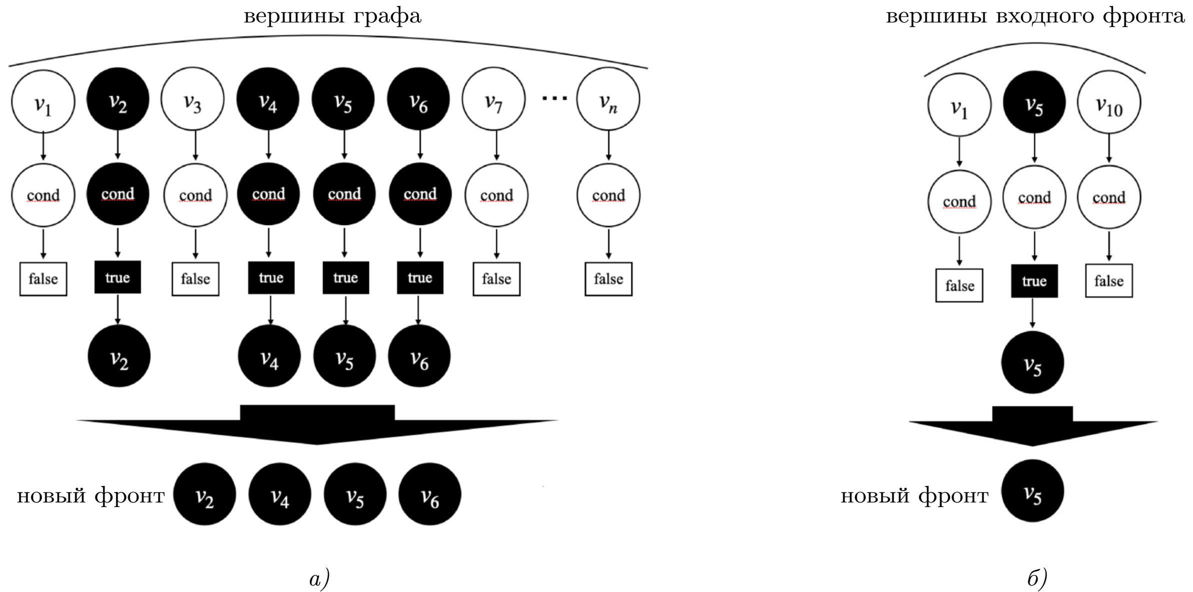

Рис. 2. Схемы работы и основные отличия примитивов
а) generate_new_frontier; б) filter

