ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧЕРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»
(Н И У « Б е л Г У »)

ИНСТИТУТ ИНЖЕНЕРНЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК

КАФЕДРА МАТЕМАТИЧЕСКОГО И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

РАЗРАБОТКА ПРОГРАММНО-АЛГОРИТМИЧЕСКОГО ОБЕСПЕЧЕНИЯ ДЛЯ МЕТОДОВ ИНТЕРПОЛЯЦИИ НА ОСНОВЕ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ В СИСТЕМЕ НЕДРОПОЛЬЗОВАНИЯ

Выпускная квалификационная работа
обучающегося по направлению подготовки 02.04.01 Математика и компьютерные науки, группы 07001631
Бондарева Александра Валерьевича

Научный руководитель
к.т.н., доцент
Васильев П.В

Рецензент
Д.т.н., профессор
Маторин С.И.

БЕЛГОРОД 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В цепи операций, включающей этапы от анализа исходных данных опробования до принятия решений, выполняются многочисленные расчеты в циклах статистической обработки первичных данных, интерполяции, оптимизации и планирования добычи, что требует значительных вычислительных мощностей. В этой связи, задача ускорения расчетов за счет распараллеливания вычислений на многоядерных компьютерах и GPU с интеграцией программных решений в программном коде системы моделирования недропользования является весьма актуальной.

Типичные примеры методов, основанные на геостатистических концепциях (Кригинг), локальность (ближайший сосед и конечные элементные методы), гладкость и напряжение (сплайны) или специальные функциональные формы (полиномы, мульти – полиномы). Выбор дополнительного условия зависит от характера моделируемого явления и типа применения.

Поиск подходящих методов интерполяции для задач географических информационных систем предполагает несколько проблем. Моделируемые объекты, как правило, очень сложны, пространственные данные неоднородны и часто основаны на оптимальной выборке и значительном шуме, также могут присутствовать разрывы. Кроме того, наборы данных очень большие.

Исходя из различных источников с различной точностью, надежные инструменты интерполяции, подходящие для ГИС-приложений, должны удовлетворять нескольким важным требованиям:

- точность и прогностическую мощность,
- надежность и гибкость в описании различных типов явлений,
- сглаживание для шумных данных,
- n-мерная формулировка,
- непосредственная оценка производных (градиенты, кривизна),

- применимость к крупным наборов данных,
- вычислительной эффективности и простоты использования.

В настоящее время трудно найти способ, который выполняет все вышеупомянутые требования для широкого диапазона данных с привязкой. Следовательно, выбор адекватного метода с соответствующими параметрами для конкретного приложения имеют решающее значение. Различные методы могут давать совершенно разные пространственные представления. Детальное знание всех нюансов горной добычи, а также опыт накопленный людьми необходимы для оценки метода и выбора самого точного и близкого к реальности. Использование непригодного метода или неуместных параметров могут привести к искаженной модели карьера, что приведет к потенциально неправильным решениям при добычи полезных ископаемых основанной на вводящей в заблуждение пространственной информации.

Неуместная интерполяция может иметь еще больше последствий, если результат используется как вход для симуляции, где небольшая ошибка или искажение могут создавать модели для основанные на ложных пространственных шаблонах.

Проблема исследования заключается в использовании эффективных технологий параллельного программирования, которые появились относительно недавно и еще не нашли широкого применения, для обработки больших объемов данных по методам пространственной интерполяции. Традиционно используемые библиотеки не используют параллельные технологии. Решение этой проблемы обозначило новый фронт работ.

Актуальность решения проблемы для науки и практики информационных систем недропользования состоит в решении задач текущего и стратегического планирования добычи и выполнения оперативного подсчета запасов сырья. При эксплуатации месторождений полезных ископаемых строятся масштабные сеточные и блочные горно-геологические модели.

Цель исследования состоит в том, чтобы повысить эффективность и быстродействие (производительность) систем управления запасами горнорудных предприятий при добыче и переработке путем интеграции новых технологий параллельного программирования в методы пространственной интерполяции.

Выдвижение данной цели обусловило постановку следующих исследовательских **задач**:

1. Рассмотреть процесс пространственной интерполяции в системах недропользования, определить механизмы реализации интерполяции и методы оценки ее эффективности;
2. Изучить два наиболее используемых в современной науке и практике метода пространственной интерполяции – метод обратных взвешенных расстояний и метод Кригинга;
3. Изучить возможности программно-аппаратных архитектур параллельного программирования OpenCL и CUDA;
4. Провести анализ методов интерполяции и обозначить подходы, принципы интеграции технологий параллельного программирования;
5. Исследовать возможность применения технологий параллельного программирования для создания механизма увеличения производительности алгоритмов интерполяции при обработке больших данных;
6. Выполнить проектирование и реализацию алгоритма пространственной интерполяции Кригинга с применением технологии параллельного программирования;
7. Разработать программный код OpenCL для метода пространственной интерполяции Кригинга;
8. Проанализировать результаты выполненной работы и сделать выводы.

Предметом исследования является процесс распараллеливания алгоритмов интерполяции с целью повысить быстродействия. Объектом исследования являются методы моделирования карьеров.

Гипотеза исследования заключается в решении проблемы увеличения быстродействия путем внедрения технологий параллельного программирования, которая использовала бы вычислительные ресурсы центрального процессора совместно с графическим. Классические методы осуществления распараллеливании программ на центральных процессорах не учитывают все нюансы работы с памятью графических процессоров. Поэтому возможность объединения вычислительных ресурсов центрального и графического процессора сможет дать увеличение производительности существующих алгоритмов. Кроме того, обладая кроссплатформенностью такая реализация сможет с легкостью портироваться на различные вариации платформ и аппаратных устройств.

Теоретико-методологическая основа исследования базируется на работах отечественных специалистов Васильева П.В., Петина А.Н., Каневского М.Ф., Демьянова В.В., а также на многочисленных научных работах западных специалистов Fang Huang, Shuanshuan Bu, Jian Tao, Xicheng Tan, Gisele Goulart Tavaresa, Leonardo Goliatta, Marcelo Lobosco и многих других ученых. Список научных трудов всех специалистов представлен в советующем разделе выпускной квалификационной работы, который называется «список используемой литературы».

Эмпирическую базу исследования составили результаты экспериментов на центральном процессоре от Intel и графических процессорах от NVIDIA и AMD, а также на центральных процессорах от Intel. Исходные данные были взяты с сайта http://github.com.

Научная новизна: впервые на основе анализа существующих примеров использования параллельных технологий был спроектирован и реализован алгоритм пространственной интерполяции Кригинга для обработки больших

данных, который обладает кроссплатформенностью, одновременном использовании на разного рода процессорах и может быть встроен в системы подсчета запасов полезных ископаемых.

Научно-практическая значимость исследования. Материалы исследования, а также его общие выводы свидетельствуют о возможности использования технологий параллельного программирования в методах пространственной интерполяции для гетерогенных систем недропользования.

Материал, представленный в диссертации, может быть использован при преподавании и изучении ряда дисциплин посвященным математике и компьютерным наукам: геостатистика, компьютерное и математическое моделирование, вариационные и дифференциальные вычисления, математическая статистика, параллельное программирование.

Практическое значение диссертационного исследования заключено в возможности разработки библиотек для существующих приложений обработки и анализа запасов месторождений. Библиотеки, основанные на научных исследованиях и практической реализации пространственной интерполяции, которые приведены в выпускной квалификационной работе, позволят горнодобывающим предприятиям оптимизировать свою работу на этапах анализа и проектирования с точки зрения экономики, рабочего времени и использования ресурсов организации.

Структура диссертации состоит из введения, трех глав, каждая глава имеет три раздела и общий вывод, в которых решаются поставленные исследовательские задачи, заключения, списка источников и литературы, а также приложений, необходимо дополняющих основной текст.

Первая глава посвящена изучению теоретико – методологическим основам пространственной интерполяции и программно – аппаратным архитектурам параллельных вычислений. Так же собраны идеи и уже существующие на данный момент современные системы, в которых применена концепция распараллеливания методов пространственной

интерполяции.

Во второй главе проведен анализ двух методов пространственной интерполяции, к которым была применена технология параллельных вычислений OpenCL. На основе этих разработок происходит планирование разработки алгоритма пространственной интерполяции Кригинга для гетерогенных систем недропользования на основе технологии параллельных вычислений OpenCL.

В третьей главе приведено развернутое описание реализации последовательного универсального алгоритма пространственной интерполяции Кригинга на кроссплатформенном стандарте OpenCL. Представлены сведения о проведенных исследовательских экспериментах с использованием аппаратных возможностей центрального вычислительного процессора Intel Xeon Phi, графических процессорах NVIDA GeForce 1080Ti и AMD Radeon RX 580.

1 ТЕОРЕТИКО – МЕТОДОЛОГИЧЕСКИЕ ОСНОВЫ ПРОСТРАНСТВЕННОЙ ИНТЕРПОЛЯЦИИ И ПРОГРАММНО – АППАРАТНЫХ АРХИТЕКТУР ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

1.1 Пространственная интерполяция геоданных

Пространственные и пространственно-временные распределения как физические и социально-экономические явления могут быть аппроксимирующими функциями, зависящими от местоположения в многомерном пространстве, как многомерный скаляр, вектор или тензорные поля. Типичными примерами являются высоты, климатические явления, свойства почвы, плотности населения, потоки материи и т. д. большинство из этих явлений характеризуются измеренными или оцифрованными данными точек, часто нерегулярно распределенные в пространстве и времени. Визуализация, анализ, и моделирование в рамках ГИС обычно основано на растровом представлении. Более того, явления могут измеряться с использованием различных методов (дистанционное зондирование, выборка и т. д.), приводящих к гетерогенным наборам данных с различными цифровыми представлениями и решениями, которые необходимо объединить для создания единой пространственной модели изучаемого явления.

Многие приближенные методы интерполяции были разработаны для прогнозирования значений пространственных явлений в местах с невыделенной выборкой. В ГИС приложениях эти методы были разработаны для поддержки преобразования между различными дискретными и непрерывными представлениями пространственных моделей. Как правило это было сделано для преобразования нерегулярных точек или строк данных в

растровое представление, или для повторной выборки между различными растровыми разрешениями.

В этой главе излагаются основные теоретико – методологические основы пространственной интерполяции, а также программно – аппаратной архитектуры параллельных вычислений CUDA и OpenCL. Роль и конкретные вопросы, обсуждаемые в интерполяции для приложений ГИС и методы, основанные на локальности, геостатистической, и вариационной концепции. Свойства методов интерполяции иллюстрируются примерами двумерных, трехмерных и 4-мерных интерполяций. В будущих исследованиях основное внимание уделяется надежным данным, анализу с автоматическим выбором параметров пространственной переменной интерполяции, а также моделированию основанному на процессах интерполяции.

Пространственная интерполяция является важной составляющей почти любой ГИС. Хотя базовые двумерные методы являются общими, внедрение многомерных инструментов ограничено самыми передовыми системами, из-за отсутствия структур геоданных, анализа и вспомогательных инструментов для многомерных и временных обработок. Несмотря на недавний прогресс в разработке методов и алгоритмов экспоненциального увеличения, вычислительная мощность пространственной интерполяции, особенно для больших и сложных наборов данных, является итеративной, трудоемкой задачей, требующей адекватное знание основных методов и их реализации.

В зависимости от приложения пространственная интерполяция может выполняться на трех уровнях интеграции с ГИС:

- в рамках более общей программы / команды;
- как специализированная команда;
- с использованием связи со специализированным программным обеспечением.

Интерполяцию, интегрированную на уровне подкоманды можно найти во многих прикладных программах ГИС, предназначенных для вычисления

наклона и аспекта, автоматической растровой повторной выборки, отслеживания потока, гидрологического моделирования и т. д.

В этом случае используются простые и быстрые локальные интерполяции, основанные на билинейных или локальных полиномиальных методах. Интерполяция полностью автоматическая, скрытая от пользователя, и этого достаточно для решения типовых научных задач, однако на практике в большинстве случаях это может привести к возникновению ошибок на поверхностях если применяется неправильный метод.

Интерполяции, интегрированные на командном уровне, служат функциями преобразования данных. В ГИС интегрирован ограниченный набор основных современных методов, чаще всего простые версии IDW (обратных взвешенных расстояний), TIN (нерегулярная триангуляционная сеть), Kriging (Кригинг) и сплайнов. Для коммерческих систем компромиссы в числовой эффективности, точности, надежности, ограничивают модификации методов. Поэтому необходимо тщательно оценивать результаты и, если возможно, использовать более чем одну независимую процедура интерполяции.

Интерполяции, выполненные с помощью специализированного программного обеспечения, связанного с ГИС, обеспечивает наиболее продвинутые и гибкие инструменты, однако обладают длительным импортом/экспортом данных или неудобной работой в другом программном обеспечении, где может потребоваться иная среда. Этот подход все еще является предпочтительным, особенно когда данные сложны и требуется высокая точность.

Инструменты систем моделирования с мощными возможностями интерполяции пространственных данных, часто предоставляют базовую обработку передовой поверхности объемных моделей и графическую обработку превращаются в специализированные ГИС.

Для широко распространенного, рутинного использования ГИС пользователями с небольшим опытом в области обработки пространственных данных, система предлагает полностью автоматический выбор методов интерполяции и их параметров, основанных на надежном анализе данных или априорной информации о смоделированных явлениях. С быстрым развитием коммуникационных технологий и доступности разнообразных данных в разных форматах, эта задача становиться одной из самой актуальной для практического применения.

Улучшение точности и реализма возможно с помощью адаптации интерполяции в параллельных вычислительных технологиях с дальнейшими разработками в области моделирования интерполяции. Когда массив данных объединен с предсказаниями модели, то полученные оценки позволяют фиксировать уникальные характеристик конкретной области при соблюдении общих физических принципов, которые контролируют процесс, влияя на пространственное распределение изучаемого явления. Это можно сделать, используя стохастическую / детерминированную модель процесса вместе с понятиями байесовской теории оценки.

Одним из важных событий в области геонаук является увеличение доступности данных, генерируемых различными источниками (например, локальные измерения, GPS, спутники, радар), которые имеют разнообразный характер точки зрения разрешения, точности, распределения и т. д. Это требует новых подходов к обработке данных и синтеза. Требуется извлекать информацию из всех источников данных, правильно взвешивать и оптимизировать.

Полная интеграция поддержки многомерных данных, включая структуры данных, аналитические и инструменты визуализации будут стимулировать многомерные приложения. Хотя методы были представлены уже полностью адаптированными к многомерным данными, в современных ГИС вычислительная инфраструктура имеет не эффективную поддержку

широкого применения многомерного и пространственно-временного моделирования. Высокоточная интерполяция больших наборов данных требует очень интенсивных вычислений, а производительность важна как для больших передовых приложений, а также для рутинного использования. В дальнейшем разработка алгоритмов и использование параллельных архитектуры станут одним из вариантов ускорения.

1.2 Методы пространственной интерполяции

Алгоритм пространственной интерполяции - это метод, в котором атрибуты в некоторых известных местах (точках) используются для прогнозирования атрибутов в некоторых неизвестных местах (интерполированных точках). Пространственные интерполяционные алгоритмы, такие как взвешивание обратного расстояния (IDW) [1], Кригинг [2] и дискретная гладкая интерполяция (DSI) [3, 4], обычно используются в геонауках и связанных с исследованиями, особенно в географической информационной системе(ГИС) [5]; см. краткое резюме в [6] и сравнительный обзор в [7].

Среди вышеупомянутых трех алгоритмов пространственной интерполяции только метод Кригинга является вычислительно интенсивным из-за инверсии матрицы коэффициентов, в то время как другие два легко вычислить. Однако, когда вышеупомянутые три алгоритма применяются к большому набору точек, например, более 1 миллиона точек, они все еще обладают довольно дорогостоящие вычисления, даже для простейшего интерполяционного алгоритма IDW.

Алгоритм IDW является одним из наиболее часто используемых методов пространственной интерполяции в геонауках, который вычисляет значения предсказания неизвестных точек (интерполированных точек),

взвешивая среднее значение значений известных точек (точек данных). Название, присвоенное этому типу методов, было основано на применении взвешенного среднего, поскольку оно прибегает к обратному расстоянию до каждой известной точки при вычислении весов. Разница между различными формами интерполяции IDW заключается в том, что они вычисляют весы по-разному.

Общий вид предсказания интерполированного значения Z в данной точке x на основе выборок $Z_i += Z(x_i)$ для i = 1,2, ..., n с использованием IDW является интерполяционной функцией:

$$Z(x) = \sum_{i=1}^{n} \frac{\omega_i(x) z_i}{\sum_{j=1}^{n} \omega(x)}, \qquad \omega_i(x) = \frac{1}{d(x, x_i)^a}. \tag{1.1}$$

Вышеприведенное уравнение представляет собой простую функцию взвешивания IDW, (1.1), где x обозначает местоположение предсказания, x_i - это точка данных, d - расстояние от известной точки данных x_i до неизвестной интерполированной точки x, n - общее число точек данных, используемых при интерполяции, а p - произвольное положительное действительное число, называемое параметром мощности или параметром распада (обычно a = 2 в взвешивающем алгоритме интерполяции стандартная IDW). Обратите внимание, что в стандартной IDW параметр «power» / «distance-decay» является заданным пользователем постоянным значением для всех неизвестных интерполированных точек.

Основная и самая важная идея AIDW заключается в том, что она адаптивно определяет параметр затухания расстояния a в соответствии с пространственной структурой точек данных в окрестности интерполированных точек. Другими словами, параметр расщепления расстояния a больше не является предварительно заданным постоянным значением, а адаптивно настроен для конкретной неизвестной

интерполированной точки в соответствии с распределением точек данных / выборочных местоположений.

При прогнозировании желаемых значений для интерполированных точек, использующих AIDW, обычно существует две фазы:

Первая - адаптивно определять параметр *a* в соответствии с пространственным шаблоном точек данных;

Вторая - выполнять средневзвешенное значение значений точек данных. Вторая фаза такая же, как и в стандартной IDW; см. уравнение (1.1).

В AIDW для каждой интерполированной точки адаптивное определение параметра *a* может быть выполнено на следующих этапах:

Шаг 1. Определение пространственной структуры, сравнив наблюдаемое среднее ближайшее соседнее расстояние с ожидаемым расстоянием ближайшего соседа.

- Вычислить ожидаемое расстояние ближайшего соседа r_{exp} для случайной диаграммы, используя

$$r_{\exp} = \frac{1}{2\sqrt{n/A}}, \qquad (1.1),$$

где n - количество точек в исследуемой области, а A - площадь области исследования.

- Вычислить наблюдаемое среднее расстояние ближайшего соседа r_{obs}, взяв среднее расстояние ближайшего соседа для всех точек:

$$r_{obs} = \frac{1}{k}\sum_{i=1}^{k} d_i, \qquad (1.\ 2)$$

где k - число ближайших точек, а d_i - расстояния ближайшего соседа, при чем k может быть определено до того интерполирования.

- Получить статистику ближайших соседей $R(S_0)$ на

$$R(S_0) = r\frac{r_{obs}}{r_{\exp}}, \qquad (1.3)$$

где S_0 - местоположение неизвестной интерполированной точки.

Шаг 2. Нормализовать меру R (S_0) до μ_R так, что μ_R ограничено 0 и 1 функцией нечеткого членства:

$$\mu_R = \begin{cases} 0 & R(S_0) \le R_{\min} \\ 0{,}5 - 0{,}5\cos\left[\dfrac{\pi}{R_{\max}}(R(S_0) - R_{\min})\right] & R_{\min} \le R(S_0) \le R_{\max} \\ 1 & R(S_0) \le R_{\max} \end{cases}, \qquad (1.4),$$

где $R_{\min}$ или $R_{\max}$ относится к статистическому значению локального ближайшего соседа (в общем случае $R_{\min}$ и $R_{\max}$ могут устанавливаться на 0.0 и 2.0 соответственно).

Шаг 3. Определите параметр расстояния-распада a, сопоставив значение μ_R с диапазоном a на треугольную функцию принадлежности, принадлежащую определенным уровням или категориям значений расстояния-распада как следует:

$$\alpha(\mu_R) = \begin{cases} \alpha_1 & 0.0 \le \mu_R \le 0.1 \\ \alpha_1\left[1 - 5(\mu_R - 0.1)\right] + 5\alpha_2(\mu_R - 0.1) & 0.1 \le \mu_R \le 0.3 \\ 5\alpha_3(\mu_R - 0.3) + \alpha_2\left[1 - 5\left(\mu_R - 0.3\right)\right] & 0.3 \le \mu_R \le 0.5 \\ \alpha_3\left[1 - 5(\mu_R - 0.5)\right] + 5\alpha_4(\mu_R - 0.5) & 0.5 \le \mu_R \le 0.7 \\ 5\alpha_5(\mu_R - 0.7) + \alpha_4\left[1 - 5\left(\mu_R - 0.7\right)\right] & 0.7 \le \mu_R \le 0.9 \\ \alpha_5 & 0.9 \le \mu_R \le 1.0 \end{cases}, \qquad (1.5),$$

где α_1, α_2, α_3, α_4, α_5 присваивается пять уровней или категорий значения расстояния-распада.

После адаптивного определения параметра α желаемое значение предсказания для каждой интерполированной точки может быть получено через средневзвешенное значение. Эта фаза такая же, как в стандартной IDW; см. уравнение (1.1).

Алгоритм обычного Кригинга (далее ОК) является важным методом в приложениях геостатистики для интерполяции данных, обычно основанных на полевых измерениях интересующей переменной. Используется алгоритм ОК для оценки значения переменной, основанной на дисперсии предыдущих

измерений. Алгоритм ОК имеет приложения в таких областях, как экологическая наука, гидрогеология, горная промышленность, прогнозирование погоды и дистанционное зондирование. Однако алгоритм ОК из-за его вычислительной стоимости, не способен к многократным вычислениям больших объемов данных.

Интерполяция является решающим шагом для предсказания переменной в неизвестных местах, важно использовать максимальное количество возможных измерений для увеличения производительности алгоритма прогнозирования. В главе сравнивается время выполнения эталонной последовательной реализации с разработанной параллельной, чтобы показать, что использование методов параллельного программирования может ускорить фундаментальные шаги многих важных научных исследований.

Процесс Кригинга, также известный как процесс регрессии Гаусса, представляет собой интерполяционный метод, в котором интерполированные значения моделируются Гауссовом процессом, который управляется предшествующими со-дисперсиями. Процесс Кригинга дает наилучшее линейное непредвзятое предсказание. Из его первоначальной конструкции для прогнозирования рудных сортов из пространственно-коррелированных данных проб в золотых рудниках Южной Африки этот метод интерполяции имеет, в настоящее время, приложения во многих областях, таких как нефтяная инженерия, геология, метеорология, гидрология, почвоведение, сельское хозяйство, борьба с загрязнением, общественное здравоохранение, рыболовство, растениеводство и животноводство, экология и дистанционное зондирование.

Процесс Кригинга подобен методу обратных взвешенных расстояний (IDW) в том, что он взвешивает окружающие измеренные значения, чтобы получить прогноз для неизмеримого местоположения, минимизируя дисперсию оценки причем

$$\sigma_E^2(x) = Var\left\{\hat{Z}(x) - Z(x)\right\},$$

$$\hat{Z}(x_0) = \sum_{i=1}^{n} \lambda_i Z(x_i), \tag{1.6}$$

путем моделирования его вариограммы через следующее уравнение

$$\gamma(h) = \frac{1}{2} E\left[(Z(x_i) - Z(x_j))^2\right], \forall i, j, \tag{1.7}$$

которое использует модель вариограммы γ (h) и веса λ_i в следующем случае:

$$\sum_{i=0}^{N} \lambda_i \gamma(x_i, x_j) + \phi = \gamma(x_j, x_0), \forall j, \tag{1.8}$$

переменное значение $\gamma(x_i, x_j)$, которое должно быть интерполировано в положение (i,j), λ - взвешенный коэффициент в точке i- значение, измеренное в точке.

Написав уравнение 1.8 в матричную запись, мы можем вывести вектор весов λ:

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{11} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \gamma_{n1} & \gamma_{n1} & \ddots & \gamma_{n1} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_n \\ \phi \end{bmatrix} = \begin{bmatrix} \gamma_{10} \\ \vdots \\ \gamma_{n0} \\ 1 \end{bmatrix} \Rightarrow K\lambda = k \Rightarrow \lambda = K^{-1}k. \tag{1.9}$$

Функция γ определяется как ковариация между двумя образцами ковариационной модели. Это может быть одна из многих моделей, но в этом исследовании используется модель сферической ковариации, которая определяется как

$$y(x) = \begin{cases} c_0 + c\left[\frac{3h}{2a} - (\frac{h}{2a})^3\right], & 0 < h \le a \\ c_0 + c, & h > a \end{cases}, \tag{1.10}$$

где y(0)=0.

С помощью этого набора уравнений алгоритм ОК можно суммировать в следующие шаги:

1. Используя уравнение 1.7, вычислим эмпирическую полувариантность для ввода набора данных;

2. Устанавливаем ковариационную модель в эмпирическую полувариантность (используя сферическую ковариационная модель, показанная в уравнении 1.10;

3. Вычисление ковариационной матрицы K с функцией ковариации введенной в эмпирическую полувариограмму;

4. Вычисление обратной ковариационной матрицы K^{-1};

5. Определение границы области, где точки должны быть интерполированными;

6. Разделение области в сетке пикселей для интерполяции;

7. Для каждого пикселя в области решается линейная система, показанная в уравнении 4 и находится вектор весов λ;

8. Наконец, для каждого пикселя решение уравнение 1.6 для предсказания значения $\hat{Z}(x)$ в координатах точки *x*.

Таким образом, алгоритм ОК должен соответствовать функции эмпирической полувариограммы, полученной путем осмотра в пространственной дисперсии во входных данных, а затем использования этой функции для интерполяции значения переменной по всей области. Вручную выборка области для некоторой переменной может составить пару сотен или даже тысяч образцов для интерполяции данных, и последовательная реализация может обрабатывать данные в этом порядке величин. Тем не менее, Кригинг все чаще используется по данным, собранным с использованием устройств дистанционного зондирования. Эти устройства могут легко собирать данные порядка десятков тысяч образцов, и их можно использовать для обработки в разумные сроки. Большинство соответствующих работ, представленных в исследовании, были сосредоточены на распараллеливании шага интерполяции, который имеет тенденцию быть самым дорогим. Хотя это может быть справедливо для

образцов, приобретенных вручную, но и с помощью устройств дистанционного зондирования процесс определения эмпирической полувариантности станет столь же дорогостоящим, как и шаг интерполяции.

