МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА

На правах рукописи

Афанасьев Илья Викторович

Исследование и разработка методов эффективной реализации графовых алгоритмов для современных векторных архитектур

Специальность 05.13.11 —
«Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
доктор физико-математических наук, профессор
Воеводин Владимир Валентинович

Москва — 2020

Оглавление

Стр.

Стр.

Стр.

Стр.

Введение

Разработка эффективных реализаций графовых алгоритмов является важной и актуальной задачей, поскольку графы исключительно 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. Во всех перечисленных областях моделируемые объекты представляются графами, состоящими из миллионов или даже миллиардов вершин и дуг, что делает оправданным использование суперкомпьютеров как для ускорения вычислений, так и для размещения в памяти графовых объектов столь большого размера.

Вместе с этим, вопрос, какие из современных суперкомпьютерных архитектур позволяют более быстро и эффективно решать графовые задачи остается актуальным. При решении графовых задач традиционно используются высокопроизводительные вычислительные системы как с общей, так и с распределенной памятью. Неоспоримым преимуществом систем с распределенной памятью является возможность обработки графов существенно больших размеров, что, однако часто достигается ценой значительного снижения производительности [1]. В то же время многие актуальные графовые задачи применимы и к графам меньшего размера. 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 многоядерных систем с общей памятью. На практике системы с общей памятью, как правило, значительно более эффективны и в расчете производительности на ядро и энергоэффективности [3]. Современные системы с общей памятью часто оборудованы сопроцессорами различных типов (например, графическими ускорителями), обеспечивающими еще более высокие значения производительности и энергоэффективности. Учитывая все эти соображения, основной акцент данной работы сделан на создании реализаций графовых алгоритмов именно для систем с общей памятью.

Большинство графовых задач относится к классу data–intensive, то есть требующих для их решения загрузки из памяти большого объема данных, и, как следствие, сильно нагружающих подсистему памяти целевых архи-

тектур. По этой причине решение графовых задач может быть значительно ускорено за счет использования систем с быстрой памятью (High Bandwidth Memory, HBM), которая имеет существенно большую пропускную способность по сравнению с DDR памятью, устанавливаемой вместе с большинством современных центральных процессоров. На сегодняшний день память стандарта HBM устанавливается преимущественно в векторные архитектуры (например, NEC SX–Aurora TSUBASA [4]) либо в многоядерные центральные процессоры с векторными расширениями (например, Fujitsu A64FX), поскольку векторная обработка данных позволяет эффективно задействовать широкую шину памяти за счет обращений к подсистеме памяти от векторных устройств. Другим важным классом архитектур, использующих память стандарта HBM, являются графические ускорители NVIDIA, которые, однако, так же используют векторную обработку данных: GPU–нити группируются в так называемые варпы, в каждый момент времени выполняющие одну и ту же инструкцию над различными данными, что является основным принципом векторных вычислений.

Подходы к реализации графовых алгоритмов для векторных систем с быстрой памятью на сегодняшний день исследованы слабо. Основная причина – это необходимость использования регулярной по своей природе векторной обработки данных для обработки нерегулярных структур данных, какими являются графы, а также значительная новизна векторных систем, использующих стандарт HBM памяти (большинство рассматриваемых в данной работе архитектур выпущены после 2018 года). Вследствие этого, для векторных систем на сегодняшней день не предложено подходов к эффективной реализации графовых алгоритмов, а также не существует специализированных библиотек для решения графовых задач. Существующие библиотечные реализации для многоядерных центральных процессоров на векторных системах демонстрируют крайне низкую производительность. Если рассматривать класс графических ускорителей, то реальная производительность существующих реализаций на графовых задачах очень и очень низка. Все эти аргументы в совокупности говорят о том, что разработка подходов к созданию эффективных реализаций графовых алгоритмов для современных векторных систем с быстрой памятью, являющаяся предметом данного диссертационного исследования, крайне актуальна.

Целью данной работы является разработка методов, позволяющих создавать эффективные реализации графовых алгоритмов для современных

векторных архитектур с быстрой памятью. Под словами «эффективные реализации» будем понимать такие реализации, которые позволяют решать графовые задачи в разы быстрее современных аналогов.

Объектом исследования диссертационной работы является методика организации эффективного решения графовых задач. Под эффективным решением понимается как выбор архитектур вычислительных систем, позволяющих получить значительную реальную производительность при решении графовых задач, так и создание высокоэффективных реализаций для выбранных архитектур.

Предметом исследования диссертационной работы является процесс создания эффективных реализаций графовых задач для современных векторных архитектур с быстрой памятью. В работе рассмотрены 9 часто возникающих на практике графовых задач и 17 наиболее распространенных алгоритмов их решения.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. исследовать существующие подходы к реализации графовых алгоритмов на различных современных вычислительных архитектурах;
2. исследовать взаимосвязь свойств, характеристик, а также принципов разработки и оптимизации программ для различных классов современных архитектур с быстрой памятью – векторных процессоров и графических ускорителей NVIDIA GPU;
3. разработать и обосновать подход к созданию эффективных реализаций графовых алгоритмов для векторных систем с быстрой памятью, определяющий вопросы выбора алгоритмов, структур данных и целевых оптимизаций;
4. разработать и апробировать эффективные и высокопроизводительные реализации набора фундаментальных графовых алгоритмов для различных векторных архитектур с быстрой памятью;
5. провести исследование производительности, эффективности и энергоэффективности разработанных реализаций, а также провести сравнительный анализ производительности с существующими аналогами графовых библиотек и фреймворков для многоядерных центральных процессоров и графических ускорителей NVIDIA.

Научная новизна:

1. Впервые была детально изучена и описана взаимосвязь современных векторных архитектур и графических ускорителей NVIDIA: их аппаратных свойств и характеристик, а также подходов к разработке и оптимизации графовых программ для обоих классов архитектур;
2. Был проведен анализ свойств и характеристик графовых алгоритмов на предмет их соответствия современным векторным архитектурам с быстрой памятью;
3. Для широкого класса графовых алгоритмов был выделен компактный набор типовых алгоритмических структур, которые возможно эффективно реализовать на векторных архитектурах с быстрой памятью; были детально описаны подходы к реализации и оптимизации данных типовых алгоритмических структур для векторных систем с быстрой памятью;
4. Был предложен метод, позволяющий создавать эффективные реализации графовых алгоритмов для современных векторных архитектур с быстрой памятью;
5. На основе предложенного метода впервые были созданы эффективные реализации фундаментальных графовых алгоритмов для современных векторных систем с быстрой памятью, имеющие существенно более высокую производительность и эффективность по сравнению с существующими библиотечными аналогами для многоядерных центральных процессоров и графических ускорителей NVIDIA;
6. Был реализован первый в мире архитектурно–независимый графовый фреймворк, позволяющий создавать эффективные реализации широкого класса графовых алгоритмов для современных векторных систем с быстрой памятью.

Практическая значимость диссертационной работы состоит в разработке и реализации программного комплекса для решения графовых задач на современных векторных системах с быстрой памятью. Разработанный программный комплекс обладает тремя чрезвычайно важными характеристиками:

– эффективность: реализации графовых алгоритмов на основе предложенного программного комплекса обладают значительно более высокой производительностью по сравнению с существующим аналогами графовых библиотек;

– продуктивность: процесс разработки новых реализаций графовых алгоритмов заметно упрощен за счет предоставления программной среды высокого уровня, позволяющей решать за пользователя многие сложные вопросы, возникающие в процессе реализации графовых алгоритмов;
– переносимость: за счет выбора типовых алгоритмических структур, получаемые реализации можно легко переносить между рассмотренными в работе вычислительными архитектурами с быстрой памятью с сохранением высокой производительности.

Методология и методы исследования. При получении основных результатов диссертационной работы использовались элементы теории графов, методы параллельного программирования, методы анализа информационной структуры параллельных алгоритмов, а также формальные модели оценки эффективности параллельных программ и степени локальности данных. При разработке реализаций графовых алгоритмов и создании программного комплекса использовались методы объектно–ориентированного анализа и проектирования, а также программно–аппаратная архитектура параллельных вычислений CUDA, открытый стандарт для распараллеливания программ OpenMP, специализированные компиляторы с поддержкой векторизации, а также средства анализа эффективности и производительности – CUDAToolkit, Intel Parallel Studio и инструменты компании NEC.

Основные положения, выносимые на защиту.

1. Метод создания реализаций множества распространенных графовых алгоритмов, использующий набор из четырех алгоритмических абстракций и двух абстракций данных. Показано, что с помощью этого набора можно выразить все рассмотренные графовые алгоритмы, и обоснована возможность их эффективной реализации на современных векторных архитектурах.
2. Принципы проектирования и реализации архитектурно-независимого программного комплекса VGL (графовый фреймворк) для векторных систем с быстрой памятью, позволяющего объединить сразу три важные характеристики в разработке программного обеспечения: эффективность, продуктивность и переносимость.
3. Эффективные реализации набора фундаментальных графовых алгоритмов для современных векторных архитектур с быстрой памятью

– NEC SX–Aurora TSUBASA, NVIDIA GPU, Intel KNL, созданные на основе разработанного фреймворка и демонстрирующие существенно большую производительность (как правило, в разы) и энергоэффективность по сравнению с существующими библиотечными аналогами для многоядерных центральных процессоров и графических ускорителей NVIDIA.

Степень достоверности и апробация результатов. Представленные в работе результаты докладывались на следующих научных международных и российских конференциях и семинарах::

1. «15th International Conference on Parallel Computing Technologies (PaCT–2019)», Алматы, Казахстан, 19–23 августа 2019;
2. «10th JHPCN Symposium», Токио, Япония, 12–13 июля 2018;
3. «SC17: International Conference for High Performance Computing, Networking, Storage and Analysis», Denver, CO, США, 12–17 ноября 2017;
4. «Суперкомпьютерные дни в России», Москва, Россия, 23–24 сентября 2019;
5. «Параллельные вычислительные технологии (ПаВТ)», Москва, Россия, 27–29 мая 2020;
6. «Время, хаос и математические проблемы», Москва, Россия, 19 августа 2020;
7. «Ural-PDC 2018», Екатеринбург, Россия, 15 ноября 2018;
8. «Суперкомпьютерные дни в России», Москва, Россия, 24–25 сентября 2018;
9. «3rd Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists (Ural-PDC 2017)», Екатеринбург, Россия, 19 октября 2017;
10. «Суперкомпьютерные дни в России», Москва, Россия, 25–26 сентября 2017;
11. «Суперкомпьютерные дни в России», Москва, Россия, 26–27 сентября 2016;
12. Семинар ЛИТ ОИЯИ, 16 октября 2020;
13. Научный семинар кафедры ИИТ ВМК;
14. Научные семинары кафедры СКИ ВМК и НИВЦ МГУ.

Личный вклад. Личный вклад автора заключается в выполнении всего объема теоретических и экспериментальных исследований, а также в разработке архитектуры и реализации программного комплекса, предназначенного для эффективного решения графовых задач на современных векторных архитектурах с быстрой памятью. Полученные результаты диссертационной работы были оформлены автором в виде научных публикаций, а также представлены на научных конференциях. В совместных публикациях [5–11] автором диссертации производился всесторонний анализ графовых алгоритмов и их модификация для векторных систем с быстрой памятью, реализация и оптимизация алгоритмов на данных системах, а также сравнительный анализ производительности и эффективности с существующими аналогами. В публикации [12] автором был сформулирован и описан метод анализа эффективности суперкомпьютерных приложений для графических ускорителей NVIDIA, основанный на анализе динамических характеристик, который был применен к задаче расчета структуры жидкокристаллических капель. Данный метод (и его модификации) в рамках данного диссертационного исследования применялся для анализа эффективности разработанных реализаций графовых алгоритмов для различных целевых архитектур.

Публикации.

Основные положения и выводы диссертационного исследования в полной мере изложены в 11 научных работах [5–15], опубликованными в том числе в 9 публикациях в рецензируемых научных изданиях [5–13], определенных п. 2.3 Положения о присуждении ученых степеней в Московском государственном университете имени М.В.Ломоносова.

Объем и структура работы. Диссертация состоит из введения, пяти глав и заключения. Полный объём диссертации составляет 135 страниц, включая 30 рисунков и 13 таблиц. Список литературы содержит 80 наименований.

Глава 1. Обзор подходов к реализации графовых алгоритмов для векторных систем с быстрой памятью

1.1 Три основных свойства графовых алгоритмов

Графовые алгоритмы характеризуются тремя принципиально важными свойствами, отличающими их от многих других классов алгоритмов. Первым из данных свойств является принадлежность графовых алгоритмов к классу data intensive (или memory-bound) – производящих значительно большее количество обменов с оперативной памятью в сравнении с объемом производимых вычислений. Данное свойство обусловлено спецификой графовых задач, в которых простейшие вычислительные операции (сравнения, сложения, присваивания) над вершинами и ребрами графов сочетаются с постоянной загрузкой больших объемов данных о вершинах и ребрах обрабатываемого графа. При реализации графовых алгоритмов на современных вычислительных системах данное свойство приводит к простою вычислительных устройств процессора, которые большую часть времени ожидают получения необходимых для вычислений данных из подсистемы памяти.

Вторым свойством графовых алгоритмов является их нерегулярность. При этом нерегулярность в графовых алгоритмах присутствует в двух видах. Первый вид нерегулярности обусловлен степенным (или близким к степенному) характером распределения степеней вершин для многих графов реального мира, в которых присутствуют вершины как с очень высокой, так и с очень низкой степенями. При реализации графовых алгоритмов данная нерегулярность требует использования принципиально различных подходов к параллельной обработки для вершин с различной степенью, и значительно затрудняет использование векторной обработки данных, требующей работы с векторами постоянной длины. Второй вид нерегулярности графовых алгоритмов обусловлен наличием большого числа косвенных обращений к памяти, выполняемых при загрузке информации о вершинах графа. Например, при обработке вершин графа, смежных к определенной заданной, из памяти загружается информация о вершинах, которая потенциально может находиться в очень удаленных друг от друга участках памяти, так как, вообще говоря, каждая из вершин в гра-

фе может быть соединена с любой другой. Из-за данного вида нерегулярности реализации графовых алгоритмов по свойствам близки к задаче-бенчмарку случайных обращений к памяти (random memory access). Близкий к случайному характер обращений к данным о вершинах графа не позволяет эффективно использовать кэш-память при работе с графами больших размеров: массивы с информацией о вершинах не могут быть целиком размещены в кэш-памяти в следствии большого объема входных графов, а механизм загрузки данных кэш-линиями не позволяет эффективно подгрузить используемые при следующих обращениях данные из-за близкого к случайному характера обращений.

Третьим свойством графовых алгоритмов является заложенный в них значительный ресурс параллелизма. Во многих графовых алгоритмах возможна параллельная обработка вершин и ребер входных графов, количество которых для многих графов реального мира исчисляется миллионами или миллиардами. В следствии данного свойства для решения графовых задач могут использоваться различные современные массивно-параллельные вычислительные архитектуры.

1.2 Взаимосвязь свойств современных вычислительных архитектур и графовых алгоритмов

Вопрос, какие из современных вычислительных архитектур выгоднее использовать для быстрого и эффективного решения графовых задач крайне актуален. Так как графовые алгоритмы относятся к классу data-intensive, то есть производящих большое количество обменов с оперативной памятью при малом числе арифметических операций, для решения графовых задач перспективно использовать системы с быстрой памятью (стандарта HBM). В Табл. 1 приведены модели и основные характеристики наиболее часто используемых вычислительных архитектур, использующих быструю память.

Анализ аппаратных особенностей перечисленных в Табл. 1 архитектур показывает, что все они имеют три важных общих свойства:

– присутствие различного рода **элементов векторной обработки данных**: векторных расширений инструкций, GPU-варпов, либо векторно-конвейерных инструкций;

Таблица 1 — Современные архитектуры с быстрой памятью и их основные характеристики

Архитектура	Векторные возможности	Тип памяти	Пропускная способность памяти	Объем памяти	Число ядер	Число уровней иерархии кэш-памяти и размер кэша последнего уровня
NEC SX-Aurora TSUBASA	Векторно-конвейрные векторные инструкции длины 256	HBM2	1200 GB/s	до 48 GB	8	1 (16 MB)
NVIDIA P100 GPU	warps (32 нити)	HBM2	720 GB/s	до 16 GB	3840	2 (2 MB)
NVIDIA V100 GPU	warps (32 нити)	HBM2	900 GB/s	до 32 GB	5120	2 (6 MB)
Intel KNL	AVX-512 (512-bit)	MCDRAM	400 GB/s	до 16 GB	64–72	2 (32 MB)
A64FX (узел Fugaku)	SVE (512-bit)	HBM2	1024 GB/s	до 32	48+	2 (8 MB)

– **высокую степень ресурса внутреннего параллелизма**, который достигается либо за счет использования значительного числа вычислительных ядер (NVIDIA GPU), либо векторов большой длины (NEC SX-Aurora TSUBASA), либо сочетания двух данных факторов (A64FX, Intel KNL).
– **использование быстрой памяти** стандарта HBM в сочетании с небольшим числом уровней иерархии кэш-памяти малого размера, что влечет за собой необходимость локальной работы с данными.

Три данных свойства рассматриваемых архитектур сильно связаны между собой. Так, например, наличие векторных подходов к обработке данных позволяет эффективно задействовать высокую пропускную способность шины памяти за счет использования векторных обращений к подсистеме памяти. Аналогично, без использования векторных вычислений на данных архитектурах достичь эффективного использования пропускной способности невозможно. Далее в работе будет использоваться термин **«векторные системы с быстрой памятью»**, или же «целевые архитектуры» для обозначения класса современных вычислительных архитектур, характеризующиеся тремя данными свойствами.

Далее будет исследована взаимосвязь трех данных свойств архитектур с выделенными ранее тремя основными свойствами графовых алгоритмов. Во-первых, многие графовые алгоритмы обладают значительным ресурсом па-

раллелизма, что позволяет эффективно задействовать массивный параллелизм целевых архитектур. Во-вторых, наличие в данных архитектурах быстрой памяти позволяет значительно ускорить графовые алгоритмы, принадлежащие к классу data-intensive. В-третьих, для эффективной реализации графовых алгоритмов на системах данного класса необходимо задействовать векторную обработку данных, что, однако, для графовых задач далеко не тривиально в силу присущих им свойств нерегулярности. Однако, как будет показано далее в работе, многие графовые алгоритмы подлежат эффективной векторизации, что позволяет значительно ускорить решение многих графовых задач, используя выделенный класс векторных систем с быстрой памятью.

На основе проведенного анализа свойств архитектур и графовых алгоритмов можно выделить три основные требования, которым должны удовлетворять эффективные реализации графовых алгоритмов для векторных архитектур с быстрой памятью.

- Реализации должны задействовать значительный ресурс внутреннего параллелизма целевых архитектур,
- Реализации должны использовать принципы векторной обработки данных.
- Реализации должны использовать локальную работу с данными, с целью эффективного использования высокой теоретической пропускной способности памяти целевых архитектур.

Три данных требования будут учитываться далее на протяжении всей работы в процессе создания реализаций различных графовых алгоритмов.

1.3 Исследуемые в работе графовые задачи

В данной работе была исследована возможность эффективного решения на векторных системах с быстрой памятью памятью следующих фундаментальных графовых задач:

- поиска в ширину (Breadth First Search – BFS),
- поиска всех пар кратчайших путей в графе от заданной вершины (Single Source Shortest Paths – SSSP),
- ранжирования вершин в графе (Page Rank – PR),

- поиска связных компонент в графе (Connected Components – CC),
- поиска сильно связных компонент в графе (Strongly Connected Components – CC),
- вычисление степени посредничества (Betweenness centrality – BC),
- поиска сообществ в графе (Community Detection – CD),
- поиска минимального и максимального потока в графе (Max Flow – MF),
- поиска минимального остовного дерева (Minimum Spanning Tree – MST).

Данные задачи широко используются во многих прикладных областях. На сегодняшний день существует несколько графовых фреймворков и библиотечных реализаций, позволяющих производить решение данных задач на различных вычислительных системах с общей и распределенной памятью. Однако, как будет продемонстрировано в последующих подразделах, ни одна из существующих реализаций не позволяет в достаточной мере эффективно решать данные задачи на выделенном ранее классе векторных систем с быстрой памятью.

1.4 Реализации графовых алгоритмов для современных архитектур: multicore CPU, NVIDIA GPU, векторные процессоры

В данном подразделе приведен обзор существующих параллельных реализаций графовых алгоритмов для современных вычислительных архитектур с общей памятью. Существующие реализации будут классифицированы по признаку их принадлежности к определенному классу целевых архитектур:

- многоядерных центральных процессорах (multicore CPU) различных производителей – Intel, IBM;
- графических ускорителях NVIDIA GPU;
- векторной архитектуре NEC SX-Aurora TSUBASA, а также на архитектурах многоядерных центральных процессоров с векторными расширениями инструкций (Intel Skylake и AVX-512, Intel KNL и AVX-512).

При этом каждая из приведенных реализаций так же может иметь один из следующих типов, который будет указан для большинства упомянутых далее реализаций.

– Реализации графовых алгоритмов, оформленные в виде библиотек обработки графов. Наиболее известным примером графовой библиотеки является Boost Graph Library [16], в которой представлены последовательные и параллельные реализации многих графовых задач.
– Реализации графовых алгоритмов, реализованные на основе фреймворков. Графовые фреймворки представляют собой набор высокопроизводительных реализаций некоторого набора абстракций вычислений и данных, позволяющих реализовывать широкий набор графовых алгоритмов. Обычно, в состав подобных фреймворков входят примеры реализаций некоторых графовых алгоритмов.
– Одиночные реализации конкретных графовых алгоритмов, описанные в статьях или же представленные в открытом доступе. Так как данная группа реализаций крайне обширна, будут приведены лишь работы с наиболее важными оптимизациями наиболее известных алгоритмов.

В начале будут рассмотрены реализации для многоядерных центральных процессоров. На сегодняшний день к наиболее высокопроизводительным графовым библиотекам и фрейморкам для систем данного класса относятся Galois, Ligra и GAP Benchmark Suite(GAPBS).

Galois [17] – это система, позволяющая эффективно реализовывать программы (в том числе и на GPU) с нерегулярным параллелизмом по данным без необходимости написания явно параллельного кода. Систему Galois можно рассматривать как графовый фреймворк, так как на её основе можно производить эффективную реализацию многих графовых алгоритмов. Galois включает в себя примеры реализации графовых алгоритмов (BFS, SSSP, PR и многих других). Алгоритмы, реализованные в Galois, могут свободно использовать автономное планирование (без барьеров синхронизации), что позволяет существенно уменьшить расходы на синхронизации. Кроме того, внутренний параллельный планировщик Galois аккуратно учитывает топологию ядер и сокетов целевой платформы.

Ligra [3] – это фреймворк, реализующий графовые алгоритмы на основе абстракций работы с подмножествами вершин и ребер графов. Данный фреймворк использует параллельную среду выполнения Cilk, однако в последних его

версиях так же присутствует и OpenMP-реализация. При работе с абстракциями подмножеств вершин и ребер графа Ligra использует ряд эвристик с целью определить более эффективное направление обхода графа (push или pull), а также какие структуры данных использовать (разреженные или плотные). Совокупность данных оптимизаций делает фреймворк Ligra крайне эффективным для обработки графов с низким диаметром на современных центральных процессорах.

GAPBS (GAP Benchmark Suite) [18] – высокопроизводительная библиотека графовых алгоритмов, использующая стандарт OpenMP для параллельной обработки графов. В данной библиотеке приведены сильно оптимизированные реализации многих часто используемых алгоритмов решения графовых задач BFS, PR, SSSP, CC, и другие.

Все три рассмотренные системы включают в себя реализации большинства графовых задач, исследуемых в данной работе. В дополнение к перечисленным системам существуют реализации графовых алгоритмов на основе стандарта GraphBLAS [19]. Идея GraphBLAS основана на представлении графов в виде разреженных матриц, откуда данный стандарт и берет свое название. В случае, если граф представлен в виде разреженной матрицы, разреженное умножение матрицы на вектор является шагом алгоритма поиска в ширину. Обобщая скалярные операции линейной алгебры, возможна реализация широкого набора других графовых задач и алгоритмов, в том числе Page Rank, поиска кратчайших путей, а также подсчета числа треугольников в графе. Существует несколько реализаций стандарта GraphBLAS для различных архитектур, в том числе SuiteSparse [20] для многоядерных процессоров на основе OpenMP. Подобные реализации можно рассматривать как графовые фреймворки, позволяющие реализовывать графовые алгоритмы с использованием функций стандарта GraphBLAS.

Кроме того, важно упомянуть и ряд других менее известных графовых библиотек, которые, однако, включают в себя принципиально важные оптимизации графовых алгоритмов.

Cagra [21] – библиотека обработки графов, направленная на оптимизацию использования кэш-памяти с использованием сегментированного CSR формата хранения графов, разбивающего вершины графа на множества, размещение которых возможно в определенном уровне иерархии памяти целевой архитектуры (обычно последнем, Last Level Cache).

Polymer [22] – графовый фреймворк, оптимизированный для работы на NUMA системах, демонстрирующий значительное ускорение по сравнению со фреймворками Galois и Ligra на NUMA системах с большими объемами памяти.

Существует ряд работ, посвященных сравнительному анализу производительности рассмотренных графовых библиотек и фреймворков. Так, работа [23] демонстрирует, что системы Ligra, Galois и GAPBS конкурентоспособны между собой: каждая из них демонстрирует более высокую или низкую производительность в зависимости от типа входного графа, а также типа реализуемого алгоритма. Фреймворки на основе GraphBLAS обладают меньшей производительностью для большинства алгоритмов и входных данных, так как не включают в себя оптимизации, основанные на свойствах входных графов (повышение локальности обходов вершин и ребер, сокращение количества просматриваемых элементов графа, оптимизацию направления обхода графа, и многие другие).

Второй важной группой современных архитектур, для которых существуют библиотечные реализации графовых алгоритмов, являются графические ускорители NVIDIA GPU. Первые попытки реализации графовых алгоритмов для архитектуры NVIDIA GPU предложены в работе [24], где впервые продемонстрирован потенциал использования графических ускорителей для решения графовых задач на примере реализации алгоритмов поиска в ширину и кратчайших путей. На сегодняшний день архитектура NVIDIA GPU считается крайне перспективной для решения графовых задач в силу наличия в ней массивного параллелизма и быстрой памяти [25; 26]. За последние годы для NVIDIA GPU были разработаны различные библиотеки и фреймворки, позволяющие решать многие графовые задачи. Наиболее известными системами являются Gunrock, cuSHA, Enterprise, Medusa, TOTEM, GraphCage, SIMD-X. Все данные системы являются фреймворками, скрывающими от пользователей непростые особенности программирования графических ускорителей. Подходы к оптимизации, используемые в данных фреймворках, будут подробно рассмотрены в следующем подразделе.

Gunrock [27] – один из наиболее современных, известных и высокопроизводительных графовых фреймворков на сегодняшний день. Программный интерфейс пользователя данного фреймворка основан на использовании абстракций, построенных не вокруг вычислений, а вокруг данных. В последних версиях данного фреймворка реализована поддержка multi-GPU вычислений.

Важно отметить, что данный фреймворк не использует предобработку графов. В Gunrock так же включен широкий набор примеров реализаций графовых алгоритмов, включающий в себя алгоритмы решения всех рассматриваемых в данной работе задач.

Medusa [28] – это графовый фреймворк, основанный на предложенной его авторами модели «Edge-Message-Vertex», являющейся расширением BSP(Bulk Synchronous Parallel) модели, реализованной на основе пересылки сообщений между вершинами графа внутри одного GPU. Так же Medusa поддерживает распределенную обработку графов на нескольких GPU в пределах одной вычислительной системы. В качестве примеров реализации в Medusa приведены лишь несколько алгоритмов – поиска в ширину и кратчайших путей в графе.

cuSHA [29] – графовый фреймворк, нацеленный на повышение эффективности использования иерархии памяти GPU при решении графовых задач. В отличие от большинства других фреймворков, использующих для хранения графов CSR формат, cuSHA использует форматы «G-Shards» и «Concatenated Windows (CW)», в которых графы представлены в виде независимых упорядоченных подмножеств ребер, называемых «shard». В качестве примеров реализаций графовых задач в cuSHA приведены алгоритмы решения задач BFS, SSSP и PR. Так же cuSHA может работать на multi-GPU системах.

Enterprise [30] – высокопроизводительная реализация алгоритма поиска в ширину. Несмотря на то, что данная реализация ориентируется на решение одной конкретной графовой задачи, в ней предложены многие важные подходы к оптимизации, применимые и к другим графовым алгоритмам. Enterprise также имеет поддержку использования multi-GPU систем.

Totem [31] – графовый фреймворк, ориентированный на работу на гибридных системах, состоящих из многоядерных центральных процессоров и графических ускорителях. В данном фреймворке предложены подходы к разбиению графа для его последующей обработки в рамках гибридных систем с различными конфигурациями.

GraphCage [32] – библиотека графовых алгоритмов, направленная на оптимизацию использования кэш-памяти графических ускорителей на основании методов, аналогичных применяемым в Cagra для многоядерных центральных процессоров. Данная библиотека нацелена на решение проблем, обусловленных сравнительно небольшим размеров кэш-памяти, устанавливаемой в современ-

ных графических ускорителях NVIDIA, а также коллективным характером работы с кэш-памятью большого числа нитей.

SIMD-X [33] – графовый фреймворк, разработанный на основе модели вычислений Active-Compute-Combine (ACC), позволяющей внедрить ряд важных оптимизаций, таких как эффективное распределение работы между вычислительными CUDA-ядрами при обработке разреженных списков вершин.

Реализации некоторых графовых алгоритмов для графических ускорителей NVIDIA присутствуют также и в описанной ранее системе Galois [17].

Подробный сравнительный анализ производительности рассмотренных фреймворков для графических ускорителей NVIDIA приведен в работе [25], где продемонстрировано, что не существует фреймворка, имеющего наилучшую производительность для любых рассмотренных задачах, алгоритмов и входных графов.

К третьей группе архитектур относятся реализации для современных векторных систем, а также систем с векторными расширениями инструкций. Подходы к реализаций и оптимизации графовых алгоритмов для систем данных классов практически не исследованы. Так, не существует графовых библиотек и фреймворков, нацеленных на работу с современными векторными системами, а все рассмотренные реализации для многоядерных центральных процессоров не используют векторные расширения инструкций. С одной стороны это оправдано, так как традиционные центральные процессоры не оборудованы специализированной быстрой памятью, за счет использования которой возможно было бы получить значительное ускорение для программ, интенсивно использующих подсистему памяти при использовании векторных расширений. С другой стороны, ранее уже были приведены примеры современных векторных систем с быстрой памятью (NEC SX-Aurora TSUBASA, Intel KNL, A64FX), которые потенциально могут ускорить решение многих графовых задач. Однако, существующие реализации для многоядерных центральных процессоров демонстрируют крайне низкую производительность на данных системах. Существующие немногочисленные подходы к реализации графовых алгоритмов для векторных систем описаны далее.

В работе [34] предложен специализированный формат хранения графа SlimCell, позволяющий эффективно реализовать алгоритм поиска в ширину для архитектур Intel Xeon Phi KNL и Intel Haswell с поддержкой векторных инструкций AVX-512. Реализация поиска в ширину в данной работе основана на

разреженном матрично-векторном умножении, при чем схожий подход может использоваться и для других векторных архитектур, так как для них существует большое число оптимизированных алгоритмов работы с разреженными матрицами (в том числе для NEC SX-Aurora TSUBASA [35]). Однако известных реализации графовых алгоритмов, основанных на традиционных форматах представления (например CSR) на момент написания данной работы предложено не было.

Таким образом, на сегодняшний день не существует библиотечных реализаций, позволяющих эффективно решать графовые задачи для векторных систем с быстрой памятью. Предложенных подходов к оптимизации и реализации графовых алгоритмов для систем данных классов так же крайне мало, что в очередной раз подчеркивает актуальность данной работы. Для графических ускорителей NVIDIA реальная производительность существующих реализаций на графовых задачах крайне низка, поэтому не исключено, что они обладают существенном потенциалом для дальнейшей оптимизации, что и будет продемонстрировано в ходе данного исследования. Кроме того, существующие графовые фреймворки и библиотеки не обеспечивают переносимость с сохранением высокой эффективности реализаций между различными классами архитектур с быстрой памятью – векторными системами и графическими ускорителями, что так же является их недостатком.

1.5 Существующие подходы к оптимизации графовых алгоритмов

В данном разделе рассмотрены основные существующие подходы к оптимизации графовых алгоритмов, применяемые для различных современных вычислительных систем. Так как для различных классов архитектур зачастую могут применяться схожие подходы к оптимизации, каждое из направлений оптимизации будет по возможности рассмотрено без привязки к определенной архитектуре.