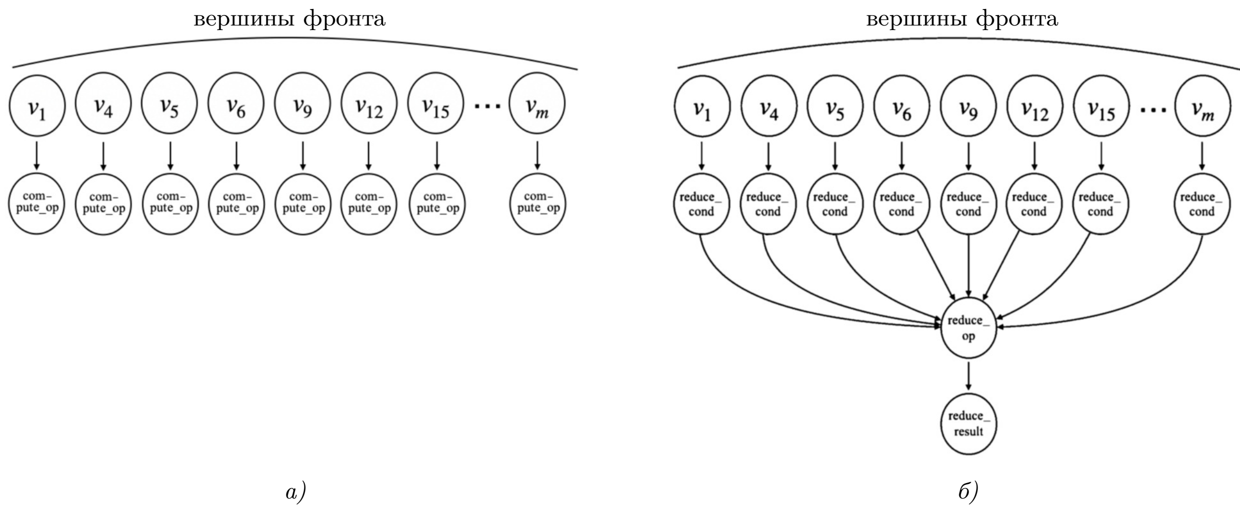

Рис. 3. Схема работы примитивов: *а) compute; б) reduce*

ного формата представления фронта вершин (и интерфейса работы с ним), а также набора описанных в предыдущем разделе вычислительных примитивов. Взаимодействие между различными элементами фреймворка организовано следующим образом: различные экземпляры объектов данных (графа и фронтов вершин) передаются в вычислительные примитивы, имеющие доступ ко всем внутренним структурам данных этих объектов на основе механизма дружественных классов языка C++. Пользователь же фреймворка имеет крайне ограниченный доступ к внутренним структурам данных фреймворка, описывая графовые алгоритмы в терминах предложенных примитивов и некоторых дополнительных методов работы с объектами данных (public API, рис. 4). Подобные дополнительные методы, например *frontier.size()* или *frontier.set_all_active(),* призваны снизить накладные расходы, связанные с необходимостью использования примитивов *reduce, generate_new_frontier* и *filter* для реализации аналогичной функциональности. Реализация данных методов основана на использовании дополнительных флагов и переменных состояния внутри класса *frontier,* что позволяет не производить дополнительных вычислений над графами.

5.1. Схема использования фреймворка VGL для реализации графовых алгоритмов. Типовые схемы использования примитивов фреймворка VGL приведены на рис. 5. Для “all–active” графовых