Методы Кригинга - «Kriging» в отличие от всех описанных выше методов, которые относятся к классу детерминированных, Кригинг является геостатистическим методом. Кригинг строит скорее статистическую модель реальности, чем модель интерполяционной функции. Геостатистические методы основываются на вероятностной модели, рассматривающей изучаемую пространственную переменную Z (*x*,*y*) как реализацию случайной функции *Z* (*x*,*y*). Такой подход позволяет учитывать пространственную корреляцию данных и дает возможность не только создавать модели поверхностей, но также получать оценку точности этих моделей.

Реализованные в различных приложениях методы Кригинга решают задачу интерполяции с применением линейных оценок. Это три основные формы Кригинга:

- простой Кригинг (используется, если известно математическое ожидание случайной функции Z),
- ординарный Кригинг (математическое ожидание случайной функции Z неизвестно, но постоянно)
- универсальный Кригинг (математическое ожидание случайной функции Z неизвестно и непостоянно).

1.3 Идея распараллеливания алгоритмов интерполяции в системах недропользования

Сила распараллеливания на основе графического процессора также используется в другом геопространственном анализе, таком как анализ видов. Например, [31] предложена основа для геопространственного анализа на

основе графического процессора и обнаружено, что реализация графического процессора может привести к ускорениям, зависящим от набора данных, в диапазоне 28-925 раз для анализа видов [35]. Представлена основанная на GPU параллельная реализацию расчета видимости с нескольких точек зрения на сетках растровых рельефов [36]. Представлен алгоритм реального времени для просмотра в трехмерных сценах с использованием параллельных вычислительных возможностей графического процессора [37]. Предложен параллельный вычислительный подход к анализу рельефа больших данных местности с использованием графических процессоров.

Кроме того, [38, 39] разработана параллельная реализация сопоставления карт и обзорного анализа с использованием CUDA, который был выполнен на современных графических процессорах [40], представлена IO-эффективная параллельная реализация алгоритма просмотра R2 для больших карт местности на графическом процессоре CUDA [41]. Представлено новое преобразование XDraw viewhed алгоритма анализа в параллельный контекст для увеличения скорости, с которой возможен рендеринг [42]. Представлен алгоритм в режиме реального времени для просмотра в системах 3D Digital Earth (GeoBeans3D) с использованием параллельных вычислений графических процессоров.

Адаптивный алгоритм интерполяции адаптивного обратного расстояния (AIDW) [43] является улучшенной версией стандартной IDW. Стандартная IDW относительно быстро и легко вычисляет и просто интерпретирует. Однако в стандартной IDW параметр расстояния-распада применяется равномерно во всей области исследования без учета распределения данных внутри него, что приводит к менее точным предсказаниям по сравнению с другими методами интерполяции, такими как Кригинг [43]. В AIDW параметр «distance-decay» больше не является постоянным значением по всей интерполяции, но может быть адаптивно рассчитан с использованием функции, полученной из шаблона точки окрестности.

В большинстве случаев AIDW лучше, чем метод постоянных параметров, и лучше, чем обычный Кригинг, в тех случаях, когда пространственная структура *i* не может эффективно моделироваться с помощью типичных функций вариограммы. Короче говоря, стандартная IDW является логической альтернативой Кригинг, но AIDW предлагает лучшую альтернативу.

Как указано выше, при использовании в крупномасштабных приложениях стандартная IDW в целом является дорогостоящей вычислительной машиной. В качестве улучшенной и сложной версии стандартной IDW, AIDW в этом случае будет также дорогостоящим вычислительным алгоритмом. Однако, по мнению авторов, в настоящее время нет существующей литературы, посвященной разработке параллельных алгоритмов AIDW на графическом процессоре.

В этой главе представлены усилия, направленные на разработку и внедрение параллельного алгоритма интерполяции AIDW [43] на одном современном графическом процессоре (GPU). Сначала представлен простой, но подходящий для параллельного метода поиск ближайших точек. Затем разработаны две версии реализаций графического процессора, то есть нативная версия, которая не использует преимущества разделяемой памяти и черепичная версия, извлеченная из общей памяти. Также реализована как нативная версия, так и версия с черепицей, используя два макета данных для сравнения эффективности. Выявлено, что реализации графического процессора могут обеспечить удовлетворительные ускорения по сравнению с соответствующей реализацией ЦП для различных размеров тестовых данных.

Вклад в эту работу можно резюмировать следующим образом:

- Проектирование параллельного алгоритма интерполяции AIDW с использованием графического процессора
- Разработка практической реализации графического адаптера параллельного алгоритма AIDW.

Остальная часть этой главы организована следующим образом. Раздел 2.2 дает краткое введение в интерполяцию AIDW. В разделе приводятся соображения и стратегии ускорения интерполяции AIDW и подробности реализации на GPU. Представлены некоторые экспериментальные тесты, которые выполняются с единственной и / или двойной точностью. Обсуждаются экспериментальные результаты. В конце сделаны некоторые выводы.

Чтобы иметь возможность применять эти алгоритмы интерполяции в крупномасштабных приложениях, необходимо повысить вычислительную эффективность. Благодаря быстрой разработке архитектуры многоядерных процессоров (CPU) и многоядерной графической архитектуры (GPU), технология параллельных вычислений добилась значительного прогресса. Одной из наиболее эффективных и часто используемых стратегий повышения вычислительной эффективности алгоритмов интерполяции является распараллеливание процедуры интерполяции в различных широкомасштабных вычислительных средах на многоядерных процессорах и / или графических процессорах.

С 1990-х годов, многие исследователи посвятили себя распараллеливания различных алгоритмов интерполяции [8 – 12]. Специально для метода Кригинга, многие параллельные программы были реализованы на высокопроизводительных и распределенных архитектурах [11, 13 – 23]. Кроме того, для сокращения вычислительных затрат в крупномасштабных приложениях алгоритм IDW был распараллелен в различных широкомасштабных вычислительных средах на многоядерных платформах ЦП и / или GPU.

Обычный Кригинг (OK) является самым популярным из всех вариантов. Единственные требования к применению OK - это знание теории вариограмм и данных проб для его реализации. Поэтому, OK является алгоритмом Кригинга по умолчанию, предлагаемым многими географическими

информационными системами (ГИС). ОК используется для создания трехмерной модели карьера для добычи полезных ископаемых, полученную при взятии проб и зондировании с определенных высот. Будучи фундаментальным шагом к определению модели, алгоритм Кригинга отличный кандидат на распараллеливании в гетерогенных вычислительных системах из-за его параллельного характера. Тем не менее, существуют различные работы, в которых распараллеливание Кригинга представлено и обсуждено. В этих работах представлены параллельные реализации Кригинга и других геостатистических процедур посредством использования MPI, OpenMP, PVM и / или GPU (графических процессоров). Несмотря на все усилия по распараллеливанию Кригинга, по-прежнему есть возможности для улучшения работы этого алгоритма.

Например, используя преимущества традиционных моделей параллельного программирования на основе процессоров [8, 9] реализована интерполяция IDW параллельно с использованием Fortran 77 на параллельных суперкомпьютерах с общей памятью и достигла эффективности, близкой к 0,9. [10]. Параллельные алгоритмы IDW выполнены с использованием открытой многопроцессорной обработки (OpenMP), работающей на Intel Xeon 5310, достигая отличной эффективности 0,92 [24]. Параллельный алгоритм интерполяции IDW разработан с интерфейсом передачи сообщений (MPI), включив в него интерфейс передачи сообщений, множественные данные (SPMD) и режимы программирования ведущего / ведомого (M / S) и достиг своего ускорения почти в 6 раз и эффективность более 0,93 в кластере Linux, связанная с шестью независимыми ПК [25]. Параллельная версия интерполяции IDW разработана с использованием виртуальной машины Java (JVM) для многопоточных функций, а затем применена для прогнозирования распределения суточных тонкодисперсных матовых PM 2.5.

Поскольку вычисления общего назначения на современных графических процессорах могут значительно сократить вычислительно время

за счет проведения массовых параллельных вычислений, текущие исследовательские усилия посвящены параллельным алгоритмам IDW на вычислительных архитектурах GPU, таких как Compute Unified Device Architecture (CUDA) [26] и Open Computing Language (OpenCL) [27]. Например, [28, 29] алгоритм IDW развернут на графических процессорах, чтобы ускорить прогнозирование глубины снежного покрова. [14]. Изучено поведение алгоритма IDW на одном графическом процессоре в зависимости от количества значений данных, количества мест прогнозирования и различных соотношений размеров данных [30]. Выполнен стандартный алгоритм IDW с использованием Thrust, PGI Accelerator и OpenCL [31, 32]. Разработана реализации GPU оптимизированного алгоритма IDW и получено 13-33-кратное ускорение во время вычисления по последовательной версии.

И совсем недавно (33) разработаны две реализации GPU интерполяции IDW: алгоритм черепичной версии и версии CUDA. Воспользовавшись разделяемой памятью и динамическим параллелизмом CUDA обнаружено, что черепичная версия имеет ускорение 120% и 670% от версии процессора, когда параметр мощности p был установлен на 2 и 3.0 соответственно, но версия CDP в 4,8-6,0 раза медленнее, чем нативная версия графического процессора. Кроме того, [34] сделано сравнение и проанализировано влияние компоновки данных на эффективность реализации IDW с ускорением GPU.

Было проведено исследование параллельной реализации алгоритма OK, который использует преимущества гетерогенных вычислительных сред с помощью использования Open-Computing Language (OpenCL). Алгоритм OK используется для оценки цифровой модели карьера. Реализация заключается в эффективном использовании всех вычислительных доступных ресурсов для ускорения процедуры OK. Реализация может масштабироваться на нескольких процессорах и устройствах графического процессора и способна обрабатывать объемы данных, которые обычно производятся с помощью реальных измерений.

В этой главе представлено определение пространственной интерполяции, приведены примеры методов, используемых в реальных приложениях ГИС для изучения процесса недропользования. Очевидно, что за прошедшее десятилетие, с точки зрения точности, надежность разнообразных приложений решило большой объем проблем. Однако выводы, изложенные Берроу еще в 1986 году, остаются в силе: «Неразумно бросать свои данные в первый доступный метод интерполяции без внимательного изучения, так как результаты будут получены по предположениям, присущим методу. Хорошая ГИС должна включать в диапазон интерполяции такие методы, которые позволяют пользователю выбирать наиболее подходящий метод для работы». Компьютеры взяли на себя большую часть этой нетривиальной задачи, но многие проблемы еще предстоит решить.

В этой главе сформулирована проблема пространственной интерполяции рассеянных данных, как метода для прогнозирования и представления трехмерных моделей карьеров для горнодобывающей промышленности.

Представлены общие понятия о методах проектирования и реализации параллельной версии алгоритма OK и IDW, демонстрации его производительности и масштабируемости, способности одновременного исполнения на множестве вычислительных устройств. Одним из важных вкладов работы является распараллеливание этапа обработки вариограммы. Это позволяет избежать обмена данными между хостами и устройствами, а также использует вычислительную мощность как для центральных процессоров, так и для графических процессоров

Таким образом, для применения метода Кригинга предварительно должна быть получена модель вариограммы - меры пространственной корреляции данных, которая и является основным параметром метода.

Основные этапы создания геостатистической модели включают:

1) анализ и предварительную обработку данных (декластеризация, выявление трендов и областей пространственной неоднородности, анализ распределения, выпадающих значений, анизотропии),

2) расчет значений эмпирической вариограммы или ковариации,

3) построение модели вариограммы или ковариации,

4) решение системы уравнений Кригинга для определения весов,

5) получение прогнозного значения и ошибки (неопределенности) оценки в произвольной точке области исследования (например, в узлах регулярной сетки).

2 АНАЛИЗ СУЩЕСТВУЮЩИХ МЕТОДОВ И ИНСТРУМЕНТОВ ДЛЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ ПРОСТРАНСТВЕННОЙ ИНТЕРПОЛЯЦИИ В СИСТЕМЕ НЕДРОПОЛЬЗОВАНИЯ

2.1 Ускоренный алгоритм интерполяции с адаптивным обратным расстоянием с применением параллельных технологий

Алгоритм AIDW по своей сути подходит для распараллеливания архитектуры GPU. Это связано с тем, что в AIDW желаемое значение предсказания для каждой интерполированной точки может быть рассчитано независимо, что означает, что естественно вычислять значения прогнозирования для многих интерполированных точек одновременно без каких-либо зависимостей между интерполирующими процедурами для любой пары интерполированных точек.

Из-за присущей функции алгоритма интерполяции AIDW разрешается один поток вычисления значения интерполяции для интерполированной точки. Например, предполагая, что существует *n* точек интерполяции, которые необходимы, чтобы предсказать значения, такие как глубины, а затем необходимо выделить *n* потоков для одновременного вычисления желаемых значений предсказания для всех этих *n* интерполированных точек. Поэтому метод AIDW вполне подходит для распараллеливания архитектуры GPU.

Ожидается, что в GPU-вычислениях общая память будет намного быстрее глобальной памяти; таким образом, любая возможность заменить доступ к глобальной памяти доступом к общей памяти должна быть использована [7]. Общая стратегия оптимизации называется «tiling», которая разделяет данные, хранящиеся в глобальной памяти, на подмножества,

называемые «плитки», так что каждая плитка вписывается в общую память [15].

Эта стратегия оптимизации «tiling» также используется для ускорения алгоритма интерполяции AIDW: координаты точек данных сначала переносятся из глобальной памяти в общую память так, что каждый поток в блоке потока может одновременно обращаться к координаторам, хранящимся в совместно используемой памяти. Поскольку разделяемая память, находящаяся в графическом процессоре, ограничена в поточном мультипроцессоре, данные в глобальной памяти, то есть координаты точек данных, должны сначала разбивать на мелкие кусочки, а затем трассировать в общую память. Используя стратегию «черепицы», глобальные обращения к памяти могут быть значительно уменьшены; и, следовательно, ожидается, что общая вычислительная эффективность будет улучшена.

Существенное различие между алгоритмом AIDW и стандартным алгоритмом IDW заключается в том, что в стандартной IDW мощность параметра α задается постоянным значением (например, 2 или 3.0) для всех точек интерполяции, тогда как, напротив, в AIDW мощность α адаптивно определяется в соответствии с распределением интерполированных точек и точек данных. Короче говоря, в IDW мощность α определяется пользователем и остается неизменной перед интерполяцией; но в AIDW мощность α больше не указана пользователем или не является константой, а определяется в процессе интерполяции.

Основные этапы адаптивного определения мощности α в AIDW перечислены в п. 2.2. Среди этих шагов наиболее вычислительно интенсивный шаг для поиска k ближайших соседей (kNN) для каждой интерполированной точки. Несколько эффективных алгоритмов kNN были разработаны области разделения с использованием различных структур данных (21, 44 – 46). Однако эти алгоритмы на практике сложны и не подходят для реализации

AIDW. Это связано с тем, что в AIDW поиск kNN должен выполняться в одном потоке CUDA, а не в поточном блоке или сетке.

В этой главе представлен простой, но подходящий для алгоритма с графическим процессором алгоритм поиска *k* ближайших точек данных для каждой интерполированной точки. Предполагая, что имеется *n* интерполированных точек и m точек данных, для каждой интерполированной точки мы выполняем следующие шаги:

Шаг 1. Вычислить первые *k* расстояний между первыми *k* точками данных и интерполированными точками; например, если значение *k* установлено равным 10, то есть 10 расстояний, которые необходимо вычислить; см. строку (*a*) на рисунке 2.1.

Шаг 2. Сортировать первые *k* расстояний в порядке возрастания; см. строку (*b*) на рисунке 2.1.

Шаг 3. Для каждой из остальных (*m* - *k*) точек данных, вычислить расстояние *dist*; например, расстояние составляет 4,8 (*dist* = 4.8);

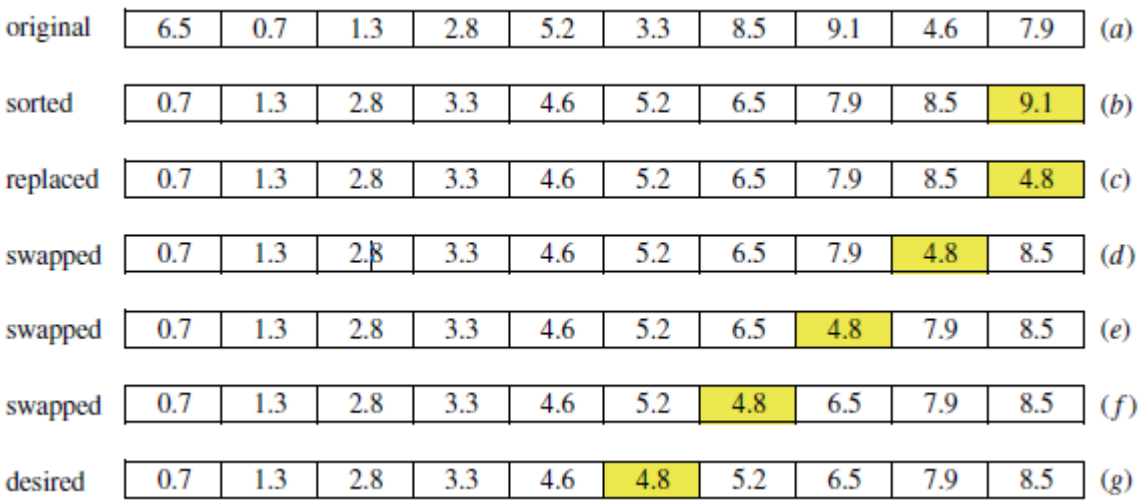


Рисунок 2.1. (*a*-*g*) Демонстрация нахождения *k* ближайших соседей (*k*=10).

- сравнить расстояние с *k*-ым расстоянием: если *dist* < *k*-го расстояния, затем заменить *k*-ое расстояние на *dist* (см. строку (*c*));
- итеративно сравнивать и менять соседние два расстояния с *k*-го расстояния до 1-го расстояния до тех пор, пока все *k*-дистанции не будут отсортированы в порядке возрастания; см. строки (*c*) - (*g*) на рисунке 2.1.

Макет данных - это форма, в которой данные должны быть организованы и доступны в памяти при работе с многозначными данными, такими как наборы трехмерных точек. Выбор подходящего макета данных является решающей проблемой при разработке приложений с ускорением GPU. Эффективность работы одного и того же приложения GPU может сильно отличаться из-за использования различных типов макетов данных.

Как правило, существуют два основных варианта компоновки данных: массивов структур (AoS) и структур (SoA) [47]; другой тип макета данных, массив выравниваемых структур (AoaS) [34], может быть очень легко сгенерирован путем добавления принудительного выравнивания на основе макете AoS. Фактически, компоновку данных AoaS можно рассматривать как улучшенный вариант компоновки AoS; см. эти три макета, то есть SoA, AoS и AoaS на рисунке 2. Организация данных в макете AoS приводит к совместным проблемам, поскольку данные чередуются. Напротив, организация данных в соответствии с компоновкой SoA обычно может в полной мере использовать пропускную способность памяти из-за отсутствия чередования данных [47]. Кроме того, глобальные обращения к памяти, основанные на макете SoA, всегда объединены.

(a)
```
struct Pt {
  float x[N];
  float y[N];
  float z[N];
};
struct Pt myPts;
```

(b)
```
struct Pt {
  float x;
  float y;
  float z;
};
struct Pt myPts[N];
```

(c)
```
struct
__align__(16) Pt
{
  float x, y, z;
 /* plus hidden 32bit
 padding element */
};
struct Pt myPts[N];
```

Рисунок 2.2. Структуры данных SoA (*a*), AoS (*b*) и AoaS (*c*).

На практике не всегда очевидно, что макет данных достигнет более высокой производительности для конкретного приложения GPU. Общим решением является внедрение в качестве конкретного приложения с использованием разных макетов данных по отдельности, а затем сравнение характеристик. Как упоминалось ранее, может быть учтен макет данных AoaS как улучшенный вариант компоновки AoS; и было сообщено, что компоновка данных AoaS может повысить эффективность больше, чем макет AoS (34). В этой главе оценивается влияние производительности двух карт данных SoA и AoaS

В этом разделе будут представлены детали реализации алгоритма интерполяции AIDW с ускорением GPU. Разработано две версии:

1. нативная версия, которая не использует преимущества разделяемой памяти;
2. версия с черепицей, которая использует память в общей памяти.

И для обеих вышеупомянутых двух версий две реализации разработаны отдельно в соответствии с двумя форматами данных SoA и AoaS. Весь исходный код представленного параллельного алгоритма AIDW является общедоступным [48].

В нативной версии используются только регистры и глобальной памяти, не получая выгоды от использования разделяемой памяти. Входные данные и выходные данные, то есть координаты точек данных и интерполированных точек, сохраняются в глобальной памяти.

Предполагая, что для оценки интерполированных значений для n точек предсказания используются m точек данных, выделяется n потоков для выполнения распараллеливания. Другими словами, каждый поток в сетке отвечает за предсказание желаемого значения интерполяции одной интерполированной точки.

Полное ядро CUDA указано в приложении 1. Координаты всех точек данных и точек предсказания сохраняются в массивах REAL dx[dnum], dy[dnum], dz[dnum], ix[inum], iy[inum] и iz[inum]. Слово REAL определяется как float и double на одиночной и двойной точности, соответственно.

В пределах каждого потока, сначала находится k ближайших точек данных для вычисления r_{obs} (см уравнение (1.3)) в соответствии с простым подходом, введенным в этой главе методом для нахождения ближайших точек данных (см. фрагмент кода из строки 11 в строку 34 в приложении 1); то вычисляется $r_{\exp}$ и $R(S_0)$ согласно уравнениям (1.2) и (1.4). После этого нормируется мера $R(S_0)$ на μ_R так , чтобы μ_R был ограничен 0 и 1 функцией нечеткого членства (см. уравнение [2. 5] и код с строки 38 по строку 40 в приложении 1). Наконец, определяется параметр расстояния-распада α путем сопоставления значений μ_R с диапазоном α функцией треугольного членства (см. уравнение (1.6) и код из строки 42 в строку 49 в приложении 1).

После адаптивного определения параметра мощности α снова вычисляется расстояние до всех точек данных, а затем в соответствии с расстояниями и заданным параметром мощности α получаем все m весов. В итоге желаемое значение интерполяции достигается с помощью

средневзвешенного значения. Эта фаза расчета средневзвешенного значения такая же, как в стандартном методе IDW.

Следует обратить внимание на то, что в нативном варианте необходимо дважды вычислить расстояния от всех точек данных до каждой точки прогнозирования. В первый раз это выполняется, чтобы найти *k* ближайших соседей / точек данных (см. код от строки 11 до строки 32 в приложении 1), а во второй, чтобы вычислять веса обратные расстояниям (см. код от строки 52 до строки 57 в приложении 1).

Рабочий процесс этой черепичной версии такой же, как у наивной версии. Основное различие между этими двумя версиями заключается в том, что в этой версии используется общая память для повышения эффективности вычислений. Основные идеи этой черепичной версии заключаются в следующем.

Ядро CUDA представленное в приложении 1 - простая реализация алгоритма AIDW, который не использует преимущества общей памяти. Каждый поток должен читать координаты всех точек данных из глобальной памяти. Таким образом, координаты всех точек данных необходимо читать *n* раз, где *n* - количество интерполированных точек.

В вычислениях с графическим процессором довольно часто используемой стратегией оптимизации является «черепица», которая разделяет данные, хранящиеся в глобальной памяти, на подмножества, называемые плитами, так что каждая плитка вписывается в общую память [15]. Эта стратегия оптимизации «tiling» принимается для ускорения интерполяции AIDW: координаты точек данных сначала переносятся из глобальной памяти в общую память; то каждый поток в блоке потока может одновременно обращаться к координатам, хранящимся в совместно используемой памяти.

В версии с черепицей размер напрямую устанавливается так же, как размер блока (т. Е. Количество потоков на блок). Каждый поток в поточном

блоке берет на себя ответственность за загрузку координат одной точки данных из глобальной памяти в общую память, а затем вычисляет расстояния и обратные веса для тех точек данных, которые хранятся в текущей общей памяти. После того, как все потоки внутри блока завершили вычисление этих частичных расстояний и весов, следующая часть данных в глобальной памяти загружается в общую память и используется для вычисления текущей волны частичных расстояний и весов.

Следует отметить, что: в черепичной версии необходимо дважды вычислить расстояния от всех точек данных до каждой точки прогнозирования. В первый раз это выполняется, чтобы найти *k* ближайших соседей / точек данных; и второй раз - вычислять расстояния-обратные веса. В этой черепичной версии обе эти две волны расчета расстояний оптимизируются с использованием стратегии «черепицы».

Используя стратегию «мозаики» и использования общей памяти, глобальный доступ к памяти может быть значительно уменьшен, поскольку координаты всех точек данных считываются только (n / threadsPerBlock) раз, а не *n* раз из глобальной памяти, где *n* - количество точек прогнозирования и threadsPerBlock обозначает количество потоков на блок. Кроме того, как указано выше, стратегия «черепицы» применяется дважды.

После вычисления каждой волны частичных расстояний и весов каждая нить накапливает результаты всех парциальных весов и всех взвешенных значений в два регистра. Наконец, значение предсказания каждой интерполированной точки может быть получено в соответствии с суммами всех парциальных весов и взвешенных значений, а затем записана в глобальную память.

На практике влияние компоновки данных на вычислительную эффективность сильно зависит от:

1. конкретной проблемы, которая должна быть решена,

2. графических процессоров, используемых для решения целевой задачи.

Таким образом, влияние компоновки данных на вычислительную эффективность может различаться на разных графических процессорах. Следует также отметить, что не всегда очевидно, какой формат данных достигнет лучшей производительности для конкретного приложения.

В архитектуре GPU общая память по своей сути намного быстрее, чем глобальная память; поэтому должна быть использована любая возможность заменить доступ к глобальной памяти доступом к общей памяти.

В черепичной версии координаты точек данных, первоначально сохраненных в глобальной памяти, делятся на мелкие фрагменты / плитки, которые соответствуют размеру разделяемой памяти, а затем загружаются из медленной глобальной памяти в быструю разделяемую память. Эти координаты, хранящиеся в общей памяти, могут быть доступны довольно быстро всеми потоками в поточном блоке при расчете расстояний. Блокируя вычисления таким образом, мы используем быструю разделяемую память и значительно уменьшаем доступ к глобальной памяти: координаты точек данных считываются только (n / threadsPerBlock) раз из глобальной памяти, где n - количество точек предсказания.

Вот почему черепичная версия быстрее, чем нативная версия. Поэтому, с точки зрения практического использования, пользователям рекомендовано использовать черепичную версию реализации GPU.

В отличие от вычислений на процессоре, вычислительная эффективность в архитектуре графического процессора значительно варьируется в зависимости от различных предикатов (например, при одной точности и двойной точности). Более конкретно, вычисления по одинарной точности значительно быстрее, чем вычисления с двойной точностью. Такое поведение присуще графическим процессорам.

