Федеральное государственное бюджетное учреждение науки
Институт вычислительной математики и математической геофизики
Сибирского отделения Российской академии наук

На правах рукописи

Снытникова Татьяна Валентиновна

Эффективная реализация модели ассоциативных вычислений на графических ускорителях для решения задач на графах

2.3.5 – Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата технических наук

Научный руководитель
д. т. н.
Глинский Борис Михайлович

Новосибирск – 2022

Оглавление

Введение 3

Глава 1. Развитие ассоциативных параллельных моделей и архитектур 9
1.1. STARAN и ASPRO 11
1.2. Процессор IXM2 12
1.3. Процессор Rutgers CAM2000 14
1.4. FastTrack Processor для ATLAS 15
1.5. Модели ASC и MASC 20
1.6. Модель STAR 25
1.7. Выводы к первой главе 29

Глава 2. Реализация модели ассоциативных параллельных вычислений на GPU 31
2.1. Ключевые отличия ассоциативных систем от PRAM-архитектур 31
2.2. Реализация STAR-машины на графических ускорителях 32
2.3. Оптимизация ассоциативных алгоритмов под исполнение на графических ускорителях 60
2.4. Выводы ко второй главе 62

Глава 3. Ассоциативные параллельные алгоритмы для решения задач на графах 64
3.1. Алгоритм Уоршалла транзитивного замыкания 64
3.2. Алгоритм Дейкстры нахождения кратчайших путей 74
3.3. Динамический алгоритм Рамалингама для проблемы достижимости в потоковом графе с одним источником 87
3.4. Выводы к третьей главе 100

Заключение 102

Список сокращений и терминов 106

Список литературы 107

Введение

Актуальность работы

По статистике в современном мире объем цифровой информации удваивается каждые восемнадцать месяцев. Большая часть информации не является структурированной (организованной в базы данных и знаний). Поэтому обеспечение быстрого поиска образца среди больших объемов информации является актуальной проблемой современной информационной науки. Это привело к выделению науки о данных, как отдельной области науки. При этом известно, что алгоритмы поиска по несортированным данным являются узким местом для вычислительных машин фон-неймановского типа.

В то же время ассоциативные параллельные модели и архитектуры обладают следующим свойством: время выполнения базовых операции поиска ($=$,$<$, $>$, min, max) в массиве не зависит от числа строк. Поэтому все ассоциативные архитектуры разрабатывались и создавались для выполнения конкретных задач, в которых критичны лимиты времени поиска по большому массиву неструктурированных данных.

Развитие ассоциативных моделей остается актуальным. Ведется работа над реализацией ассоциативных моделей на существующем оборудовании. Также создается новое аппаратное обеспечение для ассоциативных параллельных вычислений (чипы ассоциативной памяти, ассоциативные процессоры и системы). Для этих моделей и систем разрабатываются алгоритмы. В качестве ассоциативной модели ранее Непомнящей Анной Шмилевной была предложена STAR-машина. Таким образом актуальность работы заключается в предоставлении возможности использовать преимущества ассоциативных вычислений на графических ускорителях.

Степень разработанности темы С одной стороны, реализация ассоциативных процессоров на системах неассоциативного типа встречаются достаточно регулярно. Но обычно речь идет об экспериментальных образцах ассоциативных процессоров. Целью таких реализаций является предсказание свойств этих процессоров, таких как производительность и электроемкость. При этом моделируются только низкоуровневые операции. С другой стороны на построенных

ассоциативных системах, использовались языки высокого уровня, расширенные операциями для ассоциативной обработки данных.

Необходимо отметить, что за исключением первой ассоциативной системы STARAN, которая считалась универсальной, остальные системы строились для решения конкретных узких задач. Поэтому параллельно с конструированием новых ассоциативных систем развивались две модели ассоциативных вычислений. Обе абстрактные модели основаны на системе STARAN и используют языки высокого уровня: STAR-машина[1] и ASC[2]. Для абстрактной модели ассоциативных вычислений STAR-машины была доказана эквивалентность модели ASC. Для этой модели ассоциативных алгоритмов было разработано большое число ассоциативных алгоритмов для решения задач на графах. Кроме классических алгоритмов разрабатывались также динамические алгоритмы, которые позволяют перестраивать решение после добавления и/или удаления ребер или вершин в графе.

Таким образом реализация STAR-машины на графических ускорителях позволяет получать параллельные алгоритмы для решения задач на графах, в том числе и динамические. И если распараллеливание классических алгоритмов для вычислений на графических ускорителях развивается, то параллельных динамических алгоритмах на графических ускорителях обнаружить не удалось.

Цель диссертационной работы состоит в построении эффективной реализации STAR-машины на графических ускорителях с использованием технологии CUDA. Это позволит использовать на практике ассоциативные параллельные алгоритмы, разработанные для этой модели.

Для достижения поставленной цели были решены следующие задачи:

- построена реализация базовых операций языка Star на графическом ускорителе;
- для увеличения эффективности реализации выделены операции языка Star, критичные к синхронизации;
- реализована на графическом ускорителе библиотека стандартных процедур языка Star;

---

[1] ВЦ СО АН СССР и далее в ИВМиМГ СО РАН, Непомнящая А.Ш.

[2] Kent State University, Department of Computer Science, Distributed and Parallel Processing

- на основании анализа существующих форматов входных/выходных данных для тестовых графов (более 5000 вершин) разработан модуль ввода/вывода данных отобранных форматов во внутреннее представление реализации Star-машины;
- обоснована эффективность реализации Star-машины как оценкой теоретической сложности процедур реализации, так и практическим сравнением времени работы с временем работы аналогов (для доказательства эффективности реализации базовых операций сравниваются время выполнения ассоциативной версии алгоритма Уоршалла с временем выполнения других параллельных реализаций этого алгоритма; для обоснования эффективности реализации библиотеки стандартных процедур проводится сравнение времени работы базовых процедур с временем работы аналогов из библиотек STL и CUDA thrust);
- разработаны методы оптимизации ассоциативных алгоритмов для выполнения на графических ускорителях, учитывающие различия Star-машины и GPU.

Научная новизна Предложена уникальная реализация модели ассоциативных вычислений на графических ускорителях: эффективно сохраняет ассоциативные свойства; расчитана на выполнение ассоциативных алгоритмов модели, а не прогнозирование ее свойств.

Для динамических алгоритмов решения задач теории графов нет неассоциативных параллельных алгоритмов, поскольку последовательные алгоритмы используют структуры данных, сложные для распараллеливания. Но использование данной технологии позволяет разрабатывать параллельные динамические алгоритмы.

Разработанные методы оптимизации ассоциативных алгоритмов для выполнения на графических ускорителях позволяют легко локализовать точки синхронизации в ассоциативных алгоритмах при реализации на GPU. Это значительно уменьшает трудозатраты разработчиков при их реализации.

Практическая значимость Для STAR-машины разработаны как классические, так и динамические алгоритмы для решения задач на графах. Реализация этих алгоритмов на графических ускорителях дает возможность их практического использования, сохраняя преимущества ассоциативной обработки.

Заметим также, что при распараллеливании алгоритмов возникают трудности с определением точек синхронизации. С одной стороны, неоптимальное расположение точек синхронизации потоков вычислений может сильно снизить производительность параллельной программы. С другой стороны, отсутствие необходимой точки синхронизации приводит к появлению ошибок, которые из-за большой степени недетерминизма параллельного исполнения трудно обнаружить традиционными методами отладки. Использование предложенной технологии значительно уменьшает трудозатраты разработчиков при разрабатывании и отладки параллельных алгоритмов.

В работе было показано, что некоторые ассоциативные параллельные алгоритмы могут быть адаптированы для выполнения на графических ускорителях, если принимать во внимание архитектурные различия STAR-машины и графических ускорителей. Так, адаптация ассоциативной версии алгоритма Уоршалла под графические ускорители дала ускорение в 2 378 раз на графе с 5 000 вершин.

Методология и методы исследования При получении основных результатов диссертационной работы использовались методы параллельного программирования, методы анализа информационной структуры параллельных алгоритмов, элементы теории графов, 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, а также средства анализа эффективности и производительности – nvprof.

На защиту выносятся следующие основные положения и результаты: Построена реализация абстрактной модели ассоциативной обработки данных (STAR-машины) на современной параллельной архитектуре (графических ускорителях).

Указаны операции языка Star, критичные к синхронизации.

Разработана классификация библиотеки стандартных процедур языка Star по способу обработки данных.

Выработаны методы оптимизации ассоциативных алгоритмов для выпол-

нения на графических ускорителях, учитывающие различия Star-машины и GPU.

Эффективность реализации обоснована в теории и на примере выполнения ассоциативных алгоритмов.

Апробация работы Результаты работы обсуждались на семинарах «Математическое обеспечение высокопроизводительных вычислительных систем» ИВМиМГ СО РАН, «Дискретные экстремальные задачи» Института математики СО РАН, «Высокопроизводительные вычисления» ИВМиМГ СО РАН, лаборатории программных систем машинной графики ИАиЭ СО РАН, « ruSTEP по-русски» совместно Иннополис и ИСИ СО РАН, «Интеллектуальные системы и системное программирование» совместно ИСИ СО РАН и кафедры программирования НГУ. Основные результаты диссертации докладывались на конференциях МАРЧУКОВСКИЕ НАУЧНЫЕ ЧТЕНИЯ в 2017 и 2020 годах.

Публикации Материалы диссертации опубликованы в 8 печатных работах, из них в 4 статьях в научных журналах, входящих в перечень ВАК. Получено свидетельство о регистрации программ для ЭВМ.

Личный вклад автора Содержание диссертации и основные положения, выносимые на защиту, отражают персональный вклад автора в опубликованных работах. Подготовка к публикации полученных результатов проводилась совместно с соавтором. Соавтор претензий не имеет. Все представленные в диссертации результаты получены лично автором.

В работе [1] автору принадлежат следующие результаты: реализация операторов языка STAR на графических ускорителях, адаптация и оптимизация ассоциативного алгоритма Уоршалла под исполнение на GPU, а также проведение вычислительных расчетов и сравнение с неассоциативными реализациями алгоритма Уоршалла на GPU.

В работе [2] автору принадлежит реализация библиотеки стандартных ассоциативных алгоритмов на графических ускорителях, разработана классификация библиотеки стандартных процедур языка Star по способу обработки данных;

В работах [3, 4, 5] ассоциативные параллельные алгоритмы были разработаны и доказана их корректность совместно с соавтором. Оптимизация и реа-

лизация алгоритмов на графических ускорителях, а также численные расчеты произведены лично автором.

Структура и объем диссертации Диссертация состоит из введения, 3-х глав, заключения, списка сокращений и терминов, а также библиографии. Общий объем диссертации 114 страниц, из них 105 страницы текста, включая 28 рисунков. Библиография включает 79 наименований на 8 страницах.

Глава 1

Развитие ассоциативных параллельных моделей и архитектур

Разработки многопроцессорных компьютеров для параллельного решения математических задач начинаются с начала 60-х годов XX века. Изначальный подход к распараллеливанию вычислений заключается в проектировании и создании такой архитектуры машины, чтобы она могла одновременно выполнять несколько операций. Однако, если компьютер уже создан, то для решения конкретной задачи необходимо придумать или выбрать алгоритм, который мог бы исполняться наиболее эффективно на этой архитектуре. Эта актуальная проблема в своей современной трактовке была сформулирована академиком Г.И. Марчуком еще в 70-е годы как «отображение алгоритмов на архитектуру вычислительной системы»[1]. Таким образом, проблема параллелизма является триединой: здесь переплетаются фундаментальные проблемы и вычислительной техники, и алгоритмов, и программирования.

И если в настоящее время на первый план выдвинуты проблемы параллельных алгоритмов и параллельного программирования, то в 60-е и 70-е годы большой упор делался на решении проблем вычислительной техники. Так 1970 г. Хоббс и Хейс предложили обстоятельную классификацию паралельных систем:

- многомашинные и многопроцессорные системы;
- ассоциативные процессоры;
- сетевые или матричные процессоры;
- функциональные машины.

Основное развитие высокопроизводительных вычислительных машин пошло в направлении многомашинных и многопроцессорных систем. Но нужно отме-

[1] мировой лозунг - "mapping of algorithms on the computer architecture дословный перевод на английский язык)

тить, что как на любом уровне развития вычислительной машины остается понятия большой задачи, с которой за разумное время не может справиться машина типа RAM, так и возникают задачи, которые не могут быть эффективно решены на системах PRAM из-за большого числа операций сравнения или поиска. И тогда возникает необходимость в использовании ассоциатихных параллельных системах.

В архитектурах фон-немановского типа используется запоминающее устройство с произвольным доступом (Random Access Memory, RAM), позволяющее получить доступ к любой ячейки по ее адресу на чтение или запись.

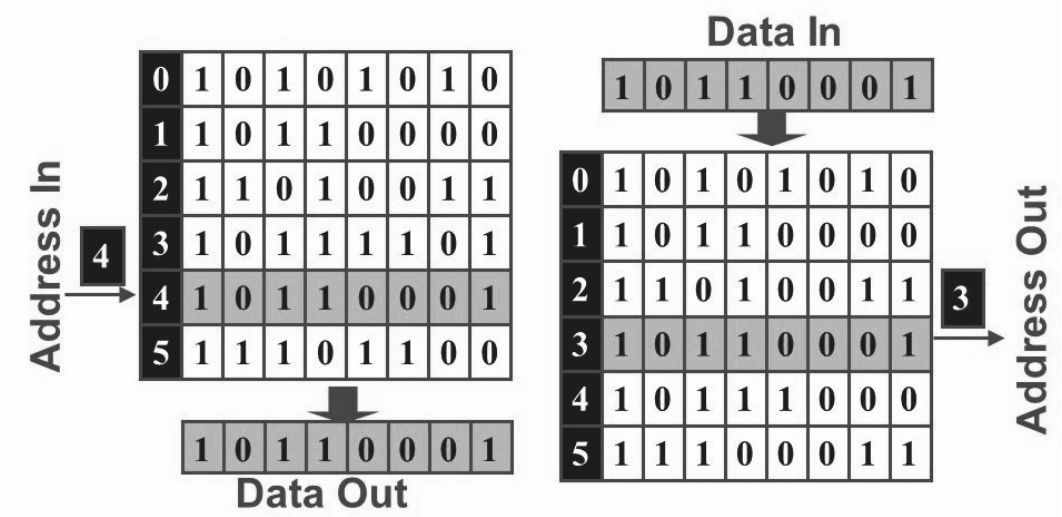


Рис. 1.1. Принцип адресации в моделях RAM и CAM.

В отличие от RAM ассоциативная память (память, адресуемая по содержимому, content-addressable memory, CAM)[6, 7] сравнивает входные данные с содержанием табличной памяти и возвращает адрес соответствующих данных (рис. 1.1). Поиск данных по табличной памяти CAM производится за один тактовый цикл, поэтому ассоциативная память используется в приложениях, требующих высокой скорости поиска: скоростных кэшах, сетевых маршрутизаторах, в специализированных системах обработки баз данных и знаний.

Кроме этого ведутся разработки TCAM (ternary CAM)[8, 9], позволяющей производить более гибкий поиск за счет добавления к “0” и “1” третьего значения для сравнения “x” (или “не важно”).

Отметим, что ассоциативные процессоры (АП) [10, 11] отличаются от ассо-

циативной памяти тем, что могут производить не только ассоциативный поиск по данным, но и обработку данных табличной памяти. Ниже мы приведем описание ассоциативных архитектур и области, для которых они разрабатывались. Также представим две модели ассоциативных вычислений.

1.1. STARAN и ASPRO

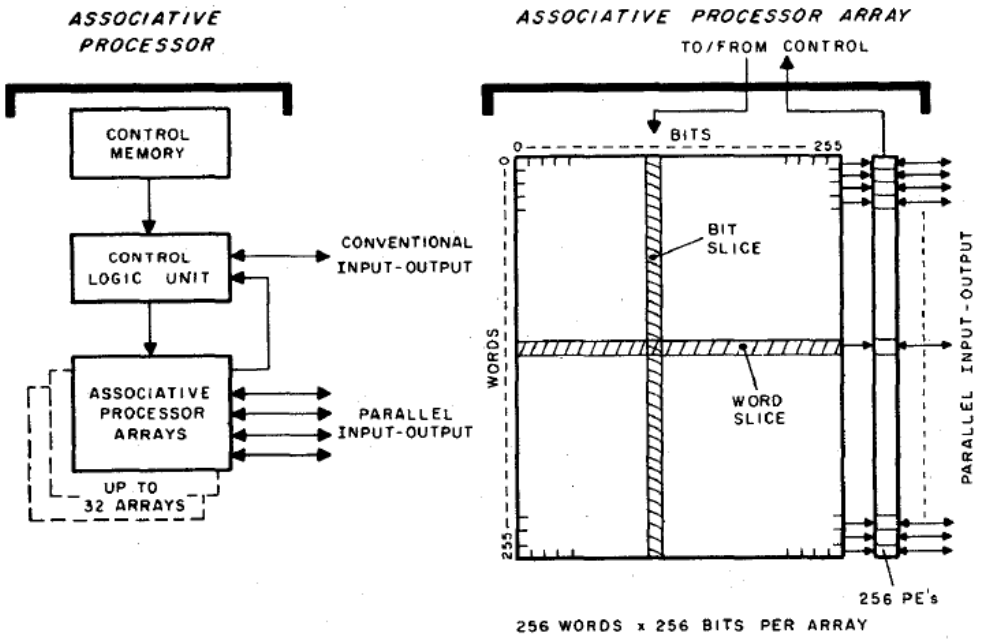


Рис. 1.2. Структура ассоциативного процессора STARAN

Первой коммерчески успешной версией ассоциативного параллельного процессора был STARAN (Stellar Attitude Reference and Navigation), разработанный Goodyear Aerospace и выпущенный в мае 1972 г [12, 13, 14, 15].

Сердцем архитектуры STARAN является ассоциативный процессорный массив (рис. 1.2), который состоит из 256 1-битовых процессорных элементов (ПЭ, PE), матричной памяти, флип-сети. Память содержит 256 слов длиной по 256 бит. Доступ к ней производится в двух режимах: в режиме битового слайса (столбца) или в режиме слова. В режиме битового слайса можно получить доступ к одному биту каждого слова, позволяя массиву из 256 ПЭ элементов работать с данными параллельно. В режиме слова все 256 бит одного слова

могут быть доступны для эффективного ввода или вывода. Флип-сеть позволяет перемещать данные между ПЭ параллельно. К логическому устройству управления (CONTROL LOGIC UNIT) подключается до 32 ассоциативных процессорных массивов.

Под этот процессор был разработан язык программирования низкого уровня APPLE (Associative Processor Procedure Language) [16, 17].

В 1982 году на базе архитектуры STARAN была разработана система ASPRO (Airborne Associative Processor) [18, 19], использующая чипы VLSI (very-large-scale integration, СБИС). Каждый чип содержал 32 ПЭ, соединенных флип-сетью. Таким образом, 1024-процессорная система занимала менее $0,03$ м^3. ASPRO разрабатывалась для систем управления воздушными сообщениями США. По данным 1983 года эта система использовался в радарах самолетов-разведчиков E-2 Hawkeye AWACS ВМС США.

На базе систем STARAN и ASPRO Поттер разработал модели ассоциативного процессора ASC (класс SIMD) и MASC (класс MIMD). В Кентском Государственном Университете ведутся работы по созданию современного ассоциативного параллельного процессора для системы MASC [20] и эффективной реализации этой модели на других архитектурах [21].

1.2. Процессор IXM2

В 1991 году для ETL (ElectroTechnical Laboratory, Япония) был разработан параллельный ассоциативный процессор IXM2 для обработки знаний [22] и для обработки семантических сетей[23]. IXM2 состоит из 64 АП и 9 сетевых процессоров, имеющих вместе 256 тысяч слов ассоциативной памяти. Это позволяет выполнять базовые операции параллельно над 65 536 вершин семантической сети за константное время. На рисунке 1.3 показана структура IXM2. Восемь АП и один сетевой процессор образуют вычислительный модуль, в котором ассоциативные процессоры связаны все со всеми. Аналогично, восемь вычислительных модулей соединены все со всеми и с выделенным сетевым процессором. Этот сетевой процессор подключен к компьютеру SUN-3.

IXM2 предлагалось использовать как процессор макроинструкций и как

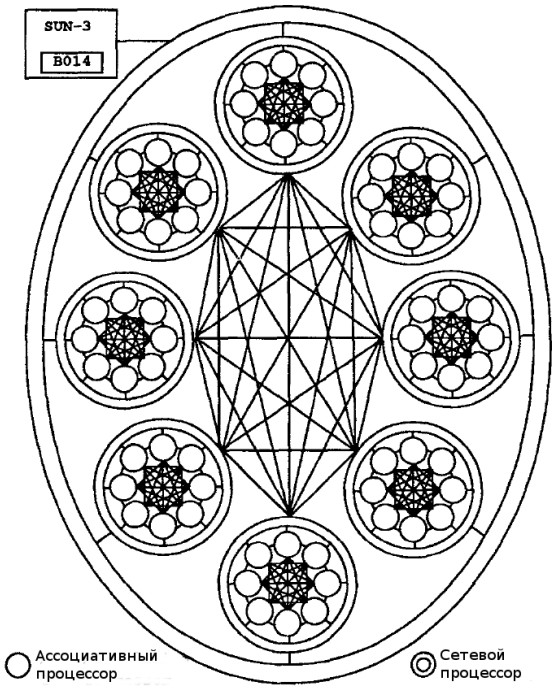


Рис. 1.3. Структура IXM2

процессор семантических сетей [24]. IXM2 в качестве макро-инструкций поддерживает арифметические и логические операции: add, sub, multiplication, less-then, greater-then. Они вычисляются алгоритмами последовательно по битам, но параллельно по словам. Макроинструкции могут быть вызваны из C-программы на управляющей машине и выполняться параллельно на IXM2.

Обработка семантических сетей — одно из основных приложений IXM2. IXM2 выполняет программы, написанные на языке представления знаний IXL, расширении языка Prolog. Он использует специальные предикаты для обработки семантических сетей в дополнение к предикатам, определенным в языке Prolog.

Также IXM2 использовался для выполнения генетических алгоритмов и поиска по базам знаний, а также в различных исследовательских проектах ETL, Carnegie Mellon University и the ATR Interpreting Telephony Research Laboratory.

На основе процессора IXM2 были построены компьютернная система машинного перевода ASTRAL [25], система машинного перевода EBMT (Example-Based Machine Translation) [26], система TDMT (Transfer-Driven Machine Translation) для перевода устной речи в режиме реального времени [27].

1.3. Процессор Rutgers CAM2000

В 1993 году при поддержке NASA в Ратгерском университете был разработан чип Rudgers CAM2000 [28, 29]. Он объединяет возможности ассоциативного процессора (AP), ассоциативной памяти (CAM) и динамической памяти с произвольным доступом (DRAM) в одном кристалле.

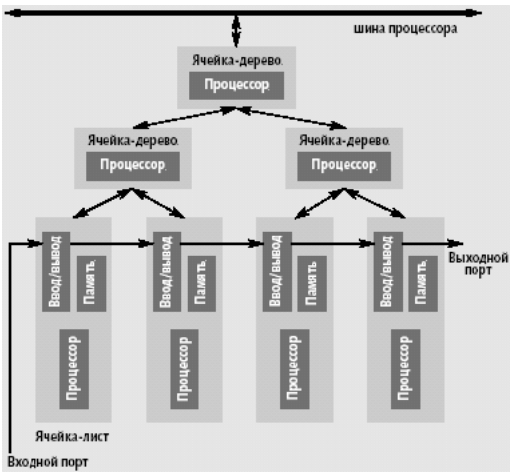


Рис. 1.4. Структура чипа Rudgers CAM2000

Архитектура CAM2000 представляет машину с древовидной структурой, состоящей из четырех попарно соединенных компонентов: дерева, листьев, памяти и устройств ввода/вывода. Ячейка-дерево состоит из трех ячеек, соединенных в виде бинарного дерева. Они выполняют глобальные операции над данными, расположенными в ячейках-листьях. Ячейка-лист состоит из процессора, банка локальных регистров, локальной памяти и одного регистра параллельного сдвига, формирующего компоненты ввода/вывода. Ячейки-листья выполняют локальные операции над данными, расположенными в своей памяти и множестве регистров. На рисунке 1.4 показан пример архитектуры CAM2000 с четырьмя ячейками-листьями.

Rudgers CAM2000 использует расширенные версии свойств классических CAM. Следующие четыре расширения существенны для производительности архитектуры CAM2000:

- **Длинные слова:** архитектура CAM2000 включает в себя как одноразрядные, так и многоразрядные системы, позволяющие производить вы-

численния над 32-х разрядными словами.

- **Глобальные операции:** архитектура обеспечивает на аппаратном уровне выполнение таких операций как "число ответчиков"и "сумма всех значений".

- **Сегментирование:** архитектура обеспечивает аппаратный контроль, поддерживающий произвольное разбиение на сегменты всех глобальных операций, что позволяет выполнять глобальные операции одновременно, также как выполняются несегментированные глобальные операции.

- **Локальная адресация:** конструкция CAM2000 позволяет производить вычисления над разными полями в различных ячейках-листьях.

Для процессора были разработаны языки низкого уровня CAML и высокого уровня Linear C [30].

1.4. FastTrack Processor для ATLAS

В режиме опытной эксплуатаци находится крупнейший проект с использованием ассоциативной архитектуры [31, 32]. ATLAS Fast Tracker (FTK) процессор состоит из 8 192 чипов. Каждый чип ассоциативной памяти хранит банк данных со 128 000 паттернов. Любой запрос к данным выполняется по всем элементам памяти одновременно за одинаковое время ($10^{-9}c$) вне зависимости от размера банка данных. Система ориентирована на решение задач физики высоких энергий.

Система FastTracker Processor (FTK) [33] состоит из подсистем с разными функциями. Устройство форматирования данных собирает данные и пересылает их в обрабатывающие устройства. Устройства ассоциативной памяти выполняют распознавание паттернов заряженных частиц и определение треков частиц.

