Санкт–Петербургский государственный университет
Кафедра компьютерного моделирования и многопроцессорных систем

ФАТЬКИНА Анна Игоревна

Магистерская диссертация

Разработка систем поддержки многопоточных вычислений платформы Global Neutrino Analysis

Направление 02.04.02
Фундаментальная информатика и информационные технологии
Основная образовательная программа магистратуры
Вычислительные технологии

Научные руководители:
профессор, кафедра компьютерного
моделирования и многопроцессорных систем,
к.т.н. Дегтярев Александр Борисович

научный сотрудник,
Лаборатория ядерных проблем ОИЯИ,
к.ф.-м.н. Гончар Максим Олегович

Санкт-Петербург
2019 г.

Содержание

Используемые термины и обозначения

GNA (global neutrino analysis) — платформа для статистического анализа данных.

Трансформация — узел вычислительного графа GNA.

cuGNA — библиотека, входящая в состав GNA, реализующая поддержку вычислений на графических процессорах.

CUDA (Compute Unified Device Architecture) — архитектура параллельных вычислений с использованием GPU.

GPU (graphics processing unit) — графический процессор.

CPU (central processing unit) — центральный процессор.

RAM (Random Access Memory) — оперативная память, используется при работе алгоритмов для CPU.

Матрица PMNS (Pontecorvo–Maki–Nakagawa–Sakata) — матрица смешивания нейтрино.

ROOT — фреймворк, разработанный в CERN, для анализа данных.

Постановка задачи

Основной задачей данной работы является ускорение вычислений алгоритмов статистического анализа нейтринных экспериментов на платформе GNA. Для этого рассмотрена возможность частичного переноса вычислений на GPU. Задача рассматривается в рамках моделей нейтринных экспериментов JUNO и Daya Bay, для обработки данных которых используется GNA.

Вспомогательная задача — разработка библиотеки поддержки вычислений на GPU для общих вычислений в GNA. Платформа GNA использует принципы data flow. Вычисления представлены в виде графа, в котором каждый узел графа — функция — компилируется отдельно. Узлы графа могут соединяться в произвольном порядке. Переключение производится со стороны интерфейса без непосредственной компиляции всего графа. Необходимо было реализовать библиотеку поддержки GPU таким образом, чтобы сохранить общую схему работы платформы. При решении такой задачи неизбежно возникают проблемы синхронизации данных меж-

ду устройствами. Обмен данными необходимо производить незаметно для пользователя на стороне ядра фреймворка.

Таким образом, общая задача состоит в реализации поддержки вычислений на графических процессорах с сохранением гибкости построения моделей в GNA и увеличением производительности на основных задачах, решаемых фреймворком.

Обзор литературы

При подготовке данной магистерской диссертации использовался ряд источников для ознакомления как с физической стороной изучаемой области, так и с технической. В процессе подготовки работы возникла необходимость общего рассмотрения задач JUNO и Daya Bay. В источнике [1] приведено описание эксперимента JUNO и его задач. В [2] приведен обзор эксперимента Daya Bay, а в [3] описаны подробно описаны некоторые задачи, решаемые Daya Bay.

Явления, изучаемые в этих экспериментах, описаны в трудах [4, 5], например, нейтринные осцилляции. Явление нейтринных осцилляций было открыто как решение задачи о солнечных нейтрино. Впервые солнечные нейтрино были обнаружены в середине прошлого века. Подробнее о солнечных нейтрино написано в статье 1964 года [6].

Работа посвящена разработке библиотеки вычислений на графических процессорах в платформе GNA, разрабатываемой в Лаборатории ядерных проблем ОИЯИ (ЛЯП ОИЯИ). Общая информация о проекте может быть найдена на домашней странице GNA [7]. Информация об авторах проекта и некоторых других нейтринных проектах, которыми занимаются в ЛЯП ОИЯИ, может быть найдена на странице [8]. То, как использовать этот фреймворк описано в [9, 10].

GNA использует платформу ROOT и, в частности, библиотеку PyRoot. Информация об этом продукте находится на официальной странице [11]. Также GNA использует библиотеку Eigen [12], предоставляющую инструменты работы с матрицами и набор математических функций.

Библиотека поддержки вычислений на GPU реализована средствами CUDA [13]. Ряд рекомендаций и описания практик применения CUDA на

реальных задачах описаны в [14, 15].

Впервые поддержка GPU для платформы GNA была представлена год назад на конференции ICCSA 2018 в статье [16]. Далее, обновленный статус работы был упомянут на конференции CHEP 2018 [17].

Введение

Нейтрино — элементарная частица, открытая совсем недавно, в середине XX века. На настоящий момент, область нейтрино совсем мало изучена, и ученые со всего мира пытаются найти ответы на фундаментальные вопросы об этой частице. В разных уголках планеты строятся детекторы, которые позволяют детектировать нейтринные потоки от атомных электростанций, из космоса, гео-нейтрино. Среди них телескоп на озере Байкал (эксперимент Baikal-GVD), обсерватория на антарктической станции (эксперимент IceCube), детекторы, помещенные внутрь горы в Китае (JUNO, Daya Bay) и другие. Эти детекторы имеют множество сенсоров, которые улавливают излучение, полученное в результате взаимодействия нейтрино с веществом, находящимся в детекторе.

Платформа GNA разработана для статистического анализа экспериментальных данных. Изначально рассматривалось применение GNA в экспериментах JUNO и Daya Bay. Тем не менее, эта платформа является расширяемой, а также имеет набор общих методов, подходящих для более широкого круга задач.

Методы статистического анализа требуют значительных временных затрат. Например, профилирование функции правдоподобия требует сотни итераций минимизации по множеству параметров, каждая из которых занимает часы. Поэтому возникает необходимость увеличения производительности алгоритмов, в том числе, аппаратными способами.

В данной работе была рассмотрена возможность применения графических процессоров для ускорения обработки данных. Была реализована библиотека поддержки вычислений на GPU для платформы GNA. Библиотека была протестирована на некоторых функциях, используемых в статистическом анализе, и показала свою эффективность.

Глава 1. Расчеты физических моделей

1.1 Физическая задача

Последние несколько десятилетий ученые пытаются разгадать тайны нейтрино. Первые нейтрино были обнаружены в середине XX века. К настоящему моменту осталось еще много вопросов, на которых пока нет ответа. Например, является ли антинейтрино и нейтрино одной частицей, каковы абсолютные массы нейтрино, и многие другие. Исследования нейтрино исторически приводили к важным открытиям в физике элементарных частиц. На текущий момент они становятся все более актуальными, потому что они привели к экспериментальным указаниям на существование физики, не описываемой Стандартной моделью физики элементарных частиц.

Нейтрино — это фермион, который участвует только в слабом взаимодействии частиц, а потому его сложно обнаружить. Различают три вида нейтрино — ν_1, ν_1, ν_3, массы которых равны m_1, m_2 и m_3 соответственно, однако их точные значения неизвестны. Такие состояния нейтрино называются массовыми.

Эти частицы взаимодействуют с заряженными лептонами (электроном, мюоном и тау) с интенсивностями взаимодействия, определяемыми элементами $V_{\alpha_i, i=1..3}$ матрицы лептонного смешения V, названной в честь Понтекорво, Маки, Накагавы и Сакаты (PMNS-матрица). На данный момент известно, что эта матрица не является диагональной [5].

Существует также другая классификация нейтрино. Согласно Стандартной модели, существует электронное, мюонное и тау нейтрино, а также обратные к ним античастицы. Эти состояния называются флейворными. Флейворные нейтрино не имеют определенной массы. Массовые и флейворные состояния нейтрино связаны формулой

$$|\nu_\alpha\rangle = \sum_{i=1}^{3} V_{\alpha_i} |\nu_i\rangle,$$

где ν_α — флейворное состояние, ν_i — массовое состояние, V_{α_i} — элемент

матрицы PMNS.

Матрица PMNS выглядит следующим образом:

$$U = \begin{pmatrix} c_{12}c_{13} & s_{12}c_{13} & s_{13}e^{-i\delta_{\text{CP}}} \\ -s_{12}c_{23} - c_{12}s_{13}s_{23}e^{i\delta_{\text{CP}}} & c_{12}c_{23} - s_{12}s_{13}s_{23}e^{i\delta_{\text{CP}}} & c_{13}s_{23} \\ s_{12}s_{23} - c_{12}s_{13}c_{23}e^{i\delta_{\text{CP}}} & -c_{12}s_{23} - s_{12}s_{13}c_{23}e^{i\delta_{\text{CP}}} & c_{13}c_{23} \end{pmatrix} P,$$

где $c_{ij} = \cos\theta_{ij}$, $s_{ij} = \sin\theta_{ij}$, а P — матрица, значение которой зависит от того, является ли нейтрино частицей Дирака или Майорана. Матрица PMNS является унитарной и параметризуется тремя углами смешивания θ_{ij} и одной фазой δ_{CP} [4].

Обращаясь к истокам исследований нейтрино, вспомним задачу о солнечных нейтрино [6]. Загадка солнечных нейтрино состояла в том, что поток нейтрино, идущий от Солнца, был меньше теоретически предсказанного. В качестве решения этой задачи было предположено явление нейтринных осцилляций. Это явление заключается в том, что нейтрино, произведенное как нейтрино с определенным флейвором, через какое-то расстояние может быть обнаружено как нейтрино с отличным от изначального флейвором. Часть потока электронных нейтрино переходили в мюонные и тау-нейтрино, достигая Земли. Потому наблюдаемый от Солнца поток был в два раза меньше ожидаемого. Эффект нейтринных осцилляций возможен только в том случае, если флейворные состояния представляют собой суперпозицию массовых состояний нейтрино.

Частоты осцилляций нейтрино определяется значениями расщеплений масс $\Delta m^2_{ij} = m^2_i - m^2_j$, где m_i — это значения масс нейтрино. Формула для вычисления вероятности осцилляций в вакууме выглядит следующим образом:

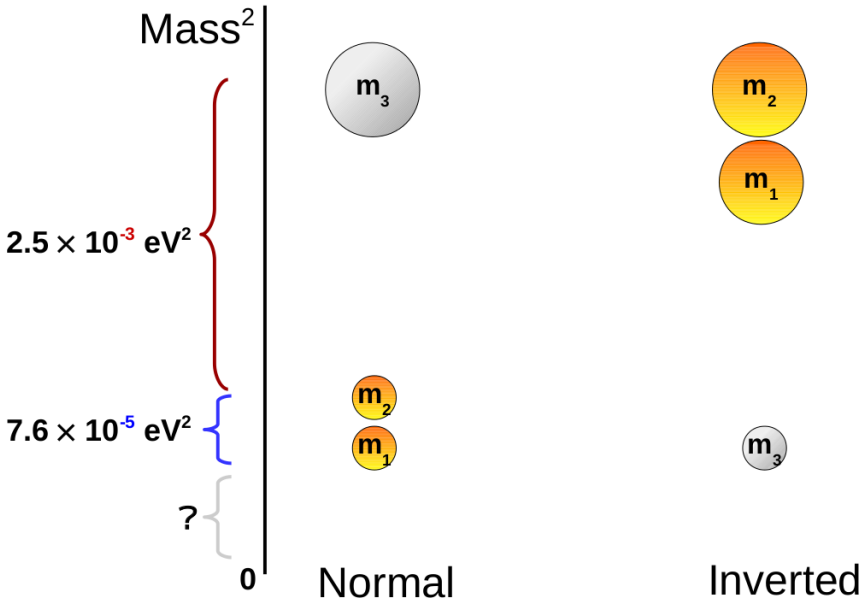

Рис. 1: Расположение значений масс нейтрино в предположениях нормальной (Normal) и обратной (Inverted) иерархий масс [18].

$$P(\nu_\alpha \to \nu_\beta) = \delta_{\alpha\beta} - 4\sum_{i>j} \mathrm{Re}(V^*_{\alpha i} V_{\beta i} V_{\alpha j} V^*_{\beta j}) \sin^2 \frac{\Delta m^2_{ij} L}{4E_\nu} + \\ +2\sum_{i>j} \mathrm{Im}(V^*_{\alpha i} V_{\beta i} V_{\alpha j} V^*_{\beta j}) \sin \frac{\Delta m^2_{ij} L}{2E_\nu},$$

В настоящий момент абсолютные значения масс неизвестны, как и не известна иерархия масс. Под иерархией масс понимается порядок их следования. С некоторой точностью известны разности квадратов масс Δm^2_{ij}. Исходя из этих данных, значения масс различных массовых нейтрино могут располагаться одним из способов, представленных на рис. 1. Эти способы расположения масс относительно друг друга называются нормальной (NH) и обратной иерархией (IH) соответственно.

Для изучения этих и других вопросов, связанных с нейтрино, по всему миру строятся детекторы нейтрино, на которых ставятся физические эксперименты. Среди них такие эксперименты JUNO [1] и Daya Bay [2]. Оба эксперимента находятся в Китае и имеют международные коллабо-

рации. ОИЯИ также принимает участие в этих экспериментах. В ЛЯП работает группа людей, которые являются частью коллабораций JUNO и Daya Bay [19]. Оба эксперимента направлены на детектирование потоков антинейтрино, порождаемых в ядерных реакторах электростанций. Ниже эти эксперименты описаны более подробно.

Ядерный реактор является мощным источником электронных антинейтрино. Они рождаются при бета-распаде продуктов деления изотопов урана и плутония в радиоактивном топливе. Электронные антинейтрино детектируются с помощью реакции обратного бета-распада, представленной формулой 1. В этой реакции антинейтрино взаимодействует с протоном, в результате чего рождается нейтрон и позитрон.

$$\bar{\nu}_e + p \rightarrow e^+ + n. \tag{1}$$

Daya Bay — реакторный нейтринный эксперимент, позволяющий изучать нейтринные осцилляции. Эксперимент состоит из четырех ближних детекторов, распределенных по двум залам, и четырех дальних детекторов в экспериментальном зале. Схема их расположения представлена на рис. 2. Ближние детекторы находятся в непосредственной близости от атомных электростанций Daya Bay и LingAo в Китае. Расстояние между этими двумя станциями 1100 м. Дальний детектор находится на удалении около 2 км от реакторов. Использование ближнего и дальнего детектора позволяет компенсировать неопределенности потока нейтрино. Расстояние до дальнего детектора выбрано не случайно. Именно на этой отметке находится первый минимум вероятности выживания электронных антинейтрино. Именно этот тип нейтрино рождается в реакциях ядерного реактора.

Детекторы находятся под землей в горной местности на глубине от 100 до 324 метров. Это соответствует 270 и 1200 метрам в водном эквиваленте. Такая прослойка из горных пород позволяет защитить детектор от космического излучения. Ближние и дальние детекторы соединены между собой подземными тоннелями.