– **Балансировка параллельной нагрузки при обработке вершин с различной степенью.** Необходимость в данном направлении оптимизации возникает при обработке графов со степенным характером распределения вершин, когда в графе присутствуют как вершины с

очень большой, так и вершины с очень маленькой степенью. В случае, если обработка списков ребер различных вершин графа осуществляется параллельно различными вычислительными устройствами (например отдельными ядрами CPU или GPU), могут возникать существенные задержки из-за простоя нитей, выделенных для обработки вершин с небольшой степенью, в то время как нити, выделенные для обработки вершин с большой степенью еще не завершили свою работу. Для центральных процессоров с относительно небольшим числом ядер (1-20), данная проблема не столь значительна и может решаться аккуратным распределением параллельной работы между нитями с использованием, к примеру, средств OpenMP (static, guided, dynamic). В то же время для процессоров с большим числом легковесных ядер, например Intel KNL или NVIDIA GPU, насчитывающим до 80 и до 5000 ядер соответственно, требуется реализация специализированных подходов к балансировке нагрузки. Для векторных архитектур данная проблема еще более усложняется, так как дополнительно необходимо эффективно задействовать параллелизм внутри векторных инструкций. Основные варианты решения проблемы балансировки – разбиение вершин графа на группы с примерно одинаковой степень: либо динамическое [33], либо на основе предобработки графа [36].

– **Эффективное использование векторных возможностей архитектур при обработке вершин с различной степенью.** Данное направление оптимизации является обобщением предыдущего для архитектур, использующих принципы SIMD обработки данных (GPU варпы, векторные инструкции), так как данные архитектуры могут обрабатывать сразу несколько вершин или ребер графа с использованием одной векторной инструкции. При этом важный вопрос заключается в том, как лучше распределить работу между элементами векторных инструкции: в случае, если с использованием векторных инструкций обрабатывать смежные ребра отдельно взятых вершин, то для вершин с низкой степенью(меньшей, чем длина вектора) будет простаивать значительная часть векторных элементов. Данной проблеме для GPU посвящено сразу несколько работ. В работе [37] описан подход, основанный на разбиении вершин на группы по степени, с последующей обработкой каждой из групп различным образом: нитью на верши-

ну, варпа на вершину, или блока из нитей на вершину. В работе [38] предложена концепция виртуального варпа, при которой один варп обрабатывает несколько вершин с одинаковой степенью в определенном интервале, что позволяет организовать обработку вершин с невысокой степенью достаточно эффективным образом. Другой подход, основанный на разбиении ребер графа по независимым группам примерно одинакового размера с последующей их обработкой, был предложен в работе [39].

– **Оптимизация направления обхода графа.** В большинстве графовых алгоритмов для каждой обрабатываемой вершины может либо собираться информация о состоянии соседних с ней вершинах (pull), либо информация о состоянии обрабатываемой вершины может рассылаться всем соседним вершинам (push). Оба подхода принципиально отличаются друг от друга за счет: необходимости в использования атомарных операций, наличия различного числа и типов конфликтов на чтение и запись, различия в итоговой сложности и скорости сходимости алгоритмов. Как следствие, pull- и push-подходы могут быть более или менее эффективны как для различных архитектур, так и для различных графовых алгоритмов. Комплексное сравнение pull- и push-подходов представлено в работе [40], где произведен сравнительный анализ данных подходов для 11 графовых алгоритмов и реализаций, ориентированных на работу в системах с общей и распределенной памятью.
– **Эффективное использование кэш-памяти для хранения данных о вершинах, запрашиваемых при косвенных адресациях.** Косвенные обращения к памяти, неизбежно возникающие при запросах о состоянии вершин графа, являются одним из главных факторов, снижающих производительность графовых алгоритмов на современных вычислительных архитектурах. В случае, если подобные обращения не могут быть обработаны кэш-памятью целевой архитектуры, производительность реализуемого графового алгоритма существенно снижается. Большинство призванных решить данную проблему оптимизаций нацелены на более эффективное хранение данных о вершинах в различных уровнях иерархии кэш-памяти целевой архитектуры. Примерами подобных оптимизаций являются кластеризация [41] и сегментирование [21;

41]. Основной идей кластеризации является хранение в смежных участках памяти информации о наиболее часто запрашиваемых вершинах графа (обычно с самой высокой степенью), после чего участки массивов с информацией о данных вершинах подгружаются в кэш-память [42]. Расширением идеи кластеризации для архитектуры NVIDIA GPU является механизм работы с hub-вершинами (наиболее часто запрашиваемыми) во фреймворке Enterprise [30], когда hub- вершины хранятся на регистрах и в разделяемой памяти. Основной идеей сегментирования является разбиение графа на относительно независимые подграфы, обработка каждого из которых возможна с сохранением данных о всех вершинах данного подграфа в определенном уровне иерархии кэш-памяти. В работе [43] рассмотрено расширение подхода сегментирования для различных направлений обхода графа графа (push и pull). Идеи сегментирования используются так же и для реализации обработки графов, размер которых существенно превышает доступный объем оперативной памяти системы [44]. Кроме того, во фреймворке Cagra [21] продемонстрирована возможность одновременного использования подходов CSR-сегментирования и кластеризации.

– **Улучшение шаблона доступа к памяти при загрузке информации о ребрах графа.** Многие архитектуры позволяют загружать данные из памяти значительно более эффективно в случае, когда вычислительные устройства обращаются к подсистеме памяти согласно определенному шаблону [45], тем самым существенно повышая используемую пропускную способность памяти. Обычно, данные шаблоны согласуются с понятием пространственной локальности [46], когда данные о подряд идущих ребрах или вершинах графа загружаются связанными вычислительными единицами – нитями одного и того же варпа графического ускорителя или же элементами одной и той же векторной инструкции.
– **Обработка вершин с различным приоритетом.** Производительность многих графовых алгоритмов может быть значительно увеличена при изменении порядка обхода вершин. Оптимизация данного типа применяется, к примеру, в фреймворке Gunrock [27], и обычно реализуется динамическим разбиением обрабатываемых вершин на группы с различным приоритетом обработки.

– **Использование различных форматов хранения обрабатываемых подмножеств вершин.** Многие итеративные графовые алгоритмы не требуют обработки (обхода) всех вершин и ребер графа на каждой из итераций; далее в данной работе данные алгоритмы будут именоваться «partial active» [32], в то время как алгоритмы, выполняющие обход всех вершин и ребер на каждой итерации – «all active» [32]. Для реализации «partial-active» графовых алгоритмов требуется генерация и хранение множеств вершин, которые необходимо обрабатывать на различных итерациях алгоритма. При этом подмножества вершин могут быть представлены совершенно по-разному, к примеру, в виде списка индексов входящих в подмножество вершин, или же массива флагов принадлежности. Были предложены различные условия и критерии, при которых для заданного графового алгоритма и целевой архитектуры выгоднее использовать тот или иной формат представления подмножеств активных вершин. Так, в фрейморке Ligra [3] используется критерий, основанный на числе исходящих из подмножества смежных ребер, а в [43] – критерии на основе доли от общего числа вершин в графе.
– **Специфические микроархитектурные оптимизации.** К этому направлению относятся различные программные оптимизации и подходы, сильно зависящие от целевой архитектуры. Так, например, для определенных архитектур важно устранять конфликты, возникающие при обращениях к памяти, использовать гипертрейдинг, определенный режим распределения итераций цикла между вычислительными ядрами, и многие другие.

1.6 Подходы к оценке производительности, эффективности и локальности реализаций графовых алгоритмов

При разработке эффективных реализаций графовых алгоритмов для любой архитектуры крайне важно определять, какие оптимизации необходимо применять в какой момент. Так как графовые алгоритмы относятся к классу data-intensive, локальность работы с данными является одним из центральных

понятий, определяющих эффективность их реализации. На сегодняшний день существует достаточно большое количество подходов к оценке производительности, эффективности и локальности, применимых как к процессу реализации графовых алгоритмов, так и к программам произвольных классов. Далее будут кратко описаны модели и подходы, использующиеся в данной работе.

Использование традиционных методов оценки производительности на основе метрики Floating Point Operations Per Second (FLOPS) для графовых алгоритмов не информативно, так как графовые алгоритмы слабо задействуют вычислительные ресурсы целевых архитектур. Вследствие этого производительность графовых алгоритмов измеряется при помощи специальной метрики Traversed Edges Per Second (TEPS), определяемой по формуле $\frac{edges\_count}{time}$, где $edges\_count$ – общее количество ребер в обрабатываемом графе, а $time$ – время работы исследуемого графового алгоритма. Именно на основе данной метрики и алгоритма поиска в ширину формируется список Graph500 [47]. При помощи метрики TEPS удобно сравнивать производительность различных реализаций графовых задач, в том числе между различными вычислительными архитектурами. Однако, в отличие от FLOPS, метрика TEPS никак не привязана к пиковым характеристикам целевой архитеткуры, в следствии чего на основе данной метрики невозможно оценить эффективность использования реализацией аппаратных ресурсов целевой архитектуры.

Вопрос, насколько эффективно исследуемая реализация графового алгоритма использует ресурсы целевой архитектуры, является центральным при оптимизации, так как позволяет понять, следует ли продолжать дальнейшую оптимизацию и какой прирост производительности программы еще можно получить. Ответ на данный вопрос позволяет дать roofline-модель [48], которая оценивает эффективность использования исследуемым приложением как вычислительных ресурсов, так и подсистемы памяти целевой архитектуры. Существует важное расширение roofline-модели – Cache-aware roofline-модель [49], учитывающее особенности иерархии памяти архитектуры. Для построения roofline-моделей были предложены различные инструменты, в том числе [50]. Возможность построения roofline-модели включена в некоторые современные средства анализа производительности для различных архитектур, например в пакет Intel Parallel Studio. Для векторных архитектур roofline-модель так же может применяться, однако для её генерации необходима разработка специализированных инструментов.

Несмотря на то, что roofline-модель позволяет оценить потенциал дальнейшей оптимизации приложения, она не позволяет ответить на вопрос, что является текущим узким местом исследуемой реализации – иными словами, какие аппаратные факторы препятствуют дальнейшему увеличению производительности. С ответом на данный вопрос могут помочь средства профилировки и анализа производительности для различных архитектур: NVIDIA visual profiler, Intel VTune, NEC FTrace, покрывающие все рассматриваемые в данной статье архитектуры с быстрой памятью. Однако, совместно с данными средствами для графовых алгоритмов выгодно дополнительно использовать и формальные модели пространственной и временно́й локальности, так как большинство оптимизаций графовых алгоритмов нацелено на повышение эффективности использования подсистемы памяти целевых архитектур. Подобные модели позволяют сравнить работу с подсистемой памяти для различных подходов к оптимизации: при использовании различных форматов хранения графа, шаблонов доступа к памяти, и другие. Для оценок временно́й и пространственной локальности графовых алгоритмов удобно использовать модели на основе понятий stack и reuse distance [51; 52]. Аналогичные модели были предложены и для GPU [53].

Таким образом, существует большое количество подходов и средств, позволяющих анализировать производительность и эффективность создаваемых в ходе данного исследования реализаций графовых алгоритмов. Однако, для векторных архитектур необходимы как уточнение многих из приведенных моделей, так и систематизация и формальное описание подхода, позволяющего контролировать процесс создания эффективных реализаций графовых алгоритмов.

1.7 Типы и свойства входных графов, используемых в работе

В данной работе для всестороннего анализа производительности и эффективности разработанных реализаций графовых алгоритмов используются как синтетические, так и графы реального мира с различными характеристиками. Генераторы синтетических графов позволяют варьировать различные параметры создаваемых графов, что помогает отслеживать взаимосвязь данных параметров (масштаба, средней степени, характера распределения врешин,

Таблица 2 — Свойства и основные характеристики используемых в данной работе графов.

Название	Тип	Число вершин ($\|V\|$)	Число ребер ($\|E\|$)	Размер графа (в CSR формате)	Размер косвенно запрашиваемых данных	Распределение степеней вершин	Количество нетривиальных компонент (сильной) связности	Размер максимальной (сильно) связной компоненты
rmat_20_16	синтет.	2^{20} = 1m	2^{25} = 33m	271 MB	4 MB	степенное	510	550390 (52%)
rmat_26_16	синтет.	2^{26} = 67m	2^{31} = 2b	17 GB	256 MB	степенное	14	550390 (52%)
ru_20_16	синтет.	2^{20} = 1m	2^{25} = 33m	280 MB	4 MB	равномерное	1	1048576 (100%)
ru_24_16	синтет.	2^{24} = 16m	2^{29} = 536m	4 GB	67 MB	равномерное	1	16777216 (100%)
livejournal	соц.	5.2m	49m	450 MB	20 MB	степенное	1	3306800 (63%)
orkut	соц.	3m	117m	900 MB	20 MB	степенное	1	3072441 (100%)
pokec	соц.	1.6m	30m	260 MB	6 MB	степенное	7434	1427459 (87%)
youtube	соц.	3.2m	9.3m	110 MB	12 MB	степенное	4	1522935 (47%)
wikipedia_en	веб.	12m	378m	3.1 GB	28 MB	степенное	678	12114017 (99%)
web_trackers	веб.	27m	140m	1.4 GB	110 MB	степенное	1	13892605 (50%)
wikipedia_ru	веб.	2.8m	82m	690 MB	11 MB	степенное	130	2852338 (99%)

и другие.) с производительностью разработанных реализаций. В то же время, графы реального мира позволяют более точно исследовать производительность при решении реальных задач. В данной работе используются синтетические RMAT [54] и равномерно-случайные [55] графы, и графы реального мира из коллекций KONEKT [56] и SNAP [57]. В Табл. 2 приведены основные характеристики используемых в данной работе графов, наиболее влияющие на производительность разработанных реализаций.

Приведенные в Табл. 2 характеристики входных графов имеют тесную взаимосвязь с эффективностью разрабатываемых реализаций. Например, размер графа определяется общим количеством его вершин и ребер, причём общее количество ребер в графе определяет, возможна ли обработка данного графа при помощи одного процессора (системы с общей памятью), или же требуется out-of-core обработка. В то же время, количество вершин в графе определяет возможность использования кэш-памяти процессора при обработке косвенных

обращений к информации о вершинах: в исследуемых в данной работе алгоритмах обычно косвенно запрашиваются 4-байтные или 8-байтные данные о вершинах. Однако возможность использования кэш-памяти так же определяет и характер распределения степеней вершин графа: в случае, если распределение степеней вершин неравномерно, появляется возможность кэшировать информацию только о наиболее значимых вершинах графа. В данной работе рассматриваются графы с двумя основными типами распределений: степенным и равномерным. Характер распределения степеней вершин так же влияет на подходы к параллельной и векторной обработке графа. Диаметр графа напрямую влияет на производительность многих итеративных графовых алгоритмов, основанных на поэтапных обходах графов по «слоям», в том числе основанных на поиске в ширину или поиске кратчайших путей: чем больше диаметр графа, тем дольше будут работать данные алгоритмы. Аналогичным образом на производительность ряда алгоритмов влияет количество компонент связности или сильной связности.

1.8 Выводы главы

В первую очередь в данной главе были выделены три основных свойства графовых алгоритмов: (1) принадлежность к классу data-intensive, (2) нерегулярность, а также (3) значительный ресурс параллелизма. На основе трех данных свойств был определён класс современных вычислительных систем, наиболее перспективных для ускорения решения различных графовых задач. Было показано, что все данные системы обладают тремя характерными свойствами: (1) высокой степенью ресурса внутреннего параллелизма, (2) наличием быстрой памяти, а также (3) присутствием различного рода элементов векторной обработки данных. Вследствие этого, системы данного класса могут быть характеризованы как векторные архитектуры с быстрой памятью, и именно данный класс систем используется для создания эффективных реализаций графовых алгоритмов в данном исследовании.

Так же был произведен обзор существующих реализаций графовых алгоритмов для многоядерных центральных процессоров, графических ускорителей NVIDIA GPU и векторных систем, демонстрирующий как актуальность раз-

работки новых подходов к реализации и оптимизации графовых алгоритмов для современных векторных систем с быстрой памятью, так и необходимость создания эффективных библиотечных реализаций графовых алгоритмов для систем данного класса. Наконец, были рассмотрены методы, позволяющие оценить производительность, локальность и эффективность разрабатываемых в ходе данного исследования реализаций графовых алгоритмов.

Глава 2. Целевые архитектуры: основные свойства, характеристики, взаимосвязь

Векторные архитектуры и графические ускорители имеют выделенные в разделе 1.2 общие свойства – использование принципов векторной обработки данных, наличие массивного параллелизма и быстрой памяти. Основным общим свойством является использование принципов векторной обработки данных, что обуславливает принадлежность данных архитектур к классу SIMD (Single Instruction Multiple Data) в классификации Флинна [58] и накладывает сильные ограничения на классы программ, реализуемых эффективно на данных архитектурах. Традиционно считается, что более всего для реализации на SIMD-архитектурах подходят программы с наличием data-driven параллелизма – выполнения однотипных операций над различными данными. Наличие схожих аппаратных особенностей у векторных архитектур и графических ускорителей может означать, с одной стороны, применимость для векторных систем имеющихся для архитектуры NVIDIA GPU подходов к оптимизации графовых алгоритмов, что потенциально может существенно упростить процесс создания эффективных реализаций для векторных архитектур. С другой стороны, этот же факт может означать применимость предложенных в данной работе подходов к разработке и оптимизации графовых алгоритмов не только для других векторных архитектур, но и для новейших графических ускорителей NVDIA GPU, что, как будет показано, позволяет значительно улучшить производительность существующих библиотечных реализаций.

В начале данной главы будут рассмотрены основные аппаратные и программные характеристики архитектур NVIDIA GPU, NEC SX-Aurora TSUBASA и Intel KNL, поскольку в проведенном диссертационном исследовании сделан значительный акцент на создании эффективных реализаций графовых алгоритмов именно для данных архитектур. Ориентация на эти архитектуры обусловлена тем, что: (1) данные архитектуры обладают наиболее быстрой на сегодняшней день памятью, и, кроме того, (2) подходы к реализации и оптимизации графовых приложений для данных архитектур применимы для любых других векторных архитектур с быстрой памятью. После этого в данной главе будет приведен анализ схожих и отличительных особенностей векторных процессоров и графических ускорителей, опирающийся на приведен-

ные аппаратные характеристики, программные свойства, бенчмарки, а также примеры некоторых реальных программ и приложений.

2.1 NEC SX-Aurora TSUBASA

NEC SX-Aurora TSUBASA – новейшая суперкомпьютерная архитектура семейства SX [4; 59], анонсированная и выпущенная в 2018-2019 годах. Архитектура SX-Aurora TSUBASA наследует концепции дизайна векторных суперкомпьютеров предыдущих поколений для достижения более высокой устойчивой производительности на различных классах задач. В отличие от своих предшественников [60; 61], архитектура системы SX-Aurora TSUBASA состоит из соединенных через шину PCI (Peripheral Component Interconnect) векторного устройства (vector engine – VE), являющегося основным векторным процессором, а также векторного хоста (vector host – VH), являющегося процессором архитектуры x86 (Рис. 2.1). Таким образом, система SX-Aurora TSUBASA построена на основе модели «процессор + ускоритель». Однако, в отличие от архитектуры NVIDIA GPU, VE используется в качестве основного процессора для выполнения приложений, в то время как VH используется в качестве сопроцессора для выполнения функций базовой операционной системы, а также части последовательных (не векторизуемых) вычислений, которые могут быть явно выгружены пользователем с векторного устройства. Отметим, что название SX-Aurora TSUBASA далее будет применяться для описания как всей системы целиком (VH + VE), так и только лишь одного векторного устройства (VE), второе – значительно более часто, так как реализация подавляющего большинства графовых алгоритмов возможна без использования векторного хоста для непосредственных вычислений.

Векторное устройство (VE) имеет восемь мощных векторных ядер, каждое из которых обеспечивает производительность в 537,6 Гфлоп/с для вычислений с одинарной точностью при частоте 1,40 ГГц. Таким образом, пиковая производительность всего VE достигает 4,3 Тфлоп/с. Каждое векторное ядро SX-Aurora состоит из блока скалярной обработки (scalar processing unit – SPU), блока векторной обработки (vector processing unit – VPU) и подсистемы памяти (Рис. 2.2). Большинство вычислений выполняется при помощи VPU, в то время как SPU

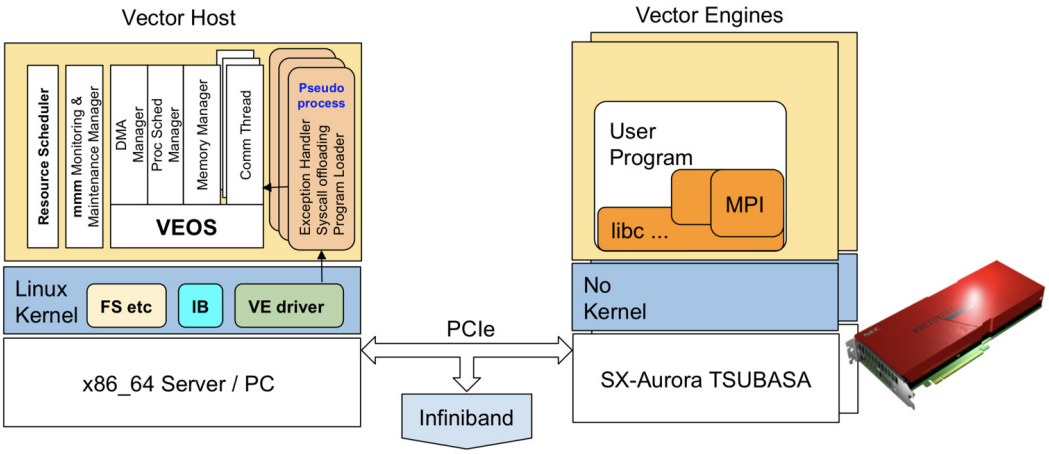

Рисунок 2.1 — Архитектура системы NEC SX-Aurora TSUBASA. Векторное устройство(VE) – справа, векторный хост (VH) – справа

обеспечивают функциональность типичного центрального процессора. Поскольку SX-Aurora является не просто ускорителем, а самодостаточным процессором, 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 переупорядочивания векторных команд, поступающих из SPU. Декодированные инструкции выполняются на векторно-параллельных конвейерах (vector parallel pipeline – VPP). Для хранения результатов промежуточных вычислений каждое векторное ядро оснащено 64 векторными регистрами с общей емкостью регистра, равной 128 КБ, при чем каждый регистр предназначен для хранения векторов из 256 элементов двойной точности.

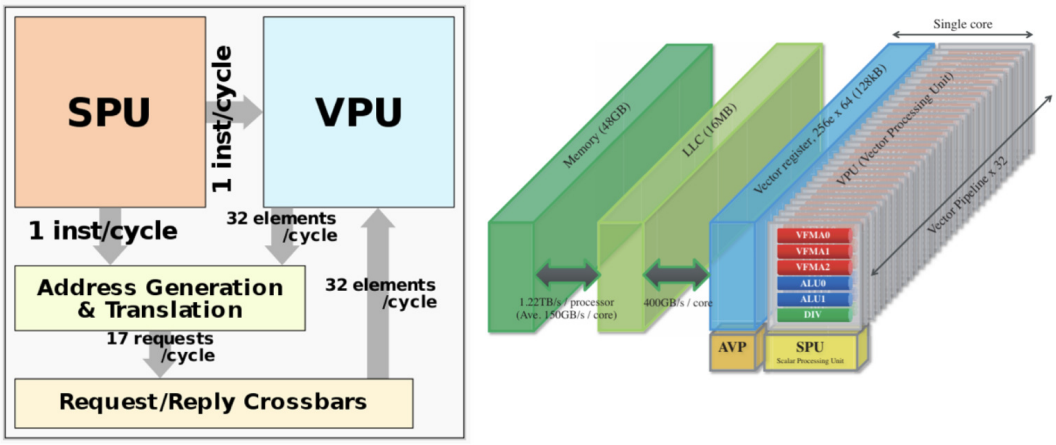

Рисунок 2.2 — Устройство векторного ядра NEC SX-Aurora TSUBASA (слева) и взаимодействие векторных конвейеров с подсистемой памяти (справа)

Векторная обработка внутри каждого из ядер VE основана на использовании 32 идентичных параллельных конвейеров, обрабатывающих расположенные на регистрах вектора частями по 32 элемента в соответствии с моделью SIMD. Таким образом, одна инструкция, оперирующая с векторами из 256 элементов, будет обработана за 8 тактов процессора при условии, что не происходило остановок, вызванных командами с высокой латентностью (например загрузки данных из подсистемы памяти). Каждый VPP имеет 3 блока FMA (Fused Multiply-Add), 2 арифметико-логических блока (arithmetic-logic unit - ALU) и 1 блок, предназначенный для обработки команд с высокой латентностью (извлечение квадратного корня, деления и другие), а также блок работы с подсистемой памяти. В зависимости от структуры программы, необходимые данные перенаправляются между вычислительными блоками VPP, образуя векторный конвейер.

Подсистема памяти векторного устройства состоит из шести модулей HBM-памяти (Рис. 2.3), обеспечивающих на момент написания данной работы самую высокую в мире пропускную способность памяти, равную 1,22 ТБ/с. Для достижения высокой производительности на memory-bound приложениях не менее важна и иерархия кэш-памяти векторного устройства: векторные инструкции используют кэш последнего уровня (LLC) объема 16 Мбайт (по 2 Мбайт на ядро), и пропускной способностью около 3 Тб/с. Важно отметить, что кэши L1 и L2 также присутствуют в Vector Engine, однако используются только для работы SPU.

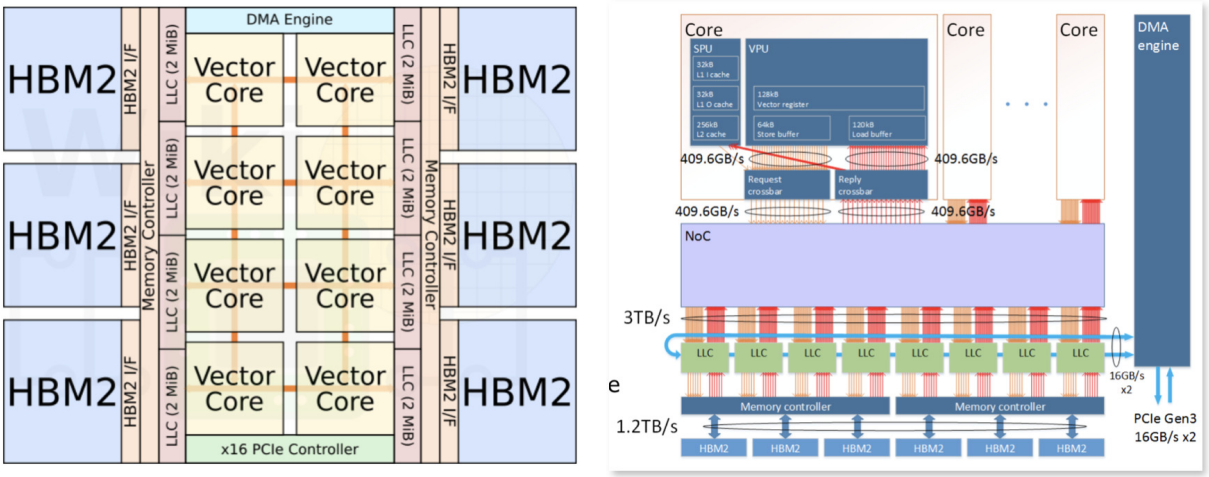

Рисунок 2.3 — Архитектура подсистемы памяти векторного устройства: обращения к быстрой HBM памяти кэшируются в LLC кэше

Векторные программы для системы NEC SX-Aurora TSUBASA разрабатываются при помощи специализированного компилятора nc++. Данный компилятор предлагает как широкие автоматические возможности для автоматического распараллеливания и векторизации программ, так и возможность использования стандарта параллельного программирования OpenMP и специализированных директив для векторизации более сложных программ вручную [62].

Приведенное в данном разделе описание архитектуры NEC SX-Aurora TSUBASA подкрепляет гипотезу о том, что данная архитектура может быть крайне перспективна для решения графовых задач благодаря наличию в ней массивного параллелизма (векторные инструкции длины 256 для 8 векторных ядер) а также быстрой HBM памяти с пропускной способностью 1.2 Тб/с.

2.2 Графические ускорители NVIDIA

Графические ускорители NVIDIA GPU так же устанавливаются в систему в виде сопроцессора, однако, используемая ими модель вычислений существенно отличается от описанной ранее для архитектуры NEC SX-Aurora TSUBASA. Как показано на Рис. 2.4, программа для графического ускорителя изначально запускается на хосте, после чего отдельные участки со значительным ресурсом параллелизма могут быть выгружены на ускоритель, с предварительным копированием необходимых входных данных. В качестве хоста для процессоров используются как процессоры Intel архитектуры x86, так и процессоры IBM Power, а для соединения используются шины PCI или NVLINK.

2.2.1 NVIDIA Pascal

Pascal [63] – кодовое имя современной архитектуры графических ускорителей NVIDIA, являющейся преемницей архитектур Maxwel и Kepler. Графический процессор NVIDIA Tesla P100, который является одним из самых известных представителей архитектуры Pascal, используется в данной работе

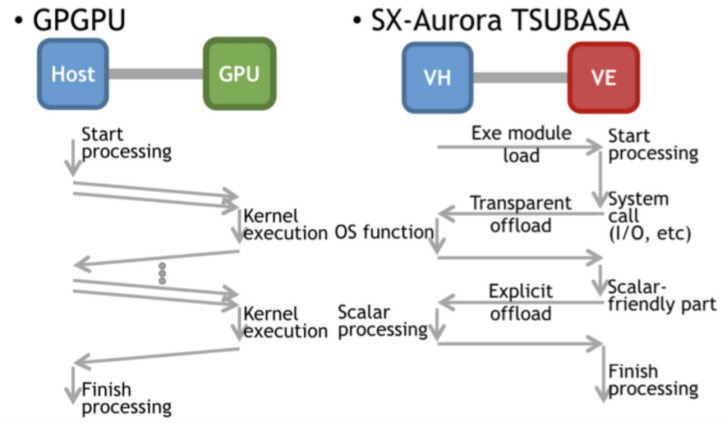

Рисунок 2.4 — Различия в модели вычислений между архитектурами NVIDIA GPU и NEC SX-Aurora TSUBASA

для разработки большинства реализаций графовых алгоритмов. Графический ускоритель P100 оснащен 3584 легковесными CUDA[1]-ядрами с тактовой частотой 1,1 ГГц, что позволяет ему достигать производительности в 9,3 ТФлоп/с при вычислениях с одинарной точностью.

CUDA-ядра в архитектуре Pascal сгруппированы в потоковые мультипроцессоры (Streaming Multiprocessor – SM), каждый из которых состоит из 64 ядер CUDA. CUDA-ядра потоковых мультипроцессоров имеют различные типы, необходимые для выполнения операций с одинарной или двойной точностью, работы с памятью или же специальных математических вычислений. Потоковые мультипроцессоры GPU так же оборудованы планировщиками варпов (warp schedulers), необходимыми для организации работы CUDA-ядер согласно SIMD модели, а также регистрами и быстрой памятью.

Модель выполнения программы на GPU схожа для всех современных поколений графических процессоров. При запуске вычислений на GPU создается большое количество нитей, причём обычно рекомендуется создавать десятки и сотни тысяч нитей для более эффективной работы с подсистемой памяти. Нити объединяются в вычислительные блоки (обычно до 1024 нитей), после чего различные блоки размещаются планировщиком Giga Thread Engine на различных потоковых мультипроцессорах, которые выполняют потоки команд каждой из нитей блока. При этом нити одного блока группируются в так называемые варпы по 32 нити круговом порядке, при чём каждый варп в любой момент времени выполняет одну и ту же инструкцию над различными данными. Имен-

[1]CUDA – Compute Unified Device Architecture

но данное свойство и обуславливает наличие принципов векторной обработки данных в архитектуре NVIDIA GPU – нити одного и того же варпа производят вычисления с использованием SIMD инструкций длины 32.

Память графического ускорителя P100 состоит из четырех стеков HBM, обеспечивающих до 16 Гб памяти с пропускной способностью до 720 Гб/с. Потоковые мультипроцессоры имеют разделяемый кэш L2 суммарного объема 2 Мбайт и приватные L1 кэши объема по 64 Кбайт на мультипроцессор. Важно подчеркнуть, что кэш-память GPU имеет существенно меньший размер по сравнению как с традиционными центральными процессорами, так и векторными устройствами NEC SX-Aurora TSUBASA.

Программирование для архитектур NVIDIA GPU часто осуществляется либо с использованием программной модели CUDA [64], либо при помощи параллельной технологии OpenACC [65], либо с использованием специализированных GPU-библиотек (Thrust, cuRand, cuBlas, cuSpase, NVGRAPH, и другие).

2.2.2 NVIDIA Volta

Архитектура Volta и её представитель V100 [66] используются в данной работе для оценки производительности разработанных реализаций графовых алгоритмов. В ряде существующих исследований производительность графических процессоров V100 сравнивалась с производительностью процессоров SX-Aurora, поскольку они имеют сопоставимые технические характеристики, в том числе производительность вычислений с двойной точностью и пропускную способность памяти. Архитектура Volta наследует большинство вычислительных и архитектурных особенностей архитектуры Pascal, но также и использует несколько важных нововведений. К примеру, графические процессоры Volta включают специализированные тензорные ядра, разработанные для ускорения приложений глубокого обучения. Каждое тензорное ядро предназначено для умножения матриц размера 4x4 в половинной точности с использованием специального конвейера. Архитектура Volta также имеет поддержку интерконнекта NVLINK версии 2.0, что удваивает пропускную способность по сравнению с предыдущим поколением NVLINK (со 40 до 80 Гб/с). Кроме того, ускорители Volta оснащены модулями памяти HBM, обеспечивающими пропускную

способность до 900 Гб/с с общим объемом до 32 Гб. Изменились и характеристики иерархии кэш-памяти – V100 GPU имеет кэш L1 размера 128 Кб и кэш L2 размер 6 Мб. В контексте реализации графовых алгоритмов при переходе от архитектуры Pascal к Volta важны, в основном, улучшенные аппаратные характеристики, связанные с подсистемой памяти [14] – увеличенный объем, пропускная способность и скорость доступа к различными уровням подсистемы памяти.