1.4.1. Цель проекта и технические особенности

Система FTK проектировалась как часть детектора ATLAS Большого Адронного Коллайдера (Large Hadron Collaider, LHC)[34, 35], предназначенного для изучения процессов с высокоэнергетическими частицами. Одна из задач - обнаружение и исследование базона Хиггса [36].

До 2015 года в LHC сталкивались пучки протонов каждые 50 наносекунд, что в среднем составляет около 20 одновременных индивидуальных протон-протонных взаимодействий (pile-up, PU). После 2015 года пучки сталкиваются каждые 25 наносекунд и составляют 40 одновременных протон-протонных соединений. С повышением светимости LHC до проектируемой в 2026-2035 годах ожидаемое количество протон-протонных взаимодействий при столкновении пучков возрастет до 200-400. При этом бозоны Хиггса образуются в 10^9 раз реже, чем происходят обычные протон-протонные столкновения. Это соответствует частоте $1 - 10$ событий в час. Поэтому задача системы - распознать и сохранить полезные события при значительном подавлении фоновых процессов. Для этого на системе FTK решаются две задачи:

PM pattern matching – сопоставление с паттерном;

TF track fitting – реконструкция треков.

Сопоставление с паттерном выполняют ассоциативные процессоры, а реконструкцию треков – платы на основе FPGA.

год	частота событий	PU	объем данных	скорость данных
2006	20Mhz	20	0,5 MB	$\approx 10\,TB/s$
2014	40Mhz	40	1 MB	$\approx 40\,TB/s$
2016	40Mhz	40	1,6 – 1,8 MB	$\approx 80\,TB/s$
2026	40Mhz	200-400	2,4 MB	$\approx 1\,PB/s$

Таблица 1.1. Объем и скорость передачи данных с детектора ATLAS.

В таблице 1.1 представлены частота столкновений пучков (событий) в LHC, количество одновременных протон-протоновых столкновений на событие

(PU), объем данных для записи одного события после подавления нулей и скорость передачи данных с детектора для обработки на FTK [37, 38]. Скорость передачи данных на 2026 оценивалась исходя из проектных показателей LS3 модернизации большого адронного коллайдера, планируемого в 2024-2026 гг, после которого LHC перейдет к работе в режиме повышенной светимости.

Таким образом, при разработке системы FTK учитывались следующие особенности эксплуатации:

- огромный объем обрабатываемых данных;
- обработка в режиме реального времени;
- ограничения по пространству и энергопотреблению;
- сопоставление с паттернами по восьми признакам одновременно;
- большое количество паттернов.

1.4.2. Эволюция и характеристики AMchip для Fast TracKer

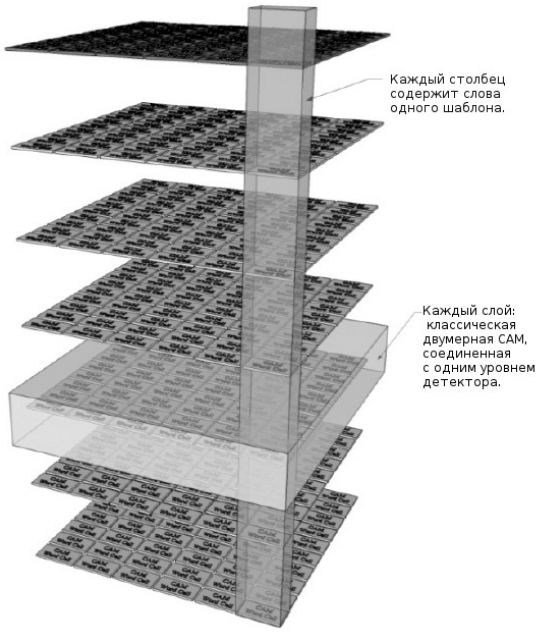


Рис. 1.5. Структура AM чипа

Чип AM представляет собой устройство, выполняющее сопоставление с паттерном, подобное памяти, адресуемой по содержимому (CAM) [39]. Однако дизайн AM концептуально отличается от дизайна CAM (рис. 1.5). В AM каждый паттерн хранится не в одном месте памяти, как в коммерческом CAM, но он состоит из 8 независимых 16-битных слов, хранящих координаты частиц, зафиксированные детектором. Инновационная характеристика AM заключается в том, что каждый из этих 8 слов имеет компаратор и триггер для сравнения непрерывно хранящихся данных (паттернами) с собственным потоком входных данных (hit). Данные отправляются по 8 параллельным шинам, по одному для каждого слова паттерна. Все слова в AM делают независимые и одновременные сравнения с данными, последовательно представленными на его собственной шине. Каждый раз, когда совпадение найдено, триггер соответствия устанавливается и остается установленным до конца обработки события, когда распространяется сигнал сброса. Паттерн совпадает, когда установоено предопределенное количество триггеров ($6 - 8$, задается пользователем). Все согласованные паттерны считываются. Подробное описание AM и его операций описываются в [40].

В эксперименте H1 использовались коммерческие CAM [41]. Каждое битовое слово CAM соответствовало каналу детектора. После роста количества детекторных каналов до $\sim 10^8$ при модернизации LHC этот подход стал невозможен. Кроме того, требовалось переформатировать данные прежде чем отправлять их на вход в CAM. Эта проблема была решена в AM чипе. Первое AM-устройство было создано для эксперимента CDF [42] на Tevatron-коллайдере Fermilab.

Используемый в ATLAS Fast TracKer чип AM представляет собой эволюцию конструкции CDF [43]. Требования к приложению LHC выше, чем требования к CDF: более мощный кремниевый детектор с большим количеством каналов требует большего количества паттернов, а более высокая частота запуска требует более высокой рабочей частоты при сохранении общей потребляемой мощности.

Последняя версия чипа AMchip06 [44] производит 1 млн. сравнений каждые 10 наносекунд и имеет следующие характеристики: объем банка паттернов

– 128k, объем памяти – 19Mb на чип, частота – 100 Mhz, энергопотребление 3 Вт, технология - 65nm, размер – 4 см × 4 см.

1.4.3. Архитектура и принцип работы системы FTK

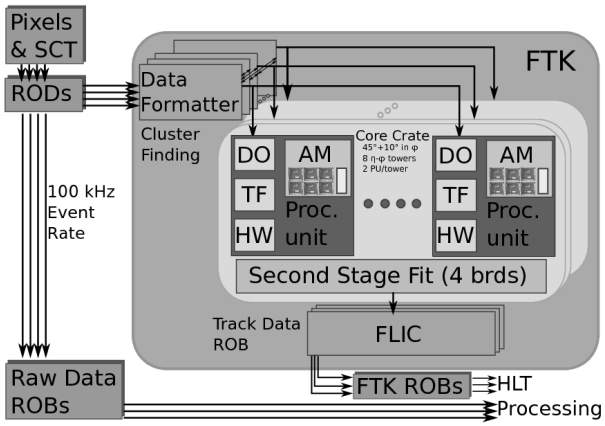


Рис. 1.6. Структура FTK

Система FTK [45, 46] включает несколько подсистем, выполняющих разные функции (рис. 1.6):

- 32 платы форматирования данных (Data Formatter, DF) и встроенные дополнительные платы (RODs), принимающие данные с пиксельных и полосковых датчиков детектора;
- 128 независимых процессоров, состоящих из ассоциативного процессора (AMBSLP: 4 локальных платы с 16 чипами AM) и auxiliary cards (AUX) на основе FPGA.
- 32 платы для второй стадии реконструкции треков (Second Stage Fit Boards, SSB)
- 2 FTK to Level-2 Interface Card (FLIC)

ROD получает данные от 4-х микростриповых датчиков с общим трафиком 500 Гб/с и выполняет кластеризацию данных [47]. После этого DF реорганизует данные для дальнейшей обработки. AUX выполняет два алгоритма: организацию данных (DO) и реконструкции треков (TF). При организации данных,

полученных от DF, формируются суперстрипы меньшей размерности (данные с 4-х пиксельных и 4-х микростриповых датчиков) для сопоставления с паттернами на AM-чипах. Для отобранных данных AUX выполняет реконструкцию треков. Далее полученные треки поступают на повторную реконструкцию в SSB: каждый трек дополняется данными с 4-х микростриповых датчиков, не использованными в сопоставлении с паттернами. FLIC заменяет в полученных треках локальные идентификаторы FTK на глобальные идентификаторы ATLAS и отправляет их через систему считывания данных на тригер высокого уровня (HLS).

1.4.4. Области применения процессоров FTK

Система, в первую очередь, разработана для детекторов физики высоких энергий. Она используется для проведения экспериментов на LHC на детекторах ATLAS и CMS [48].

Но также данная система может быть адаптирована для использования в более общих приложениях обработки изображений [44]:

- медицинская визуализация (томография и т. д.);
- системы видео-наблюдения, смарт-камеры;
- задачи высокоскоростной фильтрации данных;
- распознавание форм;
- изучение зрения и других функций мозга.

1.5. Модели ASC и MASC

ASC-модель (Associative Computing Model) была разработана на основе ассоциативных параллельных процессоров STARAN и ASPRO. Модель имеет следующие свойства 1.7:

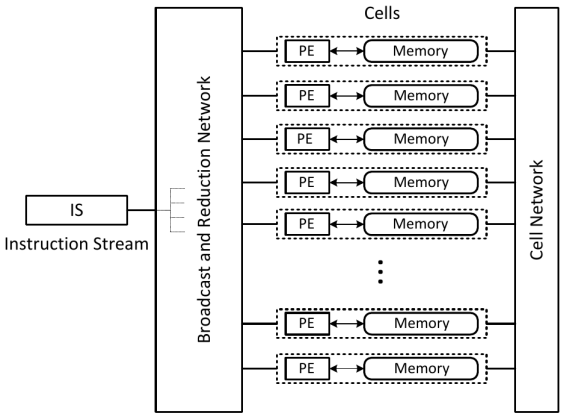


Рис. 1.7. Модель ASC

- **Свойства ячеек.**(Cells) состоящие из процессорного элемента (ПЭ) и локальной памяти: каждый ПЭ имеет доступ только к памяти своей ячейки; процессорные элементы могут быть связаны сетью между собой (Cell Network). Связанные элементы данных группируются (ассоциируются) в запись и хранятся однообразно в каждой ячейке. Предполагается, что ячеек больше, чем данных.

- **Свойства потока инструкций.** IS(Instruction Stream) поток инструкций - процессор, связанный через BR-сеть (Brodcast and Reducation Network) с каждой ячейкой. IS хранит копию выполняемой программы и передает инструкцию всем ячейкам за единицу времени. Активная ячейка выполняет команды и отсылает результат IS, в то время как неактивная ячейка принимает команду, но не исполняет ее.

- **Ассоциативные свойства.** IS может передать инструкцию всем активным ячейкам выполнить ассоциативный поиск. Ячейки, для которых поиск выполнен успешно, называются реагирующими (responder), в отличии от не реагирующих ячеек (non-responder). IS может активировать или множество реагирующих ячеек, или множество не реагирующих ячеек. IS также может восстановить предыдущее множество активных ячеек. Каждая из этих операций выполняется за единицу времени.

- **Константное время выполнения глобальных операций.** IS выполняет операции OR и AND со значениями всех активных PE за единицу времени. IS также выполняет операции minimum и maximum за константное время.

1.5.1. Операции с константным временем выполнения.

Параллелизм по данным используется при ассоциативном поиске, который выполняется за время, пропорциональное числу битов в поле, а не числу элементов данных, среди которых производится поиск. Предполагается, что все данные помещаются в память компьютера, и за константное время вычисляются сравнение, сложение и другие параллельные арифметические операции. А также базовые операции поиска ($=$, $<$, $\leq$, $>$, $\geq$, min, max). Функции $maxdex$, $mindex$, $prvdex$ и $nxtdex$, возвращающие индексы ассоциативных ячеек для соответствующих функций за константное время, используются для ассоциативной редукции. Запрос "salary[maxdex(age$)]"выражает связь между максимальным возрастом и зарплатой. Операции $mindex$, $prvdex$ и $nxtdex$ выполняются аналогично.

1.5.2. Табличная структура данных и способы кодирования структур.

Табличные структуры данных (такие как таблицы, массивы) имеют два преимущества для ASC. Во-первых, они повсеместно используются. Во-вторых, одновременную обработку целого столбца таблицы легко понять.

Другие общепринятые структуры данных (стек, очередь, деревья и графы) обычно представляются с помощью индексов и указателей. Ассоциативный процессор не использует физические адреса переменных, но используя дополнительные признаки, можно представить эти структуры в табличном виде. Например, для представления вектора используются два поля: "row"и "value". Для представления структур очереди и стека задаются поля: "time"и "value". Для реализации очереди выполняется запрос "value[mindex(time$)]". Для реализации стека — "value[maxdex(time$)]".

1.5.3. Модель MASC

Модель MASC (рисунок 1.8) является расширением модели ASC: к ячейкам ассоциативных вычислений может быть подключено несколько IS. Эта модель относится к классу MIMD.

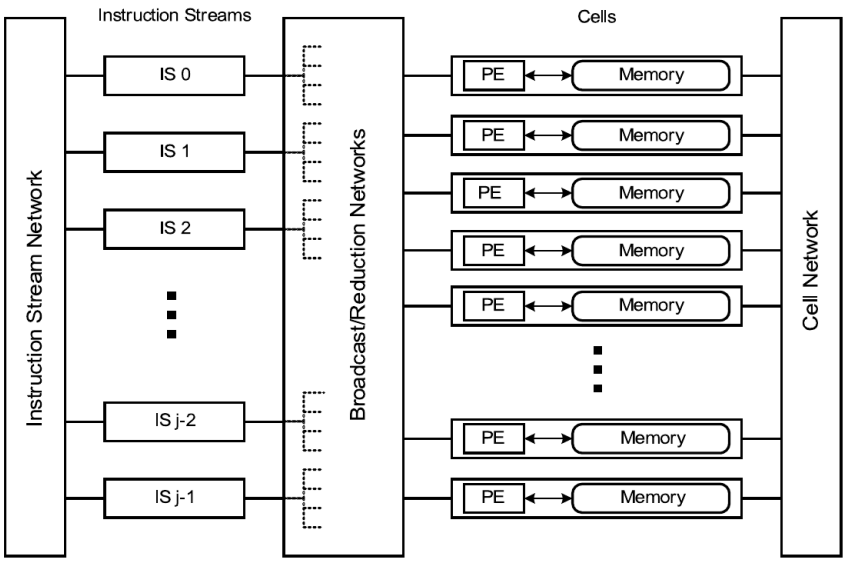


Рис. 1.8. Модель MASC

К описанным выше свойствам добавляются следующие свойства параллелизма контроля:

- Бездействующие ячейки передаются управляющему IS (без других задач). Бездействующие ячейки могут быть динамически выделены IS за единицу времени. Любое подмножество ячеек может быть переназначено как бездействующее за константное время.

- Если IS, выполняющему задачу, необходимо выполнить две или более подзадачи, допускающих разделение данных на непересекающиеся множества активных ячеек,то каждой подзадаче можно назначить бездействующий IS. После выполнения всех подзадач, ячейки возвращаются под управление исходного IS.

1.5.4. Реализации моделей ASC и MASC

Meiduo Wu и Yanping Wang [49, 50] разработали первую версию процессора ASC. Этот первый процессор имел 8-битный блок управления, 4 8-битные PE и сеть BR, способную осуществлять максимальный/минимальный поиск, обнаружение ответчика. Набор команд был смоделирован на RISC-процессорах, таких как MIPS и DLX, с ассоциативными функциями, аналогичными STARAN. Процессор был ориентирован на FPGA Altera FLEX 10K70. Однако окончательный дизайн не соответствовал FPGA, и поэтому он никогда не был полностью реализован.

Hong Wang [51] позже улучшил первый процессор, сделав его масштабируемым, так что количество PE могло легко варьироваться. Чтобы сделать процессор масштабируемым, была перепроектирована BR-сеть, так как сеть в первом процессоре могла обрабатывать только четыре PE. Существенных изменений в архитектуре набора инструкций не произошло, поэтому масштабируемый процессор мог запускать большинство программ, написанных для первого. Новый масштабируемый процессор затем был реализован в Altera APEX 20K1000 FPGA, которая может содержать блок управления до 50 PE.

Позднее Hong Wang [52] и Lei Xie добавили в процессор сеть, соединяющую PE. Сеть может работать либо как линейный массив 1D, либо как 2D-сетка. Затем процессор с сетью был протестирован в ряде приложений, включая сопоставление строк и обработку изображений.

Hong Wang [53] также разработал первый процессор для реализации модели MASC, добавив несколько блоков управления в процессор ASC. Этот процессор способен динамически разбивать массив PE и назначать управление различными PE на разные блоки управления. Это позволяет процессору использовать как параллелизм управления, так и параллелизм данных. Процессор MASC был реализован на FPGA Altera APEX 20K1000 с тремя блоками управления и 9 PE.

Кроме разработки процессора для реализации моделей, язык ASC эмулировался на ПК и рабочих станциях неассоциативного типа. Одной из последних работ в этом направлении была попытка реализовать конструкции языка ASC

на графических ускорителях [21]. Но результаты реализации ASC-алгоритмов на GPU на данный момент в публикациях не представлены.

1.6. Модель STAR

Параллельно и независимо от модели ASC разрабатывалась другая модель ассоциативных вычислений. STAR-машина — абстрактная модель типа SIMD (Single Instruction Multiple Data) с вертикальной обработкой данных была представлена в работах [54, 55]. Модель (рис. 1.9) была разработана на основе процессоров STARAN и отечественного АПП ЕС-1020. Она состоит из следующих частей:

- последовательного устройства управления(ПУУ), в котором хранятся инструкции;
- устройства ассоциативной обработки (УАО), состоящего из p процессорных элементов (ПЭ связаны сетью все-со-всеми и с ПУУ);
- матричной (табличной) памяти.

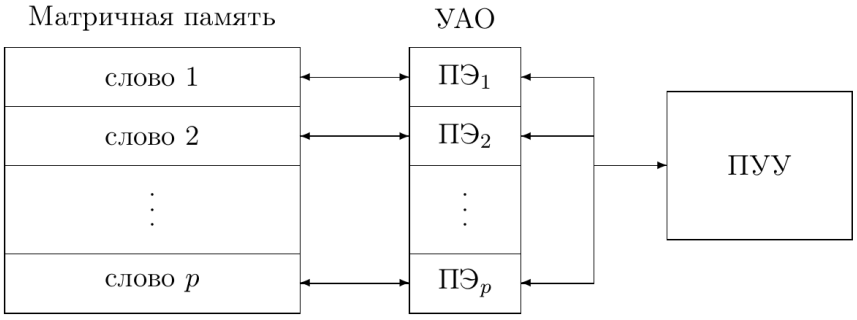


Рис. 1.9. Модель STAR-машины

Хотя структуры и свойства моделей совпадают, но эти модели отличаются исполняемыми операциями.

Если разработчики модели ASC больше сосредотачивались на аппаратной реализации модели, то для STAR-машины больший упор ставился на разработку языка и ассоциативных алгоритмов.

1.6.1. Язык STAR

Для STAR-машины в работах [54, 55] был разработан язык высокого уровня Star, похожий на язык Pascal.

Для моделирования обработки информации в матричной памяти, в языке Star имеются три типа данных: **slice** (далее слайс), **word** (слово) и **table** (таблица). С помощью переменной типа **slice** моделируется доступ к таблице по столбцам, а с помощью переменной типа **word** — доступ по строкам. С каждой переменной типа **table** ассоциируется бинарная таблица из n строк и k столбцов, где $n \leq p$.

Вначале приведем основные операции для переменной типа **slice**. Пусть X и Y — слайсы, i — целочисленная переменная. Тогда операции имеют следующий вид:

$SET(Y)$ записывает в слайс Y все единицы;

$CLR(Y)$ записывает в слайс Y все нули;

$Y(i)$ выделяет i-ю компоненту в слайсе Y ($1 \leq i \leq n$);

$FND(Y)$ выдает порядковый номер i позиции первой (или старшей) единицы в слайсе Y, где $i \geq 0$;

$STEP(Y)$ выдает такой же результат, как и операция $FND(Y)$ и затем обнуляет старшую единицу (эта операция используется для построения неравномерных циклов);

$NUMB(Y)$ выдает число компонент ‘1’ в слайсе Y;

$CONVERT(Y)$ выдает слово, i-й бит которого совпадает с $Y(i)$;

$FRST(Y)$ сохраняет первую (старшую) единицу в слайсе Y и обнуляет остальные компоненты.

Общепринятым способом вводятся предикат $SOME(Y)$, а также следующие побитовые логические операции: $X\,and\,Y$, $X\,or\,Y$, $not\,X$, $X\,xor\,Y$.

Для переменных типа **word** выполняются те же операции, что для переменных типа **slice**.

Для переменных типа **table** определены следующие две операции:

$COL(i, T)$ обеспечивает доступ к i-му столбцу таблицы T для выполнения чтения и записи;

$ROW(i, T)$ обеспечивает доступ к i-й строке таблицы T для выполнения чтения и записи.

В последнюю версию языка были добавлены следующие операции для переменных типа **word**:

$TRIM(i, j, w)$ возвращает подстроку слова w с i-й по j-й биты, где $1 \leq i < j \leq |w|$;

$REP(i, j, v, w)$ замещает подстроку $w(i)w(i+1)\ldots w(j)$ слова w на слово v, где $|v| = j - i + 1$ и $1 \leq i < j < |w|$.

1.6.2. Библиотека стандартных процедур

В библиотеки базовых ассоциативных процедур приведены универсальные алгоритмы для ассоциативной обработки данных. Библиотека включает:

- процедуры для нечисловой обработки,
- процедуры для числовой обработки,
- процедуры копирования данных.

Процедуры для нечисловой обработки данных.

Эти процедуры используют управляющий слайс (битовый столбец), в котором отмечены позиции анализируемых строк соответствующей матрицы.

Процедуры для нахождения позиции строк заданной матрицы T, в которых записан минимальный / максимальный элемент: MIN(T, X, Z) и MAX(T, X, Z), соответственно.

Процедуры, которые по заданной матрице T, по образцу v и по управляющему слайсу X находят позиции строк, удовлетворяющих условию поиска:

- MATCH(T, X, v, Z) для ROW(j, T)=v;
- LESS(T, X, v, Y) для ROW(j, T)<v;
- GREAT(T, X, v, Y) для ROW(j, T)>v.

Процедура GEL(T, v, Y, Z) обобщает процедуры LESS и GRAET: в слайсе Y отмечены позиции строк, которые больше образца, а в слайсе Z – меньше образца.

Процедуры, которые по двум заданным матрицам T и F и по управляющему слайсу X находят позиции строк, удовлетворяющих условию поиска:

- HIT(T, F, X, Z) для ROW(j, T)=ROW(j, F),
- SETMIN(T, F, X, Z) для ROW(j, T)<ROW(j, F) и
- SETMAX(T, F, X, Z) для ROW(j, T)>ROW(j, F).

Процедуры для числовой обработки.

Процедуры ADDV(T, F, X, R) для построчного сложения двух матриц и ADDC(T, X, v, R) добавления к строкам матрицы T двоичного слова v (строки матрицы R, которые отмечены ‘0’ в слайсе X, состоят из нулей).

Процедура ADDC1(T, X, v, R) отличается от ADDC тем, что в строки матрицы R, которые отмечены ‘0’ в слайсе X, записывает соответствующие строки исходной матрицы T.

Аналогично определены процедуры SUBTV(T, R, X, R) для построчного вычитания из одной матрицы другую и процедур вычитания из строк матрицы T двоичного слова v SUBTC(T, X, v, R), SUBTC1(T, X, v, R).

Процедуры копирования.

Процедура CLEAR(n, T) записывает нули во все n столбцов матрицы T.

Процедура WCOPY(v, X, F) записывает двоичное слово v в те строки матрицы F, которые отмечены ‘1’ в слайсе X. Остальные строки матрицы F состоят из нулей.

Процедура WMERGE(v, X, F) записывает двоичное слово v в те строки матрицы F, которые отмечены ‘1’ в слайсе X, при этом остальные строки матрицы F не меняются.

Процедура TCOPY(T, F) копирует все столбцы матрицы T в соответствующие столбцы матрицы F. Размеры матриц совпадают.

Процедура TCOPY1(T, j, h, F) выделяет из матрицы T j-ю полосу шириной h и копирует ее в матрицу F.

TCOPY2(T, j, h, F) копирует матрицу T в j-ю полосу матрицы F.

Процедура TMERGE(T, X, F) записывает строки матрицы T, отмеченные ‘1’ в слайсе X, в соответствующие строки матрицы F. Остальные строки матрицы F не меняются.

1.7. Выводы к первой главе

Хотя ассоциативные параллельные системы со времени своего появления не стали широко распространенными, они не утратили своей актуальности. Все реализованные архитектуры были построены под решение конкретных задач, которые не могли быть эффективно решены на системах другой архитектуры: ASPRO для задач контроля воздушного движения, IMX2 для машинного перевода, FastTracker Processor для детектора ATLAS большого адронного коллайдера.

Для использования в проекте ATLAS [32] большого адронного коллайдера были разработаны новые устройства ассоциативной памяти [33, 43]. ATLAS Fast Tracker (FTK) – крупнейшая ассоциативная система из 8 192 чипов ассоциативной памяти. Разработчики планируют ассоциативный процессор, спроектированный для ATLAS FastTracKer, сделать пригодными для портативных устройств, а также использовать его для решения задач физики высоких энергий, медицинской визуализации, изучения визуальных функций мозга [44, 56].

Развитие продолжается в трех направлениях: разработка аппаратного обес-

печения для ассоциативных параллельных вычислений (чипы ассоциативной памяти [57, 58], ассоциативные процессоры и системы [44, 59, 60]), разработка ассоциативных моделей и алгоритмов для этих моделей, реализация ассоциативных моделей на существующем оборудовании [20, 21, 61, 1].