Одна из основных причин этого была раскрыта в Руководстве по программированию CUDA [26]. Обработчик pGo GPU мощно определяется числом постоянных перекосов на каждом мультипроцессоре для данного ядра; и количество регистров, используемых ядром, может существенно повлиять на количество резидентных перекосов. Каждая двойная переменная использует два регистра, тогда как каждой переменной float нужен только один регистр. В этом случае для конкретного ядра, как правило, количество регистров, используемых ядром при двойной точности, намного больше, чем число одинарной точности. Таким образом, количество резидентных переходов на многопроцессорном процессоре уменьшается, и вычислительная эффективность уменьшается.

Более того, при различных научных вычислениях, если вычисление, работающее на одинарной точности, достигает требуемой вычислительной точности, то предпочтительной должна быть одинарная точность. Напротив, если требуется высокая вычислительная эффективность, то требуется двойная точность, и также необходим подходящий графический процессор, который может хорошо поддерживать вычисления при двойной точности, например, графический процессор Tesla K40c.

2.2 Параллельные технологии в алгоритме интерполяции Кригинга

Технология OpenCL, акроним для Open Computing Language, изначально была разработана Apple Incorporated, и это, возможно, первая общая программа - стандарт, предназначенный для решения гетерогенных вычислительных сред. Она является бесплатной, кроссплатформенной и имеет хорошую функциональную совместимость. Были написаны ядра OpenCL для всех шагов алгоритма Кригинга, кроме шага 4 (см. раздел 1.3.). Далее

предоставлена реализация подробности относительно шагов алгоритма. В обоих шагах 5 и 8 существует общая операция, которая должна быть выполнена:

Прямым способом реализации этапа 5 является нахождение минимальной и максимальной точки из входного набора данных. Эта операция может быть выполнена с помощью операции уменьшения, используя операторы минимума и максимума. При решении уравнения 1 на шаге 8 выполняется векторное умножение матриц, за которым следует точечный продукт между λ и $Z(x)$., при этом умножающая часть точечного произведения выполняется параллельно, используя ядро OpenCL, посвященное задаче. Затем параллельное сокращение с оператором суммы используется для суммирования всех значений, создающих скалярное значение, которое представляет предсказание Z (x).

Дополнительные сведения об использовании OpenCL для параллельной работы сокращения относятся к [8]. Чтобы определить эмпирическую полувариантность, уравнение 1.10 должно быть решено. Это уравнение включает вычисление разностного квадрата между значениями парных местоположений. Часто каждая пара местоположений имеет уникальное расстояние так, чтобы избежать насыщенной вариограммы и уменьшить число вычислений на шаге 1, вместо генерации записи для каждой пары, они сгруппированы в лотки. Например, вычислить среднюю полувариантность для всех пары образцов в диапазоне от 40 м до 50 м. Эмпирическая полувариантность, следовательно, является графиком усредненных значений вариограммы на оси y и задержки на оси x. Можно сделать замечание относительно параллелизма, который мог бы быть в этой операции. Каждый интервал задержки может обрабатываться независимо от других и, в пределах каждого запаздывания, все парные расчеты по полувариантности также могут выполняться параллельно. Чтобы вычислить среднее расстояние и значения полувариантности, можно использовать операцию суммирования суммы

убывания, оставляя только деление и умножение (Уравнение 1.2), которые должны быть выполнены хостом.

Выполнение параллельного сокращения позволяет избежать ненужной памяти передачи и значительно снижает расходы на хост с дорогостоящими вычислениями. Количество блоков задержки является параметром для алгоритма, и обычно это значение от 10 до 30. Количество задержек соответствует числу точек эмпирического графа полувариантности. Таким образом, установка функции использования ковариационной модели (уравнение 5) в полуварианте является почти недорогой операцией.

Шаг 8 выполняется «на лету» с двумя вложенными циклами, где позиция (x, y) каждого пикселя, который должен быть интерполирован. Значения этих пикселей отправляются в ядро для вычисления ковариации между местоположением и будут интерполированы относительно его расстояния до соседних образцов, используя модель эмпирической вариограммы. Процесс, по сути, вычисляет веса вектора λ. Далее заканчивается решение уравнения 1.5, как описано ранее.

Как указывалось до этого, каждый интервал задержки может обрабатываться независимо. Чтобы максимально использовать ресурсы платформы OpenCL для каждого доступного устройства создается очередь запросов. Каждая очередь команд может использовать поток хоста для одновременной отправки ядер и выполнения операции памяти с соответствующими устройствами. В начале очередь команд назначается каждому потоку OpenMP через схему округления. Это гарантирует, что реализация может масштабироваться через любое количество доступных вычислительных устройств.

В этом исследовании алгоритм ОК используется для выполнения интерполяции ландшафта с конечной целью - нормализовать массив точек. Набор данных, используемый в этом исследовании, состоит из текстовых файлов в простом XYZ формате, где каждая строка представляет собой

местоположение (*x, y, z*) в трехмерном пространстве. Предшествующая для выполнения интерполяции местности результирующая дискретизация данных выполняется посредством нахождения точек с наименьшим *z*-значением в сетке размером 0,25 x 2м. Это будет целью сегментирования точек, которые принадлежат земле. Файл, который был экспортирован программное обеспечение сканера имеет 15 070 181 точек, которые были сэмплированы до 7176 точек используя минимальный фильтр сетки. Это размер данных, используемых в интерполяции Кригинга.

Чтобы интерполировать карьер, точки обрабатываются как измеренные сэмплы в точке (*x, y*) с ответом *z*. Идея состоит в том, чтобы предсказать новые *z*-значения из «выбранных» данных. Эксперименты проводились на машине с 64 AMD Opteron (TM) процессором, 3136 ядра сгруппированы в 4 физических чипа с 16 ГБ оперативной памяти и 4 графических процессора NVIDIA GeForce 1070 TI с 8 ГБ оперативной памяти GDDR5 384-разрядной оперативной памятью. Эта машина использовалась для экспериментов, чтобы показать, что реализация может масштабироваться на нескольких вычислительных устройствах.

Чтобы проверить эффективность алгоритма, мы выполнили обычную Интерполяция Кригинга с использованием сетки 300x300 с 10 задержками. Все эксперименты используют один и тот же входной набор данных из 7176 точек, но с различным числом вычислительных устройств, также каждый эксперимент выполнялся 5 раз, а результаты представляют среднее время автономной работы.

Чтобы показать, что алгоритм OK может получить большую производительность с параллелизмом проведено грубое сравнение с эталонной реализацией с использованием языка C. Используя тот же набор данных, что и другие эксперименты, простой последовательной реализации в C ++, получается ускорение 7.1x. C помощью OpenCL ускорение увеличивается до 8.0x при сравнении общего времени выполнения. Хотя это

может не показаться большим улучшением, при сравнении общего время выполнения, подавляющее большинство времени тратится на ковариационную матрицу (см. таблицу 2.2). Другие эксперименты показывают масштабируемость реализации.

Первое измерение соответствует общей продолжительности работы программы, включая операции ввода / вывода (IO) и времени установки OpenCL. В таблице 2.1 показывается это измерение, изменяющее количество устройств графического процессора, используемых в выполнение.

Таблица 2.1

Общее время выполнения алгоритма Кригинга с использованием нескольких устройств графического процессора

Количество графических процессоров	Время выполнения
1	170,9764
2	149,8212
3	143,5154
4	140,2316

Легко заметить, что достигается плохое значение ускорения даже с четырьмя очень мощными графическими процессорами. В этом исполнении ускорение было всего 1,2 с 4 графических процессоров. Однако это объясняется тем, что подавляющее большинство времени выполнения затрачивается на инвертирование матрицы ковариации. Как объяснялось ранее, Шаг 4 выполняется хост-приложением (ЦП). Эта операция обратной матрицы выполняется Eigen, свободной библиотекой шаблонов C ++ для линейной алгебры и связанные с ней алгоритмы. Несмотря на возможность использования OpenMP как бекенд обработки для Eigen, это все еще самая трудоемкая часть алгоритм. Это показано в таблице 2.2, 69% времени выполнения инвертируя ковариационную матрицу. Хотя это и значительная величина, но выполнение этой процедуры требуется только один раз.

С увеличением количества значений для интерполяции, обратное время матрицы может быть превзойдено к лучшему ускорению.

Таблица 2.2

Таблица, показывающая процент времени выполнения, затраченного шагами алгоритма Кригинга. Остальные 4,54% соответствуют IO и времени инициализации OpenCL

Операция	Процент от времени выполнения
Обработка матрицы расстояний	0,01%
Обработка матрицы ковариации матриц	0,00%
Создание обратной матрицы ковариации	69,00%
Построение полувариограммы	4,97%
Кригинг прогноз	21,47%
Всего	95,46%

Вторая самая дорогая часть алгоритма является этапом 8. Согласно таблице 2.2, он отвечает за 21,47% от времени выполнения, поэтому этот шаг достоин отдельного внимания, чтобы показать, что реализация идет на всех доступных вычислительных устройствах. На этом этапе один поток раздается на каждое вычислительное устройство, работающее независимо друг от друга в очереди команд. На рисунке 2.3 (сверху) видно, что время выполнения прогнозирования значительно уменьшается с увеличением количества графических процессоров. Чтобы показать, что алгоритм может хорошо масштабироваться, представлен график на рисунке 2.3 (снизу) можно добиться квазилинейного ускорения.

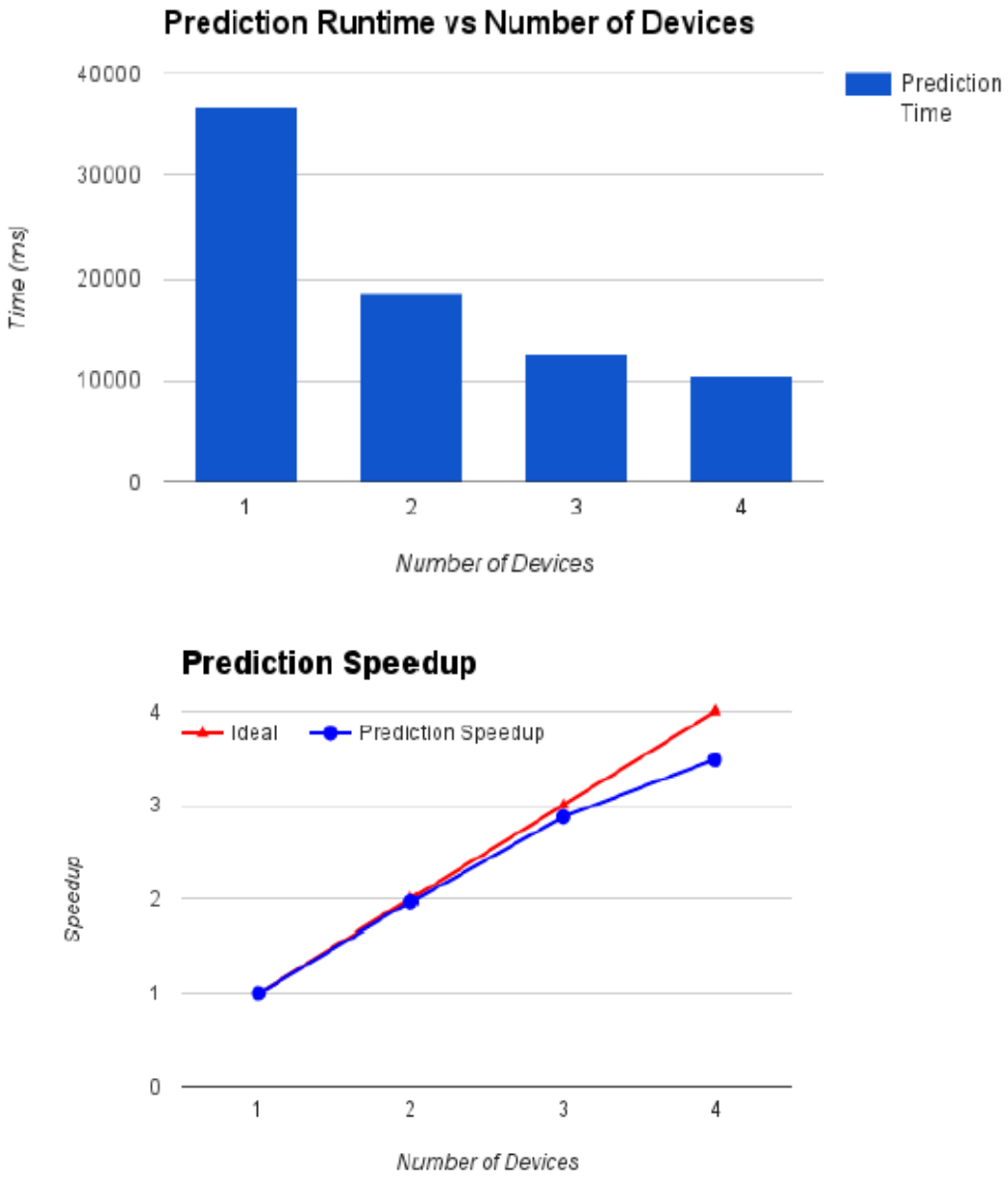


Рисунок 2.3: Кригинг-прогноз (шаг 8): время выполнения в миллисекундах по сравнению с числом устройств с графическим процессором (сверху) и ускорение, когда идеальная линия может использоваться как ссылка (снизу).

Еще одно важное измерение при оценке параллельности – Эффективность. Эффективность - это мера, которая суммирует, насколько эффективно используются ресурсы. Эффективность обычно немного падает в хорошей реализации из-за накладных расходов на параллельную программу.

Тем не тем не менее, в таблице 2.3 показана эффективность с количеством вычислительных устройств для этапа прогнозирования. Понятно, что в исследовании удалось добиться очень хорошей эффективности даже с 4 устройствами.

Таблица 2.3

Эффективность этапа предсказания Кригинга с увеличением количество вычислительных устройств

Количество графических процессоров	Эффективность
1	1,00
2	0,98
3	0,95
4	0,87

2.3 Определение принципов универсального алгоритма пространственной интерполяции Кригинга

Пространственная интерполяция (ПИ) - это метод, используемый для оценки значений свойств в неизвестных точках в пределах области, охватываемой существующими наблюдаемыми точками [1]. Во многих ситуациях ПИ выполняется для огибающих контуров, поэтому данные могут отображаться графически, для вычисления значений свойств для поверхности при заданных точках или анализировать и прогнозировать поверхность тренда. В исследованиях ПИ всегда был мощным инструментом моделирования [2, 3]. Технологические разработки значительно обогатили методы, которые доступны для получения данных, а во многих крупномасштабных инженерных приложениях необходимо обрабатывать

огромные объемы данных с использованием алгоритмов интерполяции. Действительно, ПИ является особенно важным для прогнозирования и представления во многих областях, в том числе географических информационных системах и дистанционного зондирования [4 – 6], геологии [7], добычи полезных ископаемых [8], гидрогеологии [9], исследования почв [10], геофизики [11], океанографии [12], метеорологии [13], экологии и экологических исследований [14, 15].

Несколько разных типов методов классификации используются процедурами ПИ, например, точечная область, глобально-локальная и точно-приближенная интерполяция [16]. Многие методы существуют как для глобальной, так и для локальной интерполяции. Анализ поверхности тренда и ряды Фурье являются примерами глобальных методов, тогда как аппроксимальные, Кригинг и B-сплайны являются локальными методами. В частности, алгоритм Кригинга ПИ является типичным локальным алгоритмом интерполяции. Алгоритм универсальной интерполяции Кригинга является типом линейного и объективного оптимального алгоритма Кригинга ПИ, который широко используется во многих научных и технических приложениях. Однако во многих приложениях возникают проблемы с высокой производительностью при использовании алгоритма последовательного универсального Кригинга, поскольку вычислительная стоимость экспоненциально возрастает с размером входных данных [17, 18].

Чтобы ускорить процесс и получить более высокую производительность, в последние десятилетия исследователи разработали различные методы для реализации параллельных алгоритмов ПИ, которые ориентированы на высокопроизводительные вычислительные системы, например, MasPar [19], Cray T3D [20], параллельные кластеры [21], многоядерные платформы [22] и срединные вычислительные среды [23]. В частности, было проведено несколько исследований разработки алгоритма параллельной интерполяции Кригинга. Например, [24] использовали

выделенный высокопроизводительный компьютер для реализации параллельного алгоритма Кригинга, который значительно сокращает время процессора. Однако этот метод достаточно эффективен, и для ускорения обработки требуется высокая стандартная аппаратная конфигурация. Также [25] реализовали параллельный алгоритм Кригинга, используя параллельную аппроксимирующую схему интерфейса передачи сообщений на товарном кластере, где реализация достигла удовлетворительной производительности и хорошей эффективности. Однако требование обработки в реальном времени и быстрый рост размера данных требует еще больше вычислительных узлов, что неизбежно увеличивает как затраты на аппаратное обеспечение, так и затраты на техническое обслуживание, а также требует высокого потребления энергии [26]. В некоторых исследованиях также были разработаны параллельные алгоритмы Кригинга с использованием метода многоядерного параллелизма. Например, [27] разработан параллельный алгоритм Кригинга на основе нескольких ядер, но были проблемы с эффективностью, когда многоядерный параллелизм применялся к специализированным научным приложениям, например, к большой обработке данных из-за медленного доступа к системной памяти.

В последнее время из-за быстрого увеличения вычислительной мощности ускорителей, таких как графические процессоры (GPU) и Intel Xeon Phi (Intel Many Integrated Core architecture (MIC)), использование ускорителей для большой обработки данных стала обостренной исследовательской темой в различных областях. Многие исследования чипов были проведены в области геонаук [28 – 31]. В частности, были проведены исследования универсальных алгоритмов Кригинга; например, реализован параллельный универсальный алгоритм Кригинга с использованием архитектуры унифицированных устройств NVIDIA Compute Unified Device Architecture (CUDA) на платформе GPU [32]. Однако на платформе Intel MIC было мало исследований, поскольку MIC является относительно новой технологией ускорителей. Предыдущие

исследования на основе MIC были сосредоточены главным образом на сравнении с GPU или аспектами программирования платформы вместо разработки или реализации параллельного алгоритма или приложений ПИ. Например, [33] сравнили архитектуру и производительность GPU общего назначения (GPGPU) с Intel MIC и продемонстрировали преимущества MIC. (34) Описаны меры по устранению узких мест в емкости памяти, пропускной способности сети, т.е. улучшена расширяемость потоков параллельного программирования на платформе MIC. Однако их реализация не была переносимой на разных платформах, потому что разные вычислительные платформы, то есть GPU и MIC, требуют разных моделей и инструментов программирования.

Неоднородная вычислительная система представляет собой вычислительную систему, которая может интегрировать компоненты ускорения GPU и Intel Xeon Phi в обычные вычислительные системы для реализации вычислительных задач вместе с процессором. Гетерогенные вычисления интегрируют каждую гетерогенную платформу асинхронно, используя отдельные ресурсы для вычисления или планирования задач, тем самым максимизируя общую эффективность вычислительной системы, назначая задачи на основе соображений возможностей каждой вычислительной системы [35].

Гетерогенные вычисления играют все более важную роль в обработке больших данных, и быстрый переход на использование гетерогенных вычислений для интерполяции крупномасштабных пространственных данных с использованием улучшенных алгоритмов. Такой подход может потенциально обеспечить высокую производительность на вычислительных платформах сопроцессора со скоростями в 10 раз больше, а также эффективно избежать вышеупомянутых проблем, которые встречаются в традиционном кластере только для CPU. Также желательно иметь оптимизированную кроссплатформенную реализацию алгоритма параллельного универсального

Кригинга, который выполняется на разных гетерогенных платформах. Насколько известно, в нескольких исследованиях было рассмотрено применение гетерогенных вычислений в области геологических наук.

Универсальный алгоритм Кригинга - это тип линейного несмещенного оптимального алгоритма ПИ. В отличие от других широко используемых алгоритмов ПИ, таких как ячейки Вороного и метод взвешивания обратного расстояния (36), который рассматривает пространственную корреляцию между точками, которые должны быть интерполированы и их соседних точек, а также дает ошибку оценки. Универсальный алгоритм Кригинга обеспечивает более точные результаты интерполяции и широко применяется в области геологической интерполяции. Принцип алгоритма выражается уравнением (2.1):

$$\hat{Z}(x_0) = \sum_{i=1}^{n} \lambda_i Z(x_i) \tag{2.1}$$

где $\hat{Z}(x_0)$ - значение в точке, которая должна быть интерполирована, и представляет собой взвешенный коэффициент точки i с измеренным значением $Z(x_i)$. Когда ожидание случайной величины $Z(x)$ является переменной в интересующей области, имеем,

$$E\left[Z(x)\right] = m(x) \tag{2.2}$$

В этом случае для интерполяции требуется универсальный алгоритм Кригинга. В уравнении (2.2) $m(x)$ - функция дрейфа, которая может быть представлена следующим образом:

$$m(x) = \sum_{l=0}^{k} u_l f_l(x) \tag{2.3}$$

где f_l (x) является известным уравнением, а u_l - неизвестный параметр. Чтобы обеспечить оцененным и оценочным значениям, насколько это возможно, универсальную интерполяцию Кригинга алгоритмы должны обладать следующими условиями:

• Условие 1: Ожидаемое значение разности между оцененным и оценочным значениями равно нулю, $E\left[\hat{Z}(x_0)-Z(x_0)\right]\equiv 0$, что приводит к уравнению: $\sum_{i=1}^{N}\lambda_i=1$. Объединив это с уравнениями (2.1) и (2.3), универсальный алгоритм Кригинга может быть выражен уравнением (2.4).

$$\sum_{i=1}^{N}\lambda_i f_l(x_i)=f_l(x_0),\quad (l=0,1,...,k). \tag{2.4}$$

• Условие 2: Оценочные условия минимальной дисперсии. Чтобы минимизировать дисперсию между оцененным и измеренным значением, универсальный алгоритм интерполяции Кригинга должен удовлетворять следующему условию.

$$\sigma_E^2=E\left[\hat{Z}(x_0)-Z(x_0)\right]^2 \tag{2.5}$$

Используя уравнения (2.1) и (2.5), получим уравнение (2.6).

$$\sigma_E^2(x)=-\sum_{i=1}^{n}\sum_{j=1}^{n}\lambda_i\lambda_j\gamma(x_i,x_j)+2\sum_{i=1}^{n}\lambda_i\gamma(x_i,x_0), \tag{2.6}$$

где γ означает функцию вариации. Используя метод множителя Лагранжа и уравнения (2.4) и (2.6), можно получить целевую функцию, как показано в уравнении (2.7).

$$F=\sigma_E^2-2\sum_{l=0}^{k}u_l\left[\sum_{i=1}^{n}\lambda_i f_i(x_i)-f_i(x_0)\right] \tag{2.7}$$

Беря частные производные λ_i и u_l делает их равными нулю, получим уравнение наборf универсального алгоритма Кригинга. Оценочные значения интересующих точек получаются путем решения соответствующих уравнений в матричной форме, которая включает в себя ряд операций продукта для измеренного значения и соответствующего веса каждой точки.

Неоднородная вычислительная система включает в себя различные процессоры с различными функциями или производительностью, которые соединяются через определенную структуру межсоединений. В общем, они

содержат один или несколько микропроцессоров общего назначения и специальные ускоренные процессоры. В настоящее время наиболее широко используемые гетерогенные вычислительные платформы содержат как CPU, так и GPU. NVIDIA выпустила первый GPU общего назначения в 1999 году, который был специализированным сопроцессором, предназначенным для решения проблемы сложных вычислений [37]. Благодаря параллельной многоядерной структуре и более высокой пропускной способности доступа к памяти, GPU предлагает более высокие пиковые вычислительные мощности и более высокую пропускную способность памяти, чем современный CPU. При поддержке CUDA и OpenCL он постепенно стал типом процессора общего назначения. 18 июня 2012 года Intel Cooperation представила платформу MIC (Many Integrated Core Architecture), которая представляет собой многоядерную архитектуру, которая отличается от TGA графического процессора [38]. MIC - это сопроцессор с кратной архитектурой x86. Эти ядра интегрированы в единый вычислительный узел как аппаратные периферийные устройства сопроцессора, и они работают вместе с CPU. MIC совместим с набором инструкций процессора x86 и наборами инструкций с несколькими инструкциями для нескольких команд, что может уменьшить сложность трансплантации из традиционного кластера в архитектуру MIC. Кроме того, он поддерживает сложные, но гибкие стратегии программирования. Таким образом, MIC привел разработку приложений к новому периоду. Комбинация CPU и MIC предоставляет новую возможность для гетерогенных вычислений.

OpenCL - это первый общий стандарт параллельного программирования для гетерогенных вычислений. Он был первоначально разработан Apple Incorporated, он бесплатный, кроссплатформенный, с хорошей совместимостью [39]. OpenCL обеспечивает унифицированную среду программирования для разработчиков программного обеспечения. Это облегчает разработку программного обеспечения для высокопроизводительных вычислительных серверов, настольных

вычислительных систем и ручных устройств, а также приложений в многоядерных процессорах (CPU / MIC), графических процессорах и цифровых сигнальных процессорах. OpenCL имеет множество областей применения, и многообещающее будущее на потребительском рынке.

OpenCL - это реализация, а не развивающийся язык. Он предоставляет C-подобный язык программирования (на основе C99) для разработки функции ядра, которая может работать на разных устройствах OpenCL и группе интерфейсов прикладного программирования (API), которые могут определять и контролировать гетерогенные платформы. OpenCL предоставляет два параллельных вычислительных механизма [35], то есть:

1. на основе сегментации задачи;

2. на основе сегментации данных.

Согласно официальному руководству по разработке OpenCL, построенные алгоритмы / приложения могут работать на различных устройствах. Кроме того, OpenCL поддерживает реализацию нескольких уровней параллелизма, и каждый уровень параллелизма может эффективно отображаться на жестких дисках на однородных или гетерогенных архитектурах. Во время разработки и разработки параллельного алгоритма, который соответствует спецификации OpenCL, важно следовать четырем предписанным моделям, т.е. модели платформы, режиму реализации l, модели памяти и модели программирования. Модель платформы представляет собой высокоуровневое описание гетерогенной вычислительной системы с абстракциями на аппаратной основе системы. Модель реализации описывает, как ядра запускаются в OpenCL и как ядро взаимодействует с концом хоста. Модель памяти представляет собой абстракцию базового пространства памяти, которое описывает область памяти, установленную в OpenCL, и определяет взаимодействия различных пространств памяти во время вычислений. Модель программирования представляет собой высокоуровневую абстракцию приложений, реализованных разработчиками

программ, которая определяет сопоставления программы OpenCL с хостом и процессором с пространствами памяти.