2.3 Intel Knight Landing

Intel Knight Landing (KNL) является массивно-параллельной векторной архитектурой сопроцессоров Intel Xeon Phi. Данные процессоры оснащены 64–72 ядрами, каждое из которых имеет относительно низкую тактовую частоту от 1,3 до 1,5 ГГц и способно эффективно выполнять до 4 потоков (технология гипертрейдинга). В результате один процессор Intel KNL достигает производительности до 6 Тфлоп/с при вычислениях с одинарной точностью и до 2,6 Тфлоп/с при вычислениях с двойной точностью. Подсистема памяти процессора Intel KNL имеет два уровня: высокоскоростную память MCDRAM с емкостью 16 Гб и пропускной способностью до 400 Гб/с, а также более медленную DDR4 память с емкостью 384 Гб и пропускной способностью до 90 Гб/с. Ядра процессора сгруппированы попарно, так что каждые два ядра имеют общий кэш L2 объемом 1 Мб. Кроме того, каждое ядро имеет собственную кэш-память L1 объемом 64 Кб.

Процессоры Intel KNL имеют поддержку векторных расширений инструкций AVX-512, которые позволяют одновременно обрабатывать по 16 чисел одинарной точности. Несмотря на то, что архитектура Intel KNL на момент написания данной работы является несколько устаревшей, она имеет аппаратные характеристики, сопоставимые с рассмотренным ранее представителями архитектур NVIDIA GPU Pascal и Volta, а также векторными процессорами NEC SX-Aurora TSUBASA. Кроме того, архитектура Intel KNL использует актуальную для реализации графовых алгоритмов модель памяти – сочетание относительно небольшой быстрой памяти вместе с большой по объему, но медленной DDR4, что даёт возможность обработки крупномасштабных графов.

Далее в работе архитектура Intel KNL будет использоваться в основном для демонстрации того факта, что разработанные и оптимизированные векторные реализации графовых алгоритмов будут эффективно работать не только на архитектуре NEC SX-Aurora TSUBASA, но и на других векторных архитектур с минимальными модификациями.

2.4 Векторные процессоры и NVIDIA GPU как представители SIMD-архитектур

В данном разделе будут подробно рассмотрены схожие и отличительные особенности векторных архитектур (на примере NEC SX-Aurora TSUBASA) и графических ускорителей NVIDIA GPU, которые обусловлены использованием SIMD-подхода к вычислениями в данных архитектурах. Для удобства далее будет использоваться понятие «SIMD-инструкции» в качестве обобщения понятий векторной инструкции и варпа GPU.

Общие принципы SIMD-вычислений и структура потока команд Графические процессоры NVIDIA GPU выполняют программный код с использованием большого числа нитей, которые сгруппированы в варпы в циклическом порядке. Каждый варп состоит из 32 нитей (потоков), управление которыми производится с помощью планировщиков варпов, установленных в каждый из потоковых мультипроцессоров GPU. Таким образом, потоковые мультипроцессоры GPU исполняют код при помощи SIMD-инструкций длины 32. С точки зрения программиста каждая GPU-нить выполняет свои собственные скалярные инструкции, однако с аппаратной точки зрения все нити одного и того же варпа выполняют одну и ту же инструкцию (обычно над разными данными) в любой заданный момент времени. В то же время нити различных варпов могут выполнять различные инструкции над различными данными, работая в соответствии с принципами модели MIMD (Multiple Instruction Multiple Data). Данное сочетание принципов SIMD (внутри одного варпа) и MIMD (для различных варпов) моделей выполнения получило название SIMT (Single Instruction Multiple Thread) модели.

Для архитектуры SX-Aurora TSUBASA (а также других векторных архитектур – Intel KNL, A64FX и других процессоров с векторными расширениями) модель вычислений SIMD используется в рамках одного векторного ядра. Каждое векторное ядро имеет свой отдельный поток команд, а значит различные ядра работают в соответствии с MIMD моделью. Таким образом, и графические ускорители NVIDIA, и векторные архитектуры используют сочетание SIMD и MIMD моделей вычислений, в результате чего базовые принципы обработки данных крайне схожи для обеих архитектур. Это может приводить к эквивалентному поведению различных классов программ и алгоритмов на обеих архитектурах в таких ситуациях, как выполнение операторов ветвлений и условных переходов, загрузки данных из подсистемы памяти, а также выполнения различных типов вычислений. В то же время принципы взаимодействия между потоковыми мультипроцессорами графических ускорителей и векторными ядрами SX-Aurora сильно различаются, точно так же, как и присутствуют существенные различия в иерархии памяти данных систем, что в совокупности приводит к значительным различиям в классах эффективно реализуемых алгоритмов для обеих архитектур. Подобные сходства и различия будут более подробно рассмотрены далее в данном разделе.

Дивергенция потока управления

Важной особенностью SIMD-обработки данных для любой вычислительной платформы является последовательность аппаратных действий, выполняемых в случае дивергенции (расхождения, различного поведения) внутри одной SIMD-инструкции. Ситуации дивергенции могут существенно снижать производительность векторизованного кода, и могут относиться к одному из двух типов. Первый тип дивергенции обычно называют дивергенцией потока управления.

Ситуация дивергенция потока управления происходит в процессе выполнения условных операторов ветвления и перехода, примерами которых служат операторы if-then-else, switch-case, а также операторы циклов for и while в языках C и C++. Обычно, дивергенция потока управления происходит в случае, если процесс выполнения SIMD-инструкций зависит от некоторых данных. Для обработки дивергенции потока управления архитектура SX-Aurora TSUBASA генерирует дополнительные векторные инструкции с маскированием, в то время как графические процессоры NVIDIA GPU реализуют специальное поведение для нитей, принадлежащих одному варпу. Простейшая программа,

иллюстрирующая проблему дивергенции потока управления, приведена в листинге 2.1.

Листинг 2.1 — Пример дивергенции потока управления внутри SIMD-инструкции для оператора if-then-else.

```
#pragma simd
for (i = 0; i < 256; ++i)
  if (a[i] >= b[i])          // (1)
    c[i] = a[i]               // (2)
  else                        // (3)
    c[i] = b[i]               // (4)
```

Ситуация дивергенции потока управления в архитектуре SX-Aurora обрабатывается с использованием инструкций маскирования. К примеру, условный оператор из листинга 2.1 преобразуется в 4 отдельных векторных операции: (1) сравнение векторов A и B, на основании которого генерируется векторная маска-результат, (2) маскированное копирование данных, (3) инвертирование векторной маски, и (4) еще одно маскированное копирование данных с использованием инвертированной маски. Таким образом, независимо от структуры ветвления внутри оператора if-then-else, дополнительные векторные инструкции будут генерироваться и выполняться, даже если код следует по одной и той же ветви условного оператора.

Графические ускорители NVIDIA производят обработку конструкции if-then-else немного иным способом. Для программы из листинга 2.1 каждый варп GPU инициирует выполнение ветви if, а затем переходит к ветви else. При выполнении ветви if все нити, имеющие значения условия false, деактивируются. Когда выполнение переходит к ветви else, ситуация меняется на противоположную. Таким образом, ветви if и else выполняются последовательно, а не параллельно, как этого можно было бы ожидать, что крайне похоже на поведение SX-Aurora в той же ситуации. Однако в случае, когда все нити одного варпа должны выполнить одну и ту же ветвь условного оператора, выполнения ветви else не производится, что приводит к снижению производительности для архитектуры GPU, в то время как для архитектуры SX-Aurora производительность всегда будет заведомо ниже независимо от структуры ветвления, поскольку всегда генерируются дополнительные векторные инструкции. Таким образом, для обеих архитектур обработка операции ветвления if-then-else может приводить к двукратному замедлению программы, и к n-кратному замедлению выполнения в случае n условных ветвей (оператор switch-case и другие операторы условно-

го перехода), однако для архитектуры GPU замедление зависит от структуры ветвления, в то время как для векторных архитектур оно постоянно.

Другой менее очевидный пример дивергенции потока управления приведен в листинге 2.2.

Листинг 2.2 — Пример дивергенции потока управления внутри SIMD-инструкции для оператора while.

```
#pragma simd
for (i = 0; i < 256; ++i)
  while(j < borders[i])
        do_some_computations();
```

Ситуация, проиллюстрированная в листинге 2.2, часто возникает при векторизации нескольких вложенных циклов, когда внутренний цикл имеет недостаточно большое число итераций для векторизации, из-за чего производится векторизация внешнего цикла. При компиляции программы из листинга 2.2 на архитектуре SX-Aurora TSUBASA для фрагмента *do_some_computations* будут сгенерированы маскированные векторные инструкции, в то время как архитектура NVIDIA GPU будет деактивировать нити, для которых не верно условие внутреннего цикла. Оба подхода приводят к простою значительного числа вычислительных устройств и существенному снижению производительности в случае, если значения соседних элементов в массиве borders сильно различаются. Как будет показано далее в работе, при реализации графовых алгоритмов данная ситуация встречается крайне часто при обработке вершин графа с различной степенью.

Дивергенция при обращениях к памяти

Другая ситуация дивергенции для SIMD-инструкций связана с обращениями к подсистеме памяти. При загрузке данных из подсистемы памяти зачастую возникает ситуация, когда нескольким элементам SIMD-инструкции не удаётся загрузить данные из кэш-памяти, в результате чего этим элементам SIMD-инструкции приходится ожидать завершения загрузки данных из кэша более высокого уровня или даже основной памяти. Данная ситуация приводит к одновременному простою и тех элементов SIMD-инструкции, которые успешно загрузили данные из кэша более высокого уровня, поскольку аппаратура обрабатывает всю SIMD-инструкцию одинаково и одновременно.

Обе архитектуры NVIDIA GPU и NEC SX-Aurora TSUBASA обрабатывают ситуацию дивергенции памяти схожим образом: в случае, если происходит

промах в кэш по крайней мере для одного элемента SIMD-инструкции, вся инструкция останавливается до тех пор, пока не завершится запрос к более высокому уровню иерархии памяти, что может значительно снижать производительность для memory-bound задач (к которым относятся графовые алгоритмы).

Сочетание дивергенции потока управления и обращений к памяти

Важно также учитывать следующую особенность SIMD-вычислений, возникающую в ситуации условной загрузки данных из памяти, пример которой приведен в листинге 2.3.

Листинг 2.3 — Пример сочетания ситуаций дивергенции потока управления и обращений к памяти.

```
#pragma simd
for (i = 0; i < 256; ++i)
    if (i % 2 == 0)
        C[i] += A[i]
```

В данном примере нити варпов GPU производят загрузку не только тех элементов массивов A и C, которые необходимо загрузить согласно условию if оператора, но так же и части смежных им элементов, попадающих в транзакции размера 32 или 128 байт в зависимости от режима использования кэша L1. Аналогичным образом архитектура SX-Aurora TSUBASA, несмотря на применение маскирования, производит загрузку из памяти всех элементов массивов A и C, даже тех, которые загружать с точки зрения потока управления программы не требуется (с нечетными индексами i). Таким образом для обоих архитектур шина памяти может существенно загружаться за счет пересылок не требуемых программе данных, при чем ситуация становится более проблематичной при работе с сильно разреженными структурами данных, когда для каждой из SIMD-инструкций необходимо загрузить лишь несколько элементов.

Шаблон доступа к памяти SIMD-инструкций

Для обоих рассматриваемых классов архитектур крайне важен шаблон доступа, с которым элементы SIMD-инструкций обращаются к массивам данных. Далее будут приведены наиболее часто встречающиеся шаблоны доступа к данным в реальных программах, проиллюстрированные на Рис. 2.5. Данные

шаблоны одинаково или в разной степени влияют на эффективность использования пропускной способности памяти – основной характеристики, определяющей скорость работы memory-bound программ, и, как следствие, многих графовых алгоритмов.

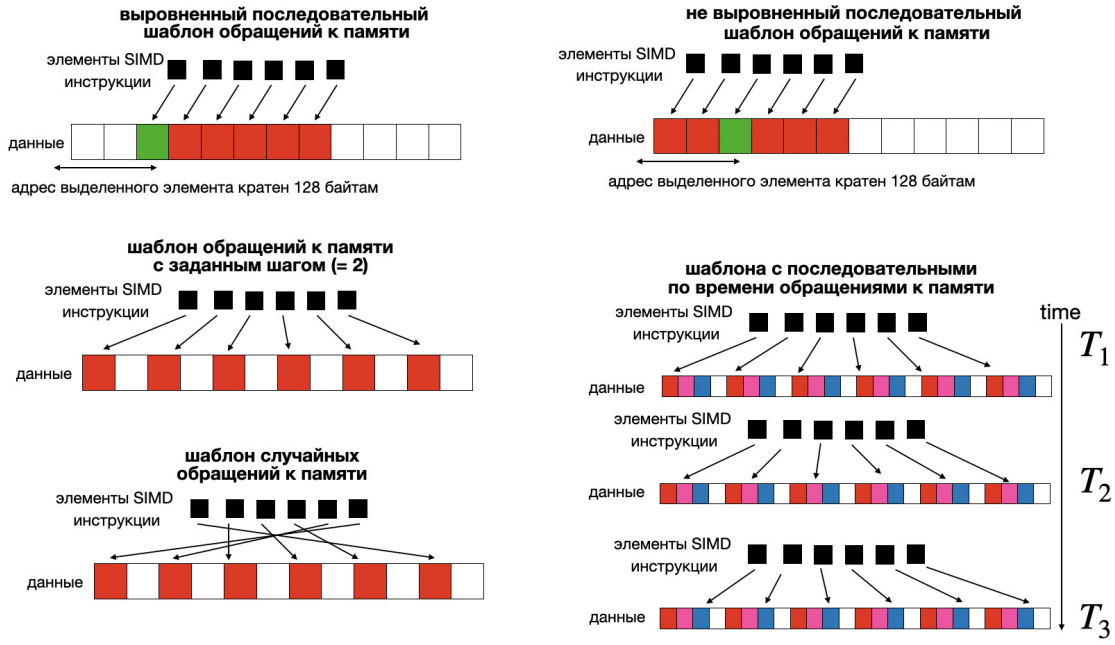

Рисунок 2.5 — Наиболее часто встречаемые шаблоны доступа к памяти для SIMD-архитектур

Для обеих классов рассматриваемых архитектур наиболее предпочтительно использование последовательного шаблона доступа к памяти, когда соседние элементы SIMD-инструкций обращаются к подряд идущим элементам массивов. Данный шаблон позволяет эффективно подгружать данные кэш-линиями, тем самым максимизируя используемую пропускную способность памяти. Однако, для архитектуры NVIDIA GPU так же важно и 128-байтное выравнивание в памяти запрашиваемых элементов, в то время как для архитектуры SX-Aurora выравнивание сильного влияния на достигаемую пропускную способность не имеет. Данный шаблон является одним из наиболее эффективных и, зачастую, позволяет достигать близких к пиковым значений используемой пропускной способности памяти. Реальная пропускная способность при последовательном шаблоне доступа может быть достаточно точно оценена при помощи бенчмарка Stream [67] или его аналогов для GPU [68], и равна 980 Гб/с для SX-Aurora TSUBASA, 560 Гб/с для ускорителей P100 GPU, 760 Гб/с для ускорителей V100 GPU, и 360 Гб/с для Intel KNL.

Таблица 3 — Сравнительная эффективность наиболее часто встречаемых шаблонов доступа к памяти для рассматриваемых архитектур

Шаблон доступа к памяти	NEC SX-Aurora TSUBASA, эффективная пропускная способность, GB/s	NEC SX-Aurora TSUBASA, эффективная пропускная способность, % от пика	Pascal (P100) GPU, эффективная пропускная способность, GB/s	Pascal (P100) GPU, эффективная пропускная способность, % от пика
последовательный	980 Гб/с	80%	560 Гб/с	74%
с постоянным шагом (2)	258 Гб/с	21.5%	182 Гб/с	25%
с постоянным шагом (4)	122 Гб/с	10.1%	91 Гб/с	12%
с постоянным шагом (8)	81 Гб/с	6.7%	46 Гб/с	6.9%
с постоянным шагом (128)	4.19 Гб/с	0.34%	21 Гб/с	2.8%
последовательный по времени	29 Гб/с	2.41%	32 Гб/с	4.4%
случайный (4-байтные данные, помещаются в LLC кэш)	512 Гб/с	42%	467 Гб/с	65%
случайный (4-байтные данные,НЕ помещаются в LLC кэш)	81 Гб/с	6.7%	61 Гб/с	8.5%
случайный (8-байтные данные, помещаются в LLC кэш)	863 Гб/с	71%	644 Гб/с	89%
случайный (8-байтные данные, НЕ помещаются в LLC кэш)	113 Гб/с	9.4%	87 Гб/с	12%

При шаблоне доступа с заданным шагом (stride) элементы SIMD-инструкций обращаются не к соседним элементам массива, а к элементам, расположенным на заданном и постоянном расстоянии друг от друга. В случае, если шаг достаточно велик, то эффективная пропускная способность памяти значительно снижается схожим образом для обеих рассматриваемых архитектур (Табл. 4).

При последовательном по времени шаблоне доступа различные элементы SIMD-инструкций изначально обращаются к произвольным элементам массива, однако, в каждый последующий момент времени доступ для каждого элемента SIMD-инструкции производится к элементам массива, смежным к запрошенным на предыдущем шаге по времени (Рис. 2.5). Данный шаблон одинаково неэффективен для обеих рассматриваемых архитектур, так как реализуется на основе косвенных обращений к памяти (gather и scatter), происходящих при каждом последующем по времени обращении.

Наконец, случайный шаблон доступа к памяти происходит при обращении элементов SIMD-инструкций к случайным (или псевдослучайным) элементам

массива в каждый момент времени выполнения программы, и так же реализуется на основе косвенных обращений к памяти. Обе рассматриваемые архитектуры, во-первых, более эффективно обрабатывают косвенные адресации к 8-байтным данным, чем к 4-байтным. Кроме того, эффективность данного шаблона доступа к данным зависит от того, насколько эффективно можно производить кэширование косвенно запрашиваемых данных: в случае, если косвенно запрашиваемые данные можно расположить в одном из уровней иерархии кэш-памяти, эффективная пропускная способность для обеих архитектур будет значительно выше или по крайней мере сопоставима со скоростью последовательного доступа к памяти. Для графовых алгоритмов объем кэша напрямую отвечает за то, информацию о скольких вершинах графа можно хранить в кэш-памяти в каждый момент времени. Архитектура SX-Aurora TSUBASA имеет LLC размера 16 Мбайт, в то время как ускорители – от 2 до 6 Мбайт.

Обмен данными между различными элементами SIMD-инструкций

Еще одной важной характеристикой рассматриваемых архитектур является подход к реализации обменов данными между различными элементами SIMD-инструкций. Зачастую, во многих алгоритмах необходимо производить обмены (циклические сдвиги данных), полученных на предыдущих шагах алгоритма, результаты которых хранятся на регистрах и еще не помещены в память. Выполнение подобных операций необходимо, в том числе, и при реализации алгоритма параллельного подсчета префиксной суммы [69], используемого во многих графовых алгоритмах, или же при реализации обменов в сетях сортировок. Несмотря на то, что и NVIDIA GPU, и NEC SX-Aurora TSUBASA не имеют возможности прямого обмена данными между регистрами, в архитектуре NVIDIA GPU реализация подобных обменов возможна с использованием разделяемой памяти, скорость доступа к которой лишь незначительно меньше скорости доступа к регистрам. В то же время в архитектуре SX-Aurora TSUBASA данные обмены необходимо производить через основную память, что значительно снижает эффективность приложений, требующих подобных операций обмена.

Обмен данными и синхронизации вычислений между различными векторными ядрами и потоковыми мультипроцессорами
Графические процессоры NVIDIA GPU и архитектура SX-Aurora TSUBASA так же имеют несколько принципиальных отличий в выполнении синхронизаций и обменов данными между потоковыми мультпроцессорами (SM) и векторными ядрами соответственно. Графические процессоры позволяют синхронизировать вычисления только внутри одного вычислительного блока CUDA и, следовательно, только для одного SM. Чтобы синхронизировать вычисления между разными SM, GPU требуется явно запустить новое CUDA-ядро, что приводит к значительным накладным расходам. SX-Aurora позволяет синхронизировать вычисления между каждой парой векторных инструкций, используя стандартную барьерную синхронизацию OpenMP. Кроме того, SX-Aurora предоставляет функциональные возможности для эффективного обмена данными между различными векторными ядрами, реализованные на основе хранения требуемых данных внутри общего для всех векторных ядер LLC-кэша. Таким образом, отсутствие эффективных механизмов синхронизации для различных нитей графических ускорителей накладывает значительные ограничения на алгоритмы, которые могут быть эффективно реализованы на GPU. Операция редукции является актуальным примером, реализуемым на архитектуре SX-Aurora TUBASA значительно более эффективно. Так, эффективная пропускная способность реализации операции редукции для архитектуры SX-Aurora TSUBASA составляет 85% (сопоставима с SAXPY), в то время как для NVIDIA GPU [70] – 62%..

Количество элементов, обрабатываемых SIMD-инструкциями.
Количество элементов, обрабатываемых SIMD-инструкциями архитектуры также играет большую роль при реализации многих приложений. Так, например, при реализации графовых алгоритмов с использованием SIMD-инструкций возможно обрабатывать ребра графа, смежные к заданным вершинам. При меньшей длине SIMD-инструкций появляется возможность более эффективно обрабатывать графы со значительно меньшей степенью. Длина SIMD-инструкции для архитектуры SX-Aurora равна 256, а NVIDIA GPU – 32.

Функционирование на основе модели «хост + сопроцессор».
Так как обе рассматриваемые архитектуры устанавливаются в систему в виде

сопроцессора, то копирования данных между хостом и устройством могут зачастую становиться узким местом в следствии относительно низкой пропускной способности шин PCI и NVLINK. Однако, для архитектуры SX-Aurora TSUBASA, в отличии от GPU, необходимые инициализации и последовательная обработка данных может осуществляться при помощи скалярных ядер векторного устройства, тем самым не требуя передачи данных по шине, что является существенным преимуществом архитектуры SX-Aurora TSUBASA.

Таким образом, обе рассматриваемые в данной работе архитектуры NVIDIA GPU и SX-Aurora TSUBASA, имеют большое число общих свойств, обусловленных использованием SIMD-подхода к вычислениям. **Кроме того, все данные свойства и различия с графическими ускорителями также характерны и для других современных векторных систем**, за исключением некоторых отличий в числовых данных (например, для значений, приведенных в Табл. 4). Однако, как было продемонстрировано в данном разделе, между векторными архитектурами и графическими ускорителями присутствует и ряд отличий, иногда принципиальных, а иногда нет. Данные отличия являются причиной различной эффективности многих реальных приложений и алгоритмов, реализованных на данных архитектурах, примеры которых приведены далее.

2.5 Примеры приложений различных классов, подтверждающие взаимосвязь векторных архитектур и графических ускорителей NVIDIA

В Табл. 4 приведены примеры некоторых часто используемых алгоритмов и приложений общего назначения, имеющих схожую и различную эффективность для архитектур NVIDIA GPU и NEC SX-Aurora TSUBSA. Понятие эффективности в Табл. 4 определяется как процент использования пиковых аппаратных характеристик каждой из архитектур – пиковой производительности или пропускной способности памяти в зависимости от того, к какому классу относится исследуемое приложение (compute-bound или memory-bound). При построении таблицы так же используется понятия «низкой» и «высокой»

Таблица 4 — Сравнительная эффективность различных приложений и алгоритмов для архитектур NVIDIA GPU и NEC SX-Aurora TSUBASA

SX-Aurora \ GPU	Низкая эффективность	Высокая эффективность
Низкая эффективность	**двоичный поиск** **поиск в глубину** *и другие сугубо-последовательные алгоритмы* **бенчмарк random memory access** **бенчмарк HPCG**	**пересечение множеств** **prefix_sum (scan)** **remove_if, copy_if** **сортировка слиянием** *неэффективные обмены данными между элементами SIMD-инструкций*
Высокая эффективность	**операции редукции** *отсутствие механизмов обмена данными и синхронизации для SM* **сложение векторов** **(с учетом времени передачи данных)** *необходимость передач данных через шину для GPU*	**умножение плотных матриц** **stream benchmark** **сеточные задачи** **spmv** **parallel for_each**

эффективности, которые определяются через порог в 40% от пиковых аппаратных характеристик архитектур.

Важно отметить отсутствие графовых приложений и алгоритмов в данной Табл. 4, так как подробный сравнительный анализ для данной группы алгоритмов будет приведен в последующих главах работы, в то время как существующие на момент написания данной работы исследования не позволяют сделать выводы о сравнительной эффективности реализаций алгоритмов данной группы.

Приведенные в Табл. 4 данные подтверждают существенное влияние перечисленных в разделе 2.4 сходств и отличий архитектур NVIDIA GPU и NEC SX-Aurora TSUBASA на сравнительную эффективность многих реальных приложений. Классы приложений, расположенные на диагонали таблицы – крайне широки, в следствии чего можно ожидать, что реализации многих графовых алгоритмов так же будут иметь схожую эффективность для рассматриваемых архитектур.

2.6 Выводы главы

В данной главе были описаны аппаратные характеристики и архитектурные особенности исследуемых в данной работе основных представителей класса

систем с быстрой памятью: векторных процессоров NEC SX-Aurora TSUBASA, графических ускорителей NVIDIA GPU, а также процессоров Intel KNL. Для векторных архитектур (на примере NEC SX-Aurora TSUBASA) и графических ускорителей NVIDIA было выделено большое число схожих (и зачастую идентичных) особенностей организации потока управления, вычислений, а также работы с подсистемой памяти, большая часть которых обусловлена принадлежностью данных архитектур к классу SIMD, то есть использованием векторного подхода к обработке данных. Выделенные схожие вычислительные особенности могут позволить создавать эффективные реализации графовых алгоритмов для векторных архитектур и графических ускорителей с использованием схожих подходов к реализации и оптимизации, что позволит им как иметь высокую степень переносимости между различными рассматриваемыми архитектурами, так и существенно опережать существующие библиотечные аналоги для других классов вычислительных архитектур за счет согласованного со свойствами целевых архитектур выбора подходов к реализации и оптимизации (принцип суперкомпьютерного кодизайна).

Кроме того, в данной главе были выделены различные классы приложений, имеющих как сравнимую, так и принципиально различную эффективность для векторных архитектур и графических ускорителей NVIDIA – различные бенчмарки, операции над плотными и разреженными матрицами, алгоритмы сортировки, и многие другие. Благодаря тому, что классы приложений, имеющие схожую степень эффективности крайне широки и включают в себя, в том числе, операции над разреженными матрицами, по свойствам во многом аналогичные графовым алгоритмам, можно ожидать, что многие графовые алгоритмы так же будут иметь схожую степень эффективности для векторных систем и графических ускорителей, что является важной предпосылкой для выбора векторных систем с быстрой памятью в качестве основного класса целевых архитектур.

Глава 3. Типовые алгоритмические структуры графовых алгоритмов и подходы к их эффективной реализации на векторных системах с быстрой памятью

3.1 Почему важно исследовать типовые алгоритмические структуры графовых алгоритмов?

Существует большое число графовых алгоритмов, обладающих различной информационной структурой и свойствами: сложностью, вычислительной мощностью, ресурсом параллелизма, и другими. Зачастую, существуют и принципиально различные графовые алгоритмы, направленные на решение одной и той же графовой задачи (Рис. 3.1).

Рисунок 3.1 — Примеры различных алгоритмов решения задач поиска в ширину (слева), и поиска связных компонент (справа)

Различные свойства алгоритмов имеют связь с определенными характеристикам вычислительных архитектур, вследствие чего алгоритмы реализуются на разных целевых архитектурах зачастую с принципиально различной степенью эффективности. Важным инструментом исследования информационной структуры алгоритмов являются информационные графы [71], которые позволяют выделить типовые алгоритмические структуры в исследуемых алгоритмах. Дуги информационных графов отражают информационные зависимости между элементарными операциями программы (одна операция использует результат другой), в то время как сами операции – это вершины информационного графа.

Анализируя структуру информационных графов различных графовых алгоритмов важно ответить на следующий вопрос: можно ли для широкого класса графовых алгоритмов выделить типовые алгоритмические структуры? В случае, если:

1. такие алгоритмических структуры выделить можно,
2. данные алгоритмические структуры можно эффективно реализовать на рассматриваемых векторных архитектурах с быстрой памятью,
3. таких алгоритмических структур будет не много (значительно меньше числа исследуемых алгоритмов),

то можно предложить метод создания эффективных реализаций графовых алгоритмов для векторных архитектур с быстрой памятью, опирающийся на использование типовых алгоритмических структур. Данный метод основан на представлении различных графовых алгоритмов в виде комбинации выделенных алгоритмических структур, каждая из которых будет иметь эффективную реализацию на различных рассматриваемых целевых архитектурах, что позволит поддерживать высокую степень производительности, переносимости и продуктивности для разрабатываемых реализаций графовых алгоритмов.

3.2 Исследование информационных графов фундаментальных графовых алгоритмов

В ходе работы были рассмотрены информационные графы следующих графовых алгоритмов:

- алгоритм top-down (задача поиска в ширину);
- алгоритм bottom-up (задача поиска в ширину);
- алгоритм direction-optimizing (задача поиска в ширину);
- алгоритм Беллмана-Форда (задача поиска кратчайших путей);
- алгоритм Дейкстрыи его параллельные модификации (задача поиска кратчайших путей);
- алгоритм delta-stepping (задача поиска кратчайших путей);
- алгоритм Шиллоаха-Вышкина (задача поиска связных компонент);
- алгоритм на основе поиска в ширину (задача поиска связных компонент);
- алгоритм случайного соседа (задача поиска связных компонент);
- алгоритм распространения меток (задача поиска связных компонент, поиска сообществ в графе);
- алгоритм coloring (задача поиска связных компонент);

– алгоритм Forward-Backward (задача поиска сильно связных компонент);
– алгоритм Page Rank (задача ранжирования вершин в графе);
– алгоритм Брандеса (задача вычисления степени посредничества);
– алгоритм, основанный на поиске в ширину (задача вычисления транзитивного замыкания);
– алгоритм Борувки (задача поиска минимального остовного дерева);
– алгоритм, основанный на поиске в ширину (задача вычисления минимального/максимального потока в графе).

Примеры информационных графов некоторых из перечисленных графовых алгоритмов приведены на Рис. 3.2 3.3 3.4 3.5. Приведенные информационные графы состоят из элементарных операций – загрузки из памяти информации о различных компонентах графа (вершинах и ребрах), а также базовых операций над ними (сложений, умножений, сравнений и других). Информационные графы на Рис. 3.2 3.3 3.4 3.5 приведены в ярусно-параллельной форме (на одном ярусе расположены операции, которые могут выполняться параллельно), позволяющей более наглядо отразить параллельные свойства исследуемых алгоритмов.

Для решения задачи поиска кратчайших путей от заданной вершины-источника существует несколько параллельных алгоритмов, включая алгоритм Дейкстры [6; 72], Беллмана-Форда [73] и delta-stepping [74], а также их параллельные обобщения. В алгоритме Беллмана-Форда, информационный граф которого приведен на Рис. 3.2, на каждой итерации все вершины графа и их смежные ребра используются для обновления текущих длин путей до каждой из вершин. Однако для данного алгоритма возможно несложное обобщение, когда на каждой итерации в вычислениях участвуют лишь те вершины, дистанции от и до которых были обновлены на предыдущей итерации, что позволяет значительно уменьшить необходимый объем вычислений.

Для решения задачи поиска в ширину существует три основных алгоритма: top-down, bottom-up и гибридный [11]. Подробный сравнительный анализ и описание всех трех алгоритмов приведен в работе [75]. Алгоритм top-down, информационный граф которого приведен на Рис. 3.3 обходит граф, начиная от вершины-источника, на каждой итерации посещая все вершины, смежные к уже посещенным.