Сами детекторы представляют собой несколько цилиндров, которые в свою очередь состоят из трех вложенных цилиндров. Внутренний, самый

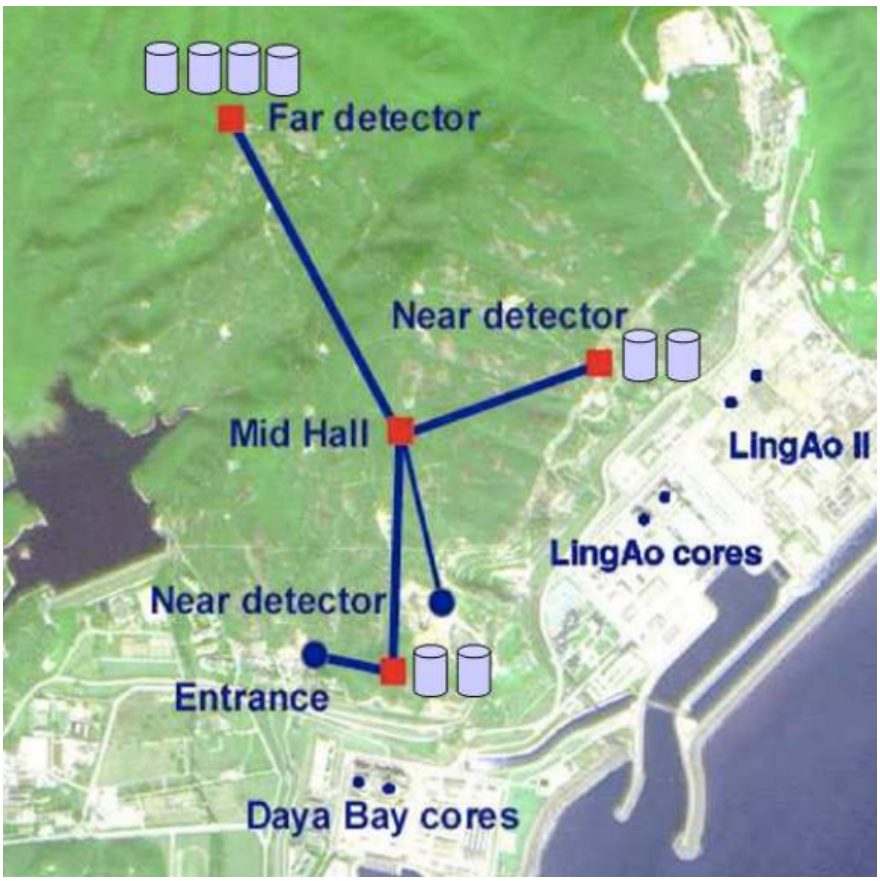

Рис. 2: Схема расположения эксперимента Daya Bay [3].

малый объем заполнен 20 тоннами жидкого сцинтиллятора с добавлением гадолиния. Он помещен в цилиндр большего объема. Объем между его стенками и стенками внутреннего цилиндра заполнен сцинтиллятором без добавления гадолиния. Наконец, третий, еще больший цилиндр содержит минеральное масло. Оно не обладает свойствами сцинтиллятора и выполняет функцию дополнительной защиты от внешнего фона. Во внешнем цилиндре установлены фотоумножители, позволяющие детектировать свечение сцинтиллятора.

В ближних детекторах используется по два таких цилиндра, а в дальнем — четыре. Эти цилиндры помещены в бассейн с водой. Прослойка воды составляет минимум 2.5 метра от стенок цилиндров. Вода служит дополнительным экраном от естественной радиации.

JUNO (Jiangmen Underground Neutrino Observatory) — это многоцелевой эксперимент, разработанный для определения иерархий масс нейтрино, точного измерения параметров осцилляций, изучения атмосферных, сол-

нечных и гео-нейтрино, а также для решения других нейтринных задач. Детектор эксперимента JUNO расположен на юге Китая, примерно в 53 км от строящихся атомных электростанций Янцзян и Тайшан. В 215 км от детектора находится также атомная электростанция Daya Bay, поток антинейтрино с которой будет являться фоном для эксперимента JUNO.

Детектор представляет акриловую сферу, заполненную 20 тысячами тонн жидкого сцинтиллятора. Нейтрино, проходя через этот сцинтиллятор, взаимодействуют с его частицами и приводят к его свечению. Лишь малая часть нейтрино возбуждает сцинтиллятор. Порожденные этим взаимодействием фотоны затем детектируются фотоумножителями, которые расположены вокруг акриловой сферы. Всего вокруг сферы расположено 18000 фотоумножителей диаметром 20 дюймов. Пространство между акриловой сферой и фотоумножителями заполнено водой. Это позволяет снизить детектируемые шумы, вызванные излучением, исходящих от горных пород, окружающих установку.

Детектор расположен под горой высотой 270 метров. Слой гранита обеспечивает экранирование детектора от космических мюонов. Мюоны являются основным источником детектируемого фона. Для лучшего подавления фона детектор был расположен глубоко под землей. Общая высота слоя пород, находящегося над детектором, составляет, таким образом, 700 метров.

Поток антинейтрино, порожденный на атомных электростанциях, проходит сквозь сферу детектора JUNO. С некоторой вероятностью нейтрино взаимодействует со сцинтиллятором, находящимся в ней, и результат этого взаимодействия регистрируется фотоумножителями. Собранную статистику необходимо обрабатывать и анализировать. Фреймворк GNA подходит для статистического анализа данных JUNO. В GNA уже реализована модель JUNO, позволяющая строить спектры потока антинейтрино.

1.2 GNA

Измерение физических параметров на основе экспериментальных данных в значительной степени основано на методах статистического анали-

за. Основным методом поиска значений параметров является метод поиска максимального правдоподобия. Для оценки доверительных интервалов применяется такие методы как профилирование функции правдоподобия, байесовский метод или метод Фельдмана-Казинса. Эти подходы включают в себя массивные вычисления при работе с моделями с большим количеством параметров, либо свободных, либо ограниченных.

Вычисления в GNA представлены в виде графа функций. Вычислительные графы, реализующие такие типичные задачи, как предсказание спектра антинейтрино, содержат сотни узлов. На один проход и вычисление такого графа затрачивается время порядка десятых долей секунды - секунд при использовании CPU. Однако вычисление такого графа — лишь часть реальной задачи. Результатом предсказания является гистограмма, зависящая от параметров. Это предсказание используется в алгоритме многомерной минимизации. Минимизация по 15 параметрам занимает около получаса на современных CPU, а полная минимизация — около шести часов. Статистический анализ требует многократной минимизации — сотни итераций для профилирования функции правдоподобия.

Методы оценки доверительных интервалов на основе Монте-Карло требуют еще большего времени выполнения. Например, метод Фельдмана-Казинса требует миллионы повторений процедур минимизации. Такие вычисления потребуют значительных затрат времени даже на многоядерных и многопроцессорных системах.

Global Neutrino Analysis — платформа для статистического анализа данных нейтринных экспериментов [7]. Она разрабатывается в Лаборатории ядерных проблем Объединенного института ядерных исследований (ЛЯП ОИЯИ). Фреймворк изначально разрабатывался для обработки данных экспериментов JUNO и Daya Bay. Тем не менее, архитектура платформы такова, что она может быть использована и для других экспериментов, в том числе, за пределами области нейтринной физики. Кроме того, в GNA представлены инструменты для реализации совместного анализа экспериментов [9, 10].

GNA использует принципы data flow [17]. Вычисление представляются в виде направленного графа, где каждый узел этого графа содержит

в себе функцию. В терминах GNA узлы графа называются трансформациями. Ребра графа показывают поток данных между трансформациями. Высокая гибкость в использовании платформы достигается наличием универсальных интерфейсов трансформаций и их инкапсуляцией.

Трансформации реализованы на языке C++. Функция трансформации имеет одну или несколько реализаций, к которым можно обращаться по именам. Трансформация имеет ноль, один или несколько входов и хотя бы один выход. Входы представляют собой ссылки на выходы трансформаций, с которыми они соединены ребрами. Выход является контейнером данных шаблонного типа, в который записывается возвращаемый результат, в общем случае представляющий собой массив. Кроме того, трансформация может зависеть от переменных.

Пользовательский интерфейс реализован на языке Python. Он позволяет выстраивать зависимости между трансформациями. Сам граф не компилируется. Компилируются только отдельные трансформации, реализованные на C++. Далее, на стороне Python они собираются в небольшие вычислительные цепочки или в вычислительный граф. Использование Python позволяет избежать дополнительных временных затрат при изменении графа и достигнуть высокой гибкости использования. Две этих части соединены посредством библиотеки PyROOT [11] — расширением фреймворка ROOT, которое производит автоматическое зеркалирование объектов C++ в Python .

Предполагается, что пользователь взаимодействует с фреймворком через оболочку Python. Тем не менее, рассматривается также использование фреймворка внешними разработчиками. Документация GNA подробно описывает, каким образом можно внедрить свои алгоритмы на C++ в среду GNA и работать с ними, как с частью платформы. Для этого необходимо реализовать алгоритм на C++ и обернуть его в трансформацию согласно документации, определив конструкторы и деструкторы класса трансформации. Далее необходимо указать имена входов и выходов трансформации, по которым можно будет обращаться со стороны Python, а затем сослаться на реализацию алгоритма и также дать ей имя. При добавлении реализации функции в трансформацию необходимо помнить, что функция по

умолчанию должна носить имя main. В случае, если трансформация содержит лишь одну функцию, то допускается имя не указывать – оно будет положено равным main автоматически. Однако если функций несколько, то вторая и последующие функции трансформации должна быть именованными в обязательном порядке.

Работа с графом происходит в два этапа. Первый этап включает в себя конфигурирование графа. Вычислительный граф описывает то, как взаимодействуют между собой отдельные трансформации и переменные. При создании графа выделяется память для выходных массивов трансформаций, определяются связи между трансформациями. Типы данных наследуются, то есть при присоединении входов одной трансформации к выходам другой размер и тип выхода присоединяемой трансформации определяется на основе размеров и типов входных массивов. Также проходит проверка составленного пользователем графа на корректность — удовлетворение всех требований к входным массивам трансформаций. Этот этап происходит единожды непосредственно перед вычислением. Второй этап — это сами вычисления. Запуск вычислительного графа может происходить многократно для различных входных значений.

Контейнер данных, соответствующий выходам трансформаций, реализован как класс `Data`. Он содержит в себе указатель, который по умолчанию равен `nullptr`. Значение, указывающее на ячейку памяти, в которой будут храниться данные, ему присваивается при выделении памяти на начальном этапе работы GNA. Выходные массивы трансформации доступны для записи только текущей трансформации. Экземпляры класса `Data` могут быть также использованы как вид на данные. В таком случае память специально для этого экземпляра не выделяется.

При выстраивании связей с другими трансформациями выходные данные текущей доступны только для чтения. Кроме указателя класс `Data` содержит в себе ряд иных представлений данных. Среди них представление в виде одномерного и двумерного массива, вектора и матрицы Eigen [12]. Все они являются лишь оболочками, которые ссылаются на один и тот же указатель и добавлены для удобства работы с данными. Например, библиотека Eigen предоставляет широкий функционал работы с матрицами,

набор реализованных матричных операций и функций над ними.

Графы вычисляются лениво. Это происходит только после чтения результата какой-либо трансформации. Притом, если необходим результат промежуточной трансформации, то вычисляться будет только тот подграф, от которого зависит эта трансформация. Таким образом, пользователь может использовать уже описанные модели и рассчитывать их частично, что не приведет к чрезмерным накладным расходам из-за наличия "лишних"трансформаций в используемой модели.

После того, как результат вычислен, он сохраняется в памяти, выделенной под выход трансформации. Трансформация содержит в себе флаг, который сигнализирует о том, актуальна ли информация, находящаяся в выходном массиве. Если значение флага говорит о том, что данные актуальны, то при повторном обращении к этой трансформации перевычисления происходить не будет. Вместо этого будет взят ранее посчитанный результат. Такой подход позволяет избежать лишних затрат на вычисления при многократном пересчете графа с неизменяемыми подграфами. В случае, если какой-либо результат перестает быть актуальным или изменяются значения переменных, то флаги всех трансформаций, которые от нее зависят, сбрасываются.

В Приложении 1 представлен граф модели JUNO. Этот граф содержит 123 узлов и 206 ребер. Результат вычислений этого графа — ожидаемый спектр антинейтрино для детектора в эксперименте JUNO. Этот спектр показан на рис. 3. Он представлен массивом из 280 значений, каждое из которых — число событий, соответствующее величине наблюдаемой энергии Evis. Спектр построен в предположениях нормальной (NH) и обратной (IH) иерархий. Также на графике представлена разность этих спектров. В модели использовалось 43 свободных параметра. Частичный граф модели Daya Bay может быть найден в Приложении 2. Он содержит 79 узлов и 94 ребра. Этот граф соответствует одному из восьми детекторов, одному из шести реакторов и одному из четырех изотопов. Полный граф содержит 2455 узел и 5672 ребра.

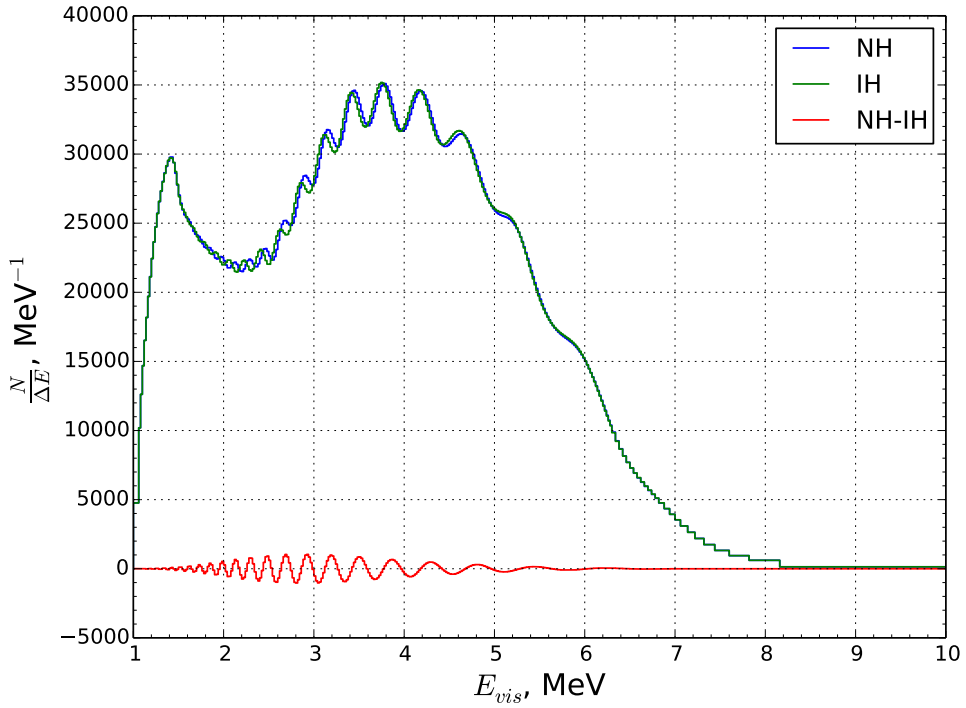

Рис. 3: Гистограмма, построенная с помощью GNA. Предсказание спектра антинейтрино для нормальной и обратной иерархий.