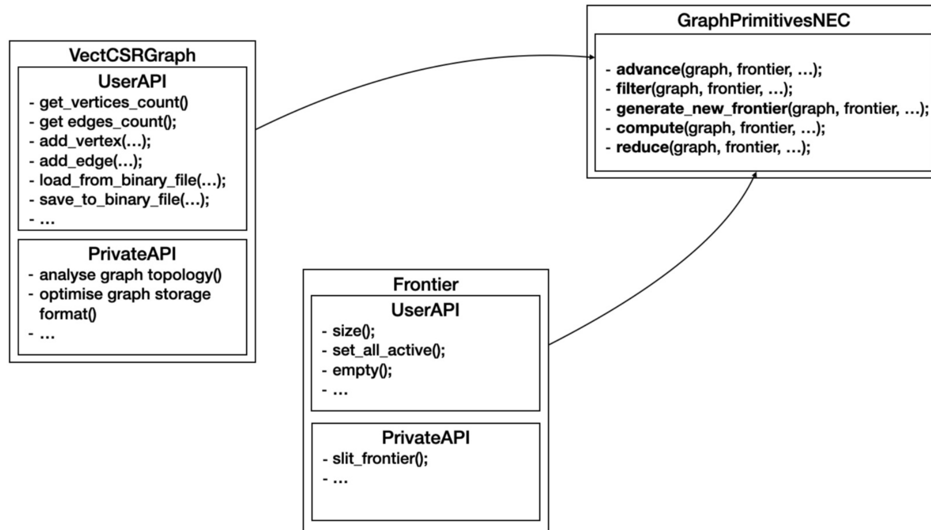

Рис. 4. Программная структура фреймворка VGL

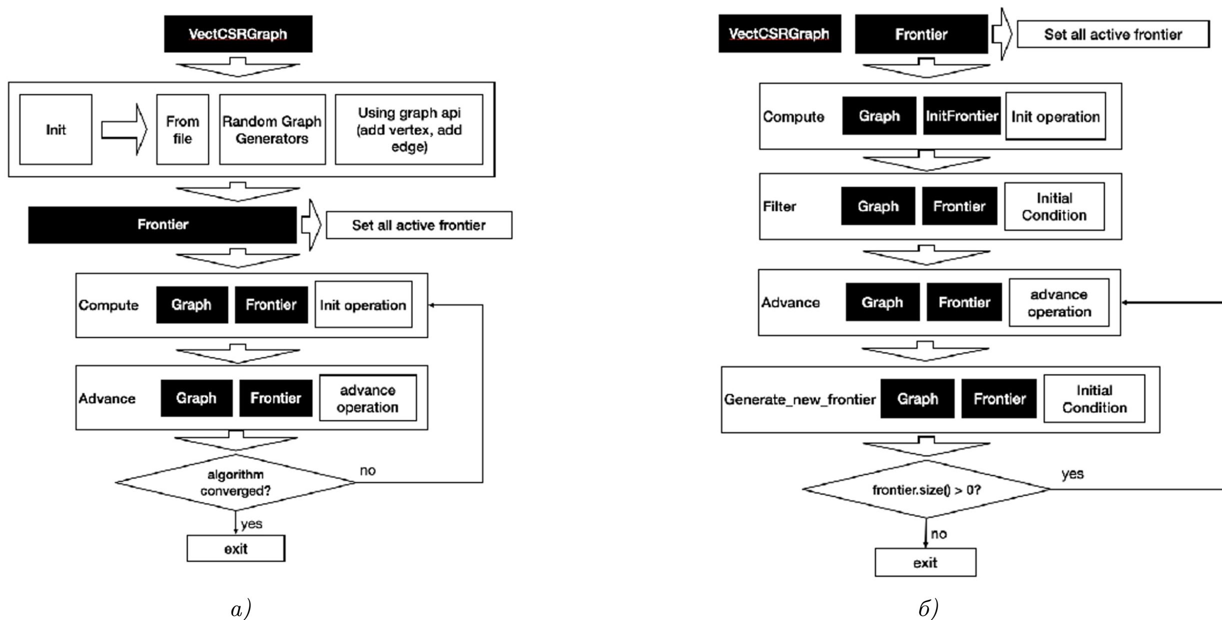

Рис. 5. Типовые схемы использования API фреймворка VGL для реализации графовых алгоритмов: *а)* “all-active”; *б)* “partial-active”

алгоритмов (Дейкстры, page rank, Шилоаха–Вишкина и др.) вначале производится инициализация графа, после чего во фронте все вершины помечаются как активные (участвующие в вычислениях). Затем с помощью примитива compute инициализируются начальные данные, к примеру изначальные дистанции для задачи поиска кратчайших путей. Далее выполняется основная вычислительная часть алгоритма — последовательность вызовов примитива advance до тех пор, пока не будет выполнен некоторый критерий остановки (сходимости) алгоритма.

Первые шаги “partial–active” алгоритмов (например, поиска в ширину) идентичны, однако основная вычислительная часть имеет отличия: примитив advance чередуется с примитивом генерации нового фронта вершин, в то время как критерий остановки зависит от числа оставшихся во фронте вершин.

6. Пример использования API фреймворка VGL для реализации алгоритмов поиска кратчайших путей и поиска в ширину. Использование разработанного фреймворка VGL будет рассмотрено применительно к реализации двух фундаментальных графовых алгоритмов: алгоритма top–down поиска в ширину и обобщенного алгоритма Дейкстры поиска кратчайших путей в графе. Данные алгоритмы были выбраны как простейшие, представляющие два обширных класса алгоритмов: “partial–

active” и “all-active” соответственно. Пример реализации алгоритма поиска в ширину приведен в листинге 7.

В приведенном листинге используются несколько принципиальных техник оптимизации, необходимых для написания эффективных программ с использованием API фреймворка VGL. Во-первых, для подсчета количества произошедших на каждой итерации изменений используется специальный векторный регистр NEC_REGISTER_INT вместо скалярной переменной changes, что позволяет производить бесконфликтные векторные записи при обходе графа. Во-вторых, для устранения конфликтов доступа к данному векторному регистру от различных векторных ядер используется параллельный регион openMP, создающий локальные копии векторного регистра changes для каждой из нитей, а по окончании каждой из итераций редуцирующий количество произошедших изменений с использованием атомарных операций. Реализовать алгоритм Дейкстры можно было бы и без использования двух данных оптимизаций: однако эффективность подобной реализации будет существенно ниже. Так, для RMAT графа с 8 миллионами вершин и средней степенью связанности 32 производительность у оптимизированного варианта кода в 6 раз выше.

В листинге 8 приведен пример реализации алгоритма top–down поиска в ширину в графе. Можно выделить следующие принципиальные отличия от предыдущего примера: необходимость в генерации фронта вершин перед каждой итерацией, условие остановки алгоритма, зависящее от числа вершин в текущем фронте, а также отсутствие необходимости применения описанных ранее оптимизаций, так как в вычислительной операции edge_op не используются разделяемые скалярные переменные.