Глава 2

Реализация модели ассоциативных параллельных вычислений на GPU

В этой главе рассмотрим реализацию модели ассоциативных вычислений на графических ускорителях. Заметим, что реализация отличается от эмуляции работы архитектуры. Эмуляция предполагает воспроизведение поведения моделируемой архитектуры. В то время как при реализации модели необходимо не только воспроизвести поведение модели, но и максимально сохранить ключевые свойства.

Поэтому в разделе 2.1 приводятся отличительные свойства ассоциативных архитектур. В разделе 2.2 обосновывается выбор графических ускорителей в качестве архитектуры реализации и приводятся ключевые моменты реализации. Также приводится теоретическая оценка моделирования и оценка производительности в качестве доказательства того, что свойства ассоциативной архитектуры моделируются эффективно. В разделе 2.3 даются рекомендации для оптимизации ассоциативных алгоритмов для выполнения на графических ускорителях.

2.1. Ключевые отличия ассоциативных систем от PRAM-архитектур

Ниже перечислим свойства ассоциативных параллельных процессоров, отличающих их от других типов архитектур. Более подробно свойства ассоциативных архитектур приведены в 2.2.

1. Синхронизация операций всех ячеек, как активных, так и неактивных.

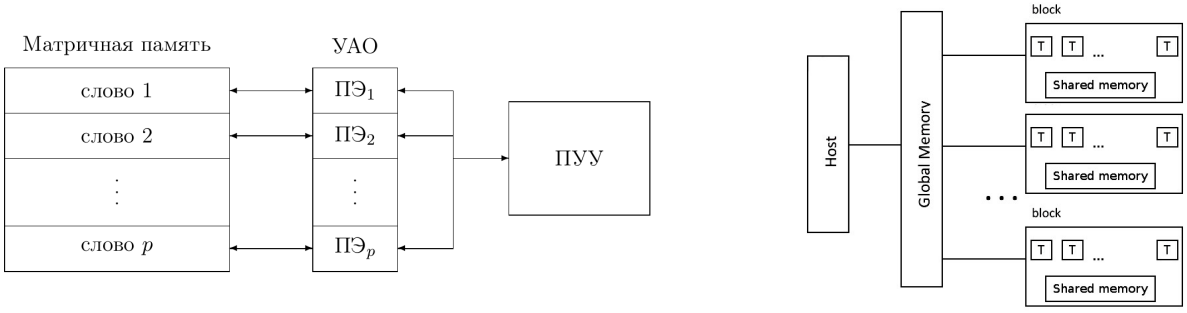
2. Константное время выполнения глобальных операций: побитовые логические операции, доступ к столбцам и строкам матричной памяти как на чтение, так и на запись.

3. Ассоциативные свойства: (следующие операции выполняются за единицу времени)

   - передача данных выбранной ячейкой всем остальным;
   - выбор старшей ячейки из множества активных ячеек.

Эти свойства позволяют производить базовые операции поиска ($=$, $<$, $>$, min, max) и арифметические операции за время, не зависящее от числа строк. С другой стороны, необходимость частых синхронизаций и передачи данных от одного процессора всем остальным приводит к тому, что ассоциативные архитектуры нельзя эффективно реализовать на кластерных системах.

2.2. Реализация STAR-машины на графических ускорителях



($а$) Модель STAR-машины

($б$) Модель графических ускорителей

Рис. 2.1. Сравнение моделей

Как уже упоминалось, для STAR-машины разработано большое число ассоциативных алгоритмов для решения задач на графах. Для их использования необходима реализация STAR-машины на современных параллельных архитектурах. В качестве такой архитектуры были выбраны графические ускорители фирмы NVidia, поскольку они относятся к классу $MIMD \supseteq SIMD$ (2.1) и получают все более широкое распространение как в кластерных системах, так и в персональных устройствах.

В работе [10] Поттер выделил основные свойства ассоциативных параллельных моделей и архитектур, которым, в частности, удовлетворяет Star-машина. Графические ускорители дают возможность смоделировать данные свойства частично на аппаратном, а частично на программном уровне. Перечислим соответствующие свойства и способ их моделирования на GPU.

1. Свойства ячеек памяти:
   память ассоциативного компьютера состоит из массива ячеек (процессорного элемента ПЭ и слова в матричной памяти); каждый ПЭ имеет доступ только к памяти своей ячейки; ячеек больше, чем данных.
   Каждый ПЭ моделируется отдельным потоком вычислений в блоке, слово матричной памяти моделируется элементом массива данных в глобальной памяти (индекс элемента в массиве выражается через индекс потока/блока).

2. Свойства последовательного устройства управления (ПУУ): ПУУ представляет собой процессор, связанный шиной с каждой ячейкой; ПУУ передает инструкцию всем ячейкам за единицу времени; активные ячейки выполняют команду, полученную от ПУУ, в то время как неактивные ячейки принимают команду, но не выполняют ее; ПУУ может активировать ячейки.
   На графический ускоритель инструкции от процессора поступают пакетом (через вызов ядра). В ядрах предусмотрена возможность синхронизации по потокам одного блока, но возможность синхронизации по блокам отсутствует. При необходимости синхронизации вычислений инструкции должны быть разнесены в разные ядра. При этом накладные расходы на запуск ядер занимают около $15 - 20\mu c$.

3. Константное время выполнения глобальных операций: побитовые логические операции, доступ к столбцам и строкам матричной памяти как на чтение, так и на запись и другие базовые операции языка Star [54].
   В главе 2.2.2 будет приведена эффективная реализация базовых операций языка Star и показано, что базовые операции, не требующие синхронизации по данным, выполняются за константное время.

4. Ассоциативные свойства.

   - ПУУ может дать команду выбранной ячейке передать данные по шине; все другие ячейки, принимающие команды от этого ПУУ, получают значение с шины за единицу времени.
     Только несколько операций языка Star нуждаются в передаче данных. При этом возможно два варианта такой передачи: через копирование данных на процессор или через передачу указателя на данные.
   - ПУУ может выбрать старшую ячейку из множества активных ячеек за единицу времени.
     Реализация данной операции критична к синхронизации по данным. В главе 2.2.2 будет приведена реализация этой базовой операции с временной сложностью $O(\lceil \log_{64}(N) \rceil)$, где N - число битовых строк таблицы.
   - Базовые операции поиска (=, <, , >, , min, max) и арифметические операции выполняются за время, пропорциональное числу битовых столбцов в таблице, а не числу ее строк.
     Данные операции производятся процедурами библиотеки стандартных процедур языка Star [55]. Особенности их реализации и обоснование оценки временной сложности будут приведены в главе 2.2.3.

Таким образом будет показано, что с помощью графических ускорителей можно реализовать Star-машину с сохранением всех ее основных свойств.

2.2.1. Форматы входных и выходных данных

Как уже отмечалось ассоциативные алгоритмы используют преимущественно обработку таблиц по столбцам, поэтому для использования в реализации было выбрано представление данных, отличающееся от стандартных.

На рисунке 2.2 показаны представление структуры vector<int> из 64-х элементов для библиотек STL и CUDA thrust и представление этих же данных для реализации Star–машины. В структуре vector<int> каждое из целых чисел

Рис. 2.2. Представление данных

массива представляет собой 32-х разрядную бинарную строку. Для использования в реализации Star–машины такое представление не подходит, поскольку обработка происходит не по строкам, а по столбцам. Поэтому каждый столбец из 64-х битов составляет одно 64-х разрядное целое число. За счет такого вертикального представления данных пустые разряды можно не включать в таблицу для экономии памяти и уменьшения времени работы.

Такое внутреннее представление подразумевает перевод данных в более привычные форматы ввода/вывода:

- двоичный формат:
  - переменные типа **Word** или **Slice** могут вводиться/выводится как битовая строка или битовый столбец;
  - переменные типа **Table** выводятся построчно;
- десятичный формат для переменной типа **Table**:
  - целочисленный вектор;
  - целочисленная матрица.

Отдельно отметим форматы ввода/вывода графов. Во-первых, Star-машина использует следующие представления графов:

- 2 бинарные таблицы размером $n \times h$ как список ребер для невзвешенных графов и три бинарные таблицы для взвешенных;

- бинарная таблица размером $n \times n$ ($n \times hn$) для представления матрицы смежности графа и бинарная таблица размером $n \times hn$ для кодирования матрицы весов.

Формат данных *.gr

Данный формат широко используется для представления больших графов, например графов дорог [62] или генераторе R-MAT графов GraphHPC-1.0 [63], моделирующих реальные графы из социальных сетей и Интернета. Эти графы часто используются в качестве тестовых данных для оценки производительности параллельных алгоритмов на графах.

Первым символом в строке может быть

c: строка используется для комментария

p: в строке задается число вершин и дуг графа

a: в строке задается дуга с весом

Пример первых строк файла данных в этом формате приведен в листинге 2.1.

```
1 c rmat-10
2 p sp 1024 32768
3 c graph contains 1024 nodes and 32768 arcs
4 a 1 148 336
5 a 1 240 718
```

Листинг 2.1. Пример данных в формате gr

Формат данных проекта Snap

Для тестирования и проверки производительности алгоритмов на графах кроме графов дорог и синтетических графов часто используются графы Автономных Систем, графы интернет-ресурсов и социальных сетей.

В базе данных Snap [64] эти данные представляются в следующем формате:

- символ '#' используется для комментирования данных;

- каждая строка содержит номера вершин, задающих дугу.

Для неориентированных графов ребро сохраняется дважды как ij и ji. В листинге 2.2 приведены первые строки файла as19971108.txt из архива as-733.

```
1 # Undirected graph (each pair of nodes is saved twice):
    as19971108.txt
2 # Autonomous systems (from the BGP logs) - 11 08 1997
3 # Nodes:3015    Edges: 5347
4 # FromNodeId    ToNodeId
5 1   1740
6 1   5413
7 1   701
8 1   286
9 1   1239
```

Листинг 2.2. пример данных из архива as-733

2.2.2. Представление типов данных и базовых операций для них

Как было отмечено выше, для доступа к матричной памяти в STAR-машине используются типы данных **word**, **slice** и **table**.

Так как CUDA C является расширением языка C++, то для реализации типов данных используются классы Slice[1] и Table. Класс Slice используется и для типа **word**, поскольку базовые операции над этими типами одинаковые.

Ниже описано представление этих типов на языке CUDA C++.

```
1 class Slice{
2 // основная память CPU
3  /* length - длина слайса в битах;
4     N - количество 64-х разрядных элементов для хранения
    слайса */
5  unsigned int length,N;
6 // глобальная память GPU
7 /* *d_v хранит адрес первого элемента массива длины $N$ */
8  Unsigned long long int *d_v;
```

[1] Чтобы избежать путаницы между языком Star и его реализацией на CUDA C, типы языка Star выделяются полужирным шрифтом.

```
9 ...}
```

Здесь переменная N равна числу 64-разрядных переменных, необходимых для хранения битового столбца длины $length$. Битовый столбец хранится в массиве d_v из N переменных типа *unsigned long long int*, расположенном в глобальной памяти GPU.

Класс Table содержит массив из объектов класса Slice и несколько указателей.

```
1 class Table{
2 /*length - число строк, size - число столбцов */
3     unigned int length,size;
4     Slice table[size];
5     LongPointer *slice_device_pointer_table;
6 // глобальная память GPU
7     LongPointer *d_table, *d_slice_device_pointer_table;
8 ...}
```

Указатель на GPU $*d\_table$ хранит адрес первого элемента массива $length \times N$ элементов типа *unsigned long long int*, расположенного в глобальной памяти GPU. Указатель на CPU $*slice\_device\_pointer\_table$ и указатель на GPU $*d\_slice\_device\_pointer\_table$ хранят адреса первых элементов массивов из $size$ указателей на столбцы d_table. С помощью такой системы указателей осуществляется доступ как целиком к таблице, так и к каждому столбцу в отдельности.

Базовые операции для типа Slice

Базовые операции над переменными типа **slice** реализованы как одноименные методы класса Slice. В свою очередь, каждый метод запускает функцию на GPU, исполняющую операцию. Таким образом, все вычисления производятся на графическом ускорителе параллельно.

Ниже приводятся реализации базовых операций. Отметим, что при операции присваивания указателей на объект класса возможна неоднозначность. Чтобы ее избежать, побитовые логические операции реализованы как совмещенные с присваиванием, аналогично подобным операциям языка C++. Т. е.

$Z := X\,and\,Y$ записывается как $Z = X$; $Z.AND(Y)$.

Операция *and* реализована следующим образом:

```
1 void Slice::AND(Slice *Y)
2 {    and_long_values<<<N,1>>>(d_v,Y->d_v);}
3 __global__ void and_long_values( *d_v, *d_v1)
4 {    d_v[blockIdx.x] &= d_v1[blockIdx.x];}
```

Остальные побитовые логические операции, а также операции $SET(X)$ и $CLR(X)$ реализованы аналогично.

Заметим, что каждый из N блоков вычисляет свой элемент массива независимо от других. Поэтому время выполнения этих операций не зависит от длины битового столбца **slice**.

Операция $MASK$ реализована следующим образом:

```
1 void Slice::MASK(int i)
2 {   set_mask_values<<<NN,1>>>(d_v,i);}
3 __global__ void set_mask_values(unsigned long long int *d_v
     , int num)
4 { unsigned long long int mask_el;
5   int num_el=num>>6; // номер элемента, содержащий переход
     от 0 к 1;
6   if (blockIdx.x==num_el)
7       mask_el=1>>(num % SIZE_OF_LONG_INT)-1;
8   else {
9       mask_el=0;
10      if (blockIdx.x>num_el)
11          mask_el=!mask_el;
12  }
13   d_v[blockIdx.x]=mask_el;
14 }
```

Время выполнения операции $MASK$ также не зависит от длины битового столбца **slice**.

Операция $FND(X)$ выполняется в два этапа.

- На первом этапе с помощью глобальной процедуры *find* массив $d\_v[N]$ за несколько шагов сворачивается до одной переменной типа *unsigned long*

long int. На каждом шаге массив сворачивается по следующему правилу:

- если $dv[level, i] \neq 0$, то в i-й бит элемента массива $dv[level + 1]$ устанавливается ‘1’;
- если $dv[level, i] = 0$ или $i > N$, то в i-й бит элемента массива $dv[level+1]$ устанавливается ‘0’.

Таким образом, за шаг свертки длина массива $dv[level + 1]$ уменьшается в 64 раза и $dv[level+1]$ — битовая маска текущего массива $dv[level]$. Свертка заканчивается, когда в массиве $dv[level + 1]$ остается один элемент.

- На втором этапе глобальная процедура $first\_backward$ вычисляет результат, разворачивая свертку.

```
1  __global__ void first_backward(LongPointer *d_v, int *
       d_first_non_zero, int level)
2  {
3      int f[LEVELS];
4      unsigned long long int *dvl,u;
5      char lprt[100];
6      f[level+1] = 1;
7      while(level >= 0)
8      {
9          dvl = d_v[level];
10         int index = f[level+1]-1;
11         u=dvl[index];
12         f[level]=__ffsll(u)+index*SIZE_OF_LONG_INT;
13         level--;
14     }
15     *d_first_non_zero = f[0];
16 }
```

Листинг 2.3. first_backward

Время выполнения операции $FND(X)$ оценивается как $O\left(\log_{64}(N)\right)$, где N — длина массива данных объекта класса Slice. Зависимость количества уровней свертки l_{max} от длины битового слайса N^* представлена в таблице 2.1. Таким

Таблица 2.1. Оценка глубины свертки

N^*	64	4 096	16 384	65 536	4 194 304	16 777 216	1 073 741 824
$log_{64}(N)$	0	1	2	3	4	5	6

образом можно заключить, что хотя оценка теоретической сложности и не является константной, но вполне приемлема.

Операция $STEP(X)$ реализована на базе операции FND следующим образом. Пусть $i = FND(X)$ и $i > 0$. Тогда в операции $STEP(X)$ i-й бит обнуляется, а в качестве результата возвращается число i. Если $FND(X) = 0$, то результат $STEP(X)$ также равен нулю.

Для вычисления предиката $SOME(X)$ достаточно сравнить с нулем результат свертки после первого этапа вычисления FND. Операция $X = FRST(Y)$ реализуется следующим образом: $CLR(X)$; $i := FND(Y)$; $X(i) := 1$.

Для вычисления $NUMB(X)$ используется модификация алгоритма подсчета количества битов из книги [65].

```
1 m[6]={0x5555555555555555,
2        0x3333333333333333,
3        0x0f0f0f0f0f0f0f0f,
4        0x00ff00ff00ff00ff,
5        0x0000ffff0000ffff,
6        0x00000000ffffffff};
7 for(int n=0; n<6; n++) x=(x&m[n])+((x>>(1<<n))&m[n])
```

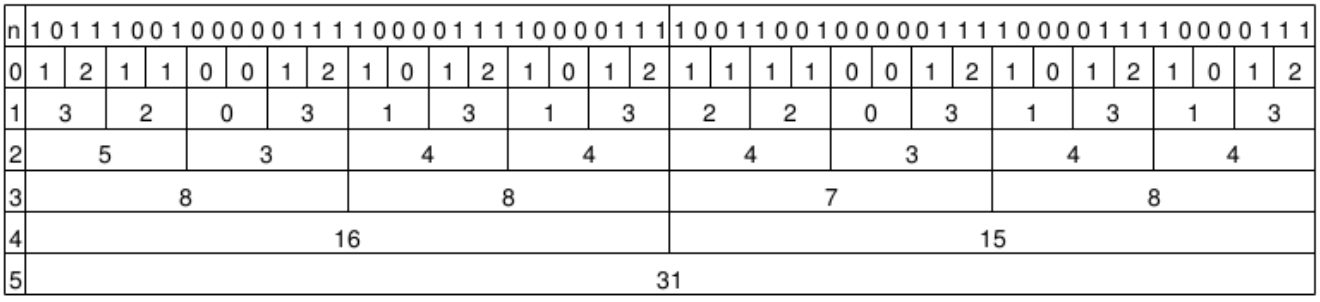


Рис. 2.3. Подсчет числа единичных битов в 64-х битовом слове.

Покажем работу алгоритма на примере подсчета единичных битов в одном 64-х разрядном беззнаковом целом числе (рис. 2.3).

Заметим, что на шаге n получаются слова, состоящие из 2^{n+1} бит в бинарной записи, и со значением не больше 2^{n+1}. При этом максимальное значение для бинарных слов длины 2^{n+1} равняется $2^{2^{n+1}} - 1$. Поскольку нам необходимо найти количество единиц в N числах, то можно делать свертку массива (редуцированную сумму). В зависимости от количества элементов в исходном массиве свертка делается два раза: при $n = 2$ не более чем в 16 раз и при $n = 5$ не более чем в 128 раз. Если после завершения процедуры подсчета количество элементов выходного массива больше единицы, то от массива вычисляется редуцированная сумма.

Операция $TRIM(i, j, w)$ реализована следующим образом:

```
1  Slice::TRIM(int i, int j, Slice *w)
2  {   trim_long_values<<<N,1>>>(d_v,i,j,w->d_v);}
3  __global__ void trim_long_values(*d_v, int i, int j, *d_v1)
4  {   unsigned long long int head, tail;
5      int k, n, index;
6      index=blockIdx.x;
7  // сдвиг по битам
8      k = i % SIZE_OF_LONG_INT -1;
9  // сдвиг по элементам
10     n=(k==0)?(i/SIZE_OF_LONG_INT-1):(i/SIZE_OF_LONG_INT);
11     head = d_v1[n+index]>>k;
12     tail = d_v1[n+index+1]<<(SIZE_OF_LONG_INT-k);
13     head =head|tail;
14     d_v[index] = head;
15 }
```

Время выполнения операции $TRIM$ не зависит от длины битового столбца **slice**.

Отметим, что операция $CONVERT$ не нуждается в реализации, поскольку класс *Slice* используется для представления как слайсов, так и слов.

Замечание 2.2.1. *Базовые операции языка Star над слайсами выполняются на графическом ускорителе или за константное время, или за время* $O\left(\log_{64}(N)\right)$.

Базовые операции для типа table

Операции над типом **table** обеспечивают доступ к столбцам и строкам таблицы для выполнения чтения и записи.

Операция $COL(i, T)$ выполняется вызовом метода T.col(i):

```
Slice * Table::col(int i) {return &(table[i-1]);}
```

Он обеспечивает доступ к столбцу как на чтение, так и на запись.

Для реализации операции $ROW(i, T)$ используются два метода класса *Table*. Метод *SetRow(Slice *s, int i)* записывает слово s в i-ю строку таблицы T. Метод *GetRow(Slice *s, int i)* записывает i-ю строку таблицы в слово s. Процедуры вычисляются на $s.N$ блоках по 64 нити, где $s.N$ — число битовых элементов в слове s. Поэтому каждый блок производит вычисления над своим элементом слова, а нити — над своим столбцом.

Напомним, что переменная T типа **table** представляется в глобальной памяти графического ускорителя массивом слайсов. Слайсы, в свою очередь, представляются массивами 64-х разрядных элементов. Поэтому для доступа к j-му биту i-й строки необходимы следующие индексы:

- $n\_col = \lceil i/64 \rceil$ — номер элемента массива в столбце;
- $b\_col = i \bmod 64$ — номер бита в этом элементе.
- $n\_row = blockIdx.x$ — номер элемента в слове (равен номеру блока);
- $b\_row = threadIdx.x$ — номер бита в элементе слова (равен номеру нити в блоке);
- $k = 64 \cdot blockIdx.X + threadIdx.x$ — номер столбца.

В качестве вспомогательных переменных используется массив $tmp[64]$ в разделяемой памяти графического ускорителя (у каждого блока свой экземпляр массива, к которому имеют доступ все нити блока) и col — указатель на k-й столбец.

Тогда запись слова s в таблицу происходит следующим образом.

```
1 if (s(k)==1){
2    tmp[b_row]=1<<(b_col-1);
3    col[n_col]|=tmp[b_row];
4 }else{
5    tmp[b_row]=~(1<<(b_col-1));
6    col[n_col]&=tmp[b_row]
7 }
```

При чтение строки таблицы элементы массива *tmp* для каждого блока вычисляются следующим образом.

```
1    bit=(col[n_col]>>(b_col-1))&1;
2    tmp[b_row]=bit<<(b_col-1);
3    s[n_row]=tmp[0]|...|tmp[63];
```

Для каждого блока вычисляются 64 элемента массива *tmp*. Отметим, что в элементе $tmp[bit\_row-1]$ бит с номером b_row совпадает с i-м битом k-го столбца, остальные биты равны 0. После этого все элементы массива *tmp* объединяются с помощью *побитового или* в один элемент для каждого блока.

Замечание 2.2.2. *Операции* $COL(i,T)$ *и* $ROW(i,T)$ *выполняются за константное время. Тем не менее* $COL(i,T)$ *выполняется быстрее* $ROW(i,T)$.

2.2.3. Реализация и оптимизация библиотеки базовых ассоциативных алгоритмов

Для того, чтобы избежать многократного запуска ядер на графическом ускорителе, введены следующие изменения.

- В *__device__* - процедурах различаются слайсы глобальные (параметры процедур) и локальные (внутренние переменные процедур). Глобальные слайсы представляют собой указатель на адрес первого элемента массива, обозначаются с префиксом *d_*, аналогично соответствующему полю класса *Slice*. Локальные слайсы - это одна переменная *unsigned long long int* на каждом потоке. Записываются без префикса. Отдельно стоит выделить переменные *unsigned long long int ** с префиксом *col_*, которые используются для доступа к столбцам таблицы.

- Для корректной работы с таблицами в $\_\_global\_\_$ и $\_\_device\_\_$ процедурах кроме указателя на первый элемент массива данных таблицы необходимо передавать число столбцов.

- Работа с локальными слайсами производится стандартными операциями языка C++.

Базовые операции для обработки глобальных слайсов в $\_\_device\_\_$ процедурах: $\_and(X, Y)$, $\_not(X)$, $\_or(X, Y)$, $\_xor(X, Y)$, $\_get\_bit(X, i)$, $\_clr(X)$, $\_set(X)$, $\_mask(X, num)$, $\_col(d_tab, i)$, $\_assign(X, Y)$.

Базовые операции для обработки локальных слайсов в $\_\_device\_\_$ процедурах: &, $\sim$, |, $\wedge$, $x = 0$, $x =\sim 0$. Операции между глобальными и локальными слайсами: $x = \_assign(d\_y)$ и $\_assign(d\_y, x)$. Операции $\_get\_bit(X, i)$, $\_mask(X, num)$, $\_col(d\_tab, i)$ к локальным слайсам не применимы.

Замечание 2.2.3. *В языке STAR существуют базовые операции, критичные к синхронизации по данным:* $SOME(X)$, $ZERO(X)$, $FND(X)$, $STEP(X)$. *И, поскольку в CUDA не предусмотрен механизм блочной синхронизации данных, эти процедуры не реализумы как* $\_\_device\_\_$*-процедуры.*

Замечание 2.2.4. *Операции со строками* $ROW(i, T)$*, в зависимости от размеров таблицы, могут требовать другую конфигурацию запуска на графическом ускорителе, чем операции со столбцами, поэтому реализуются только как* $\_\_global\_\_$*-процедуры.*

Теперь рассмотрим реализацию и возможную оптимизацию библиотеки базовых ассоциативных алгоритмов [55]. Базовые алгоритмы можно разделить на следующие группы по способу работы с таблицами:

I в алгоритме используются базовые операции, критичные к синхронизации;

II в алгоритме не используются базовые операции, критичные к синхронизации, при этом есть зависимость по данных по столбцам;

III арифметические алгоритмы;

IV в алгоритме не используются базовые операции, критичные к синхронизации, при этом столбцы могут обрабатываться независимо друг от друга.

I группа алгоритмов.

Алгоритмы первой группы реализуются только как $\_\_host\_\_$-процедуры. К этой группе относятся алгоритм поиска минимального значения MIN и алгоритм поиска максимального значения MAX.