Параллельный алгоритм будет обеспечивать лучшую производительность. Соответствующие типы памяти учитываются при программировании OpenCL. OpenCL определяет четырехуровневую иерархию памяти для вычислительного устройства: глобальную, постоянную, локальную и частную память. Глобальная память может совместно использоваться всеми элементами обработки, но она является высокой задержкой доступа. Постоянная память также видна всем вычислительным устройствам на устройстве, где она является частью глобальной памяти. Любой элемент постоянной памяти доступен одновременно всем рабочим элементам. Локальная память хранится в вычислительном блоке, и она обычно реализуется на чипе, где она разделяется всеми рабочими элементами внутри рабочей группы. Рабочая группа имеет низкую задержку доступа, но ее емкость ограничена. Частная память принадлежит рабочему элементу, и она обычно реализуется на чипе в регистрах.

OpenCL получила широкую поддержку от крупных производителей сопроцессоров и имеет преимущества открытого доступа и кроссплатформенной работы [39], поэтому в этом исследовании использовался OpenCL как инструмент разработки кода для реализации алгоритма параллельного универсального Кригинга на разных гетерогенных вычислительных платформах.

В этой работе представлена параллельная реализация Обычного Кригинг алгоритма интерполяции, который использует преимущества гетерогенных вычислений, используя среду OpenCL и Открытые параллельные расширения OpenMP. В исследовании удалось добиться квазилинейного ускорения на втором наиболее трудоемком этапе применения алгоритма Кригинга – шаге прогнозирования. Также показано, что реализация

имеет высокую эффективность (87%) даже при увеличении количества вычислительных устройств

3 РАЗРАБОТКА АЛГОРИТМА ИНТЕРПОЛЯЦИИ НА ОСНОВЕ OPENCL

3.1 Определение путей интеграции с OpenCL для проведения гетерогенных вычислений

Основная задача, связанная с реализацией последовательного алгоритма, заключается в выборе подходящей модели функции изменения пространства, а также разработке кода для этой модели вариационной функции. В этом исследовании вычисляется оценочное значение для каждой точки с использованием подхода поиска соседних точек.

Модель вариационной функции может быть разделена на три категории в геостатистике:

1. Модель с порогом [40], которая включает в себя сферическую модель, индексную модель и гауссову модель;

2. Модель без порога [2], которая включает в себя функцию мощности и линейную модель;

3. Модель эффекта полости [41].

В исследовании используется сферическая модель, которая часто используется в геостатистике, как функция вариограммы универсального алгоритма Кригинга. Сферическая модель может быть выражена уравнением (3.1):

$$\gamma(h) = \begin{cases} c_0 + c\left[\frac{3h}{2a} - (\frac{h}{2a})^3\right], & 0 < h \le a \\ c_0 + c, & h > a \end{cases}, \quad (3.1)$$

где c_0 - эффект самородка, c - частичный порог полувариограммной модели, а - диапазон влияния.

Следуя принципу универсального алгоритма Кригинга, используя выбранную функцию вариации, серийный алгоритм может быть реализован в основном на трех компонентах: (1) модуль манипуляции с файлами (FMM); (2) модуль поиска соседних точек (APSM); и (3) универсальный криптографический функциональный модуль интерполяции (UKIFM). Когда результаты интерполяции читаются, этот модуль FMM также записывает данные как выходные данные. Эти функции реализованы с использованием библиотеки с пространственными данными с открытым исходным кодом, GDAL (Библиотека абстракции геопространственных данных) [16]. APSM фокусируется главным образом на вычислении плоских координат (x, y) неизвестных точек, соответствующих диапазону координат известных точек, используя фиксированный шаг. Процедура поиска соседних точек каждой неизвестной точки использует алгоритм ближайшего соседства [42], которая ищет k соседей точки поиска. Модуль UKIFM - это числовое ядро, используемое для интерполяции (см. Рисунок 3.1) .

Рисунок 3.1. Модули в алгоритме последовательного универсального Кригинга.

Все модули используют общие глобальные переменные для завершения взаимодействия данных и обработки данных. Для последовательного алгоритма необходимы два глобальных массива: dp[n_known] и ip[n_unknown] в FMM, где первая переменная для известных точек и последняя - для точек, где происходит интерполяция. Таким образом, массив dp[n_known] инициализируется для хранения данных, извлеченных из файла формы файла, и массив ip[n_unknown] заполняется формацией координат плоскости, и (*x, y*), для каждой неизвестной точки путем поиска подходящих точек с фиксированной длиной шага в глобальном масштабе для всего изображения. В процессе поиска, дополнительный массив называется near_points [n_nearby] вводится для хранения [n_nearby] соседней точки для каждой неизвестной точки. UKIFM использует координату плоскости (*x, y*) для координаты неизвестной точки и координаты их соответствующих смежных точек для вычисления оценочных значений, которые выводятся вместе с координатами (*x , y, z*) для этих неизвестных точек.

В частности, последовательный алгоритм может быть подробно выражен следующими четырьмя шагами:

Шаг (1): Прочитать информацию данных, то есть, (*x, y, z*), трехмерные координаты известных точек из исходных файлов.

Шаг (2): Вычислить плоские координаты неизвестных точек в соответствии с диапазоном координат известных точек. На основе известных точек выберите точки, которые необходимо интерполировать с заданным интервалом пробега, а затем вычислить их соответствующие плоские координаты (*x, y*).

Шаг (3): Создать дерево k-d, используя информацию трехмерных координат для известных точек, и затем найти соседние точки (среди известных точек) для каждой неизвестной точки в соответствии с алгоритмом.

Шаг (4): Передать информацию координат (x, y) для неизвестных точек и трехмерные координаты (x, y, z) их соседних точек в UKIFM для вычисления оценочных значений неизвестных точек.

В частности, шаги (2) и этап (3) используются для предоставления известных точек и информации о плоских координатах для точек, которые необходимо вставить, причем шаг (4) является основным компонентом расчета серийного универсального алгоритма Кригинга. Фактически реализация этапа (4) сложна и его можно разделить на семь подэтапов (рисунок 3.2), следующим образом.

Рисунок 3.2. Основные этапы универсальной кригинговой интерполяционной функции.

1 подэтап (Sub-step a): Вычислить расстояние между точкой, которая должна быть интерполирована, и ее смежными точками (известными точками, где сумма равна n),

2 подэтап (Sub-step b): Сортировка значений расстояния, полученных в порядке возрастания.

3 подэтап (Sub-step c): Разделить отсортированные значения на k групп.

4 подэтап (Sub-step d): Рассчитать среднее расстояние h в каждой группе в соответствии с их значениями.

Оцененные параметры вариационной функции вычисляются с использованием уравнения (3.8). В соответствии; к выбранному режиму функции изменения, при подгонке, проводимом для определения вариационной функции и коэффициента регрессии.

5 подэтап (Sub-step e): указать значения расстояния от места деления и точки деления для вариационной функции для построения матрицы коэффициентов.

6 подэтап (Sub-step f): решении матрицы коэффициентов в подэтапе 5.

7 подэтап (Sub-step g): рассчитать оценочное значение неизвестной точки пока все точки были обработаны.

После выполнения анализа потребления времени с использованием алгоритма последовательного универсального Кригинга обнаружилось, что для шага 4, то есть модуля функции интерполяции Кригинга, требуется 85,2%-97,6% от общего количества прошедшего времени для разных размеров набора данных. Поэтому, чтобы полностью ускорить последовательный универсальный алгоритм Кригинга для получения хорошей производительности, т. е. шаг 4, требует полного рассмотрения. Расчеты неизвестных точек не зависят друг от друга, что облегчает процедуру распараллеливания. Таким образом, в исследовании основное внимание уделялось реализации параллелизма для этого шага. Следует отметить, что некоторые конкретные вопросы применения, такие как оценка параметров от

параметров вариограммы не рассматриваются в параллельной реализации. Таким образом, параметры, используемые в предлагаемом параллельном алгоритме, идентифицируются с помощью интерполяции.

Полностью рассмотреть такие аспекты, как разные платформы, количества ускоренных процессоров и ограничений системной памяти, мы предлагаем общую структуру параллельного алгоритма, показанного на рисунке 3.

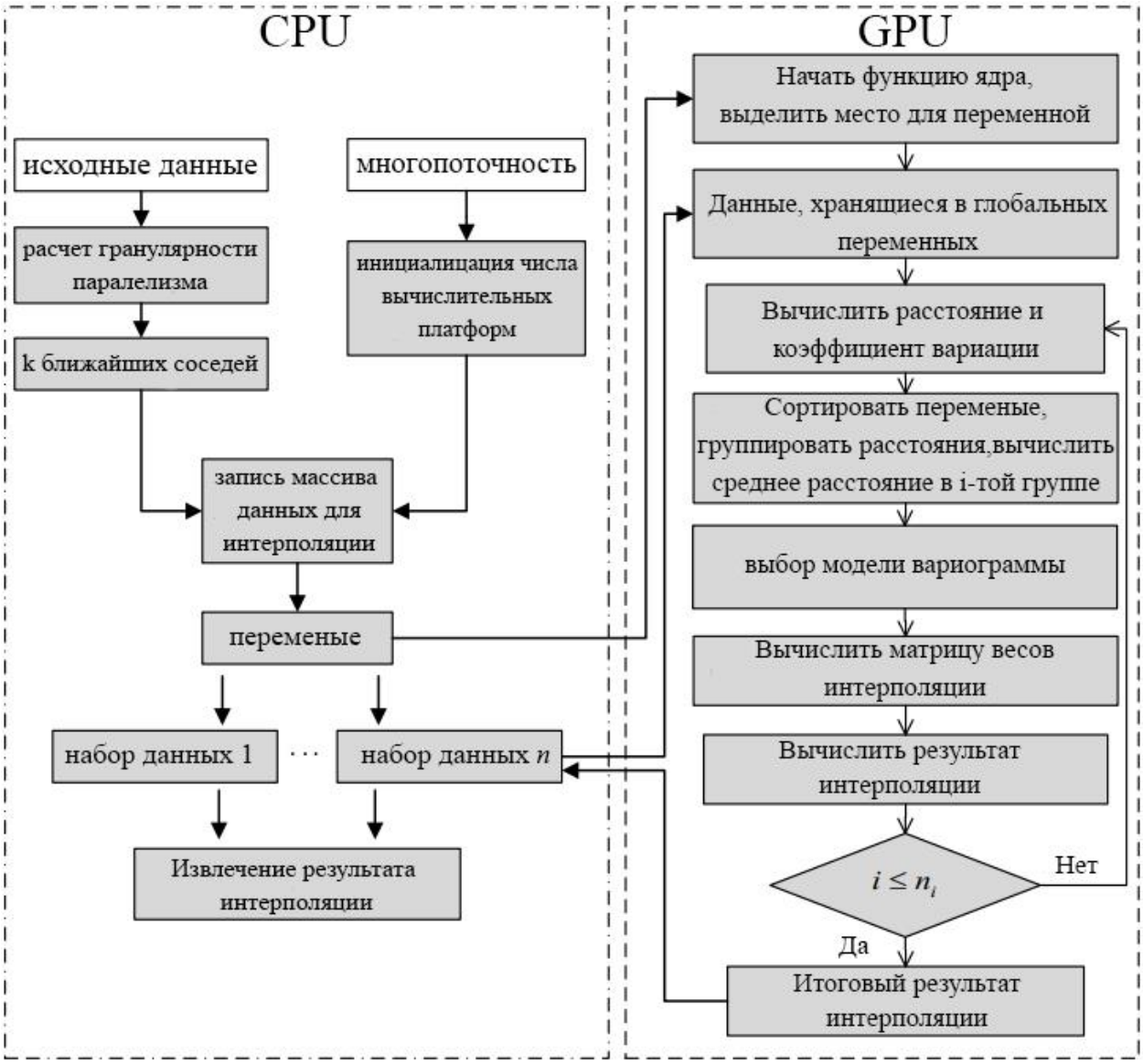


Рисунок 3.3. Общие рамки реализации алгоритма параллельного универсального алгоритма Кригинга.

Согласно приведенному выше анализу, ясно, что шаг 4 должен быть распараллелен. Однако при разработке и внедрении соответствующего эффективного параллельного алгоритма также необходимо учитывать все аспекты, такие как структура данных, передача данных между хостом и устройствами, детализацию разделов задач и балансировку нагрузки для нескольких типов взаимодействующего оборудования [43]. Некоторые из этих вопросов независимы от других, тогда как существуют особые зависимости от других.

Согласно рисунку 3.3, структуру можно разделить на две части: хост и устройство. Очевидно, что основной расчет выполняется в устройстве. Хост выполняет только контрольные задачи, такие как распределение данных и сбор результатов. Хост и устройство привязаны к некоторым общим переменным. Чтобы разработать параллельный алгоритм с OpenCL, очень важно разработать и внедрить модули, которые совместимы с каркасом OpenCL. При разработке параллельного универсального алгоритма Кригинга для высокопроизводительного вычислительного оборудования основное внимание будет уделено тому, как сделать в шаге 4 точку доступа полностью в параллельной структуре OpenCL. Эта проблема в основном зависит от конкретного сочетания проектирования и реализации четырех различных моделей программирования:

1. модель платформы;
2. модель исполнения;
3. модель памяти;
4. модель программирования

Эти модели программирования дополняют друг друга, и они интегрированы в общую работу. Таким образом, при разработке и реализации конкретной модели могут быть задействованы другие модели. Согласно этому правилу, в дальнейшем подробно описывается четыре модели и предлагаемая структура.

Модель платформы OpenCL, при использовании гетерогенной платформы блокирует базовую реализацию оборудования, которая может использоваться разработчиками только в форме устройства, поэтому необходимо разработать дополнительный шаблон совместной работы между хостом и несколькими устройствами. Во-первых, функция создания потоков создаст подходящие потоки в соответствии с их количеством. Затем последующие шаги включают в себя использование платформ, выбор оборудования и создание буферов. Таким образом, устройства могут выполнять все вычислительные задачи совместно, а соответствующая схема реализации подхода показана на рисунке 3.4.

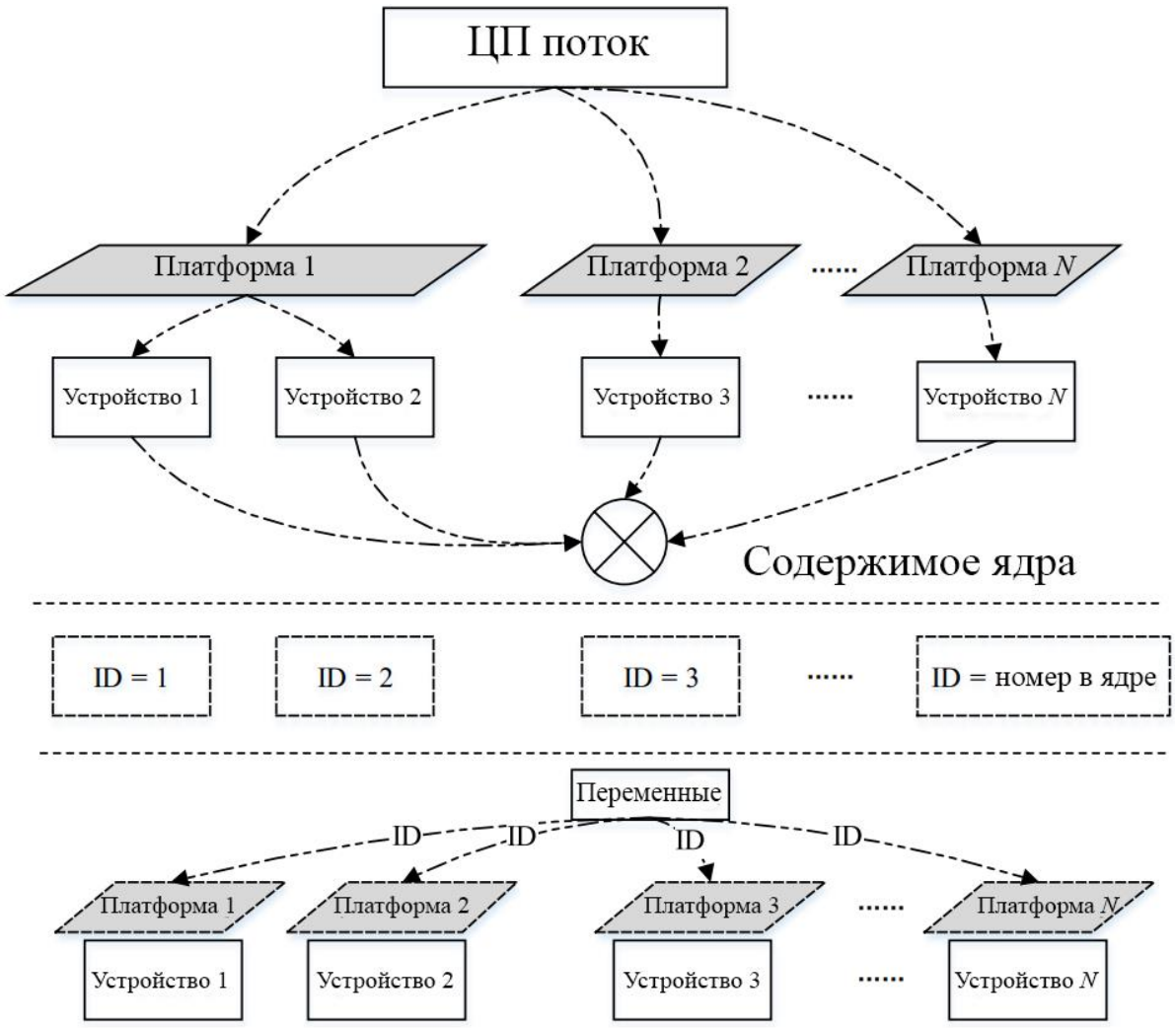


Рисунок 3.4. Внедрение платформы.

На рисунке 3.4, основной поток делится информацией с дочерними потоками. Информация включает в себя информацию назначения задачи, информацию о платформе, указывающую на текущее оборудование и переменную информацию для времени, разделяемого на потоки. Основной поток сначала получает количество доступных платформ и устройств на каждой платформе, прежде чем вычислять общее количество всех доступных устройств. Затем создается равное число потоков, а некоторые общие переменные, которые преобразуют информацию с помощью дочерних потоков. Таким образом, инициализируется идентификатор устройства, записывается в переменную, совместно используемую каждой единицей оборудования. Впоследствии дочерний поток выполняет операции по воссозданию платформ, и они могут получить соответствующую информацию об устройстве, предоставленную основными потоками.

В следующих экспериментах использовались две гетерогенные вычислительные платформы: платформа GPU, оборудованная двумя картами графического процессора и платформой Intel Xeon Phi, оборудованная тремя MIC-картами. В целом аналогичная процедура выполнялась и для реализации платформ. На вычислительных платформах GPU две карты графического процессора рассматривались как устройства OpenCL. В первую очередь основной процесс был создан в конце узла, где его задачей было управлять платформами OpenCL и устройствами OpenCL. Затем были созданы два подпроцесса после того, как основной процесс нашел эти два устройства OpenCL, которые в основном работают после инициализации устройства OpenCL. Процесс инициализации состоял главным образом из создания контекста, создания буфера, создания очереди команд, создания программы и установки параметров ядра. Наконец, параллельный алгоритм отправил ядро и соответствующие данные соответствующим устройствам, как описано выше.

3.2 Реализация алгоритма последовательного универсального Кригинга. Разработка кода для стандарта разработки приложений OpenCL

Модель выполнения OpenCL определяет способ выполнения функции ядра, которое работает с поддержкой OpenCL. Приложение OpenCL состоит из двух частей: главная вычислительная программа и одно, либо несколько ядер. Тем не менее, модель исполнения OpenCL для программы не определяет детали схемы хост-машины. Таким образом, имея несколько устройств, необходимо выполнить тщательную работу по распределению нагрузки и совместной разработки задач. В этом исследовании функция ядра была преобразована из части универсальной функции Кригинга, которая должна быть разобщена. При использовании метода распараллеливания данных, хост-устройство распределяет функцию ядра на каждое устройство. Это ядро было выполнено с соответствующими потоками, но на самом деле эти потоки были созданы хостом (см. рис. 3.5). Следует отметить, что вычисления для каждой неизвестной точки не зависят от каждой из них, поэтому нет необходимости рассматривать связь между ядрами или 1 потоком.

Как показано на рисунке 3.5, расчет может быть выполнен за один раз для небольшого набора данных. Процесс включает операции, такие как заполнение данных, вычисление и получение результатов. Учитывая, что необходимо вычислить достаточно большие наборы данных, полезно использовать операцию цикла для выполнения задачи из-за ограничений памяти. В этом случае данные сначала делятся на n групп в соответствии с объемом памяти и количеством CU (показано слева на рисунке 3.5). Каждая группа, например, группа i, преобразуется в устройства во время итерации, а затем разлагается на p частей, которые можно вычислить на один CU.

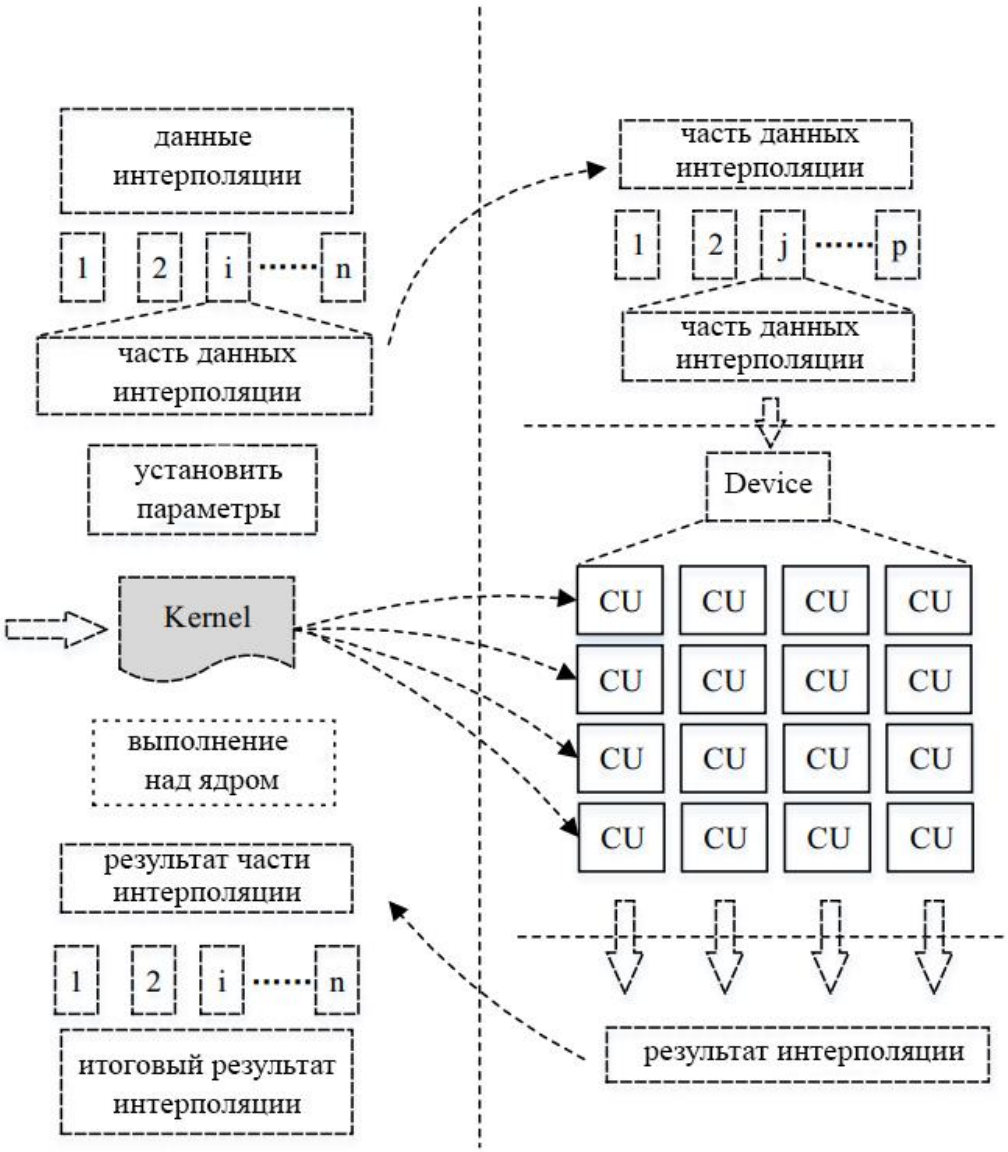


Рисунок 3.5. Обзор, показывающий выполнение параллельной программы OpenCL.

Таким образом, эти операции могут повторяться много раз в течение одной итерации для обработки больших объемов данных. Кроме того, хост должен сформулировать распределение данных по нескольким устройствам, чтобы обеспечить упорядоченную обработку данных.

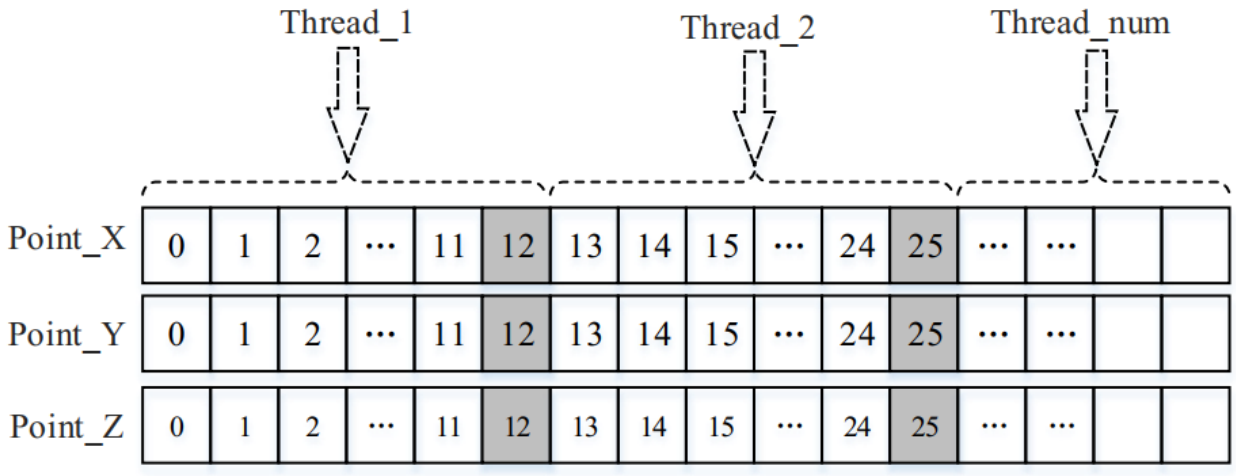
Модель памяти показывает, как OpenCL делит память между хостом и устройствами для взаимодействия данных там, где используется объект

памяти для завершения передачи данных. В этом исследовании проблема реализации памяти имеет два аспекта:

(1) режим памяти; реализация на хосте;

(2) реализация режима памяти на вычислительных устройствах;

В хосте трехмерная информационная координатная информация для точек данных интерполяции сначала считывается в системную память. Размер массива определяется количеством точек *n_points*, которое предварительно считывается из форматированного шепфайла. Затем число единиц, которые должны быть интерполированы, создается в соответствии с количеством параллельных потоков и размером памяти, которые могут использоваться операционной системой. Размер интерполяции единиц *search_n+1*, где *search_n* блоки хранятся с информационными данными *search_n* соседних точек для последней точки, которую необходимо интерполировать. Организация структуры данных для единицы интерполяции в каждом потоке показанных на рисунке 3.6.