Глава 2. Вычисления на GPU

2.1 Особенности вычислений на GPU

Архитектура устройств GPU значительно отличается от центральных процессоров. Современные популярные CPU для ПК имеют 4-8 вычислительных ядер (до 16-и потоков). Самые новые процессоры имеют до 32-х ядер и 64-х потоков. Количество потоков у современных видеокарт в десятки и сотни раз больше. Однако, сравнивать эти типы устройств лишь по количеству ядер было бы некорректным. Ядро GPU выполняет элементарные задачи и не применяется для однопоточных задач. CPU ядра несравнимо более мощные.

Основная область применения GPU — это обработка изображений и рендеринг. Изображение представляет собой матрицу пикселей, к каждому из которых применяется набор функций. Набор операций, применяемый к каждому пикселю, одинаков, а их количество велико. Тем не менее, обработка изображений — это не единственная область применения графиче-

ских устройств. Любая задача, в которой необходимо произвести однотипные вычисления над элементами массивов данных подходит для выполнения на архитектуре GPU в том случае, если соблюдается ряд условий:

- В задаче отсутствуют или сведены к минимуму зависимости по данным. В случае, если результат вычислений одного элемента массива зависит от результата вычислений другого элемента того же массива, они не могут быть вычислены параллельно.
- Размер обрабатываемого массива достаточно велик, чтобы выигрыш от ускорения превышал временные затраты на издержки применения GPU (инициализация, копирование данных).
- В задаче сведена к минимуму необходимость взаимодействия с центральным процессором и оперативной памятью в ходе выполнения алгоритма. Это условие вытекает из предыдущего пункта. При обращении к CPU и RAM возникают временные издержки, которые влияют на конечное время работы программы.

2.2 CUDA

CUDA (Compute Unified Device Architecture) — архитектура для вычислений на графических процессорах, разработанная компанией NVIDIA. Технология позволяет реализовывать алгоритмы класса SIMT (single instruction multiple threads) и запускать их на GPU. Перечень графических устройств, на которых могут запускаться программы, написанные с использованием этого инструмента, представлен на сайте для CUDA-разработчиков [20]. В этот перечень входят все современные GPU от NVIDIA, в том числе, дискретные видеокарты для персональных компьютеров и ноутбуков.

Архитектура графических устройств выглядит, как представлено на рис. 4. GPU содержит набор потоков, распределенных по мультипроцессорам. Потоки объединены в группы по несколько потоков, одна такая группа потоков называется warp. На современных видеокартах размер одной такой группы — 32 потока. Warp является неделимым. То есть при запуске

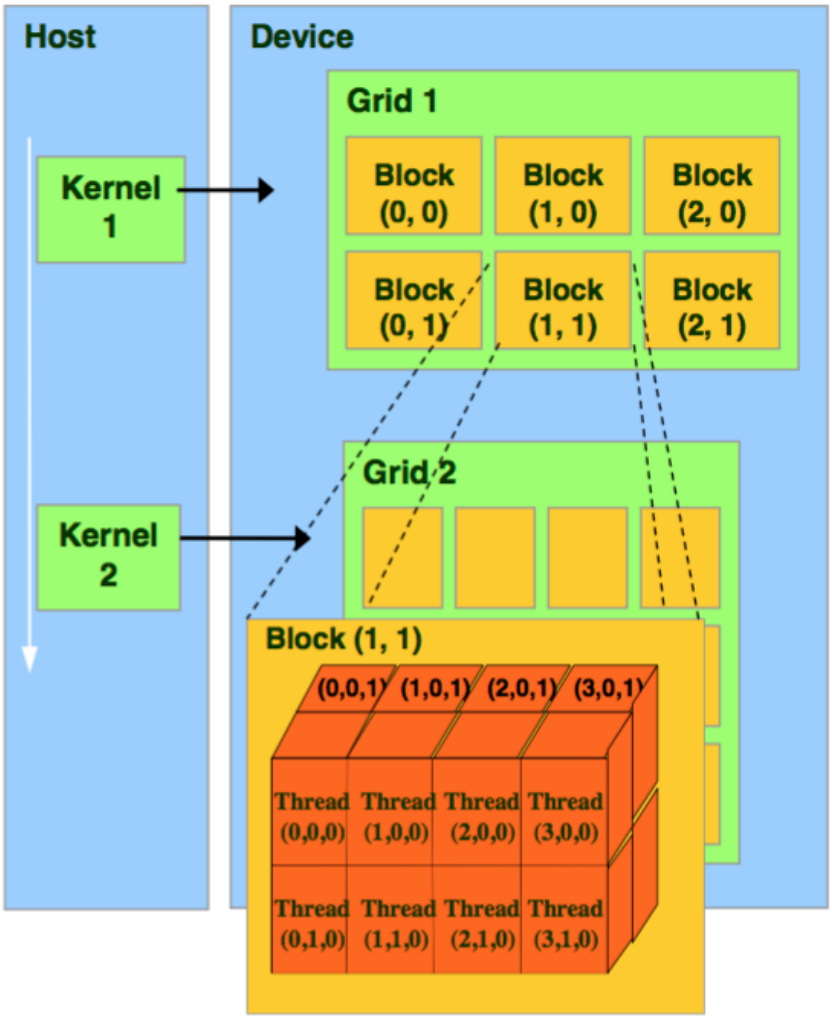

Рис. 4: Архитектура CUDA [?]

задачи на GPU выделяется количество рабочих потоков, кратное размеру warp (см. рис. 5). Логически потоки объединяются в блоки. Размер одного блока задается разработчиком при вызове GPU-функции. Блоки в свою очередь также объединяются в группы, называемые Grid, размер которых также задается разработчиком. Размеры блоков и Grid могут иметь до 3 измерений.

Как было сказано ранее, warp — это неделимая группа потоков. Несколько иллюстраций приведены на изображениях ниже. В случае, если алгоритм задействует всего 1 поток, то исполняется он 32-я потоками, даже если параметры запуска указывают на наличие одного блока из одного потока (см. рис. 6). Часть рабочих потоков будет простаивать в таком случае. Если размер блока указан некратным размеру warp, часть потоков также

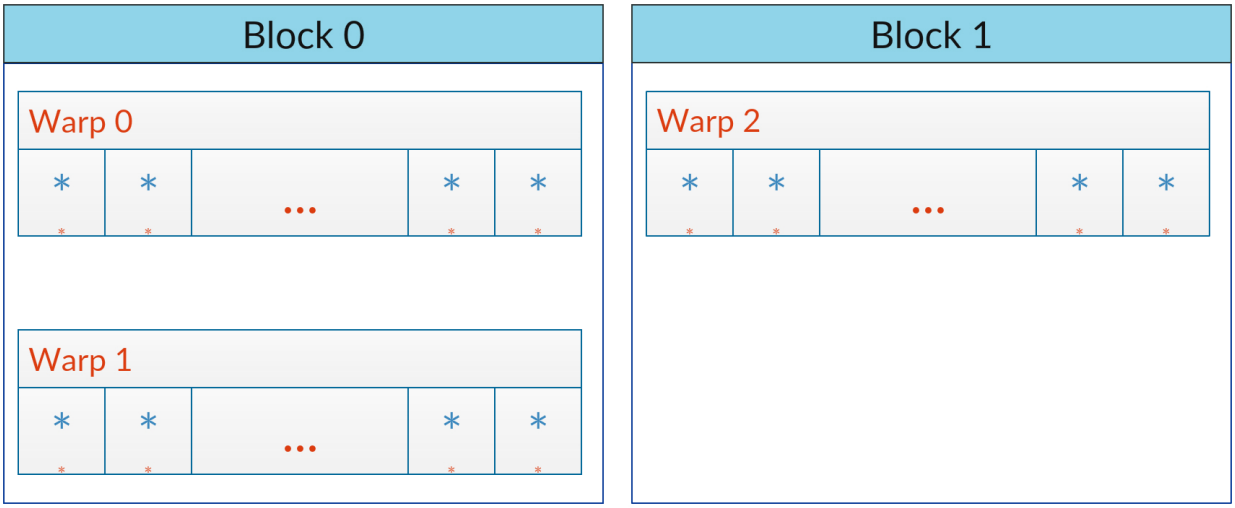

Рис. 5: Распределние потоков по группам warp и блокам. Размер блока — (32,2,1). Количество используемых рабочих потоков — 96.

будет простаивать, как показано на рис. 7. Кроме того, warp не может быть распределен между двумя мультипроцессорами, как это показано на втором примере (см. рис. 8). По этой причине рекомендуется устанавливать размер блока кратным размеру warp во избежание простаивания рабочих потоков.

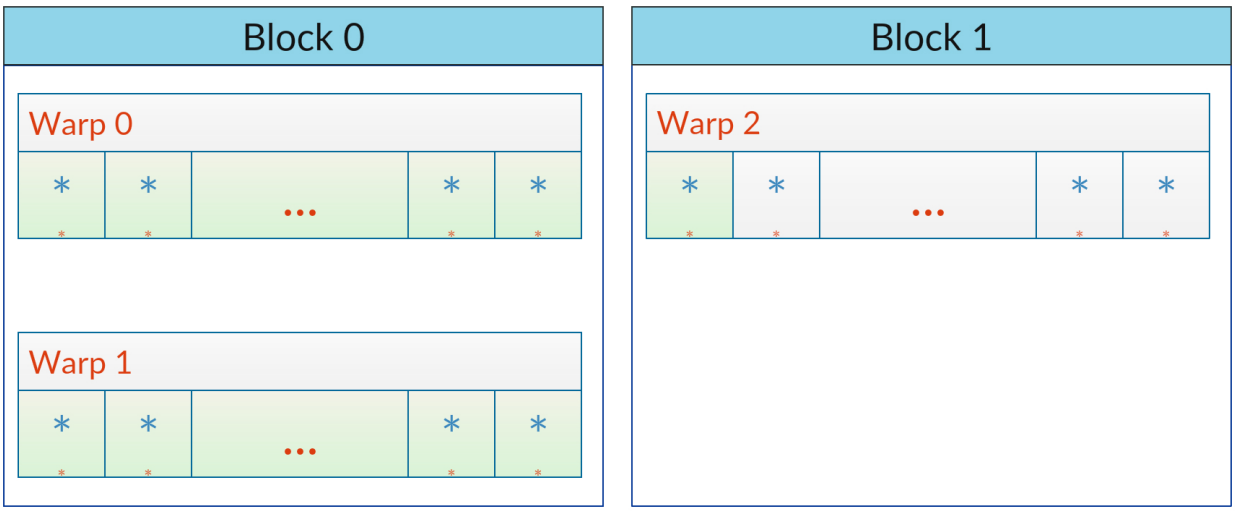

Рис. 6: Распределние потоков по группам warp и блокам. Размер блока (32,2,1). Количество фактически используемых рабочих потоков — 65. Количество зарезервированных потоков — 96.

Архитектура CUDA предполагает несколько типов памяти на устройстве. Иерархия памяти продемонстрирована на рис. 9. Три типа памяти могут быть доступны для любого потока внутри устройства: глобальная,

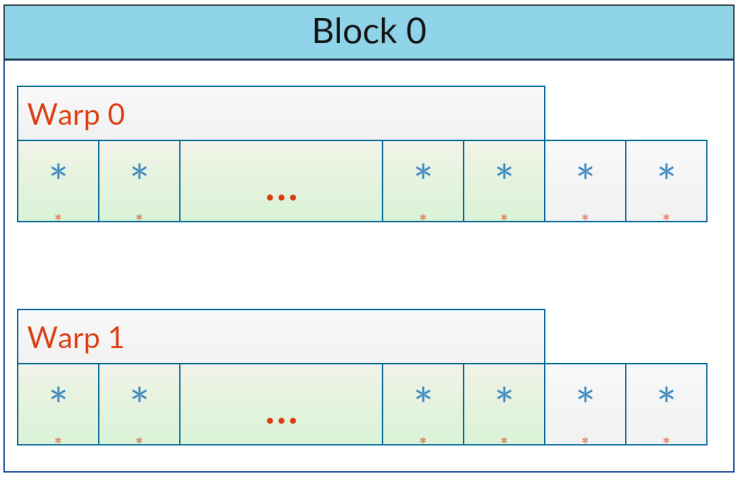

Рис. 7: Распределние потоков по группам warp и блокам. Размер блока — (34,2,1). Количество фактически используемых рабочих потоков — 64. Количество зарезервированных потоков — 68.

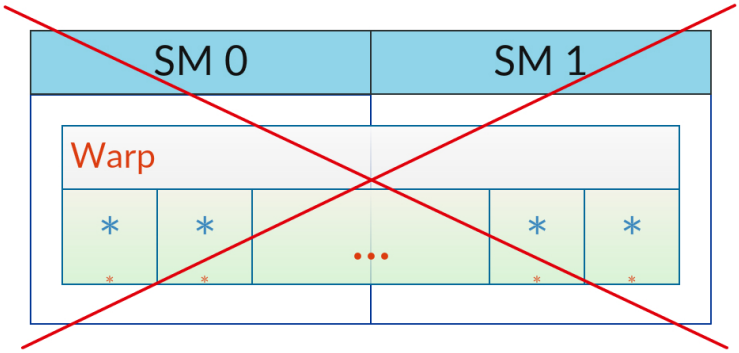

Рис. 8: Warp не может быть разделен между двумя мультипроцессорами (SM).

константная и текстурная память. Чтение и запись в глобальную память может происходить из любого потока. Однако взаимодействие с ней — самое медленное. Константная память доступна только для чтения любым потоком. Заполнение данными происходит до вызова GPU-функции. Она имеет малый размер, но в некоторых случаях имеет большую скорость доступа по сравнению с глобальной памятью. Текстурная память также доступна только для чтения из GPU-функций и оптимизирована для операций над двумерными массивами.

Для потоков в пределах одного блока для чтения и записи доступна разделяемая память. Потоки могут обращаться только к разделяемой памяти, который соответствует текущему блоку. Доступ к такому типу памяти значительно более быстрый, чем к глобальной.