В алгоритмах MIN и MAX используются предикат $SOME$, поэтому алгоритмы разбиваются на 2 части: непосредственно вычисления и проверка предиката. При вычислении предиката вызывается процедура поиска первого единичного элемента, потом результат копируется с графического ускорителя на процессор и на процессоре вычисляется значение предиката. Но реализацию алгоритмов MIN и MAX можно оптимизировать, если отказаться от копирования результата поиска первого единичного элемента и проверять его перед вычислениями. Данный подход возможно использовать для оптимизации и других алгоритмов, вызывающих базовые операции, критичные к синхронизации. Сравнение производительности неоптимизированной и оптимизированной реализаций алгоритма MIN будет приведена в 2.2.4.

II группа алгоритмов.

Алгоритмы второй группы реализуются в виде как $\_\_host\_\_$-, так и $\_\_global\_\_$- и $\_\_device\_\_$-процедуры. Имена таких процедур строятся следующим образом:

$\_\_device\_\_$ — название строчными буквами;

$\_\_global\_\_$ — название строчными буквами с суффиксом $\_kernel$;

$\_\_host\_\_$ — название прописными буквами;

При вызове $\_\_global\_\_$-процедуры используются следующие параметры $<<< blocks, threads >>>$, где

$$threads = min(128, NN),\ blocks = (NN - 1)/threads + 1,$$

NN-число 64-х разрядных слов для записи слайса.

Таким образом алгоритмы второй группы можно вызывать как с хоста, так и использовать в других $\_\_host\_\_/\_\_global\_\_$ процедурах.

Ко второй группе алгоритмов относятся следующие базовые ассоциативные алгоритмы.

- Процедура MATCH(T, X, v, Y) находит строки таблицы T, отмеченные '1' в слайсе X, равные слову v. Такой слайс X называется управляющим. Результат записывается в слайс Y.

- Процедура LESS(T, X, v, Y) по управляющему слайсу X ищет строки таблицы T, которые меньше[2] слова v. Результат записывается в слайс Y.

- Процедура GREAT(T, X, v, Y) по управляющему слайсу X ищет строки таблицы T, которые больше слова v. Результат записывается в слайс Y.

- Процедура GEL(T, w, X, Y) сравнивает строки таблицы T со словом w. Если $row(i,T) > w$, то $X(i) = 1$. Если $row(i,T) < w$, то $Y(i) = 1$. Если $row(i,T) = w$, то $X(i) = 0$ и $Y(i) = 0$.

- Процедура SETMIN(T, F, X, Z) по управляющему слайсу X находит строки $row(i,T) < row(i,F)$.

- Процедура SETMAX(T, F, X, Z) по управляющему слайсу X находит строки $row(i,T) > row(i,F)$.

- Процедура HIT(T, F, X, Z) по управляющему слайсу X находит строки $row(i,T) = row(i,F)$.

III группа алгоритмов.

К третьей группе алгоритмов относятся арифметические ассоциативные алгоритмы сложения AddV(T, R, X, S, B), AddC(T, X, v, F, B), AddC1(T, X, v,

[2] Под отношениями $<, >$ и $=$ подразумевается обычный порядок на числах в двоичном исчислении. При этом количество битов в строке и слове совпадают.

F, B) и вычитания SubTV(T, R, X, S, B), SubTC(T, X, v, F, B), SubTC1(T, X, v, F, B).

Эти алгоритмы при вычислении используют B, слайс переноса на следующий разряд. Если после окончания вычислений B не пуст, то использовались некорректные данные (в случае сложения матрица результата должна быть шире на один столбец, в случае вычитания в некоторых строках получается отрицательное число). В самих алгоритмах операции, критичные к синхронизации, не используются и их можно было бы отнести ко второй группе алгоритмов, но для корректности вычислений необходимо использовать $SOME(B)$ или $ZERO(B)$.

Кроме того, процедуры $AddC$, $AddC1$, $SubtC$ и $SubtC1$ используют для вычислений вспомогательную таблицу. Поэтому для экономии памяти при реализации используются другие алгоритмы.

Рассмотрим процедуры прибавления слова w к строкам таблицы T.

```
 1 ADDC(T,X,v,F,B)
 2 Var T_i, F_i: Slice;
 3     i:         integer;
 4 Begin
 5     CLR(B)
 6     for i=h downto 1 do
 7     begin
 8         T_i=col(T,i);
 9         M=T_i xor B;
10         if w(i)=0 then
11            B=B and T_i;
12         else
13         begin
14            M=not M;
15            B=B or T_i;
16         end
17         Col(F,i)=X and M;
18         B=X and B;
19      end;
20 end;
```

Листинг 2.4. процедура ADDC

Утверждение 2.2.1. *В результате работы процедуры ADDC таблица F имеет следующий вид:*

$$Row(F,i) = \begin{cases} \emptyset, & если\, X(i) = 0; \\ Row(T,i) + w, & если\, X(i) = 1. \end{cases}$$

Доказательство. Рассмотрим сложение на i-м разряде. Нам необходимо найти столбец результата $Col(F,i)$ и слайс переноса на следующий разряд B.

Будем обозначать за B перенос на текущем разряде i, а за B^* перенос на следующий разряд, T_i - i-й столбец таблицы T, а F_i - i-й столбец результирующей таблицы F.

Рассмотрим таблицу истинности (таб. 2.2) в зависимости от значения $w(i)$:

$w(i)=0$				$w(i)=1$			
T_i	B	F_i	B^*	T_i	B	F_i	B^*
0	0	0	0	0	0	1	0
0	1	1	0	0	1	0	1
1	0	1	0	1	0	0	1
1	1	0	1	1	1	1	1

Таблица 2.2. Таблицы истинности для вычисления ADDC

Таким образом получаем следующие булевы формулы:

$$F_i^* = \begin{cases} T_i\ xor\ B, & \text{если}\, w(i) = 0; \\ not(T_i\ xor\ B), & \text{если}\, w(i) = 1. \end{cases}$$

$$B^* = \begin{cases} T_i\ and\ B, & \text{если}\, w(i) = 0; \\ T_i\ or\ B, & \text{если}\, w(i) = 1. \end{cases}$$

Поскольку складываются только строки, помеченные $'1'$ в слайсе X, а остальные строки остаются нулевыми, то в конечном итоге имеем:

$$F_i = F_i^*\ and\ X \text{ и } B^* = B^*\ and\ X.$$

Нетрудно видеть, что процедура $ADDC$ из листинга 2.4 вычисляет данные формулы. □

Результат процедуры ADDC1 отличается от результата процедуры ADDC тем, что

$$Row(F,i) = \begin{cases} Row(T,i), & \text{если } X(i) = 0; \\ Row(T,i) + w, & \text{если } X(i) = 1. \end{cases}$$

В этом случае формула для вычисления слайса переноса на следующий разряд B не изменяется, а формула для столбца результата $Col(F,i)$ имеет следующий вид:

$$F_i = (F_i^* \; and \; X) \; or \; (T_i \; and \; notX).$$

Теперь рассмотрим процедуры вычитания слова w из строк таблицы T. Значения столбца результата F_i и слайса переноса на следующий разряд B^* в зависимости от значения $w(i)$ приведены в таблице 2.3.

$w(i)=0$				$w(i)=1$			
T_i	B	F_i	B^*	T_i	B	F_i	B^*
0	0	0	0	0	0	1	1
0	1	1	1	0	1	0	1
1	0	1	0	1	0	0	0
1	1	0	0	1	1	1	1

Таблица 2.3. Таблицы истинности для вычисления SubTC

Следующие логические формулы вычисляют нужные значения:

$$F^* = \begin{cases} T \; xor \; B, & \text{если } w(i) = 0; \\ not(T \; xor \; B), & \text{если } w(i) = 1. \end{cases}$$

$$B^* = \begin{cases} B \; and \; notT, & \text{если } w(i) = 0; \\ not(T \; xor \; B) \; or \; B, & \text{если } w(i) = 1. \end{cases}$$

Поскольку операция вычитание применяется только к строкам, помеченным $'1'$ в слайсе X, а остальные строки остаются нулевыми, то в конечном итоге имеем:

$$F_i = F_i^* \; and \; X \text{ и } B^* = B^* \; and \; X.$$

Процедура *SubTC* из листинга 2.5 вычисляет данные формулы.

```
1  SubTC(T,X,v,F,B)
2  Var T_i, F_i: Slice;
3      i:         integer;
4  Begin
5      CLR(B)
6      for i=h downto 1 do
7      begin
8          Y=col(T,i)
9          A=Y xor B
10         if w(i)=0 then
11             M=B and not Y;
12         else
13             A=not A;
14             B=A or M;
15         endif
16         col(F,i)=A and X;
17         B = M and X;
18       end;
19 end;
```

Листинг 2.5. процедура SubTC

Как и при сложении, в случае, когда строки, помеченные '0' в слайсе X, остаются без изменений, формула для вычисления слайса переноса на следующий разряд B не изменяется, а формула для столбца результата $Col(F, i)$ имеет следующий вид:

$$F_i = (F_i^* \; and \; X) \; or \; (T_i \; and \; notX).$$

IV группа алгоритмов.

Алгоритмы четвертой группы, также как и второй, реализуются как в виде $\_\_host\_\_$-, так и в виде $\_\_global\_\_$- и $\_\_device\_\_$-процедур. Но в них обработка столбцов производится независимо друг от друга, поэтому может происходить одновременно. *Таким образом, параметры вызова соответствующей* $\_\_global\_\_$*-процедуры* $<<<(blocks, k), threads>>>$, где $k \in (1, h)$ (h — ширина обрабатываемой таблицы).

К алгоритмам этой группы относятся:

- заполнение таблицы нулями CLEAR(h, F);
- копирование выделенных строк одной таблицы в другую, остальные строки в результирующей таблице не изменяются TMERGE(T, X, F);
- запись слова в выделенные строки таблицы, остальные строки не изменятся WMERGE(w, X, F);
- запись слова в выделенные строки таблицы, остальные строки обнуляются WCOPY(w, X, F);
- копирование таблицы TCOPY(T, F), таблицы T и F одинакового размера;
- выделение полосы из таблицы TCOPY1(T, j, h, F);
- вставка полосы в таблицу TCOPY2(T, j, h, F).

2.2.4. Оценка производительности реализации STAR-машины на GPU и реализации библиотеки стандартных процедур

Оценка производительности базовых операций языка Star

Таблица 2.4. Профилирование базовых операций.

Операция	Процедура	Время выполнения (μs)			
		100	1 000	3 000	5 000
ROW (чтение)	get_row	18,6 (18,5 – 19,7)	19, 5	19,6	19,8
ROW (запись)	set_row	4,1 (2,9 – 5,6)	3,9	4,3	4,1
or	or_long_value	1,9 (1,9 – 3,0)	2,2	2,0	2,1
FND	find	2,5 (2,3 – 2,8)	2,8	3,5	3,8
	first_backward	3,2 (3,1 – 4,0)	3,2	3,3	3,4
NUMB	numb_backward	1,3 (1,0 – 2,1)	1,3	1,3	1,3
	numb_shift	4,8 (4,2 – 8,2)	5,9	7,2	7,5

Определение времени работы базовых операций выполнялось на таблицах размера $N \times N$ при $N = 100,\ 1\,000,\ 3\,000$ и $5\,000$. Результаты профилирования показаны в таблице 2.4. В первом столбце приведено название базовой операции, во втором — имя процедуры, выполняющейся на графическом ускорителе (реализации операций FND и $NUMB$ вызывают две глобальных процедуры). Отметим, что время выполнения процедуры включает в себя время запуска ядра, поэтому может существенно отличаться для значений порядка нескольких микросекунд. В таблице 2.4 приведено среднее время запуска, а также для 100 вершин в скобках показаны минимальное и максимальное время выполнения процедур.

Замечание 2.2.5. *Время выполнения базовых операций практически не зависит от размера данных, что соответствует теоретическим оценкам в разделе 2.2.2:* $O(\lceil \log_{64}(n) \rceil)$ *для реализации операции* FND, $SOME$, $STEP$ *и* $FRST$, *и константное время для реализации остальных операций.*

Оценка производительности алгоритмов библиотеки базовых процедур

Напомним, что теоретическая оценка сложности всех ассоциативных алгоритмов из библиотеки базовых процедур равна $O(h)$, где h - ширина таблицы, N - длина таблицы. При этом теоретическая оценка сложности реализации этих алгоритмов зависит от группы, в которую они входят. В таблице 2.5 приведены теоретическая оценка и время выполнения реализации базовых алгоритмов. Расчеты проводились на графических ускорителях GeForce920m и узеле k40 Kepler кластера nks-30T ССКЦ.

Ниже приводится более подробное сравнение производительности для одного алгоритма из каждой группы с аналогами из библиотек STL и CUDA thrust.

Оценка производительности алгоритмов первой группы

Первая группа алгоритмов состоит из алгоритмов, использующих базовые операции, критичные к синхронизации. В библиотеке базовых процедур это

Группа алгоритмов	Оценка сложности	Размер данных (одновременно)	Время μs	
			GF920m	k40
I	$O(h \cdot \lceil \log_{64}(N) \rceil)$	до 4 096 4 097 – 100 000	400 570 – 590	713 865 – 890
II	$O(h)$	100 000	58 – 61	71 – 73
III	$O(h + \lceil \log_{64}(N) \rceil)$	50 000	36 – 41	52 – 58
IV	$O(1)$	10 000	6-10	8-9

Таблица 2.5. Теоретическая сложность и время выполнения реализации библиотеки стандартных алгоритмов.

процедуры поиска минимального и максимального значений.

Теоретическая сложность ассоциативных алгоритмов MIN и MAX $O(h)$, где h - ширина таблиц. Теоретическая сложность ассоциативных алгоритмов, реализованных на графическом ускорителе, $O(h(1 + \lceil \log_{64}(N) \rceil))$.

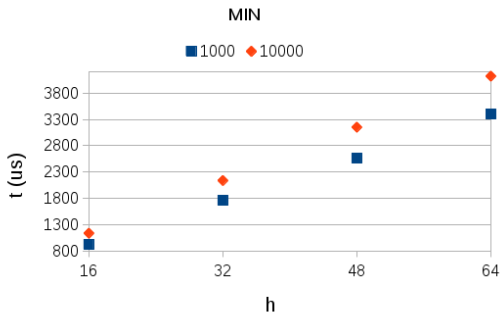


Рис. 2.4. Время работы с таблицами разной ширины.

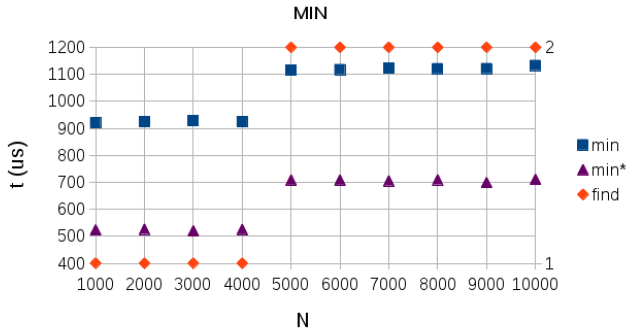


Рис. 2.5. Время работы с таблицами разной длины.

На рисунке 2.4 приведена зависимость времени работы процедуры от ширины обрабатываемых таблиц h при длине таблиц $N = 1000$ и $N = 10000$. На рисунке 2.5 показаны время работы процедуры MIN при разной длине таблиц (левая шкала, min - неоптимизированная реализация, min^* - оптимизированная реализация) и количество сверток при вычислении предиката SOME (правая шкала, $find$). Как видно из рисунков, время работы процедуры линейно зависит от ширины таблицы h и незначительно отличается для таблиц с одинаковым количеством сверток, необходимых для вычисления предиката SOME.

Оптимизация позволяет уменьшить время работы на 400μс.

Производится сравнение производительности следующих реализаций:

- min – неоптимизированная реализация ассоциативного алгоритма MIN;
- min^* – оптимизированная реализация ассоциативного алгоритма MIN;
- $std :: min\_element$ – последовательная реализация с использованием векторов из библиотеки стандартных шаблонов C++ *STL* [66];
- $thrust :: min\_element$ – реализация на CUDA с использованием библиотеки thrust [67];
- $thrust\_min\_element^*$ – реализация на CUDA с использованием библиотеки thrust, выдающая вектор, в котором помечены единицей позиции минимального элемента.

Отметим, что сравнение не совсем корректно, поскольку формат входных данных и выходные параметры сравниваемых реализаций различаются:

- $std :: min\_element$ и $thrust :: min\_element$ возвращают указатель на минимальный элемент в массиве;
- min, min^* и $thrust\_min\_element^*$ возвращают столбец (вектор для $thrust\_m$ в котором $'1'$ помечены все позиции минимальных элементов.

Тем не менее, некоторое представление о производительности получить можно.

На рисунке 2.6 показано отношение времени работы различных реализаций к оптимизированной реализации ассоциативного алгоритма MIN. Из рисунка 2.6 видно следующее:

- для векторов достаточно малой размерности (< 5000) все реализации на GPU уступают в производительности $std :: min\_element$, но для векторов размерностью 10 000 и более реализации на GPU дают заметный выигрыш;

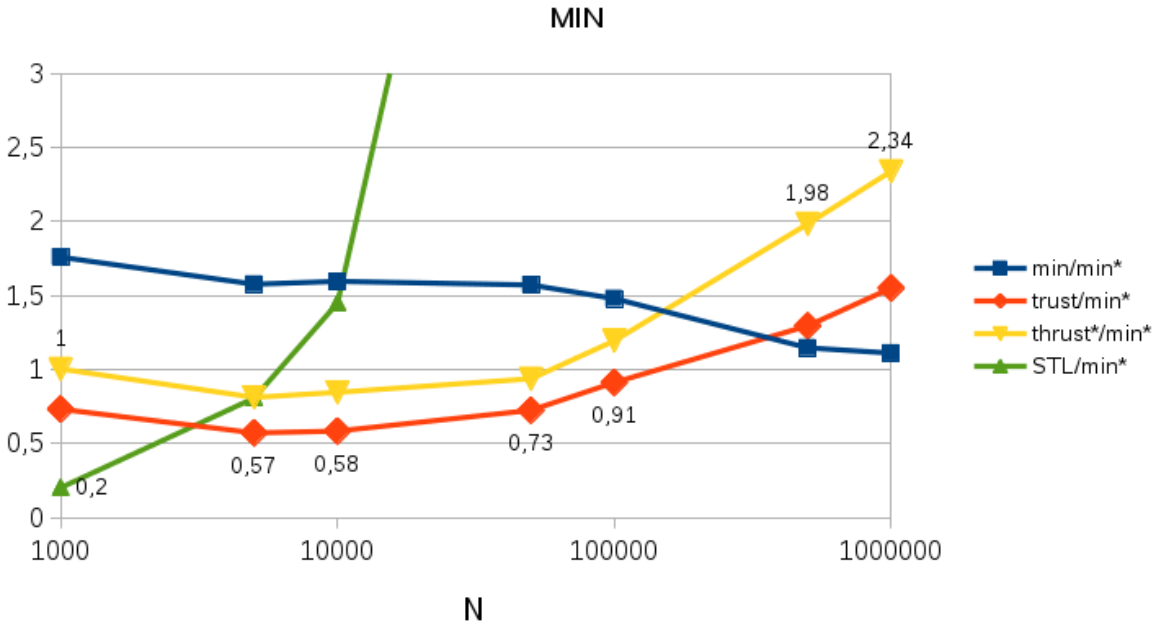


Рис. 2.6. Отношение времени работы реализации.

- при размерности векторов $\leq 64^2 = 4\,096$ (1 свертка для вычисления предиката $SOME$) времена работы реализаций min^* и $thrust\_min\_element^*$ практически совпадают;
- при размерности векторов от $4\,097$ до $64^3 = 262\,114$ (2 свертки для вычисления предиката $SOME$) время работы реализаций min^* больше времени работы $thrust\_min\_element^*$ в начале диапазона, сравнивается к размерности $50\,000$ и меньше уже к середине диапазона;
- при размерности векторов $\leq 262\,115$ (3 и более свертки для вычисления предиката $SOME$) время работы $min*$ гораздо меньше времени работы $thrust\_min\_element^*$.

Оценка производительности алгоритмов второй группы

Рассмотрим время выполнения процедуры $MATCH$. Реализация ассоциативной процедуры $MATCH$ будет сравниваться со следующими реализациями:

- std::find();

- thrust::find();

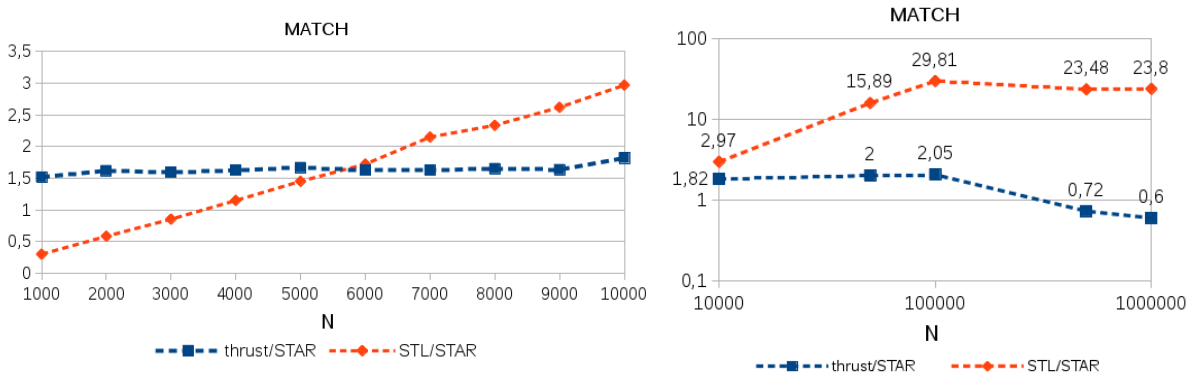


Рис. 2.7. Сравнение время работы алгоритмов поиска.

Отметим, что процедуры std::find() и thrust::find() выдают указатель на первое вхождение искомого элемента, в то время как результатом процедуры MATCH будет слайс, в котором $'1'$ отмечены все вхождения. А также в MATCH поиск может быть произведен только среди выделенных элементов, в то время как такой поиск с помощью процедур библиотек std и thrust ведет к дополнительным накладным расходам.

Из 2.7 видно, что для векторов небольшой размерности (до 4 000 - 6 000 элементов) библиотека STL выигрывает по производительности у реализаций на GPU, но с ростом размерности обрабатываемых векторов значительно им уступает. В случае, когда все блоки могут обрабатываться одновременно, $MATCH$ работает в $1,5 - 2$ раза быстрее, чем thrust::find().

Оценка производительности алгоритмов третьей группы

Рассмотрим время выполнения процедуры $SUBTV$ при различных размерах таблицы. Напомним, что теоретическая сложность ассоциативной процедуры $SUBTV$ оценивается как $O(h)$, где h- ширина таблицы, и не зависит от длины таблицы N.

На рисунках 2.8 и 2.9 показаны зависимости время выполнения процедуры subtv_kernel от размеров обрабатываемых таблиц (время указывается в микросекундах). На рисунке 2.8 показано среднее время работы процедуры

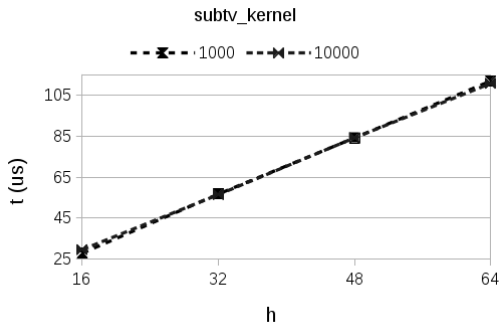


Рис. 2.8. Время работы с таблицами разной ширины.

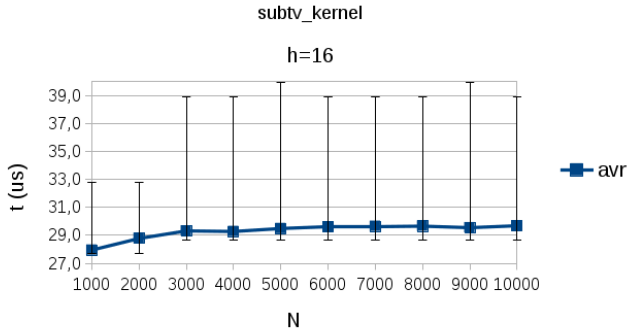


Рис. 2.9. Время работы с таблицами разной длины.

subtv_kernel для различных h, $N = 100, 10000$. На рисунке 2.9 показано время работы процедуры при фиксированном $h = 16$: среднее значение и отрезки с минимальным и максимальным временем выполнения.

Эксперимент полностью согласуется с теоретической оценкой: время работы не зависит от длины таблиц, и зависит линейно от их ширины.

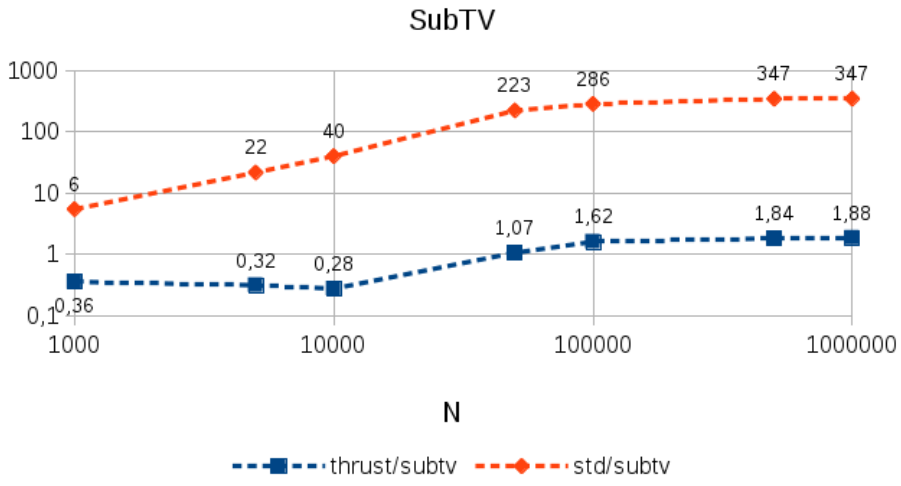


Рис. 2.10. Сравнение времени работы алгоритмов вычитания.