Исходя из двух рассмотренных примеров реализации графовых алгоритмов можно сделать следующие выводы. Во-первых, разработанный фреймворк позволяет скрыть от пользователя большинство особенностей реализации графовых алгоритмов на векторных архитектурах: балансировку параллельной нагрузки, векторизацию с использованием максимальной длины вектора, оптимизацию шаблонов доступа

Листинг 7. Пример реализации обобщенного алгоритма Дейкстры поиска кратчайших путей в графе с использованием API разработанного фреймворка VGL

```
frontier.set_all_active();
auto init_distances = [_distances, _source_vertex] (int src_id, int connections_count,
                                                    int vector_index)
{
    if(src_id == _source_vertex)
        _distances[_source_vertex] = 0;
    else
        _distances[src_id] = FLT_MAX;
};
graph_API.compute(_graph, frontier, init_distances);
int changes = 1;
while(changes)
{
    changes = 0;
    #pragma omp parallel
    {
        NEC_REGISTER_INT(changes, 0);
        auto edge_op= [outgoing_weights, _distances, &reg_changes](int src_id, int dst_id,
                                        int local_edge_pos, long long int global_edge_pos,
                                        int vector_index, DelayedWriteNEC &delayed_write)
        {
            float weight = outgoing_weights[global_edge_pos];
            if(_distances[dst_id] > _distances[src_id] + weight)
            {
                distances[dst_id] = _distances[src_id] + weight;
                reg_changes[vector_index]++;
            }
        };
        graph_API.advance(_graph, frontier, edge_op_push);

        #pragma omp atomic
        changes += register_sum_reduce(reg_changes);
    }
}
```

к памяти, и многие другие. Однако, несмотря на это, для достижения максимальной производительности с использованием представленного API все же важно учитывать ряд особенностей векторизации графовых алгоритмов, описанных ранее. Во-вторых, представленный API позволяет существенно сократить количества строк кода, необходимых для описания алгоритма. Так, оптимизированные "вручную" версии приведенных в данном разделе алгоритмов имеют объем 500–1000 строк кода, предложенные в работе [3], в то время как реализации, использующие API фреймворка VGL — менее 50 строк, что позволяет существенно сократить время на разработку, отладку и оптимизацию графовых алгоритмов.

7. Оценка эффективности разработанного фреймворка. В данном разделе приведено сравнение производительности фреймворка VGL с двумя высокопроизводительными библиотеками для графических ускорителей и многоядерных центральных процессоров — NVGraph и Ligra. Так как данная статья описывает прототип фреймворка, сравнение приводится только для реализаций алгоритмов решения задачи поиска кратчайших путей в графе (рис. 6). Комплексное сравнение производительности реализаций для других алгоритмов и задач с другими графовыми библиотеками и фреймворками является одним из наиболее приоритетных направлений дальнейших исследований.

Реализации на основе фреймворка VGL многократно тестировались на векторном ускорителе NEC SX–Aurora TSUBASA Type 10B, NVGRAPH — на графических ускорителях NVIDIA V100, Ligra — на многоядерных центральных процессорах Intel(R) Xeon(R) CPU E3–1230 v6. Сравнение приведено для синтетических RMAT [17] и равномерно-случайных графов различного масштаба (имеющих от 2^{20} до 2^{26} вершин), а также для некоторых графов реального мира [18, 19] на основе метрики производительности графовых алгоритмов TEPS (Traversed Edges Per Second) [20]. Все синтетические графы, используемые в данном сравнении,— ориентированные и имеют среднюю степень связанности каждой из вершин рав-

Листинг 8. Пример реализации алгоритма Top–Down поиска в ширину в графе с использованием API разработанного фреймворка VGL

```
frontier.set_all_active();
auto init_levels = [_levels, _source_vertex] (int src_id, int connections_count,
                                                      int vector_index)
{
    if(src_id == _source_vertex)
        _levels[_source_vertex] = FIRST_LEVEL_VERTEX;
    else
        _levels[src_id] = UNVISITED_VERTEX;
};
graph_API.compute(_graph, frontier, init_levels);
auto on_first_level = [_levels] (int src_id)->int
{
    return (_levels[src_id] == FIRST_LEVEL_VERTEX) ? NEC_IN_FRONTIER_FLAG :
                                                    NEC_NOT_IN_FRONTIER_FLAG;
};
graph_API.filter(_graph, frontier, on_first_level);

int current_level = FIRST_LEVEL_VERTEX;
while(frontier.size() > 0)
{
    auto edge_op = [_levels, _current_level](int src_id, int dst_id,
                        int local_edge_pos, long long int global_edge_pos,
                        int vector_index, DelayedWriteNEC &delayed_write)
    {
        if(_levels[dst_id] == UNVISITED_VERTEX)
            _levels[dst_id] = _current_level + 1;
    };

    graph_API.advance(_graph, frontier, edge_op);
    auto on_next_level = [_levels, current_level] (int src_id)->int
    {
        return ((_levels[src_id] == (current_level + 1))) ? NEC_IN_FRONTIER_FLAG :
                                                           NEC_NOT_IN_FRONTIER_FLAG;
    };
    graph_API.generate_new_frontier(_graph, frontier, on_next_level);
    current_level++;
}
```