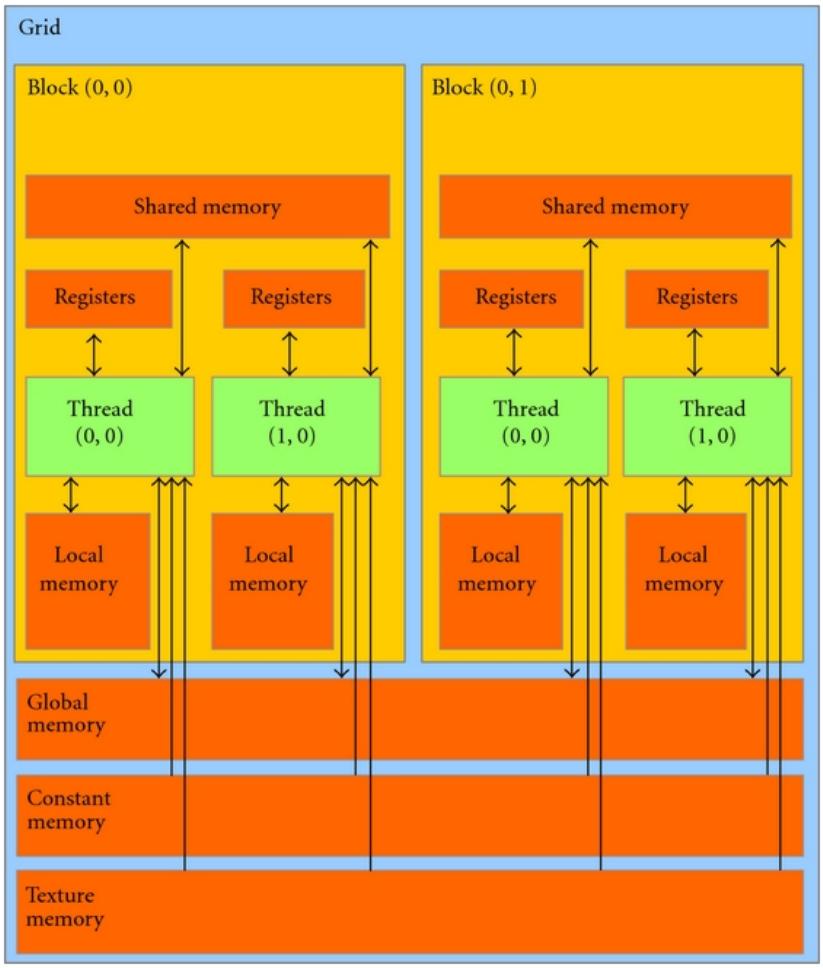

Рис. 9: Иерархия памяти CUDA.

Существует два типа памяти, доступной эксклюзивно для потока. Первый из них — локальная память потока. Она представляет собой фрагмент глобальной памяти, однако операции над данными в этом участке данных могут производиться только одним потоком. Такое владение участком память позволяет ускорить доступ по сравнению с временем доступа к глобальной памяти. Второй тип — регистры. Они являются памятью с самым быстром доступом к данным. Однако управление регистрами также является наиболее сложным с точки зрения разработки.

С точки зрения разработки важным отличием большинства графических процессоров от центральных является различия в производительности при вычислениях с числами двойной точности по сравнению с вычислениями с числами одинарной точности. При вычислениях на современных CPU обработка массивов чисел с плавающей точкой одинарной точности проис-

ходит примерно в два раза быстрее, чем в случае чисел с двойной точностью. При работе с GPU выигрыш производительности за счет снижения точности вычислений может составлять несколько десятков раз. Значение отношения производительности вычислений с разной точностью зависит от конкретной архитектуры, а точнее того, предусмотрены ли в конкретной архитектуре DPU (double precision unit). Каждое ядро CUDA имеет ALU (arithmetic logic unit), которые предназначены для целочисленных операций, и FPU (floating precision unit), предназначенные для операций над числами с плавающей точкой одинарной точности. В некоторых GPU есть также DPU, позволяющие работать и с двойной точностью. Обработка чисел двойной точности возможна и в устройствах без DPU, однако в этом случае для обработки одного такого числа требуется несколько циклов работы FPU, что приводит к замедлению вычислений. Согласно приведенным в книге [15] данным, пиковая производительность GTX 780, не имеющего DPU, в 24 раза выше при использовании одинарной точности. Производительность GTX Titan Z, в архитектуре которого DPU есть, отличается всего в 3 раза.

Еще одна архитектурная особенность CUDA — наличие SFU (special function unit). Они предназначены для подсчета специальных функций (синус, косинус, квадратный корень числа). Эти модули поддерживают одинарную точность и позволяют затрачивать меньшее время на выполнение трансцендентных функций.

Глава 3. Поддержка вычислений на GPU в GNA

3.1 Мотивация для использования GPU

Вычисления в GNA имеют некоторые особенности, позволяющие предположить целесообразность использования для них графических процессоров. В задачах, решаемых с помощью GNA значительная часть функций применяются к массивам данных, в которых нет зависимости между различными элементами массивов. Кроме того, в таких задачах как минимизация необходимо вычисление одних и тех же функций множество раз

при различных значениях параметров.

Ряд модулей (трансформаций) в GNA содержит в себе трансцендентные функции. Так, например, формула для вычисления вероятности осцилляций нейтрино содержит в себе синус. Кроме того, большая часть вычислений в GNA содержит ограниченный набор арифметических и тригонометрических функций. То, что архитектура CUDA оптимизирована для исполнения подобных функций, также позволяет предположить, что использование GPU в GNA является целесообразным.

Перед внесением изменений непосредственно в архитектуру платформы были реализованы тестовые примеры, не интегрированные в GNA. Было произведено сравнение реализованной с помощью CUDA версии алгоритма вычисления вероятности осцилляций с версией для CPU, имеющейся в GNA. Время, затраченное на этот алгоритм на GPU без учета затрат на копирование данных оказалось примерно в 20 раз меньше, чем на центральном процессоре. Тестирование производилось на ноутбуке, укомплектованном видеокартой GTX 970M и центральным процессором Intel Core i7, используемый тип данных — числа двойной точности с плавающей точкой. Более подробно результаты описаны в работе [16]. Этот тест подтвердил предположение о целесообразности применения GPU в алгоритмах GNA.

Тем не менее, при добавлении в расчет времени выполнения затрат на обмен данными, тесты показывают небольшое замедление в сравнение с версией для центрального процессора. В связи с этим был введен ряд новых для GNA архитектурных решений, связанных с добавлением поддержки вычислений на GPU. Важно отметить, что при реализации библиотеки поддержки GPU необходимо сохранить основные преимущества платформы, позволяющие работать эффективно на центральном процессоре. Это повлекло ряд компромиссов, описанных в следующих разделах. Необходимость сохранения основных свойств работы с CPU вызвана тем, что предполагается применение GNA на различных вычислительных устройствах, в том числе, на машинах без GPU. Использование видеокарты при вычислениях является опциональным.

3.2 Архитектурные решения

Поддержка вычислений на видеокартах в GNA реализована с помощью CUDA. Для ее использования необходима вычислительная машина с CUDA-совместимой видеокартой, а также установленным пакетом CUDA Toolkit версии 7.5 и выше. Перечень всех видеокарт, которые могут быть использованы для вычислений, можно найти в [20].

CUDA-ориентированная часть фреймворка реализована в виде библиотеки cuGNA, компилируемой условно. Кроме того, для компиляции CUDA-ориентированного кода необходим компилятор nvcc, в то время как основная часть GNA компилируется с помощью gcc/clang. Реализация поддержки вычислений на графических процессорах в виде отдельной библиотеки позволяет избежать перехода к компилятору nvcc для всей платформы GNA. Также при компиляции без включения cuGNA наличие установленного компилятора nvcc не требуется.

В случае, если пользователь не собирается или не имеет возможности использовать графический процессор, он может компилировать пакет GNA без поддержки GPU. Эта конфигурация предоставляется по умолчанию. Для компиляции с поддержкой GPU должен быть установлен соответствующий флаг компиляции. При истинном значении этого флага в основную сборку платформы GNA включаются участки кода, обращающиеся к функциям библиотеки cuGNA. В противном случае при отсутствии этого условия компиляции и не скомпилированной библиотеке cuGNA, наличие кода, взаимодействующего с GPU, повлекло бы возникновение ошибок компиляции пакета GNA.

Кроме флага, сигнализирующего о наличии или отсутствии поддержки вычислений на видеокарте, при компиляции может быть установлено значение, показывающее, насколько подробный вывод в консоль требуется. Для этого используется флаг `CUDA_DEBUG_INFO` и его значение может быть от 0 до 3. Ноль означает, что никакой отладочной информации выводиться не будет. При установке значения 1 будет выведена информация только об ошибках исполнения CUDA-функций, в том числе, копирования и выделения памяти. При установке значения 2 дополнительно выводится

информация о предупрежденях при синхронизациях, в том числе, в случае успешного их завершения. Такое предупреждение может возникнуть, например, в случае повторного копирования данных в память GPU при условии, что там уже находятся актуальные данные. В случае установки значения 3 на консоль будет выведена информация обо всех синхронизациях данных, даже тогда, когда они были завершенным корректно и без предупреждений, а также инициализации массивов в памяти GPU.

Наличие такой иерархии вывода информации на консоль мотивирована несколькими факторами. Первый фактор – удобство редактирования своего кода разработчиками, в том числе сторонними. Второй фактор заключается в том, что при возникновении не критических ошибок на стороне GPU-устройства, возвращаемое значение будет считаться формально корректным, однако его содержание будет неверным и негативно скажется на конечном результате вычислений. Во избежание подобных неточностей, все возвращаемые значения GPU-функций проверяются, а при максимально возможном значении отладочного флага, все обмены данными журналируются.

Библиотека cuGNA представлена в виде cmake проекта, включаемого в GNA. Он содержит в себе несколько частей:

- Ядро, содержащее логику обмена данными, вспомогательные структуры данных, конфигурационные данные и обертки для функций, через которые происходит общение библиотеки с основной частью GNA.

- Основные математические операции, такие как сложение и перемножение матриц, а также иные элементарные поэлементные функции.

- Более сложные алгоритмы, имеющиеся в GNA, такие как вычисление вероятности осцилляций, интерполяция, статистические методы.

Основной сущностью библиотеки cuGNA является класс GpuArray. Он представляет собой оболочку над массивом данных, обеспечивающую автоматическую синхронизацию между памятью GPU и RAM, а также инструменты взаимодействия с CPU-ориентированной частью GNA.

GpuArray содержит в себе поля указателя на память в RAM и указателя на участок памяти видеокарты. Вся синхронизация происходит между этими двумя участками памяти.

Сразу заметим, что мы намеренно отказались от использования CUDA unified memory, поскольку использование этой абстракции влечет потерю контроля над передачей данных. Также функционал unified memory не позволяет достичь необходимого уровня взаимодействия с ядром GNA.

При включении флага поддержки GPU вычислений в классе Data фреймворка GNA становится доступным поле, содержащее указатель на GpuArray. При инициализации графа происходит выделение памяти на GPU, а полю указателя на память RAM присваивается то же значение, что записано в соответствующее поле указателя в экземпляре класса Data. Таким образом, GpuArray представляет собой дополнительное представление массива данных, хранящихся в выходе трансформации.

GpuArray содержит флаг синхронизации *syncFlag*. Его значение может быть одним из трех: *Synchronized, Unsynchronized* или *SyncFailed*. Значение *Synchronized* присваивается этому флагу тогда, когда и в памяти GPU, и в RAM данные актуальные. Такое значение присваивается непосредственно после синхронизации и сбрасывается при пересчете функции либо на стороне GPU, либо на стороне CPU. Значение *SyncFailed* устанавливается тогда, когда синхронизация данных завершилась с ошибкой. Значение Unsynchronized является значением по умолчанию. Оно означает, что данные, хранящиеся по указателям, соответствующим GPU и CPU, различны.

Этот флаг используется совместно с другим флагом *dataLoc* типа `DataLocation`, который указывает, где находятся актуальные данные. Он может принимать следующие значения:

- *NotInitialized* — значение по умолчанию. Означает, что экземпляр класса `GpuArray` создан, но память на еще GPU не выделена.
- *InitializedOnly* означает, что память в GPU выделена, однако данные

туда еще не были скопированы. Значение указателю на участок памяти в RAM не присвоено.

- *Host* означает, что актуальные данные находятся в RAM, и к ним можно обращаться по соответствующему указателю.
- *Device* означает, что актуальные данные находятся в памяти GPU.
- *NoData* устанавливается тогда, когда в качестве указателя на память в RAM был принят нулевой указатель.
- *Crashed* означает, что выделение памяти или синхронизация данных были завершены с ошибкой.

В случае, если значение *syncFlag* равно *Synchronized*, то при значении флага dataLoc равным как *Host*, так и *Device* будет предполагаться, что актуальные данные находятся и в памяти GPU, и в RAM. Проверка обоих флагов добавлена во избежание дополнительного обмена данных в случаях, когда эти данные одинаковы.

В `GpuArray` есть несколько функций синхронизаций. Две функции синхронизации *syncH2D* и *syncD2H* используются исключительно в библиотеке cuGNA. Они не принимают никаких аргументов, направление синхронизации в них строго определено. В реализацию этих методов включена проверка флагов синхронизации. В случае, если данные уже синхронизированы на момент вызова одного из этих методов, то никаких дополнительных операций не происходит и считается, что синхронизация завершена корректно. Если же данные не синхронизированы, то происходит копирование данных, а по его завершении флаг *syncFlag* устанавливается в значение *Synchronized*.

Для работы со стороны основного кода GNA используется метод *sync*, которая принимает в себя аргумент типа DataLocation. В зависимости от его значения используется один из описанных выше методов синхронизации. Если принимаемое в качестве аргумента значение не равняется ни *Host*, ни *Device*, то выводится соответствующее сообщение об ошибке.

Реализован набор конструкторов класса `GpuArray`. Он может быть создан как без инициализации содержимого, так и с ней. В случае, если данные значения указателей не инициализированы при конструировании объекта, они могут быть определены позднее с помощью метода *Init*. Именно такой способ создания объектов используется при инициализации `GpuArray` как поля класса `Data`.

Метод *Init*, как и другие методы, задающие значения массивов в `GpuArray`, возвращают значение типа `DataLocation`. Это необходимо для того, чтобы в случае возникновения ошибок возвращаемое значение о них сигнализировало. Задание значения внутренних указателей по значениям, принимаемым в качестве аргументов сопряжено с копированием данных между памятью GPU и RAM или вызовом GPU-функции. Возвращаемое значение соответствует коду, с которым завершилось обращение к GPU-ориентированным функциям.

Со стороны ядра GNA работа с GpuArray происходит при взаимодействии с контейнером данных для выходов трансформации. Каждому выходу соответствует ссылка на участок памяти GPU. Инициализируется эта память при построении графа в случае, если соответствующий флаг поддержки GPU установлен для трансформации. Он включается либо при явном указании имени функции в трансформации, либо при использовании конструкции `"with cuda:"` в Python скрипте перед блоком описания GPU-ориентированных трансформаций. Во втором случае будут использоваться функции, имя которых *gpu*. Это считается именем по умолчанию для таких функций. Рекомендуется именно так называть первую GPU-ориентированную функцию в трансформации при пользовательском ее определении.

При переключении на функцию GPU устанавливается флаг, который включает участки кода инициализации памяти, синхронизации и другие части GNA, использующие библиотеку cuGNA. Теперь при обращении к контейнеру данных для выхода трансформации будет происходить обращение к памяти видеокарты. При вызове пользователем команды вывода содержимого такого контейнера на экран будет происходить сначала копирование данных, а затем вывод скопированного содержимого.