Сравним время работы реализации ассоциативного алгоритма с соответствующими процедурами из библиотек STL и thrust:

- std::transform(...,std::minus<int>());
- thrust::transform(...,thrust::minus<int>());

Из рисунка 2.10 видно, что реализации на GPU существенно превосходят по производительности последовательные реализации. Реализация ассоциативного алгоритма дает выигрыш на больших данных.

Отметим, что возможность варьировать ширину обрабатываемой таблицы h в ассоциативном алгоритме (т. е. диапазон вычисляемых величин $[0; 2^h - 1]$) может, с одной стороны, дать дополнительную возможность ускорить вычисления, но, с другой стороны, приводить к ошибкам из-за выхода значения из допустимого диапазона и требует проверки слайса переноса.

Оценка производительности алгоритмов четвертой группы

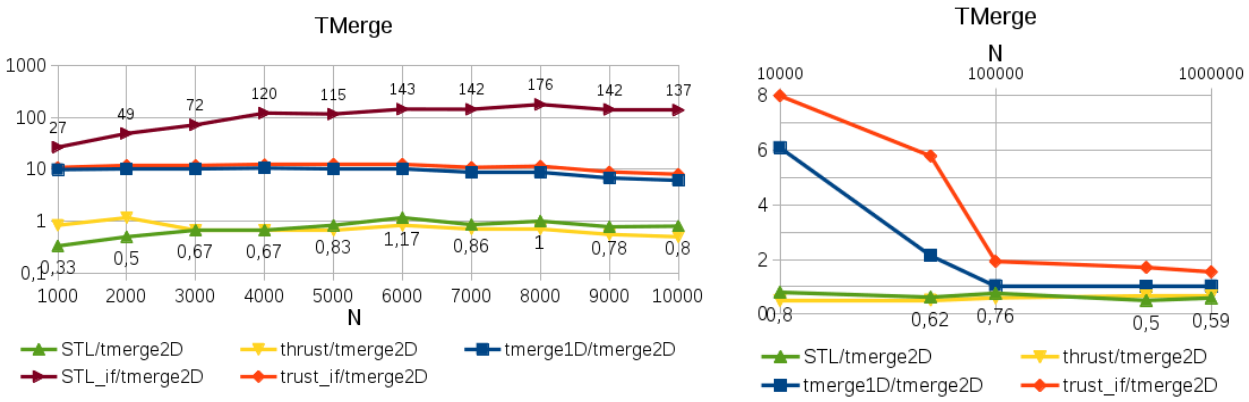


Рис. 2.11. Сравнение время работы алгоритмов копирования.

Приведем оценку производительности алгоритмов из третьей группы на примере реализации ассоциативной процедуры TMerge, которая копирует строки таблицы T, отмеченные $'1'$ в слайсе X, в соответствующие строки таблицы F, остальные строки F остаются без изменений.

Рассматриваем две реализации ассоциативного алгоритма TMERGE:

- tmerge1D: поочередная обработка столбцов, как для алгоритмов второй группы (одномерная организация блоков);
- tmerge2D: параллельная обработка столбцов (двумерная организация блоков: количество блоков по y равно ширине таблицы $gridDim.y = h$, организация нитей внутри блока остается одномерной $blockDim.y = 1$).

Аналоги этой процедуры в библиотеках STL и thrust являются процедуры *copy* и *copy_ if*. Процедура copy производит копирование всех элементов, а *copy_ if* - только тех элементов, которые удовлетворяют некоторому предикату. В приведенных тестах проверялась четность номера элементов. Отметим, что процедура *copy_ if* не входит в библиотеку STL, ее реализация взята из [66].

Из рисунка 2.11 видно, что реализация tmerge2D сравнима по времени работы с беспредикатными процедурами копирования стандартных библиотек в случае, если у графического ускорителя хватает ресурсов выполнять блоки одновременно. В случае, когда необходимо копировать не все строки, последовательная процедура копирования уступает параллельным реализациям (более чем в 100 раз для векторов с размерностью более 4 000 элементов). Если ресурсов для одновременного выполнения блоков не достаточно, время работы процедур tmerge1D и tmerge2D сравнивается и они уступают беспредикатным реализациям из библиотек STL и trast. Тем не менее, они привосходят предикатные реализации: thrust::copy_if в $1.5 - 2$ раза и stl_copy_if в $140 - 150$ раз.

2.3. Оптимизация ассоциативных алгоритмов под исполнение на графических ускорителях

Как было показано в главе 2, при реализации STAR-машины на графических ускорителях ассоциативные свойства модели сохраняются. Но время выполнения для некоторых базовых операторов яыка STAR заметно больше, в то время как в модели предполагается выполнение всех базовых операций за один такт. Поэтому для оптимизации ассоциативных алгоритмов под исполнение на графических ускорителях проводятся следующие этапы.

1. **Определение точек синхронизации и уменьшение их числа.** А именно, выделение операций, критичных к синхронизации: чтение/запись строк, циклы и условные операторы с предикатами SOME или ZERO.
   В модели STAR-машины предполагается, что предикат SOME и операция STEP выполняются за один такт, но в реализации это не так. А также каждый вызов этих базовых операторов требует синхронизации. Отметим

Ассоциативный алгоритм	Опимизация под GPU
1 while SOME(X) 2 begin i=STEP(X); 3 ... 4 end;	1 i=STEP(X); 2 while (i>0) 3 begin ... 4 i=STEP(X); 5 end;
Оптимизация циклов.	
1 if SOME(X) then 2 begin i=fnd(X) 3 ... 4 end;	1 i=fnd(X) 2 if (i>0) then 3 begin ... 4 end;
Оптимизация условного оператора.	

Таблица 2.6. Уменьшение числа операций, критичных с синхронизации.

также, что $SOME(X) = true$ тогда, и только тогда, когда $STEP(X) > 0$ (или $FND(X) > 0$).

В таблице 2.6 приведен способ исключения обращения к предикатам в цикле и условном операторе. Предложенная замена с одной стороны немного ухудшает понимание STAR-алгоритма, но с другой стороны уменьшает количество синхронизаций в циклах с $2k$ до $k+1$, а в условном операторе в два раза (здесь k - количество единичных битов в слайсе X).

В случае, когда в алгоритме вызовов операций чтения и записи строк больше, чем вызовов операций обращения к столбцам, целесообразно пересмотреть исходный алгоритм. Один из вариантов - на вход подавать транспонированную матрицу.

2. **Двухуровневый параллелизм.** Проследить, есть ли зависимость по данным при обработке столбцов.
   Такой вид параллелизма встречается в IV группе базовых ассоциативных алгоритмов и алгоритме Уоршалла, который обсуждается ниже.

3. **Уменьшение числа __global__-процедур.**
   Для всех операторов, некритичных к синхронизации, и базовых алгорит-

мов II-IV групп предпочтительно использовать __device__-процедуры, сокращая таким образом накладные расходы.

Данные рекомендации **универсальны** и легки в реализации.

2.4. Выводы ко второй главе

Представлена эффективная реализация STAR-машины на графических ускорителях с помощью технологии CUDA. Базовые операции языка Star представлены в виде процедур, выполняемых на графическом ускорителе за константное время или за время $O(\lceil \log_{64}(n) \rceil)$, близкое к константному.

Star-машина использует существенно другое представление данных, поэтому механический перенос ассоциативных алгоритмов на ЭВМ стандартной архитектуры невозможен. Поэтому был проведен анализ существующих форматов входных/выходных данных для тестовых графов, содержащих более 5000 вершин. На его основании были выбраны два формата: *.gr (используется в генераторе R-MAT графов GraphHPC-1.0, синтетические графы, моделирующие графы социальных сетей и интернета) и Autonomus System(графы протоколов интернета). В реализацию STAR-машины на GPU был добавлен модуль с процедурами ввода/вывода данных в выбранных форматах с переводом в любой из внутренних форматов STAR-машины (матрица смежности, матрица весов или список дуг).

Реализована на графических ускорителях библиотека стандартных ассоциативных алгоритмов с оценкой времени, совпадающей с теоретической. Произведено сравнение времени работы библиотеки стандартных ассоциативных параллельных алгоритмов с временем работы подобных алгоритмов из библиотек C++ STL и CUDA thrust.

Показано, что для векторов размерности более 5 000 элементов реализации представленных алгоритмов на GPU работают быстрее процедур библиотеки STL.

Созданная реализация базовых ассоциативных алгоритмов дает существенное ускорение по сравнению с библиотекой STL и, зачастую, выигрывает в про-

изводительности у библиотеки CUDA thrust.

У представленной реализации Star-машины есть потенциал для оптимизации, связанный с вычислением конфигурации блоков с учетом свободных ресурсов.

Разработаны и представлены рекомендации для оптимизации ассоциативных алгоритмов для выполнения на графических ускорителях. Оптимизация направлена на уменьшение числа точек синхронизации и выделения участков кода, в которой нет зависимости по данных не только по строкам, но и по столбцам. Данные рекомендации универсальны и легки в реализации.

Глава 3

Ассоциативные параллельные алгоритмы для решения задач на графах

В этой главе приводятся несколько ассоциативных алгоритмов для решения задач на графах. Для каждого из них доказывается корректность, а также демонстрируется их оптимизация под вычисление на графических ускорителях и результаты вычислительных экспериментов. Расчеты проводились как на видеокарте NVIDIA GEFORCE 920m, так и на различных графических ускорителях кластеров.

3.1. Алгоритм Уоршалла транзитивного замыкания

Алгоритм Уоршалла является одним из классических алгоритмов теории графов. Данный алгоритм по матрице смежности графа определяет наличие пути между всеми парами вершин. Ниже представлена последовательная версия алгоритма Уоршалла. Теоретическая сложность последовательного алгоритма $O(n^3)$.

```
1 for k = 1 to n
2     for i = 1 to n
3         for j = 1 to n
4             W[i][j] = W[i][j] or (W[i][k] and W[k][j])
```

Листинг 3.1. Псевдокод алгоритма Уоршалла

Ассоциативный алгоритм Уоршалла с теоретической оценкой сложности $O(n^2)$ был представлен в [68]. Данный алгоритм разрабатывался для Star-машины, в которой доступ как к строкам, так и к столбцам выполняется за один такт. При этом в данной ассоциативной версии алгоритма Уоршалла число обращений к строкам на чтение и запись превышает количество обращений к столбцам. Но для использования реализации на GPU такой подход является существенным недостатком, поскольку приводит к большому числу синхрони-

заций. Поэтому ассоциативный алгоритм был адаптирован под исполнение на графических ускорителях.

3.1.1. Адаптация ассоциативного алгоритма Уоршалла под выполнение на GPU

В процессе адаптации ассоциативного алгоритма Уоршалла хорошо прослеживаются все шаги оптимизации, описанные в разделе 2.3.

Уменьшение числа точек синхронизации

На первом этапе уменьшаем число точек синхронизации как за счет изменения условия цикла while, так и за счет уменьшения числа обращений к строкам. Первая модификация тривиальна, а правомерность второй модификации доказывается ниже.

```
 1 proc WARSHALL-C(n:integer; var P: table);
 2 var x:word; V, W: Slice; i,k: integer;
 3 begin for k:=1 to n do
 4    Begin
 5       x:=ROW(k,P);
 6       W:=COL(k,P);
 7       i=STEP(x);
 8       while i>0 do
 9       Begin
10          V:=COL(i,P);
11          V=V or W;
12          COL(i,P):=V;
13          i:=STEP(x);
14       end;
15    end;
16 end;
```

Листинг 3.2. Модифицированная Star-версия алгоритма Уоршалла

Для этого рассмотрим выполнение одной итерации ($k = 2$) процедур WARSHALL и WARSHALL-C (рисунок 3.1). В процедуре WARSHALL в слайс X записан 2-й столбец, а в слово w — 2-я строка матрицы P^1. В процессе итера-

X	$w =$	**1**	0	**1**	0	0	0
0		0	0	1	0	0	0
0		1	0	1	0	0	0
1		***1***	*1*	***1***	*0*	*0*	*1*
1		***1***	*1*	***1***	*0*	*0*	*0*
0		0	0	1	0	0	0
0		0	0	0	1	1	0

a) WARSHALL

W	$x =$	***1***	0	***1***	0	0	0
0		*0*	0	*1*	0	0	0
0		*1*	0	*1*	0	0	0
1		***1***	1	***1***	0	0	1
1		***1***	1	***1***	0	0	0
0		*0*	0	*1*	0	0	0
0		*0*	0	*0*	1	1	0

b) WARSHALL-C

Рис. 3.1. Результат выполнения итерации ($k = 2$) для Star-версий алгоритма Уоршалла. Обрабатываемые элементы матриц выделены курсивом, элементы, изменяющие свои значения, — жирным шрифтом.

ции со словом w объединяются те стро́ки матрицы P^1, которые отмеченные ‘1’ в слайсе X. На рисунке 3.1а такие строки выделены курсивом. Значения элементов строк, выделенных только курсивом, не изменяются, потому что соответствующие им элементы слова w равны нулю. Поэтому изменить значения могли только те элементы строк, которые находятся в столбцах, отмеченных ‘1’ в слове w. Такие элементы отмечены жирным шрифтом.

Аналогично в процедуре WARSHALL-C, в слайс W записан 2-й столбец, а в слово x — 2-я строка матрицы P^1. В цикле (3–15) столбцы, отмеченные ‘1’ в строке x , объединяются со слайсом W. Эти столбцы выделены курсивом на рисунке 3.1б. На этой итерации могли измениться значения только тех элементов столбцов, которые находятся в строках, отмеченных ‘1’ в слайсе W. Эти элементы также отмечены жирным шрифтом, и их позиции совпадают с позициями элементов, которые могли измениться в процедуре WARSHALL. Приведем формальное доказательство.

Утверждение 3.1.1. *Процедуры WARSHALL и WARSHALL-C вычисляют одну и ту же логическую функцию.*

Доказательство. Для доказательства утверждения восстановим вычисляемую логическую функцию для каждой процедуры.

Заметим, что $COl(k, P)$ соответствует $\forall j, P(j, k)$, а $ROW(k, P)$ — $\forall j, P(k, j)$.

Тогда в этих терминах для каждого фиксированного k тело цикла (4–15) процедуры WARSHALL записывается в следующем виде:

$$\forall j\,\forall i\,P^k(i,j) = \begin{cases} P^{k-1}(i,j), & \text{если } P^{k-1}(i,k) = 0; \\ P^{k-1}(i,j)\,or\,P^{k-1}(k,j), & \text{если } P^{k-1}(i,k) = 1. \end{cases}$$

Это эквивалентно

$$\forall j\,\forall i\,P^k(i,j) = P^{k-1}(i,j)\,or\,\left(P^{k-1}(i,k)\,and\,P^{k-1}(k,j)\right) \tag{3.1}$$

Для тела цикла (4–15) процедуры WARSHALL-C имеем:

$$\forall j\,\forall i\,P^k(j,i) = \begin{cases} P^{k-1}(j,i), & \text{если } P^{k-1}(k,i) = 0; \\ P^{k-1}(j,i)\,or\,P^{k-1}(j,k), & \text{если } P^{k-1}(k,i) = 1. \end{cases}$$

Это эквивалентно

$$\forall j\,\forall i\,P^k(j,i) = P^{k-1}(j,i)\,or\,\left(P^{k-1}(j,k)\,and\,P^{k-1}(k,i)\right) \tag{3.2}$$

Формулы (3.1) и (3.2) совпадают с точностью до индексов на каждой итерации $k = 1 \ldots n$.

□

Замечание 3.1.1. *Теоретическая оценка обеих ассоциативных версий совпадает (не более $O(n^2)$, фактически $O(j)$, где j – количество ненулевых элементов в результирующей матрице, $n \le j \le n^2$). Число точек синхронизации реализации WARSHALL-C на $2j$ меньше, чем число точек синхронизации реализации WARSHALL.*

Доказательство. В процедуре WARSHALL операция COL вызывается n раз, а операция ROW вызывается $n + j$ раз на чтение и j раз на запись. В процедуре WARSHALL-C операция ROW вызывается n раз на чтение, а операция COL вызывается $n + j$ раз на чтение и j раз на запись. □

2D-параллелизм

Заметим, что в процедуре WARSHALL-C цикл (3–15) может выполняться параллельно по всем столбцам. Приведем листинг кода (3.3), используя в

основном язык Star. Для записи процедуры нам потребуются из CUDA C++ переменная $blockIdx.x$, возвращающая номер исполняемого блока, и директивы __global__ и $\lll n \ggg$, указывающие, что процедура выполняется на n блоках. При этом вся итерация выполняется одним ядром.

```
1  __global__ Warshall_kernel(P,k)
2  Begin
3  // номер блока индексируется с 0, а номер столбца с 1
4      i:=blockIdx.X+1;
5      W:=COL(k,P);
6      V:=COL(i,P);
7      if(V(k)=1) then V=V or W;
8      COL(i,P):=V;
9  end;
10 proc WARSHALL-adapt(n:integer; var P: table);
11 var x:word; V, W: Slice; i,k: integer;
12 Begin
13   for k:=1 to n do
14     Warshall_kernel<<<n>>>(P,k);
15 end;
16 }
```

Листинг 3.3. Адаптированный Star-алгоритм Уоршалла

Такой подход позволяет снизить временную сложность алгоритма до $O(n)$ и значительно уменьшить время выполнения алгоритма. Отметим, что не все блоки физически выполняются одновременно, поэтому экспериментальное время счета отличается от линейного. Но производители графических ускорителей постоянно наращивают количество ядер графического ускорителя.

Время выполнения данной версии на различных графических ускорителях будет показано в 3.1.3 при сравнении производительности реализаций алгоритма Уоршалла на графических ускорителях.

3.1.2. Обзор решений на графических ускорителях

Стандартный алгоритм Уоршалла на GPU

Хариш и Нарайанан [69] реализовали алгоритм Уоршалла на CUDA (Nvidia GeForce 8800GTX) самым простым способом. Они создали ядро, содержащее одну итерацию внутреннего цикла, и параллельно вызывали это ядро для каждого элемента матрицы смежности A.

Алгоритм 3.1.1 Алгоритм Флойда-Уоршалла в реализации Хариша и Нарайанана

```
1: for k from 1 to V do
2:     for all A[i, j], 1 ≤ i, j ≤ V in parallel do
3:         A ← min(A[i, j], A[i, k] + A[k, j])
```

В работе [70] предлагается этот же подход. Авторы приводят корректность и делают профилирование работы алгоритма на графическом ускорителе NVIDIA GeForce 8600 GT. Из-за большого объема данных (16 байт на каждый поток), считываемых с глобальной памяти, авторы получают ускорение в $3,3$ раза по сравнению с выполнением на CPU (Core 2 Duo E6550).

Блочный алгоритм Уоршалла

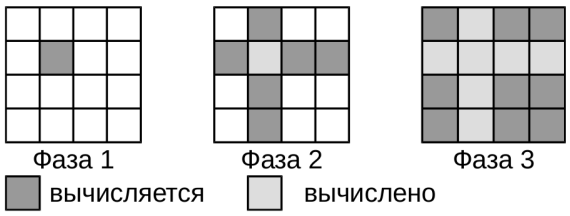


Рис. 3.2. Фазы выполнения одного этапа блочного алгоритма Флойда-Уоршалла.

Кац и Кидер в работе [71] применили блочный подход для повышения эффективности кэширования алгоритма Флойда-Уоршалла для практического использования разделяемой памяти, связанной с каждым мультипроцессором в графическом процессоре.

Алгоритм 3.1.2 Блочный алгоритм Уоршалла. Фаза 1.

```
1: {Предполагается, что в w(x,y) записан вес ребра (x, y)}
2: for all  0 ≤ b < n/s do
3:     {Вычисляются самозависимые блоки.}
4:     for all b * s ≤ k < (b + 1) * s do
5:         for all b * s ≤ i < (b + 1) * s do
6:             for all b * s ≤ j < (b + 1) * s do
7:                 w_ij ← min(w_ij, w_ik + w_k,j)
```

В блочном алгоритме матрица разбивается на блоки размером 32×32, и задачи выполняются поэтапно. Каждый этап требует выполнения трех фаз. На рисунке 3.2 показаны фазы выполнения этапа блочного алгоритма Флойда-Уоршалла.

Алгоритм 3.1.3 Блочный алгоритм Уоршалла. Фаза 2. Вычисление однозависимых блоков по слабозависимому блоку.

```
8:     for all 0 ≤ ib ≤ n/s do                          ▷ Выровнены по i
9:         for all b * s ≤ k < (b + 1) * s do
10:            for all b * s ≤ i < (b + 1) * s do
11:                for all ib * s ≤ j < (ib + 1) * s do
12:                    w_ij ← min(w_ij, w_ik + w_k,j
13:    for all 0 ≤ jb ≤ n/s do                          ▷ Выровнены по j
14:        for all b * s ≤ k < (b + 1) * s do
15:            for all jb * s ≤ i < (jb + 1) * s do
16:                for all b * s ≤ j < (b + 1) * s do
17:                    w_ij ← min(w_ij, w_ik + w_k,j)
```

На первой фазе первое ядро (алгоритм 3.1.2) вычисляет 32 задачи для каждого элемента данных в одном блоке по диагонали матрицы смежности, называемом *самозависимым блоком*. Задачи в самозависимом блоке зависят только от других задач в этом блоке или от задач, которые были вычислены на предыдущем этапе.

Второе ядро (3.1.3) вычисляет 32 задачи для каждого элемента данных в

каждом блоке, выровненной по самозависимому блоку в i- или j-направлении. Это *однозависимые блоки*. Задачи в одиночно зависимых блоках имеют одну зависимость данных внутри блока и одну зависимость в уже вычисленном самозависимом блоке. На каждой стадии есть $O(n)$ отдельно зависимых блоков.

Алгоритм 3.1.4 Блочный алгоритм Уоршалла. Фаза 3. Вычисление двузависимых блоков.

```
18:  for all 0 ≤ ib ≤ n/s do
19:      for all 0 ≤ jb ≤ n/s do
20:          for all jb * s ≤ i < (jb + 1) * s do
21:              for all ib * s ≤ j < (ib + 1) * s do
22:                  for all block * size ≤ k < (block + 1) * size do
23:                      w_ij ← min(w_ij, w_ik + w_k,j)
```

Наконец, третье ядро (3.1.4) вычисляет 32 задачи для каждого из оставшихся блоков. Это *двузависимые блоки*, и каждый из этих блоков полностью зависит от двух однозависимых блоков. Поскольку в двузависимом блоке задачи независимы между собой, то они могут выполняться в любом порядке.

На каждом этапе имеется $O(n^2)$ двузависимых блоков, и скорость, с которой выполняется этот этап, определяет скорость алгоритма.

Блочная реализация алгоритма Уоршалла с разделяемой памятью

Смит и Ланд в работе [72] предложили дополнительную оптимизацию блочного алгоритма Уоршалла для графических ускорителей.

Во-первых, они заменили операции деления и умножения операциями битового сдвига. Во-вторых, авторы использовали разделяемую память вместо глобальной.

Таким образом им получить ускорить реализаций Каца примерно в 5 раз.

3.1.3. Сравнение производительности

Для сравнения времени работы различных реализаций алгоритма Уоршалла на графических ускорителях будут использованы модифицированный и адаптированный Star-алгоритмы Уоршалла и неассоциативные реализации, приведенные в работах [69, 71, 72].

Таблица 3.1. Время работы последовательного алгоритма Уоршалла и его Star-версий на GeForce920m

Количество вершин	1 000	2 000	3 000	4 000	5 000
Последовательный алгоритм	8,037	59,812	197,111	461,785	884,622
WARSHALL-C	4,827	11,161	23,363	42,661	64,454
WARSHALL-adapt	0,003	0,027	0,093	0,236	0,372

В таблице 3.1 сравнивается время работы последовательного алгоритма Уоршалла с его ассоциативными версиями. WARSHALL-C – модифицированная ассоциативная версия (листинг 3.2, раздел 3.1.1), а WARSHALL-adapt – ассоциативная версия, адаптированная под выполнение на GPU (листинг 3.3, раздел 3.1.1).

Сравнительное время расчета ассоциативных версий алгоритмов Уоршалла для графа, имеющего 5000 вершин, на различных графических ускорителей показано на рисунке 3.3. Расчеты проводились на следующих системах:

GeForce920m — ноутбук;

nks Kepler — кластер nks-30T ССКЦ (узел k40 Kepler);

Titan X — кластер ФИТ НГУ;

Volta — кластер НГУ.

На рисунке 3.4 показано время работы различных реализаций алгоритма Уоршалла:

- **S** — последовательный алгоритм Уоршалла;
- **AP** — модифицированный Star-алгоритм WARSHALL-C;

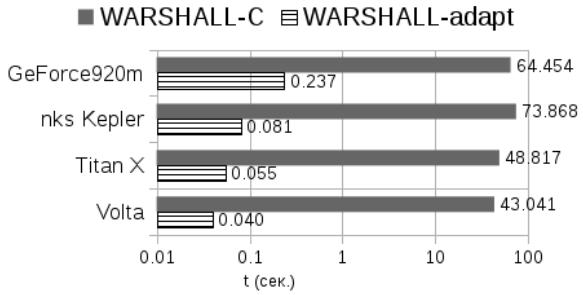


Рис. 3.3. Время работы ассоциативных версий для графа с 5000 вершин.