(Point_X [i], Point_Y [i], Point_Z [i]) делает координаты точки (x, y, z)

Рисунок 3.6. Структура организации данных.

Чтобы не тратить время на распределение и освобождение памяти, а также что бы решить проблему нехватки памяти в вычислительных устройствах, память должна контролироваться некоторым дежурным механизмом при интерполяции. В этом исследовании максимальный предел равен *V*, который определяется путем контроля процента памяти или представления определенного размера. Код оптимизации описывается следующим образом в листинге 3.1.

Листинг 3.1. Оптимизация памяти

```
MAX_MEM_FOR_DATA_STRUCT = 1 * 1024 * 1024;
     // Память установлена в 1 МБ в методе расчета по
     //умолчанию
MAX_MEM _FOR_DATA_STRUCT = s_info.totalram * Ration;
     //Использовать метод управления процентом использования
     //памяти
```

Конец листинга.

Наконец, буферы создаются соответствующими потоками в каждом устройстве. Таким образом, операции, такие как передача блока данных интерполяции в буфер и отправка результатов, могут обрабатываться непрерывно. Передача данных между ведущим и оборудованием завершается в течение нескольких итераций времени.

Реализация режима памяти на вычислительных устройствах – модулях данных, которые необходимо обработать, может совместно использоваться в виде глобальной памяти между функцией ядра и функцией вызова ядра. Функция, расположенная на вычислительных устройствах, определяет некоторые переменные в виде частной памяти для завершения обработки данных в устройствах.

Большое количество данных связано с массовым приложением обработки данных, поскольку большое количество потоков выполняется на вычислительных устройствах, поэтому общая память и частная память становятся ценными, поскольку эти ресурсы обычно ограничены аппаратным

обеспечением. Поэтому используется метод, который непосредственно читает и записывает глобальную переменную несколько раз, вместо того, чтобы загружать все данные интерполяции из глобальной переменной в устройства каждый раз. Этот метод оптимизации в некоторой степени повлияет на скорость обработки, но увеличивает количество потоков, которые могут быть запущены на устройствах одновременно, тем самым улучшая общую производительность.

Модель программирования определяет операционную стратегию, которая позволяет распараллеливать алгоритм и запускаться на устройствах OpenCL. В OpenCL существуют две разные модели программирования: параллельные задачи и параллельные данные [39]. Для этого алгоритма важно использовать наиболее подходящий режим.

При реализации параллельного универсального алгоритма Кригинга, ключевой шаг — это распараллеливание шага 4, который реализуется универсальной функцией Кригинга. Эта функция также включает в себя семь подэтапов, которые представляют операции, такие как сортировка, группировка и вычисление инверсии матрицы. Согласно анализам времени, затраченного на каждом подэтапе, используя профессиональный анализатор производительности, результаты для различных размеров набора данных с различными настройками проиллюстрированы на рисунке 3.7.

Рисунок 3.7 показывает, что процедура вычисления инверсии матрицы составляет 61,9% -72,9% от времени выполнения для последовательного алгоритма, который является наивысшей пропорцией среди всех подэтапов и не содержит других шагов, таких как чтение или поиск данных смежных точек. Кроме того, существуют зависимости между другими этапами, поэтому режим распараллеливания уровня задачи не подходит для этого алгоритма. Напротив, данные, которые необходимо обработать, могут выполняться параллельно с использованием сконструированной структуры, где каждый

рабочий элемент не требует взаимодействия с другими данными, и нет зависимостей данных.

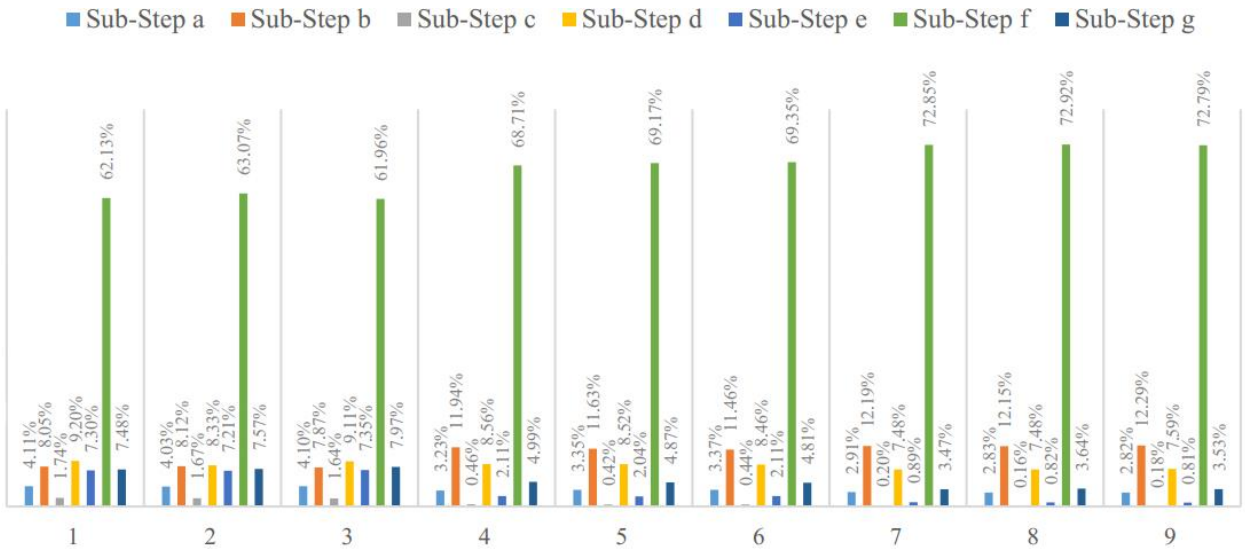


Рисунок 3.7. Потребление времени каждым подэтапом в универсальной функции кригинга .

Таким образом, это наиболее подходящий уровень для распараллеливания данных. Такая форма распараллеливании данных для реализации универсального алгоритма Кригинга показана на рисунке 3.3

Гранулярность разделов задач и проблема балансировки нагрузки должны решаться при наличии нескольких вычислительных устройств. Основной поток должен распределить общую задачу на несколько устройств для обработки, прежде чем собирать и комбинировать результаты вычислений для получения окончательных результатов ПИ. На основе экспериментальной проверки параллельный алгоритм использует стратегию, которая делит конкретный объём памяти в соответствии со средним количеством оборудования, где он учитывает максимальное количество точек, которые память может допускать вставку в качестве детализации параллелизма. Кроме того, используется механизм блокировки для реализации динамической балансировки нагрузки вычисления расписания распределения задач.

Детальная реализация стратегии планирования балансировки нагрузки проиллюстрирована на рисунке 3.8.

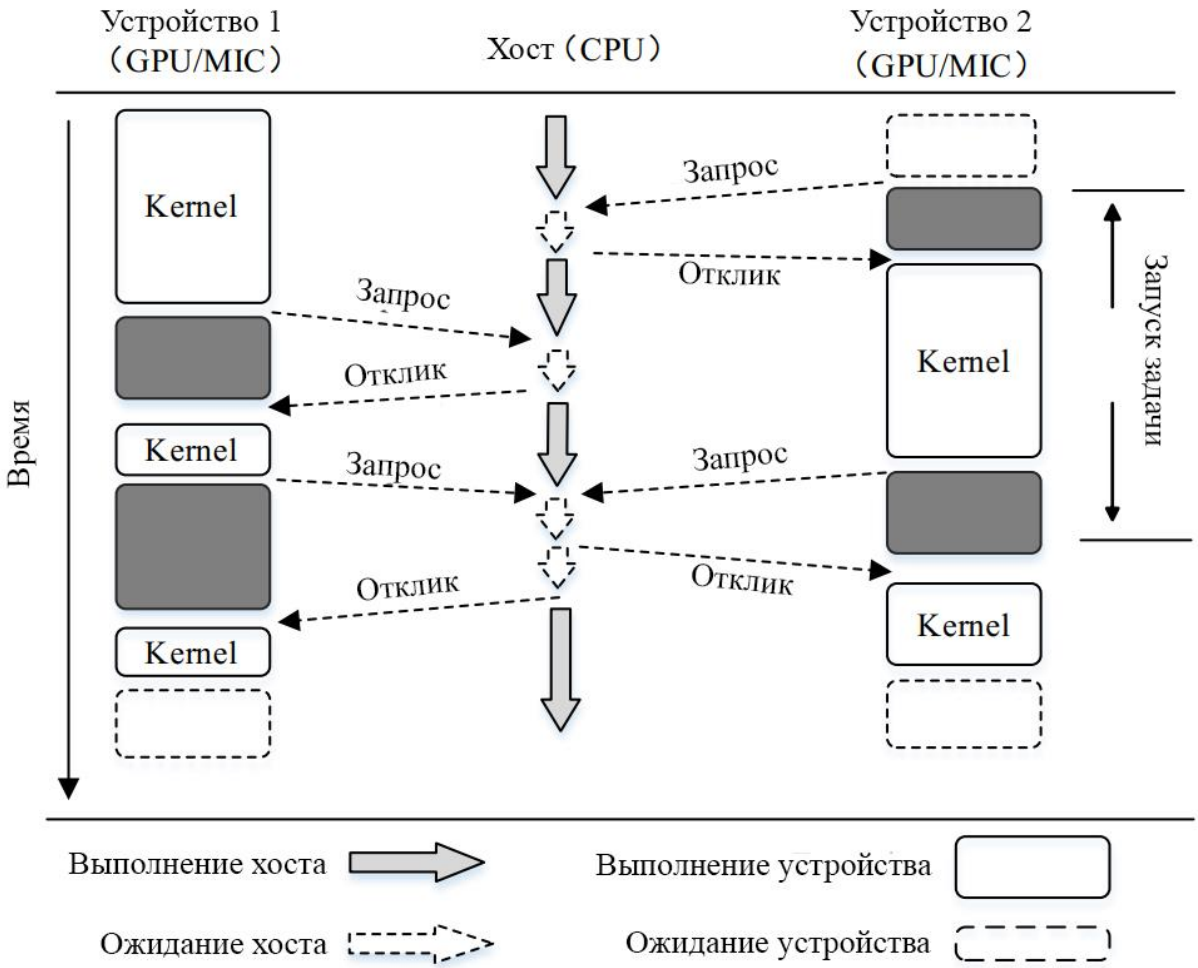


Рисунок 3.8. Стратегия балансировки нагрузки между несколькими устройствами.

3.3 Апробация программно – алгоритмического обеспечения систем недропользования

При проведении экспериментов были рассмотрены следующие особенности:

- Разница в производительности на разных гетерогенных вычислительных платформах также отражает кроссплатформенную возможность параллельного алгоритма.
- Каждая гетерогенная вычислительная платформа была оснащена несколькими картами ускорения, то были две карты GPU и три встроеные карты Intel Xeon Phi, поэтому, возможно, были различия в использовании аппаратного обеспечения.
- Количество созданных потоков может меняться, и поэтому исследовано, как выбрать наиболее подходящее число для экспериментов.
- Экспериментальные наборы данных различались по размеру, т. е. количество дискретных точек, требующих интерполяции, различалось по шкале.
- Используемый алгоритмом, может влиять на производительность, например, размер пикселя выходного изображения и количество найденных соседних точек.

Для решения вопросов, указанных выше, использовался следующий экспериментальный проект:

1. Эксперименты были разделены на две группы в зависимости от количества карт ускорителей. Первая использовала одну карту на каждой гетерогенной вычислительной платформе, а вторая использовала все карты ускорителей.

2. В каждой экспериментальной группе для каждой вычислительной платформы было определено подходящее число потоков. Наборы данных отличались по размеру, и параметры в алгоритме также были изменены в каждом эксперименте.

(1) платформа на базе графического процессора;

(2) платформа Intel Xeon Phi.

Конфигурация оборудования показана в таблице 3.1. Были различия в архитектуре оборудования и дизайне этих двух платформ. В этих

экспериментах важно было определить разницу в полученном ускорении и гетерогенность параллельного алгоритма на двумерных гетерогенных вычислительных платформах.

Таблица 3.1

Подробные сведения об экспериментальных площадках

Платформа	**Конфигурация оборудования**	
NVIDIA	GPU	GeForce® GTX 1070 Ti 8.0 GB, Overclocked 1607 MHz Chip clock, 1683 MHz
AMD	GPU	AMD Radeon RX 580 8.0 GB, Overclocked 1350 MHz
Intel Xeon Phi	CPU	Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70 GHz. Processor number: 24. Device global memory: 64,390 MB. Cache size: 30,720 KB

Чтобы оценить различия в производительности, последовательные и параллельные алгоритмы тестировались в экспериментах. Для параллельного алгоритма количество потоков, было установлено на интеграл, умноженный на количество ядер, предоставляемых платформой. Однако конечное количество потоков зависело от прошедшего времени. Таким образом, производительность аппаратного обеспечения может быть использована в максимальной степени. Согласно различным экспериментам с использованием различных наборов данных, количество потоков с самым коротким временем истекшего времени составляло 24 000 для платформы Intel Xeon Phi.

Чтобы проверить правильность параллельной программы и производительности ускорения, использовалось несколько наборов данных разных размеров для оценки последовательных и параллельных алгоритмов. Подробная информация, относящаяся к наборам данных, приведена в таблице 3.2.

Таблица 3.2

Подробная информация о наборах данных, используемых в экспериментах.

Размеры данных	**Атрибуты данных**	**Источники данных**
Малый набор данных	Число дискретных точек: 23447; сохранен в формате Shapefile; размер файла - 642 КБ	Параметр, разбиения сетки установлен на 6
Средний набор данных	Число дискретных точек: 300431; сохранен в формате Shapefile; Размер файла - 8,02 МБ.	Параметр, разбиения сетки установлен на 20
Большой набор данных	Число дискретных точек: 999894; сохранен в формате Shapefile; размер файла составляет 26,7 МБ.	Параметр, разбиения сетки установлен на 250

Таблица 3.2 показано, что три группы наборов данных были сгенерированы из исходного изображения с помощью операций последовательного анализа. Число точек в малых, средних и больших наборов данных могут быть отнесены к трем различным шкалам: 20 тысяч, 300 тысяч и один миллион соответственно.

Для оценки эффективности предлагаемого параллельного алгоритма использовался параметр ускорения, который является самым популярным индексом для этой цели. Ускорение S_p определяется следующим образом:

$$S_p = \frac{T_1}{T_p}, \tag{3.2}$$

где T_1 означает время выполнения последовательной программы, а T_p - время выполнения параллельного алгоритма с p процессорами. В этом исследовании параллельный универсальный алгоритм кригинга был под влиянием различных факторов, таких как размер набора данных и параметры. Чтобы определить различия в характеристиках в различных условиях, вычисляется несколько значений ускорения в этом эксперименте. Во-первых, согласно уравнению (3.2), вычисляется ускорение всего параллельного алгоритма и конкретное ускорение только для части UKIFM. Во-вторых, относительное потребление времени по текущей точке изменилось из-за различных факторов, поэтому также вычислялось и текущее теоретическое ускорение. Теоретический ускорение было рассчитано с использованием закона Амдаля [45].

$$Speedup \leq \frac{1}{(1-pctPar)+\frac{pctPar}{p}}. \tag{3.3}$$

Для неравенства *pctPar* указывает процентную долю последовательной программы, которая должна быть распараллелена, а число потоков деленное на число ядер - p. Когда p достигает наивысшего значения в теории, уравнение ограничения ускорения для программы выглядит следующим образом.

$$Speedup \leq \frac{1}{(1-pctPar)}. \tag{3.4}$$

Для простоты используется Speedup A, Speedup I и Speedup T в представлении вышеупомянутых значений ускорения соответственно.

В таблицах 3.3 – 3.5 показаны экспериментальные результаты, полученные с использованием одного ускорительного устройства с тремя

различными размерами данных. Экспериментальные результаты, полученные на платформе Intel Xeon Phi, приведены в таблицах 6 - 8. В этих таблицах *n* - это параметр для номера поиска окрестности, а *p* - исходное значение пространственного разрешения, где разрешение выходного изображения было установлено как меньше, равное или большее *p* в этом эксперименте. Данные, используемые для расчета SpeedupT, взяты из обнаружения текущих точек последовательного алгоритма.

Таблица 3.3

Ускорение получено с помощью одной карты на Intel Xeon Phi (небольшой набор данных).

Ускорение	Небольшой набор данных								
	n=6			n=12			n=18		
	>*p*	=*p*	<*p*	>*p*	=*p*	<*p*	>*p*	=*p*	<*p*
Ускорение I	11,43	11.30	11,42	10,96	10,87	10,76	11,77	11,67	11,66
Ускорение A	5,02	5,74	6,09	7,46	7,53	7,74	9,18	9,23	9,37
Ускорение T	10,64	10,66	11,15	21,26	22,89	22,89	37,33	38,24	40,15

Таблица 3.4

Ускорение, полученное с помощью одной карты на Intel Xeon Phi (средний набор данных)

Ускорение	Небольшой набор данных								
	n=6			n=12			n=18		
	>*p*	=*p*	<*p*	>*p*	=*p*	<*p*	>*p*	=*p*	<*p*
Ускорение I	11,55	11,59	11,56	11,00	11,03	11,00	11,77	11,79	11,72
Ускорение A	4,39	4,87	5,57	6,61	7,02	7,43	8,53	8,8	9,05
Ускорение T	6,84	7,67	8,91	15,52	16,8	19,22	28,04	30,15	33,56

Таблица 3.5

Ускорение получено с помощью одно карта на Intel Xeon Phi (большой набор данных)

Ускорение	Небольшой набор данных								
	n=6			n=12			n=18		
	>*p*	=*p*	<*p*	>*p*	=*p*	<*p*	>*p*	=*p*	<*p*
Ускорение I	11,50	11,58	11,62	10,91	10,92	10,96	11,78	11,66	11,72
Ускорение A	2,91	3,53	4,57	5,35	5,94	6,78	7,5	7,93	8,66
Ускорение T	4,21	5,11	6,95	9,95	11,79	15,27	18,54	22,85	27,99

В общем, параллельный алгоритм получил хорошие характеристики ускорения, и это могло в определенной степени сократить время обработки. В частности, когда рассматривался только часть интерполяции универсальный Кригинг, самая высокое ускорение доходило до 40 раз, но значение Speedup I уменьшилось, а количество точек поиска увеличилось.

Ускорение I и Ускорение T также уменьшалось по мере увеличения размера набора данных. Теоретически эти снижения тенденций были вызваны главным образом необходимостью большего количества регистров в каждом потоке, когда число потоков было исправлено. В этих условиях гранулярность параллелизма уменьшалась и требовалось более длительное время работы.

Используя несколько устройств, в обоих случаях ускорение почти удвоилось. Более того, коэффициент ускорения увеличивался, когда размер набора данных был больше.

Рисунок 3.9 сравнивает результаты ускорения, полученные с помощью одной карты и нескольких карт на платформе Intel Xeon Phi.

Аналогично, по сравнению с одним устройством, ускорение I и ускорение A увеличились в три раза, что было *n* единиц множественных устройств. Кроме того, коэффициент ускорения также увеличился.

Поэтому результаты этих экспериментов показывают, что ускорение изменилось линейным образом по сравнению с тем, что на одном устройстве, когда использовались несколько устройств, которые подтвердили, что стратегия балансировки нагрузки полезна и эффективна.

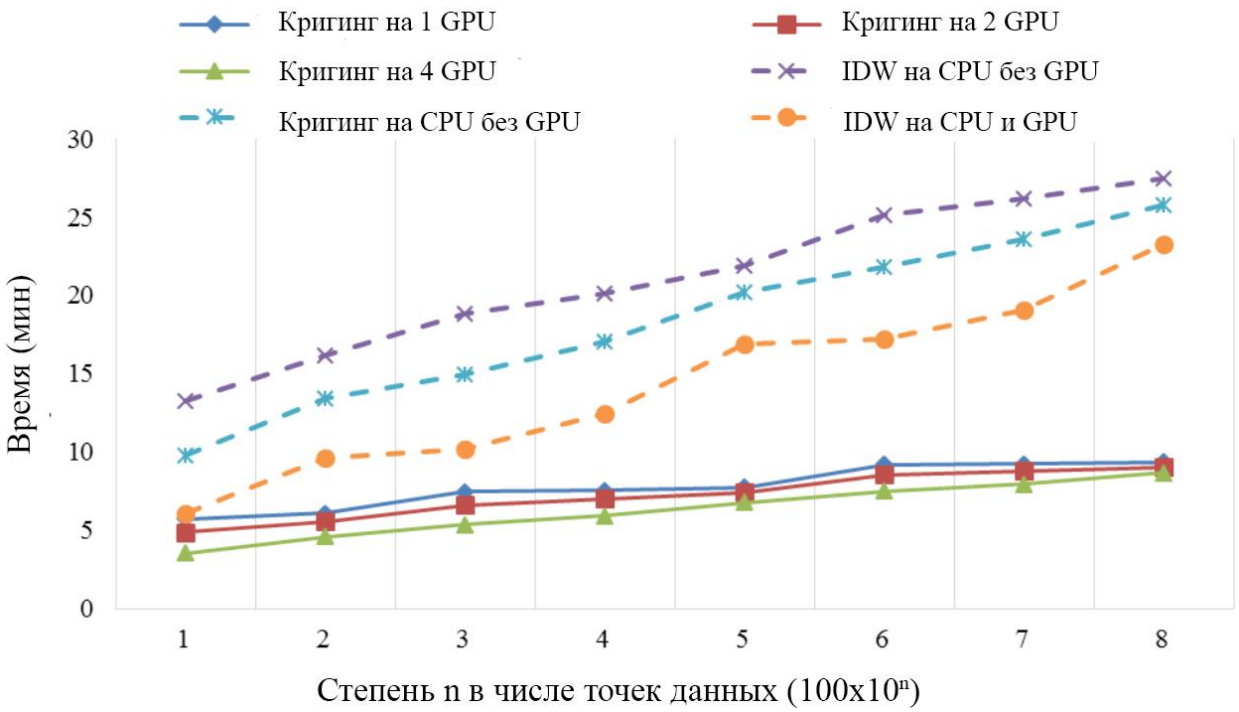


Рисунок 3.9. Сравнение выполнения алгоритма Кригинга и IDW с использованием GPU и CPU.

В этой главе представлен дизайн и реализация параллельного универсального алгоритма Кригинга, демонстрация его производительности и кроссплатформенных функций. Остальная часть этой главы организована следующим образом. В разделе 3.1 приводится краткое введение в универсальный алгоритм Кригинга и модель разработки OpenCL. В разделе основное внимание уделяется реализации алгоритма последовательного Кригинга, анализа точек и соответствующих методов распараллеливания. В разделе 3.2 описаны проектирование и реализация алгоритма параллельного

универсального Кригинга. В разделе 3.3 представлены экспериментальные результаты и анализ.

В этой главе основное внимание сфокусировано на развертывании параллельного Кригинга в алгоритме интерполяции с использованием технологии программирования OpenCL на гетерогенных платформах. В соответствии с экспериментальными результатами можно сделать следующие выводы.

1. Алгоритм универсальной интерполяции Кригинга, разработанный с помощью OpenCL, может работать на разных гетерогенных вычислительных платформах без каких-либо изменений, т.е. на основе графических процессоров или на основе MIC. Таким образом, предлагаемый метод обладает удовлетворительной кроссплатформенной способностью.

2. По сравнению с последовательным алгоритмом, который работает только на платформе ЦП, алгоритм параллельного универсального Кригинга, особенно его часть вычисления интерполяции, может достичь хорошего коэффициента ускорения на разных гетерогенных платформах.

3. Использование нескольких вычислительных устройств, т.е. большего количества графических/микропроцессорных карт для вычислений, привело к почти линейному увеличению коэффициента передачи и лучшей помехоустойчивости, чем одиночные вычислительные устройства. Тем не менее, гетерогенный вычислительный подход на основе OpenCL имеет свои слабые стороны, включая комплексное программирование, высокие требования к памяти, использование рекурсивных функций и относительно низкую эффективность по сравнению с CUDA.

Кроме того, есть возможность дальнейшей оптимизации, предлагаемой программы параллельного универсального Кригинга. Например, используя более масштабную шкалу интерполяции, шаг, используемый для поиска соседних точек в параллельном прогнозе, постепенно станет главной отправной точкой для работы. Тем не менее, эта отправная точка требует

больше внимания к дальнейшему повышению производительности параллельного алгоритма.

ЗАКЛЮЧЕНИЕ

В ходе исследования в выпускной квалификационной работе изучена возможность увеличения эффективности и быстродействия (производительности) систем управления запасами горнорудных предприятий при добыче и переработке путем интеграции новых технологий параллельного программирования в методы пространственной интерполяции.

В некоторых цифровых инженерных приложениях для обработки и анализа больших объемов данных требуются алгоритмы пространственной интерполяции. В этом исследовании изучено проектирование и реализация алгоритма параллельной универсальной пространственной интерполяции Кригинга с использованием модуля программирования OpenCL на гетерогенных вычислительных платформах для массивной обработки геопространственных данных. В этом исследовании основное внимание уделяется преобразованию точек в последовательных алгоритмах, т.е. универсальной функции интерполяции Кригинга, в соответствующую функцию ядра в OpenCL. Также используются методы распараллеливания и оптимизации в реализации для улучшения производительности кода. Наконец, на основе результатов экспериментов, выполненных на двух разных высокопроизводительных гетерогенных платформах, то есть с графической системой NVIDIA, AMD и системой Intel Xeon Phi, показано, что алгоритм параллельного универсального Кригинга может достичь наивысшего ускорения в 4 раза на единичных вычислительных устройствах и ускорения в 8 раз с несколькими устройствами.

Результаты данного исследования можно свести к следующим основным положениям:

1. Применение технологий параллельного программирования к существующим алгоритмам пространственной интерполяции положительно

влияет на быстродействие в процессах анализа и подсчета запасов полезных ископаемых;

2. Анализируя практические результаты можно понять, что полученное ускорение вычислений получилось далеко не идеальным. Оно не сможет тотально уменьшить время построения графических моделей, однако, рассматривая процесс интерполяции отдельно, ускорение в 8 раз можно считать отличным результатом;

3. Объединение вычислительных ресурсов центрального и графического процессоров в единые гетерогенные платформы экономически выгодное решение для предприятий. Использование по максимуму имеющихся аппаратных ресурсов без состояния простоя позволяет быстрее получать обработанные итоговые данные. Такая концепция развития предприятий в целом сможет существенно уменьшить этап вычислений и анализа в планировании горно-добывательных работ;

4. Этап компьютерного моделирования, а именно графическая обработки и визуализация, не затронуты данным исследованием. Основная задача была в произведении расчетов без графической обработки, но использование гетерогенных систем для визуализации сможет также ускорить этот процесс. Обработка изображений как правило ведется только на процессорах графических карт и зачастую число кадров в секунду проседает при развертывании крупномасштабных моделей. Использование центральных процессоров также повысит плавность графики и построения большей детализации.