Над `GpuArray` определен ряд операторов, таких как приравнивание, поэлементное перемножение, сложение и разность, умножение на число и отрицание. Каждый оператор содержит в своей реализации вызов GPU-функции. Поэтому при написании трансформаций не рекомендуется использовать эти операторы в случае, если функция трансформации содержит какие-либо операции, кроме тех, что описаны в операторах. Это приведет к дополнительным накладным расходам на вызов функции, которых можно избежать, использовав всего одно обращение к GPU с помощью GPU-функции, которая выполняет все операции без передачи выполнения CPU.

При использовании GPU необходимо свести к минимуму обмен данными между оперативной памятью и графическим устройством, так как копирование данных является затратной операцией. Копирование данных в пределах памяти GPU также приводит к некоторым накладным расходам, однако значительно меньшим, чем в случае обмена данными с оперативной памятью [14].

Цель GNA состоит в гибкости использования и его применимости к широкому кругу задач. Это достигается мелкозернистостью задач, отведенных отдельным трансформациям. При внедрении поддержки GPU нам хотелось сохранить эти свойства GNA. В связи с этим, был выбран компромиссный подход к синхронизации данных во время выполнения задач с использованием GPU. Схематично взаимодействие трансформаций в случае поддержки CUDA приведено на рис. 10.

Предположим, вычислительная цепочка выстроена таким образом, что часть трансформаций исполняются на GPU. Для каждой трансформации выделена память для выходного массива как в RAM, так и в памяти GPU. Когда происходит вызов результата трансформации T_n, поочередно запрашиваются результаты вычислений трансформаций, находящихся выше, а затем запускается их последовательное вычисление, начиная с T_0. После вычисления ее результата происходит копирование данных на устройство внутри контейнера данных, соответствующего T_0. После копирования устанавливаются флаги синхронизации, сообщающие о том, что актуальные данные находятся в обоих участках памяти. При вычислении

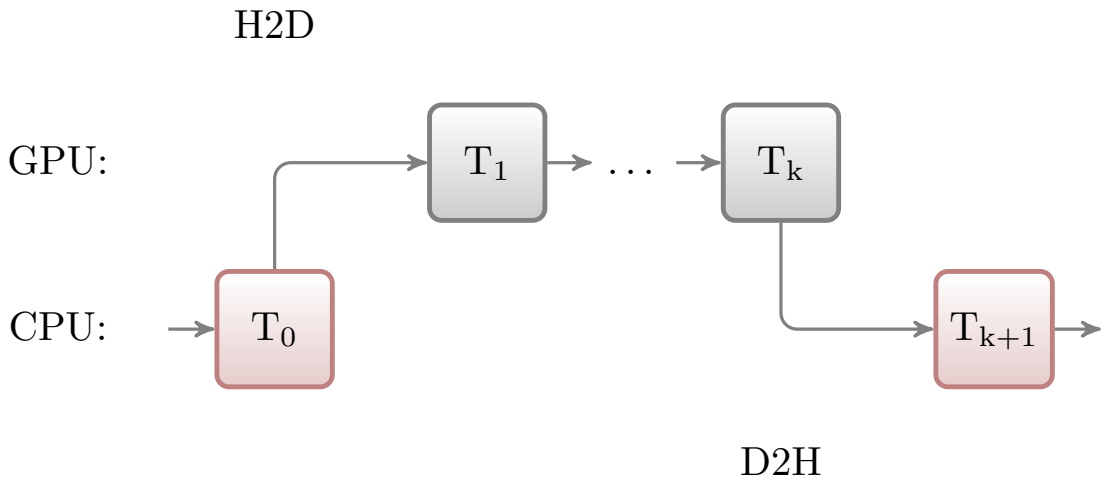

Рис. 10: Схема взаимодействия GPU и CPU трансформаций. D2H — копирование данных из памяти GPU в RAM, H2D — копирование обратно.

функции в трансформации T_1 берется результат, находящийся в памяти GPU. При вычислении дальнейших CUDA-ориентированных трансформаций, идущих подряд, программа также обращается к памяти видеокарты при считывании результата работы предыдущих трансформаций, несмотря на то, что непосредственно запуск функций трансформаций происходит со стороны ядра GNA, работающего исключительно на центральном процессоре. Копирование данных обратно происходит только при переходе к CPU-ориентированной трансформации.

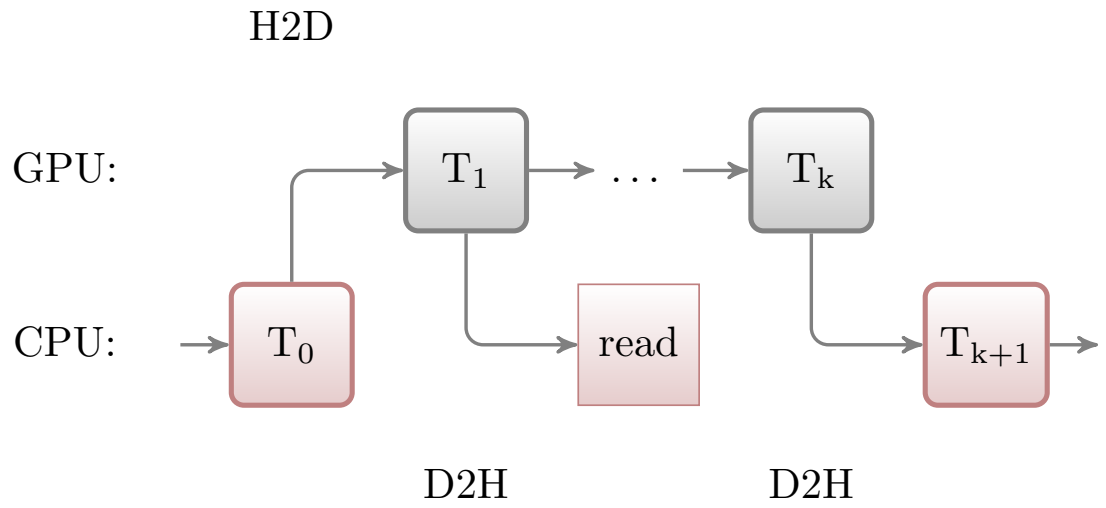

Рис. 11: Схема взаимодействия GPU и CPU трансформаций при промежуточном чтении данных. D2H — копирование данных из памяти GPU в RAM, H2D — копирование обратно.

Еще один случай, когда может происходить обмен данными, показан на рис. 11. В случае, если пользователь хочет вывести на экран результат промежуточных вычислений, он может воспользоваться соответствующей dump-функцией, спровоцировав дополнительное копирование выходных массивов. Такие функции предназначены, в основном, для отладки модели. Копирование в таком случае будет произведено лишь в одном направлении. Для дальнейшего вычисления будут использованы данные, уже находящиеся в памяти графического устройства после вычислений, и дополнительное копирование в обратном направлении не потребуется. Так как dump-функции только читают данные, а не изменяют их, флаги синхронизации будут указывать на то, что актуальные данные находятся на обоих устройствах.

Отметим, что наиболее выгодным будет составление вычислительного графа таким образом, чтобы избегать чередования трансформаций с различными целевыми устройствами. В случае построения цепочки трансформаций образом, показанным на рис. 12, обмен данными будет происходить до и после каждой GPU-ориентированной трансформации. Такое построение вычислительной цепочки приведет к значительным затратам на копирование данных. В случае, если отдельные трансформации просты с вычислительной точки зрения и требуют на выполнение на центральном процессоре количество времени меньшее, чем обмен данными, то скорость работы GNA лишь снизится при включении поддержки GPU. К сожалению, метод построения моделей пользователем не может быть предсказан разработчиками. Однако для предостережения пользователей от неэффективного использования cuGNA, мы предоставляем набор тестов трансформаций на производительность для некоторых GPU. Это поможет пользователю составить представление о целесообразности использования cuGNA в его модели.

Еще одно решение, введенное в GNA — переменная точность вычислений. Ранее в GNA предполагалось использование двойной точности вычислений. Была добавлена поддержка одинарной точности и возможность задания ее через пользовательский интерфейс. В случае, если пользователь наиболее заинтересован в производительности вычислений, нежели в точ-

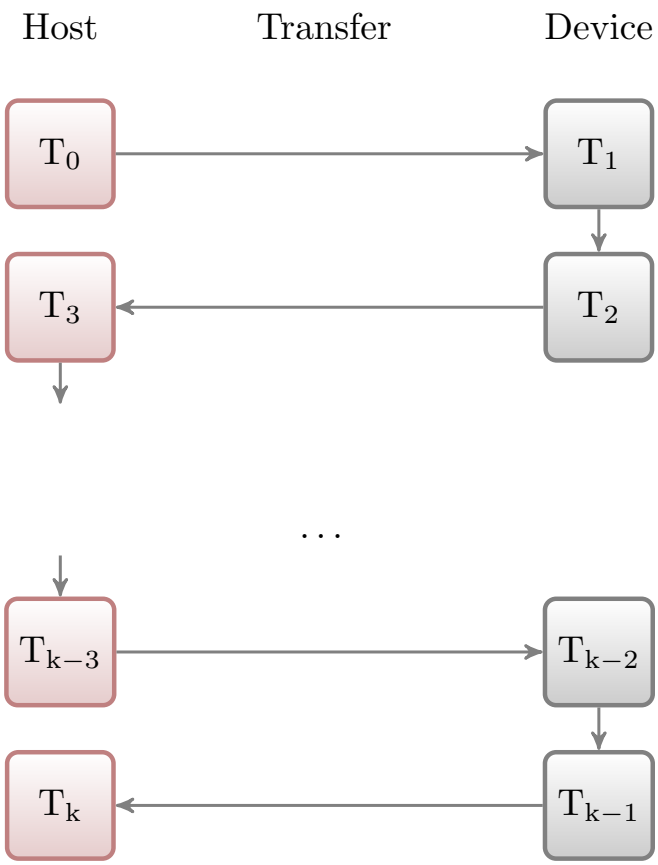

Рис. 12: Чередование вызовов трансформаций Для корректной работы необходимо $N/2$ операций обмена данными при четном N и $N/2+1$ — при нечетном. N — количество трансформаций.

ности, или же если для его алгоритма не требуется двойная точность, то он может ее переключить со стороны Python. При этом в зависимости от архитектуры конкретной GPU пользователь может получить значительный выигрыш в производительности.

Глава 4. Примеры и тесты

В этом разделе будет описано несколько групп основных, часто используемых трансформаций, перенесенных на GPU. Первая группа трансформаций состоит из базовых алгоритмов. Среди них сумма, взвешенная сумма массивов, поэлементное перемножение массивов. Эти трансформации не являются вычислительно сложными. Хоть алгоритм поэлементного сложения или перемножения и не имеет зависимостей по данным, использовать GPU для одного лишь сложения векторов нецелесообразно.

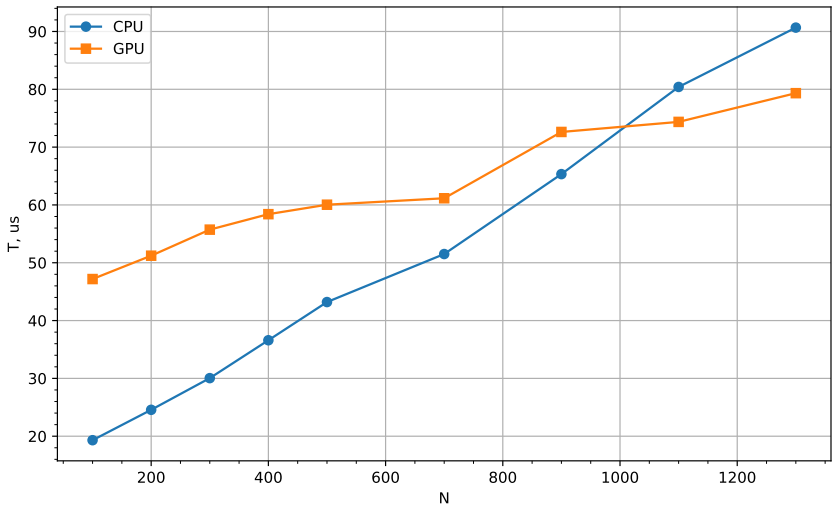

Рис. 13: Тесты для функции `Exp`

Более сложные поэлементные векторные трансформации — `Exp` и `SelfPower`. Первая из них в качестве результата возвращает массив данных, который состоит из значений e^{x_i}, где x_i — элементы входного массива. `SelfPower` возвращает массив чисел, каждый элемент которого считается по формуле

$$f_i = \left(\frac{x_i}{a}\right)^{\frac{x_i}{a}}.$$

Замеры времени, необходимого для вычисления функции `Exp` с учетом затрат на копирование данных, представленные на рис. 13 графики показывают, что выигрыш во времени от использования этой функции на GPU получен при значениях выше 1000 элементов. Стоит однако заметить, что в обоих случаях затрачиваемое время всего лишь порядка десятков микросекунд.

Хоть эта функция и является вычислительно более сложной, чем описанные ранее базовые трансформации, затраты непосредственно на вызов GPU функции и копирование данных значительно выше, чем на подсчет одного или нескольких таких значений. Тесты показали, что накладные расходы на вызов CUDA-функции на тестируемом устройстве составляют порядка 15 мкс.

Несмотря на то, что использование этих трансформаций с примене-

нием GPU не является целесообразным, они необходимы для GNA. Фреймворк построен модульным принципом. В состав фреймворка входят такие простые трансформации как сложение и поэлементное перемножение, а также другие элементарные трансформации. Эти модули используются как промежуточные звенья. Их перенос на GPU позволяет избегать лишних накладных расходов в вычислительных цепочках, где такие трансформации являются промежуточным звеном.

Следующая группа трансформаций — простые трансформации для матриц и гистограмм. Среди них транспонирование матрицы, а также трансформации `Normalize` и `RenormalizeDg`. Normalize преобразует гистограммы в соответствии с формулой

$$B_{j,\dots} = \frac{A_{j,\dots}}{\sum_{i,\dots} A_{i,\dots}}.$$

Трансформация `RenormalizeDiag` преобразует матрицу C в соответствии с формулой

$$E_{ij} = \frac{D_{ij}}{\sum_k D_{kj}},$$

где D_{ij} в зависимости от режима ее использования считается по формуле 2 или 3, где s — задаваемый пользователем множитель.

$$D_{ij} = \begin{cases} sC_{ij}, & |i-j| < n \\ C_{ij}, & \text{в другом случае} \end{cases}, \tag{2}$$

$$D_{ij} = \begin{cases} C_{ij}, & |i-j| < n \\ sC_{ij}, & \text{в другом случае} \end{cases}. \tag{3}$$

Статистические трансформации `Poisson` и `Chi2`. Эти трансформации позволяют рассчитать вероятность наступления событий для соответствующих событий. Эти функции вычислительно более сложные, чем описанные ранее в этом разделе.