Алгоритм Шилоаха-Вышкина [7; 76] решения задачи поиска связных компонент, информационный граф которого приведен на Рис. 3.4, основан на

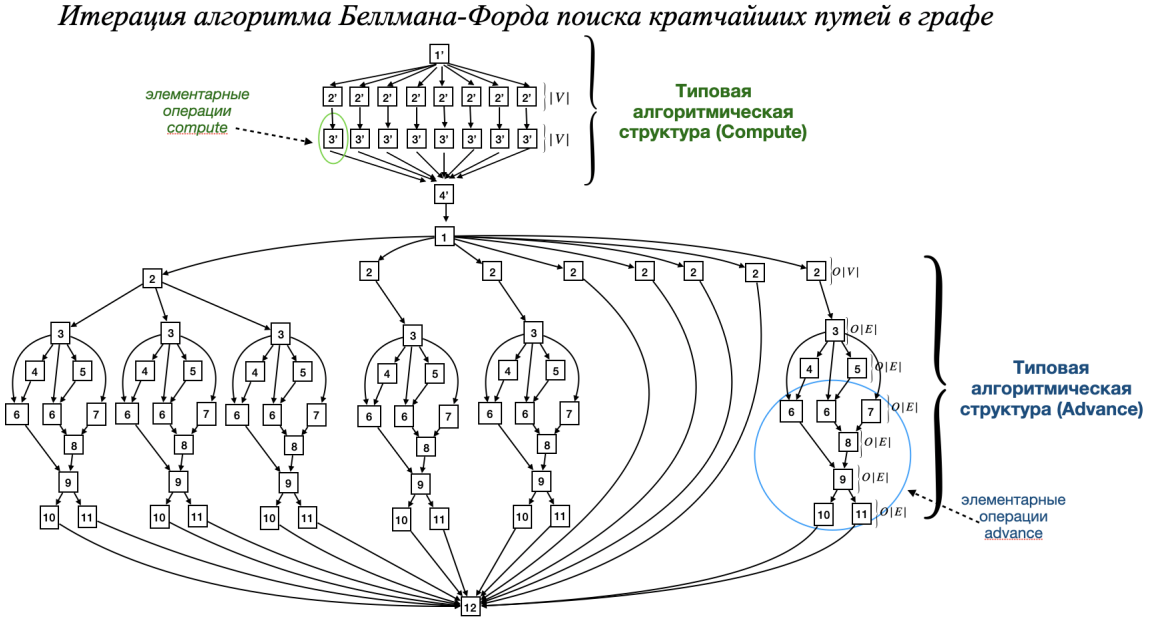

Рисунок 3.2 — Информационный граф одной итерации алгоритма Беллмана-Форда поиска кратчайших путей и его характерные типовые алгоритмические структуры

формировании соответствующих различным связанным компонентам деревьев при помощи операций «закрепления ребер» и «сокращения указателей». Предложенные в работе [76] реализации данных операций позволяют избежать возникновения циклических зависимостей, за счет чего данный алгоритм завершает работу за $O(log|V|)$ шагов, каждый из которых имеет максимальную сложность $O(|E|)$.

Алгоритм Page Rank [6; 77], информационный граф которого приведен на Рис. 3.5, присваивает числовую характеристику каждому элементу из набора документов, связных гиперссылками (например страниц в веб-графе), с целью количественной оценки относительной важности каждого элемента в наборе. Для данного алгоритма также существуют модификации, когда не все вершины и ребра графа на каждой итерации участвуют в вычислениях, а лишь недавно обновленные.

На приведенных информационных графах отмечены «типовые алгоритмические структуры» исследуемых графовых алгоритмов. «Типовая алгоритмическая структура» – подграф в информационном макрографе алгоритма, изоморфный для всех исследуемых алгоритмов при условии их применения к одним и тем же входным данным. Вершины информационного макрографа, соответствующего типовой алгоритмической структуре, отвечают за выполне-

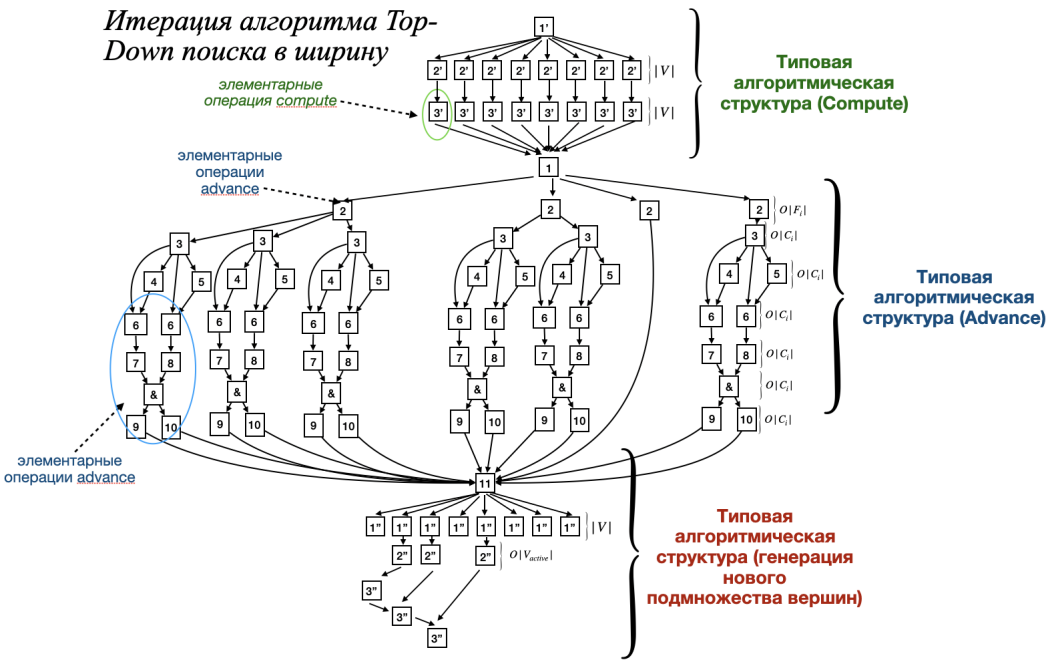

Рисунок 3.3 — Информационный граф одной итерации алгоритма Top-Down поиска в ширину и его характерные типовые алгоритмические структуры

ние групп из элементарных операций – сложений, сравнений, присваиваний и другие, применяемых к вершинами и ребрам обрабатываемого алгоритмом входного графа.

Группы из элементарных операции могут значительно отличаться для различных графовых алгоритмов, как показано на Рис. 3.6. Далее в работе для обозначения данных групп из элементарных операций, соответствующих вершинами информационного графа типовой алгоритмической структуры, будет использоваться термин «алгоритмический шаблон». Итак, «алгоритмический шаблон» – группа из элементарных операций в информационном графе алгоритма, применяемых в рамках выделенных типовых алгоритмических структур для обработки различных вершин и ребер обрабатываемого алгоритмом входного графа.

Тогда макрограф информационного графа алгоритма может быть получен объединением каждого из алгоритмических шаблонов в отдельную вершину информационного графа, как проиллюстрировано на Рис. 3.7 для алгоритма Беллмана-Форда поиска кратчайших путей в графе.

Несмотря на различия в информационной структуре алгоритмических шаблонов для различных графовых алгоритмов, выделенные типовые алгоритмические структуры на уровне макрографа изоморфны между собой для любых

Итерация алгоритма Шилоаха-Вышкина поиска связанных компонент в графе

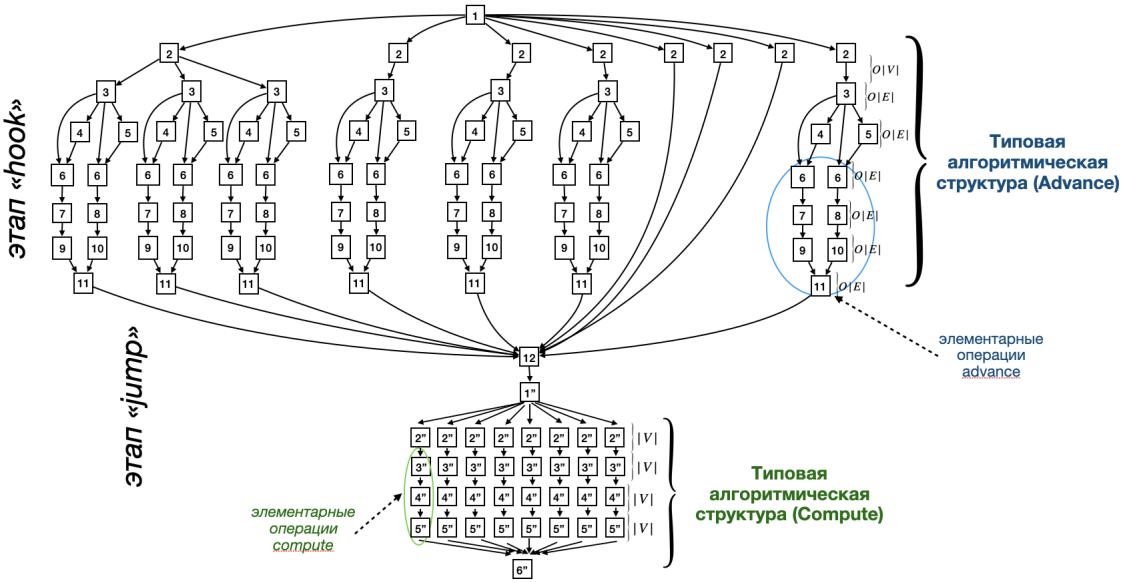

Рисунок 3.4 — Информационный граф одной итерации алгоритма Шилоаха-Вышкина поиска связных компонент и его характерные типовые алгоритмические структуры

двух рассматриваемых в данной работе графовых алгоритмов, при условии их применения к одним и тем же входным данным.

Таким образом, благодаря наличию изоморфизма в рассмотренных графовых алгоритмах можно выделить четыре типовые алгоритмические структуры. Полученные информационные макрографы всех выделенных типовых алгоритмических структур приведены на Рис. 3.8. Для обозначения совокупности выделенных информационных структур, соответствующих им макрографов, а также подходов к реализации для векторных архитектур с быстрой памятью, далее в работе будет использоваться термин «алгоритмическая абстракция» или просто «абстракция».

Первой выделенной алгоритмической абстракцией является применение заданного алгоритмического шаблона к каждому из ребер графа, смежных к некоторому подмножеству вершин. Данная алгоритмическая абстракция далее в работе обозначена как «advance».

Второй выделенной алгоритмической абстракцией является применение алгоритмического шаблона к каждой из вершин заданного подмножества вершин графа. Данная алгоритмическая абстракция далее в работе обозначена как «compute».

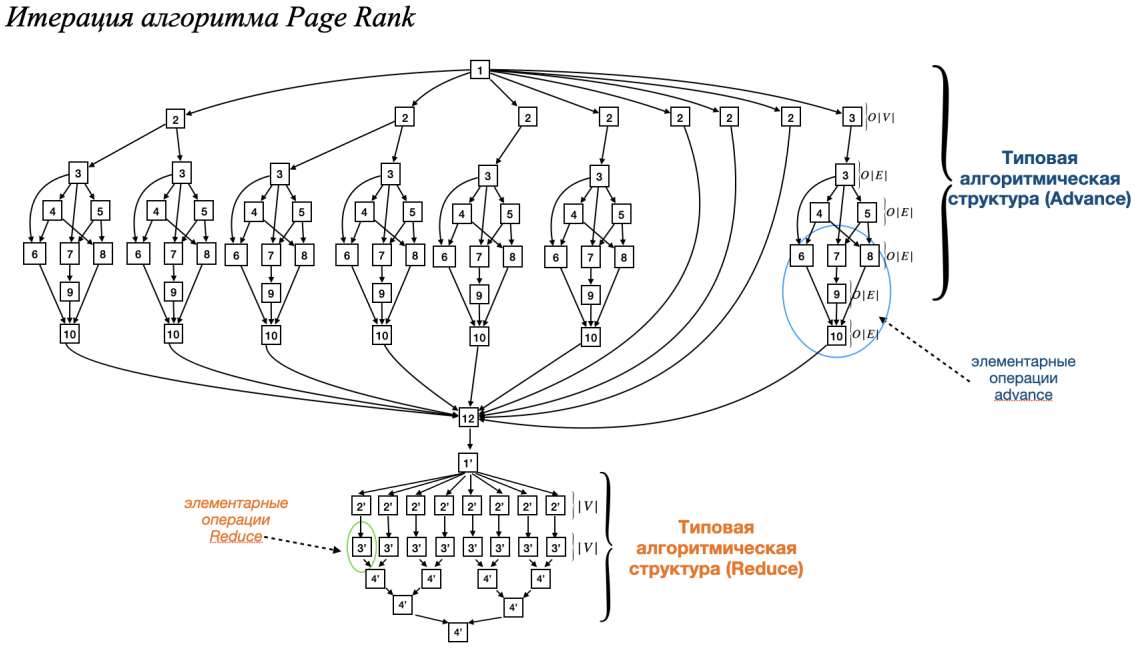

Рисунок 3.5 — Информационный граф одной итерации итерации алгоритма Page Rank и его характерные типовые алгоритмические структуры

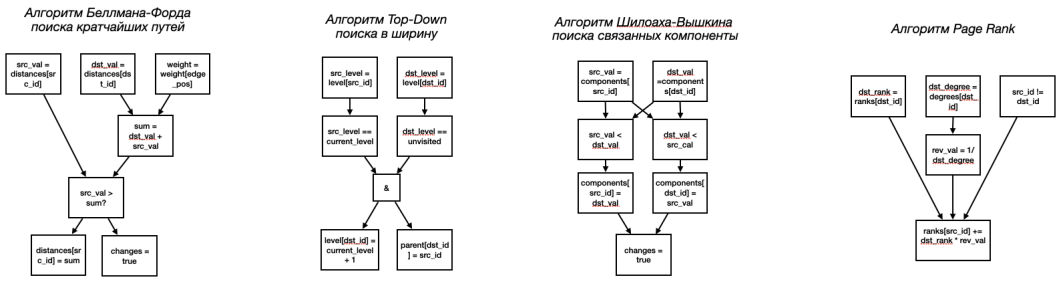

Рисунок 3.6 — Алгоритмические шаблоны нескольких графовых алгоритмов. Каждый из алгоритмических шаблонов представляют собой группу из элементарных операций, выполняемых для обработки ребер графа

Третьей выделенной алгоритмической абстракцией является процедура создания подмножества вершин графа по некоторому заданному условию. Данная алгоритмическая абстракция далее в работе обозначена как «генерация подмножества вершин».

Четвертой выделенной алгоритмической абстракцией является процедура применения алгоритмического шаблона к каждой из вершин заданного подмножества вершин графа, с последующей аккумуляцией некоторых значений, вычисленных для каждой из вершин. Данная алгоритмическая абстракция далее в работе обозначена как «reduce».

Таблица 5 — Наличие выделенных типовых алгоритмических абстракций в различных графовых алгоритмах

графовый алгоритм	графовая задача	advance	compute	reduce	генерация подмножества вершин
Top-Down	поиск в ширину	+	+	-	+
Bottom-up	поиск в ширину	+	+	-	-
Direction-optimizing	поиск в ширину	+	+	+	+
Беллмана-Форда	поиск кратчайших путей	+	+	-	-
Дейкстры и его параллельные модификации	поиск кратчайших путей	+	+	+	+
delta-stepping (задача поиска кратчайших	поиск кратчайших путей	+	+	+	+
Шиллоаха-Вышкина	поиск связных компонент	+	+	-	-
алгоритм на основе поиска в ширину	поиск связных компонент	+	+	+	+
случайного соседа	поиск связных компонент	+	+	-	-
распространения меток	поиск связных компонент/сообществ в графе	+	+	-	+
coloring	поиск связных компонент	+	+	-	-
Флойда-Уоршелла	вычисление транзитивного замыкания	+	+	-	-
алгоритм, основанный на поиске в ширину	вычисление транзитивного замыкания	+	+	-	+
Forward-Backward	поиск связных компонент	+	+	+	+
Page Rank	ранжирование вершин в графе	+	+	+	+
Брандеса	вычисление степени посредничества	+	+	+	+
Борувки	поиск минимального остовного дерева	+	+	-	+
Push/Relabel	вычисление минимального/-максимального потока в графе	+	+	-	+
алгоритм, основанный на поиске в ширину	вычисление минимального/-максимального потока в графе	+	+	+	+

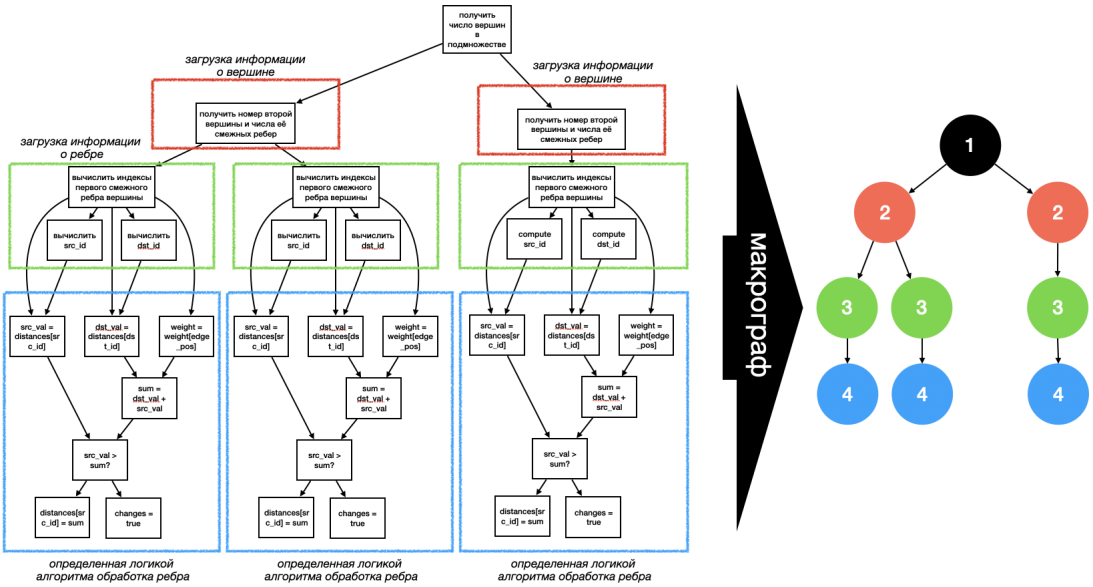

Рисунок 3.7 — Пример преобразования информационного графа типовой алгоритмической структуры advance в алгоритме Беллмана-Форда с уровня элементарных операций на уровень макрографа. Синими рамками выделены алгоритмические шаблоны, характерные для данного алгоритма, красными и зелеными – части типовой алгоритмической структуры advance, соответствующие получению информации о вершинах и ребрах графа

Факт наличия либо отсутствия выделенных типовых алгоритмических структур (абстракций) в различных исследуемых в работе графовых алгоритмов приведен в Табл. 5. Таким образом, анализ информационной структуры приведенного списка графовых алгоритмов подтвердил высказанные в предыдущем разделе гипотезы:

1. Алгоритмы решения всех рассматриваемых в работе задач обладают типовыми фрагментами, причем для рассмотренных 9 задач и 17 графовых алгоритмов удалось выделить всего лишь 4 типовые алгоритмические структуры, входящих в их состав.
2. Вся содержательная часть рассмотренных алгоритмов может быть выражена в терминах этого набора типовых алгоритмических структур.

В случае, если выделенные алгоритмические структуры (а также необходимые сопутствующие структуры данных) возможно эффективно реализовать на целевых векторных архитектурах с быстрой памятью, то будет получен метод создания эффективных реализаций широкого класса графовых алгоритмов для векторных систем.

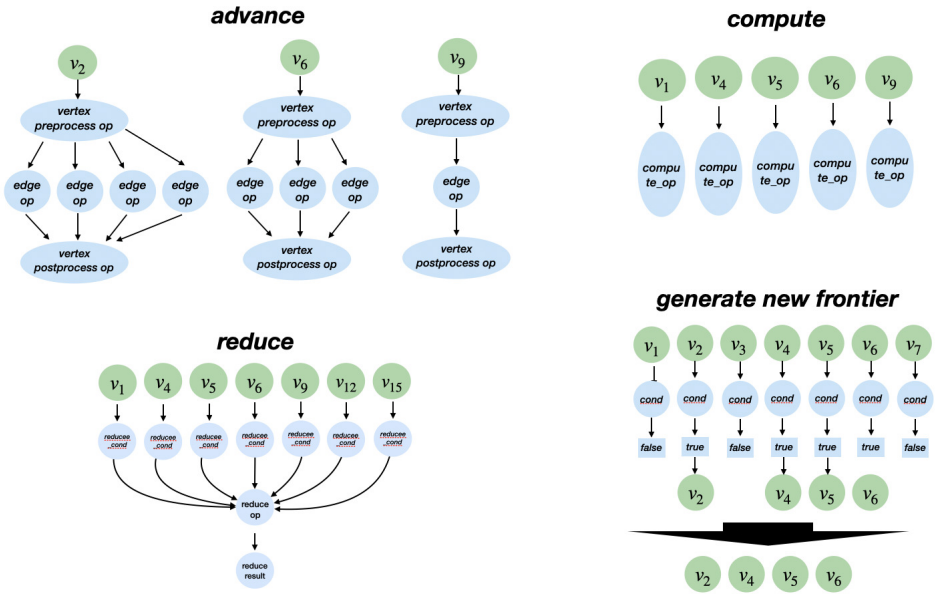

Рисунок 3.8 — Макрографы выделенных типовых алгоритмических структур: advance, compute, reduce и создания подмножества вершин

3.3 Типовые абстракции данных

В алгоритмах нет понятия данных, однако все выделенные в предыдущем разделе алгоритмические абстракции оперируют над следующими графовыми объектами: граф, а также заданные подмножества вершин и ребер графа (Рис. 3.9). В результате, были выделены три следующие абстракции данных:

– граф
– подмножество вершин графа
– подмножество ребер графа

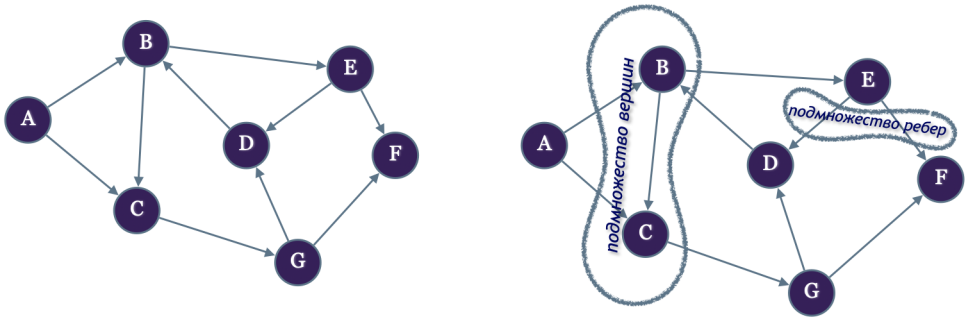

Рисунок 3.9 — Выделенные абстракций данных: граф (слева) и подмножества вершин и ребер данного графа (справа)

Граф – это основная абстракция данных для любого графового алгоритма. Графы бывают ориентированными или неориентированными, взвешенными или не взвешенными, а значит реализации данной абстракции должны естественным образом учитывать эти случаи. Особый интерес представляет выбор формата хранения графа, являющийся наиболее важным атрибутом реализации данной абстракции, причём для различных вычислительных архитектур зачастую более эффективны различные графовые абстракции.

Другие две не менее важные абстракции – подмножества вершин и ребер графа, так как описанные ранее алгоритмические абстракции оперируют с различными подмножествами вершин и ребер графа. К примеру, в алгоритме поиска в ширину на каждой итерации производится обработка вершин, посещенных на предыдущем шаге, а также смежных им ребер – двух явно выделенных подмножеств вершин и ребер графа, с которыми и работает алгоритм. Многие другие алгоритмы на каждой итерации обрабатывают все вершины и ребра графа, которые тоже являются частным случаем подмножеств вершин и ребер. Реализация данных подмножеств может отличаться как в зависимости от числа и типа принадлежащих к ним вершин и ребер, так и для различных целевых архитектур.

Важно отметить, что все перечисленные ранее графовые алгоритмы обрабатывают все ребра графа, смежные к заданному подмножеству вершин, что является важной особенностью выделенной абстракции advance (единственной из четырех, которая работает с ребрами графа). Поэтому, для выделенных алгоритмических абстракций отсутствует необходимость в явной реализации абстракции данных, соответствующей подмножеству ребер графа, так как необходимые для вычислений в абстракции advance подмножества ребер могут быть легко получены из подмножества вершин и непосредственно самого графа. При этом данный факт накладывает на формат представления графа важное ограничение – он должен позволять эффективно загружать информацию о ребрах, смежных к определенной вершине. Таким свойством, к примеру, обладает формат CSR (Compressed Sparse Row), однако не обладает формат списка ребер.

Одним из трех выделенных в разделе 1.2 требований к реализациям графовых алгоритмов для векторных систем является необходимость локальной работы с данными, которая, в свою очередь, определяется как выбранными структурами данных, так и подходами к их реализации. Для поддержки свойств

локальности и векторной обработки данных, необходимо предложить такие структуры данных которые хорошо соответствуют:

1. выделенным ранее алгоритмическим абстракциям,
2. аппаратным и программным особенностям целевых векторных архитектур.

Далее детально описаны подходы к реализации выделенных абстракций данных (графа и подмножества его вершин) для исследуемых векторных архитектур с быстрой памятью.

3.4 Реализация абстракций данных для векторных систем с быстрой памятью

3.4.1 Векторно-ориентированный формат хранения графов

Формат хранения графов в совокупности с используемым алгоритмом в значительной мере определяет шаблоны доступа к памяти, а также принципы распределения параллельной нагрузки между различными вычислительными устройствами и ядрами. Вследствие этого, разработке эффективных векторно-ориентированных форматов хранения графов необходимо уделять большое внимание. Наиболее часто используемым форматом хранения графа является сжатый список смежности (Compressed Sparse Row – CSR). CSR-формат позволяет эффективно хранить информацию об относительной смежности вершин в графе, критичную для реализации выделенной абстракции advance. Однако, традиционный CSR-формат для векторных архитектур использовать затруднительно вследствие необходимости обработки вершин с низкой степенью векторными инструкциями фиксированной длины. Таким образом необходимо либо применение других форматов хранения, либо расширение и модификация CSR-формата.

Другими часто используемым форматами хранения графов является список ребер, список смежности, матрица смежности, а также векторно-ориентированные форматы представления разреженных матриц, например Cell-C-Sigma [78] и Sliced ELLPACK [79], которые можно использовать в том

числе и для хранения графов, вследствие тождественности понятий графа и разреженной матрицы. Каждый из данных форматов имеет существенные недостатки для реализации выделенных алгоритмических абстракций на векторных системах. Так, формат списка ребер и векторно-ориентированные форматы представления разреженных матриц не позволяют эффективно производить быструю загрузку информации о ребрах, смежных к заданному подмножеству вершин графа, что влечет за собой необходимость загрузки из памяти всех ребер графа и существенно увеличивает вычислительную сложность многих реализуемых алгоритмов. Формат списка смежности (не сжатый) не позволяет эффективно копировать граф в память GPU, а также обладает существенно меньшей локальностью вследствие разброса по памяти информации о смежных ребрах для различных вершин, в то время как матрицы смежности имеют слишком большой объем для размещения в оперативной памяти для многих графов реального мира.

Вследствие приведенных недостатков существующих форматов, для векторных архитектур необходима разработка формата, удовлетворяющего следующим требованиям:

- формат хранения графов должен поддерживать реализацию графового алгоритма для обоих направлений обхода ребер (push и pull);
- данный формат должен поддерживать векторную обработку и вершин и ребер графа с использованием максимально возможной длиной вектора (до 256);
- шаблон доступа к памяти о ребрах графов должен быть преимущественно последовательным;
- шаблон доступа к информации о вершинах графа должен быть либо последовательным (абстракции compute, reduce, а также генерации подмножества вершин), либо иметь высокую пространственную и временную локальность для эффективного использования кэш-памяти;
- размер графа в данном формате должен быть сопоставим с существующими традиционными форматами (CSR, CSC, EL и другими) для большинства графов реального мира;
- на основе данного формата должна быть возможна эффективная балансировка нагрузки между различными ядрами и элементами векторных инструкций.

В данной работе для векторных систем был предложен формат VectCSR, который удовлетворяет всем поставленным выше требованиям и является расширением CSR-формата. Формат VectCSR опирается на предобработку графа, позволяющую более эффективно использовать кэш-память при обходах графов, эффективно задействовать векторную обработку данных, а также производить эффективную балансировку параллельной нагрузки между различными ядрами и вычислительными устройствами. Схема предложенного формата VectCSR приведена на Рис. 3.10.

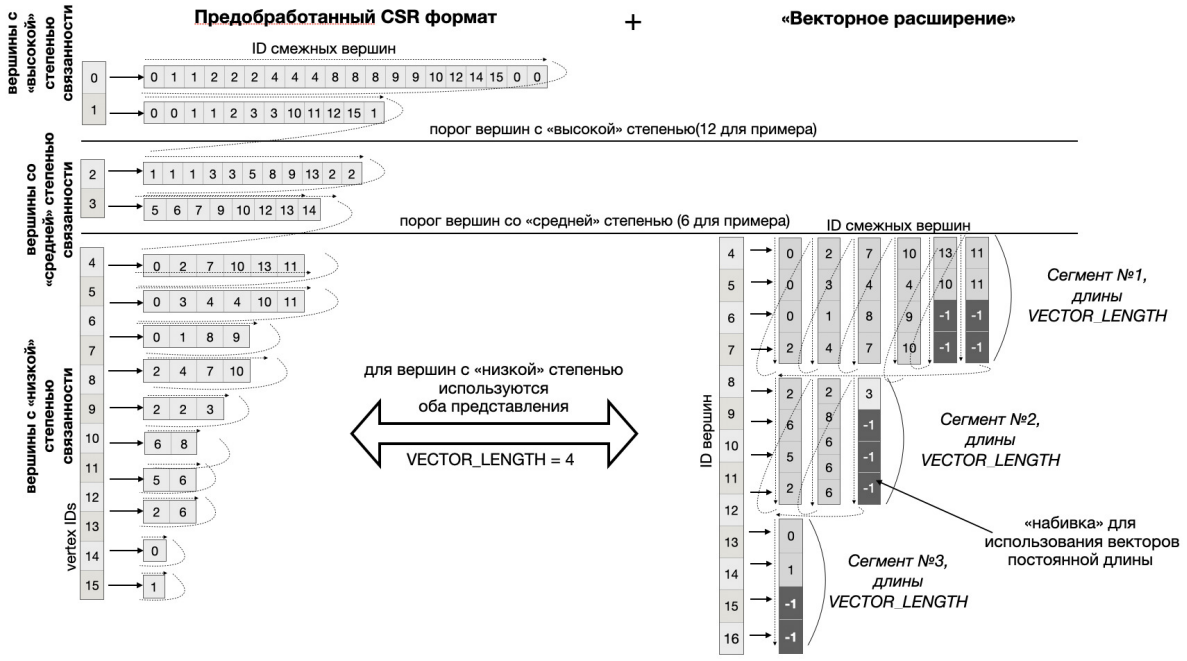

Рисунок 3.10 — Формат хранения графа VectCSR: граф в предобработанном формате CSR (слева) и его векторное расширение (справа)

Для преобразования графа в формат VectCSR используется следующий алгоритм.

1. Выполнить преобразование входного графа во временный CSR-формат (в случае, если граф представлен, к примеру, списком ребер).
2. Произвести сортировку всех вершин графа в порядке убывания количества смежных ребер.
3. Изменить номера направляющих вершин для всё ребер согласно их позициям после сортировки. Важно отметить, что полученный при данных преобразованиях граф эквивалентен исходному, при чем массив соответствий номеров вершин естественным образом генерируется во время сортировки на этапе 2.

4. (опционально) Провести сегментную сортировку смежных ребер для каждой вершины в порядке возрастания идентификатора направляющей вершины.
5. Разделить все вершины графа на группы по степени в соответствии с архитектурно-зависимыми пороговыми значениями. К примеру, на Рис. 3.10 вершины разбиваются на три группы с пороговыми значениями 12 и 6. Порог значений последней группы часто берется равным максимальной длине векторной инструкции целевой архитектуры.
6. Для группы вершин с низкой степенью сгенерировать «векторное расширение» следующим образом:
    а) Разбить вершины последней группы на подряд идущие сегменты, по $vector\_length$ вершин в каждом.
    б) Смежные ребра каждой из вершин сегмента дополнить «петлями», так, чтобы все вершины из одного сегмента имели одинаковое количество смежных ребер.
    в) Инициализировать структуру данных векторного расширения: смежные ребра в векторном расширении хранятся в памяти не подряд, а с интервалом $vector\_length$, с целью гарантировать максимально эффективный шаблон доступа к памяти при коллективной обработке сегментов векторными инструкциями максимальной длины.
7. Для хранения итогового графа в формате VectCSR использовать сочетание отсортированного CSR-формата и векторного расширения.
8. В случае, если для решаемой задачи необходимо использовать обратное направление обхода (pull) для ориентированных графов, дополнительно сгенерировать граф, транспонированный к исходному.

«Векторное расширение» графа используется в зависимости от структуры реализуемого алгоритма. Иногда использовать «векторное расширение» не всегда эффективно, так как в случае работы с сильно разреженными подмножествами вершин будет происходить загрузка излишних данных о ребрах при векторной обработке группы вершин с низкой степенью (ситуация сочетания дивергенции потока управления и обращений к памяти, описанная в разделе 2.4, вследствие чего пропускная способность памяти целевой архитектуры будет использоваться неэффективно. Поэтому «векторное расширение» используется в случае, если на текущем шаге алгоритма необходимо обходить либо все, либо

значительную часть вершин графа, в то время как граф в предобработанном CSR-формате используется для работы с «разреженными» итерациями алгоритма, на которых требуется обработать лишь небольшое число вершин.