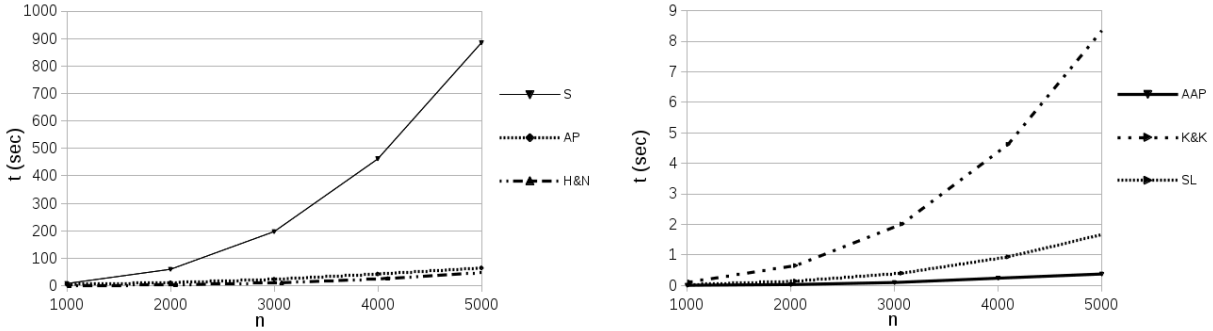


Рис. 3.4. Время работы реализаций алгоритма Уоршалла на графических ускорителях.

- **N&H** — реализация алгоритма Уоршалла на графических ускорителях, приведенная в работе [69];
- **AAP** — адаптированный Star-алгоритм WARSHALL-adapt;
- **K&K** — блочная реализация, приведенная в работе [71];
- **SL** — блочная реализация с использованием разделяемой памяти, приведенная в работе [72].

Отметим, что **K&K**, **SL** являются оптимизациями для **N&H** реализации алгоритма Уоршалла. Время выполнения этих реализаций приведено в работе [72].

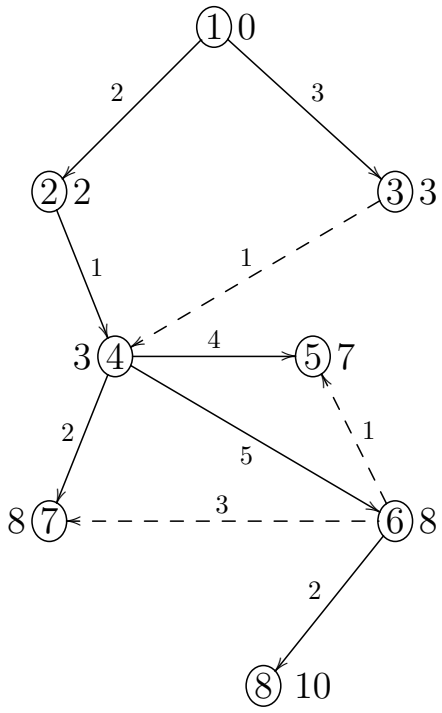
Как видно из рисунка 3.4, Star-алгоритм выполняется за время, сравнимое с временем работы неоптимизированной реализации алгоритма Уоршалла на графических ускорителях. В то же время, адаптация Star-алгоритма к архитектуре GPU приводит к большему ускорению, чем различные оптимизации **N&H** реализации алгоритма на GPU.

3.2. Алгоритм Дейкстры нахождения кратчайших путей

В разделе приводятся статическая и динамическая версии алгоритма Дейкстры нахождения кратчайших путей от одной из вершин ориентированного взвешенного графа до всех остальных вершин в случае положительных весов.

Статическая версия алгоритма используется для первичного построения дерева кратчайших путей SPT и матрицы расстояний Dist. В случае, когда в граф добавляется дуга, динамический (инкрементальный) алгоритм обновляет матрицы SPT и Dist быстрее, чем статический алгоритм находит решение для измененного графа.

В приводимых ниже ассоциативных алгоритмах используются следующие данные.



∞=1111

Weight								**Dist**
∞	∞	∞	∞	∞	∞	∞	∞	0000
0010	∞	∞	∞	∞	∞	∞	∞	0010
0011	∞	∞	∞	∞	∞	∞	∞	0011
∞	0001	0001	∞	∞	∞	∞	∞	0011
∞	∞	∞	0100	∞	0001	∞	∞	0111
∞	∞	∞	0101	∞	∞	∞	∞	1000
∞	∞	∞	0101	∞	0011	∞	∞	1000
∞	∞	∞	∞	∞	0010	∞	∞	**1010**

Рис. 3.5. Представление данных для ассоциативного алгоритма Дейкстры.

- Weight - матрица весов графа размером $N \times hN$, где N - число вершин графа, а h - число битов, необходимых для кодирования максимального расстояния.
- inf - строка, длиной в h бит, задающая код бесконечности.

- G - матрица смежности графа размером $N \times N$.
- SPT - матрица смежности дерева кратчайших расстояний размером $N \times N$.
- Dist - матрица размером $N \times h$, в i-й строке которой в бинарном виде записано расстояние от корня до вершины i (или inf, если вершина i недостижима из корня).

На рисунке 3.5 представлен взвешенный ориентированный граф и матрицы Weight и Dist, соответствующие ему. Кратчайшее расстояние показано меткой у соответствующей вершины. Дуги, принадлежащие дереву кратчайших расстояний отмечены сплошной линией. Хотя в реализации по умолчанию ширина поля h имеет 32 бита, можно устанавливать произвольную ширину поля. Так, в приведенном примере максимальное расстояние от вершины 0 равняется 10, поэтому за код бесконечности inf можно взять 15, для бинарной записи которого достаточно 4 бита.

3.2.1. Статический ассоциативный алгоритм Дейкстры

Ассоциативный алгоритм Дейкстры поиска кратчайших расстояний для связанного графа описан в [73]. В этом варианте для задания графа кроме матрицы весов необходима еще матрица Cost, являющейся транспонированной матрицей весов. Кроме того, обрабатываемый граф должен быть связанным. Чтобы обрабатывать произвольные графы, алгоритм был изменен. Приведем описание модифицированного ассоциативного алгоритма Дейкстры (алгоритм 3.2.1).

Заметим, что в шагах 5-9 обрабатываются параллельно все дуги, исходящие из вершины j. Таким образом сложность ассоциативного алгоритма оценивается как $O(k)$, где k – число достижимых вершин из корня s, и не зависит от количества дуг в графе.

В случае неориентированного графа с неотрицательными весами дуг данный алгоритм можно использовать для получения компонент связности. Если граф несвязный, то у вершин i, недостижимых из корня s, $Dist(i) = \inf$. Для

Алгоритм 3.2.1 Статический ассоциативный алгоритм Дейкстры.

1: \\ Инициализация
2: Dist(s):=0; $\forall i \neq s$ Dist(i):=inf;
3: AffectedV(s):=1; $\forall i \neq s$ Affected(i):=0;
4: **while** SOME(AffectedV) **do**
5: Из аффектных вершин выбирается вершина j с минимальным текущим расстоянием, Affected(j):=0.
6: Для множества вершин $\{i|\exists j : (j,i) \in E(G)\}$ считается длинна пути через вершину j: $D\_temp(i) := Dist(j) + Weight(j,i)$.
7: Из этого множества выбираются вершины, для которых длина пути через вершину j меньше посчитанного ранее.
8: Для выбранных вершин обновляются матрица расстояний и матрица кратчайших путей.
9: Эти вершины помечаются как аффектные.

построения следующей компоненты связности корнем берется любая вершина, не принадлежащая множеству достижимых вершин из s.

Изменения в модификации: отличие модифицированной версии в том, что вершина для следующей итерации выбирается из непройденных аффектных вершин, т. е. текущее расстояние до них было уменьшено и не равно бесконечности. Использование вершины с минимальным расстоянием позволяет предотвратить повторное попадание вершины в аффектные и значительно уменьшает время счета.

```
1 procedure DistSPT2(Weight: Table; s: integer; h: integer;
    Var Dist: Table; Var FVer: Table)
2 Var   k :integer
3       R1, G : Table;
4       AffectedV, X, Y: Slice;
5       v, inf: Word(Dist)
6 Begin
7 /* Инициализация */
8        SET(inf);
9        ADJ(Weight,inf,G);
```

```
10          SET(AffectedV); AffectedV(s):=0;
11          WCOPY(inf,AffectedV,Dist);
12          NOT(AffectedV); //  единицей помечен только корень s
13        while SOME(AffectedV) do
14        begin
15  /* Выбираем вершину с минимальным расстоянием от корня. */
16           MIN(Dist,AffectedV,X);
17           k=STEP(X); AffectedV(k):=0;
18  /* Вычисляем расстояния через вершину k */
19           X:=COL(k,G);
20           if SOME(X) then
21           begin
22               v:=Row(k,Dist);
23               TCOPY1(Weight,k,h,R1);
24               ADDC1(X,v,R1);
25               SETMIN(R1,Dist,X,Y);
26  /* Обновляем Dist и SPT для вершин, расстояние до которых
       уменьшилось*/
27               TMERGE(R1,Y,Dist);
28               CLR(w); w(k):=1;
29               WMERGE(w,Y,SPT);
30  /* Помечаем такие вершины как аффектные.  */
31               AffectedV:=AffectedV or Y;
32           end;
33       end
34  End;
```

Листинг 3.4. Модифицированный ассоциативный алгоритм Дейкстры

3.2.2. Инкрементальный алгоритм Дейкстры для динамической обработки дерева кратчайших путей после добавления новой дуги

Теперь рассмотрим ассоциативную версию динамического алгоритма. Пусть дуга (i, j) добавляется к графу G. Тогда проверяем, уменьшится ли кратчайшее расстояние от корня до вершины j, если кратчайший путь до вершины j будет включать дугу $(i.j)$. Если это верно, то вершина j становится аффектной и но-

вое кратчайшее расстояние до вершины j записывается в матрицу кратчайших расстояний. Поскольку в каждую вершину дерева входит единственная дуга, то в дереве кратчайших путей заменяем дугу, которая первоначально заходила в вершину j, на дугу (i, j). После этого необходимо вычислить расстояние до тех вершин, в которые заходят дуги из вершины j. В случае, если до каких-то вершин расстояние уменьшилось, то эти вершины отмечаются как аффектные, расстояния до них записываются в соответствующие строки матрицы кратчайших расстояний, а соответствующие дуги вносятся в дерево кратчайших путей.

Вначале приведем ассоциативный параллельный алгоритм для обработки дуг, выходящих из текущей аффектной вершины. Этот алгоритм использует матрицу смежности G, матрицу кратчайших путей SPT, матрицу кратчайших расстояний $Dist$, матрицу весов $Weight$ и текущую аффектную вершину k. Заметим, что перед выполнением этого алгоритма в k-й строке матрицы $Dist$ записано новое кратчайшее расстояние до вершины k.

Алгоритм 3.2.2 ассоциативный параллельный алгоритм UpdateOutgoingEdges

---

Require: $k, h, G, Weight, Dist, SPT$

Ensure: $Dist, SPT, Y$

1: С помощью слайса X сохранить позиции дуг, выходящих из k в G.

2: С помощью матрицы $R1$ сохранить веса дуг, выходящих из k в $Weight$.

3: С помощью матрицы $R1$ сохранить новые расстояния от корня до тех вершин графа G, позиции которых отмечены $'1'$ в слайсе X.

4: С помощью слайса Y сохранить номера тех вершин из слайса X, для которых новые кратчайшие расстояния от корня уменьшились. ▷ Слайс Y будет хранить новые аффектные вершины.

5: Записать в матрицу $Dist$ новые кратчайшие расстояния от корня до тех аффектных вершин, позиции которых хранятся в слайсе Y.

6: Перестроить дерево кратчайших путей следующим образом. Пусть слово $w0$ содержит единственную '1' в k-ом бите. Тогда в дереве кратчайших путей SPT записываем слово $w0$ в каждую строку, отмеченную '1' в слайсе Y.

На STAR-машине этот алгоритм реализован в виде процедуры UpdateOutgoingEdges.

Теперь приведем ассоциативный параллельный алгоритм для динамической обработки дерева кратчайших путей после добавления к графу G дуги (i, j) весом $v1$. Этот алгоритм будет использовать матрицу дерева кратчайших путей SPT, матрицу кратчайших расстояний $Dist$, матрицу весов $Weight$ и слайс $AffectedV$.

Алгоритм 3.2.3 ассоциативный параллельный алгоритм для динамической обработки дерева кратчайших путей после добавления к графу G дуги (i, j) весом $v1$

Require: $(i, j), v1, G, Weight, Dist, SPT$

Ensure: $Dist, SPT$

1: Включить дугу (i, j) в матрицу смежности G, а ее вес $v1$ добавить в матрицу $Weight$.

2: С помощью слова $v2$ сохранить кратчайшее расстояние до вершины i, с помощью слова $v3$ сохранить длину нового пути до вершины j

3: С помощью слова $v4$ сохранить старое кратчайшее расстояние до вершины j

4: **if** $v3 < v4$ **then**

5: Включить вершину j в слайс $AffectedV$ и записать новое кратчайшее расстояние $v3$ до вершины j в матрицу $Dist$.

6: Удалить из матрицы SPT дугу, которая заходила в вершину j перед добавлением к графу G дуги (i, j), и затем включить в матрицу SPT дугу (i, j).

7: **while** $AffectedV \neq \emptyset$ **do**

8: В слайсе $AffectedV$ выделить верхнюю вершину k с помощью операции STEP.

9: Выполнить процедуру UpdateOutgoingEdges.

10: Добавить в слайс $AffectedV$ новые аффектные вершины, позиции которых хранятся в слайсе Y.

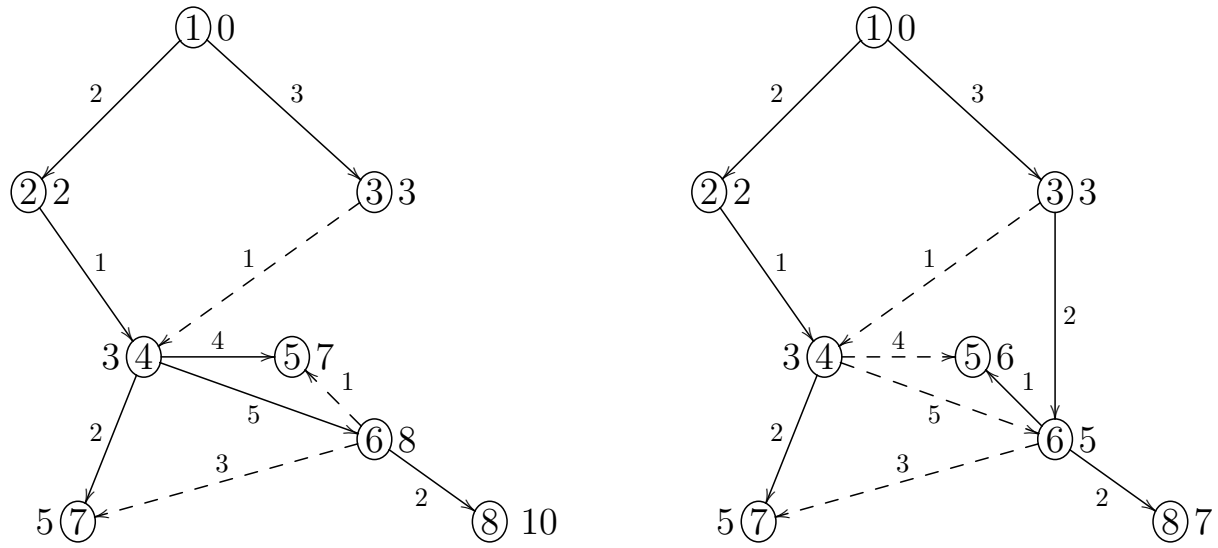


Рис. 3.6. Дерево кратчайших расстояний до и после добавления дуги (3,6) с весом 2.

На рисунке 3.6 показан пример как изменяются кратчайшие расстояния и дерево кратчайших расстояний после добавления дуги (3,6) с весом 2. Метками у вершин показаны кратчайшие расстояния, а сплошными стрелками обозначены дуги, принадлежащие дереву кратчайших расстояний. Аффектными будут 6-я, 5-я и 8-я вершины. При этом дуга $(6, 8)$ принадлежала дереву кратчайших расстояний и до добавления дуги, в то время как дуга $(6, 5)$ заменила в дереве дугу $(4, 5)$.

На STAR-машине этот алгоритм реализован в виде процедуры InsertArcSPT.

Выполнение инкрементального ассоциативного алгоритма на STAR-машине

В данном разделе вначале рассмотрим выполнение на STAR-машине вспомогательной процедуры UpdateOutgoingEdges, а затем приведем основную процедуру InsertArcSPT.

Приведем вспомогательную процедуру UpdateOutgoingEdges. Зная аффектную вершину k и текущие матрицы SPT, $Weight$ и $Dist$, эта процедура возвращает слайс Y, который хранит аффектные вершины, смежные с вершиной k, и матрицу $Dist$, которая хранит новые кратчайшие расстояния от корня до этих вершин.

```
procedure UpdateOutgoingEdges(k,h:integer; G:table; Weight:table;
  var Dist:table; var SPT:table; var Y:slice(G));
/* Здесь k - это аффектная вершина.*/
var v: word(Dist); w:word(SPT);
 X:\,slice(SPT);
R1,R2: table;
i:integer;
 1. Begin X:=COL(k,G);
 2.    TCOPY1(Weight,k,h,R1);
/* Матрица R1 хранит веса дуг, выходящих из вершины k. */
 3.    v:=ROW(k,Dist);
/* Слово v хранит новое кратчайшее расстояние до вершины k. */
 4.    ADDC(R1,X,v,R2);
/* Матрица R1 хранит новые расстояния от корня до вершин,
выходящих из вершины k в матрице SPT. */
 5.    SETMIN(R2,Dist,X,Y);
/* Слайс Y хранит новые аффектные вершины.\,*/
 6.    TMERGE(R2,Y,Dist);
/* Новые расстояния до аффектных вершин смежных с вершиной k
записываются в соответсвующие строки матрицы Dist. */
 7.    CLR(w); w(k):='1';
/* Вершина k помечается как вероятный отец для этих вершин */
 8.    WMERGE(w,Y, SPT);
13. End;
```

Утверждение 3.2.1. *Пусть дуга* (i, j) *весом* $v1$ *добавлена к графу* G*. Пусть заданы параметр* h *и аффектная вершина* k*. Пусть также заданы текущие матрицы* G*,* SPT*,* $Weight$ *и* $Dist$*. Тогда после выполнения процедуры* UpdateOutgoingArcs *слайс* Y *будет хранить аффектные вершины, смежные с вершиной* k*, матрица* $Dist$ *будет хранить новые кратчайшие расстояния от корня до этих вершин, а матрица* SPT *– обновленное дерево кратчайших путей.*

Доказательство. Будем доказывать от противного. Пусть вершина r смежна с вершиной k в матрице G. Однако после выполнения процедуры UpdateOutgoingArcs r-й бит слайса Y будет равен нулю. Докажем, что это противоречит выполнению процедуры UpdateOutgoingArcs.

Действительно, после выполнения строк 1–2, слайс X будет хранить позиции дуг, выходящих в матрице G из вершины k, а матрица $R1$ будет хранить веса дуг, выходящих из этой вершины. По предположению вершина r смежна с вершиной k в матрице G. Поэтому $X(r) =' 1'$, а в r-й строке матрицы $R1$ записан вес дуги (k, r). После выполнения строк 3–6 в r-й строке матрицы $R2$ будет записано новое кратчайшее расстояние от корня до вершины r. Все вершины, для которых это расстояние будет меньше, чем $\mathrm{ROW}(r, Dist)$, отмечаются '1' в слайсе Y. Поэтому после выполнения строк 5–6 получаем, что $Y(r) =' 1'$ и новое кратчайшее расстояние от корня до вершины r запишется в r-ю строку матрицы $Dist$. После выполнения строки 7 слово w содержит единственную '1' в k-ом бите, а в слайсе X отмечены все вершины, смежные k, для которых расстояние уменьшилось. Тогда в дереве кратчайших путей SPT записываем слово w в каждую строку i, отмеченную '1' в слайсе X, таким образом вставляя дуги (k, i) в дерево кратчайших расстояний.

Это противоречит нашему допущению. □

Теперь приведем основную процедуру InsertArcSPT. Зная параметр h, добавляемую в граф дугу (i, j) весом $v1$, текущие матрицы G, $Weight$, $Dist$ и SPT, эта процедура возвращает измененные матрицы G, SPT, $Weight$, и $Dist$.

```
procedure InsertArcSPT(i,j,h:integer; v1:word(Dist);
    var Weight:table; var G,SPT:table; var Dist:table);
/* Дуга (i,j) весом v1 добавляется в граф G.*/
var k,r,l,l1,l2:integer;
   AffectedV,X,Y:slice(G);
   v2,v3,v4:word(Dist);  v5:word(G);
   v6:word(Weight);
 1. Begin X:=COL(i,G); X(j):='1'; COL(i,G):=X;
```

```
 /* Дуга (i,j) добавлена в граф G. */
 2.   l1:=1+(i-1)h; l2:=ih;
 3.   v6:=ROW(j,Weight); Rep(l1,l2,v1,v6);
 4.   ROW(j,Weight):=v6;
 /* В матрицу Weight записывается вес новой дуги (i,j). */
 5.   CLR(AffectedV); v2:=ROW(i,Dist);
 /* Строка v2 хранит кратчайшее расстояние от корня до вершины i. */
 6.   v3:=ADD(v1,v2);
 /* Строка v3 хранит новое кратчайшее расстояние до вершины j.*/
 7.   v4:=ROW(j,Dist);
 /* Строка v4 хранит старое кратчайшее расстояние до вершины j. */
 8.   if LESS(v3,v4) then
 9.   begin AffectedV(j):='1';
10.       ROW(j,Dist):=v3;
 /* Новое кратчайшее расстояние от корня до вершины j
 записывается в матрицу Dist. */
11.       CLR(v5); v5(i):=‘1’; ROW(j,SPT):=v5;
 /* Дуга (i,j) добавлена в матрицу SPT. */
12.       while SOME(AffectedV) do
13.       begin k:=STEP(AffectedV);
14.          UpdateOutgoingEdges(k,h,G,Weight,Dist,SPT,Y);
 /* Слайс Y хранит новые аффектные вершины, которые будут добавляться
 в слайс AffectedV. */
15.          AffectedV:=AffectedV or Y;
16.       end;
17.   end;
18. End;
```

Утверждение 3.2.2. *Пусть ориентированный взвешенный граф задан в виде матрицы смежности G и матрицы весов $Weight$. Пусть дерево кратчайших путей задано в виде матрицы смежности SPT, а множество кратчайших расстояний задано в виде матрицы $Dist$. Пусть заданы параметр h и дуга*

(i, j) весом $v1$, которая добавлена к графу G. Тогда после выполнения процедуры InsertArcSPT *будут изменены матрицы G,$Weight$, SPT и $Dist$.*

Доказательство. Будем доказывать индукцией по числу аффектных вершин $r \geq 0$, которые образуются после добавления дуги (i, j) к графу G.

Базис докажем для случая, когда $r \leq 1$. После выполнения строк 1-4 дуга (i, j) будет добавлена к графу G, а ее вес будет добавлен в матрицу $Weight$. После выполнения строки 5-6 слайс $AffectedV$ состоит из нулей, переменная $v3$ хранит новое кратчайшее расстояние от корня до вершины j, а переменная $v4$ хранит старое кратчайшее расстояние до вершины j. Если $v3 \geq v4$, то переходим на конец процедуры (строка 18). В противном случае переходим на строку 9. После выполнения строк 9–10 позиция вершины j отмечается $'1'$ в слайсе $AffectedV$ и в матрицу $Dist$ записывается новое кратчайшее расстояние до вершины j. Нам осталось внести изменения в матрицу SPT. Поскольку в каждую вершину любого дерева входит единственная дуга, а в матрицу SPT надо включить новую дугу (i, j). После выполнения строки 11 в j строку матрицы SPT записывается слово, содержащее единственную $'1'$ в i-й позиции Так как слайс $AffectedV \neq \emptyset$, то выполняем цикл `while SOME(AffectedV) do` (строки 12–16). После выполнения строки 13 получаем, что $k = j$ и $AffectedV = \emptyset$. По предположению индукции имеется не более одной аффектной вершины. Поэтому в результате выполнения строки 14 получаем, что $Y = \emptyset$. А тогда после выполнения строки 15 переходим на конец нашей процедуры.

Шаг индукции. Пусть утверждение теоремы выполняется для случая, когда $r \geq 1$. Докажем справедливость теоремы для $r + 1$ аффектных вершин. По индуктивному предположению после обработки первых r аффектных вершин дуга (i, j) будет добавлена в матрицу G, ее вес добавляется в матрицу $Weight$, дуга (l, j) удаляется из матрицы SPT, а дуга (i, j) туда добавляется. Кроме того, в матрицу $Dist$ записываются новые кратчайшие расстояния до первых r аффектных вершин. Поскольку после обработки первых r аффектных вершин в слайсе $AffectedV$ останется единственная аффектная вершина, то после выполнения строки 14 по Утверждению 1 для этой вершины обновлены матрица

Dist и *SPT*. □

3.2.3. Оптимизация алгоритма под выполнение на графических ускорителях

В инкрементальном алгоритме Дейкстры оптимизация под вычисление на GPU состоит только из последнего пункта в рекомендациях: уменьшение числа __global__- процедур.

В процедуре UpdateOutgoingEdges строки 5 и 6 могут быть занесены в одну __global__- процедуру и выполняться параллельно по столбцам (в блоках $<<< NN, h >>>$).

3.2.4. Выполнение инкрементального алгоритма Дейкстры на графических ускорителях

Оценка производительности проводилась на R-MAT-графах. Они хорошо моделируют реальные графы из социальных сетей и Интернета, а также являются достаточно сложными для анализа. Генерация графов производилась пакетом GraphHPC-1.0 [63] со следующими параметрами: количество вершин задается степенью двойки, средняя степень связности вершины равна 32. В таком R-MAT-графе имеется одна большая связная компонента и некоторое количество небольших связных компонент или висящих вершин.

Напомним, что порядок сложности ассоциативного параллельного алгоритма и его реализации на CUDA отличается на $O(log_{64}(n))$, если в алгоритме используются операции, критичные к синхронизации($STEP(X)$, $FND(X)$, $SOME(X)$). Таким образом сложность реализации алгоритма InsertArcSPT $O(hk \cdot lod_6 4(n))$, а реализации DistSPT – $O(hn \cdot lod_{64}(n))$. Здесь n – число вершин в графе, k – число аффектных вершин, h – число битов, которое требуется для кодирования длины максимального кратчайшего пути (по умолчанию 32).