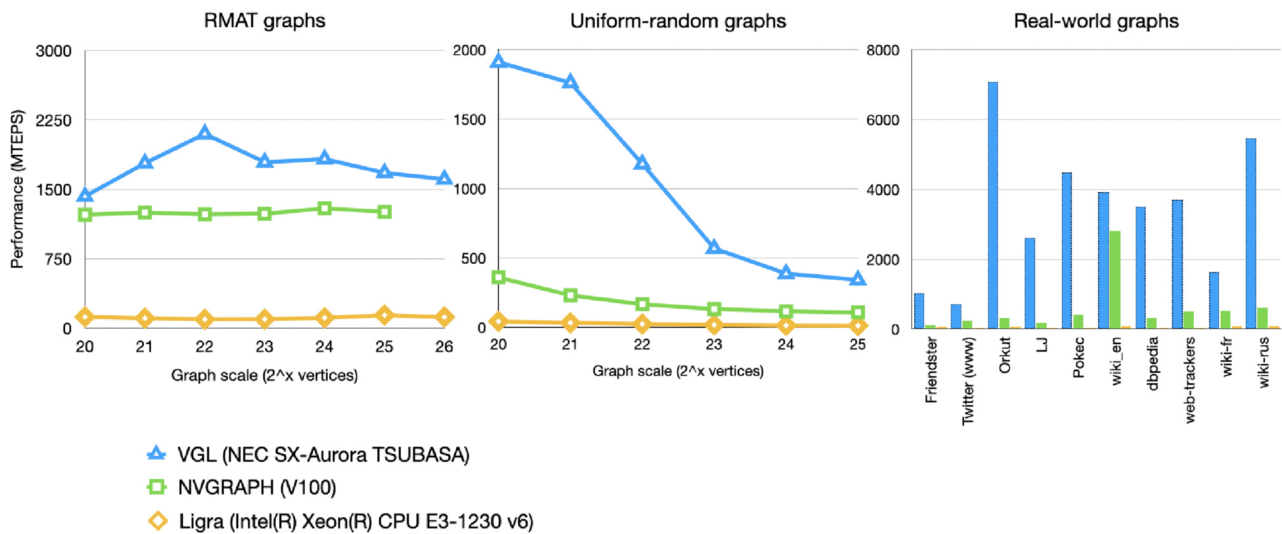

Рис. 6. Сравнение производительности реализаций алгоритма поиска кратчайших путей на основе фреймворков VGL, Ligra и библиотеки NVGraph. По горизонтальной оси отложены масштабы синтетических графов и типы реальных графов из коллекции SNAP и KONECT, по вертикальной оси — производительность в MTEPS

ную 32. Приведенное сравнение демонстрирует, что реализации на основе предложенного фреймворка VGL могут существенно опережать как существующие реализации для многоядерных центральных процессоров архитектуры Intel Skylake, так и для современных графических ускорителей NVIDIA. Кроме того, реализации на основе фреймворка VGL не уступают по производительности оптимизированным "вручную" реализациям для архитектуры NEC SX–Aurora TSUBASA, которые были предложены и подробно описаны авторами в работе [3].

8. Заключение и планы на будущее. В данной статье был представлен прототип графового фреймворка VGL, нацеленного на эффективную реализацию графовых алгоритмов для современной векторной архитектуры NEC SX–Aurora TSUBASA. Данный фреймворк является первой в мире попыткой разработки эффективного API для реализации графовых алгоритмов на векторных архитектурах. В работе были исследованы вопросы выбора функций графового API, позволяющих эффективно реализовывать векторную обработку графов, а также продемонстрированы основные схожести и отличия предложенного API от существующих аналогов для многоядерных центральных процессоров и графических ускорителей NVIDIA GPU. Также в работе были приведены примеры использования данного фреймворка для реализации фундаментальных алгоритмов поиска в ширину и поиска кратчайших путей в графе. Было продемонстрировано, что разработанные на основе фреймворка VGL реализации графовых алгоритмов практически не уступают в производительности оптимизированным "вручную" аналогам за счет инкапсуляции большого числа оптимизаций, характерных для векторных систем, а также могут существенно опережать реализации графовых алгоритмов для смежных классов архитектур — графических ускорителей и многоядерных центральных процессоров. Предложенный фреймворк позволяет значительно упростить процесс разработки графовых алгоритмов для векторных систем, на порядок сокращая объем кода реализуемых алгоритмов и скрывая от пользователя особенности программирования систем данного класса.