Данные результаты исследования позволили сделать и обосновать следующие проектно-практические рекомендации:

1. Для использования описанных в выпускной квалификационной работе технологий и практического их применения следует подробно изучить все нюансы и специфику как самих методов пространственной интерполяции, так и технологий параллельного программирования OpenCL и CUDA;

2. Специалистам, занимающихся анализом и обработкой проб месторождений полезных ископаемых, которые выполняют данные работы находясь на аутсорсинге или в целях научных исследований, чтобы достичь результатов необходимо обладать достаточно мощными аппаратными ресурсами для сложных вычислений;

3. Исследования дают возможности масштабирования и кроссплатформенной разработки при условии, что будут соблюдены особенности интеграции в системы недропользования.

Перспективы исследования данной проблемы состоят в том, что современная горнодобывающая промышленность Российской Федерации не использует все имеющиеся новейшие технологии для экономической оптимизации процесса добычи. Мощнейшие ресурсы, которыми обладают ГОКи, как показывает практика, не дают ощутимого результата при обработке и анализе проб, потому как используются обособлено. Гетерогенные системы позволят выявить экономическую выгоду и сократить простой как оборудования, так и человеческих рабочих ресурсов. Научные исследования методов интерполяции с точки зрения параллельного программирования смогут обеспечить импульс к экономическому прорыву в области недропользовании. Оптимальное освоение огромных запасов полезных ископаемых, сосредоточенных на территории государства, глобально влияет на экономику Российской Федерации, которая добывает 18% сырья во всей мировой экономике. По оценкам экспертов страна обладает неосвоенными ресурсами на 2000 трлн. рублей.

Таким образом, поставленная во введении цель исследования в выпускной квалификационной работе достигнута, а исследовательские задачи выполнены. Их выполнение в итоге подтвердило и гипотезу о решении проблемы увеличения быстродействия путем внедрения технологий параллельного программирования, которая использовала бы вычислительные ресурсы центрального процессора совместно с графическим.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Li, J. Обзор пространственных методов интерполяции для ученых-экологов; Geoscience Australia: Канберра , Австралия, 2008.
2. Майерс, DE; Бегович К.Л. Буц, TR; Кейн, В. Е. Вариограммы для геохимических данных подземных вод. Математика Геол. 1982, 14, 629-644. [CrossRef]
3. Ozelkan, E .; Chen, G .; Устундаг, Б. Б. Пространственная оценка скорости ветра: новая интегративная модель с использованием коэффициента обратного расстояния и степенного закона. Int J Digit Earth. 2016. [CrossRef]
4. Bhattacharjee, S .; Mitra, P .; Ghosh, SK Spat ial интерполяция для прогнозирования отсутствующих атрибутов в ГИС с использованием семантического кригинга. IEEE Ttrans. Geosci. Удаленный датчик. 2014, 52, 4771-4780. [CrossRef]
5. Rossi, RE; Дунган, Дж. Beck, LR Кригинг в тени: геостатистическая интерполяция для дистанционного зондирования. Удаленный датчик. 1994, 49, 32-40. [CrossRef]
6. Zhang, X .; Jiang, H .; Zhou, G .; Xiao, Z .; Чжан, З. Геостатистическая интерполяция отсутствующих данных и уменьшение масштаба пространственного разрешения для концентрации зондовых концентраций метана в дистанционном зондировании. Int. J. Remote Sens., 2012, 33, 120-134. [CrossRef]
7. Carr, JR; Deng, ED; Применение дизъюнктивного кригинга для оценки земного землетрясения. Математика Геол. 1986, 18, 197-213. [CrossRef]
8. Dag, A .; Ozdemir, AC. Сравнительное исследование трехмерного поверхностного моделирования угольного месторождения с помощью пространственных интерполяционных подходов. Resour. Геол. 2013, 63, 394-403. [CrossRef]

9. Рид, П .; Минскер, Б .; Valocchi, AJ C ost-эффективный долгосрочный мониторинг подземных вод с использованием генетического алгоритма и глобальной массовой интерполяции. Water Resour Res. Водный ресурс. Местожительство 2000, 36, 3731-3741. [CrossRef]

10. Cressie, N .; Kang, EL Цифровое почвенное картографирование высокого разрешения: кригинг для очень больших наборов данных. В проксимальном почве; Springer: Dordrecht, Нидерланды, 2010; С. 49-63.

11. Kao, CS Автоматизированная интерполяция двумерных сейсмических сеток в объем трехмерных данных. G eophysics 1990, 55, 433-442. [CrossRef]

12. Sahlin, J .; Мостафави, Массачусетс; Forest, A .; Бабин М. Оценка трехмерных пространственных методов интерполяции для изучения морской пелагической среды. Мар. Геоды. 2014, 37, 238-266. [CrossRef]

13. Loubier, JC Оптимизация интерполяции температур с помощью ГИС: подход к пространственному анализу. В пространственной интерполяции для климатических данных; ISTE: Лондон, Великобритания, 2010; С. 97-107.

14. Ch ang, CL; Lo, SL; Ю., SL Оптимизация параметров методом обратного расстояния по генетическому алгоритму для оценки осадков. Environ. Монит. Оценка. 2006, 117, 145-155. [CrossRef] [PubMed]

15. Джеффри, SJ; Картер, Джо; Moodie, KB; Beswick, AR Использование пространственной интерполяции для построения всеобъемлющего архива австралийских климатических данных. Environ. Модель. Softw. 2001, 16, 309-330. [CrossRef]

16. Huang, F .; Liu, D .; Tan, X .; Wang, J .; Chen, Y .; Он, B. Исследования реализации параллельного алгоритма интерполяции IDW в параллельной ГИС на базе кластера Linux. Вычи. Geosci. 2011, 37, 426-434. [CrossRef]

17. Cressie, N .; Johannesson, G. Исправлен кригинг для очень больших пространственных наборов данных. JR Stati. Soc. 2008, 70, 209-226. [CrossRef]

18. Ingram, B .; Dan, C. Параллельная геостатистика для редких и плотных наборов данных. Quant. Геол. Geostat. 2010, 16, 371-381.

19. Армстронг, депутат; Марчиано, P. Массовая параллельная обработка пространственных статистик. Int. J. Geogr. Inf. Sci. 1995, 9, 169-189. [CrossRef]

20. Gajraj, A.; Joubert, W .; Джонс, Дж. Параллельная реализация Кригинга с трендом. Доступно в Интернете: http://www.osti.gov/scitech/servlets/purl/544698-DhOroO/webviewable/ (доступ к нему осуществляется 10 июня 2017 года).

21. Hawick, KA; Coddington, PD; James, HA Распределенные структуры и параллельные алгоритмы обработки крупномасштабных географических данных. Parall. Вычи. 2003, 29, 1297-1333. [CrossRef]

22. Гуань, X .; Wu, H. Использование мощностей многоядерных платформ для крупномасштабной обработки геопространственных данных: например, генерация DEM из массивных облачных точек LiDAR. Вычи. Geosci. 2010, 36, 1276-1282. [CrossRef]

23. Wang, S .; Армстронг, МП. Квадтритный подход к декомпозиции домена для пространственной интерполяции в средах грид-вычислений. Parall. Вычи. 2003, 29, 1481-1504. [CrossRef]

24. Керри, KE; Hawick, KA Kriging для высокопроизводительных компьютеров. В высокопроизводительных вычислениях и в сети; Springer: Берлин, Германия, 1998; С. 429-438.

25. Педели, Дж. А.; Morisette, JT; Smith, JA; Schnase, JL; Crosier, CS; Stohlgren, TJ Высокоэффективное геостатистическое моделирование биосферных ресурсов. Доступно в Интернете: h ttps: //esto.nasa.gov/conferences/ estc2003 / doc/ A4P4 (Schnase) .pdf (доступ к 10 июня 2017 г.).

26. Валентини, GL; Lassonde, W .; Khan, SU; Min-Allah, N .; Madani, SA; Li , J; Zhang, L .; Wang, L .; Ghani, N .; Kolodziej, J. Обзор методов энергоэффективности в кластерных вычислительных системах. Cluster Comput. 2011, 16, 1-13. [CrossRef]

27. Strzelczyk, J .; Porzycka, S .; Лесняк А. Анализ наземных деформаций на основе параллельных геостатистических вычислений данных PSInSAR. В Трудах Международной конференции по геоинформатике, Fairfax, VA, США, 12-14 августа 2009 года; С. 195-206.

28. Li, J; Jiang, Y .; Yang, C .; Hua ng, Q .; Рис, М. Визуализация данных об окружающей среде 3D / 4D с использованием многоядерных графических процессоров (GPU) и многоядерных центральных процессоров (ЦП). Вычи. Geosci. 2013, 59, 78-89. [CrossRef]

29. Liu, P .; Юань, Т .; Ян, Массачусетс; Wang, L .; Liu, D .; Yue, S .; Колодзей, Дж. Параллельная обработка массивных изображений дистанционного зондирования в архитектуре графического процессора. Вычи. Поставить в известность. 2014, 33, 197-217.

30. Xue, W .; Yang, C .; Fu, H .; Wang, X .; Xu, Y .; Liao, J .; Gan, L .; Lu, Y .; R anjan, R .; Wang, L. Ультра-масштабируемое ускорение CPU-MIC мезомасштабного атмосферного моделирования на Tianhe-2. IEEE Trans. Вычи. 2015, 64, 2382-2393. [CrossRef]

31. Chen, D .; Li, D; Xiong, M .; Bao, H .; Li, X. GPGPU-аннигилированный эмпирический режим разложения для анализа ЭЭГ во время анестезии. IEEE Trans. Inf. Technol. Biomed. 2010, 14, 1417-1427. [CrossRef] [PubMed]

32. Cheng, T. Ускорение универсального алгоритма интерполяции Кригинга с использованием графического процессора с поддержкой CUDA. Вычи. Geosci. 2013, 54 , 178-183. [CrossRef]

33. Heinecke, A.; Klemm, M .; Bungartz, HJ От GPGPU к многоядерному: NVIDIA Fermi и Intel много интегрированной основной архитектуры. Вычи. Sci. Eng. 2012, 14, 78-83. [CrossRef]

34. Wang, Y .; Deng, G .; Zhiliang, XU Оптимизация производительности для ускорения параллельных вычислений на основе MIC. J. Shenzhen Inst. Inf. Technol. 2013,11, 87-93.

35. Gaster, B .; Howes, L .; Kaeli, DR; Mistry, P .; Schaa, D. Неоднородные вычисления с OpenCL; Морган Кауфманн: Берлингтон, Массачусетс, США, 2011.

36. Бартье, премьер-министр; Келлер, СР Многомерная интерполяция для включения тематических данных поверхности с использованием инверсного взвешивания расстояний (IDW). Вычи. Geosci. 1996,22,795-799. [CrossRef]

37. Lindholm, E .; Nickolls, J .; Oberman, S .; Montrym, J. NVIDIA Tesla: Единая графическая и вычислительная архитектура. IEEE Micro 2008, 28, 39-55. [CrossRef]

38. Duran, A .; Klemm, M. В Intel® много интегрированной архитектуры ядра. В материалах Международной конференции 2012 года по высокопроизводительным вычислениям и симуляции (HPCS), Мадрид, Испания, 2-6 июля 2012 года; pp. 365-366.

39. Камень, JE; Gohara, D .; Shi, G. OpenCL: Параллельный стандарт программирования для гетерогенных вычислительных систем. Вычи. Sci. Eng. 2010, 12, 66-73. [CrossRef] [PubMed]

40. Чилес, JP; Delfiner, P. Геостатистика: M odeling Пространственная неопределенность; John Wiley & Sons, Inc .: Нью-Йорк, Нью-Йорк, США, 2012.

41. Горшич, ди-джей; Гентон, М. Г. Выбор модели вариограммы с помощью оценки непараметрических производных. Математика Геол. 2000 , 32, 249-270. [CrossRef]

42. Альтман Н.С. Введение в ядро и непараметрическую регрессию ближайших соседей. Am. Стат. 1992, 46, 175-185.

43. Кирк, БД; Hwu, программирование WMW Массивно параллельные процессоры: практический подход; Морган Кауфманн: Берлингтон, Массачусетс, США, 2012.

44. Tsuchiyama, R .; Nakamura, T .; Lizuka, T .; Asahara, A.; Miki, S .; Tagawa, S. Книга программирования OpenCL; Корпорация Fixstars: Токио, Япония, 2010 год.

ПРИЛОЖЕНИЯ

cmake_minimum_required.cpp

```
project(ParallelDTM)

set(CMAKE_INCLUDE_CURRENT_DIR ON)
set(CMAKE_CXX_FLAGS "-std=c++11 -fopenmp")
SET(CMAKE_CXX_LINK_FLAGS "-fopenmp")

list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/cmake")

set(CLUSTER ON)

if(${CLUSTER})
  set(OpenCL_LIBRARY $ENV{AMDAPPSDKROOT}/lib/x86_64/libOpenCL.so)
endif()

find_package(OpenCL REQUIRED)

#include_directories(${OpenCL_INCLUDE_DIRS})
include_directories($ENV{AMDAPPSDKROOT}/include)

set(CMAKE_INSTALL_PREFIX "${CMAKE_CURRENT_SOURCE_DIR}/install" CACHE PATH
STRING FORCE)

add_executable(
  ParallelDTM
  CommandLineParser.cpp
  ComputePlatform.cpp
  DistancesMatrixOperation.cpp
  KrigingOperation.cpp
  ReductionOperation.cpp
  FillBufferOperation.cpp
  LinearAlgebraOperation.cpp
  XYZFile.cpp
  Point.cpp
  Timer.cpp
  main.cpp
)
```

```
add_executable(
        Tests
        FillBufferOperation.cpp
        LinearAlgebraOperation.cpp
        ReductionOperation.cpp
        Point.cpp
        CommandLineParser.cpp
        ComputePlatform.cpp
        Timer.cpp
        Tests.cpp
)

target_link_libraries(ParallelDTM ${OpenCL_LIBRARIES})
target_link_libraries(Tests ${OpenCL_LIBRARIES})

set(
  KERNELS_FILELIST
  kernels/DistancesMatrix.cl
  kernels/Kriging.cl
  kernels/Reduction.cl
  kernels/Buffers.cl
  kernels/LinearAlgebra.cl
)

INSTALL(TARGETS Tests DESTINATION ${CMAKE_INSTALL_PREFIX})
INSTALL(TARGETS ParallelDTM DESTINATION ${CMAKE_INSTALL_PREFIX})
INSTALL(FILES ${KERNELS_FILELIST} DESTINATION "kernels")
```

main.cpp

```
//  ParallelDTM
//

#include <iostream>
#include <algorithm>
#include <numeric>
#include <thread>

#include "CommandLineParser.h"
#include "XYZFile.h"
```

```
#include "ComputePlatform.h"
#include "KrigingOperation.h"
#include "KrigingSerial.h"
#include "Timer.h"

using namespace std;

template<class TContainer>
bool BeginsWith(const TContainer& input, const TContainer& match)
{
	return input.size() >= match.size()
		&& equal(match.begin(), match.end(), input.begin());
}

int main(int ArgC, char* ArgV[])
{
	try
	{
#if defined(WIN32) || defined(UNIX)
	unsigned int ConcurentThreadsSupported = std::thread::hardware_concurrency();
		omp_set_num_threads(ConcurentThreadsSupported);
		Eigen::setNbThreads(ConcurentThreadsSupported);
		Eigen::initParallel();
#endif

		CommandLineParser CmdParser(ArgC, ArgV);

		if ((!CmdParser.OptionExists("--input") &&
			!CmdParser.OptionExists("--output")) || ArgC < 3)
		{
			cout << "USAGE: " << ArgV[0] << " --input [XYZ File] --output [Output File] {--lags-count [N] --grid-size [Size] --platform [ID] --num-devices [N] --profile --run-serial}" << endl;
			return EXIT_FAILURE;
		}

	int PlatformID = 0;
	if(CmdParser.OptionExists("--platform"))
	{
	   auto PlatformIDStr = CmdParser.GetOptionValue("--platform");
	   PlatformID = std::atoi(PlatformIDStr.data());
```

```
}

        int NumDevices = -1;
        if (CmdParser.OptionExists("--num-devices"))
        {
            auto NumDevicesStr = CmdParser.GetOptionValue("--num-devices");
            NumDevices = std::atoi(NumDevicesStr.data());
        }

int LagsCount = 10;
if(CmdParser.OptionExists("--lags-count"))
{
   auto LagsCountStr = CmdParser.GetOptionValue("--lags-count");
   LagsCount = std::atoi(LagsCountStr.data());
}

int GridSize = 30;
if(CmdParser.OptionExists("--grid-size"))
{
   auto GridSizeStr = CmdParser.GetOptionValue("--grid-size");
   GridSize = std::atoi(GridSizeStr.data());
}

bool bRunSerial = CmdParser.OptionExists("--run-serial");
bool bProfile = CmdParser.OptionExists("--profile");

auto InputFilepath = CmdParser.GetOptionValue("--input");
auto OutputFilepath = CmdParser.GetOptionValue("--output");

auto InputPoints = ReadXYZFile(InputFilepath);

int NumberOfPoints = static_cast<int>(InputPoints.size());
cout << "Number of Points: " << NumberOfPoints << endl;

if(bRunSerial)
{
   // Run Serial Code
   Serialkriging SerialKrigingOperation;

   Timer SerialKrigingTimer;
```

```
            SerialKrigingOperation.SerialKrigFit(InputPoints, NumberOfPoints, LagsCount);

            auto SerialKrigFitElapsed = SerialKrigingTimer.elapsedMilliseconds();
            SerialKrigingTimer = Timer();

            auto KrigGrid = SerialKrigingOperation.SerialKrigPred(InputPoints, GridSize);

            auto SerialKrigPredElapsed = SerialKrigingTimer.elapsedMilliseconds();

            if(bProfile)
            {
               cout << "Profiling Info:" << endl;
               cout << "\t" << "Serial Kriging Fit : " << SerialKrigFitElapsed << " ms" << endl;
               cout << "\t" << "Serial Kriging Pred: " << SerialKrigPredElapsed << " ms" << endl;
               cout << "\t" << "Total: " << SerialKrigFitElapsed + SerialKrigPredElapsed << " ms" << endl;
            }

            WriteXYZFile(OutputFilepath, KrigGrid);
         }
         else
         {
            // Run Parallel Code
            ComputePlatform TheComputePlatform(PlatformID, NumDevices);
            cout << TheComputePlatform << endl;

            TheComputePlatform.bProfile = bProfile;

            KrigingOperation KrigingOperation(TheComputePlatform);

            Timer KrigingTimer;

            KrigingOperation.KrigFit(InputPoints, NumberOfPoints, LagsCount);

            TheComputePlatform.RecordTime({ "TotalKriging", "KrigFit" },
KrigingTimer.elapsedMilliseconds());

            KrigingTimer = Timer();

            auto KrigGrid = KrigingOperation.KrigPred(InputPoints, GridSize);
```

```
            TheComputePlatform.RecordTime({ "TotalKriging", "KrigPred" },
KrigingTimer.elapsedMilliseconds());

            WriteXYZFile(OutputFilepath, KrigGrid);

            if (TheComputePlatform.bProfile)
            {
               long int TotalTime = 0;

               cout << "Profiling Info:" << endl;
               for (auto ProfilePair : TheComputePlatform.ProfilingMap)
               {
                  auto EventName = ProfilePair.first;
                  auto EventTime = ProfilePair.second;

                  cout << "\t" << EventName << ": " << EventTime << " ms" << endl;

                  if (!BeginsWith(EventName, string("Total")))
                  {
                     TotalTime += ProfilePair.second;
                  }
               }
               cout << endl;
               cout << "\tTotal: " << TotalTime << " ms" << endl;
            }
         }
            }
            catch (const cl::Error& Exception)
            {
                  cout << "[CL ERROR] " << Exception.what() << " " << Exception.err() << endl;
            }
            catch (const exception& Exception)
            {
                  cout << "[ERROR] " << Exception.what() << endl;
            }

      return EXIT_SUCCESS;
   }
```