Основным недостатком предложенного VectCSR-формата является увеличение объема памяти, необходимого для хранения графов, поскольку одновременно приходится хранить и предобработанный CSR-граф, и его векторное расширение. Однако, для многих графов реального мира со степенным характером распределения вершин данное увеличение не существенно (и составляет до 20%), так как большая часть ребер данных графов сконцентрировано у вершин с высокой степенью, не хранящихся в векторном расширении.

Далее будет приведен анализ формата VectCSR на предмет соответствия трем выделенным в разделе 1.2 требованиям к реализациям графовых алгоритмов для векторных систем с быстрой памятью. Во-первых, формат VectCSR использует предварительную сортировку вершин графа (оптимизацию кластеризации), что позволяет эффективно использовать кэш-память при обработки косвенных обращений к памяти для многих графов реального мира, что соответствует требованию локальности работы с данными. Во-вторых, разбиение вершин на группы по степени позволяет производить эффективную балансировку параллельной нагрузки, выделяя на обработку вершин различных групп разные вычислительные мощности (число ядер и различный подход к векторной обработке данных), что соответствует требованию использования массивного параллелизма целевых архитектур. Наконец, предложенный формат VectCSR использует векторное расширение, что позволяет эффективно задействовать векторные инструкции максимальной длины для обработки вершины (или группы вершин) с любой степенью, что соответствует требованию о необходимости использовать векторную обработку данных. Таким образом, предложенный в данном разделе формат VectCSR соответствует всем сформулированным ранее требованиями, вследствие чего хорошо подходит для всех исследуемых векторных архитектур с быстрой памятью.

Важно отметить, что формат VectCSR может использоваться как составной блок для более сложных представлений графа. К примеру, при применении подхода к сегментированию графа (разбиение обрабатываемого графа на независимые подграфы) каждый из сегментов может представляться графом в формате VectCSR.

3.4.2 Подмножество вершин и его векторно-ориентированная реализация

Предложенная в данной работе реализация подмножества вершин графа для векторных систем зависит от количества (а также степеней) составляющих его вершин. Для эффективной реализации выделенных алгоритмических абстракций реализация подмножества вершин автоматически должна отслеживать количество вершин, помещенных в данное подмножество, одновременно не позволяя поместить в подмножество вершины с дублирующимися номерами.

Подмножества, в которые входят все вершины графа (далее используется обозначение «all-active»), могут быть реализованы без проверок факта принадлежности вершин к ним, что позволяет значительно сократить накладные расходы на инициализацию обработку подмножества данного типа.

Подмножества, в которые входят большая часть вершин графа (далее используется обозначение «dense») реализованы в виде массива флагов принадлежности каждой из вершин к данному подмножеству. В зависимости от типа целевой архитектуры, тип флага может различаться – int для NEC SX-Aurora TSUBASA, char или bool для других архитектур. Каждый флаг может быть легко вычислен исходя из условия принадлежности вершины к подмножеству, которое определяется реализуемым алгоритмом.

В случае, если подмножество включает в себя лишь небольшое число вершин графа, оно имеет смешанный тип (далее используется обозначение «mixed»). Вершины подмножества смешанного типа разделены на группы по степени, идентичные используемым группам в формате представления VectCSR (с точно такими же порогами). Каждая из групп может иметь плотный либо разреженный тип. Плотные группы вершин представлены в виде массива флагов, в то время как разреженные – в виде списка номеров принадлежащих к группе вершин. Благодаря тому, что в предложенном представлении VectCSR вершины графа отсортированы по убыванию степени, для векторных архитектур возможна достаточно эффективная генерация подобного подмножества, алгоритм которого подробно описан в последующих разделах. Необходимость реализации отдельных типов разреженности для вершин каждой из групп с различной степенью обусловлена тем, что многие графовые алгоритмы обычно посещают сначала вершины с высокой степенью, а затем уже со средней и низкой, в ре-

зультате чего различные группы могут иметь разную плотность на одной и той же итерации алгоритма.

Критерии принадлежности подмножеств вершин к различными типам могут определяться как на основе процента от абсолютного числа вершин в графе (например разреженными считаются подмножества, в которых находится меньше 30% вершин графа), так и на основе суммарной степени принадлежащих подмножеству вершин. Наиболее оптимальный подход и пороговые значения зависят как от целевой архитектуры, так и от реализуемого алгоритма.

3.5 Реализация алгоритмических абстракций для векторных систем с быстрой памятью

Крайне важным для исследования является вопрос, как эффективно реализовать выбранные алгоритмические абстракции с учетом выбранных абстракций данных. Далее в данном разделе подробно описаны подходы к реализации выделенных алгоритмических абстракций для различных целевых архитектур, а также основные подходы к их оптимизации.

3.5.1 Алгоритмическая абстракция: генерация подмножества вершин

При создании (генерации) подмножества вершин на первом этапе создается массив флагов, каждый из которых определяет факт принадлежности каждой из вершин графа к генерируемому подмножеству. Каждый из флагов может быть легко получен на основе условия принадлежности вершины к подмножеству, при чём полученные флаги в дальнейшем также используются в плотных и смешанных представлениях подмножества. Вместе с генерацией массива флагов параллельно подсчитывается число вершин в каждой из групп вершин с различной степенью, что может быть реализовано без дополнительных накладных расходов за счет аккумуляции данных значений на векторных регистрах для векторных архитектур, и с помощью аналога операции *count_if*

для NVIDIA GPU. При этом общее количество вершин подмножества вычисляется как сумма полученных значений для от всех групп.

На втором этапе каждая из групп классифицируется как разреженная или плотная на основе количества элементов в ней, а также используемых в алгоритме критериев. В случае, если все 3 группы – плотные, считается, что подмножество также имеет плотный тип, в противном случае – «смешанный».

Наконец, на третьем этапе для каждой разреженной группы выполняется параллельная операция условного копирования, которая генерирует список номеров вершин, принадлежащих данной группе. Под параллельным условным копированием понимается операция, которая на вход принимает массив данных и условие, определяющие необходимость копирования каждого из элементов входного массива данных в выходной. На выходе данная операция формирует массив скопированных данных, хранящихся подряд в памяти.

Эффективная реализация параллельного условного копирования для векторных систем представляет собой значительную сложность, и сильно отличается для векторных систем и графических ускорителей NVIDIA. Для графических ускорителей NVIDIA реализация параллельного условного копирования основывается на алгоритме вычисления параллельной префиксной суммы [69], используемого для вычисления индексов в выходном массиве, куда необходимо поместить скопированные данные. Однако, алгоритм подсчета параллельной префиксной суммы не может быть реализован эффективно для NEC SX-Aurora TSUBASA и других векторных архитектур, поскольку требует большого числа обменов данными (циклических сдвигов) внутри быстрой памяти, отсутствующей в векторных процессорах (например NEC SX-Aurora TSUBASA или Intel KNL). Поэтому, для векторных архитектур был предложен следующий алгоритм: в начале каждое векторное ядро выделяет временные буферы для каждого элемента векторной инструкции, при чем суммарный размер всех буферов равен $O|V|$. Таким образом каждый буфер имеет размер $\frac{|V|}{vector\_cores*vector\_length}$, а выделяется всего $vector\_cores*vector\_length$ буферов. После этого все необходимые для копирования данные переносятся в выделенные временные буферы с использованием векторной операции SCATTER и векторных регистров, хранящих текущее число элементов внутри каждого из буферов. После этого каждое векторное ядро подсчитывает количество элементов в каждом буфере (редуцируя данные о размере буферов на векторных регистрах), после чего все векторные ядра обмениваются полученными значени-

ями и вычисляют начальные сдвиги в массиве-результате при помощи средств OpenMP. Наконец, векторные ядра копируют подряд идущие элементы буферов в массив-результат по вычисленном смещению.

3.5.2 Алгоритмическая абстракция Advance

Алгоритмическая абстракция advance осуществляет обработку ребер графа, смежных к определенному подмножеству вершин. Эффективная реализация данной абстракции для векторных архитектур представляет собой значительную сложность по двум причинам:

– для многих графов реального мира данная абстракция имеет крайне нерегулярную структуру из-за неравномерного распределения степеней вершин, вследствие чего необходима аккуратная балансировка параллельной нагрузки между различными вычислительными ядрами и элементами векторных инструкций;
– при загрузке информации о вершинах графа выполняется большое число косвенных обращений к памяти, которые значительно снижают производительность абстракции advance.

Эффективная реализация абстракции advance возможна благодаря использованию векторно-ориентированных структур данных – формату представления графа VectCSR и векторному представлению подмножества вершин. Далее будут перечислены основные подходы к оптимизации, используемые при реализации абстракции advance для векторных систем с быстрой памятью, а также приведены их детальные описания.

Балансировка параллельной нагрузки при обработке вершин с различной степенью. Подход к балансировке параллельной нагрузки и использованию векторных вычислений при реализации абстракции advance имеет как схожие, так и отличные черты для векторных архитектур и графических ускорителей. Для векторных архитектур вершины графа разбиваются на три группы – с высокой, средней и низкой степенью (Рис. 3.11). Каждая из вершин с высокой степенью обрабатывается всеми доступными векторными ядрами, при чём каждое векторное ядро использует векторные инструкции для обработки групп из *vector_length* смежных к данной вершине ребер, что реализуется од-

новременно параллельным и векторизованным циклом. Каждая из вершин со средней степенью обрабатывается одним векторным ядром, при чем векторные инструкции снова используются для обработки групп из *vector_length* смежных к данной вершине ребер. Наконец, каждые *vector_length* подряд идущих вершин из группы с низкой степенью обрабатываются векторным ядром, при чем каждая из вершин обрабатывается одним элементом векторной инструкции. Благодаря тому, что вершины графа предварительно отсортированы по убыванию степени (формат VectCSR), обработка одной векторной инструкцией *vector_length* вершин не приводит к ситуации дивергенции потока управления, описанной в разделе 2.4. Вершины различных групп обрабатываются по очереди (последовательно), а балансировка параллельной нагрузки осуществляется внутри каждой из групп путём статического распределения вершин (или смежных ребер в случае первой группы) между различными векторными ядрами.

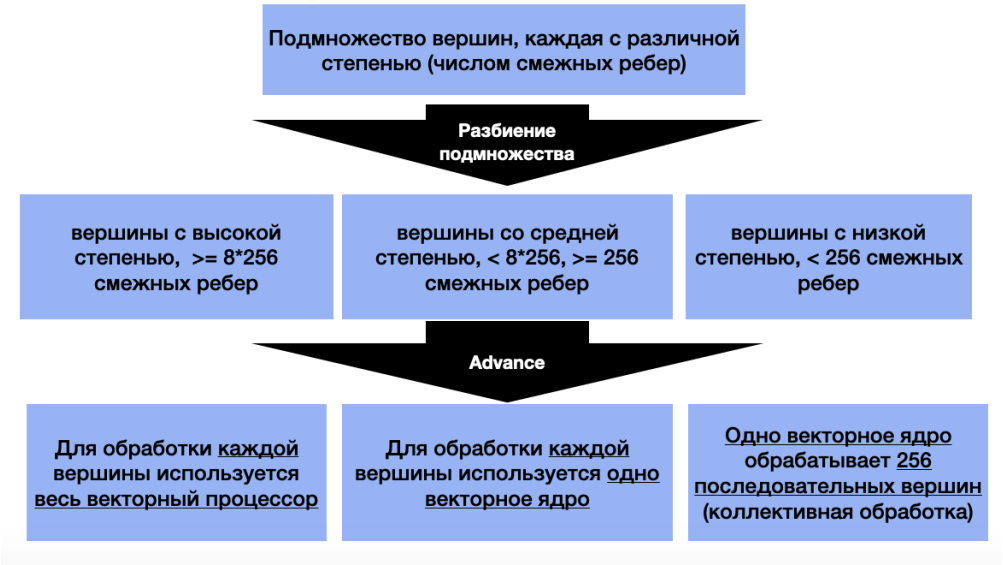

Рисунок 3.11 — Схема обработки групп вершин с различной степенью на векторной архитектуре NEC SX-Aurora TSUBASA в абстракции advance

Для графических ускорителей NVIDIA используется аналогичная стратегия балансировки параллельной нагрузки, однако вершины графа разбиваются на большее число групп. Каждая вершина первой группы (с самой высокой степенью) обрабатывается при помощи сетки GPU из нитей, при чём создается число нитей, равное количеству смежных ребер для каждой из обрабатываемых вершин. Каждая из вершин второй группы – при помощи блока из нитей (обычно 1024 нити), третей группы – при помощи варпа нитей (32 нити). Вершины последней группы обрабатываются группами по 32 вершины, каждая из

которых обрабатывается варпом GPU, полностью аналогично обработке векторными инструкциями вершин с низкой степенью на NEC SX-Aurora TSUBASA. В случае, если подмножество вершин последней группы – сильно разрежено, то обработка смежных ребер производится при помощи так называемого «виртуального варпа» – концепции, когда каждый варп обрабатывает фиксированное число вершин со степенью в заданном диапазоне (например от 8 до 16).

Важно отметить, что для графических ускорителей, в отличие от векторных архитектур, вершины различных групп обрабатываются в конкурентном режиме при помощи механизма CUDA-потоков (streams), что позволяет более эффективно загрузить ресурсы GPU в случае недостаточно большого числа вершин в одной из групп. Данный подход обусловлен как большим числом групп, на которые разбивается подмножество вершин графа на графическом ускорителе, так и существенно большим внутренним ресурсом параллелизма GPU.

Использование различных параллельных программных моделей. Для эффективной балансировки параллельной нагрузки важен выбор наиболее эффективной программной модели, используемой для создания параллельных программ. Для векторных архитектур используется модель OpenMP, позволяющая эффективно производить параллельную обработку циклов, возникающих при обходах вершин и ребер графов. Для архитектуры NVIDIA GPU используется программная модель CUDA, которая, при аккуратном использовании позволяет создавать значительно более высокопроизводительные программы в сравнении с другими подходами, основанными на использовании программной модели OpenACC или библиотеки Thrust.

Использование различного числа нитей и режимов балансировки параллельной нагрузки. Крайне важен выбор числа вычислительных нитей, необходимых для максимально эффективного задействования всех вычислительных ядер целевой архитектуры. Для NEC SX-Aurora TSUBASA каждое векторное ядро может эффективно обрабатывать лишь одну вычислительную нить, таким образом оптимальное число нитей для данной архитектуры – 8. Процессоры Intel (в том числе KNL) поддерживают технологию hyperthreading, позволяющую частично скрывать латентность доступа к памяти посредством смены контекстов нитей, из-за чего для данной архитектуры оптимально использовать (1–4)*x нитей, где x – число ядер. Экспериментально было получено, что для реализации алгоритмических абстракции на Intel KNL эффективнее всего запускать по две нити на одно ядро.

Для архитектуры NVIDIA GPU число нитей обычно определяется параметрами сетки, которая, в свою очередь, определяется размерами задачи – числом обрабатываемых вершин и ребер графа.

Использование векторных инструкций максимальной длины. Архитектура NEC SX-Aurora TSUBASA позволяет использовать векторные инструкции переменной длины (от 1 до 256), однако максимально эффективное использование аппаратных ресурсов достигается только при использовании векторных инструкций длины 256. Для процессоров Intel так же важно использовать векторные расширения AVX-512, позволяющие работать с векторами максимальной длины. Для NVIDIA GPU размер SIMD-инструкции (варпа) постоянен, однако важно использовать параметры программы (размер блока, сетки) позволяющие эффективно группировать вычислительные нити в варпы.

В абстракции advance векторные инструкции используются для параллельной обработки ребер графа, смежных к обрабатываемым вершинам. Описанное в разделе 3.4 разбиение вершин графа на группы по степени помогает в том числе и обеспечить эффективное использование векторных инструкций максимальной длины: для вершин с высокой степенью векторные инструкции используются для обработки *vector_length* подряд идущих смежных ребер, в то время как для вершин с низкой степенью векторные инструкции используются для обработки *vector_length* ребер, по одному от каждой из *vector_length* обрабатываемых вершин.

Улучшение шаблона доступа к памяти при доступе к информации о ребрах графа. Работу с памятью в графовых алгоритмах можно разделить на три основные группы по типу обрабатываемых объектов: работа с информацией о ребрах графа, о вершинах графа а также с прочими вспомогательными структурами данных.

Предложенный подход к реализации абстракции advance позволяет загружать из памяти информацию о ребрах графа с использованием последовательного шаблона доступа: в случае работы с вершинами с высокой и средней степенью, векторные инструкции (или варпы) обрабатывают подряд идущие ребра графа (хранящиеся последовательно в CSR-представлении графа), в то время как для вершин с низкой степенью группами обрабатываются ребра, хранящиеся в векторном расширении, что так же гарантирует последовательный характер обращений к памяти. В случае, если производится работа со взвешенным графом, то информация о направляющих вершинах и весах ребер хранится

в отдельных массивах (а НЕ в массиве структур), что позволяет организовать последовательный доступ к данным.

Улучшение шаблона доступа к памяти при доступе к информации о вершинах графа. При обработке ребер графа дополнительно необходимо считывать информацию о вершинах, смежных к обрабатываемой (направляющих вершинах), что требует выполнения значительного количества косвенных обращений к памяти, которые могут сильно снижать производительность абстракции advance. Поэтому, при обработке косвенных обращений к памяти требуется минимизировать количество промахов в кэш-память, для чего в данной работе используются оптимизации кластеризации и сегментирования.

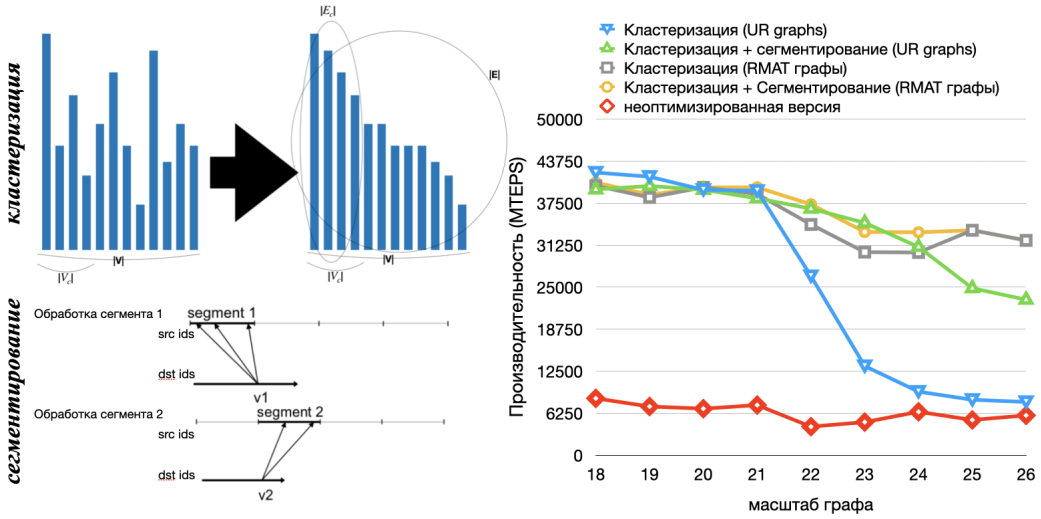

Рисунок 3.12 — Основные идеи кластеризации и сегментирования (слева), а также эффект от применения данных оптимизаций для векторной архитектуры NEC SX-Aurora TSUBASA для графов различных типов(справа)

Основная идея кластеризации – хранение в кэш-памяти только тех вершин графа, к которым производятся наиболее частые обращения (Рис. 3.12). Идея кластеризации заложена в используемый формат VectCSR: так как наиболее часто запрашиваемые вершины графа имеют самое большое количество смежных ребер (это верно для всех неориентированных и большинства ориентированных графов реального мира), то при сортировке вершин графа по убыванию степени наиболее часто запрашиваемые при косвенных адресациях данные будут находиться в смежных ячейках массива. Благодаря данному свойству формата VectCSR возможно производить префетчинг наиболее часто запрашиваемых

данных в определенный уровень иерархии кэш-памяти. Для векторных архитектур используются специализированные директивы компилятора pragma prefetch, в то время как для графических ускорителей необходимо явно размещать информацию о данных вершинах на регистрах, в разделяемой памяти и текстурном кэше. Для работы с регистрами и разделяемой памятью используются соответствующие типы данных CUDA API, в то время как для работы с текстурным кэшем – инструкции __ldg(). Можно отметить, что в новейшей анонсированной архитектуре NVIDIA GPU Ampere заявлена поддержка префетчинга данных в L2 кэш (существенно увеличенного размера до 40 Мбайт), что может позволить еще сильнее повысить эффект от данной оптимизации для будущих поколений графических ускорителей NVIDIA.

Однако, кластеризация имеет и существенный недостаток: в случае, если в графе отсутствуют вершины с высокой степенью (например равномерно-случайные графы), кластеризация становится значительно менее эффективной (Рис. 3.12). Для подобных графов выгодно использовать оптимизацию сегментирования, когда ребра графа разбиваются на отдельные группы, в каждой из которых ID направляющих вершин относятся только лишь к определенному сегменту, который может быть размещен в кэш-памяти (Рис. 3.12). Данные группы ребер обрабатываются последовательно с префетчингом информации о вершинах каждого из сегментов сегмента в кэш, с последующим объединением полученных результатов. Зачастую, для обработки больших графов применяется сочетание оптимизаций кластеризации и сегментирования.

Из данных, приведенных на Рис.3.12, следует важность применения данных оптимизаций для векторных архитектур: реализации, в которых не применяются нацеленные на повышение локальности косвенных запросов к данным оптимизации, демонстрируют существенно меньшую производительность и эффективность. В то же время одновременное применение оптимизаций кластеризации и сегментирования позволяет практически без потери производительности обрабатывать большинство типов графов даже большого масштаба.

Упаковка 4-байтных косвенно запрашиваемых данных в 8-байтные. При работе с косвенными адресациями для векторных систем с быстрой памятью важен размер косвенно запрашиваемых данных. Для всех рассматриваемых архитектур пропускная способность памяти используется значительно более эффективно при работе с 8-байтными данными (double, long long), по сравнению с 4-байтными (float, int), как показано на Рис. 3.13. Поэтому, для многих

графовых алгоритмов выгодно производить «упаковку» косвенно запрашиваемых данных: если для каждой из вершин графа запрашивается информация из нескольких различных массивов, то выгоднее объединять 4-байтные значения в 8-байтные, и после каждой косвенной адресации на чтение(запись) производить «распаковку»(«упаковку») отдельных элементов массива, осуществляемую при помощи операций приведения типов и двоичных сдвигов. Подобная оптимизация применима для многих рассматриваемых графовых алгоритмов – поиска в ширину («упаковка» уровней и номеров родительских вершин), Page Rank («упаковка» значений рангов и числа исходящих дуг из направляющих вершин) и другие.

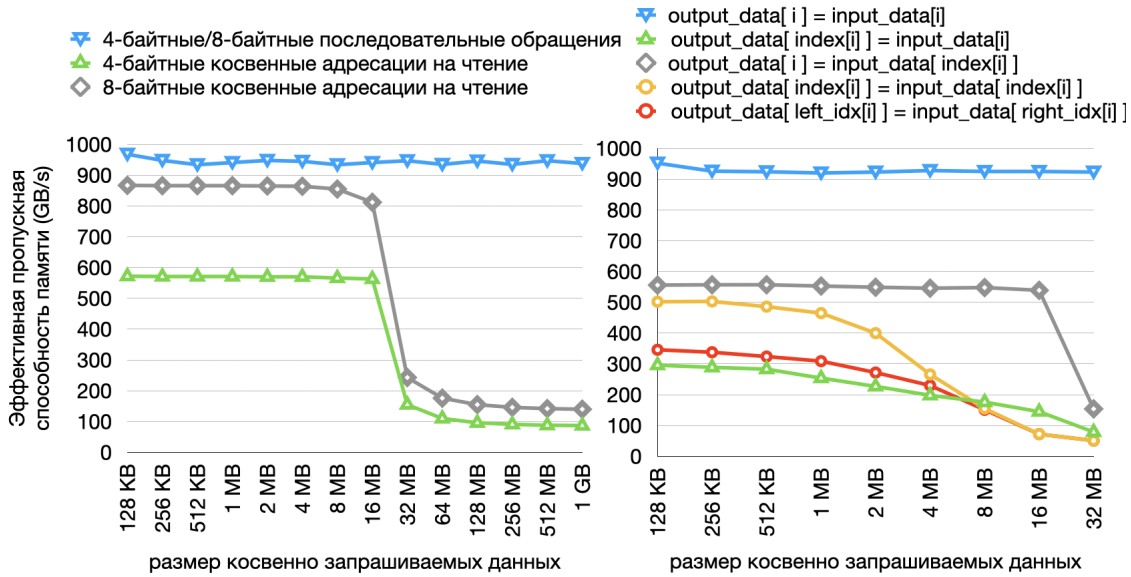

Рисунок 3.13 — Различия в пропускной способности памяти для 4-байтных и 8-байтных косвенных адресаций при работе с различными режимами чтения и записи для векторной архитектуры NEC SX-Aurora TSUBASA. Для других векторных архитектур с быстрой памятью ситуация аналогична

Использование векторных регистров для хранения промежуточных результатов обработки графа. Наконец, для векторных архитектур и графических ускорителей важно использовать регистровую память. Во многих графовых алгоритмах регистры можно использовать для аккумуляции промежуточных значений при обработке ребер в абстракции advance. К примеру, для алгоритма кратчайших путей на регистрах можно хранить минимальные дистанции до обрабатываемых векторным ядром вершин, для поиска в ширину – факт посещения, уровень и номера родителей вершин, для поиска связных компонент, и. т. д. Таким образом использование регистров для

хранение промежуточных значений позволяет существенно сократить число обращений к памяти тем самым ускорить реализацию абстракции advance.

Использование наиболее эффективного направления обхода ребер графа. Важным подходом к оптимизации является выбор более эффективного направления обхода графа – pull или push. При направлении push каждая вершина на основе своего собственного состояния обновляет состояние смежных с ней вершин (например, top-down BFS), в то время как для направления pull каждая вершина, наоборот, обновляет своё собственное состояние на основе состояния смежных с ней вершин (например bottom-up BFS). Направление обхода напрямую влияет на тип выполняемых в алгоритме косвенных адресаций – чтение или запись (gather и scatter инструкции в векторных архитектурах). Производительность косвенных адресаций на чтение и запись для рассматриваемых архитектур могут различаться, как показано на Рис. 3.13 для NEC SX-Aurora TSUBASA. Кроме того, направление обхода может влиять на скорость сходимости алгоритма (например для поиска кратчайших путей), или на на необходимость использования атомарных операций (алгоритм page rank).

Реализация алгоритмических шаблонов без использования атомарных операций. Эффективность атомарных операций для векторных архитектур существенно ниже, чем для графических ускорителей и центральных процессоров, вследствие чего в реализуемых алгоритмах необходимо по-возможности избегать их использования при реализации абстракции advance.

3.5.3 Алгоритмическая абстракция Compute

Абстракция compute может быть реализована для векторных архитектур и графических ускорителей прямолинейным образом, поскольку применяемые для каждой вершины алгоритмические шаблоны не имеют информационных зависимостей друг от друга, а значит их применение к различным вершинам входного подмножества может быть реализовано с использованием векторной и параллельной обработки данных. Абстракция compute для большинства алгоритмов не выполняет косвенных обращений к памяти (за исключением случаев, когда этого требует используемый алгоритмический шаблон), вследствие чего

её реализация крайне эффективна для всех рассматриваемых векторных архитектур с быстрой памятью.

3.5.4 Алгоритмическая абстракция Reduce

Реализация абстракции reduce аналогична реализации абстракции compute, однако после применения алгоритмических шаблонов производится редуцирование вычисленных значений при помощи операции редукции заданного типа (сложения, умножения, поиска минимума, максимума, и других). В случае векторных архитектур операция редуцирования может быть эффективно реализована на основе аккумуляции редуцируемых значений в векторных регистрах, с последующей обработкой регистров при помощи средств взаимодействия OpenMP-нитей. В то же время для графических ускорителей NVIDIA используется более сложный подход, основанный на попарном редуцировании элементов в разделяемой памяти [80], который одновременно с этим имеет и более низкую эффективность [8].

3.6 Анализ эффективности реализованных абстракций

3.6.1 Использование Roofline-модели для анализа эффективности разработанных реализаций типовых алгоритмических абстракций

Для оценки эффективности реализованных абстракций в данном диссертационном исследовании используется roofline-модель, а также следствия её применения для графовых алгоритмов. Roofline-модель [48] – это визуально-аналитическое средство анализа программ, позволяющее оценить эффективность различных вычислительных ядер в сравнении с пиковыми показателями производительности целевой архитектуры. Существует важное расширение данной модели – Cache-Aware Roofline Model (CARM) [49], в которой во внимание принимаются особенности иерархии памяти целевой архитектуры.

При построении Roofline-модели наиболее важными характеристиками работы программы являются: (1) число выполняемых программой арифметических операций и (2) количество байт данных, запрошенных программой из различных уровней подсистемы памяти. Для графовых алгоритмов, в которых зачастую отсутствуют операции с плавающей точкой, в данной работе для построения Roofline-модели вместо числа арифметических операций используется характеристика ops_per_byte, равная максимуму из числа целочисленных операций и операций с плавающей точкой, выполняемых программой.

Для графических ускорителей значения всех необходимых для построения Roofline-модели характеристик несложно вычислить на основе значений аппаратных счетчиков, доступных через средство профилировки nvprof: для подсчета числа выполненных операций различного типа есть отдельные счетчики, в то время как объем загруженных из памяти данных может быть вычислен на основании числа транзакций к каждому из уровней подсистемы памяти. Для архитектуры NEC SX-Aurora TSUBASA количество операций можно получить из утилиты ftrace, а число запрошенных из подсистемы памяти данных – на основе трассы обращений программы к памяти, а также процент попаданий в LLC кэш.

На Рис. 3.14 приведены сгенерированные описанным образом roofline-модели для архитектур NVIDIA GPU и NEC SX-Aurora TSUBASA, на которых отмечены точки, соответствующие разработанным реализациям типовых абстракций. Положение точек на Рис. 3.14 значительно зависит от того, какие именно типовые шаблоны используются в данных абстракциях, иными словами – для реализации какого именно графового алгоритма используются данные абстракции.Для построения модели на Рис. 3.14 использовались абстракции, использующие типовой шаблон алгоритма поиска в ширину Беллмана-Форда, приведенный в разделе 3.2 на Рис. 3.6 (слева).

Модели, аналогичные приведенным на Рис. 3.14, можно построить для всех исследуемых в работе вычислительных архитектур и графовых алгоритмов. Анализируя (1) положение точек на roofline-модели, соответствующих реализациями абстракций для различных архитектур и графовых алгоритмов, а также (2) свойства различных исследуемых графовых алгоритмов, можно сделать следующие выводы:

– значение ops_per_byte для всех алгоритмических абстракций всех рассмотренных графовых алгоритмов находится в интервале от 0.01 до

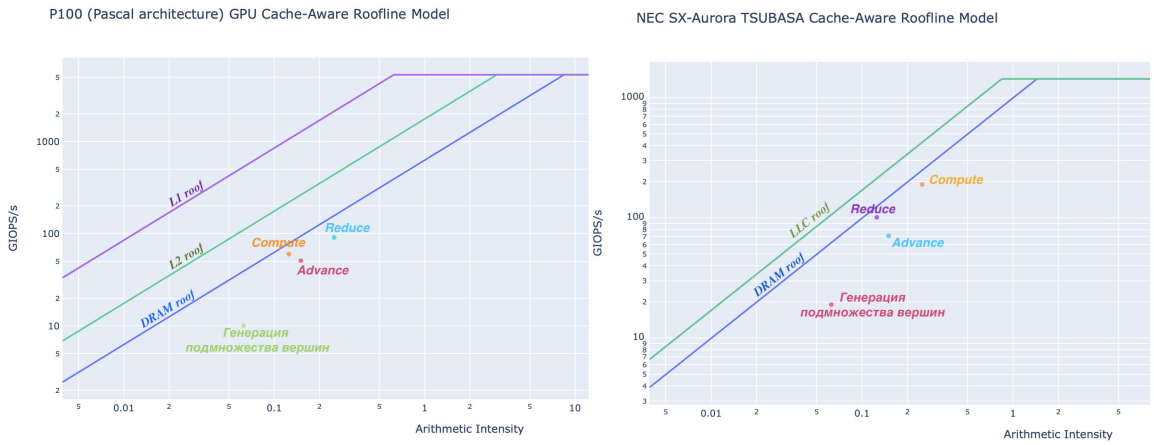

Рисунок 3.14 — Roofline-модели для архитектур NVIDIA P100 GPU (слева) и NEC SX-Aurora TSUBASA (справа), построенные для реализованных абстракций advance, compute, reduce и генерации подмножества вершин

0.25, а значит основной аппаратной характеристикой, ограничивающей производительность реализованных абстракций является пропускная способность различных компонент подсистемы памяти целевой архитектуры.

– реализации выделенных типовых графовых абстракций используют комбинацию из двух типов шаблонов доступа к памяти – последовательных и косвенных обращений на чтение и запись. Учитывая используемые оптимизации, нацеленные на повышение локальности доступа к данным при работе с графовыми структурами данных, точки на roofline-модели для разработанных реализаций должны располагаться вокруг DRAM roof, характеризующего теоретическую пиковую пропускную способность памяти.