В планы на будущее входит разработка фреймворка, включающая в себя: оптимизацию вычислительных примитивов фреймворка, расширение списка реализованных на его основе графовых алгоритмов, комплексное сравнение производительности с существующими библиотечными аналогами для многоядерных центральных процессоров и графических ускорителей, а также разработка распределенной версии фреймворка для систем Aurora4 и Aurora8 с несколькими Vector Engine (аналог multi-GPU). Кроме того, планируется реализация данного фреймворка для других типов векторных архитектур, а также графических ускорителей.

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 19–37–90002. Работа выполнена с использованием оборудования Центра коллективного пользования сверхвысокопроизводительными вычислительными ресурсами МГУ имени М. В. Ломоносова.

СПИСОК ЛИТЕРАТУРЫ

1. *McSherry F., Isard M., Murray D.G.* Scalability! but at what COST? // Proc. 15th Workshop on Hot Topics in Operating Systems, 2015. https://www.usenix.org/conference/hotos15.
2. *Backstrom L., Boldi P., Rosa M., Ugander J., Vigna S.* Four degrees of separation // Proc. 4th Annual ACM Web Science Conference. New York: ACM Press, 33–42, 2012.
3. *Afanasyev I.V., Voevodin Vad.V., Voevodin Vl.V., Komatsu K., Kobayashi H.* Developing efficient implementations of shortest paths and page rank algorithms for NEC SX–Aurora TSUBASA architecture // Lobachevskii Journal of Mathematics. 2019. **40**, N 11. 1753–1762.
4. *Afanasyev I.V., Antonov A.S., Nikitenko D.A., Voevodin V.V., Voevodin V.V., Komatsu K., Watanabe O., Musa A., Kobayashi H.* Developing efficient implementations of Bellman–Ford and forward–backward graph algorithms for NEC SX–ACE // Supercomputing Frontiers and Innovations. 2018. **5**, N 3. 65–69.
5. *Shun J., Blelloch G.E.* Ligra: a lightweight graph processing framework for shared memory // ACM SIGPLAN Notices. 2013. **48**, N 8. 135–146.
6. *Nguyen D., Lenharth A., Pingali K.* A lightweight infrastructure for graph analytics // Proc. 24th ACM Symposium on Operating Systems Principles. New York: ACM Press, 456–471, 2013.
7. *Wang Y., Davidson A., Pan Y., Wu Y., Riffel A., Owens J.D.* Gunrock: a high-performance graph processing library on the GPU // Proc. 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM SIGPLAN Notices. 2016. **48**. doi 10.1145/3016078.2851145.
8. *Khorasani F., Vora K., Gupta R., Bhuyan L.N.* Cusha: vertex-centric graph processing on GPUs // Proc. 23rd International Symposium on High-Performance Parallel and Distributed Computing. New York: ACM Press, 239–252. 2014.
9. *Zhong J., He B.* Medusa: Simplified graph processing on GPUs // IEEE Transactions on Parallel and Distributed Systems. 2013. **25**, N 6. 1543–1552.
10. *Liu H. and Huang Howie H.* Enterprise: breadth-first graph traversal on GPUs // Proc. Int. Conf. on High Performance Computing, Networking, Storage, and Analysis. Piscataway: IEEE Press, 2015. doi 10.1145/2807591.2807594.
11. *Komatsu K., Momose S., Isobe Y., Watanabe O., Musa A., Yokokawa M., Aoyama T., Sato M., Kobayashi H.* Performance evaluation of a vector supercomputer SX–Aurora TSUBASA // Proc. Int. Conf. on High Performance Computing, Networking, Storage, and Analysis. Piscataway: IEEE Press, 2018. doi 10.1109/SC.2018.00057.
12. *Yamada Y., Momose S.* Vector engine processor of NEC's brand-new supercomputer SX-Aurora TSUBASA // Proc. Int. Symposium on High Performance Chips. https://www.hotchips.org/hc30/2conf/2.14_NEC_vector_NEC_SXAurora_TSUBASA_HotChips30_finalb.pdf.
13. *Egawa R., Komatsu K., Momose S., Isobe Y., Musa A., Takizawa H., Kobayashi H.* Potential of a modern vector supercomputer for practical applications: performance evaluation of SX–ACE. The Journal of Supercomputing. 2017. **73**, 3948–3976.
14. *Komatsu K., Egawa R., Isobe Y., Ogata R., Takizawa H., Kobayashi H.* An approach to the highest efficiency of the HPCG benchmark on the SX–ACE supercomputer // Proc. Int. Conf. on High Performance Computing Networking, Storage, and Analysis. http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/poster_files/post277s2-file3.pdf
15. *Afanasyev I.V., Voevodin Vad.V., Voevodin Vl.V., Komatsu K., Kobayashi H.* Analysis of relationship between SIMD-processing features used in NVIDIA GPUs and NEC SX-Aurora TSUBASA vector processors // Lecture Notes in Computer Science. Vol. 11657. Heidelberg: Springer, 2019. 125–139.
16. *Besta M., Podstawski M., Groner L., Solomonik E., Hoefler Y.* To push or to pull: on reducing communication and synchronization in graph computations // Proc. 26th International Symposium on High-Performance Parallel and Distributed Computing. New York: ACM Press, 2017. 93–104.
17. *Chakrabarti D., Zhan Y., Faloutsos C.* R-MAT: a recursive model for graph mining // Proc. 2004 SIAM International Conference on Data Mining. Philadelphia: SIAM Press, 2004. 442–446.
18. *Kunegis J.* KONECT: the Koblenz network collection // Proc. Int. 22nd Conf. on World Wide Web. New York: ACM Press, 2013. 1343–1350.
19. Stanford Large Network Dataset Collection. https://snap.stanford.edu/data/.
20. *Murphy R.C., Wheeler K.B., Barrett B.W., Ang,J.A.* Introducing the graph 500. http://www.richardmurphy.net/archive/cug-may2010.pdf.