В экспериментах использовались два режима:

Rmat* добавлялись дуги с весом 0 (пессимистичный сценарий);

Rmat добавлялись дуги со случайным весом (реалистичный сценарий).

В ходе экспериментов учитывалось не только время работы процедуры, но и количество аффектных вершин. Для каждого теста проводилось $\approx 10\% n$ запусков с добавлением дуги, где n – число вершин графа. После этого запуски распределялись по количеству аффектных вершин.

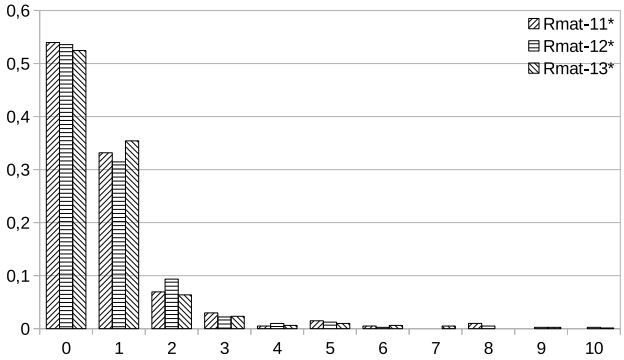


Рис. 3.7. Распределение запусков при добавлении дуги с весом 0.

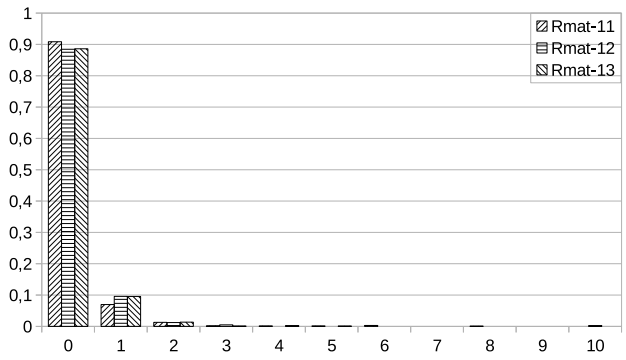


Рис. 3.8. Распределение запусков при добавлении дуги со случайным весом.

На рисунке 3.7 показано распределение запусков по количеству аффектных вершин при добавлении дуги с весом 0, а на рисунке 3.8 – при добавлении дуги со случайным весом. На рисунках видно, что для R-MAT–графов в большинстве случаев (более 50% в случае добавлении дуги с нулевым весом и более 88% в случае случайных весов) при добавлении дуги кратчайшие пути и расстояния не изменяются. В 99% случаев число аффектных вершин не превышает 5 в первом режиме и 2 во втором. Отметим, что в отличии от ассоциативного алгоритма, ведущего обход графа по вершинам (исходящие дуги обрабатываются параллельно), в последовательном варианте обход графа совершается по дугам, и число дуг, которые необходимо проверить, может доходить до 2000 в первом режиме и до 100 во втором.

Таблица 3.2. Сравнение времени работы статического и динамического ассоциативных алгоритмов на R-MAT графах.

граф	n	DistSPT	InsertArcSPT*	k*	InsertArcSPT	k
R-MAT-11	2048	6,171	0,012	6/8	0,009	1/8
R-MAT-12	4096	9,643	0,017	5/10	0,012	3/10
R-MAT-13	8192	29,475	0,038	7/15	0,020	4/13

В таблице 3.2 приводятся среднее время выполнения процедуры DistSPT и максимальное время выполнения процедуры InsertArcSPT в двух режимах добавления дуг. Здесь n – число вершин в графе, k – число аффектных вершин в худшем случае при добавлении дуги (число аффектных вершин, при добавлении дуги с максимальным временем обработки, максимальное число аффектных вершин при добавлении дуги). Время и количество аффектных вершин для режима Rmat* отмечены '*'.

Таким образом, при использовании динамического алгоритма время определения кратчайших расстояний уменьшается на несколько порядков, так как InsertArcSPT в большей степени зависит от числа аффектных вершин, чем от общего числа вершин в графе.

3.3. Динамический алгоритм Рамалингама для проблемы достижимости в потоковом графе с одним источником

В данном разделе приводится ассоциативная версия алгоритма Рамалингама для решения динамической проблемы достижимости в потоковых графах с одним источником при добавлении новой дуги. Эта задача возникает в различных приложениях, таких как компиляторы, системы верификации [74, 75], а также анализ и синтез информации в геоинформационных системах (ГИС) [76] и обработка запросов в базах данных с транзитивными отношениями [77].

3.3.1. Описание алгоритма Рамалингама

Алгоритм [78] использует структуру «динамическое дерево» [79], с заданными операциями:
link(u, v) – к вершине u остовного дерева T добавить поддерево с корнем в вершине v («подвесить» поддерево с корнем в v к вершине u);
cut(u, v) – в остовном дереве T удалить дугу (u, v) вместе с поддеревом с корнем в вершине v («разрезать» дугу (u, v)).

Заданы:

$G = (V, E)$ – ориентированный граф;

T – динамическое остовное дерево;

(u, v) – дуга, добавляемая в граф;

WorkSet множество дуг.

В этом алгоритме Рамалингам использует следующие понятия. Для каждой вершины $v \in V$ задано *support(v)* – множество поддерживающих вершин для v. Поддерживающая вершина определяет ся следующим образом: u – поддерживающая вершина для v, если u – достижима, дуга $(u, v) \in E$ и u не является потомком v в дереве T. Для каждой вершины v ? V reachable(v) истинно тогда и только тогда, когда v – достижима. $Succ(v)$ – множество всех вершин, в каждую из которых заходит дуга из вершины v. Алгоритм Рамалингама работает следующим образом.

```
1:  Добавить дугу (u,v) в E(G)
2: if reachable(u)=true then
3:     WorkSet:=(u,v)
4:     while WorkSet ≠ ∅ do
5:         Выбрать и удалить очередную дугу, скажем (x,y), из WorkSet
6:         if reachable(y)=faulse then
7:             support(y):=x
8:             reachable(y):=true
9:             link(x,y) в T
10:            for ∀z ∈ Succ(y) do
11:                Добавить дугу (y,z) в множество WorkSet
12:        else
13:            if x не потомок y в T then
14:                Добавить x в support(y)
```

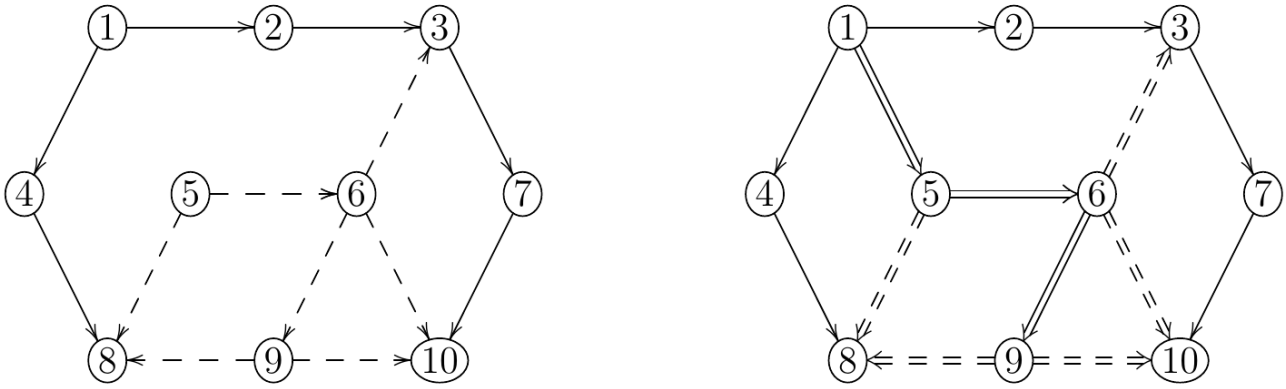


Рис. 3.9. Остовное дерево на множестве достижимых вершин до и после добавления дуги (1, 5)

На рис. 3.9 показано изменение остовного дерева после добавления к графу дуги (1, 5). Дуги, которые добавлялись в множество WorkSet в процессе исполнения алгоритма Рамалингама, обозначены двойной чертой.

3.3.2. Представление динамических деревьев на STAR-машине

Во многих алгоритмах на STAR-машине деревья задаются как матрица смежности (допустим, T). Чтобы задать структуру «динамическое дерево» на STAR-машине, такое представление необходимо дополнить матрицей потомков по дереву Des: в i-м столбце отмечены все потомки вершины i (т.е. те вершины, которые достижимы из вершины i в T).

В ассоциативном инкрементальном алгоритме Рамалингама из операций над динамическими деревьями [78] используются только операция link и ее вариация linkAll, которые мы приведем ниже.

```
1 procedure link( u, v: integer; Var T,  Desc: Table)
2 Var
3     X, Y :  Slice;
4     w :     Word;
5     k :     integer;
6 begin
7     X:=COL(u,T); X(v):=1; COL(u,T):=X;
8     w:=ROW(u,Desc);
9     X:=COL(v,Desc);
```

```
10      while SOME(w) do
11      begin
12          k:=STEP(w);
13          Y:=COL(k,Desc);
14          Y:=Y or X;
15          COL(k,Desc):=Y;
16      end;
17  end;
```

Поскольку в ассоциативных алгоритмах у выбранной вершины параллельно обрабатываются все исходящие ребра, то предпочтительно использовать пакетную перестройку поддеревьев: несколько поддеревьев подвешивается к одному корню за один проход.

```
 1  procedure linkAll( u: integer; X:Slice; Var T, Desc: Table)
 2  Var
 3      Y, Z, Z1:   Slice;
 4      w:          Word;
 5      k:              integer;
 6  begin
 7      Y:=COL(u,T); Y:=Y or X; COL(u,T):=Y;
 8      w:=ROW(u,Desc);
 9      Y:=X; CLR(Z);
10      while SOME(Y) do
11      begin
12          k:=STEP(Y);
13          Z1:=COL(k,Desc);
14          Z:=Z or Z1;
15      end;
16      while SOME(w) do
17      begin
18          k:=STEP(w);
19          Y:=COL(k,Desc);
20          Y:=Y or Z;
21          COL(k,Desc):=Y;
22      end;
23  end;
```

Также приведем соответствующие процедуры для операций cut и support,

```
 1 procedure cut(u, v: integer; Var T, Desc: Table)
 2 Var
 3     X,Y: Slice;
 4     w: Word;
 5     k: integer;
 6 Begin
 7    w:=ROW(u, Desc);
 8     Y:=COL(u,T); Y(v):=0; COL(u,T):=Y;
 9     X:=COL(v,Desc); X:=NOT(X);
10     while SOME(w) do
11     begin
12         k=STEP(w);
13         Y:=COL(k,Desc); Y:=Y and X; COL(k,Desc):=Y;
14     end;
15 end;
```

В алгоритме Рамалингама для каждой вершины определено множество поддерживающих вершин. Заметим, что убирая из множества $w = \{u \in RV : (u, v) \in G)\}$ потомков вершины v, при необходимости мы легко получаем множество поддерживающих вершин из матриц $Desc$ и G.

```
 1 procedure support( s, v:integer; Desc, G: Table; Var X:
     SLice)
 2 Var
 3    RV,Y :Slice;
 4    w    :Word;
 5    k    :integer;
 6 begin
 7    RV:=COL(s, Desc);
 8    Y:=COL(v, Desc);
 9    w:=ROW(v, G);
10    X:=RV and w;
11    X:=X and (not(Y));
12 end;
```

Отметим, что цикл (10-14) в процедурах link и cut и цикл (12-18) в процедуре linkAllна GPU могут выполняться параллельно по столбцам, аналогично оптимизации в алгоритме Уоршалла (глава 3.1.1).

3.3.3. Ассоциатианая версия динамического алгоритма Рамалингама

Для представления алгоритма Рамалингама на STAR-машине будем использовать следующую структуру данных:

- матрица смежности G размером $n \times n$, каждый i-й столбец которой хранит головы дуг, выходящих из вершины i;
- вершина s – источник;
- остовное дерево ST (Spanning Tree) задается матрицей размером $n \times n$, каждый i-й столбец которой хранит головы дуг, выходящих из вершины i и принадлежащих остовному дереву;
- матрица потомков $Desc$ размером $n \times n$, в i-м столбце которой отмечены '1' потомки вершины i;
- вспомогательный слайс $WorkSet$. На каждой текущей итерации этот слайс хранит те недостижимые вершины исходного графа, которые являются головами дуг, выходящих из текущей обрабатываемой вершины.

Алгоритм 3.3.1 Ассоциатианая версия динамического алгоритма Рамалингама.

Require: G, s

Ensure: RV, SPT

1: Включить вершину s в множество достижимых вершин RV и в множество граничных вершин Y
2: **while** $Y \neq \emptyset$ **do**
3: В слайсе Y выделить верхнюю вершину k с помощью операции STEP.
4: В слайсе X отметить те вершины, в которые заходят дуги из вершины k и которые не были достигнуты на предыдущих шагах.
5: Добавить в слайсы Y и VR достигнутые на этом шаге вершины.
6: Отметить вершину k отцем всех тех вершин, которые отмечены в слайсе X.

Замечание 3.3.1. *Заметим, что в s-м столбце матрицы Desc отмечены '1' все вершины, которые достижимы в остовном дереве из вершины s. Обозначим этот столбец как слайс RV (Reachable Vertices).*

Определим RV^n как множество достижимых вершин за n итераций цикла *While*. При этом $RV^0 = \{s\}$. И определим $BV^n \subseteq RV^n$ как множество тех вершин $v \in RV^n$, для которых существует дуга $(v, w) : w \notin RV^n$.

```
 1 procedure ReachableVerSPT(G: Table; s: integer; Var RV:
     Slice; Var SPT: Table)
 2 Var   k :integer;
 3       X, Y: Slice;
 4 Begin
 5 /* Шаг 1. */
 6        CLR(RV); RV(s):=1;
 7        Y:= COL(s,G);
 8        COL(s,SPT):=Y;
 9        RV:= RV or Y;
10      while SOME(Y) do
11      begin
12 /* Шаг 2. */
13         k:=STEP(Y);
14         X:=COL(k,G);
15         X:=X and (not RV);
16         if SOME(X) then
17         begin
18 /* Шаг 3. */
19             RV:= RV or X;
20             Y:=Y or X;
21 /* Шаг 4. */
22             COL(k,SPT):=X;
23         end;
24     end;
25 End;
```

Листинг 3.5. Ассоциативный алгоритм построения множества достижимых вершин и остовного дерева

Заметим, что после выполнения n-ной итерации цикла в слайсе RV отмечено

множество RV^n, а в слайсе Y – множество BV^n

```
 1 procedure ReachableVerFromSPT(SPT: Table; s,n: integer; Var
       RV: Slice)
 2 Var   k :integer;
 3       X, Y: Slice;
 4 Begin
 5      CLR(RV); RV(s):=1;
 6      For k:=1 to n do
 7      begin
 8         X:=COL(k,SPT);
 9         RV:=RV or X;
10      end
11 End;
```

Листинг 3.6. Ассоциативный алгоритм построения множества достижимых вершин по остовному дереву

```
 1 procedure InsertEdgeReachability(i,j,s: integer; Var G, ST,
       Desc: Table)
 2 Var
 3    WorkSet, X, RV, Z: Slice ;
 4 Begin
 5    CLR(WorkSet);
 6    RV:=COL(s,Desc);
 7    Z:= not RV;
 8    X:=COL(i,G); X(j):=1; COL(i,G):=X;
 9    if (RV(i)=1) and (RV(j)=0) then
10    begin
11       WorkSet(j):=1;
12       Z(j):=0;
13       link(i,j,ST,Desc);
14         While SOME(WorkSet) do
15         begin
16          k:=STEP(WorkSet);
17          X:=COL(k,G);
18          X:= X and Z;
19          if SOME(X) then
20          begin
```

```
21                WorkSet:= WorkSet or X;
22                Z:= Z and not X ;
23                linkAll(k,X,ST,Desc);
24             end;
25          end;
26       end;
27    End;
```

Утверждение 3.3.1. *Пусть G – матрица смежности потокового графа с источником в вершине s. Тогда после выполнения процедуры ReachableVerSPT слайс RV будет задавать множество вершин, достижимых из s и SPT - остовное дерево на множестве этих вершин. .*

Доказательство.

Базис индукции. Предположим, что в графе нет вершин, достижимых из корневой вершины s, т.е. из вершины s нет исходящих дуг и множество достижимых вершин состоит только из корня $\{s\}$. После выполнения строки 6 слайс RV содержит единственную единицу в позиции, соответствующей корню s. По предположению, после выполнения строки 7 слайс Y состоит только из '0'. Поэтому после выполнения строки 9 слайс RV не изменяется и условие цикла $9-29$ нарушается. Процедура заканчивает выполнение.

Шаг индукции.

Пусть множество RV содержит l вершин. Возможны два случая.

Случай первый: Слайс Y не пуст. Тогда, выполняется цикл (10-29). В строке 13 из слайса Y выбирается верхняя '1', которая соответствует вершине k, достижимой из корня s. В результате выполнения строк $14-15$ в слайсе записаны позиции вершин, которые достигаются на этом шаге и не были достигнуты ранее. Если таких вершин нет, то итерация цикла завершается, и из слайса Y выбирается следующая вершина.

Если же слайс X не пуст, то в строке 19 эти вершины добавляются к множеству достижимых вершин, в строке 20 они добавляются в слайс Y. Поскольку эти вершины еще не были достигнуты на предыдущих шагах (т. е. соответствующие строки в SPT пусты), и вершина k ранее не участвовала в обходе (т. е. $COL(k, SPT) = \emptyset$), то в результате выполнения строки 22 соответствующие

дуги будут добавлены в дерево.

Случай второй: Слайс Y пуст. Докажем, что в слайсе RV помечены '1' все вершины, достижимые из корня. Будем доказывать от противного. Предположим, что $\exists v \notin RV : \exists$ путь $s = v_0, v_1, ..., v_l = v$. Тогда $\exists k(0 \leq k \leq l) : v_k \in RV$ и $v_{k+1} \notin RV$, но это означает, что v_k была добавлена в слайс Y (строки 18-19) и не была исключена из него позднее в строке 13, т.е. слайс $Y \neq \emptyset$ что противоречит условию. □

Замечание 3.3.2. *В данном алгоритме остовное дерево рассматривается на множестве вершин, достижимых из источника s. Поэтому множество достижимых вершин легко восстанавливается по заданному остовному дереву.*

Замечание 3.3.3. *Если дерево на множестве достижимых вершин не задано, то необходимо использовать алгоритм 3.5*

3.3.4. Определение точек синхронизации и оптимизация под выполнение на GPU

Операторы, критичные к синхронизации, расположены в строках 9,14,16 и 19. При этом конструкция из последовательных операторов $whileSOME(WorkSet)$ и $i := STEP(WorkSet)$ заменяется следующией: $i := STEP(WorkSet)\ Whilei > 0$. Таким образом при n итераций цикла исключаются $(n-1)$ операций сложности $log_{64}(n)$ и требующих синхронизацию.

```
 1 procedure InsertEdgeReachability(i,j,s: integer; Var G, ST,
       Desc: Table)
 2 Var
 3    WorkSet, X, RV, Z: Slice ;
 4    k,l: int;
 5 Begin
 6 /* отдельная __global__-процедура*/
 7     CLR(WorkSet);
 8    RV:=COL(s,Desc);
 9    Z:= not RV;
10    X:=COL(i,G); X(j):=1; COL(i,G):=X;
11    (G->col(i))->set(j,1)
```

```
12 /*************** */
13    if (RV(i)=1) and (RV(j)=0) then
14    begin
15 /* отдельная __global__-процедура*/
16       WorkSet(j):=1;
17       Z(j):=0;
18 //     link(i,j,ST,Desc);
19        X:=COL(i,ST); X(j):=1; COL(i,ST):=X;
20
21
22          X:=COL(j,Desc);
23 /* 2D-параллелизм  *переписать*/
24 /*   i, Desc*/
25          w:=ROW(i,Desc);
26          While SOME(w) do
27          begin
28            k:=STEP(w);
29            Y:=COL(k,Desc);
30            Y:=Y or X;
31            COL(k,Desc):=Y;
32          end;
33 /*******************************/
34
35         k:=STEP(WorkSet);
36         While (k>0) do
37         begin
38 /* ?отдельная __global__-процедура?*/
39          X:=COL(k,G);
40          X:= X and Z;
41 /*******************************/
42          if SOME(X) then
43          begin
44 /*  отдельная */
45             WorkSet:= WorkSet or X;
46             Z:= Z and not X ;
47                 Y:=COL(u,ST);
48                 Y:=Y or X; COL(k,ST):=Y;
49                 Y:=X; CLR(Z2);
```

```
50 /**************/
51      w:=ROW(k,Desc);
52    CLR(Z2);
53    l:=STEP(Y);
54    While (l>0) do
55    begin
56     Z1:=COL(l,Desc);
57     Z2:=Z2 or Z1;
58     l:=STEP(Y);
59    end;
60    l:=STEP(w);
61    While(l>0) do
62    begin
63     (Desc->col(i))->OR(Z2)
64     l:=STEP(w);
65    end;
66 ////////////////////////
67            end;
68            k:=STEP(WorkSet);
69         end;
70     end;
71 End;
```

Тестирование инкрементального алгоритма на графических ускорителях

Тестирование проводилось на видеокарте NVIDIA GEFORCE 920m. В ходе тестирования время работы динамического ассоциативного алгоритма (D_A) сравнилось со временем работы статического ассоциативного алгоритма (S_A). А также подсчитывались изменение числа достижимых вершин после добавления дуги и число дуг, исходящих из вершин, которые стали достижимыми после добавления дуги. Отметим, что эти величины определяют число итераций ассоциативного (D_A) и последовательного (D_S) динамических алгоритмов, соответственно.

Как видно из рисунка 3.10, время работы статического алгоритма зависит от мощности обновленного множества достижимых вершин ($|VRout|$), а время

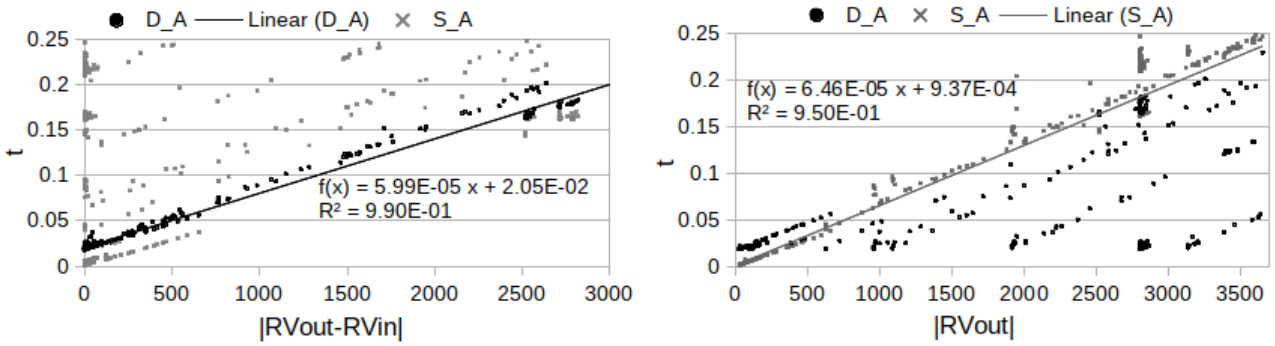


Рис. 3.10. Сравнение времени работы динамического ассоциативного алгоритма со временем работы статического ассоциативного алгоритма.

работы динамического алгоритма линейно зависит от изменения мощности множества достижимых вершин ($|VRout| - |VRin|$). Отметим, что динамический алгоритм выполняется до 10 раз быстрее, чем статический.

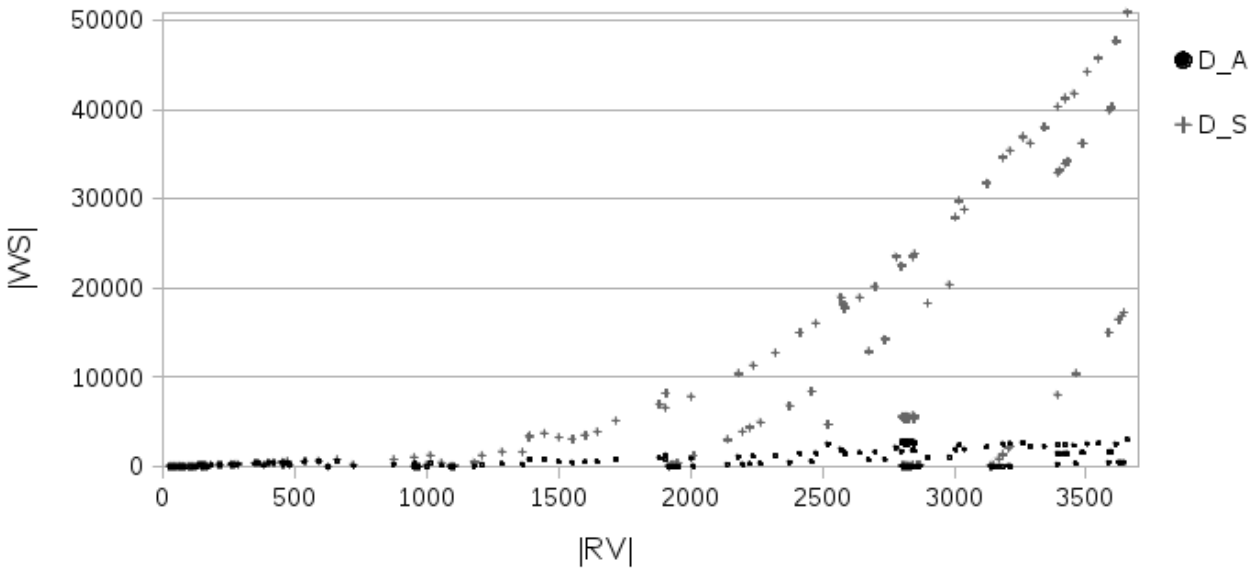


Рис. 3.11. Количество итераций (|WorkSet|), необходимых для обработки графа после добавления дуги, заходящей в вершину v.