Tests.cpp

```
#include "CommandLineParser.h"
#include "ComputePlatform.h"
#include "LinearAlgebraOperation.h"
#include "FillBufferOperation.h"

#include <iostream>
#include <vector>

#include <omp.h>

using namespace std;

int main(int Argc, char* ArgV[])
{
	try
	{
		CommandLineParser CmdParser{ Argc, ArgV };

		int PlatformID = 0;
		if (CmdParser.OptionExists("--platform"))
		{
			auto PlatformIDStr = CmdParser.GetOptionValue("--platform");
			PlatformID = std::atoi(PlatformIDStr.data());
		}

		int TestToPerform = 0;
		if (CmdParser.OptionExists("--test"))
		{
			auto TestIDStr = CmdParser.GetOptionValue("--test");
			TestToPerform = std::atoi(TestIDStr.data());
		}

		ComputePlatform TheComputePlatform(PlatformID);
		cout << TheComputePlatform << endl;

		if (TestToPerform == 0)
		{
			auto Queue = TheComputePlatform.GetNextCommandQueue();
```

```
                int N = 10;
                int BufferSize = N * sizeof(float);
                cl::Buffer aBuffer(TheComputePlatform.Context, CL_MEM_READ_WRITE,
BufferSize);

                const float One = 1.0f;
                cl::Event FillBufferEvent;
                Queue.enqueueFillBuffer(aBuffer, One, 0, BufferSize, nullptr,
&FillBufferEvent);
                FillBufferEvent.wait();

                vector<float> BufferContents(N);
                Queue.enqueueReadBuffer(aBuffer, CL_TRUE, 0, BufferSize,
BufferContents.data());

                for (auto i : BufferContents)
                {
                        cout << i << " ";
                }
                cout << endl;
        }

        if (TestToPerform == 1)
        {
                auto Queue = TheComputePlatform.GetNextCommandQueue();

                cl::Buffer aBuffer(TheComputePlatform.Context, CL_MEM_READ_WRITE,
sizeof(int));

                int Value;
                Queue.enqueueReadBuffer(aBuffer, CL_TRUE, 0, sizeof(int), &Value);

                cout << Value << endl;
        }

        if (TestToPerform == 2)
        {
                int NumDevices = static_cast<int>(TheComputePlatform.Devices.size());
                int NumThreads = 2;
```

```
                LinearAlgebraOperation LinAlg{ TheComputePlatform };
                FillBufferOperation FillBuffer{ TheComputePlatform };

                int N = 10000;
                const int GridSize = 30;
                int MatrixSize = N * N * sizeof(double);
                int VectorSize = N * sizeof(double);

                vector<double> ResultVector(GridSize * GridSize);

#               pragma omp parallel num_threads(NumThreads)
                {
                        int ThreadId = omp_get_thread_num();

#                       pragma omp critical
                        cout << "Thread ID: " << ThreadId << endl;

                        //auto Queue = TheComputePlatform.GetNextCommandQueue();
                        int DeviceID = ThreadId % NumDevices;
                        cl::CommandQueue Queue(TheComputePlatform.Context,
TheComputePlatform.Devices[DeviceID], CL_QUEUE_PROFILING_ENABLE);

                        cl::Buffer MatrixBuffer(TheComputePlatform.Context,
CL_MEM_READ_WRITE, MatrixSize);
                        cl::Buffer VectorBuffer(TheComputePlatform.Context,
CL_MEM_READ_WRITE, VectorSize);
                        cl::Buffer ResultBuffer(TheComputePlatform.Context,
CL_MEM_READ_WRITE, VectorSize);
                        cl::Buffer CacheBuffer(TheComputePlatform.Context,
CL_MEM_READ_WRITE, VectorSize);

                        FillBuffer.FillDoubleBuffer(Queue, MatrixBuffer, 1.0, N * N).wait();
                        FillBuffer.FillDoubleBuffer(Queue, VectorBuffer, 1.0, N).wait();

#                       pragma omp for
                        for (int i = 0; i < GridSize; ++i)
                        {
#                       pragma omp critical
                                cout << i << " " << flush;
```

```
                                        for (int j = 0; j < GridSize; ++j)
                                        {
                                                LinAlg.MatVecMul(Queue, MatrixBuffer,
VectorBuffer, ResultBuffer, N).wait();
                                                double DotResult = LinAlg.DotProduct(Queue,
VectorBuffer, ResultBuffer, N, CacheBuffer);

                                                ResultVector[i + j * GridSize] = DotResult;
                                        }
                                }
                        }
                }
                catch (cl::Error& err)
                {
                        cout << "CL ERROR: " << err.what() << " " << err.err() << endl;
                }
                catch (runtime_error& e)
                {
                        cout << "ERROR: " << e.what() << endl;
                }
                catch (...)
                {
                        cout << "Unknown Error" << endl;
                }

                return 0;
        }
```

```
//
//  ComputePlatform.cpp
//  ParallelDTM
//
//  Created by Thales Sabino on 8/25/16.
//  Copyright © 2016 Thales Sabino. All rights reserved.
//

#include "ComputePlatform.h"

#include <iostream>
#include <fstream>
#include <stdexcept>
#include <algorithm>
```

```
#ifdef _WIN32
# define STDCALL __stdcall
#else
# define STDCALL
#endif

using namespace std;

void STDCALL OpenCLContextNotify(const char * info, const void *, ::size_t, void*)
{
   cout << info << endl;
}

static vector<cl::Platform> GetPlatforms()
{
   vector<cl::Platform> TempPlatforms;
   cl::Platform::get(&TempPlatforms);
   return TempPlatforms;
}

double ComputePlatform::GetEventElapsedTime(cl::Event Event)
{
  auto StartTime = Event.getProfilingInfo<CL_PROFILING_COMMAND_START>();
  auto EndTime = Event.getProfilingInfo<CL_PROFILING_COMMAND_END>();
  auto ElapsedNanoSec = EndTime - StartTime;
  auto                                                ElapsedMilliSec                                                =
chrono::duration_cast<chrono::milliseconds>(chrono::nanoseconds(ElapsedNanoSec)).count();

  return ElapsedMilliSec;
}

int ComputePlatform::GetPlatformCount()
{
   return static_cast<int>(GetPlatforms().size());
}

ComputePlatform::ComputePlatform(int PlatformIndex, int NumDevices)
{
   auto PlatformList = GetPlatforms();

   if(PlatformIndex >= static_cast<int>(PlatformList.size()))
   {
      throw std::runtime_error("Invalid platform index");
   }

   Platform = PlatformList.at(PlatformIndex);
  Platform.getDevices(CL_DEVICE_TYPE_ALL, &Devices);

  if (Devices.empty())
  {
        throw std::runtime_error("No devices found for platform " + to_string(PlatformIndex));
  }

  if (NumDevices != -1)
  {
        while (Devices.size() > NumDevices)
        {
              Devices.pop_back();
        }
  }
```

```
    auto DeviceIsntValidPred = [](cl::Device aDevice)
    {
       return aDevice.getInfo<CL_DEVICE_DOUBLE_FP_CONFIG>() == 0;
    };
    Devices.erase(remove_if(Devices.begin(), Devices.end(), DeviceIsntValidPred), Devices.end());

    cl_context_properties ContextProperties[3] =
    {
       CL_CONTEXT_PLATFORM,
       (cl_context_properties)(Platform)(),
       0
    };

    Context = cl::Context(Devices, ContextProperties, OpenCLContextNotify);

   for (auto aDevice : Devices)
   {
            CommandQueues.emplace_back(Context, aDevice, CL_QUEUE_PROFILING_ENABLE);
   }
}

cl::Program ComputePlatform::CreateProgram(const std::string &SourceFilepath)
{
    ifstream KernelFile(SourceFilepath);

    if(!KernelFile.is_open())
    {
       throw std::runtime_error("Error opening " + SourceFilepath);
    }

    string KernelSource( istreambuf_iterator<char>(KernelFile), (istreambuf_iterator<char>()) );
    cl::Program::Sources ProgramSource(1, make_pair(KernelSource.data(), KernelSource.size()));
    cl::Program Program(Context, ProgramSource);

    try
    {
       Program.build(Devices);
    }
    catch(...)
    {
       for(auto aDevice : Devices)
       {
          cout << Program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(aDevice) << endl;
       }

            rethrow_exception(std::current_exception());
    }

    return Program;
}

cl::CommandQueue ComputePlatform::GetNextCommandQueue()
{
   std::unique_lock<std::mutex> Lock(CommandQueuesMutex);

    // Round Robin Scheduling
    rotate(CommandQueues.begin(), CommandQueues.begin() + 1, CommandQueues.end());

    return CommandQueues.back();
```

```
}

void ComputePlatform::PrintDeviceName(const std::string& Description, cl::CommandQueue Queue)
{
#if PRINT_DEVICE_NAMES
   auto Device = Queue.getInfo<CL_QUEUE_DEVICE>();
   auto DeviceName = Device.getInfo<CL_DEVICE_NAME>();

   cout << Description << ": " << DeviceName << endl;
#endif
}

void ComputePlatform::RecordEvent(const std::vector<std::string>& Tags, cl::Event Event)
{
  if (bProfile)
  {
        double ElapsedMilliSec = GetEventElapsedTime(Event);

        unique_lock<mutex> Lock(RecordEventMutex);
        for (auto Tag : Tags)
        {
              ProfilingMap[Tag] += static_cast<long int>(ElapsedMilliSec);
        }
  }
}

void ComputePlatform::RecordTime(const std::vector<std::string>& Tags, long int Time)
{
  if (bProfile)
  {
        unique_lock<mutex> Lock(RecordEventMutex);
        for (auto Tag : Tags)
        {
              ProfilingMap[Tag] += Time;
        }
  }
}

ostream& operator<< (ostream& out, const ComputePlatform& aComputingPlatform)
{
   auto Platform = aComputingPlatform.Platform;

   out << "Platform" << endl;
   out << "\tName   : " << Platform.getInfo<CL_PLATFORM_NAME>() << endl;
   out << "\tVendor : " << Platform.getInfo<CL_PLATFORM_VENDOR>() << endl;
   out << "\tProfile: " << Platform.getInfo<CL_PLATFORM_PROFILE>() << endl;
   out << "\tVersion: " << Platform.getInfo<CL_PLATFORM_VERSION>() << endl;

   out << endl;

  out << "Devices" << endl;
  for (auto aDevice : aComputingPlatform.Devices)
  {
        out << "\tName          : " << aDevice.getInfo<CL_DEVICE_NAME>() << endl;
        out << "\tVendor        : " << aDevice.getInfo<CL_DEVICE_VENDOR>() << endl;
        out << "\tGlobal Mem Size: " << aDevice.getInfo<CL_DEVICE_GLOBAL_MEM_SIZE>() /
(1024 * 1024) << " Mb" << endl;
        out << "\tLocal Mem Size : " << aDevice.getInfo<CL_DEVICE_LOCAL_MEM_SIZE>() / 1024
<< " Kb" << endl;
```

```
                vector<size_t> MaxWorkGroupSize = aDevice.getInfo<CL_DEVICE_MAX_WORK_ITEM_SIZES>();
                out << "\tMax Work Item Sizes: ";
                for (auto WorkGroupSize : MaxWorkGroupSize)
                {
                        out << WorkGroupSize << " ";
                }
                out << endl;

                out << "\tMax Work Group Size: " << aDevice.getInfo<CL_DEVICE_MAX_WORK_GROUP_SIZE>() << endl;
                out << "\tMax Compute Units: " << aDevice.getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>() << endl;
                out << "\tMax Work Item Dimensions: " << aDevice.getInfo<CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS>() << endl;
                out << "\tMem Base Addr Align: " << aDevice.getInfo<CL_DEVICE_MEM_BASE_ADDR_ALIGN>() << endl;
            out << endl;
          }
          out << endl;

           return out;
        }


        #include "DistancesMatrixOperation.h"

        #include <iostream>

        using namespace std;

        DistancesMatrixOperation::DistancesMatrixOperation(ComputePlatform& Platform) :
          ThePlatform(Platform)
        {
          DistancesMatrixProgram = ThePlatform.CreateProgram("kernels/DistancesMatrix.cl");
        }

        cl::Event DistancesMatrixOperation::ComputeMatrix(cl::Buffer InputBuffer, int NumberOfPoints, cl::Buffer OutputBuffer)
        {
          auto DistancesMatrixKernel = cl::make_kernel<cl::Buffer, int, cl::Buffer>(DistancesMatrixProgram, "DistancesMatrixKernel");

          auto Queue = ThePlatform.GetNextCommandQueue();

           DEBUG_OPERATION;

          cl::NDRange GlobalRange(NumberOfPoints);
          return DistancesMatrixKernel(cl::EnqueueArgs(Queue, GlobalRange), InputBuffer, NumberOfPoints, OutputBuffer);
        }



        #include "FillBufferOperation.h"

        #include <iostream>

        using namespace std;
```

```
FillBufferOperation::FillBufferOperation(ComputePlatform& Platform) :
   ThePlatform(Platform)
{
   FillBufferProgram = ThePlatform.CreateProgram("kernels/Buffers.cl");
}

cl::Event FillBufferOperation::FillDoubleBuffer(cl::CommandQueue Queue, cl::Buffer Buffer, double Value,
int Count)
{
   DEBUG_OPERATION;

   auto FillBufferKernel = cl::make_kernel<cl::Buffer, double>(FillBufferProgram, "FillDoubleBuffer");

   return FillBufferKernel(cl::EnqueueArgs(Queue, cl::NDRange(Count)), Buffer, Value);
}

cl::Event FillBufferOperation::FillFloatBuffer(cl::CommandQueue Queue, cl::Buffer Buffer, float Value, int
Count)
{
   DEBUG_OPERATION;

   auto FillBufferKernel = cl::make_kernel<cl::Buffer, float>(FillBufferProgram, "FillFloatBuffer");

   return FillBufferKernel(cl::EnqueueArgs(Queue, cl::NDRange(Count)), Buffer, Value);
}

cl::Event FillBufferOperation::FillIntBuffer(cl::CommandQueue Queue, cl::Buffer Buffer, int Value, int
Count)
{
   DEBUG_OPERATION;

   auto FillBufferKernel = cl::make_kernel<cl::Buffer, int>(FillBufferProgram, "FillIntBuffer");

   return FillBufferKernel(cl::EnqueueArgs(Queue, cl::NDRange(Count)), Buffer, Value);
}

cl::Event FillBufferOperation::FillDoubleBuffer(cl::Buffer Buffer, double Value, int Count)
{
   auto Queue = ThePlatform.GetNextCommandQueue();

   return FillDoubleBuffer(Queue, Buffer, Value, Count);
}

cl::Event FillBufferOperation::FillFloatBuffer(cl::Buffer Buffer, float Value, int Count)
{
   auto Queue = ThePlatform.GetNextCommandQueue();

   return FillFloatBuffer(Queue, Buffer, Value, Count);
}

cl::Event FillBufferOperation::FillIntBuffer(cl::Buffer Buffer, int Value, int Count)
{
   auto Queue = ThePlatform.GetNextCommandQueue();

   return FillIntBuffer(Queue, Buffer, Value, Count);
}


#include "KrigingCommon.h"
```

```
#include <numeric>
#include <cmath>

using namespace std;

vector<float> GetLagRanges(float Cutoff, int LagsCount)
{
   vector<float> LagRanges;

   for (int LagIndex = 1; LagIndex < LagsCount + 1; ++LagIndex)
   {
      float LagMin = (LagIndex - 1) * Cutoff / LagsCount;
      float LagMax = LagIndex * Cutoff / LagsCount;
      LagRanges.push_back(LagMin);
      LagRanges.push_back(LagMax);
   }

   return LagRanges;
}

pair<float, float> LinearModelFit(const vector<float>& X, const vector<float>& Y)
{
            float MeanX = accumulate(X.begin(), X.end(), 0.0f) / X.size();
   float MeanY = accumulate(Y.begin(), Y.end(), 0.0f) / Y.size();

   float Sum1 = 0.0f;
   float Sum2 = 0.0f;
   for (int i = 0; i < X.size(); ++i)
   {
      Sum1 += (X[i] - MeanX) * (Y[i] - MeanY);
      Sum2 += pow(X[i] - MeanX, 2);
   }

   float b = Sum1 / Sum2;
   float a = MeanY - b * MeanX;

   return{ a, b };
}

double SphericalModel(double h, double Nugget, double Range, double Sill)
{
   if (h >= Range)
   {
      return Sill;
   }

   return (Sill - Nugget) * (1.5 * (h / Range) - 0.5 * pow(h / Range, 3)) + Nugget;
}

#include "KrigingOperation.h"
#include "KrigingCommon.h"
#include "ReductionOperation.h"
#include "DistancesMatrixOperation.h"
#include "FillBufferOperation.h"
#include "LinearAlgebraOperation.h"
#include "Timer.h"

#include <iostream>

using namespace std;
```

```
KrigingOperation::KrigingOperation(ComputePlatform& Platform) :
    ThePlatform(Platform)
{
    KrigingProgram = ThePlatform.CreateProgram("kernels/Kriging.cl");
}

void KrigingOperation::KrigFit(const PointVector& InputPoints, int NumberOfPoints, int LagsCount)
{
  this->NumberOfPoints = NumberOfPoints;

  auto Queue = ThePlatform.GetNextCommandQueue();

  DEBUG_OPERATION;

  cl::Buffer PointsBuffer(ThePlatform.Context, CL_MEM_READ_ONLY, NumberOfPoints * sizeof(PointXYZ));
  Queue.enqueueWriteBuffer(PointsBuffer, CL_TRUE, 0, NumberOfPoints * sizeof(PointXYZ), InputPoints.data());

  ReductionOperation ReductionOperation{ ThePlatform };
  DistancesMatrixOperation DistancesMatrixOperation{ ThePlatform };
  FillBufferOperation FillBufferOperation{ ThePlatform };

  MinPoint = ReductionOperation.ReducePoints(PointsBuffer, NumberOfPoints, ReductionOp::Min);
  MaxPoint = ReductionOperation.ReducePoints(PointsBuffer, NumberOfPoints, ReductionOp::Max);

  cout << "MinPoint: " << MinPoint << endl;
  cout << "MaxPoint: " << MaxPoint << endl;

  const float Cutoff = Dist(MaxPoint.x, MaxPoint.y, MinPoint.x, MinPoint.y) / 3.0f;
  auto LagRanges = GetLagRanges(Cutoff, LagsCount);

  const int DistancesMatrixElementCount = NumberOfPoints * NumberOfPoints;
  const int DistancesMatrixBufferSize = DistancesMatrixElementCount * sizeof(float);
  cl::Buffer DistancesMatrixBuffer(ThePlatform.Context, CL_MEM_READ_WRITE, DistancesMatrixBufferSize);

  cout << "Computing Distances Matrix ... " << flush;
  auto ComputeDistMatrixEvent = DistancesMatrixOperation.ComputeMatrix(PointsBuffer, NumberOfPoints, DistancesMatrixBuffer);
  if (ThePlatform.bProfile)
  {
      ComputeDistMatrixEvent.wait();
      ThePlatform.RecordEvent({ "DistancesMatrix" }, ComputeDistMatrixEvent);
  }
  cout << "done" << endl;

  auto SemivariogramKernel = cl::make_kernel<
      cl::Buffer,
      cl::Buffer,
      int,
      float,
      float,
      cl::Buffer,
      cl::Buffer,
      cl::Buffer>
      (KrigingProgram, "SemivariogramKernel");

  cout << "Computing Semivariogram ... " << flush;
```

```
vector<float> EmpiricalSemivariogramX(LagsCount, std::numeric_limits<float>::infinity());
vector<float> EmpiricalSemivariogramY(LagsCount, std::numeric_limits<float>::infinity());

Timer SemivariogramTimer;

# pragma omp parallel num_threads(static_cast<int>(ThePlatform.Devices.size()))
{
        auto SemivarQueue = ThePlatform.GetNextCommandQueue();

        cl::Buffer LocalPointsBuffer;
        cl::Buffer LocalDistancesMatrixBuffer;

        if (SemivarQueue() != Queue())
        {
                LocalPointsBuffer = cl::Buffer(ThePlatform.Context, CL_MEM_READ_ONLY, NumberOfPoints * sizeof(PointXYZ));
                LocalDistancesMatrixBuffer = cl::Buffer(ThePlatform.Context, CL_MEM_READ_ONLY, DistancesMatrixBufferSize);

                SemivarQueue.enqueueCopyBuffer(PointsBuffer, LocalPointsBuffer, 0, 0, NumberOfPoints * sizeof(PointXYZ));
                SemivarQueue.enqueueCopyBuffer(DistancesMatrixBuffer, LocalDistancesMatrixBuffer, 0, 0, DistancesMatrixBufferSize);
        }
        else
        {
                LocalPointsBuffer = PointsBuffer;
                LocalDistancesMatrixBuffer = DistancesMatrixBuffer;
        }

        cl::Buffer ValidValuesCountBuffer(ThePlatform.Context, CL_MEM_WRITE_ONLY, sizeof(int));
        cl::Buffer DistancesValuesBuffer(ThePlatform.Context, CL_MEM_READ_WRITE, DistancesMatrixBufferSize);
        cl::Buffer SemivarValuesBuffer(ThePlatform.Context, CL_MEM_READ_WRITE, DistancesMatrixBufferSize);

#       pragma omp for
        for (int LagIndex = 0; LagIndex < LagsCount; ++LagIndex)
        {
                auto FillEvent1 = FillBufferOperation.FillFloatBuffer(SemivarQueue, DistancesValuesBuffer, 0.0f, DistancesMatrixElementCount);
                auto FillEvent2 = FillBufferOperation.FillFloatBuffer(SemivarQueue, SemivarValuesBuffer, 0.0f, DistancesMatrixElementCount);
                auto FillEvent3 = FillBufferOperation.FillIntBuffer(SemivarQueue, ValidValuesCountBuffer, 0, 1);

                vector<cl::Event> FillBufferEvents = { FillEvent1, FillEvent2, FillEvent3 };

                const float RangeMin = LagRanges[LagIndex * 2 + 0];
                const float RangeMax = LagRanges[LagIndex * 2 + 1];

                auto SemivarKernelEvent = SemivariogramKernel(
                        cl::EnqueueArgs(SemivarQueue, FillBufferEvents, cl::NDRange(NumberOfPoints)),
                        LocalPointsBuffer,
                        LocalDistancesMatrixBuffer,
                        NumberOfPoints,
                        RangeMin,
                        RangeMax,
```

```
                                        DistancesValuesBuffer,
                                        SemivarValuesBuffer,
                                        ValidValuesCountBuffer);

                                int ValidValuesCount;
                                SemivarQueue.enqueueReadBuffer(ValidValuesCountBuffer, CL_TRUE, 0, sizeof(int),
&ValidValuesCount);

                                if (ValidValuesCount > 0)
                                {
                                        float AvgDistance = ReductionOperation.Reduce(SemivarQueue,
DistancesValuesBuffer, NumberOfPoints * NumberOfPoints, ReductionOp::Sum);
                                        float AvgSemivar = ReductionOperation.Reduce(SemivarQueue,
SemivarValuesBuffer, NumberOfPoints * NumberOfPoints, ReductionOp::Sum);

                                        AvgDistance /= ValidValuesCount;
                                        AvgSemivar /= ValidValuesCount;

                                        EmpiricalSemivariogramX[LagIndex] = AvgDistance;
                                        EmpiricalSemivariogramY[LagIndex] = 0.5f * AvgSemivar;
                                }
                        }
                }

                 ThePlatform.RecordTime({ "Semivariogram" }, SemivariogramTimer.elapsedMilliseconds());

                cout << "done" << endl;

                // Remove Invalid Elements
                auto IsInfPred = [](float Element) { return isinf(Element); };
                EmpiricalSemivariogramX.erase(remove_if(EmpiricalSemivariogramX.begin(),
EmpiricalSemivariogramX.end(), IsInfPred), EmpiricalSemivariogramX.end());
                EmpiricalSemivariogramY.erase(remove_if(EmpiricalSemivariogramY.begin(),
EmpiricalSemivariogramY.end(), IsInfPred), EmpiricalSemivariogramY.end());

                cout << "Fitting Linear Model to Semivariogram ... " << flush;
                auto LinearModel = LinearModelFit(EmpiricalSemivariogramX, EmpiricalSemivariogramY);

                const float LinearModelA = LinearModel.first;
                const float LinearModelB = LinearModel.second;

                Nugget = LinearModelA;
                Range = *max_element(EmpiricalSemivariogramX.begin(), EmpiricalSemivariogramX.end());
                Sill = Nugget + LinearModelB * Range;
                cout << "done" << endl;

                cout << "Nugget: " << Nugget << endl;
                cout << "Range : " << Range << endl;
                cout << "Sill  : " << Sill << endl;

                cout << "Calculating Covariance Matrix ..." << flush;
                const int CovarianceMatrixBufferCount = (NumberOfPoints + 1) * (NumberOfPoints + 1);
                const int CovarianceMatrixBufferSize = CovarianceMatrixBufferCount * sizeof(float);
                cl::Buffer CovarianceMatrixBuffer(ThePlatform.Context, CL_MEM_READ_WRITE,
CovarianceMatrixBufferSize);

                // Cria a matriz de covari�ncia com preenchida com 1's e um �nico zero no �ltimo elemento
                auto CovMatrixFillBufferEvent = FillBufferOperation.FillFloatBuffer(CovarianceMatrixBuffer, 1.0f,
CovarianceMatrixBufferCount);
```

```
    auto CovMatrixKernel = cl::make_kernel<
            cl::Buffer,
            cl::Buffer,
            int,
            float,
            float,
            float>
            (KrigingProgram, "CovarianceMatrixKernel");

    auto CovMatrixKernelEvent = CovMatrixKernel(
            cl::EnqueueArgs(Queue, CovMatrixFillBufferEvent, cl::NDRange(NumberOfPoints)),
            DistancesMatrixBuffer,
            CovarianceMatrixBuffer,
            NumberOfPoints,
            Nugget,
            Range,
            Sill
    );

    Eigen::MatrixXf CovMatrix(NumberOfPoints + 1, NumberOfPoints + 1);
    Queue.enqueueReadBuffer(CovarianceMatrixBuffer, CL_TRUE, 0, CovarianceMatrixBufferSize,
CovMatrix.data());

    ThePlatform.RecordEvent({ "CovarianceMatrix" }, CovMatrixKernelEvent);

    CovMatrix(NumberOfPoints, NumberOfPoints) = 0.0f;
    cout << "done" << endl;

    Timer InvertingMatrixTimer;

    cout << "Inverting Covariance Matrix ..." << flush;
    InvCovMatrix = CovMatrix.cast<double>();
    InvCovMatrix = InvCovMatrix.inverse();
    cout << "done" << endl;

    ThePlatform.RecordTime({ "InverseMatrix" }, InvertingMatrixTimer.elapsedMilliseconds());
  }

  vector<PointXYZ> KrigingOperation::KrigPred(const PointVector& InputPoints, int GridSize)
  {
    cout << "Predicting ... " << flush;

     LinearAlgebraOperation LinAlgOperation{ ThePlatform };
     FillBufferOperation FillBufferOperation{ ThePlatform };

    vector<PointXYZ> Grid(GridSize * GridSize);
    float GridDeltaX = (MaxPoint.x - MinPoint.x) / GridSize;
    float GridDeltaY = (MaxPoint.y - MinPoint.y) / GridSize;

     vector<double> ZValues(NumberOfPoints + 1);
     for(int i = 0; i < NumberOfPoints; ++i)
     {
        ZValues[i] = InputPoints[i].z;
     }
     ZValues[NumberOfPoints] = 1.0;

    auto PredicionCovarianceKernel = cl::make_kernel<
            cl::Buffer,
            cl::Buffer,
            double,
```

```
                double,
                double,
                double,
                double>(KrigingProgram, "PredictionCovariance");

        const int CovMatrixRowsCount = NumberOfPoints + 1;
        const int PredBuffersSize = CovMatrixRowsCount * sizeof(double);
        const int InvCovMatrixSize = CovMatrixRowsCount * CovMatrixRowsCount * sizeof(double);

    # pragma omp parallel num_threads(static_cast<int>(ThePlatform.Devices.size()))
        {
                auto Queue = ThePlatform.GetNextCommandQueue();

                cl::Buffer PointsBuffer(ThePlatform.Context, CL_MEM_READ_ONLY, NumberOfPoints *
sizeof(PointXYZ));
                cl::Buffer InvCovMatrixBuffer(ThePlatform.Context, CL_MEM_READ_ONLY,
InvCovMatrixSize);
                cl::Buffer ZValuesBuffer(ThePlatform.Context, CL_MEM_READ_ONLY, PredBuffersSize);
                cl::Buffer InvAXRBuffer(ThePlatform.Context, CL_MEM_READ_WRITE, PredBuffersSize);
                cl::Buffer RBuffer(ThePlatform.Context, CL_MEM_READ_WRITE, PredBuffersSize);
                cl::Buffer Cache(ThePlatform.Context, CL_MEM_READ_WRITE, CovMatrixRowsCount *
sizeof(double));

                cl::Event WriteMatrixEvent;
                cl::Event WriteZBufferEvent;
                cl::Event WritePointsEvent;

                Queue.enqueueWriteBuffer(PointsBuffer, CL_FALSE, 0, NumberOfPoints * sizeof(PointXYZ),
InputPoints.data(), nullptr, &WritePointsEvent);
                Queue.enqueueWriteBuffer(InvCovMatrixBuffer, CL_FALSE, 0, InvCovMatrixSize,
InvCovMatrix.data(), nullptr, &WriteMatrixEvent);
                Queue.enqueueWriteBuffer(ZValuesBuffer, CL_FALSE, 0, PredBuffersSize, ZValues.data(),
nullptr, &WriteZBufferEvent);
                auto FillRBufferEvent = FillBufferOperation.FillDoubleBuffer(Queue, RBuffer, 1.0,
CovMatrixRowsCount);

                cl::WaitForEvents({ WriteMatrixEvent, WriteZBufferEvent, FillRBufferEvent, WritePointsEvent
});

        #       pragma omp for
                for (int i = 0; i < GridSize; ++i)
                {
        #               pragma omp critical