Исходя из данных фактов, эффективность(E) реализованных абстракций можно оценить на основе используемой теоретической пропускной способности подсистемы памяти. Для этого необходимо вычислить используемую реализацией пропускную способность памяти (EBW) согласно следующей формуле:

$EBW = \frac{bytes\_requested}{T}$ – эффективная(используемая) пропускная способность, где T – время работы исследуемой абстракции, а $bytes\_requested$ – объем запрошенных из подсистемы памяти данных;

$E = 100 * \frac{EBW}{TB}$ – эффективность, где TB – теоретическая пиковая пропускная способность памяти целевой архитектуры.

Таблица 6 — Сравнительная эффективность реализации различных компонент абстракции, отвечающей за генерацию подмножества вершин графа, для архитектур NEC SX-Aurora TSUBASA и NVIDIA GPU. Абстракция применяется для RMAT графа с числом вершин 2^{24}

алгоритм	SX-Aurora, EBW(Гб/с)	SX-Aurora, E(%)	P100 GPU, EBW(Гб/с)	P100 GPU, E(%)
генерация массива входных данных	667 Гб/с	55%	411 Гб/с	57%
условное копирование (или параллельная префиксная сумма для GPU)	261 Гб/с	21%	229 Гб/с	31%
вся абстракция целиком	261 Гб/с	21%	294 Гб/с	40%
STREAM бенчмарк	983 Гб/с	81%	560	77%

Выделенные в ходе данной работы абстракции были реализованы для следующих архитектур: NEC SX-Aurora TSUBASA, NVIDIA GPU (P100 и V100), Intel KNL, а также многоядерных центральных процессоров IBM Power 8. Наиболее производительные реализации были получены для следующих систем с быстрой памятью: векторных процессоров NEC SX-Aurora TSUBASA и графических ускорителей NVIDIA P100 и V100, для которых далее будет подробно рассмотрена эффективность реализации каждой из абстракций на основе величин EBW и E.

3.6.2 Анализ эффективности абстракции «генерация подмножества вершин»

Эффективность абстракции, отвечающей за генерацию подмножества вершин графа, можно оценить, анализируя эффективность двух её основных компонент: генерации массива флагов и параллельного условного копирования. Оценки эффективности для различных составляющих компонент данной абстракции приведены в Табл. 6. Для сравнения, в данной таблице также приведены аналогичные оценки для бенчмарка STREAM [67], отражающего реальную скорость последовательного доступа к памяти для исследуемых архитектур.

Значение EBW (и, как следствие, эффективности E) для этапа генерации массива входных данных ожидаемо сопоставима с бенчмарком STREAM для всех рассматриваемых архитектур, в силу используемого на этапе генерации входных данных последовательного шаблона доступа к памяти. В то же время, для архитектуры NEC SX-Aurora TSUBASA эффективность этапа условного копирования значительно ниже (по сравнению со STREAM и пиковой пропускной способностью памяти) в силу наличия косвенных адресаций на запись и чтение и запись, возникающих при работе с временными буферами. Аналогично, для архитектуры NVIDIA GPU эффективность этапа условного копирования также значительно ниже в силу использования операции вычисления параллельной префиксной суммы.

3.6.3 Анализ эффективности абстракции advance

Эффективность абстракции advance значительно зависит от: (1) типа входного графа и (2) свойств подмножества вершин, с которыми работает данная реализация и (3) структуры типового алгоритмического шаблона, который в свою очередь зависит от реализуемого абстракцией графового алгоритма. Привести столь большое число вариантов – крайне непросто, поэтому в Табл. 7 и 8 приведены оценки эффективности абстракции advance, полученные для различных типов входных графов и типового алгоритмического шаблона, соответствующего алгоритму поиска кратчайших путей Дейкстры (процесса релаксации ребра). Типовые алгоритмические шаблоны данного алгоритма будут также использованы и для оценки двух других абстракций – compute и reduce. При этом для построения Табл. 7 во входное подмножестве вершин включены все вершины графа, в то время для Табл. 8 – каждая третья вершина графа (примерно 33% от общего числа вершин).

Таблица 7 — Сравнительная эффективность абстракции advance для архитектур NEC SX-Aurora TSUBASA и NVIDIA GPU. Все вершины и ребра графа участвуют в вычислениях

граф	SX-Aurora, EBW (Гб/с)	SX-Aurora, E (%)	P100 GPU, EBW (Гб/с)	P100 GPU, E (%)
RMAT(число вершин 2^{20})	536 Гб/с	44%	417 Гб/с	57%
RMAT(число вершин 2^{24})	589 Гб/с	49%	405 Гб/с	56%
liveJournal	458 Гб/с	38%	187 Гб/с	25%
twitter	340 Гб/с	28%	186 Гб/с	25%
wiki-ru	437 Гб/с	36%	341 Гб/с	47%

Таблица 8 — Сравнительная эффективность абстракции advance для архитектур NEC SX-Aurora TSUBASA и NVIDIA GPU. Приблизительно 33% вершин графа (каждая третья) участвуют в вычислениях

граф	SX-Aurora, EBW (Гб/с)	SX-Aurora, E (%)	P100 GPU, EBW (Гб/с)	P100 GPU, E (%)
RMAT(число вершин 2^{20})	271 Гб/с	22%	319 Гб/с	44%
RMAT(число вершин 2^{24})	312 Гб/с	26%	763 Гб/с	105%
liveJournal	141 Гб/с	11%	144 Гб/с	20%
twitter	243 Гб/с	20%	300 Гб/с	42%
wiki-ru	181 Гб/с	15%	299 Гб/с	42%

3.6.4 Анализ эффективности абстракции compute

Эффективность абстракции compute для различных архитектур и типов входных графов приведена в Табл. 9, из которой можно сделать следующие выводы. Во-первых, эффективность абстракции compute сопоставима с эффективностью бенчмарка STREAM, характеризующего скорость последовательного доступа к памяти. Во-вторых, для архитектуры NVIDIA GPU эффективность абстракции compute значительно снижена для графов небольших размеров (имеющих до 16 миллионов вершин), что обусловлено ограниченным ресурсом параллелизма данной абстракции – не более $O|V|$ параллельных операций, по

Таблица 9 — Сравнительная эффективность абстракции compute для архитектур NEC SX-Aurora TSUBASA и NVIDIA GPU

граф	число вершин графа	SX-Aurora, используемая пропускная способность (Гб/с)	SX-Aurora, Эффективность (%)	P100 GPU, используемая пропускная способность (Гб/с)	P100 GPU, Эффективность (%)
rmat (число вершин 2^{20})	$1 * 10^6$	639 Гб/с	53%	167 Гб/с	23%
wiki-ru	$2.8 * 10^6$	765 Гб/с	63%	304 Гб/с	42%
rmat (число вершин 2^{22})	$4 * 10^6$	764 Гб/с	63%	353 Гб/с	49%
rmat (число вершин 2^{24})	$16 * 10^6$	886 Гб/с	73%	553 Гб/с	76%
twitter	$32.3 * 10^6$	879 Гб/с	73%	523 Гб/с	72%
STREAM бенчмарк	-	983 Гб/с	81%	560 Гб/с	77%

сравнению с $O|E|$ для абстракции advance (в случае обработки всех вершин и ребер графа).

3.6.5 Анализ эффективности абстракции reduce

Эффективность последней абстракции – reduce, приведена в Табл. 10. На основании приведенных а таблице данных можно отметить в два раза меньшую (по сравнению с бенчмарком STREAM) эффективность данной абстракции для архитектуры NVIDIA GPU, что обусловлено значительными накладными расходами на редуцирование значений, полученных каждой из нитей, в силу использования алгоритма сдваиваний и разделяемой памяти. В то же время для векторных архитектур можно наблюдать примерно идентичную эффективность(по сравнению с бенчмарком STREAM), что достигается благодаря аккумуляции промежуточных результатов редукции на векторных регистрах, производимой без накладных расходов. Зависимость эффективности абстракции reduce от размеров входного графа полностью аналогична описанной ранее зависимости для compute.

Таблица 10 — Сравнительная эффективность абстракции reduce для архитектур NEC SX-Aurora TSUBASA и NVIDIA GPU

граф	число вершин графа	SX-Aurora, используемая пропускная способность (Гб/с)	SX-Aurora, Эффективность (%)	P100 GPU, используемая пропускная способность (Гб/с)	P100 GPU, Эффективность (%)
rmat (число вершин 2^{20})	$1 * 10^6$	660 Гб/с	55%	57 Гб/с	7%
rmat (число вершин 2^{24})	$16 * 10^6$	817 Гб/с	68%	231 Гб/с	32%
twitter	$32.3 * 10^6$	886 Гб/с	73%	253 Гб/с	35%
STREAM бенчмарк	-	983 Гб/с	81%	560 Гб/с	77%

3.6.6 Исследование динамических характеристик реализованных абстракций

С целью более детального исследования эффективности, а также других вычислительных свойств реализованных абстракций, необходимо исследовать динамические характеристики работы отвечающих им реализаций для различных рассматриваемых архитектур. Динамическими характеристиками параллельной программы будем называть такие показатели работы параллельной программы, которые характеризуют процесс её выполнения на целевой архитектуре: количество обращений к памяти, выполняемых операций различных типов, промахов в кэш-памяти и другие. Наиболее широкий набор динамических характеристик доступен для архитектуры NVIDIA GPU с использованием средства nvprof, позволяющего анализировать значения большого числа аппаратных датчиков, отвечающих разнообразным динамическим характеристикам программы. Значения собранных динамических характеристик с использованием средства nvprof для реализованного набора абстракций приведены в Табл. 11.

Приведенные в Табл. 11 характеристики отражают: (1) используемую пропускную способность различных уровней иерархии памяти – метрики, оканчивающиеся на «throughput», (2) эффективность транзакций к памяти (gld_efficiency и gst_efficiency), (3) эффективность потока управления варпа (warp_execution_efficiency), и (4) эффективность загрузки GPU (achieved_occupancy). Важно отметить, что значения метрик из группы (1) не обязательно в точности соотносится с подсчитанными в предыдущих раз-

Таблица 11 — Динамические характеристики реализованных абстракций для архитектуры NVIDIA GPU

метрика	advance вершины с высокой степенью	advance вершины со средней степенью	advance вершины с низкой степенью	compute	reduce	генерация фронта вершин
gld_throughput	567 Гб/с	458 Гб/с	487 Гб/с	0.0 б/с	133 Гб/с	460 Гб/с
gst_throughput	0.0 байт/с	0.0 байт/с	0.0 байт/с	550 Гбайт/с	0.0 байт/с	94 Гб/с
gld_efficiency	47%	45%	37%	0.0%	100%	99%
gst_throughput	0%	0%	0%	100%	0.0%	97%
dram_read_throughput	321 Гб/с	286 Гб/с	288 Гб/с	56 Мб/с	133 Гб/с	271 Гб/с
dram_write_throughput	665 Мб/с	646 Мб/с	5.9 Гб/с	548 Гб/с	3 Гб/с	91 Гб/с
l2_read_throughput	455 Гб/с	424 Гб/с	430 Гб/с	26 Мб/с	134 Гб/с	345 Гб/с
tex_throughput	593 Гб/с	441 Гб/с	560 Гб/с	0.0б/с	133 Гб/с	340 Гб/с
warp_execution_efficiency	99%	45%	83%	100%	99%	98%
achieved_occupancy	0.88	0.94	0.95	0.79	0.82	0.72

делах значениями EBW, так как значения данных метрик вычисляются (а) для каждого из уровней иерархии памяти отдельно и (б) на основе общего объема данных, загруженных из подсистемы памяти при помощи механизма транзакций, который при неэффективном шаблоне доступа может значительно превышать объем требующихся программе данных (как было рассказано в разделе 2.4).

С точки зрения трех выделенных требований к создаваемым программам для векторных систем (раздел 1.2), метрики из групп (1) и (2) соответствуют характеристикам локальности работы с данными, метрики из группы (3) – векторной обработке данных, а метрики из группы (4) – эффективному использованию массивного параллелизма целевых архитектур. Таким образом, приведенные в Табл. 11 данные доказывают высокую степень соответствия реализованных абстракций целевым векторными архитектурам с быстрой памятью.

3.7 Метод создания эффективных реализаций графовых алгоритмов для векторных систем

Описанный в работе подход к определению типовых алгоритмических абстракций позволяет предложить метод создания эффективных реализаций

графовых алгоритмов для векторных систем с быстрой памятью. Согласно данному методу, для того, чтобы решить поставленную графовую задачу, необходимо:

1. выбрать итеративный алгоритм решения данной задачи, который возможно представить в виде комбинации выделенного набора алгоритмических абстракций;
2. представить выбранный алгоритм в виде комбинации описанных алгоритмических абстракций;
3. использовать реализованные абстракции вычислений и данных для создания реализации выбранного алгоритма;
4. выбрать наиболее производительную из векторных архитектур с быстрой памятью для решения данной задачи на определенных входных данных.

3.8 Выводы главы

В данной главе был проведен анализ репрезентативной группы итеративных графовых алгоритмов, позволяющих решать широкий класс графовых задач. Проведенный анализ показал, что каждый из исследуемых графовых алгоритмов состоит не более чем из четырех типовых алгоритмических структур, а также двух абстракций данных, каждая из которых может быть эффективно реализована на векторных системах с быстрой памятью.

Для каждой из выделенных абстракций были детально описаны подходы к её реализации и оптимизации: выбор векторно-ориентированного формата хранения графа, балансировка параллельной нагрузки между векторными ядрами и векторными элементами, оптимизации, нацеленные на повышение локальности работы с данными и другие. Было показано, что данные подходы практически идентичны для векторных систем и графических ускорителей в силу описанной в главе 2 взаимосвязи между данными классами архитектур. Так же в данной главе был предложен подход к оценке эффективности реализованных абстракций, основанный на roofline-модели. На основе анализа предложенных метрик эффективности на основе roofline-модели и динамических характеристик была показана высокая степень соответствия реа-

лизованных абстракций свойствам целевых векторных архитектур с быстрой памятью.

Наличие типовых алгоритмических структур в каждом из исследуемых графовых алгоритмов вместе с возможностью их эффективной реализации для векторных систем позволило предложить метод создания эффективных реализаций графовых алгоритмов. Предложенные в данной главе алгоритмические абстракции и абстракции данных естественным образом могут служить основой для разработки программного комплекса, который позволит создавать эффективные архитектурно-независимые реализации графовых алгоритмов для рассматриваемых в работе векторных архитектур с быстрой памятью. Данный программный комплекс будет подробно описан в следующей главе диссертации.

Глава 4. Программный комплекс для создания эффективных архитектурно-независимых реализаций графовых алгоритмов

Из-за значительной сложности создания эффективных реализаций графовых алгоритмов для любых современных вычислительных систем возникает необходимость в разработке графовых фреймворков – программных сред для решения графовых задач. Графовые фреймворки включают в себя набор вычислительных функций и структур данных, используя которые пользователи могут реализовывать различные графовые алгоритмы. При этом пользователь самостоятельно определяет лишь несложные последовательные фрагменты кода, описывающие базовую вычислительную логику графовых алгоритмов. Графовые фреймворки позволяют скрывать от пользователя сложные вопросы микроархитектурной оптимизации, выбора форматов представления графов и вспомогательных данных, реализации эффективного распараллеливания и другие, что позволяет пользователям быстро создавать эффективные и высокопроизводительные реализации графовых алгоритмов.

Выделенные в предыдущей главе абстракции вычислений и данных могут быть естественным образом использованы в качестве основы для графового фреймворка, поскольку данные абстракции позволяют описать достаточно широкий класс итеративных графовых алгоритмов, как перечисленных в начале предыдущей главы, так и многих других. Тот факт, что данные абстракции могут быть в равной степени эффективно реализованы и для векторных процессоров, и для графических ускорителей NVIDIA, и для многоядерных центральных процессоров, позволяет реализовать архитектурно-независимый фреймворк. Однако, важно в начале прояснить вопрос – действительно ли необходима разработка подобного программного комплекса?

4.1 Актуальность разработки архитектурно-независимого фреймворка для векторных систем с быстрой памятью

На данный момент разработано немало готовых фреймворков для реализации различных графовых задач. Для многоядерных центральных процес-

соров существуют широко известные фреймворки, такие как Ligra, Galois и Cagra, в то время как для графических ускорителей разработаны фреймворки Gunrock, CuSHA, Medusa, Enterprise и многие другие. Однако существующие фреймворки обладают тремя принципиальными недостатками. Во-первых, для определенных классов архитектур (например, векторных систем) до сих пор не предложено эффективных графовых фреймворков, в то время как их разработка крайне актуальна, поскольку системы данного класса, как показано в проведенном диссертационном исследовании, позволяют значительно ускорять решение различных классов графовых задач. Во-вторых, существующие фреймворки для многоядерных центральных процессоров и графических ускорителей имеют значительный потенциал для оптимизации. Так, многие фреймворки не используют подходы к предобработке графов, позволяющие значительно повысить локальности работы с данными и, тем самым, повысить эффективность использования аппаратных ресурсов целевых архитектур. Кроме того, фреймворки для многоядерных центральных процессоров далеко не всегда учитывают архитектурные особенности современных процессоров, в том числе IBM Power или ARM, которые все чаще используются для решения вычислительноемких задач. В-третьих, различные существующие фреймворки принципиально отличаются как подходами к реализации заложенных в них графовых абстракций, так и принципами построения их программного интерфейса, что не позволяет эффективно переносить реализации на основе данных фреймворков между различными целевыми архитектурами.

Наличие указанных принципиальных недостатков позволяет говорить о том, что вопрос разработки эффективных и удобных графовых фреймворков далек от решения. Поэтому актуальной представляется разработка единого архитектурно-независимого фреймворка, направленного на быструю и эффективную реализацию широкого класса графовых задач, с акцентом на векторные системы с быстрой памятью. Подобное решение будет обладать тремя чрезвычайно важными характеристиками.

– Эффективность: создаваемые на основе предложенного в данной работе фреймворка реализации графовых алгоритмов будут обладать высокой производительностью за счет использования во фреймворке оптимизированных вычислительных абстракций для каждой из целевых архитектур.

– Продуктивность: процесс разработки новых эффективных графовых реализаций будет заметно упрощен за счет предоставления программной среды, позволяющей решать за пользователя многие сложные вопросы, возникающие в процессе реализации графовых алгоритмов.
– Переносимость: получаемые реализации можно будет легко и быстро переносить на другие архитектуры с сохранением высокой производительности благодаря архитектурной независимости разрабатываемого фреймворка.

Далее в данной главе приведено подробное описание фреймворка VGL (Vector Graph Library) [13], разработанного в ходе выполнения проведенного исследования. Разрабатываемый фреймворк нацелен на следующие современные вычислительные системы: векторные процессоры NEC SX-Aurora TSUBASA, центральные процессоры с векторными расширениями Intel KNL, IBM Power, Intel Xeon, а также графические ускорители NVIDIA GPU архитектур Kepler, Pascal и Volta, что позволяет обеспечить переносимость разработанных на его основе реализаций графовых алгоритмов между большим числом современных вычислительных архитектур. Основные оценки производительности, эффективности и энергоэффективности разработанных на основе фреймворка реализаций графовых алгоритмов (в том числе по сравнению с существующими аналогами) будут приведены в главе 5 данной работы.

4.2 Основные абстракций VGL: описание, характеристики, реализация

В главе 3 было показано, что выделенные алгоритмические абстракции и абстракции данных позволяют описать многие итеративные графовые алгоритмы, вследствие чего фреймворк VGL, основанный на использовании данных абстракций, так же нацелен на поддержку эффективной реализации алгоритмов данного класса. Итеративные графовые алгоритмы устроены следующим образом: на каждой итерации алгоритма производится обработка (выполнение некоторых вычислительных операций) над некоторым подмножеством вершин и ребер графа. Данная схема вычислений естественным образом реализуется через четыре абстракции вычислений и две абстракции данных, описанные в

предыдущей главе, которые и положены в основу разработанного фреймворка VGL.

Двумя центральными абстракциями данных предложенного фреймворка VGL являются подмножество вершин (для краткости называемое далее фронтом) и граф. Пользователь VGL формирует различные подмножества вершин графа на основании некоторых определяемых им критериев, после чего применяет к вершинам фронта, а также, при необходимости, к их смежным ребрам, различные вычислительные операции. Для работы с абстракциям данных во фреймворк VGL включены четыре основные вычислительные абстракции, каждая из которых основана на одноименной алгоритмической абстракции из предыдущей главы: *advance*, *generate_new_frontier*, *compute* и *reduce*.

Кроме того, во фреймворк VGL дополнительно включена реализация двух вспомогательных структуры данных – массивов информации о вершинах и ребрах графа. Данные структуры позволяют хранить дополнительную информацию о графе, необходимую для работы конкретного алгоритма. Так, для реализации алгоритма поиска кратчайших путей массивы информации о вершинах могут использоваться для хранения кратчайших дистанций, а о ребрах – для хранения весов графа графа.

Далее в главе будут подробно описаны программные интерфейсы пользователя для каждой из используемых во фреймворке вычислений абстракций и данных, а также будут приведены простейшие примеры их использования для реализации различных графовых алгоритмов. Важно отметить, что каждая из абстракций фреймворка VGL использует **все** описанные в предыдущей главе подходы к реализации и оптимизации алгоритмических абстракций и абстракций данных для векторных систем с быстрой памятью, что позволяет скрыть от пользователя фреймворка особенности создания эффективных программ для систем данного класса.

4.2.1 Абстракции данных: граф

Представление графов в фреймворке VGL использует предложенный в разделе 3.4 формат VectCSR. Граф в VGL реализован при помощи C++ класса *VectCSRGraph*, предоставляющего пользователю достаточно широкую функ-

циональность: добавление и удаление вершин и ребер, сохранение и загрузка графа на диск, генерацию различного набора синтетических графов, а также базовые функции визуализации. Фреймворк VGL поддерживает хранение как ориентированных, так и неориентированных графов; для ориентированных графов для каждой вершины производится хранение как всех входящих, так и исходящих дуг, для чего в классе *VectCSRGraph* производится хранение двух графов – исходного и транспонированного к нему.

Дополнительно, во фреймворке VGL реализовано хранение сегментированного графа. Для этого реализован отдельный класс *ShardedCSRGraph*, в котором каждый из сегментов (подграфов) так же хранится в формате VectCSR. Все описанные далее абстракции могут работать с любым из описанных классов.

4.2.2 Абстракции данных: подмножество вершин

Второй абстракцией данных фреймворка VGL является подмножество вершин графа. Подмножество (фронт) вершин при помощи класса языка C++ Frontier, конструктор которого принимает на вход единственный параметр – максимальное количество вершин в данном подмножестве, либо, альтернативно, обрабатываемый граф. Реализация подмножества вершин в VGL основано на описанном ранее векторно-ориентированном формате представления подмножества вершин графа.

Представление (и, как следствие, тип) подмножества вершин может сильно отличаться в зависимости от состава вершин в нём. Так, фронт вершин может быть полностью активным, плотным или смешанным. Далее будет часто использоваться понятие «активной» вершины графа – вершины, присутствующей в обрабатываемом на текущей итерации алгоритма фронте. При работе с вычислительными примитивами, изменяющими состав фронта (*advance* и *generate_new_frontier*), во фреймворке VGL реализовано автоматическое переключение между различными типами фронтов. При этом пользователь фреймворка может изменять критерии перехода между различными состояниями как для всего фронта, так и для его отдельных групп вершин, при чём критерии могут зависеть как от абсолютного числа вершин, включенных

в фронт, так и от количества исходящих (или входящих) из (в) фронта ребер. Некоторые из данных критериев задаются в виде специализированных глобальных констант в специальном файле, другие же могут быть изменены путём вызова специальных методов класса *Frontier*. Стандартный критерий разреженности фронта выбран как наиболее эффективный на основе экспериментов с реализацией алгоритмов BFS (Breadth-First Search), SSSP (Single Source Shortest Paths), PR (Page Rank) и CC (Connected Components).

4.2.3 Вычислительные абстракции: advance

Вычислительная абстракция *advance* является основным способом обхода графа в фремворке VGL. Реализация *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 в начале 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. 4.1.

При обходе графа с использованием абстракции *advance*, как было подробно описано в разделе 3.5, обработка вершин векторными инструкциями производится двумя принципиально отличными способами: вершины с низкой степенью обрабатываются векторными инструкциями коллективно, в то время как каждая из вершин с высокой степенью 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 (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

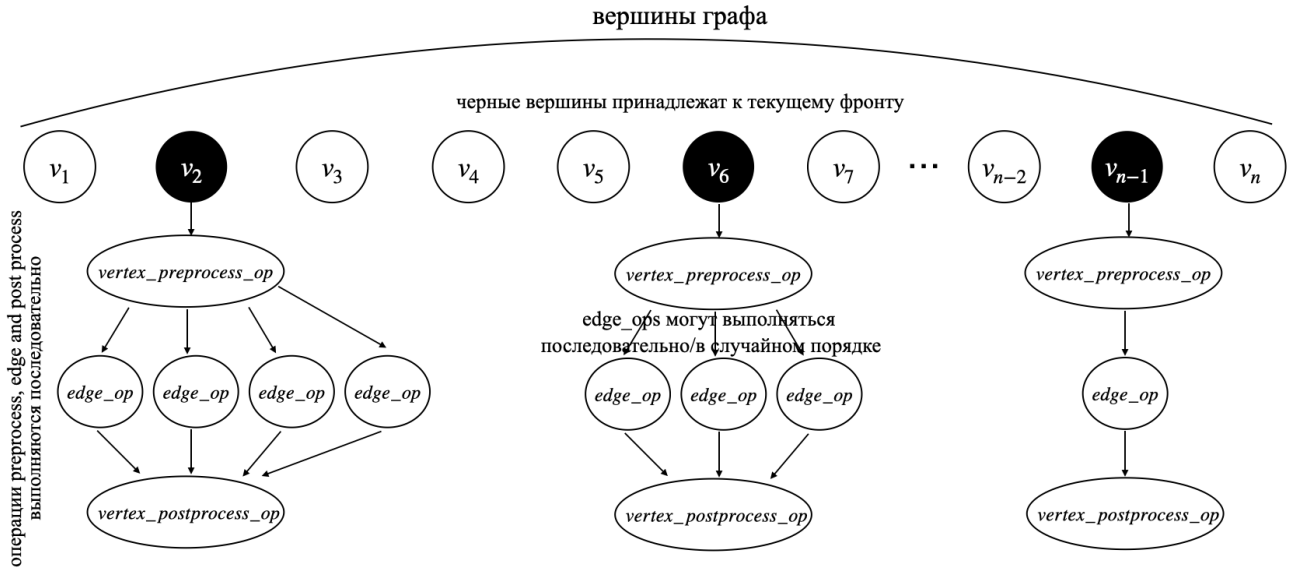

Рисунок 4.1 — Схема применения определяемых пользователем функций-обработчиков в реализации абстракции *advance* к вершинам фронта (активные вершины графа отмечены чёрным)

различные обработчики для двух классов вершин (коллективно и индивидуально обрабатываемых), вследствие чего реализация абстракции *advance* имеет прототипы, приведенные в листинге 4.1.

Листинг 4.1 — Вариант прототипа абстракции advance.

```
void advance(Graph<_TVertexValue, _TEdgeWeight> &_graph, // граф
             Frontier &_frontier, // фронт (подмножество) вершин графа
             EdgeOperation &&edge_op, // индивидуальная операция обработки ребер
             VertexPreprocessOperation &&vertex_preprocess_op, // индивидуальная опер
    ация предобработки вершин
5            VertexPostprocessOperation &&vertex_postprocess_op, // индивидуальная оп
    ерация постобработки вершин
             CollectiveEdgeOperation &&collective_edge_op, // коллективная операция оп
    ерация обработки ребер
             CollectiveVertexPreprocessOperation &&collective_vertex_preprocess_op, //
     коллективная операция предобработки вершин
             CollectiveVertexPostprocessOperation &&collective_vertex_postprocess_op,
    // коллективная операция постобработки вершин
             );
```

Каждая из операций, передаваемых в реализацию абстракции *advance*, может определяться пользователем, либо при помощи lambda-функций, либо с использованием функторов языка C++. Данные инструменты языка C++ позволяют, во-первых, получать данные от реализации фреймворка, характеризующие структуру графа и фронта вершин, и, во-вторых, захватывать произвольное количества данных пользователя (механизм lambda capture). Передаваемые в абстракцию *advance* прототипы определяемых пользователем

lambda-функций, соответствующих операциям над ребрами и вершинами графа, приведены в листинге 4.2.

Листинг 4.2 — Прототипы операций, принимаемых абстракцией advance. Для операций vertex_preprocess_op и vertex_postprocess_op используется идентичный формат VERTEX_OP.

```
auto EDGE_OP = [] (int src_id, // индекс вершины-начала исходящего ребра
                   int dst_id, // индекс вершины-конца исходящего ребра
                   int local_edge_pos, // индекс ребра среди смежных ребер текущей вершины
                   long long int global_edge_pos, // индекс ребра среди всех ребер
                   int vector_index, // индекс внутри текущей векторной инструкции
                   DelayedWrite &delayed_write // специализированная структура для отложенной векторной записи
);

auto VERTEX_OP = [] (int src_id, // индекс текущей обрабатываемой вершины
        int connections_count, // количество исходящих дуг из обрабатываемой вершины
        int vector_index, // индекс внутри текущей векторной инструкции
        DelayedWrite &delayed_write // специализированная структура для отложенной векторной записи
);
```

Несложно видеть, что почти все принимаемые данными операциями переменные характеризуют какую-либо из характеристик графа, например, индекс обрабатываемой вершин и её степень, индекс обрабатываемого ребра и другие. Как уже говорилось, данные переменные передаются в определяемые пользователем операции непосредственно из фреймворка. Однако данные операции также принимают два дополнительных параметра, позволяющих производить более эффективную векторизацию. Параметр *vector_index* характеризует индекс внутри векторной инструкции, которой выполняется данная операция, что позволяет организовать бесконфликтную работу с дополнительными структурами данных. Для примера использования параметра *vector_index* в листинге 4.3 приведена операция обработки ребра в алгоритме поиска кратчайших путей Беллмана-Форда. Векторизация данного примера будет производиться компилятором неэффективно, так как при коллективной обработке различных ребер при помощи одной векторной инструкцией неизбежно будет возникать конфликт при записи в скалярную переменную *changes*, отвечающую за наличие произошедших изменений на текущей итерации алгоритма.

Листинг 4.3 — Пример векторного конфликта по записи в алгоритме поиска кратчайших путей Беллмана-Форда.

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

```
    changes = true; // элементы векторных инструкций записывают данные в одну и ту же с
     калярную переменную, конфликт!
}
```

Исправить ситуацию как раз и помогает переменная *vector_index*, позволяющая организовать бесконфликтную запись в векторные массивы-регистры, API для работы с которыми так же приведен во фреймворке. Пример бесконфликтной реализации аналогичной операции приведен в листинге 4.4: запись о наличии изменений производится в специализированный векторный регистр, с последующей его редукцией в скаляр по завершению итерации алгоритма с использованием специализированных функций API работы с векторными регистрами.

Листинг 4.4 — Пример бесконфликтной записи в алгоритме поиска кратчайших путей Беллмана-Форда.

```
if(dst_weight > src_weight + weight)
{
  _distances[dst_id] = src_weight + weight;
   reg_changes[vector_index] = 1; // элементы векторных инструкций записывают данные в
     различные элементы векторного регистра reg_changes, НЕТ конфликта !
}
```

Специализированная структура *DelayedWrite*, передаваемая в определяемые пользователем операции, 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. *DelayedWrite* позволяет производить аккумуляцию промежуточных результатов при обходе ребер во временных структурах данных, с последующей отложенной бесконфликтной записью внутри операции *vertex_postprocess_op*.

4.2.4 Обертки вычислительной абстракции advance: gather и scatter

Важно отметить, что в случае работы с ориентированными графами для абстракции advance важно учитывать направление обхода графа, а именно для каждой вершины обрабатывать либо только исходящие, либо только входящие ребра. Для этой цели во фреймворке VGL реализованы функции-обертки

gather и *scatter*, которые вызывают абстракцию *advance* для нужного направления: *scatter* – для исходящих дуг, *gather* – для входящих. Учитывая, что входящие и исходящие дуги хранятся в предобработанных VectCSR графах (оригинальном и транспонированном к нему), в которых вершины, вообще говоря, могут быть перенумерованы, во фреймворке VGL реализована функция *change_traversal_direction*, позволяющая подготовить подмножества вершин и массивы информации о вершинах к смене направления обхода графа.

4.2.5 Вычислительные абстракции: generate_new_frontier

Абстракция *generate_new_frontier* позволяет генерировать новое подмножество (фронт) вершин графа на основании заданного пользователем условия. Данная абстракция 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 абстракции *generate_new_frontier* вместе с прототипом условия *cond* принадлежности вершины к фронту приведены в листинге 4.5.