На рисунке 3.11 приводится количество итераций, которое требуется для обработки заданного графа после добавления новой дуги как ассоциативным динамическим алгоритмом, так и последовательным динамическим алгоритмом, в зависимости от мощности множества достижимых вершин. Из этого рисунка видно, что число итераций для динамического ассоциативного алгоритма и динамического последовательного алгоритма могут значительно различаться в

зависимости от структуры обрабатываемого графа. Полученное ускорение зависит от усредненной степени вершин графа и исходного множества достижимых вершин.

3.4. Выводы к третьей главе

В этой главе представлены рекомендации по оптимизации ассоциативных алгоритмов для выполнения на графических ускорителях. Эти рекомендации учитывают различие архитектур и являются универсальными для алгоритмов, написанных на языке STAR. Также приведены реализации нескольких ассоциативных алгоритмов на графических ускорителях.

Ассоциативный алгоритм выполняется за время, сравнимое с временем работы классической реализации алгоритма Уоршалла на графических ускорителях. В то же время, адаптация Star-алгоритма к архитектуре GPU по универсальным рекомендациям приводит к большему ускорению, чем различные оптимизации реализации алгоритма на GPU. Адаптированный ассоциативный алгоритм Уоршалла имеет не только теоретическую оценку сложности $O(N)$, но и выполняется за время, близкое к линейному.

В отличии от статических алгоритмов на графах, динамические алгоритмы трудно поддаются распараллеливанию. В то время как ассоциативные алгоритмы - параллельные. Поэтому динамические алгоритмы, приведенные в этой главе сравнивались со статическими ассоциативными алгоритмами.

При тестировании ассоциативного параллельного алгоритма для динамической обработки дерева кратчайших путей после добавления новой дуги к ориентированному графу было получено, что на R-MAT графах в более чем 50% случаев при добавлении дуги кратчайшие расстояния не изменяются. А количество аффектных вершин не превышает 5 в 99% случаев. Таким образом динамический алгоритм дает выигрыш на несколько порядков по сравнению со статическим ассоциативным алгоритмом.

Тестирование ассоциативной версии динамического алгоритма Рамалингама для проблемы достижимости в потоковых графах были получены похожие результаты: ассоциативный динамический алгоритм выполняется быстрее, чем

статический ассоциативный алгоритм, поскольку обрабатывает только необходимый подграф. Также ему требуется гораздо меньше итераций для обработки графа, чем последовательному алгоритму Рамалингама, поскольку обработка идет по вершинам, а не по дугам.

Заключение

В диссертационной работе представлена реализация Star машины модели ассоциативных вычислений на графических ускорителях.

В первой главе представлен обзор развития ассоциативных параллельных архитектур от первого коммерчески успешного процессора STARAN до настоящего времени. Все реализованные архитектуры были построены под решение конкретных задач, которые не могли быть эффективно решены на системах другой архитектуры: ASPRO для задач контроля воздушного движения, IMX2 для машинного перевода, FastTracker Processor для детектора ATLAS большого адронного коллайдера.

Так же, как в случае параллельных вычислений на машинах типа PRAM, в случае ассоциативных вычислений необходимо решать фундаментальные проблемы и вычислительной техники, и алгоритмов, и программирования. Развитие продолжается в трех направлениях: разработка аппаратного обеспечения для ассоциативных параллельных вычислений, ассоциативные процессоры и системы, разработка ассоциативных моделей и алгоритмов для этих моделей, реализация ассоциативных моделей на существующем оборудовании.

Во второй главе представлена реализация STAR-машины на графических ускорителях с помощью технологии CUDA. Базовые операции языка Star представлены в виде процедур, выполняемых на графическом ускорителе за константное время или за время $O(\lceil \log_{64}(n) \rceil)$, близкое к константному.

В реализацию STAR-машины на GPU был добавлен модуль с процедурами ввода/вывода данных в форматах *.gr (R-MAT графы, синтетические графы) и Autonomus System(графы протоколов интернета) с переводом в любой из внутренних форматов STAR-машины (матрица смежности, матрица весов или список дуг).

Реализована на графических ускорителях библиотека стандартных ассоциативных алгоритмов с оценкой времени, совпадающей с теоретической. Произведено сравнение времени работы библиотеки стандартных ассоциативных параллельных алгоритмов с временем работы подобных алгоритмов из библиотек C++ STL и CUDA thrust. Созданная реализация базовых ассоциативных

алгоритмов дает существенное ускорение по сравнению с библиотекой STL (для векторов с более чем 5000 элементов до 2-х порядков) и выигрывает в 1,5-2 раза в производительности у библиотеки CUDA thrust.

Также представлены рекомендации по оптимизации ассоциативных алгоритмов для выполнения на графических ускорителях. Эти рекомендации учитывают различие архитектур и являются универсальными для алгоритмов, написанных на языке STAR.

Для достижения построения эффективной реализации STAR-машины на графических ускорителях с помощью технологии CUDA были решены следующие задачи:

- построена реализация базовых операций языка Star на графическом ускорителе;
- для увеличения эффективности реализации выделены операции языка Star, критичные к синхронизации;
- реализована на графическом ускорителе библиотека стандартных процедур языка Star;
- на основании анализа существующих форматов входных/выходных данных для тестовых графов (более 5000 вершин) разработан модуль ввода/вывода данных отобранных форматов во внутреннее представление реализации Star-машины;
- обоснована эффективность реализации Star-машины как оценкой теоретической сложности процедур реализации, так и практическим сравнением времени работы с временем работы аналогов (для доказательства эффективности реализации базовых операций сравниваются время выполнения ассоциативной версии алгоритма Уоршалла с временем выполнения других параллельных реализаций этого алгоритма; для обоснования эффективности реализации библиотеки стандартных процедур проводится сравнение времени работы базовых процедур с временем работы аналогов из библиотек STL и CUDA thrust);

- разработаны методы оптимизации ассоциативных алгоритмов для выполнения на графических ускорителях, учитывающие различия Star-машины и GPU.

В третьей главе приведены реализации нескольких ассоциативных алгоритмов на графических ускорителях.

Ассоциативный алгоритм выполняется за время, сравнимое с временем работы классической реализации алгоритма Уоршалла на графических ускорителях. В то же время, адаптация Star-алгоритма к архитектуре GPU по универсальным рекомендациям приводит к большему ускорению, чем различные оптимизации реализации алгоритма на GPU. Адаптированный ассоциативный алгоритм Уоршалла имеет не только теоретическую оценку сложности $O(N)$, но и выполняется за время, близкое к линейному.

В отличии от статических алгоритмов на графах, динамические алгоритмы трудно поддаются распараллеливанию. В то время как ассоциативные алгоритмы - параллельные. Поэтому динамические алгоритмы, приведенные в этой главе сравнивались со статическими ассоциативными алгоритмами.

При тестировании ассоциативного параллельного алгоритма для динамической обработки дерева кратчайших путей после добавления новой дуги к ориентированному графу было получено, что на R-MAT графах в более чем 50% случаев при добавлении дуги кратчайшие расстояния не изменяются. А количество аффектных вершин не превышает 5 в 99% случаев. Таким образом динамический алгоритм дает выигрыш на несколько порядков по сравнению со статическим ассоциативным алгоритмом.

Тестирование ассоциативной версии динамического алгоритма Рамалингама для проблемы достижимости в потоковых графах были получены похожие результаты: ассоциативный динамический алгоритм выполняется быстрее, чем статический ассоциативный алгоритм, поскольку обрабатывает только необходимый подграф. Также ему требуется гораздо меньше итераций для обработки графа, чем последовательному алгоритму Рамалингама, поскольку обработка идет по вершинам, а не по дугам.

Таким образом можно заключить, что реализация Star-машины на гра-

фических ускорителях является актуальной, эффективной и дает возможность использовать ассоциативные алгоритмы на практике. При этом в отличии от распараллеливания алгоритмов, не возникает трудности с определением точек синхронизации. Это связано с тем, что в языке Star выявлены операции, критичные к синхронизации, и выработаны методы оптимизации алгоритмов для выполнение на GPU.

В качестве перспективы дальней разработки тематики рассматривается приложение Star-машины и ее реализации на GPU к задачам биоинформатики. Эти задачи отличаются чертами для которых ассоциативная обработка является оптимальной:

- поиск по большому массиву данных: от десятков тысяч до нескольких миллиардов нуклеотидных оснований;
- ограниченный алфавит: A,T,C,G для нуклеотидов (3 бита) и 20 для аминокислот (5 битов);
- управляющие слайсы дают возможность троичного поиска.

Список сокращений и терминов

RAM: Random Access Memory - запоминающее устройство с произвольным доступом.

CAM: Content Adresseble Memory - память, адресуемая по содержимому (ассоциативная память).

TCAM: Ternary Content Addressable Memory -троичная ассоциативная память.

АП(AP): ассоциативный процессор.

ПЭ (PE): процессорный элемент.

VLSI(СБИС): very-large-scale integration - сверхбольшие интегральные схемы.

DRAM: динамической памяти с произвольным доступом.

LHC (БАК): Large Hadron Collider - большой адронный коллайдер.

FTK: FastTrack Procesor - ассоциативный процессор, созданный для детектора ATLAS LHC.

ПУУ: последовательное устройство управления.

УАО: управляющее ассоциативное устройство.

IS: instruction stream - ПУУ.

Список литературы

1. Снытникова Т. В., Непомнящая А. Ш. Решение задач на графах с помощью STAR-машины, реализуемой на графических ускорителях // Прикладная дискретная математика. 2016. Vol. 3(33). P. 98–115.
2. Снытникова Т. В., Непомнящая А. Ш. О реализации на GPU базовых ассоциативных процедур языка STAR // Марчуковские научные чтения. Vol. 2017. 2017.
3. Непомнящая А. Ш., Снытникова Т. В. Ассоциативный параллельный алгоритм для динамической обработки дерева кратчайших путей после добавления новой дуги // Прикладная дискретная математика. 2019. Vol. 46. P. 49–62.
4. Непомнящая А. Ш., Снытникова Т. В. Ассоциативная версия инкрементального алгоритма Рамалингама для решения проблемы достижимости в потоковых графах с одним источником // Вестн. Том. гос. ун-та. Управление, вычислительная техника и информатика. 2021. Vol. 54. P. 86–96.
5. Снытникова Т. В., Непомнящая А. Ш. Реализация на GPU инкрементального алгоритма Рамалингама для динамической обработки потоковых графов с одним источником // Марчуковские научные чтения. Vol. 2020. 2020.
6. Krikelis A., Weems C. C. Associative processing and processors // Computer. 1994. — Nov. Vol. 27, no. 11. P. 12–17.
7. Pagiamtzis K., Sheikholeslami A. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey // IEEE Journal of Solid-State Circuits. 2006. — March. Vol. 41, no. 3. P. 712–727.
8. Kocak T., Basci F. A power-efficient TCAM architecture for network forwarding tables // Journal of Systems Architecture. 2006. Vol. 52, no. 5. P. 307 – 314. URL: `http://www.sciencedirect.com/science/article/pii/S1383762105001451`.
9. Noda H., Inoue K., Kuroiwa M. et al. A cost-efficient high-performance dynamic TCAM with pipelined hierarchical searching and shift redundancy architecture.

2005. — 02. Vol. 40. P. 245 – 253.

10. Potter J. L. Associative Computing: A Programming Paradigm for Massively Parallel Computers. Perseus Publishing, 1991. ISBN: 0306439875.
11. Jalaleddine S. M. Associative Memories and Processors: The Exact Match Paradigm // Journal of King Saud University - Computer and Information Sciences. 1999. Vol. 11, no. Supplement C. P. 45 – 67. URL: http://www.sciencedirect.com/science/article/pii/S1319157899800032.
12. Rudolph J. A. A Production Implementation of an Associative Array Processor: STARAN // Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I. AFIPS '72 (Fall, part I). New York, NY, USA: ACM, 1972. P. 229–241. URL: http://doi.acm.org/10.1145/1479992.1480023.
13. Foster C. C. Content addressable parallel processors / Caxton C. Foster. Van Nostrand Reinhold New York, 1976. P. xiii, 233 p. :. ISBN: 0442224338.
14. Batcher K. E. Design of a Massively Parallel Processor // IEEE Transactions on Computers. 1980. — Sept. Vol. C-29, no. 9. P. 836–840.
15. Batcher K. E. STARAN Parallel Processor System Hardware // Proceedings of the May 6-10, 1974, National Computer Conference and Exposition. AFIPS '74. New York, NY, USA: ACM, 1974. P. 405–410. URL: http://doi.acm.org/10.1145/1500175.1500260.
16. GER-15637A. STARAN S APPLE Programming Manual. GOODYEAR AEROSPACE CORPORATION, OHIO, 1973. — September.
17. Davis E. W. STARAN Parallel Processor System Software // Proceedings of the May 6-10, 1974, National Computer Conference and Exposition. AFIPS '74. New York, NY, USA: ACM, 1974. P. 17–22. URL: http://doi.acm.org/10.1145/1500175.1500179.
18. Batcher K. E. Bit-Serial Parallel Processing Systems // IEEE Transactions on Computers. 1982. — May. Vol. C-31, no. 5. P. 377–384.
19. Uhr L. M. Algorithm-Structured Computer Arrays and Networks: Architectures and Processes for Images, Precepts, Models, Information. Orlando, FL, USA: Academic Press, Inc., 1984. ISBN: 0127069607.
20. Wang H., Walker R. A. Implementing a scalable ASC processor // Parallel and Distributed Processing Symposium, 2003. Proceedings. International. 2003. P. 7

pp.–.

21. Mingxian J. Associative Operations from MASC to GPU // PDPTA'15 – The 21st International Conference on Parallel and Distributed Processing Techniques and Applications. Las Vegas: CSREA Press, 2015. P. 388–393.
22. Higuchi T., Kitano H., Furuya T. et al. IXM2: A Parallel Associative Processor for Knowledge Processing. // AAAI / Ed. by T. L. Dean, K. McKeown. AAAI Press / The MIT Press, 1991. P. 296–303.
23. Higuchi T., Furuya T., Handa K. et al. IXM2: A Parallel Associative Processor // Proceedings of the 18th Annual International Symposium on Computer Architecture. ISCA '91. New York, NY, USA: ACM, 1991. P. 22–31.
24. Higuchi T., Furuya T., Handa K., Kokubu A. Initial evaluation of a parallel associative processor IXM2 // Microprocessing and Microprogramming. 1991. Vol. 31, no. 1. P. 89 – 92.
25. Kitano H. Speech-to-Speech Translation: A Massively Parallel Memory-Based Approach // Speech-to-Speech Translation: A Massively Parallel Memory-Based Approach. Boston, MA: Springer US, 1994. P. 135–155.
26. Kitano H., Higuchi T., Tomita M. Massively parallel spoken language processing using a parallel associative processor IXM2 // The First International Conference on Spoken Language Processing, ICSLP 1990, Kobe, Japan, November 18-22, 1990. 1990.
27. Oi K., Sumita E., Furuse O. et al. Real-time Spoken Language Translation Using Associative Processors // Proceedings of the Fourth Conference on Applied Natural Language Processing. ANLC '94. Stroudsburg, PA, USA: Association for Computational Linguistics, 1994. P. 101–106. URL: https://doi.org/10.3115/974358.974381.
28. Smith D., Hall J., Miyake K. Rutger's CAM2000 Chip Architecture. LCSR-TR-. Rutgers University, Department of Computer Science, Laboratory for Computer Science Research, 1993. URL: https://books.google.ru/books?id=k1tDAQAAMAAJ.
29. Smith D. E., Hall J. S., Miyake K. Rutger's CAM2000 chip architecture: Tech. rep.: Rutgers University, 1993. URL: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19930017905.pdf.

30. Hsu C., Smith D., Levy S. Linear-C: A Data-Parallel Extension to C: Tech. Rep. LCSR-TR-273: Computer Science Department, Rutgers University, 1996.
31. Volpi G. Associative Memory computing power and its simulation.: Tech. Rep. ATL-DAQ-PROC-2014-018. Geneva: CERN, 2014. — Jun. URL: https://cds.cern.ch/record/1712666.
32. ATLAS Experiment at CERN. https://atlas.cern/. 2018.
33. Annovi A., Beretta M., Bossini E. et al. Associative memory design for the FastTrack processor (FTK) at ATLAS // Real Time Conference (RT), 2010 17th IEEE-NPSS. 2010. — May. P. 1–3.
34. Collaboration T. A. The ATLAS Experiment at the CERN Large Hadron Collider // Journal of Instrumentation. 2008. Vol. 3, no. 08. P. S08003. URL: http://stacks.iop.org/1748-0221/3/i=08/a=S08003.
35. Collaboration T. A. Expected performance of the ATLAS experiment: detector, trigger and physics. Geneva: CERN, 2009. URL: https://cds.cern.ch/record/1125884.
36. Cho A. The Discovery of the Higgs Boson // Science. 2012. Vol. 338, no. 6114. P. 1524–1525. http://science.sciencemag.org/content/338/6114/1524.full.pdf. URL: http://science.sciencemag.org/content/338/6114/1524.
37. Maznas, Ioannis. FTK: The hardware Fast TracKer of the ATLAS experiment at CERN // EPJ Web Conf. 2017. Vol. 137. P. 12001. URL: https://doi.org/10.1051/epjconf/201713712001.
38. Biesuz N., Citraro S., Donati S. et al. Highly parallelized pattern matching execution for atlas event real time reconstruction // on IEEE Transactions on Nuclear Science (TNS). 2016.
39. Pagiamtzis K., Sheikholeslami A. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey // IEEE Journal of Solid-State Circuits. 2006. — March. Vol. 41, no. 3. P. 712–727.
40. Dell'Orso M., Ristori L. VLSI Structures Track Finding // Nucl. Instr. and Meth. A. 1989. Vol. 278. P. 436–440.
41. Wissing C., Baird A., Baldinger R. et al. Performance of the H1 Fast Track Trigger: Operation and Commissioning Results // Proceedings of the 14th IEEE-NPSS Conference on Real Time. RTC'05. Washington, DC, USA: IEEE

Computer Society, 2005. P. 233–236. URL: http://dl.acm.org/citation.cfm?id=1867821.1867874.

42. Bardi A. et al. The CDF online silicon vertex tracker // Nucl. Instrum. Meth. 2002. Vol. A485. P. 178–182.
43. Annovi A., Bardi A., Bitossi M. et al. A VLSI Processor for Fast Track Finding Based on Content Addressable Memories // IEEE Transactions on Nuclear Science. 2006. — Aug. Vol. 53, no. 4. P. 2428–2433.
44. Masi S. Periodic Report Summary 1 - FTK (Fast Tracker for Hadron Collider Experiments): Tech. Rep. 324318. Italy: Industry-Academia Partnerships and Pathways, 2015. — July. URL: http://cordis.europa.eu/result/rcn/167746_en.html.
45. Asbah N. A hardware fast tracker for the ATLAS trigger // Physics of Particles and Nuclei Letters. 2016. — Sep. Vol. 13, no. 5. P. 527–531. URL: https://doi.org/10.1134/S1547477116050368.
46. Maznas I. FTK: The hardware Fast TracKer of the ATLAS experiment at CERN: Tech. Rep. ATL-DAQ-PROC-2016-033. Geneva: CERN, 2016. — Nov. URL: https://cds.cern.ch/record/2233653.
47. Sotiropoulou C. L., Gkaitatzis S., Annovi A. et al. A Multi-Core FPGA-Based 2D-Clustering Implementation for Real-Time Image Processing // IEEE Trans. Nucl. Sci. 2014. Vol. 61, no. 6. P. 3599–3606.
48. Okumura Y., Liu T., J.Olsen et al. ATCA-based ATLAS FTK input interface system // Journal of Instrumentation. 2015. Vol. 10, no. 04. P. C04032. URL: http://stacks.iop.org/1748-0221/10/i=04/a=C04032.
49. Walker R. A., Potter J. L., Wang Y., Wu M. Implementing Associative Processing: Rethinking EarlierArchitectural Decisions // Proceedings of the 15th International Parallel &Amp; Distributed Processing Symposium. IPDPS '01. Washington, DC, USA: IEEE Computer Society, 2001. P. 195–. URL: http://dl.acm.org/citation.cfm?id=645609.662305.
50. Wu M., Walker R. A., Potter J. L. Implementing Associative Search and Responder Resolution // Proceedings of the 16th International Parallel and Distributed Processing Symposium. IPDPS '02. Washington, DC, USA: IEEE Computer Society, 2002. P. 179–. URL: http://dl.acm.org/citation.cfm?id=645610.

757957.

51. Wang H., Walker R. A. Implementing a scalable ASC processor // Proceedings International Parallel and Distributed Processing Symposium. 2003. — April. P. 7 pp.–.
52. Wang H., Xie L., Wu M., Walker R. A. A scalable associative processor with applications in database and image processing // 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings. 2004. P. 259–.
53. Implementing a Multiple-Instruction-Stream Associative MASC Processor // Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS). 2006. P. 460–465.
54. Nepomnyashchaya A. S. Star — a language for associative and parallel computation with vertical processing // Cybernetics and Systems Analysis. 1991. Vol. 28, no. 4. P. 573–579. URL: http://dx.doi.org/10.1007/BF01124994.
55. Nepomniaschaya A. S. Basic associative parallel algorithms for vertical processing systems // Bulletin of the Novosibirsk Computing Center. 2009. Vol. 9. P. 63–77.
56. Ng K., of Technology. Computer Engineering R. I. Novel Low Power CAM Architecture. Rochester Institute of Technology, 2008. ISBN: 9780549725077. URL: https://books.google.ru/books?id=IOqykFjUohQC.
57. Xu W., Zhang T., Chen Y. Design of spin-torque transfer magnetoresistive RAM and CAM/TCAM with high sensing and search Speed. 2010. — 02. Vol. 18. P. 66 – 74.
58. Imani M., Rahimi A., Rosing T. S. Resistive configurable associative memory for approximate computing // 2016 Design, Automation Test in Europe Conference Exhibition (DATE). 2016. — March. P. 1327–1332.
59. Стемпковский А. Л., Климов А. В., Левченко Н. Н., Окунев А. С. Методы адаптации параллельной потоковой вычислительной системы под задачи отдельных классов // Информационные Технологии И Вычислительные Системы. 2009. Vol. 3. P. 12–21.
60. Змеев Д. Средства проектирования высокопроизводительных потоковыхвычислительных систем // Проблемы Разработки Перспективных Микро- И Наноэлектронных Систем (МЭС). 2016.

Vol. 2. P. 159–163.

61. Snytnikova T. V., Snytnikov A. V. Implementation of the STAR-machine on GPU // NCC Bulletin. 2016. Vol. 39. P. 51–60.
62. 9th DIMACS Implementation Challenge. http://www.dis.uniroma1.it/challenge9/download.shtml. 2010.
63. GraphHPC-1.0. http://www.dislab.org/GraphHPC-2015/contest/GraphHPC-1.0.tgz. 2015.
64. Leskovec J., Krevl A. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. 2014. — jun.
65. Уоррен Г. Алгоритмические трюки для программистов:. Вильямс, 2007. ISBN: 9785845905727.
66. Страуструп Б. Язык программирования C++. Специальное издание. Бином, 2012. ISBN: 978-5-7989-0425-9.
67. Realeases - thrust/thrust - GitHub. 2017. URL: https://github.com/thrust/thrust/releases.
68. Nepomniaschaya A. S. Solution of Path Problems Using Associative Parallel Processors // 1997 International Conference on Parallel and Distributed Systems (ICPADS '97), 11-13 December 1997, Seoul, Korea, Proceedings. 1997. P. 610–617.
69. Harish P., Narayanan P. Accelerating large graph algorithms on the GPU using CUDA // Lecture Notes in Computer Science. 2007. Vol. 4873. P. 197–208.
70. Погорелый С., Трибрат М., Бойко Ю., Грязнов Д. Реализация алгоритма Флойда-Уоршалла для программно-аппаратной платформы CUDA // Управляющие системы и машины. 2011. Vol. 5. P. 64–67,72.
71. Katz G. J., Kider J. T., Jr. All-pairs Shortest-paths for Large Graphs on the GPU // Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware. GH '08. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008. P. 47–55.
72. Lund B. D., Smith J. W. A Multi-Stage CUDA Kernel for Floyd-Warshall // CoRR. 2010. Vol. abs/1001.4108.
73. Nepomniaschaya A. S., Dvoskina M. A. A simple implementation of Dijkstra's shortest path algorithm on associative parallel processors // Fundamenta Infor-

maticae. 2000. Vol. 43. P. 227–243.

74. Reps T. Program Analysis via Graph Reachability // Proceedings of the 1997 International Symposium on Logic Programming. ILPS '97. Cambridge, MA, USA: MIT Press, 1997. P. 5–19.
75. Zhang Q., Su Z. Context-Sensitive Data-Dependence Analysis via Linear Conjunctive Language Reachability // SIGPLAN Not. 2017. — jan. Vol. 52, no. 1. P. 344–358. URL: https://doi.org/10.1145/3093333.3009848.
76. Hanauer K., Henzinger M., Schulz C. Fully Dynamic Single-Source Reachability in Practice: An Experimental Study. 2020. — 01. P. 106–119. ISBN: 978-1-61197-600-7.
77. Jin R., Hong H., Wang H. et al. Computing Label-Constraint Reachability in Graph Databases // Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, 2010. P. 123–134. URL: https://doi.org/10.1145/1807167.1807183.
78. Ramalingam G. Incremental algorithms for reducible flowgraphs. Springer-Verlag Berlin Heidelberg, 1996. Vol. 1089 of Lecture Notes in Computer Science. P. 149–155.
79. Sleator D., Tarjan R. A Data Structure for Dynamic Trees // Journal of Computer and System Sciences. 1983. Vol. 26. P. 362–391.