Для подсчета каждого элемента выходного массива трансформации `Poisson` используется формула

$$\texttt{log}L(x|\mu) = \texttt{log}\prod_{i=1}^{n}\frac{\mu_i^{x_i}e^{-\mu_i}}{x_i!}$$
$$= \sum_{i=1}^{n}(x_i\texttt{log}\mu_i - \mu_i - \texttt{log}(x_i!)).$$

Вероятность осцилляций — одна из ключевых трансформаций в анализе JUNO. У этой трансформации есть две версии. В первом версии трансформация содержит полностью все вычисления в одной функции. Во втором случае она разбита на несколько компонент, которые затем суммируются. Для этого есть три трансформации, каждая из которых считает один из трех компонентов, соответствующий значению $\Delta m^2_{12}, \Delta m^2_{13}$ или Δm^2_{23} в формуле

$$P(\nu_\alpha \to \nu_\beta) = \delta_{\alpha\beta} - 4\sum_{i>j}\mathrm{Re}(V^*_{\alpha i}V_{\beta i}V_{\alpha j}V^*_{\beta j})\sin^2\frac{\Delta m^2_{ij}L}{4E_\nu}$$
$$+2\sum_{i>j}\mathrm{Im}(V^*_{\alpha i}V_{\beta i}V_{\alpha j}V^*_{\beta j})\sin\frac{\Delta m^2_{ij}L}{2E_\nu},$$
$$i = 1..3, j = 1..3.$$

Затем с помощью взвешенной суммы может быть получено значения вероятностей осцилляций для массива значений энергии. В анализе требуется многократное вычисление вероятности осцилляций. Второй подход позволяет избежать в итеративном процессе перевычисления компонент, которые не изменились по сравнению с теми, что использовались на предыдущей итерации.

На рис. 14 представлено сравнение затрачиваемого времени на выполнение алгоритма для GPU и CPU версий. Этот график отображает время, затрачиваемое на вероятность осцилляций, посчитанную как сумму трех

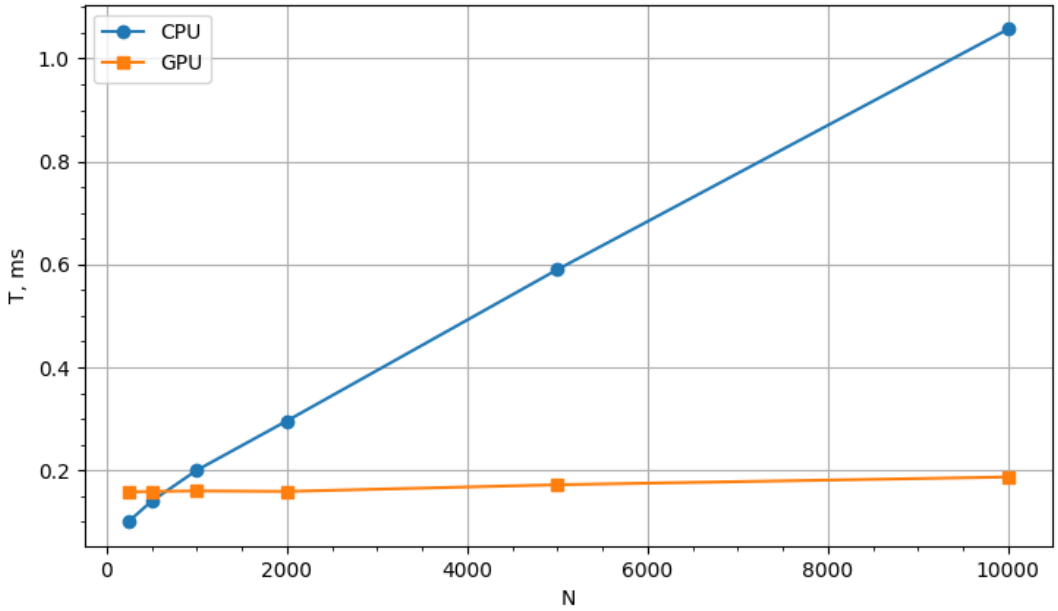

Рис. 14: Временные затраты на выполнение алгоритма вероятности осцилляций для версий на CPU и GPU. N — размер входного массива.

компонентов как отдельных компонентов в соответствии со схемой, представленной на рис. 15 Показанное время работы тестов включает обмен данными между трансформациями.

Результат работы алгоритма, представленного этим графом, показан на рис. 16. Это вероятность выживания нейтрино на расстоянии 52км при энергиях от 1 до 10 МэВ.

Экспоненциальная интерполяция представлена трансформацией `InterpExpo`. На вход она принимает веткора x, y — известные значения функции, а также вектор x' — значения, в которых требуется найти значение функции $y = f(x')$. Считаются эти значения по формуле

$$f(x'_i) = y_i e^{(-x_i - x_j)b_j},$$

$$b_j = -\frac{\ln(y_{j+1}/y_j)}{x_{j+1} - x_j}.$$

Тесты проводились на следующих устройствах:

- GPU: NVIDIA GeForce GTX 970M,

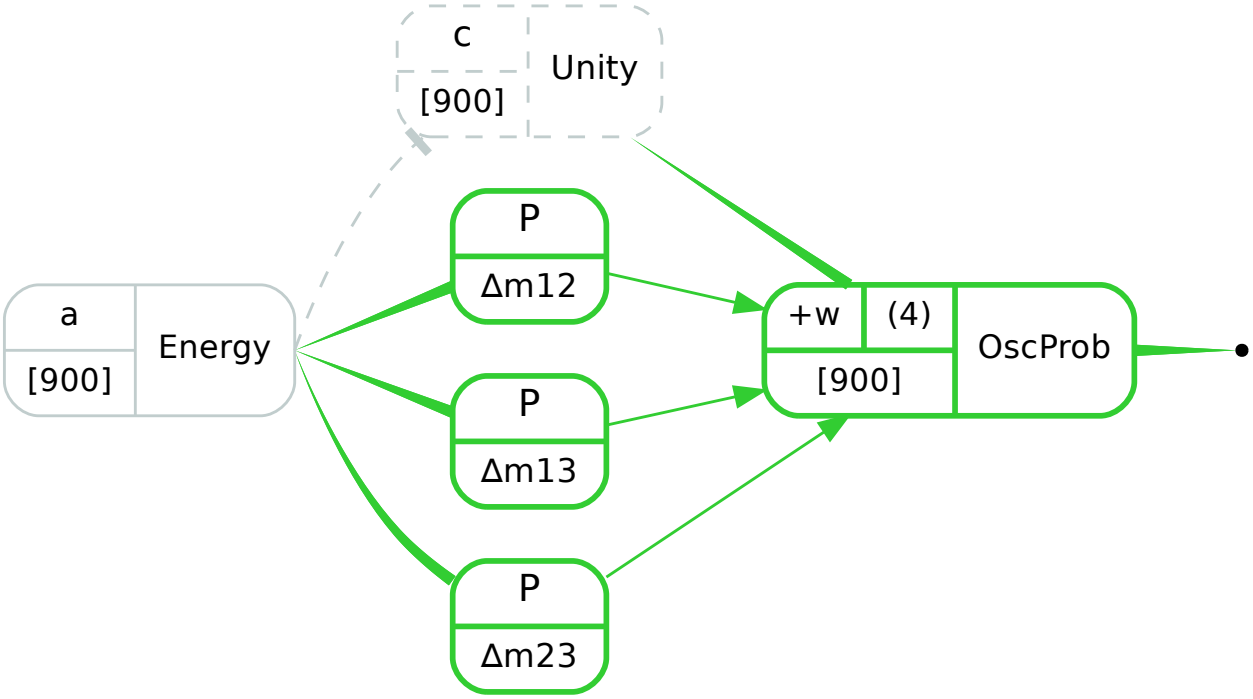

Рис. 15: Вычислительный граф для покомпонентной версии вероятности осцилляций. Зеленым цветом отмечены узлы, вычисляющиеся на GPU.

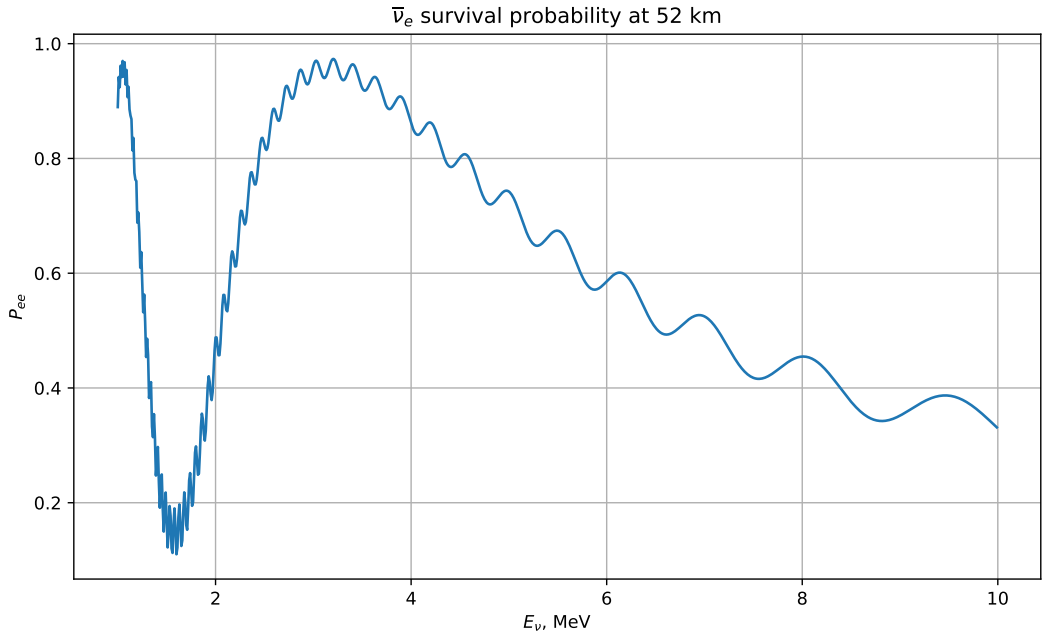

Рис. 16: Вероятность выживания нейтрино.

- CPU: Intel Core i7-6700HQ.

Количество ядер на видеокарте, на которой производилось тестирование, равно 1280 ядер CUDA. В случае, если для исполнения GPU функции требуется больше ядер, чем есть на устройстве, то общая задача делится на фрагменты. На одном ядре поочередно вычисляются значения для нескольких элементов массивов.

Часть трансформаций переносилась на GPU в рамках Летней Студенческой Программы 2018 при участии студента Ильи Лебедева под моим руководством. С отчетом о проделанной работе можно ознакомиться на сайте Летней студенческой программы [21] или на сайте группы нейтринной и астрофизики ЛЯП ОИЯИ [8].

Глава 5. Альтернативные способы ускорения

5.1 Фильтр GridFilter

В ходе работе над ускорением выполнения алгоритмов, описанных в GNA был также разработан фильтр GridFilter. Он был создан специально для алгоритма Фельдмана-Казинса, который является вычислительно сложным. Алгоритм Фельдмана-Казинса состоит из нескольких шагов:

- Разыгрывание Монте-Карло в каждой точке пространства параметров,
- На основе сгенерированной выборки определяется эмпирическое распределение $\Delta\chi^2$, полученное методом профилирования χ^2, между моделью и наилучшим значением подгонки,
- На основе этого распределения строятся уровни доверия.

Фильтр предполагает, что в рассмотрение для алгоритма Фельдмана-Казинса будут вводиться не все точки пространства параметров, а только те, которые близки к контурам распределения χ^2. Такой подход основан на наблюдении, что контуры, получаемые с помощью алгоритма Фельдмана-Казинса близки к распределению χ^2. На рис. 17 представлено их сравнение

из статьи, посвященным изучению нейтринных осцилляций в эксперименте Daya Bay [22].

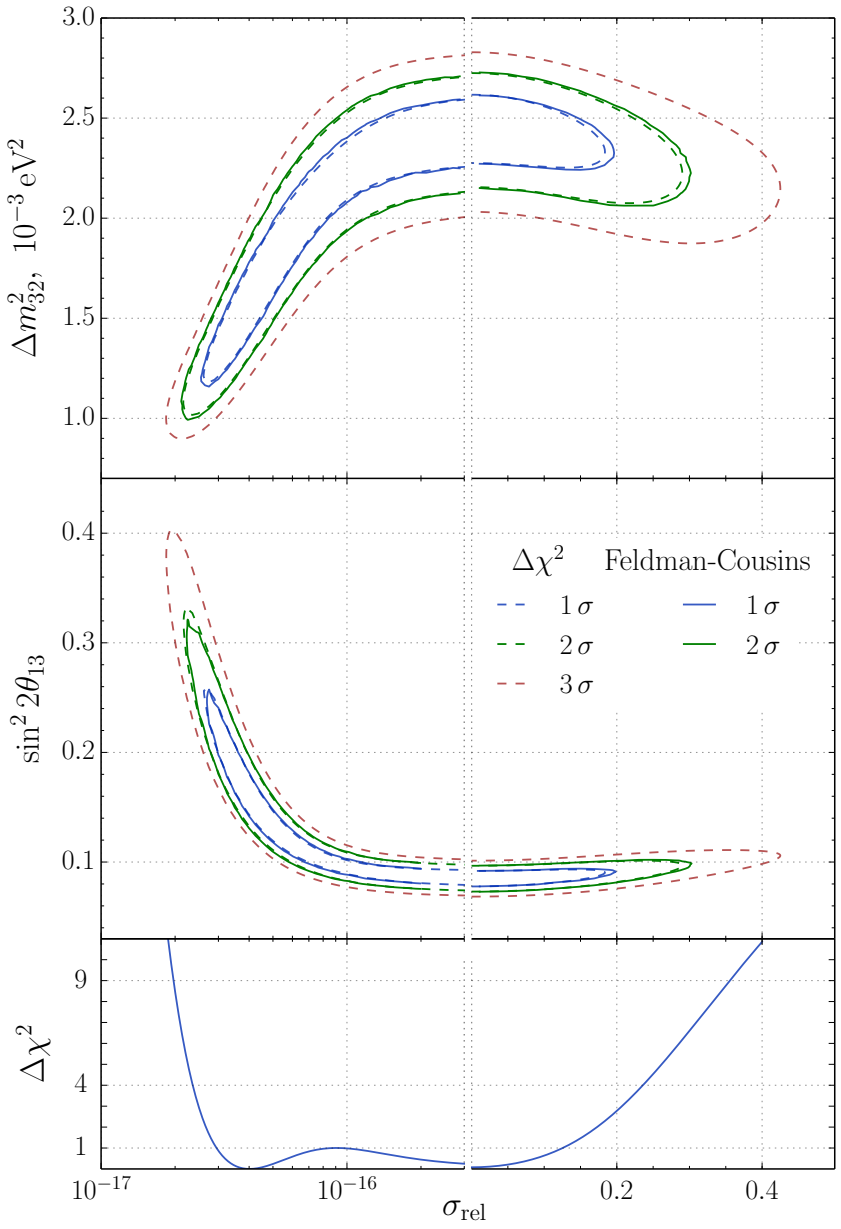

Рис. 17: Сравнение контуров χ^2 и Фельдмана-Казинса.