Поступила в редакцию
14 апреля 2020

Developing a Prototype of High-Performance Graph-Processing Framework for NEC SX–AURORA TSUBASA Vector Architecture

I. V. Afanasyev[1,2]

[1] *Research Computing Center, Lomonosov State University; Leninskie Gory, Moscow, 119992, Russia; technician, e-mail: afanasiev_ ilya@icloud.com*

[2] *Moscow Center for Fundamental and Applied Mathematics, e-mail: afanasiev_ ilya@icloud.com*

Received April 14, 2020

Abstract: This article describes a prototype of graph-processing framework VGL (Vector Graph Library), aimed at the efficient implementation of graph algorithms for the modern NEC SX–Aurora TSUBASA vector architecture. Present day vector systems can significantly speed up various memory-intensive applications, including graph algorithms. However, approaches to the efficient implementation of graph algorithms for vector systems have been studied extremely poorly as of today: due to the highly irregular structure of real-world graphs, it is difficult to effectively use vector features of target platforms. This paper shows that the implementations of graph algorithms developed on the basis of the proposed VGL framework show the performance comparable to their manually optimized versions due to the encapsulation of a large number of graph algorithm optimizations typical for vector systems. At the same time, the proposed framework makes it possible to significantly simplify the process of developing graph algorithms for vector systems, by an order of magnitude reducing the amount of code for implemented algorithms and hiding the programming features of systems of this class from the user.

Keywords: NEC SX—Aurora TSUBASA, vector architectures, graph algorithms, graph framework, graph API, finding shortest paths in a graph, breadth-first search.

References

1. F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at What COST?," In 15th Workshop in *Proc. 15th Workshop on Hot Topics in Operating Systems, Kartause Ittingen, Switzerland, May 18–20, 2015*, https://www.usenix.org/conference/hotos15. Cited August 15, 2020.

2. L. Backstrom, P. Boldi, M. Rosa, et al., "Four Degrees of Separation," in *Proc. 4th Annual ACM Web Science Conference, Web Science 2012 (Evanston, IL, USA)* (ACM Press, New York, 2012), pp. 33–42.

3. I. V. Afanasyev, Vad. V. Voevodin, Vl. V. Voevodin, et al., "Developing Efficient Implementations of Shortest Paths and Page Rank Algorithms for NEC SX–Aurora TSUBASA Architecture," Lobachevskii J. Math. **40** (11), 1753–1762 (2019).