                        cout << i << " " << flush;

                        for (int j = 0; j < GridSize; ++j)
                        {
                                float GridX = MinPoint.x + i * GridDeltaX;
                                float GridY = MinPoint.y + j * GridDeltaY;

                                PredicionCovarianceKernel(cl::EnqueueArgs(Queue,
cl::NDRange(NumberOfPoints)),
                                        PointsBuffer,
                                        RBuffer,
                                        GridX,
                                        GridY,
                                        Nugget,
                                        Range,
                                        Sill);
```

```
                                    auto MatVecMulEvent = LinAlgOperation.MatVecMul(Queue, InvCovMatrixBuffer, RBuffer, InvAXRBuffer, CovMatrixRowsCount);
                                    double GridZ = LinAlgOperation.DotProduct(Queue, InvAXRBuffer, ZValuesBuffer, CovMatrixRowsCount, Cache);

                                    Grid[i + j * GridSize] = PointXYZ(GridX, GridY, GridZ);
                                }
                            }
                        }

                        cout << "done" << endl;

                        return Grid;
                    }


                    #include "KrigingSerial.h"
                    #include "KrigingCommon.h"
                    #include "Timer.h"

                    #include <iostream>
                    #include <cmath>
                    #include <numeric>

                    using namespace std;

                    void Serialkriging::SerialKrigFit(const PointVector &InputPoints, int NumberOfPoints, int LagsCount)
                    {
                       this->NumberOfPoints = NumberOfPoints;

                        auto MinMaxXPair = minmax_element(InputPoints.begin(), InputPoints.end(), [](const PointXYZ& Point1, const PointXYZ& Point2)
                        {
                            return Point1.x < Point2.x;
                        });
                        auto MinMaxYPair = minmax_element(InputPoints.begin(), InputPoints.end(), [](const PointXYZ& Point1, const PointXYZ& Point2)
                        {
                            return Point1.y < Point2.y;
                        });
                        auto MinMaxZPair = minmax_element(InputPoints.begin(), InputPoints.end(), [](const PointXYZ& Point1, const PointXYZ& Point2)
                        {
                            return Point1.z < Point2.z;
                        });

                         MinPoint.x = (*MinMaxXPair.first).x;
                         MinPoint.y = (*MinMaxYPair.first).y;
                         MinPoint.z = (*MinMaxZPair.first).z;

                         MaxPoint.x = (*MinMaxXPair.second).x;
                         MaxPoint.y = (*MinMaxYPair.second).y;
                         MaxPoint.z = (*MinMaxZPair.second).z;

                         cout << "MinPoint: " << MinPoint << endl;
                         cout << "MaxPoint: " << MaxPoint << endl;

                         const float Cutoff = Dist(MaxPoint.x, MaxPoint.y, MinPoint.x, MinPoint.y) / 3.0f;
                         auto LagRanges = GetLagRanges(Cutoff, LagsCount);
```

```
Eigen::MatrixXf DistancesMatrix(NumberOfPoints, NumberOfPoints);

for(int i = 0; i < NumberOfPoints; i++)
{
   const auto& PointI = InputPoints[i];
   for(int j = 0; j < NumberOfPoints; ++j)
   {
      const auto& PointJ = InputPoints[j];
      DistancesMatrix(i, j) = Dist(PointI.x, PointI.y, PointJ.x, PointJ.y);
   }
}

cout << "Computing Semivariogram ... " << flush;

vector<float> EmpiricalSemivariogramX;
vector<float> EmpiricalSemivariogramY;

for (int LagIndex = 0; LagIndex < LagsCount; ++LagIndex)
{
   const float RangeMin = LagRanges[LagIndex * 2 + 0];
   const float RangeMax = LagRanges[LagIndex * 2 + 1];

   vector<float> DistValues;
   vector<float> SemivarValues;

   for(int i = 0; i < NumberOfPoints; i++)
   {
      for(int j = 0; j < NumberOfPoints; ++j)
      {
         auto DistIJ = DistancesMatrix(i, j);

         if(RangeMin < DistIJ && DistIJ < RangeMax)
         {
            const auto& PointIValue = InputPoints[i].z;
            const auto& PointJValue = InputPoints[j].z;

            auto SemivarValue = pow(PointIValue - PointJValue, 2);

                                    DistValues.push_back(DistIJ);
                                    SemivarValues.push_back(SemivarValue);
                              }
      }
   }

   float AvgDistance = accumulate(DistValues.begin(), DistValues.end(), 0.0f);
   float AvgSemivar = accumulate(SemivarValues.begin(), SemivarValues.end(), 0.0f);

   AvgDistance /= DistValues.size();
   AvgSemivar /= SemivarValues.size();

        EmpiricalSemivariogramX.push_back(AvgDistance);
        EmpiricalSemivariogramY.push_back(0.5f * AvgSemivar);
}

cout << "done" << endl;

cout << "Fitting Linear Model to Semivariogram ... " << flush;
auto LinearModel = LinearModelFit(EmpiricalSemivariogramX, EmpiricalSemivariogramY);

const float LinearModelA = LinearModel.first;
```

```
    const float LinearModelB = LinearModel.second;

    Nugget = LinearModelA;
    Range = *max_element(EmpiricalSemivariogramX.begin(), EmpiricalSemivariogramX.end());
    Sill = Nugget + LinearModelB * Range;
    cout << "done" << endl;

    cout << "Nugget: " << Nugget << endl;
    cout << "Range : " << Range << endl;
    cout << "Sill  : " << Sill << endl;

    cout << "Calculating Covariance Matrix ..." << flush;

    Eigen::MatrixXf CovarianceMatrix(NumberOfPoints + 1, NumberOfPoints + 1);
    CovarianceMatrix.fill(1.0f);
    CovarianceMatrix(NumberOfPoints, NumberOfPoints) = 0.0f;

    for(int i = 0; i < NumberOfPoints; i++)
    {
       for(int j = 0; j < NumberOfPoints; ++j)
       {
          auto DistIJ = DistancesMatrix(i, j);
          CovarianceMatrix(i, j) = SphericalModel(DistIJ, Nugget, Range, Sill);
       }
    }

    cout << "done" << endl;

    cout << "Inverting Covariance Matrix ..." << flush;
    InvCovMatrix = CovarianceMatrix.cast<double>();
    InvCovMatrix = InvCovMatrix.inverse();
    cout << "done" << endl;
}

PointVector Serialkriging::SerialKrigPred(const PointVector &InputPoints, int GridSize)
{
    cout << "Predicting ... " << flush;

    PointVector Grid(GridSize * GridSize);
    float GridDeltaX = (MaxPoint.x - MinPoint.x) / GridSize;
    float GridDeltaY = (MaxPoint.y - MinPoint.y) / GridSize;

    Eigen::VectorXd ZValues(NumberOfPoints + 1);
    Eigen::VectorXd RValues(NumberOfPoints + 1);
    for(int i = 0; i < NumberOfPoints; ++i)
    {
       ZValues[i] = InputPoints[i].z;
    }
    ZValues[NumberOfPoints] = 1.0;
    RValues[NumberOfPoints] = 1.0;

    Eigen::VectorXd InvAXR(NumberOfPoints + 1);

    for (int i = 0; i < GridSize; ++i)
    {
              cout << i << " " << flush;

       for (int j = 0; j < GridSize; ++j)
       {
          float GridX = MinPoint.x + i * GridDeltaX;
```

```
            float GridY = MinPoint.y + j * GridDeltaY;

            for(int PIndex = 0; PIndex < NumberOfPoints; PIndex++)
            {
               const auto& Point = InputPoints[PIndex];
               auto UDist = Dist(GridX, GridY, Point.x, Point.y);
               RValues[PIndex] = SphericalModel(UDist, Nugget, Range, Sill);
            }

                        InvAXR = InvCovMatrix * RValues;
            double GridZ = InvAXR.dot(ZValues);

            Grid[i + j * GridSize] = PointXYZ(GridX, GridY, GridZ);
          }
        }

        cout << "done" << endl;

        return Grid;
      }


      #include "LinearAlgebraOperation.h"

      using namespace std;

      LinearAlgebraOperation::LinearAlgebraOperation(ComputePlatform& Platform) :
        ThePlatform(Platform),
        ReduceOperation(Platform)
      {
        LinearAlgebraProgram = ThePlatform.CreateProgram("kernels/LinearAlgebra.cl");
      }

      cl::Event LinearAlgebraOperation::MatVecMul(cl::CommandQueue Queue, cl::Buffer MatrixBuffer,
cl::Buffer VectorBuffer, cl::Buffer ResultBuffer, int Count)
      {
        auto Device = Queue.getInfo<CL_QUEUE_DEVICE>();
        auto DeviceType = Device.getInfo<CL_DEVICE_TYPE>();

        DEBUG_OPERATION;

        switch (DeviceType)
        {
        case CL_DEVICE_TYPE_CPU:
                return MatVecMulCPU(Queue, MatrixBuffer, VectorBuffer, ResultBuffer, Count);
        case CL_DEVICE_TYPE_GPU:
                return MatVecMulGPU(Queue, MatrixBuffer, VectorBuffer, ResultBuffer, Count);
        default:
                break;
        }

        return cl::Event();
      }

      double LinearAlgebraOperation::DotProduct(cl::CommandQueue Queue, cl::Buffer A, cl::Buffer B, int Count,
cl::Buffer CacheBuffer)
      {
        DEBUG_OPERATION;

        if (CacheBuffer.getInfo<CL_MEM_SIZE>() != Count * sizeof(double))
```

```
        {
                throw runtime_error("CacheBuffer must have at least " + to_string(Count * sizeof(double)) + "
bytes");
        }

        auto VecMulKernel = cl::make_kernel<cl::Buffer, cl::Buffer, cl::Buffer, int>(LinearAlgebraProgram,
"VecMul");

         auto VecMulEvent = VecMulKernel(cl::EnqueueArgs(Queue, cl::NDRange(Count)), A, B, CacheBuffer,
Count);

         return ReduceOperation.ReduceDouble(Queue, CacheBuffer, Count, ReductionOp::Sum);
      }

      cl::Event LinearAlgebraOperation::MatVecMulCPU(cl::CommandQueue Queue, cl::Buffer MatrixBuffer,
cl::Buffer VectorBuffer, cl::Buffer ResultBuffer, int Count)
      {
        auto MatVecMulKernel = cl::make_kernel<
                cl::Buffer,
                cl::Buffer,
                int,
                cl::Buffer
        >(LinearAlgebraProgram, "MatVecMulCPUKernel");

        return MatVecMulKernel(
                cl::EnqueueArgs(Queue, cl::NDRange(Count)),
                MatrixBuffer,
                VectorBuffer,
                Count,
                ResultBuffer
        );
      }

      cl::Event LinearAlgebraOperation::MatVecMulGPU(cl::CommandQueue Queue, cl::Buffer MatrixBuffer,
cl::Buffer VectorBuffer, cl::Buffer ResultBuffer, int Count)
      {
        auto MatVecMulKernel = cl::make_kernel<
                cl::Buffer,
                cl::Buffer,
                cl::Buffer,
                cl::LocalSpaceArg,
                int,
                int
        >(LinearAlgebraProgram, "MatVecMulGPUKernel");

        const int PThreads = 8;

        int WorkGroupCount = Count;
        while (WorkGroupCount % PThreads != 0)
        {
                WorkGroupCount++;
        }

        int WorkItemCount = 64;
        while (WorkGroupCount % WorkItemCount != 0)
        {
                WorkItemCount >>= 1;
        }

        return MatVecMulKernel(
```

```
            cl::EnqueueArgs(
                    Queue,
                    cl::NDRange(WorkGroupCount, PThreads),
                    cl::NDRange(WorkItemCount, PThreads)
            ),
            MatrixBuffer,
            VectorBuffer,
            ResultBuffer,
            cl::Local(WorkItemCount * PThreads * sizeof(double)),
            Count,
            Count
    );
}

//
//  Point.cpp
//  ParallelDTM
//

#include "Point.h"

using namespace std;

PointXYZ::PointXYZ(float _x, float _y, float _z) : x(_x), y(_y), z(_z)
{}

std::ostream& operator<< (std::ostream& out, const PointXYZ& p)
{
    out << p.x << " " << p.y << " " << p.z;
    return out;
}


#include "ReductionOperation.h"

#include <algorithm>
#include <limits>
#include <iostream>

template<class T>
T NeutralElement(ReductionOp Operation)
{
    switch (Operation)
    {
        case ReductionOp::Min:
            return std::numeric_limits<T>::max();
        case ReductionOp::Max:
            return -std::numeric_limits<T>::max();
        case ReductionOp::Sum:
            return static_cast<T>(0.0);
        default:
            break;
    }

    return static_cast<T>(0.0);
}

template<class T>
T Reduce(T Accumulator, T Element, ReductionOp Operation)
{
```

```
    switch (Operation)
    {
       case ReductionOp::Min:
          Accumulator = (Accumulator < Element) ? Accumulator : Element;
          break;
       case ReductionOp::Max:
          Accumulator = (Accumulator > Element) ? Accumulator : Element;
          break;
       case ReductionOp::Sum:
          Accumulator += Element;
          break;
       default:
          break;
    }

    return Accumulator;
}

template<class T>
T ReduceCPU(T* Elements, int Count, ReductionOp Operation)
{
    T Accumulator = NeutralElement<T>(Operation);
  for (int Index = 0; Index < Count; ++Index)
  {
            const T Element = Elements[Index];
       Accumulator = Reduce(Accumulator, Element, Operation);
  }

  return Accumulator;
}

static PointXYZ ReduceCPU(PointXYZ* Points, int Count, ReductionOp Operation)
{
    auto Neutral = NeutralElement<float>(Operation);

    PointXYZ Accumulator{ Neutral, Neutral, Neutral };

  for (int Index = 0; Index < Count; ++Index)
  {
            const PointXYZ& Point = Points[Index];

       Accumulator.x = Reduce(Accumulator.x, Point.x, Operation);
       Accumulator.y = Reduce(Accumulator.y, Point.y, Operation);
       Accumulator.z = Reduce(Accumulator.z, Point.z, Operation);
  }

  return Accumulator;
}

ReductionOperation::ReductionOperation(ComputePlatform& Platform) :
  ThePlatform(Platform)
{
  ReductionProgram = ThePlatform.CreateProgram("kernels/Reduction.cl");
}

float ReductionOperation::Reduce(cl::CommandQueue Queue, cl::Buffer InputBuffer, int Count,
ReductionOp Operator)
{
    auto Device = Queue.getInfo<CL_QUEUE_DEVICE>();
    auto DeviceType = Device.getInfo<CL_DEVICE_TYPE>();
```

```
    DEBUG_OPERATION;

    switch (DeviceType)
    {
      case CL_DEVICE_TYPE_CPU:
        return ReduceCPUKernel(Queue, InputBuffer, Count, Operator);
      case CL_DEVICE_TYPE_GPU:
        return ReduceGPUKernel(Queue, InputBuffer, Count, Operator);
      default:
        break;
    }

    return 0.0f;
  }

  float ReductionOperation::Reduce(cl::Buffer InputBuffer, int Count, ReductionOp Operator)
  {
    auto Queue = ThePlatform.GetNextCommandQueue();

    return Reduce(Queue, InputBuffer, Count, Operator);
  }

  PointXYZ ReductionOperation::ReducePoints(cl::CommandQueue Queue, cl::Buffer InputBuffer, int
NumberOfPoints, ReductionOp Operator)
  {
    auto Device = Queue.getInfo<CL_QUEUE_DEVICE>();
    auto DeviceType = Device.getInfo<CL_DEVICE_TYPE>();

    DEBUG_OPERATION;

    switch (DeviceType)
    {
      case CL_DEVICE_TYPE_CPU:
        return ReducePointsCPUKernel(Queue, InputBuffer, NumberOfPoints, Operator);
      case CL_DEVICE_TYPE_GPU:
        return ReducePointsGPUKernel(Queue, InputBuffer, NumberOfPoints, Operator);
      default:
        break;
    }

    return PointXYZ();
  }

  PointXYZ ReductionOperation::ReducePoints(cl::Buffer InputBuffer, int NumberOfPoints, ReductionOp
Operator)
  {
    auto Queue = ThePlatform.GetNextCommandQueue();

    return ReducePoints(Queue, InputBuffer, NumberOfPoints, Operator);
  }

  double ReductionOperation::ReduceDouble(cl::CommandQueue Queue, cl::Buffer InputBuffer, int Count,
ReductionOp Operator)
  {
    DEBUG_OPERATION;

    auto Device = Queue.getInfo<CL_QUEUE_DEVICE>();
    auto DeviceType = Device.getInfo<CL_DEVICE_TYPE>();
```

```
            DEBUG_OPERATION;

            switch (DeviceType)
            {
               case CL_DEVICE_TYPE_CPU:
                  return ReduceCPUDoubleKernel(Queue, InputBuffer, Count, Operator);
               case CL_DEVICE_TYPE_GPU:
                  return ReduceGPUDoubleKernel(Queue, InputBuffer, Count, Operator);
               default:
                  break;
            }

            return 0.0;
         }

         float ReductionOperation::ReduceCPUKernel(cl::CommandQueue Queue, cl::Buffer InputBuffer, int Count,
ReductionOp Operator)
         {
            auto ReductionKernel = cl::make_kernel<cl::Buffer, int, int, cl::Buffer, int>(ReductionProgram,
"ReduceKernel");

            auto Device = Queue.getInfo<CL_QUEUE_DEVICE>();
            auto NumberOfBlocks = Device.getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>();

            int BlockSize = Count / NumberOfBlocks;
            if (Count % NumberOfBlocks != 0)
            {
                     BlockSize++;
            }

            const int ResultBufferSize = sizeof(float) * NumberOfBlocks;

            cl::Buffer ResultBuffer(ThePlatform.Context, CL_MEM_WRITE_ONLY, ResultBufferSize);

            cl::NDRange GlobalRange(NumberOfBlocks);
            cl::NDRange LocalRange(1);

            auto Event = ReductionKernel(cl::EnqueueArgs(Queue, GlobalRange, LocalRange), InputBuffer,
BlockSize, Count, ResultBuffer, static_cast<int>(Operator));

            float* PartialResult = (float*)Queue.enqueueMapBuffer(ResultBuffer, CL_TRUE, CL_MAP_READ, 0,
ResultBufferSize);

            float Result = ReduceCPU(PartialResult, NumberOfBlocks, Operator);

            Queue.enqueueUnmapMemObject(ResultBuffer, PartialResult);

            ThePlatform.RecordEvent({ "TotalReduce", "ReduceCPU" }, Event);

            return Result;
         }

         float ReductionOperation::ReduceGPUKernel(cl::CommandQueue Queue, cl::Buffer InputBuffer, int Count,
ReductionOp Operator)
         {
            auto ReductionKernel = cl::make_kernel<cl::Buffer, cl::LocalSpaceArg, int, cl::Buffer,
int>(ReductionProgram, "TwoStageReduceKernel");

            auto Device = Queue.getInfo<CL_QUEUE_DEVICE>();
            const int NumberOfComputeUnits = Device.getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>();
```

```
        const int NumberOfBlocks = NumberOfComputeUnits * 4;
        const int MaxBlockSize = 256;
         const int BlockSize = std::max(NumberOfBlocks % MaxBlockSize, 1);
         const int NumberOfGroups = std::max(NumberOfBlocks / BlockSize, 1);

        const int ResultBufferSize = sizeof(PointXYZ) * NumberOfGroups;

        cl::Buffer ResultBuffer(ThePlatform.Context, CL_MEM_WRITE_ONLY, ResultBufferSize);

        const int LocalMemSize = BlockSize * sizeof(PointXYZ);

        cl::NDRange GlobalRange(NumberOfBlocks);
        cl::NDRange LocalRange(BlockSize);
        auto Event = ReductionKernel(cl::EnqueueArgs(Queue, GlobalRange, LocalRange), InputBuffer,
cl::Local(LocalMemSize), Count, ResultBuffer, static_cast<int>(Operator));

        float* PartialResult = (float*)Queue.enqueueMapBuffer(ResultBuffer, CL_TRUE, CL_MAP_READ, 0,
ResultBufferSize);

        float Result = ReduceCPU(PartialResult, NumberOfGroups, Operator);

         Queue.enqueueUnmapMemObject(ResultBuffer, PartialResult);

        ThePlatform.RecordEvent({ "Reduce", "ReduceGPU" }, Event);

         return Result;
      }

      PointXYZ ReductionOperation::ReducePointsCPUKernel(cl::CommandQueue Queue, cl::Buffer InputBuffer,
int NumberOfPoints, ReductionOp Operator)
      {
        auto ReductionKernel = cl::make_kernel<cl::Buffer, int, int, cl::Buffer, int>(ReductionProgram,
"ReducePointsKernel");

        auto Device = Queue.getInfo<CL_QUEUE_DEVICE>();
        auto NumberOfBlocks = Device.getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>();

        int BlockSize = NumberOfPoints / NumberOfBlocks;
        if (NumberOfPoints % NumberOfBlocks != 0)
        {
                BlockSize++;
        }

        const int ResultBufferSize = sizeof(PointXYZ) * NumberOfBlocks;

        cl::Buffer ResultBuffer(ThePlatform.Context, CL_MEM_WRITE_ONLY, ResultBufferSize);

        cl::NDRange GlobalRange(NumberOfBlocks);
        cl::NDRange LocalRange(1);
        auto Event = ReductionKernel(cl::EnqueueArgs(Queue, GlobalRange, LocalRange), InputBuffer,
BlockSize, NumberOfPoints, ResultBuffer, static_cast<int>(Operator));

        PointXYZ* PartialResult = (PointXYZ*)Queue.enqueueMapBuffer(ResultBuffer, CL_TRUE,
CL_MAP_READ, 0, ResultBufferSize);

        auto Result = ReduceCPU(PartialResult, NumberOfBlocks, Operator);

         Queue.enqueueUnmapMemObject(ResultBuffer, PartialResult);
```

```
            ThePlatform.RecordEvent({ "TotalReduce", "ReduceCPU" }, Event);

             return Result;
          }

         PointXYZ ReductionOperation::ReducePointsGPUKernel(cl::CommandQueue Queue, cl::Buffer InputBuffer,
int NumberOfPoints, ReductionOp Operator)
          {
            auto ReductionKernel = cl::make_kernel<cl::Buffer, cl::LocalSpaceArg, int, cl::Buffer,
int>(ReductionProgram, "TwoStageReducePointKernel");

            auto Device = Queue.getInfo<CL_QUEUE_DEVICE>();
            const int NumberOfComputeUnits = Device.getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>();

            const int NumberOfBlocks = NumberOfComputeUnits * 4;
            const int MaxBlockSize = 256;
            const int BlockSize = std::max(NumberOfBlocks % MaxBlockSize, 1);
             const int NumberOfGroups = std::max(NumberOfBlocks / BlockSize, 1);

            const int ResultBufferSize = sizeof(PointXYZ) * NumberOfGroups;

            cl::Buffer ResultBuffer(ThePlatform.Context, CL_MEM_WRITE_ONLY, ResultBufferSize);

            const int LocalMemSize = BlockSize * sizeof(PointXYZ);

            cl::NDRange GlobalRange(NumberOfBlocks);
            cl::NDRange LocalRange(BlockSize);
            auto Event = ReductionKernel(cl::EnqueueArgs(Queue, GlobalRange, LocalRange), InputBuffer,
cl::Local(LocalMemSize), NumberOfPoints, ResultBuffer, static_cast<int>(Operator));

            PointXYZ* PartialResult = (PointXYZ*)Queue.enqueueMapBuffer(ResultBuffer, CL_TRUE,
CL_MAP_READ, 0, ResultBufferSize);

            auto Result = ReduceCPU(PartialResult, NumberOfGroups, Operator);

             Queue.enqueueUnmapMemObject(ResultBuffer, PartialResult);

            ThePlatform.RecordEvent({ "TotalReduce", "ReduceGPU" }, Event);

             return Result;
          }

         double ReductionOperation::ReduceCPUDoubleKernel(cl::CommandQueue Queue, cl::Buffer InputBuffer,
int Count, ReductionOp Operator)
          {
            auto ReductionKernel = cl::make_kernel<cl::Buffer, int, int, cl::Buffer, int>(ReductionProgram,
"ReduceDoubleKernel");

             auto Device = Queue.getInfo<CL_QUEUE_DEVICE>();
             auto NumberOfBlocks = Device.getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>();

             int BlockSize = Count / NumberOfBlocks;
             if (Count % NumberOfBlocks != 0)
             {
                BlockSize++;
             }

             const int ResultBufferSize = sizeof(float) * NumberOfBlocks;

             cl::Buffer ResultBuffer(ThePlatform.Context, CL_MEM_WRITE_ONLY, ResultBufferSize);
```

```
            cl::NDRange GlobalRange(NumberOfBlocks);
            cl::NDRange LocalRange(1);
            auto Event = ReductionKernel(cl::EnqueueArgs(Queue, GlobalRange, LocalRange), InputBuffer, BlockSize, Count, ResultBuffer, static_cast<int>(Operator));

            double* PartialResult = (double*) Queue.enqueueMapBuffer(ResultBuffer, CL_TRUE, CL_MAP_READ, 0, ResultBufferSize);

            auto Result = ReduceCPU(PartialResult, NumberOfBlocks, Operator);

            Queue.enqueueUnmapMemObject(ResultBuffer, PartialResult);

           ThePlatform.RecordEvent({ "TotalReduce", "ReduceCPU" }, Event);

            return Result;
        }

        double ReductionOperation::ReduceGPUDoubleKernel(cl::CommandQueue Queue, cl::Buffer InputBuffer, int Count, ReductionOp Operator)
        {
            auto ReductionKernel = cl::make_kernel<cl::Buffer, cl::LocalSpaceArg, int, cl::Buffer, int>(ReductionProgram, "TwoStageReduceDoubleKernel");

            auto Device = Queue.getInfo<CL_QUEUE_DEVICE>();
            const int NumberOfComputeUnits = Device.getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>();

            const int NumberOfBlocks = NumberOfComputeUnits * 4;
            const int MaxBlockSize = 256;
            const int BlockSize = std::max(NumberOfBlocks % MaxBlockSize, 1);
            const int NumberOfGroups = std::max(NumberOfBlocks / BlockSize, 1);

            const int ResultBufferSize = sizeof(PointXYZ) * NumberOfGroups;

            cl::Buffer ResultBuffer(ThePlatform.Context, CL_MEM_WRITE_ONLY, ResultBufferSize);

            const int LocalMemSize = BlockSize * sizeof(PointXYZ);

            cl::NDRange GlobalRange(NumberOfBlocks);
            cl::NDRange LocalRange(BlockSize);
            auto Event = ReductionKernel(cl::EnqueueArgs(Queue, GlobalRange, LocalRange), InputBuffer, cl::Local(LocalMemSize), Count, ResultBuffer, static_cast<int>(Operator));

            double* PartialResult = (double*)Queue.enqueueMapBuffer(ResultBuffer, CL_TRUE, CL_MAP_READ, 0, ResultBufferSize);

            auto Result = ReduceCPU(PartialResult, NumberOfGroups, Operator);

            Queue.enqueueUnmapMemObject(ResultBuffer, PartialResult);

           ThePlatform.RecordEvent({ "TotalReduce", "ReduceGPU" }, Event);

            return Result;
        }


        #include "Timer.h"

        using namespace std::chrono;
```

```
Timer::Timer()
{
   start();
}

void Timer::start()
{
   m_StartTime = steady_clock::now();
   m_bRunning = true;
}

void Timer::stop()
{
   m_EndTime = steady_clock::now();
   m_bRunning = false;
}

double Timer::elapsedMilliseconds()
{
   return duration_cast<milliseconds>(_elapsed()).count();
}

double Timer::elapsedSeconds()
{
   return duration_cast<seconds>(_elapsed()).count();
}

steady_clock::duration Timer::_elapsed() const
{
   decltype(m_StartTime) endTime;

   if(m_bRunning)
   {
      endTime = steady_clock::now();
   }
   else
   {
      endTime = m_EndTime;
   }

   return endTime - m_StartTime;
}

/
//  XYZFile.cpp
//  ParallelDTM
//

#include "XYZFile.h"

#include <iostream>
#include <fstream>
#include <sstream>
#include <iterator>

using namespace std;

PointVector ReadXYZFile(const std::string& FilePath)
{
   PointVector Data;
```

```
    ifstream InputFile(FilePath, ios::in);

    cout << "Reading file ... " << flush;
    string FileContents(istreambuf_iterator<char>(InputFile), (istreambuf_iterator<char>()));
    stringstream FileContentsStream(FileContents);
    cout << "done. Parsing ... " << flush;

    string Line;
    while (getline(FileContentsStream, Line))
    {
        istringstream iss(Line);
        vector<string> Tokens{ istream_iterator<string>{iss}, istream_iterator<string>{} };

        if (Tokens.size() == 3 || Tokens.size() == 6)
        {
            Data.emplace_back(stof(Tokens[0]), stof(Tokens[1]), stof(Tokens[2]));
        }
        else if (Tokens.size() == 8)
        {
            Data.emplace_back(stof(Tokens[2]), stof(Tokens[3]), stof(Tokens[4]));
        }
    }
    cout << "done" << endl;

    return Data;
}

void WriteXYZFile(const std::string& Filepath, PointVector Points)
{
    cout << "Writing to " << Filepath << " ... " << flush;
    ofstream OutputFile(Filepath);
    for(auto Point : Points)
    {
        OutputFile << Point.x << " " << Point.y << " " << Point.z << endl;
    }
    cout << "done" << endl;
}
```

Выпускная квалификационная работа выполнена мной совершенно самостоятельно. Все использованные в работе материалы и концепции из опубликованной научной литературы и других источников имеют ссылки на них.

«___» _____ _____ г.

_____ _____

(подпись) *(Ф.И.О.)*