Алгоритм GridFilter выделяет среди пространства параметров лишь те точки, которые близки к распределению с точностью до некоторого заданного отклонения. Рассмотрим принцип работы фильтра на примере. Пусть задана двумерная функция $z = f(x, y)$. Двумерный массив значений z принимается за входные данные для алгоритма трансформации Grid Filter. Границы значений для x и y могут быть заданы пользователем. Так-

же пользователь может указать два дополнительных параметра `val` и `err`, которые являются опциональными. Этот модуль может работать в одном из следующих режимов:

1. Если заданы оба значения `val` и `err`, то в качестве выходного массива мы получим маску — двумерный массив, состоящий из значений 0 и 1, где 1 соответствует точкам (x, y), в которых $z = \texttt{val} + -\texttt{err}$, и 0 во всех остальных точках. Переменная `err` в таком случае представляет собой радиус сечения z.

2. Второй режим принимает в качестве входных данных также фиксированное значение отклонения. Это значение позволяет расширить радиус сечения.

3. При использовании режима автоматического подсчета отклонения используется среднее значение по всем градиентам заданной функции z для подсчета отклонения. Оно полагается фиксированным для всех значений z.

4. Последний режим этой трансформации — режим переменного отклонения. В этом случае отклонение считается для каждой точки исходя из градиента z в этой конкретной точке.

Различие между третьим и четвертым режимом представлены на рис. 18 и 19

Опционально пользователь может указывать параметры:

- Множитель влияния градиента — усиливает влияние градиента на величину отклонения.

- Множитель начального отклонение — фиксированное значение, на которое умножается значение отклонения, описанное в 1-4 пунктах.

- Радиус сечения по умолчанию `err`.

Влияние этих параметров на результат работы алгоритма показано на графиках в Приложениях 3 и 4.

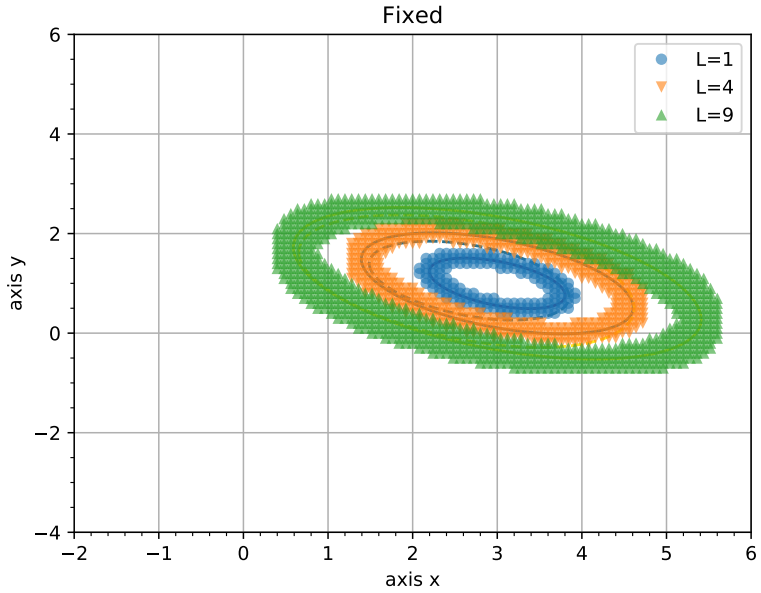

Рис. 18: Режим автоматического подсчета отклонения.

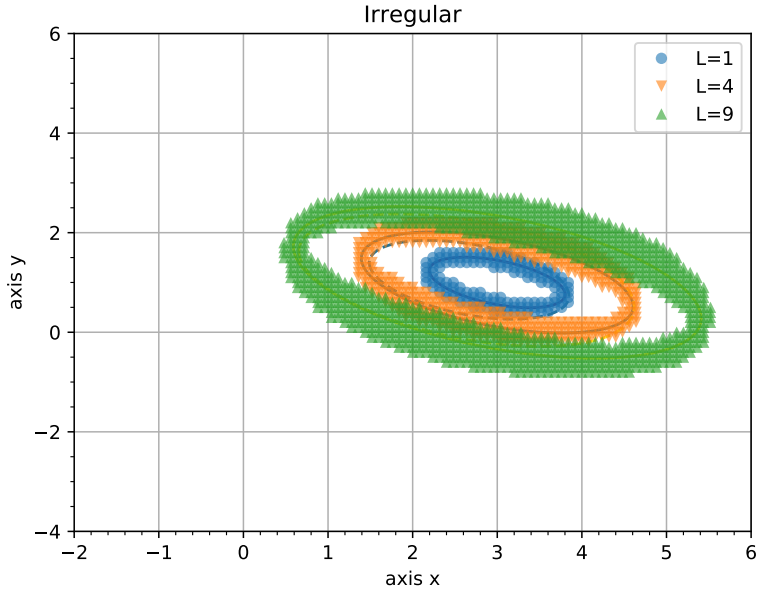

Рис. 19: Режим переменного отклонения.

5.2 Объединение трансформаций

Дополнительный вспомогательный инструмент для ускорения вычислений графа — это объединение набора однотипных трансформаций на одном уровне в одну. Рассмотрим на примере, представленном на рис. 20.

В таком графе все трансформации на втором уровне зависят от еди-

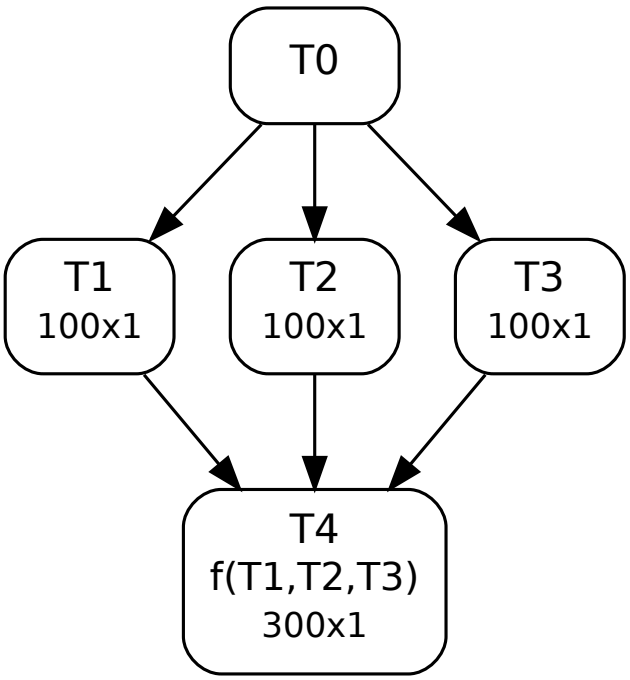

Рис. 20: Граф с тремя отдельными входами, в каждом входном массиве 100 элементов.

ного массива, который является результатом вычислений функции в трансформации первого уровня. В соответствии с архитектурой GNA, выходной массив трансформации, от которой зависит текущая, доступен только для чтения. Поэтому исключено возникновение ошибок совместного доступа к разделяемым входным данным. В случае, если каждая функция второго уровня использует GPU, то накладные расходы на непосредственно вызов CUDA функций возникнут столько раз, сколько функций второго уровня есть в графе. Если же объединить набор однотипных трансформаций в одну на этапе составления графа, то потребуется лишь один вызов GPU функции для всего графа, как это показано на рис. 21 с более сложной структурой выходного массива. Данная возможность и ее интеграция с библиотекой поддержки GPU вычислений находится в настоящий момент в разработке.

Выводы

Тестирование библиотеки cuGNA показало, что ее применение эффективно для задач нейтринной физики. Для функции вероятности осцилляций ускорение достигается на размерах входных данных, которые соответствуют модели эксперимента JUNO. Время обмена данными меж-

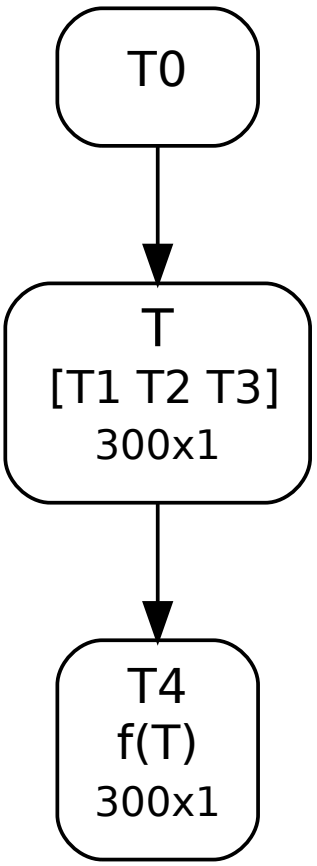

Рис. 21: Граф с одним объединенным входом.

ду оперативной памятью и графическим устройством было включено в подсчитываемое затраченное время при тестировании. Кроме того, все вычисления производились с использованием чисел двойной точности с плавающей точкой.

Для эксперимента Daya Bay использование GPU для подсчета вероятности осцилляций не является целесообразным. Тем не менее, его модель имеет ветвистую структуру с повторяющимися подграфами, так как эксперимент Daya Bay включает 8 детекторов. При объединении трансформаций одного уровня в одну, библиотека cuGNA имеет потенциал и для эксперимента Daya Bay.

Заключение

В этой работе описана библиотека поддержки вычислений на графических процессорах для платформы GNA. Библиотека была реализована как опциональный модуль платформы. В ее состав входит набор предопе-

ределенных трансформаций, а также логика синхронизации данных между оперативной памятью и памятью GPU.

Библиотека была протестирована на ряде трансформаций. Некоторые тесты приведены в этой работе. Для алгоритма вероятности осцилляций было достигнуто ускорение до 10 раз. В измеряемое время работы алгоритма входили также затраты на обмен данными между узлами вычислительного графа.

Тестирование показало, что в некоторых задачах использование GPU приведет к замедлению. При использовании входных данных малого объема выигрыш в производительности от использования параллельных потоков выполнения нивелируется затратами на обмен данными. Минимальный объем входных данных, при котором достигается ускорение, варьируется в зависимости от вычислительной сложности и используемых устройств.

Описанная библиотека показала свою эффективность на описанных в работе трансформациях на объемах данных, характерных для эксперимента JUNO. Она также имеет потенциал для более широкого круга задач, в том числе, с данными меньшего объема, при дополнительных модификациях, также описанных в этой работе.

Благодарности

Хотелось бы поблагодарить своих научных руководителей Максима Гончара и Александра Дегтярева за поддержку в написании диплома. Особую благоданость выражаю коллегам из ЛЯП ОИЯИ Максиму Гончару и Константину Трескову за неоценимую помощь в изучении области вычислительной физики нейтринных экспериментов и разработке, а также за помощь в подготовке текста этой работы. Спасибо Дмитрию Наумову, заместителю директора ЛЯП ОИЯИ, за поддержку идей, описанных здесь. Отдельная благодарность команде ЛИТ ОИЯИ, предоставившей доступ к облачным ресурсам ЛИТ, используемым при разработке GNA.

Список литературы

[1] T. Adam, F. An, G. An, Q. An, N. Anfimov, V. Antonelli, G. Baccolo, M. Baldoncini, E. Baussan, M. Bellato, and et al. JUNO Conceptual Design Report. ArXiv e-prints, August 2015.

[2] Jun Cao and Kam-Biu Luk. An overview of the daya bay reactor neutrino experiment. Nuclear Physics B, 908:62–73, 2016.

[3] Xinheng Guo. A precision measurement of the neutrino mixing angle theta_13 using reactor antineutrinos at daya bay. Technical report, 2007.

[4] Claudio Giganti, Stéphane Lavignac, and Marco Zito. Neutrino oscillations: the rise of the pmns paradigm. Progress in Particle and Nuclear Physics, 98:1–54, 2018.

[5] Carlo Giunti and Chung W. Kim. Fundamentals of Neutrino Physics and Astrophysics. 2007.

[6] Raymond Davis Jr. Solar neutrinos. ii. experimental. Physical Review Letters, 12(11):303, 1964.

[7] GNA home page. `https://astronu.jinr.ru/wiki/index.php/GNA`.

[8] Astrophysics and neutrino physics group web page. `https://astronu.jinr.ru/`.

[9] GNA documentation. `http://gna.pages.jinr.ru/gna/`.

[10] GNA reference guide. `http://gna.pages.jinr.ru/gna/reference_guide.html`.

[11] Sebastian Witowski. Python at cern. Technical report, 2017.

[12] Gael Guennebaud, Benoit Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[13] New York University. Introduction to GPUs: CUDA, 2017(27 мая 2019). `https://nyu-cds.github.io/python-gpu/02-cuda/`.

[14] Shane Cook. CUDA programming: a developer's guide to parallel computing with GPUs. Newnes, 2012.

[15] Tolga Soyata. GPU Parallel Program Development Using CUDA. Chapman and Hall/CRC, 2018.

[16] Anna Fatkina, Maxim Gonchar, Liudmila Kolupaeva, Dmitry Naumov, and Konstantin Treskov. Cuda support in GNA data analysis framework. In International Conference on Computational Science and Its Applications, pages 12–24. Springer, 2018.

[17] Anna Fatkina, Maxim Gonchar, Anastasia Kalitkina, Liudmila Kolupaeva, Dmitry Naumov, Dmitry Selivanov, and Konstantin Treskov. GNA: new framework for statistical data analysis. In Proceedings, 23nd International Conference on Computing in High Energy and Nuclear Physics (CHEP2018): Sofia, Bulgaria, July 9-13, 2018, 2018.

[18] Hyper-kamiokande. neutrino mass hierarchy. `http://www.hyper-k.org/en/physics/phys-hierarchy.html`.

[19] Максим Олегович Гончар. Измерение угла смешивания θ_{13} и расщепления масс нейтрино δm^2_{32} в эксперименте Daya Bay.

[20] CUDA GPUs, (27 мая 2019). `https://developer.nvidia.com/cuda-gpus`.

[21] Летняя студенческая программа ОИЯИ. `http://students.jinr.ru/ru`, 2018.

[22] Daya Bay Collaboration et al. Study of the wave packet treatment of neutrino oscillation at daya bay. The European Physical Journal C, 77(9):606, 2017.