4. I. V. Afanasyev, A. S. Antonov, D. A. Nikitenko, et al., "Developing Efficient Implementations of Bellman–Ford and Forward–Backward Graph Algorithms for NEC SX-ACE," Supercomput. Front. Innov. **5** (3), 65–69 (2018).

5. J. Shun and G. E. Blelloch, "Ligra: a Lightweight Graph Processing Framework for Shared Memory," ACM SIGPLAN Notices **48** (8), 135–146 (2013).

6. D. Nguyen, A. Lenharth, and K. Pingali, "A Lightweight Infrastructure for Graph Analytics," in *Proc. 24th ACM Symposium on Operating Systems Principles, Farmington, USA, November 3–6, 2013* (ACM Press, New York, 2013), pp. 456–471.

7. Y. Wang, A. Davidson, Y. Pan, et al., "Gunrock: A High-Performance Graph Processing Library on the GPU," in *Proc. 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Barcelona, Spain, March 12–16, 2016* ACM SIGPLAN Notices **48** (2016). doi 10.1145/3016078.2851145

8. F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-Centric Graph Processing on GPUs," in *Proc. 23rd International Symposium on High-Performance Parallel and Distributed Computing, Vancouver, Canada, June 23–27, 2014* (ACM Press, New York, 2014), pp. 239–251.

9. J. Zhong and B. He, "Medusa: Simplified Graph Processing on GPUs," IEEE Trans. Parallel Distrib. Syst. **25** (6), 1543–1552 (2013).

10. H. Liu and H. Howie Huang, "Enterprise: Breadth-First Graph Traversal on GPUs," in *Proc. Int. Conf. on High Performance Computing, Networking, Storage, and Analysis, Austin, USA, November 15–20, 2015* (IEEE Press, Piscataway, 2015), doi 10.1145/2807591.2807594

11. K. Komatsu, S. Momose, Y. Isobe, et al., "Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA," in *Proc. Int. Conf. on High Performance Computing, Networking, Storage, and Analysis, Dallas, USA, November 11–16, 2018* (IEEE Press, Piscataway, 2018), doi 10.1109/SC.2018.00057

12. Y. Yamada and S. Momose, "Vector Engine Processor of NEC's Brand-New Supercomputer SX-Aurora TSUBASA," in *Proc. Int. Symposium on High Performance Chips, Cupertino, August 19–21, 2018*, https://www.hotchips.org/hc30/2conf/2.14_NEC_vector_NEC_SXAurora_TSUBASA_HotChips30_finalb.pdf. Cited August 15, 2020.

13. R. Egawa, K. Komatsu, S. Momose, et al., "Potential of a Modern Vector Supercomputer for Practical Applications: Performance Evaluation of SX-ACE," J. Supercomput. **73**, 3948–3976 (2017).

14. K. Komatsu, R. Egawa, Y. Isobe, et al., "An Approach to the Highest Efficiency of the HPCG Benchmark on the SX-ACE Supercomputer," in *Proc. Int. Conf. on High Performance Computing, Networking, Storage, and Analysis, Austin, USA, November 15–20, 2015*, http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/poster_files/post277s2-file3.pdf. Cited August 15, 2020.

15. I. V. Afanasyev, Vad. V. Voevodin, Vl. V. Voevodin, et al., "Analysis of Relationship Between SIMD-Processing Features Used in NVIDIA GPUs and NEC SX-Aurora TSUBASA Vector Processors," in *Lecture Notes in Computer Science* (Springer, Heidelberg, 2019), Vol. 11657, pp. 125–139.

16. M. Besta, M. Podstawski, L. Groner, et al., "To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations," in *Proc. 26th International Symposium on High-Performance Parallel and Distributed Computing, Washington DC, USA, June 26–30, 2017* New York: ACM Press, 2017. 93–104.

17. D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *Proc. 2004 SIAM International Conference on Data Mining, Orlando, USA, April 22–24, 2004* (SIAM Press, Philadelphia, 2004), pp. 442–446.

18. J. Kunegis, "KONECT: The Koblenz Network Collection," in *Proc. Int. 22nd Conf. on World Wide Web, Rio de Janeiro, Brazil, May 13–17, 2013* (ACM Press, New York, 2013), pp. 1343–1350.

19. Stanford Large Network Dataset Collection. https://snap.stanford.edu/data/. Cited August 15, 2020.

20. R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," http://www.richardmurphy.net/archive/cug-may2010.pdf. Cited August 15, 2020.