Листинг 4.5 — Прототип абстракции generate_new_frontier и условия принадлежности вершины cond.

```
void generate_new_frontier(Graph<_TVertexValue, _TEdgeWeight> &_graph, // граф
                           FrontierNEC &_frontier, // новый фронт вершин
                           Condition &&cond); // условие принадлежности вершины к фронту

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

Абстракция *generate_new_frontier* осуществляет генерацию фронта вершин в три этапа, аналогичные описанным в предыдущей главе. На первом этапе для каждой из вершин заполняется массив флагов принадлежности вершин к фронту на основании условия *cond*, при чем условие *cond* проверяется для всех вершин графа. На втором этапе производится оценка числа вершин в создаваемом фронте, после чего принимается решение о принадлежности фронта к различным типам на основании критериев, заданных во фреймворке или пользователем. В случае, если фронт вершин (или его части) – смешанные, то для него дополнительно генерируется список индексов принадлежащих к фронту вершин.

Потенциальным недостатком данной схемы реализации примитива *generate_new_frontier* является необходимость обработки всех вершин графа для генерации массива флагов принадлежности вершин к фронту, что, вообще говоря, вычислительно не оптимально для сильно разреженных фронтов.

Для этого во фреймворке VGL реализован альтернативный вариант абстракции *advance*, который представляет из себя комбинацию вызовов абстракций *advance* и *generate_new_frontier* с одним важным отличием: в создаваемый фронт могут попасть только вершины, смежные к вершинам изначального фронта, подаваемого на вход *advance*, что значительно уменьшает объем необходимых вычислений для сильно разреженных случаев. Для реализации данной функциональности абстракция *advance* принимает два дополнительных параметра – выходной фронт вершин (передаваемый по ссылке) и условие принадлежности вершин к нему.

4.2.6 Вычислительные абстракции: compute

Абстракция *compute* применяет заданную пользователем операцию *compute_op* к каждой из вершин, принадлежащих заданному фронту. Прототипы абстракции *compute* и операции *compute_op* приведены в листинге 4.6, а схема, иллюстрирующая принцип работы примитива compute приведена на Рис. 4.2 (слева).

Листинг 4.6 — Прототип абстракции compute.

```
void compute(Graph<_TVertexValue, _TEdgeWeight> &_graph, // граф
             FrontierNEC &_frontier, // фронт вершин
             computeOperation &&compute_op); // операция, применяемая к каждой из вершин
    фронта

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

4.2.7 Вычислительные абстракции: reduce

Абстракция *reduce* выполняет редукцию значений, возвращаемых заданной пользователем операцией *reduce_op* на основе операции редукции

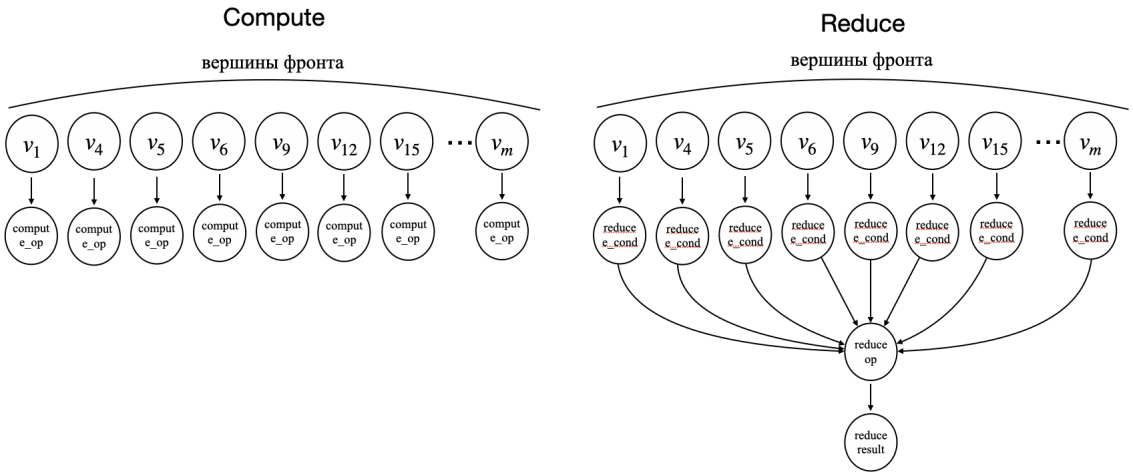

Рисунок 4.2 — Схема работы абстракций *compute* (слева) и *reduce* (справа)

reduce_type. Данный примитив имеет большое количество применений при реализации графовых алгоритмов: к примеру, данный примитив может применяться для оценки числа вершин в последующих фронтах или же вычисления вклада dangling-узлов в алгоритме page rank. Прототипы абстракции *reduce* и операции *reduce_op* приведены в листинге 4.7, а схема, иллюстрирующая принцип работы абстракции *reduce*, приведена на Рис. 4.2 (справа).

Листинг 4.7 — Прототип абстракции reduce.

```
_T GraphPrimitivesNEC::reduce(ExtendedCSRGraph<_TVertexValue, _TEdgeWeight> &_graph,
                              FrontierNEC &_frontier,
                              ReduceOperation &&reduce_op,
                              REDUCE_TYPE _reduce_type)); // операция, применяемая к к
    ждой из вершин фронта

auto reduce_OP = [] (int src_id, int connections_count, int vector_index)->_T;
```

4.3 Программная структура фреймворка VGL

В главе 3 были также подробно описаны подходы к реализации и оптимизации каждой из используемых в фреймворке VGL алгоритмических абстракций различных классов рассматриваемых в работе систем – векторных процессоров, графических ускорителей NVIDIA и центральных процессоров с векторными расширениями. В той же главе было так же показано, что состав выбранного набора абстракций не зависит от целевой архитектуры. Благодаря двум данным свойствам используемого набора абстракций, в работе был реализован

архитектурно-независимый вариант фреймворка, позволяющий создавать эффективные реализации графовых алгоритмов для следующих архитектур: NEC SX-Aurora TSUBASA, центральные процессоры с векторными расширениями Intel KNL, IBM Power, Intel Xeon, а также графические ускорители NVIDIA GPU архитектур Kepler, Pascal и Volta.

Предложенный архитектурно-независимый фреймворк имеет программную структуру, представленную на Рис. 4.3. Для трех рассматриваемых классов систем (векторные архитектуры, многоядерные центральные процессоры и графические ускорители) реализованы отдельные классы, соответствующие набору абстракций вычислений, а также двум абстракциям данных для различных архитектур. Данные классы являются производными от трех базовых классов, в которых определен единый интерфейс для каждой из абстракций вычислений и данных. Таким образом, в производных классах реализованы оптимизированные реализации абстракций и структур данных, отличающиеся для различных архитектур, однако все они имеют имеют единый интерфейс, определяемый в базовых классах.

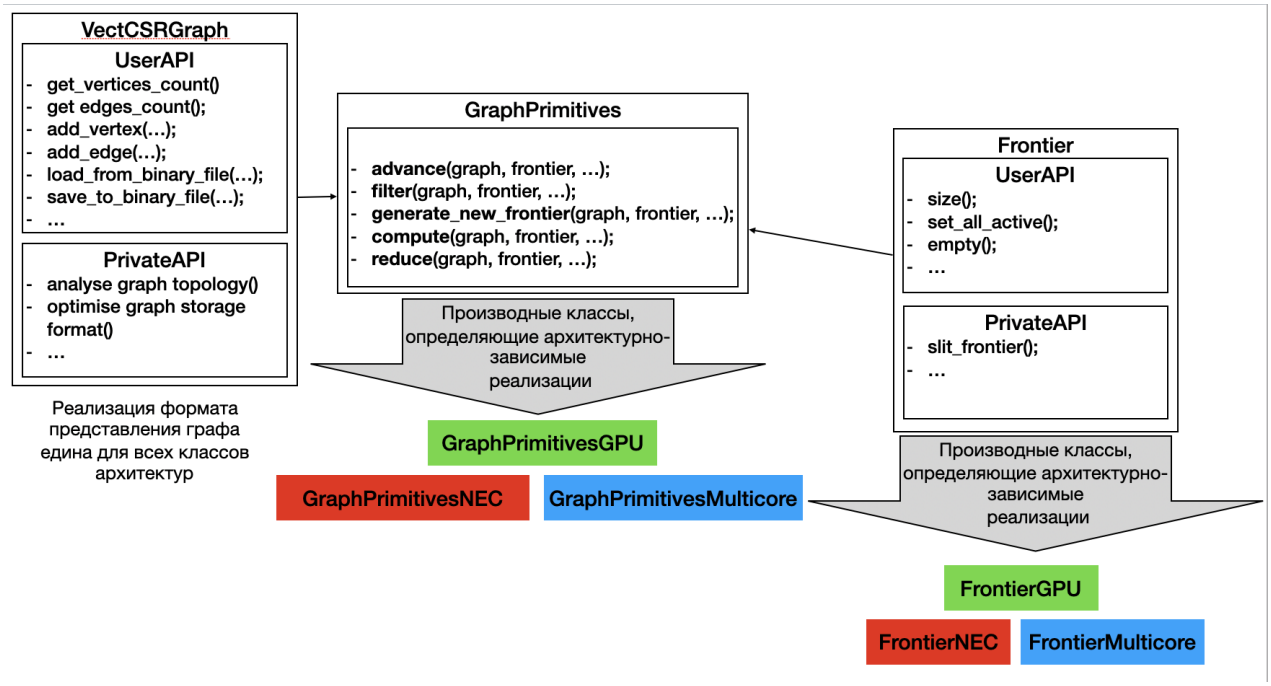

Рисунок 4.3 — Программная структура фреймворка VGL

Взаимодействие между различными элементами фреймворка организовано следующим образом: различные экземпляры объектов данных (графа и подмножеств вершин) передаются в вычислительные абстракции, имеющие

доступ ко всем внутренним структурам данных этих объектов на основе механизма дружественных классов языка C++. Пользователь фреймворка имеет крайне ограниченный доступ к внутренним структурам данных фреймворка, описывая графовые алгоритмы в терминах предложенных абстракций и некоторых дополнительных методов работы с объектами данных (public API, Рис. 4.3).

4.4 Типовые схемы использования фреймворка VGL для реализации графовых алгоритмов

Типовые схемы использования примитивов фреймворка VGL приведены на Рис. 4.4 для алгоритмов двух классов: «all-active», осуществляющих обработку всех вершин и ребер графа на каждой итерации, и «partial-active», обрабатывающих лишь определённые подмножества вершин. Для «all-active» графовых алгоритмов (Беллмана-Форда, page rank, Шиллоаха-Вышкина и других) в начале производится инициализации графа, после чего все вершины фронта помечаются как активные (участвующие в вычислениях). Затем, с помощью абстракции *compute* инициализируются начальные данные, например, изначальные дистанции для задачи поиска кратчайших путей. Далее выполняется основная вычислительная часть алгоритма – последовательность вызовов абстракции *advance* до тех пор, пока не будет выполнен некоторый критерий остановки (сходимости) алгоритма. Для «partial-active» алгоритмов в основной вычислительной части абстракция *advance* чередуется с абстракцией *generate_new_frontier*, в то время как критерий остановки зависит от числа оставшихся вершин во фронте.

Схема реализаций четырех графовых алгоритмов решения задач PR, BFS, CC и SSSP приведена на Рис. 4.5. Данная схема представляет из себя диаграмму, состояния которой описывают возможные подходы к реализации lambda-функций, описывающих основную вычислительную логику алгоритмов, а переходы между состояниями – последовательность вызова вычислительных абстракций фреймворка.

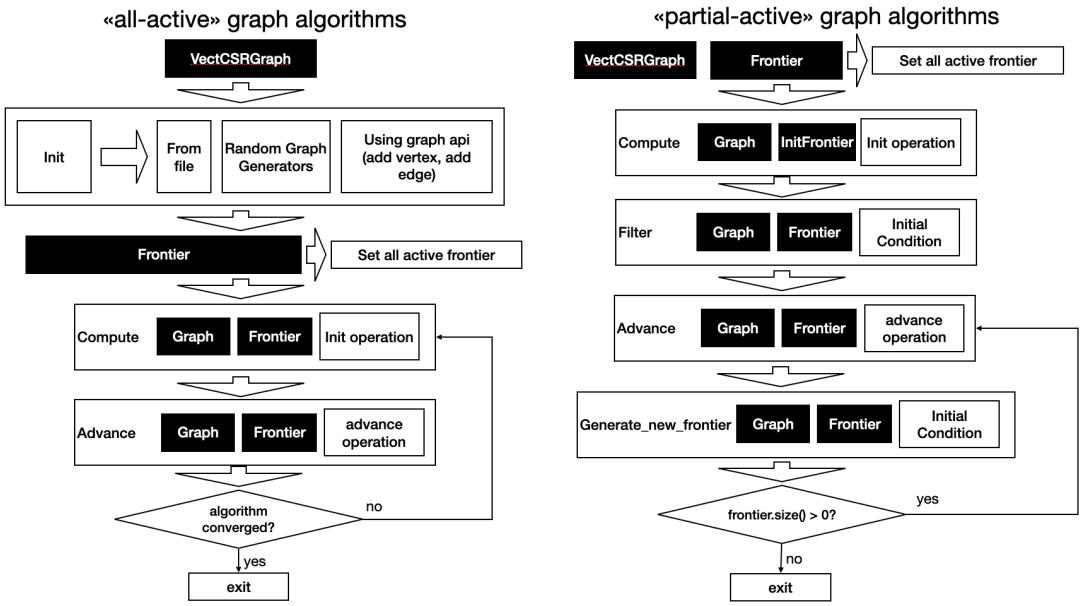

Рисунок 4.4 — Типовая схема использования API фреймворка VGL для реализации «all-active» (слева) и «partial-active» (справа) графовых алгоритмов

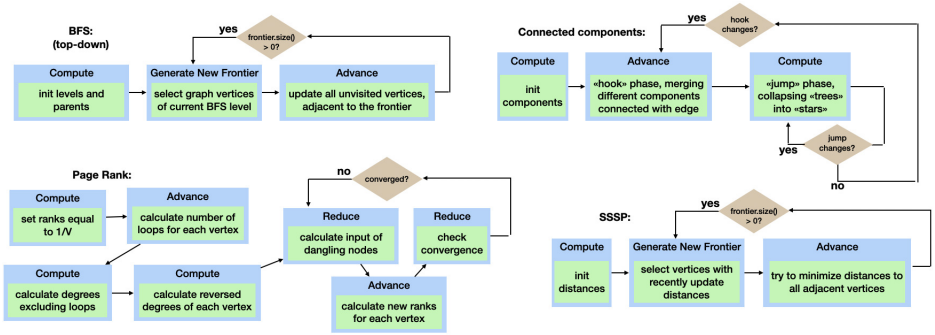

Рисунок 4.5 — Диаграмма состояний, используемая для реализации четырех графовых алгоритмов через абстракции фреймворка VGL

4.5 Пример использования фреймворка VGL для реализации графовых алгоритмов на архитектуре NEC SX-Aurora TSUBASA

Использование разработанного фреймворка VGL в данной работе рассмотрено применительно к реализации двух графовых алгоритмов – top-down поиска в ширину и Беллмана-Форда поиска кратчайших путей в графе. Данные алгоритмы были выбраны как простейшие, представляющие два обширные класса алгоритмов: «partial-active» и «all-active» соответственно. Пример реализации алгоритма поиска в ширину приведен в листинге 4.8.

Листинг 4.8 — Пример реализации алгоритма Беллмана-Форда 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; // дистанция до вершины-источника равна 0
  else
    _distances[src_id] = FLT_MAX; // дистанция до вершины-источника равна "бесконечнос
    ти"
};
graph_API.compute(_graph, frontier, init_distances); // инициализация массива дистанци
    й
int changes = 1;
while(changes) // критерий остановки - на последний итерации алгоритма не произошло ни
    каких изменений
{
  changes = 0;

  VGL_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,
    DelayedWrite &delayed_write)
  {
    float weight = outgoing_weights[global_edge_pos];
    if(_distances[dst_id] > _distances[src_id] + weight) // попытка оптимизации рассто
    яния от вершины-источника до вершины-конца каждого ребра графа
    {
      distances[dst_id] = _distances[src_id] + weight;
      reg_changes[vector_index]++; // подсчет количества изменений в массиве дистанций
    , произошедших на текущей итерации
    }
  };
  graph_API.advance(_graph, frontier, edge_op_push); // обход всех вершин и ребер граф
    а

  changes += register_sum_reduce(reg_changes); // подсчет количества произошедших изме
    нений в массиве дистанций
}
```

В листинге 4.9 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, а так же условие остановки алгоритма, зависящее от числа вершин в текущем фронте.

Листинг 4.9 — 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)
5       _levels[_source_vertex] = FIRST_LEVEL_VERTEX; // пометить вершину-источник как по
    сещенную
  else
      _levels[src_id] = UNVISITED_VERTEX; // пометить все остальные вершины как не посе
    щенные
};
graph_API.compute(_graph, frontier, init_levels);
10 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; // изначально фронту принадлежат только вершина-источник
};
graph_API.filter(_graph, frontier, on_first_level);
15
int current_level = FIRST_LEVEL_VERTEX;
while(frontier.size() > 0) // пока фронт вершин не пуст
{
  auto edge_op = [_levels, _current_level](int src_id, int dst_id, int local_edge_pos,
20                long long int global_edge_pos, int vector_index, DelayedWrite &
    delayed_write)
  {
    if(_levels[dst_id] == UNVISITED_VERTEX)
      _levels[dst_id] = _current_level + 1; // если конечная вершина ребра не посещена
    - происходит посещение вершины
  };
25
  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; // добавление посещенных на данной итерации вершин к нов
    ому фронту
30 };
  graph_API.generate_new_frontier(_graph, frontier, on_next_level);
  current_level++;
}
```

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 к памяти, и многие другие. Во-вторых, представленный API позволяет существенно сократить количество строк кода, необходимых для описания алгоритма. Так, аналогичные оптимизированные «вручную» версии приведенных в данном разделе алгоритмов для архитектуры NEC SX-Aurora TSUBASA имеют объем 500-1000 строк кода, в то время как реализации, использующие API фреймворка VGL – менее 50 строк, что позволяет существенно сократить время на разработку, отладку и оптимизацию графовых алгоритмов. Вопрос, насколько эффективность разработанных

при помощи API VGL реализаций ниже оптимизированных «вручную» аналогов рассмотрен в следующем разделе.

4.6 Сравнительная производительность реализаций на основе фреймворка с оптимизированными вручную реализациями

В данном разделе приведено сравнение производительности оптимизированных «вручную» реализаций алгоритмов поиска кратчайших путей и поиска в ширину с реализациями, разработанными на основе предложенного фреймворка VGL и описанными в предыдущем разделе. Результаты данного сравнения приведены на Рис. 4.6.

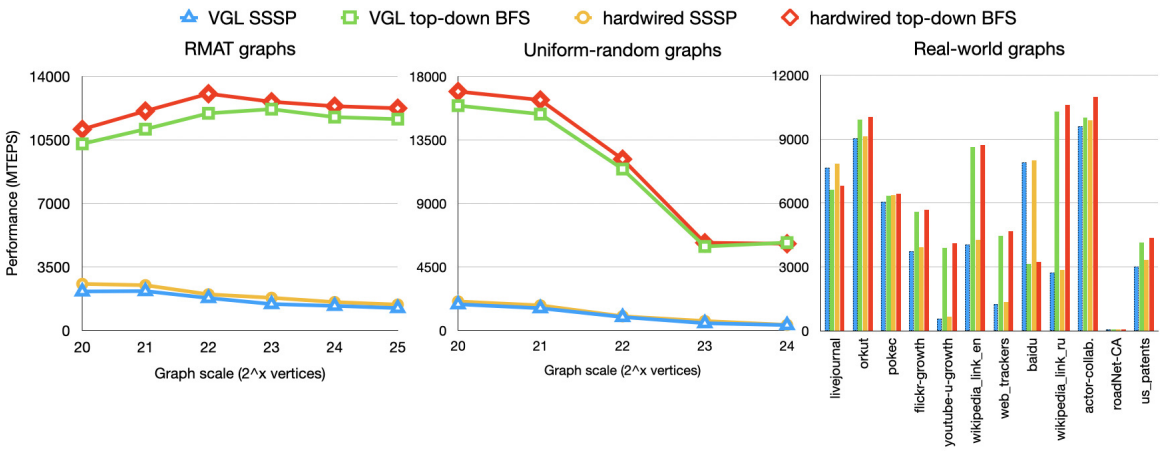

Рисунок 4.6 — Сравнение производительности оптимизированных «вручную» реализаций алгоритмов поиска в ширину и поиска кратчайших путей в графе с реализациями на основе фреймворка VGL

Сравнение приведено для синтетических RMAT [54] и равномерно-случайных [55] графов, а также для некоторых графов реального мира на основе метрики производительности графовых алгоритмов TEPS (Traversed Edges Per Second) [47]. Приведенное сравнение наглядно демонстрирует, что, во-первых, использование lambda-функций для описания графовых алгоритмов не вносит существенных задержек в скорость работы реализованных на основе фреймворка алгоритмов. Во-вторых, инкапсуляция основных оптимизаций графовых алгоритмов для векторных архитектур позволяет реализациям на основе

фреймворка VGL достигать производительности, почти идентичной производительности оптимизированных «вручную» реализаций.

Комплексное сравнение производительности реализаций на основе фреймворка VGL с существующими аналогами графовых библиотек и фреймворков для многоядерных центральных процессоров и графических ускорителей будет подробно рассмотрено в следующем разделе вместе с вопросами энергоэффективности полученных реализаций.

4.7 Выводы главы

В данной главе был подробно описан разработанный в ходе проводимого исследования архитектурно-независимый фреймворк VGL, позволяющий создавать эффективные реализации графовых алгоритмов для различных векторных систем с быстрой памятью. Были описаны основные функции программного интерфейса пользователя предложенного фреймворка, его программная архитектура, а также типовые примеры его использования для реализации различных графовых алгоритмов. Кроме того, в данной главе было показано, что реализации на основе данного фреймворка не уступают предложенным в ходе данной работы оптимизированным вручную аналогам, одновременно с тем требуя на порядок меньше исходного кода для своего создания.

Использование единого набора абстракций для каждой из рассматриваемых в работе архитектур позволяет легко переносить реализации графовых алгоритмов между различным целевыми платформами, что позволяет еще более ускорить решение графовых задач за счёт подбора максимально эффективной целевой архитектуры. Таким образом, предложенный фреймворк VGL, в отличие от существующих аналогов, позволяет одновременно обеспечить высокую продуктивность, переносимость и эффективность создаваемых реализаций графовых алгоритмов. Сравнение производительности, эффективности и энергоэффективности реализаций графовых алгоритмов на основе предложенного фреймворка VGL с существующими аналогами будет приведено в заключительной главе диссертации.

Глава 5. Анализ производительности, эффективности и энергоэффективности разработанных реализаций

5.1 Анализ производительности и сравнение с существующими библиотечными реализациями

Производительность реализаций графовых алгоритмов на основе разработанного фреймворка VGL была изучена в сравнении с производительностью реализаций на основе существующих фреймворков для многоядерных центральных процессоров и графических ускорителей NVIDIA, перечисленных в обзорной части работы. В качестве платформы для тестирования использовался сервер, оборудованный (1) 12-ядерным центральным процессором Intel(R) Xeon(R) Gold 6126 архитектуры Intel Skylake, (2) графическим ускорителем NVIDIA V100 GPU архитектуры Volta и (3) Vector Engine SX-Aurora TSUBASA Type 10B. Для центральных процессоров использовались следующие графовые библиотеки и фреймворки: Ligra, Galois, GAPBS, как наиболее производительные из существующих, имеющихся в открытом доступе. Для установки данных библиотек и фреймворков использовался компилятор GCC версии 8.3, при чём для каждого из фреймворков использовалась последняя доступная на момент написания работы версия. Для графических ускорителей NVIDIA использовались фреймворки Gunrock, cuSHA, Enterprise, а также библиотеки NVGRAPH и Lonestar GPU, которые были собраны с использованием компилятора GCC v8.3 и NVIDIA CUDA Toolkit v10.2. Для компиляции реализаций на основе фреймворка VGL использовался компилятор nc++ версии 3.0.6 от 22 мая 2020 года. Производительность реализаций была исследована для синтетических RMAT [54] и равномерно-случайных [55] графов, а также некоторых графов реального мира из коллекций [56] и [57], характеристики которых приведены в таблице 2 раздела 1.5.

Результаты сравнительного анализа производительности приведены на рисунках 5.1, 5.2, 5.3, 5.4. Данные результаты получены согласно следующей методологии: для каждой из рассмотренных графовых задач производительность реализаций на основе фреймворка VGL сравнивалась с двумя наилучшими реализациями для многоядерных центральных процессоров и графических

ускорителей NVIDIA (итого 4 сторонние реализации). Реализации на основе фреймворка VGL запускались на двух целевых платформах, позволяющих достичь наиболее высокой позволительности: графических ускорителях NVIDIA GPU V100 и векторных процессорах NEC SX-Aurora TSUBASA. С целью исключить зависимость производительности от вычислительной сложности алгоритма, сравнение производилось только между одинаковыми алгоритмами или же незначительными вариациями одного и того же алгоритма. Для наглядности реализации но основе предложенного фреймворка VGL обозначены линиями с треугольниками.

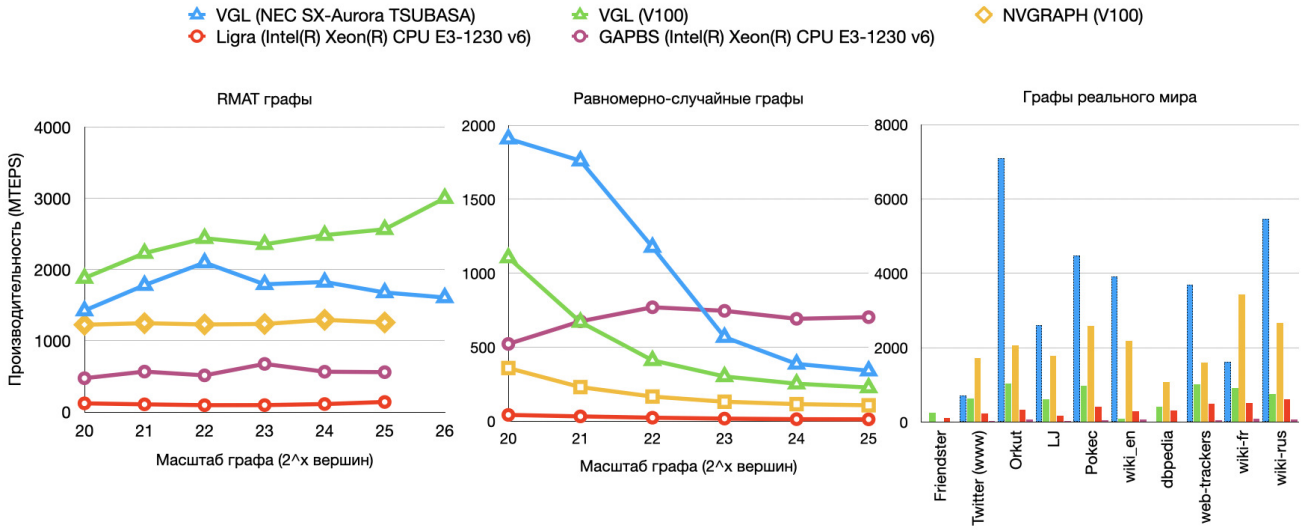

Рисунок 5.1 — Производительность реализаций алгоритма Беллмана-Форда решения задачи поиска кратчайших путей в графе по сравнению с существующими аналогами библиотек и фреймворков для multicore CPU и NVIDIA GPU

Первое, на что важно обратить внимание – реализации на основе фреймворка VGL (для NEC и GPU) обеспечивают ускорение до 14 раз по сравнению с существующими библиотечными аналогами для многоядерных процессоров. Столь существенное различие в производительности обусловлено сильно различными теоретическими значениями пропускной способности памяти для данных платформ: 90 ГБ/с для Intel Xeon против 1,2 ТБ/с для NEC SX-Aurora TSUBASA и 900 GB/s для V100 GPU. Тот факт, что разница в производительности реализаций приблизительно пропорциональна значениям теоретических пропускных способностей, подтверждает высказанный во введении тезис о значительном потенциале использования векторных систем с быстрой памятью для ускорения решения графовых задач.

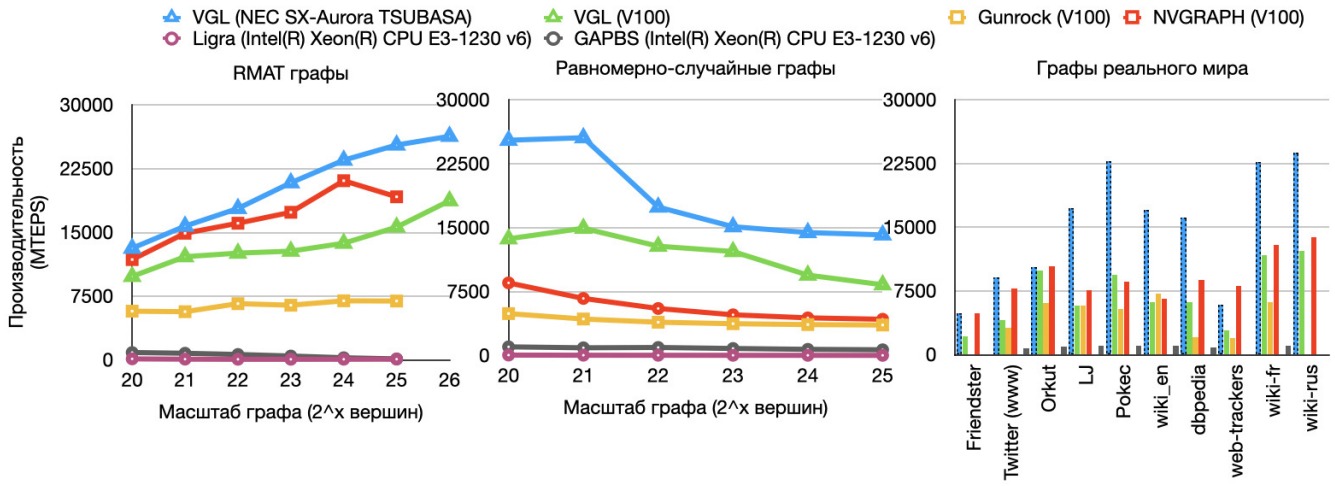

Рисунок 5.2 — Производительность реализаций алгоритма Page Rank ранжирования вершин в графе по сравнению с существующими аналогами библиотек и фреймворков для multicore CPU и NVIDIA GPU

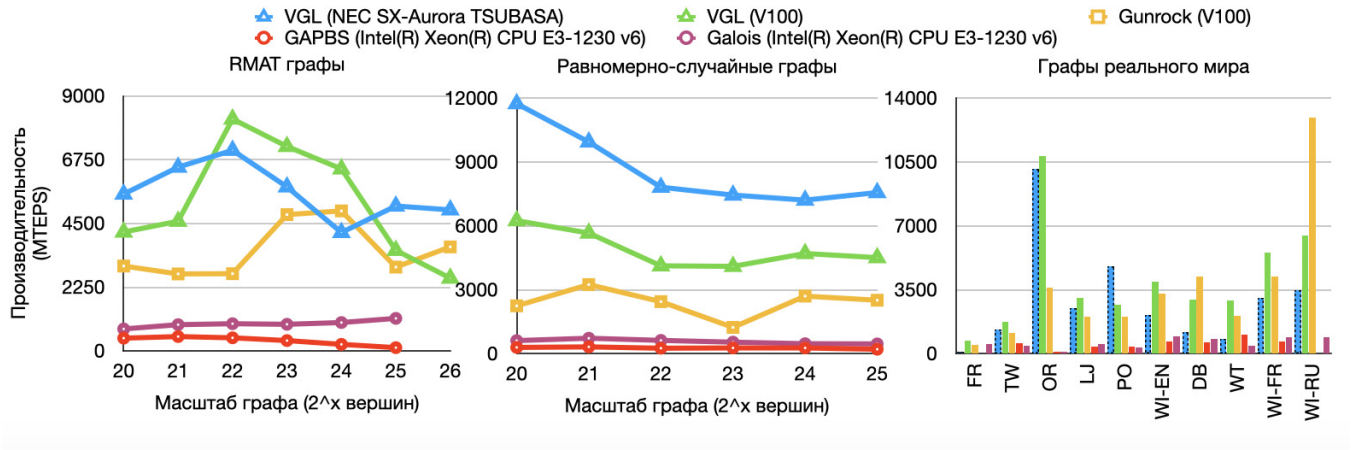

Рисунок 5.3 — Производительность реализаций алгоритма Шилоаха-Вышкина поиска связных компонент в графе по сравнению с существующими аналогами библиотек и фреймворков для multicore CPU и NVIDIA GPU

Сравнение производительности реализации на основе VGL более справедливо с существующими реализациями для графических ускорителей V100. Однако, для каждой рассмотренной графовой задачи реализации на основе VGL имеют до 3 раз лучшую производительность. При этом для различных задач и входных графов выгодно использовать VGL реализации, использующие различные целевые архитектуры – NEC SX-Aurora TSUBASA или V100 GPU. К примеру, для задачи поиска связных компонент GPU реализация для VGL демонстрирует лучшую производительность для большинства графов из-за неэффективного шаблона доступа к памяти для архитектуры SX-Aurora во

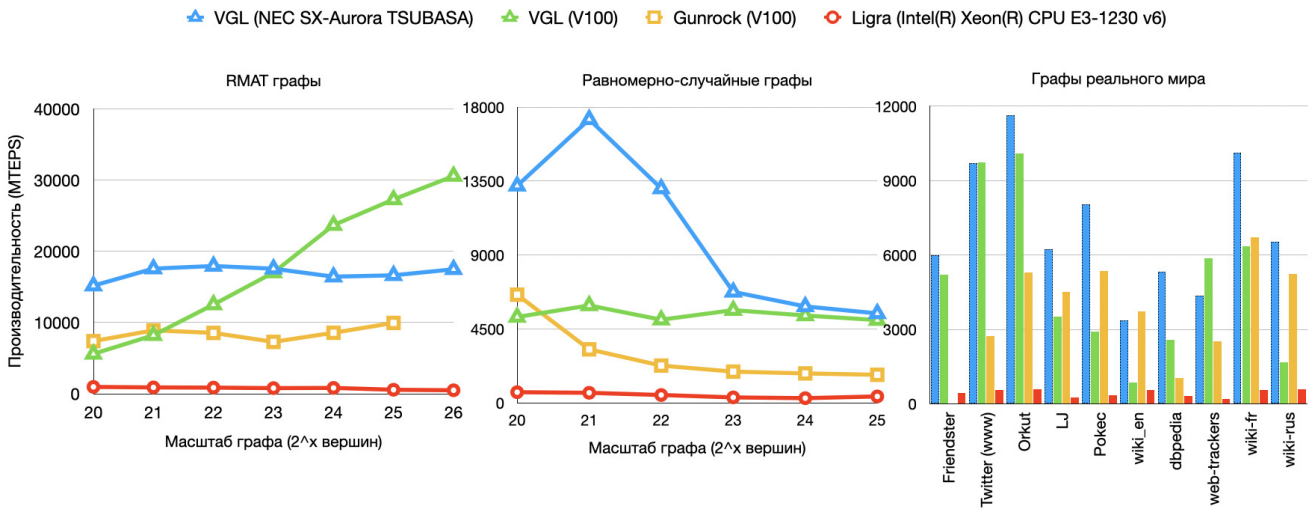

Рисунок 5.4 — Производительность реализаций алгоритма Top-Down поиска в ширину в графе по сравнению с существующими аналогами библиотек и фреймворков для multicore CPU и NVIDIA GPU

время «jump» итерации алгоритма Шилоаха-Вишкина – многоуровневых косвенных обращений к памяти. При этом благодаря тому, что фреймворк VGL архитектурно-независим, реализации графовых алгоритмов могут быть элементарным образом перенесены между данными архитектурами.