Приложение 1

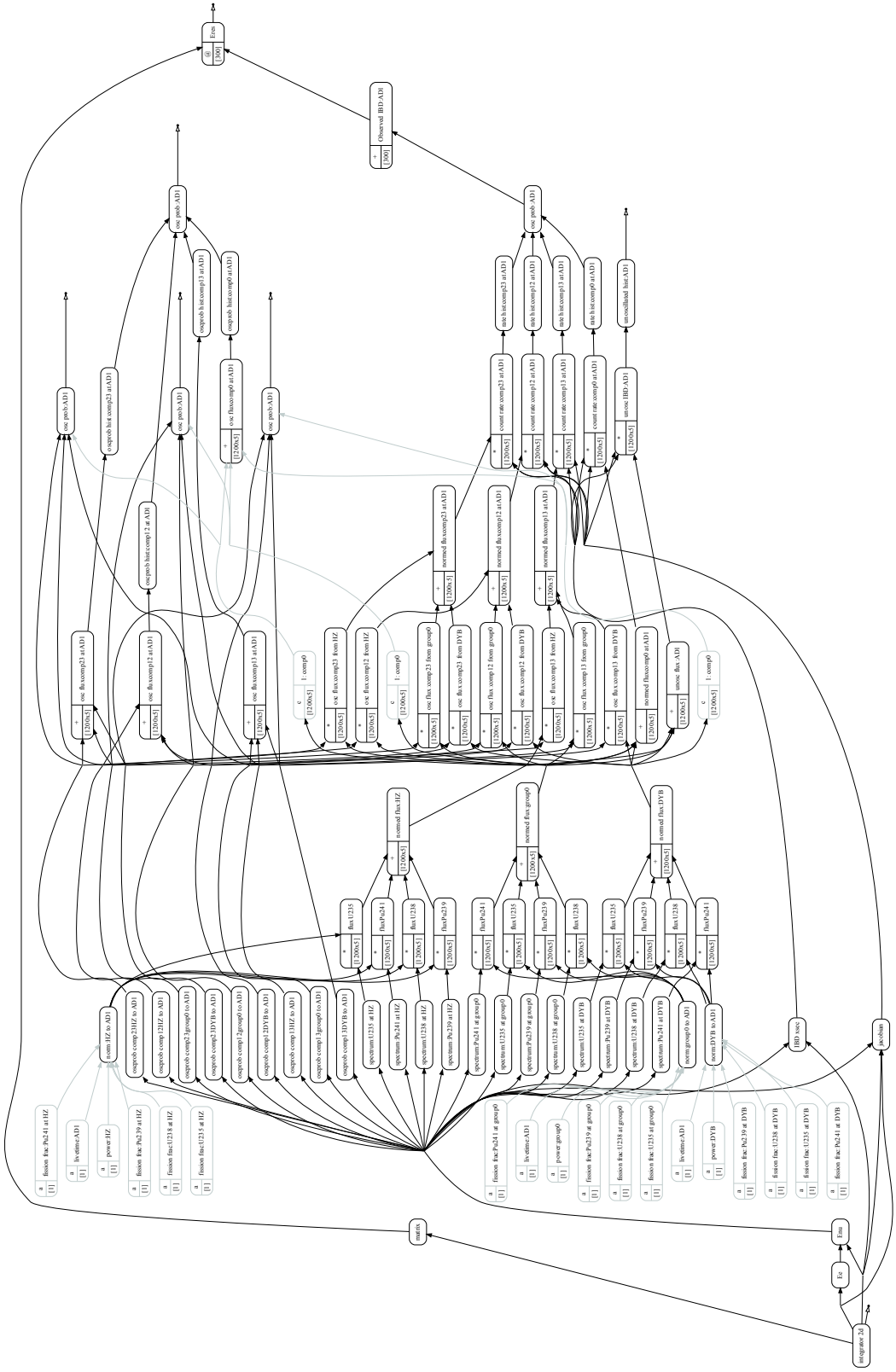

Рис. 22: Граф предсказания спектра антинейтрино в эксперименте JUNO. 123 узла, 206 ребер.

Приложение 2

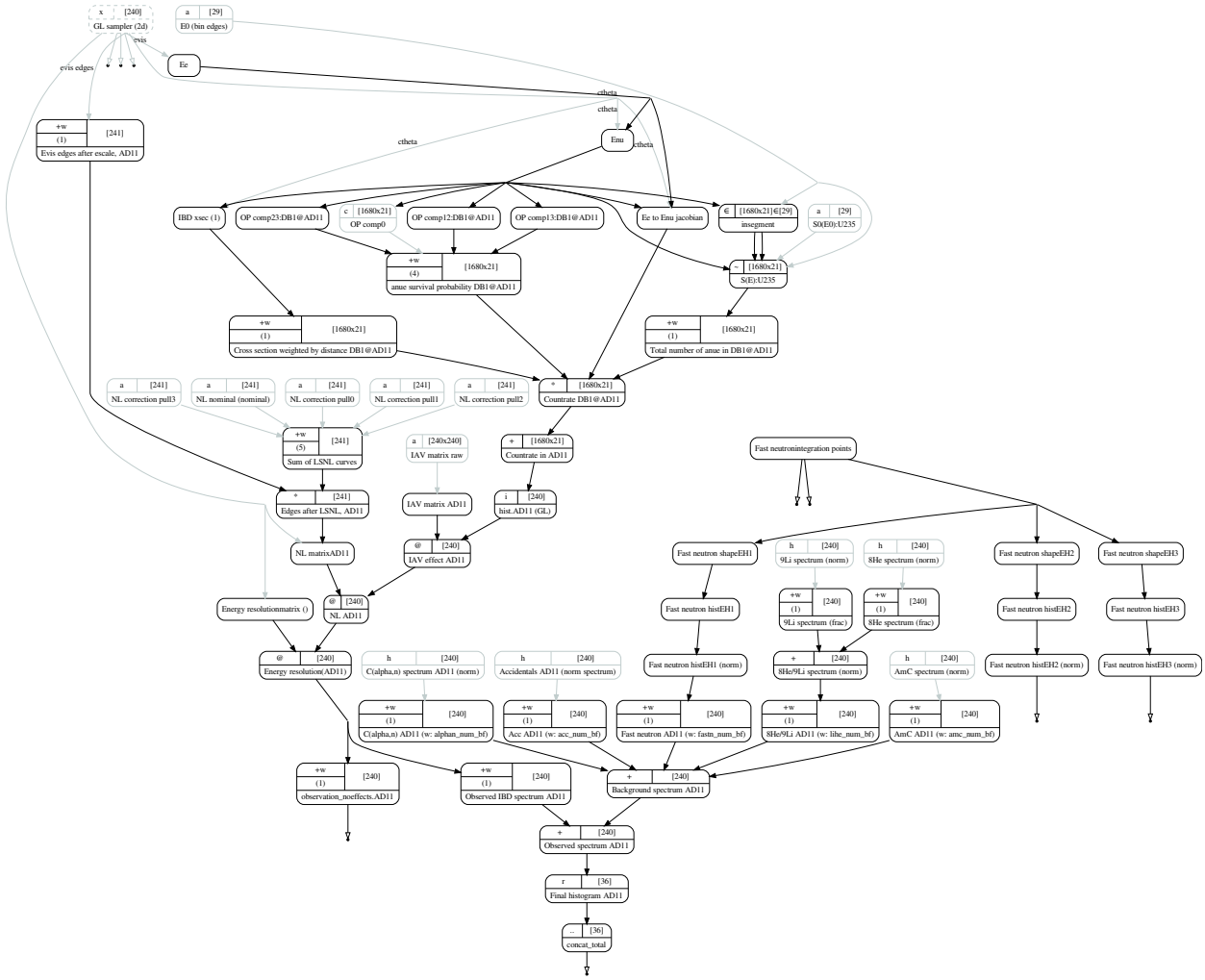

Рис. 23: Граф предсказания спектра антинейтрино в эксперименте Daya Bay для 1 из 8-и детекторов, 1 из 6-и реакторов, 1 из 4-х изотопов.

Приложение 3

Результат работы фильтра `GridFilter` для различных значений параметра влияния градиента.

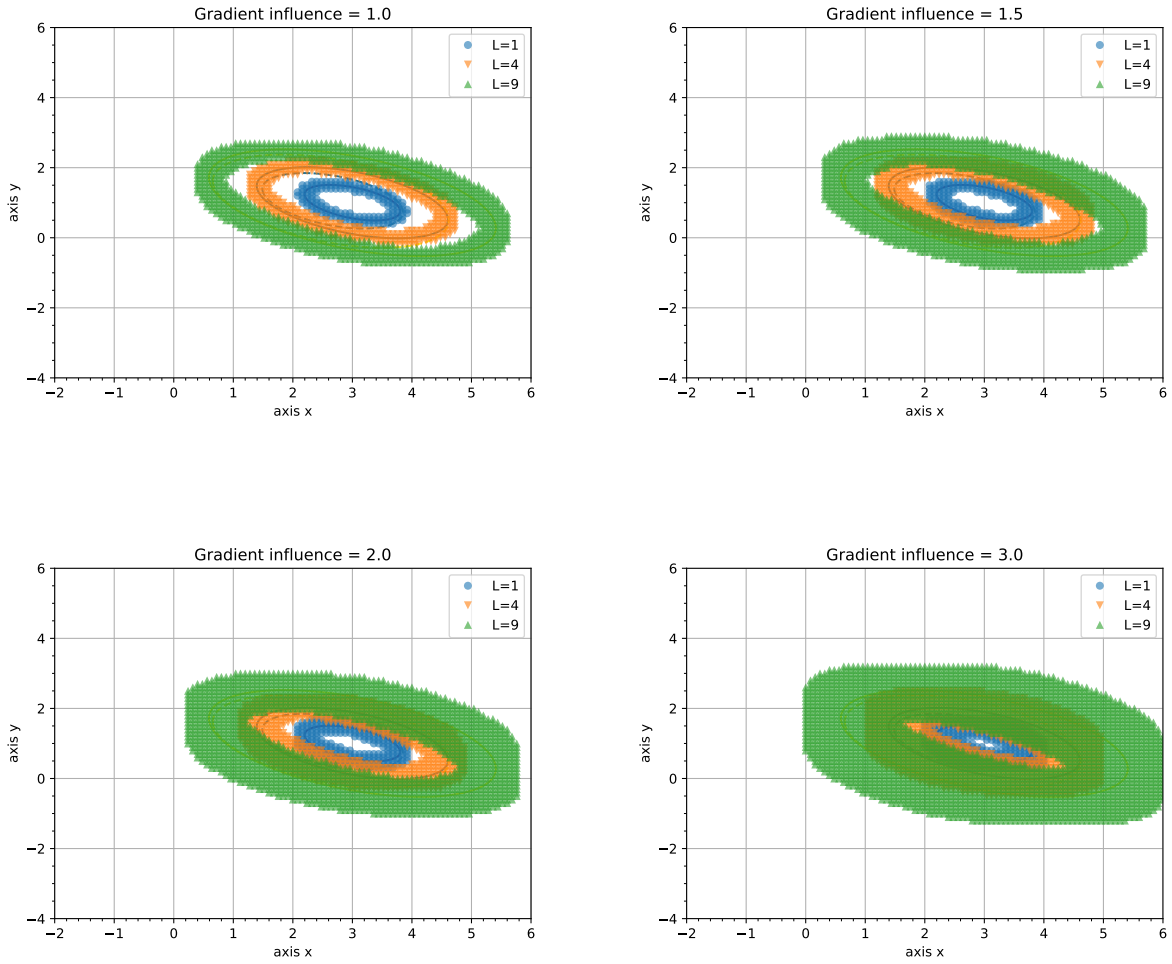

Рис. 24: Различные значения параметра влияния градиента — от 1 до 3.

Приложение 4

Результат работы фильтра `GridFilter` для различных значений входного параметра начального отклонения.

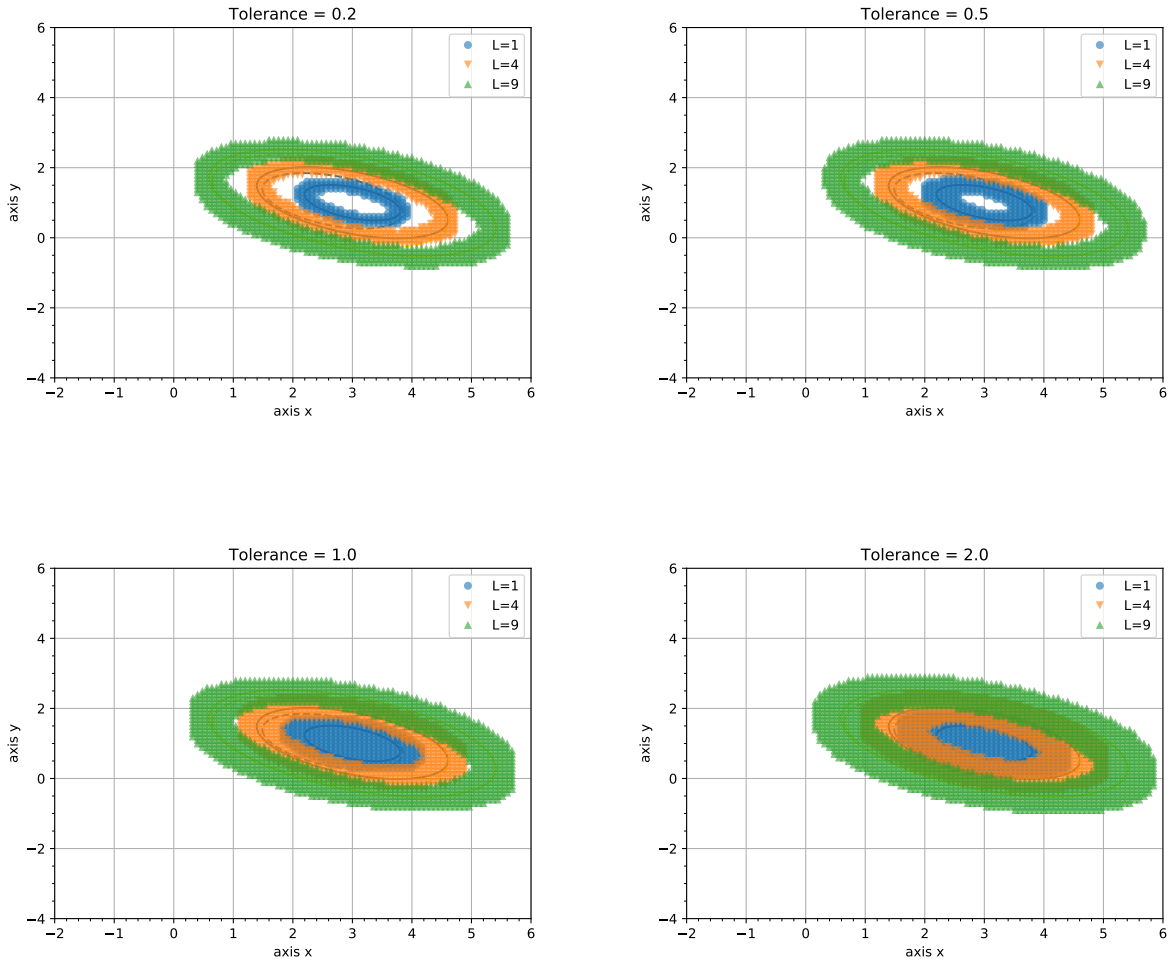

Рис. 25: Различные значения параметра начального отклонения.