5.2 Анализ эффективности

Итак, реализации на основе предложенного фреймворка VGL эффективны с точки зрения ускорения по сравнению с существующими аналогами. Однако, так же важно исследовать эффективность данных реализаций на основе roofline-модели и её следствий, описанных в разделе 3.6 данной работы. Напомним, что каждая из рассмотренных в предыдущем разделе графовых задач обладает низкой вычислительной интенсивностью, вследствие чего эффективность каждой из задач может быть вычислена как $\frac{EBW}{TB}$, где EBW – эффективная пропускная способность исследуемой реализации, равная количеству запрошенных байт данных, деленному на время работы реализации, а TB – теоретическая пропускная способность памяти целевой платформы, равная 1,2 Тб/с для NEC SX-Aurora TSUBASA, 720 Гб/с для P100 GPU и 900 Гб/с для V100 GPU.

Таблица 12 — Оценки эффективности (через используемую пропускную способность памяти) для реализаций алгоритмов решения задачи поиска кратчайших путей (SSSP) на основе фреймворка VGL

Граф	SX-Aurora, используемая ППС, Гб/с	SX-Aurora, % от пиковой ППС	P100 GPU, используемая ППС, Гб/с	P100 GPU, % от пиковой ППС
RMAT (число вершин 2^{23})	556	46%	436	60%
RMAT (число вершин 2^{25})	532	44%	513	71%
Uniform-Random (число вершин 2^{21})	809	67%	133	18%
Twitter	307	26%	174	24%
Wiki_en	497	41%	215	29%
Pockec	647	53%	200	27%

Таблица 13 — Оценки эффективности (через используемую пропускную способность памяти) для реализаций алгоритмов решения задачи поиска в ширину (BFS) на основе фреймворка VGL

Граф	SX-Aurora, используемая ППС, Гб/с	SX-Aurora, % от пиковой ППС	P100 GPU, используемая ППС, Гб/с	P100 GPU, % от пиковой ППС
RMAT (число вершин 2^{23})	246	21%	364	50%
RMAT (число вершин 2^{25})	211	18%	451	62%
Uniform-Random (число вершин 2^{21})	334	27%	172	23%
Twitter	136	12%	154	21%
Wiki_en	261	22%	43	5%
Pockec	289	24%	138	19%

В Табл. 12 и Табл. 13 приведены значения оценок эффективности для реализаций задачи поиска кратчайших путей (SSSP) и поиска в ширину (BFS) соответственно. Предложенные реализации задачи поиска в ширину на основе фреймворка VGL относятся к классу «all-active»-алгоритмов, на каждой итерации которых все вершины и ребра графа участвуют в вычислениях, в то время как поиска в ширину – к классу «partial-active».

Из приведенных в Табл. 12 и Табл. 13 данных можно сделать следующие выводы. Для «all-active»-алгоритмов (на примере SSSP) реализации на основе VGL имеют значительно большую эффективность, что обусловлено более регулярной структурой данных алгоритмов, а также значительно большим объемом вычислений, производимых на каждой итерации алгоритмов данного типа ($O|E|$). Для «partial-active»-алгоритмов используемая пропускная способ-

ность (и, как, следствие, эффективность) несколько снижаются, в основном из-за присутствия в алгоритмах итераций, обрабатывающих сильно разреженные фронты вершин.

5.3 Анализ энергоэффективности

Помимо анализа производительности, в данной работе так же оценивается и энергоэффективность разработанных реализаций графовых алгоритмов. Наиболее интересно провести сравнение энергопотребления для трех классов систем: векторных архитектур (на примере NEC SX-Aurora TSUBSA), многоядерных центральных процессоров, а также графических ускорителей NVIDIA.

Для сравнения энергоэффективности традиционно используются две метрики: средняя потребляемая системой мощность (в ватт), а также производительность на единицу мощности (в GTEPs/ватт). Используемая для оценки энергоэффективности платформа состоит из процессора Intel Xeon, Vector Engine SX-Aurora TSUBASA, а также графического процессора NVIDIA V100, установленных на одном сервере. Для измерения потребляемой мощности всей системы, а также отдельно процессора Intel Xeon, использовались средство IPMI (интеллектуального интерфейса управления платформой). Для измерения потребляемой мощности VE SX-Aurora TSUBASA использовался инструмент Monitoring and Maintenance Manager (MMM), а для V100 GPU – NVIDIA System Management Interface (Рис. 5.5).

В начале нужно отметить, что потребляемая основной вычислительной компонентой мощность практически не отличается при работе различных графовых алгоритмов. Поэтому далее приведено детальное исследование энергоэффективности при решении задачи поиска в ширину (BFS).

На Рис. 5.5 (в центре) приведено среднее энергопотребление различных компонент системы при выполнении трех различных реализаций поиска в ширину для каждой из рассмотренных платформ: Gunrock для NVIDIA V100, Ligra для Intel Xeon Vector Host, VGL для VE NEC SX-Aurora TSUBASA. Приведенные на Рис. 5.5 данные демонстрируют, что что энергопотребление компонент, непосредственно выполняющих основную часть вычислений, существенно меньше для VE Aurora и NVIDIA GPU по сравнению с центральными

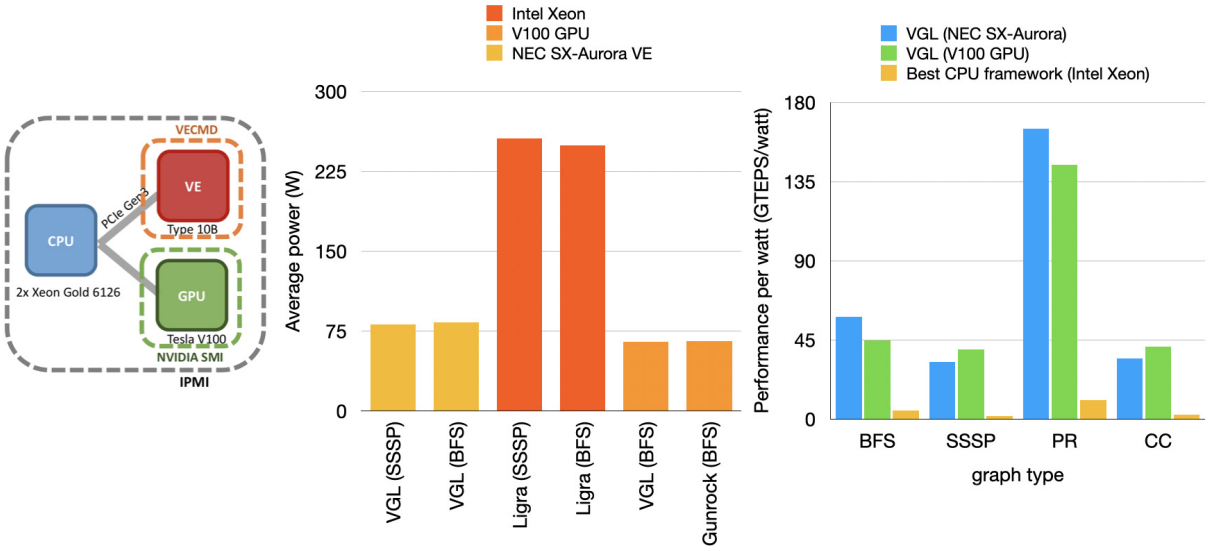

Рисунок 5.5 — Использованные средства для измерения потребляемой мощности (слева), средняя потребляемая мощность для различных рассматриваемых архитектур (в середине), производительность на единицу мощности для различных графовых задач(справа)

процессорами Intel Xeon. Так, VE Aurora при вычислениях потребляет 83 Вт (в то время как при простое – 41 Вт), GPU - 65 Вт (при простое – 39 Вт), а Intel Xeon – 255 Вт. На рисунке Рис. 5.5 (справа) приведена вторая метрика – производительность на единицу мощности для разработанных VGL реализаций (запускаемых на VE Aurora и V100 GPU) по сравнению с наиболее быстрыми реализацией для многоядерных центральных процессоров Intel Xeon различных графовых задач. Приведенные данные демонстрируют значительно более высокую энергоэффективность разработанных реализации на основе фреймворка VGL по сравнению с наиболее быстрыми существующими реализациями для многоядерных центральных процессоров. Вклад в энергоэффективность VGL реализаций вносится как за счет значительно большей производительности VGL-реализаций (до 14 раз по сравнению с аналогами для Intel Xeon), так и за счет более низкой потребляемой мощности архитектурами VE SX-Aurora TSUBASA и NVIDIA GPU (до 5 по сравнению с Intel Xeon), что делает реализации на основе VGL до 70 раз более эффективными по сравнению с реализациями для многоядерных центральных процессоров.

5.4 Выводы главы

В данной главе на основе предложенного в работе фреймворка VGL были разработаны высокопроизводительные реализации набора фундаментальных графовых алгоритмов решения различных графовых задач. Реализации на основе фреймворка VGL демонстрируют значительное ускорение по сравнению с существующими наиболее быстрыми фреймворками и библиотеками для современных многоядерных центральных процессоров и графических ускорителей NVIDIA. К примеру, реализации на основе VGL в 14 раз быстрее аналогов из библиотек Ligra, Galois и GAPBS для многоядерных центральных процессоров Intel Xeon, а также в 1,2-3 раза по сравнению с реализациями Gunrock и NVGRAPH для NVIDIA V100 GPU для большинства рассматриваемых в работе синтетических графов и графов реального мира.

Было показано, что реализации графовых алгоритмов на основе фреймворка VGL используют от 20% до 70% от пиковой теоретической пропускной способности памяти целевых архитектур благодаря применению большого числа описанных в главе 3 оптимизаций, что как раз и позволяет значительно опережать существующие аналоги.

Кроме того, реализации на основе предложенного фреймворка VGL еще и значительно более энергоэффективны по сравнению с реализациями для многоядерных центральных процессоров (до 70 раз) как за счет существенного меньшего энергопотребления основными вычислительными компонентами (Vector Engine Aurora и GPU), так и существенно более высокой производительности разработанных реализаций.

Заключение

Основные результаты работы заключаются в следующем.

1. Опираясь на принципы суперкомпьютерного кодизайна и анализ информационной структуры типовых графовых алгоритмов, в работе предложен метод создания реализаций графовых алгоритмов, использующий набор из четырех алгоритмических абстракций и двух абстракций данных. Показано, что с помощью этого набора можно выразить все рассмотренные графовые алгоритмы, и обоснована возможность их эффективной реализации на современных векторных архитектурах.
2. Предложенный набор алгоритмических абстракций и абстракций данных составил основу для проектирования и реализации архитектурно-независимого программного комплекса VGL (графовый фреймворк) для векторных систем с быстрой памятью, позволяющего объединить сразу три важные характеристики в разработке программного обеспечения: эффективность, продуктивность и переносимость.
3. На основе разработанного фреймворка были созданы эффективные реализации набора фундаментальных графовых алгоритмов для современных векторных архитектур с быстрой памятью – NEC SX–Aurora TSUBASA, NVIDIA GPU, Intel KNL, демонстрирующие существенно большую производительность (как правило, в разы) и энергоэффективность по сравнению с существующими библиотечными аналогами для многоядерных центральных процессоров и графических ускорителей NVIDIA.

В заключение автор выражает благодарность и большую признательность научному руководителю Воеводину Вл. В. за поддержку, помощь, обсуждение результатов и научное руководство.

Список сокращений и условных обозначений

SIMD Single Instruction Multiple Data, одиночный поток команд, множественный поток данных - один из классов архитектур классификации по Флинну

VE Vector Engine, векторное устройство системы NEC SX-Aurora TSUBASA, производящее основные векторные вычисления

VH Vector Host, векторный хост системы NEC SX-Aurora TSUBASA, выполняющее функции операционной системы и последовательные (не векторизуемые фрагменты программ), выгруженные пользователем с VE

SPU Scalar Processing Unit, скалярный вычислительный блок векторного ядра VE

VPU Vector Processing Unit, векторный вычислительный блок векторного ядра VE

VPP Vector Parallel Pipleing, векторно-параллельный конвейер ядра VE

FMA Fused Multiply-Add, совмещённое умножение-сложение, распространённая операция, при которой умножаются два числа и складываются с аккумулятором

ALU Arithmetic and Logic Unit, блок процессора, который под управлением устройства управления служит для выполнения арифметических и логических преобразований над данными

HBM High Bandwidth Memory, память с высокой пропускной способностью, интерфейс ОЗУ для DRAM с многослойной компоновкой кристаллов в микросборке от компаний AMD и Hynix, применяемая в высокопроизводительных видеокартах и сетевых устройствах

PCI Peripheral Component Interconnect, шина ввода-вывода для подключения периферийных устройств к материнской плате компьютера

SM Streaming Multiprocessor, потоковый мультипроцессор графических ускорителей NVIDIA GPU

LLC Last Level Cache, кэш последнего уровня архитектуры

BFS Breadth-First Search, алгоритм/задача поиска в ширину

SSSP Single Source Shortest Paths, задача поиска всех пар кратчайших путей от заданной вершины-источника

PR Page Rank, алгоритм ранжирования вершин в графе

BSP Bulk-Synchronous Parallel, модель графовых вычислений с барьерной синхронизацией по окончании каждой итерации

SoA Structure Of Arrays, модель хранения данных в виде структуры массивов, обеспечивающая эффективный доступ к апмяти для GPU и векторных систем

VGL Vector Graph Library, фреймворк для релаизации графовых алгоритмов на векторных системах, разработанный в ходе данного исследования

Словарь терминов

Граф –абстрактный математический объект, представляющий собой множество вершин графа и набор рёбер, то есть соединений между парами вершин.

SIMD-инструкция – обобщение понятия векторной инструкции и варпа графического ускорителя NVIDIA.

Векторные архитектуры с быстрой памятью (целевые) – современные вычислительные архитектуры, характеризующиеся тремя свойствами: (1) использование принципов векторной обработки данных, (2) наличие быстрой памяти, а так же (3) массивного параллелизма.

memory-bound и data-intensive приложения – программы, часто взаимодействующие с подсистемой памяти целевой архитектуры, а так же выполняющие сравнительно небольшое число вычислительных операций.

типовая алгоритмическая структура графового алгоритма – некоторый подграф макрографа информационного графа исследуемого графового алгоритма, изоморфный для всех исследуемых графовых алгоритмов при условии их применения к одному и тому же входному графу.

алгоритмические абстракции (абстракция вычислений) – совокупность информационной структуры, маркографов, а также подходы к реализации каждой из четырех выделенных типовых алгоритмических структур.

алгоритмический шаблон – группа из простейших операций в информационном графе, применяемых в рамках выделенных типовых алгоритмических для различных вершин и ребер графа, определенным в типовой алгоритмической структуре образом.

обход графа – применение различного рода вычислительных операций над вершинами и ребрами графа.

Список литературы

1. *McSherry Frank, Isard Michael, Murray Derek G.* Scalability! But at what {COST}? // 15th Workshop on Hot Topics in Operating Systems (HotOS {XV}). — 2015.

2. Four degrees of separation / Lars Backstrom, Paolo Boldi, Marco Rosa et al. // Proceedings of the 4th Annual ACM Web Science Conference. — 2012. — Pp. 33–42.

3. *Shun Julian, Blelloch Guy E.* Ligra: a lightweight graph processing framework for shared memory // ACM Sigplan Notices / ACM. — Vol. 48. — 2013. — Pp. 135–146.

4. *Yamada Yohei, Momose Shintaro.* Vector Engine Processor of NEC¡Çs Brand-New supercomputer SX-Aurora TSUBASA // Intenational symposium on High Performance Chips (Hot Chips2018). — 2018.

5. Developing Efficient Implementations of Bellman–Ford and Forward-Backward Graph Algorithms for NEC SX-ACE / Ilya V Afanasyev, Alexander S Antonov, Dmitry A Nikitenko et al. // *Supercomputing Frontiers and Innovations.* — 2018. — Vol. 5, no. 3. — Pp. 65–69. — [Scopus, Impact Factor: 0.422].

6. Developing Efficient Implementations of Shortest Paths and Page Rank Algorithms for NEC SX-Aurora TSUBASA Architecture / IV Afanasyev, Vad V Voevodin, Vl V Voevodin et al. // *Lobachevskii Journal of Mathematics.* — 2019. — Vol. 40, no. 11. — Pp. 1753–1762. — [Scopus, Impact Factor: 0.238].

7. *Afanasyev IV, Voevodin Vl V.* Developing Efficient Implementations of Connected Component Algorithms for NEC SX-Aurora TSUBASA // *Lobachevskii Journal of Mathematics.* — 2020. — Vol. 41, no. 8. — Pp. 1417–1426. — [Scopus, Impact Factor: 0.422].

8. Analysis of relationship between simd-processing features used in nvidia gpus and nec sx-aurora tsubasa vector processors / Ilya V Afanasyev, Vadim V Voevodin, Vladimir V Voevodin et al. // International Conference on Parallel Computing Technologies / Springer. — 2019. — Pp. 125–139.

9. *Afanasyev Ilya, Voevodin Vladimir.* The comparison of large-scale graph processing algorithms implementation methods for Intel KNL and NVIDIA GPU // Russian Supercomputing Days / Springer. — 2017. — Pp. 80–94. — [Scopus, Impact Factor: 0.7].

10. Techniques for Solving Large-Scale Graph Problems on Heterogeneous Platforms / Ilya Afanasyev, Alexander Daryin, Jack Dongarra et al. // Russian Supercomputing Days / Springer. — 2016. — Pp. 318–332. — [Scopus, Impact Factor: 0.7].

11. Developing an Efficient Vector-Friendly Implementation of the Breadth-First Search Algorithm for NEC SX-Aurora TSUBASA / Ilya V Afanasyev, Vladimir V Voevodin, Kazuhiko Komatsu, Hiroaki Kobayashi // International Conference on Parallel Computational Technologies / Springer. — 2020. — Pp. 131–145. — [Scopus, Impact Factor: 0.7].

12. Практика проведения анализа производительности суперкомпьютерных задач / Илья Викторович Афанасьев, Вадим Владимирович Воеводин, Владимир Юрьевич Рудяк, Александр Вячеславович Емельяненко // *вычислительные методы и программирование.* — 2019. — Vol. 20. — Pp. 346–355. — [RINC, Impact Factor: 0.429].

13. *Афанасьев ИВ.* Разработка прототипа высокопроизводительного графового фреймворка для векторной архитектуры NEC SX–Aurora TSUBASA // *Вычислительные методы и программирование.* — 2020. — Vol. 21. — Pp. 290–305. — [RINC, Impact Factor: 0.429].

14. *Ilya Afanasyev.* The Comparative Performance Analysis of Data-intensive Applications for IBM Minsky and Newell Systems // Proceedings of the 4th Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists / CEUR. — 2018. — Pp. 40–49.

15. *Afanasyev Ilya.* An Efficient Implementation of the Transitive Closure Problem on Intel KNL Architecture // Proceedings of the 3th Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists / CEUR. — 2017. — Pp. 10–19.

16. *Siek Jeremy, Lumsdaine Andrew, Lee Lie-Quan.* The boost graph library: user guide and reference manual. — Addison-Wesley, 2002.

17. *Nguyen Donald, Lenharth Andrew, Pingali Keshav.* A lightweight infrastructure for graph analytics // Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. — 2013. — Pp. 456–471.

18. *Beamer Scott, Asanović Krste, Patterson David.* The GAP benchmark suite // *arXiv preprint arXiv:1508.03619.* — 2015.

19. Mathematical foundations of the GraphBLAS / Jeremy Kepner, Peter Aaltonen, David Bader et al. // 2016 IEEE High Performance Extreme Computing Conference (HPEC) / IEEE. — 2016. — Pp. 1–9.

20. *Davis Timothy A.* Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss // 2018 IEEE High Performance extreme Computing Conference (HPEC) / IEEE. — 2018. — Pp. 1–6.

21. Making caches work for graph analytics / Yunming Zhang, Vladimir Kiriansky, Charith Mendis et al. // 2017 IEEE International Conference on Big Data (Big Data) / IEEE. — 2017. — Pp. 293–302.

22. *Zhang Kaiyuan, Chen Rong, Chen Haibo.* NUMA-aware graph-structured analytics // Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2015. — Pp. 183–193.

23. *Beamer Scott, Asanovic Krste, Patterson David.* Locality exists in graph processing: Workload characterization on an ivy bridge server // 2015 IEEE International Symposium on Workload Characterization / IEEE. — 2015. — Pp. 56–65.

24. *Harish Pawan, Narayanan PJ.* Accelerating large graph algorithms on the GPU using CUDA // International conference on high-performance computing / Springer. — 2007. — Pp. 197–208.

25. Graph processing on GPUs: A survey / Xuanhua Shi, Zhigao Zheng, Yongluan Zhou et al. // *ACM Computing Surveys (CSUR).* — 2018. — Vol. 50, no. 6. — Pp. 1–35.

26. *Khorasani Farzad, Gupta Rajiv, Bhuyan Laxmi N.* Scalable simd-efficient graph processing on gpus // 2015 International Conference on Parallel Architecture and Compilation (PACT) / IEEE. — 2015. — Pp. 39–50.

27. Gunrock: A high-performance graph processing library on the GPU / Yangzihao Wang, Andrew Davidson, Yuechao Pan et al. // Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2016. — Pp. 1–12.

28. *Zhong Jianlong, He Bingsheng.* Medusa: Simplified graph processing on GPUs // *IEEE Transactions on Parallel and Distributed Systems.* — 2013. — Vol. 25, no. 6. — Pp. 1543–1552.

29. 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. — 2014. — Pp. 239–252.

30. *Liu Hang, Huang H Howie.* Enterprise: breadth-first graph traversal on GPUs // Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. — 2015. — Pp. 1–12.

31. Efficient large-scale graph processing on hybrid CPU and GPU systems / Abdullah Gharaibeh, Tahsin Reza, Elizeu Santos-Neto et al. // *arXiv preprint arXiv:1312.3018.* — 2013.

32. *Chen Xuhao.* GraphCage: Cache Aware Graph Processing on GPUs // *arXiv preprint arXiv:1904.02241.* — 2019.

33. *Liu Hang, Huang H Howie.* Simd-x: Programming and processing of graph algorithms on gpus // 2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19). — 2019. — Pp. 411–428.

34. Slimsell: A vectorizable graph representation for breadth-first search / Maciej Besta, Florian Marending, Edgar Solomonik, Torsten Hoefler // 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS) / IEEE. — 2017. — Pp. 32–41.

35. Generalized Sparse Matrix-Matrix Multiplication for Vector Engines and Graph Applications / Jiayu Li, Fugang Wang, Takuya Araki, Judy Qiu // 2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC) / IEEE. — 2019. — Pp. 33–42.

36. *Kolganov AS.* The fastest and energy-efficient breadth-first search algorithm on a single node with various parallel architectures according to Graph500 // *Vestnik Yuzhno-Ural'skogo Gosudarstvennogo Universiteta. Seriya"Vychislitelnaya Matematika i Informatika".* — 2018. — Vol. 7, no. 2. — Pp. 5–21.

37. *Merrill Duane, Garland Michael, Grimshaw Andrew.* Scalable GPU graph traversal // *Acm Sigplan Notices.* — 2012. — Vol. 47, no. 8. — Pp. 117–128.

38. Accelerating CUDA graph algorithms at maximum warp / Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, Kunle Olukotun // *Acm Sigplan Notices.* — 2011. — Vol. 46, no. 8. — Pp. 267–276.

39. Work-efficient parallel GPU methods for single-source shortest paths / Andrew Davidson, Sean Baxter, Michael Garland, John D Owens // 2014 IEEE 28th International Parallel and Distributed Processing Symposium / IEEE. — 2014. — Pp. 349–359.

40. To push or to pull: On reducing communication and synchronization in graph computations / Maciej Besta, Michał Podstawski, Linus Groner et al. // Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing. — 2017. — Pp. 93–104.

41. Optimizing Cache Performance for Graph Analytics / Yunming Zhang, Vladimir Kiriansky, Charith Mendis et al. // *ArXiv.* — 2016. — Vol. abs/1608.01362.

42. *Nesbit Kyle J, Smith James E.* Data cache prefetching using a global history buffer // 10th International Symposium on High Performance Computer Architecture (HPCA'04) / IEEE. — 2004. — Pp. 96–96.

43. *Sun Jiawen, Vandierendonck Hans, Nikolopoulos Dimitrios S.* Accelerating graph analytics by utilising the memory locality of graph partitioning // 2017 46th International Conference on Parallel Processing (ICPP) / IEEE. — 2017. — Pp. 181–190.

44. *Roy Amitabha, Mihailovic Ivo, Zwaenepoel Willy.* X-stream: Edge-centric graph processing using streaming partitions // Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. — 2013. — Pp. 472–488.

45. Optimizing graph processing on gpus / Wenyong Zhong, Jianhua Sun, Hao Chen et al. // *IEEE Transactions on Parallel and Distributed Systems.* — 2016. — Vol. 28, no. 4. — Pp. 1149–1162.

46. Recall: Reordered cache aware locality based graph processing / Kartik Lakhotia, Shreyas Singapura, Rajgopal Kannan, Viktor Prasanna // 2017 IEEE 24th International Conference on High Performance Computing (HiPC) / IEEE. — 2017. — Pp. 273–282.

47. Introducing the graph 500 / Richard C Murphy, Kyle B Wheeler, Brian W Barrett, James A Ang // *Cray Users Group (CUG).* — 2010. — Vol. 19. — Pp. 45–74.

48. *Williams Samuel, Waterman Andrew, Patterson David.* Roofline: an insightful visual performance model for multicore architectures // *Communications of the ACM.* — 2009. — Vol. 52, no. 4. — Pp. 65–76.

49. *Ilic Aleksandar, Pratas Frederico, Sousa Leonel.* Cache-aware roofline model: Upgrading the loft // *IEEE Computer Architecture Letters.* — 2013. — Vol. 13, no. 1. — Pp. 21–24.

50. Roofline model toolkit: A practical tool for architectural and program analysis / Yu Jung Lo, Samuel Williams, Brian Van Straalen et al. // International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems / Springer. — 2014. — Pp. 129–148.

51. *CaBcaval Calin, Padua David A.* Estimating cache misses and locality using stack distances // Proceedings of the 17th annual international conference on Supercomputing. — 2003. — Pp. 150–159.

52. *Yu Jing, Baghsorkhi Sara, Snir Marc.* A new locality metric and case studies for hpcs benchmarks. — 2005.

53. A detailed GPU cache model based on reuse distance theory / Cedric Nugteren, Gert-Jan Van den Braak, Henk Corporaal, Henri Bal // 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA) / IEEE. — 2014. — Pp. 37–48.

54. *Chakrabarti Deepayan, Zhan Yiping, Faloutsos Christos.* R-MAT: A recursive model for graph mining // Proceedings of the 2004 SIAM International Conference on Data Mining / SIAM. — 2004. — Pp. 442–446.

55. *ERDdS P, R&wi A.* On random graphs i. — Vol. 6. — 1959. — P. 18.

56. The Koblenz Network Collection - KONECT. — URL: http://konect.uni-koblenz.de.

57. Stanford Large Network Dataset Collection - SNAP. — URL: https://snap.stanford.edu/data/.

58. *Flynn Michael J.* Very high-speed computing systems // *Proceedings of the IEEE.* — 1966. — Vol. 54, no. 12. — Pp. 1901–1909.

59. Performance Evaluation of a Vector Supercomputer SX-aurora TSUBASA / Kazuhiko Komatsu, Shintaro Momose, Yoko Isobe et al. // Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. — SC '18. — Piscataway, NJ, USA: IEEE Press, 2018. — Pp. 54:1–54:12.

60. Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE / Ryusuke Egawa, Kazuhiko Komatsu, Shintaro Momose et al. // *The Journal of Supercomputing.* — 2017. — Sep. — Vol. 73, no. 9. — Pp. 3948–3976.

61. An approach to the highest efficiency of the HPCG benchmark on the SX-ACE supercomputer / Kazuhiko Komatsu, Ryusuke Egawa, Yoko Isobe et al. // Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC15), Poster. — 2015. — Nov. — Pp. 1–2.

62. NEC SX-Aurora TSUBASA C/C++ Compiler User's Guide. — https://www.hpc.nec/documents/sdk/pdfs/g2af01e-C++UsersGuide-016.pdf. — Accessed: 2020-05-12.

63. *NVIDIA.* NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU // *Whitepaper.* — 2016.

64. *Kirk David et al.* NVIDIA CUDA software and GPU parallel computing architecture // ISMM. — Vol. 7. — 2007. — Pp. 103–104.

65. OpenACC—first experiences with real-world applications / Sandra Wienke, Paul Springer, Christian Terboven, Dieter an Mey // European Conference on Parallel Processing / Springer. — 2012. — Pp. 859–870.

66. *NVIDIA Tesla*. V100 GPU architecture. — 2017.

67. *Bergstrom Lars*. Measuring NUMA effects with the STREAM benchmark // *arXiv preprint arXiv:1103.3225*. — 2011.

68. *Deakin Tom, McIntosh-Smith Simon*. Gpu-stream: Benchmarking the achievable memory bandwidth of graphics processing units // IEEE/ACM SuperComputing. — 2015. — Pp. 3202–3216.

69. *Hillis W Daniel, Steele Jr Guy L*. Data parallel algorithms // *Communications of the ACM*. — 1986. — Vol. 29, no. 12. — Pp. 1170–1183.

70. *Harris Mark et al*. Optimizing parallel reduction in CUDA // *Nvidia developer technology*. — 2007. — Vol. 2, no. 4. — P. 70.

71. *Voevodin VV*. Voevodin Vl. V. Parallel'nye vychislenija // *S.-Pb.: BHV-Peterburg*. — 2002.

72. *Johnson Donald B*. A note on Dijkstra's shortest path algorithm // *Journal of the ACM (JACM)*. — 1973. — Vol. 20, no. 3. — Pp. 385–388.

73. *Goldberg Andrew, Radzik Tomasz*. A heuristic improvement of the Bellman-Ford algorithm. — 1993.

74. *Meyer Ulrich, Sanders Peter*. Δ-stepping: a parallelizable shortest path algorithm // *Journal of Algorithms*. — 2003. — Vol. 49, no. 1. — Pp. 114–152.

75. *Beamer Scott, Asanović Krste, Patterson David*. Direction-optimizing breadth-first search // *Scientific Programming*. — 2013. — Vol. 21, no. 3-4. — Pp. 137–148.

76. *Shiloach Yossi, Vishkin Uzi*. An O (log n) parallel connectivity algorithm. — 1980.

77. The PageRank citation ranking: Bringing order to the web. / Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd. — 1999.

78. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units / Moritz Kreutzer, Georg Hager, Gerhard Wellein et al. // *SIAM Journal on Scientific Computing*. — 2014. — Vol. 36, no. 5. — Pp. C401–C423.

79. *Anzt Hartwig, Tomov Stanimire, Dongarra Jack.* Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs // *University of Tennessee, Tech. Rep. ut-eecs-14-727.* — 2014.

80. *Harris Mark et al.* Optimizing parallel reduction in CUDA // *Nvidia developer technology.* — 2007. — Vol. 2, no. 4. — P. 70.

Список рисунков

Список таблиц